# 4

# Obrasci za upravljanje asinhronim izvršavanjem koda u verziji ES2015 i novijim

U prethodnom poglavlju naučili smo kako da pišemo asinhroni kôd s povratnim funkcijama i videli kako one loše utiču na kvalitet našeg koda i uvode probleme kao što je **pakao povratnih funkcija**. Povratne funkcije su gradivni blokovi za asinhromo programiranje na jeziku JavaScript i na platformi Node.js, ali tokom godina, pojavile su se i druge alternative. Te alternative su morale biti složenije da bi omogućile pisanje asinhronog koda na pogodnije načine.

U ovom poglavlju, razmotrićemo neke od najpoznatijih alternativa, kao što su **obećanja** i **generatori**. Proučićemo i **async await**, inovativnu sintaksu koja će biti na raspolaganju u JavaScriptu kao deo verzije ECMAScript 2017.

Videćemo kako te alternative mogu da pojednostave način na koji upravljamo asinhronim izvršavanjem koda. I najzad, sve te pristupe uporedićemo međusobno da bismo shvatili dobre i loše osobine svakog od njih i bili u stanju da mudro izaberemo rešenje koje najbolje odgovara zahtevima našeg narednog Node.js projekta.

## Obećanje

U prethodnim poglavljima napomenuli smo da **CPS** (engl. *Continuation Passing Style* – stil prosleđivanja narednog koraka) nije jedini način za pisanje asinhronog koda. U stvari, JavaScript ekosistem pruža zanimljive alternative za tradicionalni model s povratnim funkcijama. Jedna od najpoznatijih alternativa jeste obećanje (engl. *promise*), koje privlači sve veću pažnju, naročito od kad je uvedeno u verziji ECMAScript 2015 a standardno je na raspolaganju na platformi Node.js od verzije 4 nadalje.

## Šta je to obećanje?

Izraženo veoma jednostavnim rečima, obećanje je apstrakcija koja funkciji omogućava da vrati objekat koji se zove `obećanje` (`promise`) i koji predstavlja (budući) konačni rezultat određene asinhrone operacije. U žargonu obećanja, kažemo da je obećanje **u toku** (engl. *pending*) dok asinhrona operacija još nije završena, ili da je **ispunjeno** (engl. *fulfilled*) kada je asinhrona operacija uspešno završena, ili da je **odbačeno** (engl. *rejected*) kada se operacija završi s nekom greškom. Pošto se obećanje ispuni ili odbaci, smatra se da je **razrešeno** (engl. *settled*).

Da bismo dobili vrednost s kojom je obećanje ispunjeno ili grešku (*razlog*) s kojom je odbačeno, možemo iskoristiti metodu `then()` obećanja. Potpis te metode izgleda ovako:

```
promise.then([onFulfilled], [onRejected])
```

U prethodnom kodu, `onFulfilled()` je metoda koja će primiti konačnu vrednost s kojom je obećanje ispunjeno, a `onRejected()` je druga metoda koja prima razlog odbacivanja obećanja (ako postoji razlog). Obe funkcije su opcione.

Da biste shvatili kako bi obećanja mogla izmeniti naš kôd, razmotrite sledeće:

```
asyncOperation(arg, (err, result) => {
  if(err) {
    // obrada greške
  }
  // nešto radimo s rezultatom
});
```

Obećanja nam omogućavaju da ovaj tipičan CPS kôd preinačimo u bolje strukturiran i elegantniji kôd, kao što je sledeći:

```
asyncOperation(arg)
  .then(result => {
    // nešto radimo s rezultatom
  }, err => {
    // obrada greške
});
```

Jedna od ključnih osobina metode `then()` jeste to što asinhrono vraća drugo obećanje. Ako bilo koja od funkcija `onFulfilled()` ili `onRejected()` vrati neku vrednost *x*, obećanje koje vrati metoda `then()` biće kao što sledi:

