2
Osnovni Node.js obrasci

Razumevanje asinhrone prirode platforme Node.js uopšte nije trivijalan posao, naročito ako je neko navikao na jezik kao što je PHP, gde nije uobičajeno raditi s asinhronim kodom.

U sinhronom programiranju, navikli smo na koncept po kojem kôd zamišljamo kao niz uzastopnih koraka koje smo definisali da bismo rešili određeni problem. Svaka operacija je blokirajuća, što znači da nova operacija može započeti samo pošto se završi prethodna. Takav pristup čini kôd jednostavnim za razumevanje i otklanjanje grešaka iz njega.

Nasuprot tome, u asinhronom programiranju, neke operacije, kao što je učitavanje podataka iz datoteke ili izvršavanje zahteva u mreži, mogu se izvršavati kao pozadinske operacije. Kada se pokrene asinhrona operacija, sledeća se pokreće odmah posle nje, uprkos tome što prethodna još nije izvršena. Operacije koje su u toku izvršavanja u pozadini mogu se završiti u svakom trenutku, a cela aplikacija treba da bude programirana tako da ispravno reaguje na završetak jednog od asinhronih poziva.

Mada taj neblokirajući pristup gotovo uvek može da garantuje bolje performanse u poređenju s blokirajućim scenariom, način na koji on deluje može biti teško shvatljiv, što može postati zaista veliki problem u slučaju naprednijih aplikacija u kojima je potrebno složeno upravljanje načinom izvršavanja koda.

Node.js nudi niz alatki i modela za optimalan rad s asinhronim kodom. Važno je da savladate način na koji se one koriste da biste stekli poverenje u njih i pisali aplikacije koje osim što pružaju dobre performanse, lako su razumljive i lako se otklanjaju greške iz njih.

U ovom poglavlju, razmotrićemo dva od najvažnijih asinhronih modela: povratnu funkciju i odašiljač događaja.

Model povratne funkcije

Povratne funkcije (engl. *callbacks*) predstavljaju materijalizaciju obrađivača u reaktorskom modelu, koji smo uveli u prethodnom poglavlju. One su jedan od elemenata koji obezbeđuju platformi Node.js njen karakterističan stil programiranja. Povratne funkcije su funkcije koje se pozivaju za dalju obradu rezultata neke operacije, a to je tačno ono što nam treba kada imamo asinhrone operacije. One zamenjuju upotrebu naredbe `return` koja se uvek izvršava sinhrono. JavaScript je

odličan jezik za upotrebu povratnih funkcija, zato što kao što smo već videli, funkcije su pre svega objekti tipa klasa i zato se lako mogu dodeljivati promenljivama, prosleđivati kao argumenti, vraćati nakon izvršavanja druge funkcije ili čuvati u obliku struktura podataka. Drugi savršen konstrukt za implementiranje povratnih funkcija su **ograde** (engl. *closures*). Ograde nam omogućavaju da referenciramo okruženje u kojem je funkcija nastala; tako uvek možemo sačuvati kontekst u kojem je bila zahtevana asinhrona operacija, bez obzira na to kada ili gde je povratna funkcija pozvana.

Ako vam je potrebno da osvežite svoje znanje o ogradama, možete pročitati sledeći tekst na Mozilla Developer Networku `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures`.

U ovom odeljku, analiziraćemo konkretan stil programiranja koji se zasniva na povratnim funkcijama umesto na naredbama *x*return.

Stil prosleđivanja narednog koraka (CPS)

U JavaScriptu, povratna funkcija je funkcija koja se prosleđuje kao argument drugoj funkciji i poziva kada se dobije rezultat operacije. U funkcionalnom programiranju, taj način prosleđivanja rezultata zove se **CPS** (**stil prosleđivanja narednog koraka –** engl. *continuation-passing style*). To je opšti koncept, koji nije uvek vezan za asinhrone operacije. U stvari, on samo pokazuje da se rezultat prosleđuje na naredni korak obrade tako što se prosleđuje drugoj funkciji (povratnoj), umesto da se direktno vrati pozivaocu.