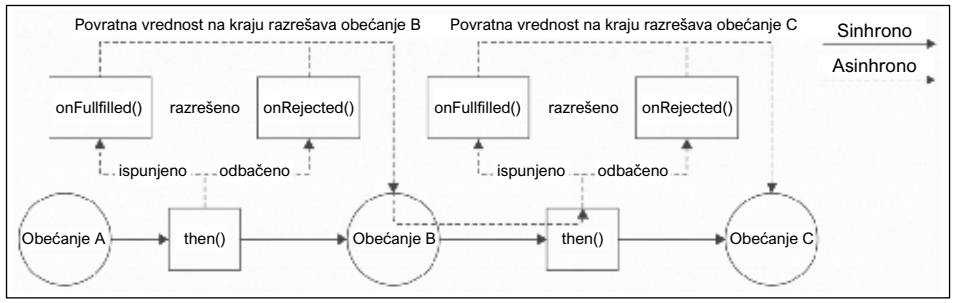
- Ispunjeno uz vrednost *x*, ako je *x* vrednost
- Ispunjeno uz vrednost of *x*, ako je *x* obećanje ili *thenable* objekat
- Odbačeno uz konačan razlog *x*, ako je *x* obećanje ili *thenable* objekat.

**Thenable** objekat je objekat sličan obećanju koji ima metodu `then()`. Izraz se koristi za opisivanje obećanja koje je *zaseban objekat*, odvojen od implementacije tekućeg obećanja.

Ta osobina nam omogućava da gradimo lance obećanja, što pak omogućava da lako grupišemo i organizujemo asinhrone operacije u više konfiguracija. Osim toga, ako ne zadamo kôd za metodu `onFulfilled()` ili `onRejected()`, vrednost s kojom je obećanje ispunjeno ili razlog zbog kojeg je odbačeno automatski se prosleđuju narednim obećanjima u lancu. To nam omogućava da, na primer, automatski prosleđujemo greške duž celog lanca sve dok ih negde ne presretne postojeća metoda `onRejected()`. Pomoću lanca obećanja, sekvencijalno izvršavanje poslova najednom postaje sasvim trivijalna operacija:

```
asyncOperation(arg)
  .then(result1 => {
    // vraća novo obećanje
    return asyncOperation(arg2);
  })
  .then(result2 => {
    // vraća vrednost
    return 'gotovo';
  })
  .then(undefined, err => {
    // ovde se presreće svaka greška koja se pojavi u lancu
  });
```

Dijagram koji sledi pruža još jedan uvid u način na koji deluje lanac obećanja.




Još jedna važna osobina obećanja je da garantuju da se funkcije `onFulfilled()` i `onRejected()` pozivaju asinhrono, čak i kada obećanje razrešimo sinhrono s nekom vrednošću, kao što smo uradili u prethodnom primeru, gde smo vratili znakovni niz `gotovo` iz poslednje `then()` funkcije u lancu. To ponašanje štiti naš kôd od svih situacija u koji bismo mogli da nenamerno pustimo Zalga s lanca (videti poglavlje 2), što bez napora čini naš asinhroni kôd doslednijim i robusnijim.

A sada dolazi ono najbolje. Ako se u metodi `onFulfilled()` ili `onRejected()` generiše izuzetak (pomoću naredbe `throw`), obećanje koje vraća metoda `then()` se automatski odbacuje, a razlog odbacivanja je upravo taj izuzetak. To je ogromna prednost u poređenju s CPS stilom kodiranja jer to znači da se kada koristite obećanja, izuzeci se automatski prosleđuju duž lanca, a naredba `throw` je konačno zaista upotrebljiva.

Tokom vremena ponuđeno više raznih implementacija biblioteka obećanja, koje najčešće nisu bile međusobno kompatibilne, što znači da nije bilo moguće formirati lance thenable objekata koje su generisala obećanja iz biblioteka različitih implementacija obećanja.

Zajednica JavaScript programera je uložila veliki trud da bi uklonila to ograničenje i ti napori su doveli do izrade specifikacije **Promises/A+**. Ta specifikacija detaljno opisuje ponašanje metode `then`, čime se obezbeđuje temelj za međuoperabilnost, što omogućava da objekti tipa obećanje iz različitih biblioteka budu u stanju da međusobno sarađuju bez ikakvih izmena.

Detaljni opis specifikacije **Promises/A+** naći ćete na njenoj zvaničnoj veb stranici, `https://promisesaplus.com`.

## Implementacije specifikacije Promises/A+

Za JavaScript, kao i za Node.js, postoji više biblioteka koje implementiraju specifikaciju Promises/A+. Najpopularnije su sledeće:

- Bluebird (`https://npmjs.org/package/bluebird`)
- Q (`https://npmjs.org/package/q`)
- RSVP (`https://npmjs.org/package/rsvp`)
- Vow (`https://npmjs.org/package/vow`)
- When.js (`https://npmjs.org/package/when`)
- ES2015 obećanja

Ono po čemu se ove implementacije zaista međusobno razlikuju jeste dodatan skup mogućnosti koje svaka nudi pored standarda Promises/A+. Kao što je već rečeno, standard definiše ponašanje metode `then()` i postupak razrešavanja obećanja, ali ne propisuje druge vrste funkcionalosti, na primer, kako se obećanje definiše u asinhronoj funkciji koja radi s povratnom funkcijom.

U našim primerima, koristićemo skup API-ja koji implementiraju ES2015 obećanja jer su standardno na raspolaganju na platformi Node.js od verzije 4 nadalje i za njih nije potrebna podrška u vidu dopunskih spoljašnjih biblioteka.

Kao referenca, sledi lista API-ja za ES2015 obećanja:

**Constructor (new Promise(function(resolve, reject) {}))**: Ovaj konstruktor definiše novo obećanje koje će biti ispunjeno ili odbačeno u zavisnosti od ponašanja funkcije koju zadate kao argument. Argumenti kosntruktora su sledeći:

- `resolve(obj)`: Ovaj argument razrešava obećanje kao ispunjeno, a vrednost ispunjenog obećanja biće `obj` ako je `obj` neka vrednost. Vrednost ispunjenog obećanja biće takođe `obj` ako je objekat `obj` i sam obećanje ili thenable objekat.
- `reject(err)`: Ovaj argument razrešava obećanje kao odbačeno, a razlog odbacivanja je `err`. Važi konvencija koja propisuje da `err` bude instanca `Error`.

**Statičke metode objekta Promise**:

- `Promise.resolve(obj)`: Pravi novo obećanje od zadatog thenable objekta ili vrednosti.
- `Promise.reject(err)`: Pravi novo obećanje, odbačeno zbog razloga `err`.
- `Promise.all(iterable)`: Pravi novo obećanje, koje će postati ispunjeno kada svaki element argumenta `iterable` bude ispunjen, ili odbačeno zbog razloga odbacivanja prvog elementa argumenta `iterable` koji bude odbačen. Svaki element argumenta `iterable` može biti obećanje, generički thenable objekat ili vrednost.
- `Promise.race(iterable)`: Pravi obećanje koje se razrešava kao ispunjeno ili odbačeno čim jedno od obećanja u argumentu `iterable` bude ispunjeno ili odbačeno, s vrednošću ili razlogom iz tog obećanja.

**Metode instance obećanja**:

- `promise.then(onFulfilled, onRejected)`: Ovo je najvažnija metoda obećanja i njeno ponašanje je kompatibilno sa ranije opisanim standardom Promises/A+.
- `promise.catch(onRejected)`: Ovo je samo lepši sintaksni oblik metode `promise.then(undefined, onRejected)`.

Vredi napomenuti da neke implementacije obećanja nude drugačiji mehanizam za definisanje novih obećanja, nazvan **deferreds**. Ovde ga nećemo opisati zato što nije deo standarda ES2015, ali ako želite da saznate više, možete pročitati dokumentaciju za Q (`https://github.com/kriskowal/q#using-deferreds`) ili When.js (`https://github.com/cujojs/when/wiki/Deferred`).