Sinhroni CPS

Da bismo razjasnili koncept, razmotrićemo jednostavnu sinhronu funkciju:

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

Ovde nema ničeg posebnog; rezultat se vraća pozivaocu pomoću naredbe `return`; to se takođe zove **direktni stil** i predstavlja najuobičajeniji način vraćanja rezultata u sinhronom programiranju. Ekvivalentni CPS oblik prethodne funkcije (s povratnom funkcijom) izgledao bi ovako:

```
function add(a, b, callback) {
  callback(a + b);
}
```

Funkcija `add()` je sada sinhrona CPS funkcija, što znači da će vratiti neku vrednost tek kada se završi izvršavanje njene povratne funkcije. Naredni kôd to ilustruje:

```
console.log('pre');
add(1, 2, result => console.log('Rezultat: ' + result));
console.log('posle');
```

Pošto je funkcija `add()` sinhrona, prethodni kod će ispisati sledeći trivijalan rezultat:

```
pre
Rezultat: 3
posle
```

Asinhroni CPS

A sada, razmotrimo slučaj kada je funkcija `add()` asinhrona, kao u sledećoj verziji:

```
function additionAsync(a, b, callback) {
  setTimeout(() => callback(a + b), 100);
}
```

U prethodnom kodu, upotrebili smo funkciju `setTimeout()` da bismo simulirali asinhrono pozivanje povratne funkcije. Ako sada pozovemo funkciju `additionAsync`, videćemo da se redosled operacija promenio:

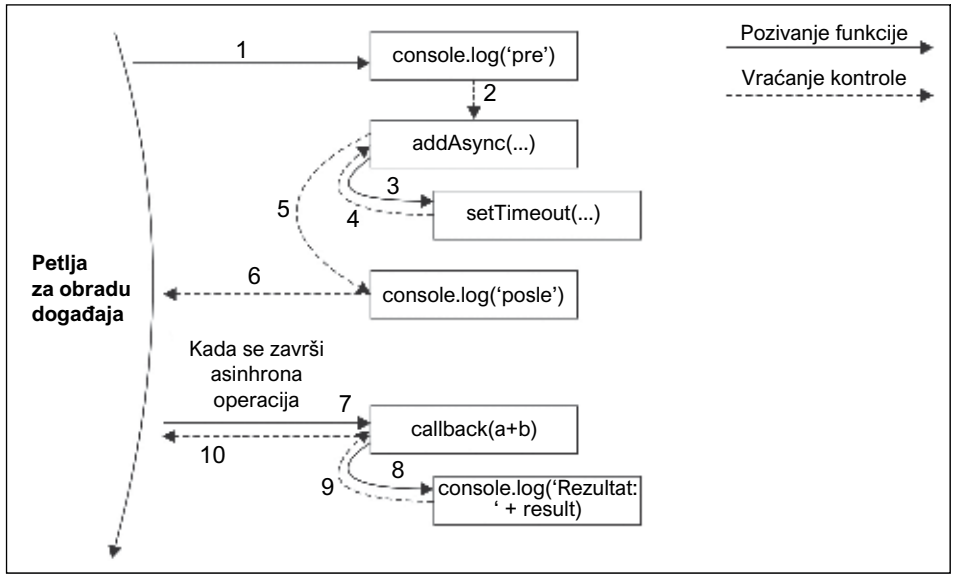
```
console.log('pre');
additionAsync(1, 2, result => console.log('Rezultat: ' + result));
console.log('posle');
```

Rezultat prethodnog koda je sledeći:

```
pre
posle
Rezultat: 3
```

Pošto funkcija `setTimeout()` pokreće asinhronu operaciju, ona neće čekati da se izvrši povratna funkcija, nego će odmah vratiti kontrolu svom pozivaocu, funkciji `additionAsync()`, a zatim i njenom pozivaocu. Ta osobina u Node.js je od ključne važnosti, budući da se tako vraća kontrola petlji za obradu događaja odmah nakon slanja asinhronog zahteva, što omogućava obradu novog događaja iz reda čekanja događaja na obradu.

Naredna slika prikazuje kako to radi:



Kada se završi asinhrona operacija, izvršavane aplikacije se nastavlja pozivanjem povratne funkcije prosleđene asinhronoj funkciji koja je izazvala prekid. Pošto će se izvršavanje pokrenuti iz **Petlje za obradu događaja**, imaće svež stek. Tu se JavaScript pokazuje kao zaista koristan. Zahvaljujući ogradama, održavanje konteksta pozivaoca asinhrone funkcije zaista je trivijalno, čak i kada se povartna funkcija poziva u različitom trenutku i sa različitog mesta.

Sinhrona funkcija blokira izvršavanje aplikacije dok ne završi svoje operacije. Asinhrona funkcija odmah vraća kontrolu aplikaciji, a njen rezultat se prosleđuje obrađivaču (što je u našem slučaju povratna funkcija) u nekom od kasnijih ciklusa petlje za obradu događaja.

Povratne funkcije u stilu bez prosleđivanja narednog koraka

Postoji više slučajeva u kojima nas prisustvo argumenta koji je povratna funkcija može navesti na pomisao da je takva funkcija asinhrona ili da je napisana u stilu prosleđivanja narednog koraka, ali to nije uvek tačno. Razmotrimo, na primer, metodu `map()` objekta `Array`:

```
const result = [1, 5, 7].map(element => element - 1);
console.log(result); // [0, 4, 6]
```

Jasno je da ovde povratna funkcija služi samo za iteraciju kroz elemente niza, a ne za prosleđivanje rezultata operacije. U stvari, metoda `map()`vraća rezultat sinhrono direktnim stilom. Svrha povratne funkcije je uglavnom jasno navedena u dokumentaciji API-ja.

Sinhrona ili asinhrona?

Videli smo kako se redosled izvršavanja naredaba radikalno menja u zavisnosti od prirode funkcije - sinhrona ili asinhrona. To veoma značajno utiče na tok izvršavanja cele aplikacije, i po pitanju ispravnosti, i po pitanju efikasnosti. U narednim odeljcima analiziramo ta dva aspekta i zamke koje oni uvode. U opštem slučaju, treba izbegavati uvođenje nedoslednosti i nedoumica u vezi s prirodom konkretnog API-ja, budući da to može dovesti do pojave skupa problema koji mogu biti vrlo teški za otkrivanje i simuliranje. Kao predmet naše analize, uzećemo primer jedne nedosledno asinhrone funkcije.

Jedna nepredvidljiva funkcija

Jedna od najopasnijih situacija jeste API koji se pod jednim uslovima ponaša sinhrono, a pod drigim uslovima asinhrono. Uzmimo kao primer sledeći kôd:

```
const fs = require('fs');
const cache = {};
function inconsistentRead(filename, callback) {
  if(cache[filename]) {
    //poziva se sinhrono
    callback(cache[filename]);
  } else {
    //asinhrona funkcija
    fs.readFile(filename, 'utf8', (err, data) => {
```

```
      cache[filename] = data;
      callback(data);
    });
  }
}
```

Prethodna funkcija smešta u promenljivu `cache` rezultate operacija učitavanja datoteka. Imajte u vidu da pošto je ovo samo primer, nema obrade grešaka, a ni logika za upravljanje kešom za podatke nije najbolja. Ali izvan toga, prethodna funkcija je opasna jer se ponaša asinhrono ako keš nije već popunjen - a on to nije sve dok funkcija `fs.readFile()` ne vrati svoje rezultate - ali će funkcija biti i sinhrona za sve naknadne zahteve za datotekom čiji se sadržaj već nalazi u kešu – što će odmah prouzrokovati izvršavanje povratne funkcije.