## Promisifikovanje funkcije u Node.js stilu

U JavaScriptu, obećanja nisu standardno podržana u svim asinhronim funkcijama i bibliotekama. U najvećem broju slučajeva, tipičnu funkciju koja radi s povratnom funkcijom moramo da prepravimo tako da vraća obećanje; taj postupak je poznat i kao **promisifikovanje**.

Srećom, konvencije za upotrebu povratnih funkcija koje se koriste na platformi Node.js omogućavaju nam da napišemo višekratno upotrebljivu funkciju koju možemo iskoristiti za promisifikovanje svakog API-ja u Node.js stilu. To je lako izvodljivo pomoću konstruktora objekta `Promise`. Zato ćemo napisati novu funkciju `promisify()` i umetnuti je u modul `utilities.js` (da bismo kasnije mogli da je upotrebimo u našoj aplikaciji veb pauka):

```
module.exports.promisify = function(callbackBasedApi) {
  return function promisified() {
    const args = [].slice.call(arguments);
    return new Promise((resolve, reject) => {                    //[1]
      args.push((err, result) => {                               //[2]
        if(err) {
```

```
            return reject(err);                                    //[3]
          }
          if(arguments.length <= 2) {                              //[4]
            resolve(result);
          } else {
            resolve([].slice.call(arguments, 1));
          }
        });
        callbackBasedApi.apply(null, args);                        //[5]
      });
    }
};
```

Prethodna funkcija vraća novu funkciju koja se zove `promisified()` i predstavlja promisifikovanu verziju funkcije `callbackBasedApi` zadate kao ulazni argument. Evo kako to radi:

1. Funkcija `promisified()` pravi novo obećanje pomoću konstruktora objekta `Promise` i odmah potom vraća kontrolu pozivaocu.
2. U funkciji koju zadajemo konstruktoru objekta `Promise`, obavezno prosleđujemo funkciju `callbackBasedApi`, što je specijalna povratna funkcija. Pošto znamo da je povratna funkcija uvek poslednji argument, samo je dodajemo na kraj liste argumenata (`args`) funkcije `promisified()`.
3. U specijalnoj povratnoj funkciji, ako dobijemo grešku, obećanje odmah odbacujemo.
4. Ako ne dobijemo nijednu grešku, obećanje razrešavamo s jednom vrednošću ili s nizom vrednosti, u zavisnosti od toga koliko je rezultata predato povratnoj funkciji.
5. I najzad, samo pozivamo funkciju `callbackBasedApi` s listom argumenata koju smo sastavili.

Većina implementacija obećanja standardno nude neku vrstu pomagala za konvertovanje API-ja u Node.js stilu tako da vraćaju obećanja. Q ima `Q.denodeify()` i `Q.nbind()`, Bluebird ima `Promise.promisify()`, a When.js ima `node.lift()`.

## Sekvencijalno izvršavanje

Nakon malo neophodne teorije, sada smo spremni da našu aplikaciju veb pauka prepravimo tako da koristi obećanja. Počećemo direktno od verzije 2, one koja preuzima sadržaj sa URL-ova na veb stranici sekvencijalnim redosledom.

U modulu `spider.js`, prvi obavezan korak jeste da učitamo našu implementaciju obećanja (koju ćemo kasnije koristiti) i da promisifkujemo funkcije koje nameravamo da koristimo a koje sada rade s povratnim funkcijama:

```
const utilities = require('./utilities');

const request = utilities.promisify(require('request'));
const mkdirp = utilities.promisify(require('mkdirp'));
const fs = require('fs');
const readFile = utilities.promisify(fs.readFile);
const writeFile = utilities.promisify(fs.writeFile);
```

Sada možemo da počnemo prepravljanje funkcije `download()`:

```
function download(url, filename) {
  console.log(`Preuzimanje sa ${url}`);
  let body;
  return request(url)
    .then(response => {
    body = response.body;
    return mkdirp(path.dirname(filename));
  })
  .then(() => writeFile(filename, body))
  .then(() => {
    console.log(`Preuzeto i upisano: ${url}`);
    return body;
  });
}
```

Važan deo na koji ovde treba obratiti pažnju jeste to da smo za obećanje koje vraća funkcija `readFile()` registrovali funkciju `onRejected()` radi obrade slučajeva kada sadržaj stranice nije preuzet (datoteka ne postoji). Osim toga, zanimljiv je način na koji smo pomoću naredbe `throw` dalje prosledili grešku iz funkcije.