Puštanje Zalga s lanca

A sada, razmotrimo kako upotreba neke nepredvidljive funkcije, kao što je ona koju smo definisali u prethodnom odeljku, lako može srušiti celu aplikaciju. Pogledajte sledeći kôd:

```
function createFileReader(filename) {
  const listeners = [];
  inconsistentRead(filename, value => {
    listeners.forEach(listener => listener(value));
  });

  return {
    onDataReady: listener => listeners.push(listener)
  };
}
```

Kada se pozove prethodna funkcija, ona pravi nov objekat koji se ponaša kao obaveštavač, što nam omogućava da za istu operaciju učitavanja datoteke zadamo više osluškivača. Svi ti osluškivači biće pokrenuti istovremeno, u trenutku kada se operacija učitavanja završi i bude podataka na raspolaganju. Prethodna funkcija koristi našu funkciju `inconsistentRead()` da bi implementirala tu funkcionalnost. Pokušajmo sada da upotrebimo funkciju `createFileReader()`:

```
const reader1 = createFileReader('data.txt');
reader1.onDataReady(data => {
  console.log('Podaci iz orvog učitavanja: ' + data);

  // ...nešto kasnije ponovo pokušavamo učitavanje
  // novih podataka iz iste datoteke
  const reader2 = createFileReader('data.txt');
  reader2.onDataReady( data => {
    console.log('Podaci iz drugog učitavanja: ' + data);
  });
});
```

Prethodni kôd će ispisati sledeći rezultat:

```
Podaci iz prvog učitavanja: neki podaci
```

Kao što vidite, povratna funkcija se kod drugog učitavanja ne poziva. Pogledajmo zbog čega:

- Tokom formiranja objekta `reader1`, naša funkcija `inconsistentRead()` ponaša se asinhrono, zato što u kešu nema spremnog rezultata. Zbog toga, na raspolaganju imamo sve vreme ovog sveta da registrujemo osluškivač, koji će se izvršiti kasnije, u nekom drugom ciklusu petlje za obradu događaja, kada se završi operacija učitavanja podataka iz datoteke.
- Nakon toga, formiramo objekat `reader2` u jednom od ciklusa petlje za obradu događaja. Tada u kešu već imamo podatke iz zahtevane datoteke. U tom slučaju, interni poziv funkcije `inconsistentRead()` biće sinhron, što znači da će njena povratna funkcija biti izvršena odmah, što dalje znači da će i svi osluškivači objekta `reader2` biti izvršeni sinhrono. Međutim, pošto te osluškivače registrujemo tek nakon formiranja objekta `reader2`, oni neće nikada biti pozvani.

Ponašanje povratnih funkcija u našoj funkciji `inconsistentRead()` je zaista nepredvidljivo, budući da zavisi od više činilaca, kao što je učestalost pozivanja funkcija, datoteke koju joj prosledimo kao argument i količine vremena koje je potrebno za učitavanje sadržaja iz te datoteke.

Greška koju smo upravo videli može biti izuzetno komplikovana za otkrivanje i simuliranje u stvarnoj aplikaciji. Zamislite upotrebu slične funkcije na veb serveru, gde može biti više istovremenih zahteva; zamislite da gledate kako se neki od njih „zamrzavaju“, bez nekog vidljivog razloga i bez da bude zabeležena bilo kakva greška. To svakako spada u kategoriju *žešćih defekata*.

Isaac Z. Schlueter, autor alatke npm i nekadašnji vođa projekta Node.js, u jednom od svojih blogova uporedio je upotrebu ove vrste nepredvidljivih funkcija s *puštanjem Zalga s lanca*.

Zalgo je internet legenda o zlom čudovištu koje prouzrokuje ludilo, smrt i propast sveta. Ako niste čuli za Zalga, pozivamo vas da otkrijete šta je on.

Izvorni prilog Isaaca Z. Schluetera možete naći na `http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony`.

Upotreba sinhronih API-ja

Lekcija koju treba naučiti iz primera puštanja Zalga s lanca jeste to da je od ključne važnosti da API jasno definiše svoju prirodu: da li je sinhron ili asinhron.

Prikladna ispravka za našu funkciju `inconsistentRead()` bila bi da je načinimo potpuno sinhronom. To je moguće zato što za većinu osnovnih U/I operacija Node.js stavlja na raspolaganje skup sinhronih API-ja direktnog stila. Na primer, umesto njenog asinhronog ekvivalenta, možemo upotrebiti funkciju `fs.readFileSync()`. Kôd bi onda izgledao ovako:

```
const fs = require('fs');
const cache = {};
function consistentReadSync(filename) {
  if(cache[filename]) {
    return cache[filename];
  } else {
```

```
    cache[filename] = fs.readFileSync(filename, 'utf8');
    return cache[filename];
  }
}
```

Vidljivo je i to da je cela funkcija prepravljena u direktni stil (bez povratnih funkcija). Nema razloga da funkcija bude u stilu prosleđivanja narednog koraka (CPS) ako je sinhrona. U stvari, možemo zadati pravilo da je uvek bolje da se sinhroni API implementira u direktnom stilu; to će isključiti svaku zabunu u vezi s prirodom API-ja, a biće i efikasnije što se performansi tiče.

Model
Za čisto sihrone funkcije direktni stil je bolje rešenje.

Imajte u vidu da prepravljanje API-ja iz CPS-a u direktni stil, ili iz asinhronog u sinhroni, ili obratno, može zahtevati i menjanje stila svog koda koji taj API koristi. Na primer, u našem slučaju, moraćemo da potpuno izmenimo interfejs našeg API-ja `createFileReader()` i da ga prepravimo tako da uvek radi sinhrono.

Osim toga, pri upotrebi sinhronog API-ja umesto asimhronog, treba imati na umu sledeća upozorenja:

- Možda za datu funkcionalnost nećete uvek imati na raspolaganju sinhroni API.
- Sinhroni API će blokirati petlju za obradu događaja i time zadržati obradu drugih istovremenih zahteva. To ruši JavaScriptov model istovremenosti i usporava celu aplikaciju. U nastavku ove knjige videćemo šta to zaista znači za naše aplikacije.

U našoj funkciji `consistentReadSync()`, rizik blokiranja petje za obradu događaja je delimično ublažen time što se sinhroni API za U/I operacije poziva samo jedanput po datoteci, a u svim kasnijim pozivima koristi se postojeći sadržaj u kešu. Ako imamo mali broj statičkih datoteka, upotreba funkcije `consistentReadSync()` neće imati značajan uticaj na rad naše petlje za obradu događaja. Stvari bi se mogle brzo promeniti ako bismo morali da učitamo više datoteka i to samo jedanput. Upotreba sinhronih U/I operacija na platformi Node.js se u mnogim slučajevima nikako ne preporučuje; međutim, u nekim situacijama, to ipak može biti najjednostavnije i najefikasnije rešenje. Uvek procenite svoj konkretan slučaj upotrebe da biste izabrali pravu alternativu. Primer stvarnog slučaja upotrebe: tokom postupka pokretanja aplikacije, potpuno je logično da njenu konfiguracionu datoteku učitavate pomoću sinhronog blokirajućeg API-ja.

Koristite blokirajuće API-je samo kada oni ne ograničavaju sposobnost aplikacije da obrađuje istovremene zahteve.