Pošto prepravimo i našu funkciju `spider()`, način njenog pozivanja možemo izmeniti kao što sledi:

```
spider(process.argv[2], 1)
  .then(() => console.log('Preuzimanje sadržaja je završeno'))
  .catch(err => console.log(err));
```

Obratite pažnju na to kako smo iskoristili, po prvi put, sintaksnu pogodnost `catch` za obradu svih grešaka do kojih bi moglo doći u funkciji `spider()`. Ako ponovo razmotrimo ceo kôd koji smo dosad napisali, prijatno bi nas iznenadila činjenica da nismo ugradili nikakvu izričitu logiku za prosleđivanje grešaka, što bismo morali da obavezno uradimo kada koristimo povratne funkcije. Jasno je da to predstavlja ogromnu prednost, zato što značajno smanjuje količinu šablonskog koda koji se ponavlja, kao i mogućnost da nam promakne neobrađena neka asinhrona greška.

Sada jedini nedostajući element za dovršavanje verzije 2 našeg veb pauka jeste funkcija `spiderLinks()`, koju ćemo implementirati malo kasnije.

## Sekvencijalna iteracija

Dosad je kôd veb pauka bio uglavnom samo uvod u šta su obećanja i kako se ona koriste, kao i ilustracija kako se lako i jednostavno implementira sekvencijalni način izvršavanja pomoću obećanja. Međutim, kôd koji smo dosad razmatrali omogućavao je samo izvršavanje grupe zadatih asinhronih operacija. Dakle, nedostajući deo koji će kompletirati naše razmatranje sekvencijalnih načina izvršavanja jeste da vidimo kako možemo implementirati iteraciju pomoću obećanja. Savršen primer za tu namenu ponovo je funkcija `spiderLinks()` iz verzije 2 našeg veb pauka.

Dodajmo nedostajući deo:

```
function spiderLinks(currentUrl, body, nesting) {
  let promise = Promise.resolve();
  if(nesting === 0) {
    return promise;
  }
  const links = utilities.getPageLinks(currentUrl, body);
  links.forEach(link => {
    promise = promise.then(() => spider(link, nesting - 1));
  });
  return promise;
}
```

Za asinhronu iteraciju kroz sve URL-ove na veb stranici morali smo da dinamički sastavimo lanac obećanja:

1. Prvo, definisali smo „prazno“ obećanje, koje se razrešava kao `undefined`. To obećanje služi samo kao početna tačka za izradu našeg lanca.
2. Zatim, u petlji ažuriramo promenljivu `promise` novim obećanjem koje dobijemo nakon pozivanja funkcije `then()` prethodnog obećanja u lancu. Taj deo je zapravo naš model asinhrone iteracije pomoću obećanja.

Na taj način, po završetku petlje, promenljiva `promise` će sadržati obećanje koje je vratila poslednja pozvana funkciija `then()` u petlji, a ono će biti razrešeno kao ispunjeno samo ako su tako razrešena i sva obećanja u lancu.

Time smo u potpunosti konvertovali verziju 2 našeg veb pauka tako da koristi obećanja. Trebalo bi da sada bude u stanju da ga ponovo isprobamo.

## Sevencijalna iteracija – obrazac

Kao završetak odeljka o sekvencijalnom izvršavanju, izvešćemo obrazac za sekvencijalnu iteraciju zadate grupe obećanja:

```
let tasks = [ /* ... */ ]
let promise = Promise.resolve();
tasks.forEach(task => {
  promise = promise.then(() => {
    return task();
  });
});
```

```
promise.then(() => {
  // Svi poslovi su završeni
});
```

Alternativa upotrebi petlje `forEach()` jeste funkcija `reduce()`, što omogućava još sažetiji kôd:

```
let tasks = [ /* ... */ ]
let promise = tasks.reduce((prev, task) => {
  return prev.then(() => {
    return task();
  });
}, Promise.resolve());
promise.then(() => {
  // Svi poslovi su završeni
});
```

Kao i uvek u slučajevima jednostavnih prilagođavanja ovog obrasca, rezultate svih poslova možemo smestiti u jedan niz, ili možemo implementirati algoritam za preslikavanje ili napraviti neki filtar i tako dalje.