Odloženo izvršavanje

Druga mogućnost za popravljanje naše funkcije `inconsistentRead()` jeste da je načinimo čisto asinhronom. U ovom slučaju „trik“ se sastoji u tome da izvršavanje sinhrone povratne funkcije odložimo u neki „budući trenutak“ umesto da se ona

izvrši odmah, u istom ciklusu petlje za obradu događaja. Na platformi Node.js, to je izvodljivo pomoću metode `process.nextTick()`, koja odlaže izvršavanje zadate funkcije do sledećeg ciklusa petlje za obradu događaja. Način delovanja je vrlo jednostavan; povratnu funkcija koju dobije kao svoj argument metoda postavlja na početak reda čekanja na obradu, pre svih U/I događaja (ako ih ima), a zatim odmah vraća kontrolu pozivaocu. Povratna funkcija će zatim biti pokrenuta čim započne sledeći ciklus petlje za obradu događaja.

Sada ćemo primeniti tu tehniku da bismo našu funkciju `inconsistentRead()` popravili kao što sledi:

```
const fs = require('fs');
const cache = {};
function consistentReadAsync(filename, callback) {
  if(cache[filename]) {
    process.nextTick(() => callback(cache[filename]));
  } else {
    // asinhrona funkcija
    fs.readFile(filename, 'utf8', (err, data) => {
      cache[filename] = data;
      callback(data);
    });
  }
}
```

Naša funkcija sada garantovano poziva svoju povratnu funkciju asinhrono, u svim okolnostima.

Drugi API za odlaganje izvršavanja jeste `setImmediate()`. Iako ima veoma sličnu namenu kao prethodni, semantika je sasvim različita. Povratne funkcije čije je izvršavanje odloženo pomoću `process.nextTick()` izvršavaju se pre obrade svih U/I događaja, dok u slučaju upotrebe metode `setImmediate()`, izvršavanje povratne funkcije se raspoređuje iza U/I događaja koji se već nalaze u redu čekanja na obradu. Pošto se `process.nextTick()` izvršava pre svih U/I događaja u redu čekanja, u nekim okolnostima može se dogoditi da se U/I operacije ne izvršavaju, na primer, kod rekurzivnog pozivanja; do toga ne može nikad doći s metodom `setImmediate()`. Razliku između ta dva API-ja naučićemo da cenimo kada u nastavku ove knjige budemo analizirali odloženo izvršavanje sihnronih poslova koji teže opterećuju CPU.

Model
Ako izvršavanje povratne funkcije odložimo pomoću `process.nextTick()`, garantujemo da će se ona izvršiti asinhrono.

Node.js konvencije za povratne funkcije

Na platformi Node.js, API-ji u stilu prosleđivanja narednog koraka i povratne funkcije slede skup određenih konvencija. Te konvencije važe za osnovni Node.js API ali primenjuju se i najvećem delu modula korisničkog prostora i aplikacija. Zbog toga

je veoma važno da ih razumemo i primenjujemo gde god nam zatreba da projektujemo neki asinhroni API.

Povratna funkcija je uvek poslednja na listi argumenata

U svim metodama koje pripadaju jezgru platforme Node.js, važi konvencija da kada funkcija prihvata povratnu funkciju kao jedan od argumenata, to treba da bude njen poslednji argument. Uzmimo kao primer sledeći API iz Node.js jezgra:

```
fs.readFile(filename, [options], callback)
```

Kao što je vidljivo iz potpisa prethodne funkcije, povratna funkcija se uvek postavlja na poslednje mesto, čak i ako postoje opcioni argumenti. Razlog za tu konvenciju jeste to da je pozivanje funkcije razumljivije ako je kôd povratne funkcije definisan lokalno, u samom API-ju.