**Obrazac (sekvencijalna iteracija pomoću obećanja)**:
Ovaj obrazac dinamički formira u petlji lanac obećanja.

## Paralelno izvršavanje

Još jedan način izvršavanja koda koji postaje trivijalan ako se upotrebe obećanja jeste paralelno izvršavanje. U stvari, sve što treba je da iskoristimo ugrađenu funkciju `Promise.all()`. Ta pomoćna funkcija pravi novo obećanje, koje postaje ispunjeno samo kada budu ispunjena sva obećanja zadata kao ulazni argumenti funkcije. To je u suštini paralelan način izvršavanja zato što ne određuje nikakav poseban redosled razrešavanja ulaznih obećanja.

Kao ilustraciju toga, razmotrićemo verziju 3 naše aplikacije veb pauka, koja na paralelan način preuzima sadržaje sa svih URL-ova na veb stranici. Funkciju `spiderLinks()` ponovo ćemo ažurirati tako da implementira paralelno izvršavanje, ali ovog puta pomoću obećanja:

```
function spiderLinks(currentUrl, body, nesting) {
  if(nesting === 0) {
    return Promise.resolve();
  }

  const links = utilities.getPageLinks(currentUrl, body);
  const promises = links.map(link => spider(link, nesting - 1));

  return Promise.all(promises);
}
```

Obrazac se u ovom slučaju sastoji od pokretanja svih `spider()` poslova u isto vreme, u petlji `elements.map()`, koja prikuplja i sva njihova obećanja. Ovog puta, u petlji ne čekamo da se završi prethodno pokrenuti postupak preuzimanja sadržaja pre nego što započnemo sledeći: sve poslove preuzimanja sadržaja pokrećemo (gotovo) istovremeno, jedan za drugim. Nakon toga, pozivamo metodu `Promise.all()`, koja vraća novo obećanje, koje će biti ispunjeno kada budu ispunjena sva obećanja zadata u ulaznom nizu. Drugim rečima, ono postaje ispunjeno kada se završe svi poslovi preuzimanja sadržaja; tačno to smo i želeli.

## Ograničeno paralelno izvršavanje

Nažalost, ES2015 API Promise nema ugrađen način za ograničavanje broja istovremenih poslova, ali uvek možemo iskoristiti ono što smo naučili o ograničavanju broja istovremenih pomoću čistog JavaScripta. U stvari, model koji smo implementirali unutar klase `TaskQueue` lako se može prilagoditi tako da podržava poslove koji vraćaju obećanje. To ćemo lako postići izmenom metode `next()`:

```
next() {
  while(this.running < this.concurrency && this.queue.length) {
    const task = this.queue.shift();
    task().then(() => {
      this.running--;
      this.next();
    });
    this.running++;
  }
}
```

Umesto da poslove obrađujemo u povratnoj funkciji, samo pozivamo funkciju `then()` obećanja koje metoda vrati. Preostali deo koda je praktično identičan staroj verziji klase `TaskQueue`.

Vratimo se na modul `spider.js` i izmenimo ga tako da podržava našu novu verziju klase `TaskQueue`. Prvo, definišemo novu instancu klase `TaskQueue`:

```
const TaskQueue = require('./taskQueue');
const downloadQueue = new TaskQueue(2);
```

Zatim, ponovo je na redu funkcija `spiderLinks()`. Izmena je i u ovom slučaju prilično lako razumljiva:

```
function spiderLinks(currentUrl, body, nesting) {
  if(nesting === 0) {
    return Promise.resolve();
  }

  const links = utilities.getPageLinks(currentUrl, body);
  // potrebno nam je sledeće zato što objekat Promise koji zatim
  // pravimo neće nikad biti razrešen ako nema poslova za obradu
  if(links.length === 0) {
    return Promise.resolve();
  }
```

```
  return new Promise((resolve, reject) => {
    let completed = 0;
    let errored = false;
    links.forEach(link => {
      let task = () => {
        return spider(link, nesting - 1)
          .then(() => {
            if(++completed === links.length) {
              resolve();
            }
          })
          .catch(() => {
            if (!errored) {
              errored = true;
              reject();
            }
          });
      };
      downloadQueue.pushTask(task);
    });
  });
}
```

U prethodnom kodu ima nekoliko stvari koje zaslužuju našu pažnju:

- Prvo, treba da nam novo obećanje koje napravimo pomoću konstruktora klase `Promise`. Kao što ćemo videti, to nam omogućava da obećanje razrešimo ručno, kada se završe svi poslovi iz reda čekanja.
- Drugo, pogledajmo pažljivije kako je sada zadatak definisan. Obećanju koje nam vraća zadatak `spider()` pridružujemo povratnu funkciju `onFulfilled()`, što nam omogućava da prebrojavamo završene poslove preuzimanja sadržaja. Kada broj završenih operacija preuzimanja sadržaja postane jednak broju URL-ova na veb stranici, znamo da je obrada završena, pa zato možemo pozvati funkciju `resolve()` spoljašnjeg obećanja.

Specifikacija Promises/A+ propisuje da se sledeće dve povratne funkcije `onFulfilled()` i `onRejected()` metode `then()` pozivaju samo po jednaput i to ekskluzivno (jedna ili druga, ali ne obe). Implementacija obećanja koja poštuje specifikaciju obezbeđuje da čak i ako funkciju `resolve` ili `reject` pozovemo više puta, obećanje je ispunjeno ili odbačeno samo jedanput.

Trebalo bi da sada bude spremna za probu verzija 4 naše aplikacije veb pauka. Možda ćemo ponovo zapaziti da se poslovi preuzimanja sadržaja odvijaju paralelno, uz ograničenje od najviše dva istovremena posla.

## Izlaganje povratnih funkcija i obećanja kao javnih API-ja

Kao što smo saznali u prethodnim odeljcima, obećanja se mogu iskoristiti kao zgodna zamena za povratne funkcije. Pokazuju se kao veoma korisna kada treba načiniti kôd čitjivijim i razumljivijim. Obećanja pružaju mnoge prednosti, ali i zahtevaju od programera da razume više netrivijalnih koncepata da bi mogao da koristi obećanja na ispravan i efikasan način. Iz tog i iz drugih razloga, u nekim slučajevima povratne funkcije mogu biti bolje rešenje od obećanja.

A sada zamislimo da želimo da napišemo javu biblioteku koja obavlja asinhrone operacije. Kako da to uradimo? Da li da napravimo API s povratnim funkcijama ili s obećanjima? Da li da se operdelimo za jednu ili za drugu stranu ili postoje načini da podržimo obe mogućnosti i da tako svi budu srećni?

To je problem s kojim se suočavaju mnoge dobro poznate biblioteke, za koji postoje barem dva rešenja koja vredi pomenuti i koja nam omogućavaju da obezbedimo raznovrstan API.

Prvo rešenje, koje koriste biblioteke kao što su `request`, `redis` i `mysql`, sastoji se od jednostavnog API-ja koji prihvata isključivo povratne funkcije a programeru pruža samo još mogućnost da promisifikuje izložene funkcije ako je potrebno. Neke od tih biblioteka stavljaju na raspolaganje pomoćne funkcije za promisifikovanje svih asinhronih funkcija koje one nude, ali programer i dalje mora da određeni način konvertuje izloženi API ako želi da koristi obećanja.

Drugo rešenje je transparentnije. Takođe nudi API koji prihvata samo povratne funkcije, ali argument povratne funkcije nije obavezan. Gde god zadate povratnu funkciju kao vrednost tog argumenta, funkcija