Greška je uvek prva na listi argumenata

U stilu prosleđivanje narednog koraka (CPS), greške se dalje prosleđuju kao i svaka druga vrsta rezultata, što znači pomoću povratnih funkcija. Na platformi Node.js, svaka greška do koje dođe u CPS funkciji uvek se prosleđuje kao prvi argument povratne funkcije, a svaki stvarni rezultat se prosleđuje počev od drugog argumenta, nadalje. Ako se operacija završila bez grešaka, prvi argument će biti null ili undefined. Kôd koji sledi pokazuje kako da definišete povratnu funkciju usklađenu sa ovom konvencijom:

```
fs.readFile('foo.txt', 'utf8', (err, data) => {
  if(err)
    handleError(err);
  else
    processData(data);
});
```

Preporučena praksa je da uvek ispitujete da li postoji greška jer ako ne radite tako, biće vam teže da otklanjate greške iz svog koda i otkrivate mesta gde može doći do greške. Još jedna važna konvencija o kojoj treba voditi računa jeste da greška mora uvek biti objekat tipa `Error`. To znači da ne treba nikad prosleđivati obične znakovne nizove i brojeve kao objekte koji predstavljaju greške.

Prosleđivanje grešaka

Prosleđivanje grešaka u sinhronim funkcijama direktnog stila obavlja se pomoću dobro poznate naredbe `throw`, koja čini da greška „putuje“ naviše duž steka pozivanja funkcija sve dok negde usput ne bude presretnuta.

Međutim, a asinhronim CPS funkcijama, pravilan način za prosleđivanje grešaka jeste da se ona samo prosledi sledećoj povratnoj funkciji u lancu. Tipičan model izgleda ovako:

```
const fs = require('fs');
function readJSON(filename, callback) {
  fs.readFile(filename, 'utf8', (err, data) => {
    let parsed;
    if(err)
```

```
      // prosleđuje grešku i izlazi iz tekuće funkcije
      return callback(err);

    try {
      // raščlanjuje sadržaj datoteke
      parsed = JSON.parse(data);
    } catch(err) {
      // presreće greške pri raščlanjivanju
      return callback(err);
    }
    // nema grešaka, proslediti dobijene podatke
    callback(null, parsed);
  });
};
```

Detalj na koji bi trebalo da obratite pažnju u prethodnom kodu jeste to da se povratna funkcija sada poziva i kada želimo da prosledimo ispravan rezultat i kada želimo da prosledimo grešku. Obrtaite pažnju i na to kada prosleđujemo grešku, koristimo nardebu `return`. To činimo zato da bismo izašli iz funkcije `readJSON` odmah nakon pozivanja povratne funkcije, bez izvršavanja redova koji joj slede.

Nepresretnuti izuzeci

U funkciji `readJSON()`, koju smo upotrebili da bismo izbegli izuzetke koji bi se mogli pojaviti u povratnoj funkciji u pozivu `fs.readFile()`, možda ste zapazili blok `try...catch` u koji je umetnuta naredba `JSON.parse()`. Neobrađen izuzetak koji se pojavi unutar asinhrone povratne funkcije „skače“ naviše do petlje za obradu događaja i nikad se ne prosleđuje sledećoj povratnoj funkciji.

Na platformi Node.js, to je nepopravljivo stanje i u tom slučaju aplikacija samo prekine rad i ispisuje grešku na interfejsu `stderr`. Kao ilustraciju toga, iz prethodne definicije funkcije `readJSON()` uklonićemo blok `try...catch`:

```
const fs = require('fs');
function readJSONThrows(filename, callback) {
  fs.readFile(filename, 'utf8', (err, data) => {
    if(err) {
      return callback(err);
    }
    // nema grešaka, proslediti dobijene podatke
    callback(null, JSON.parse(data));
  });
};
```

Sada u funkciji koju smo definisali više nemamo načina da presretnemo izuzetak ukoliko se on pojavi u naredbi `JSON.parse()`. Ako neispravnu JSON datoteku pokušamo da raščlanimo pomoću sledećeg koda:

```
readJSONThrows('nonJSON.txt', err => console.log(err));
```

Rezultat će biti bezuslovno prekidanje rada aplikacije, a na konzoli će se prikazati opis sledećeg izuzetka: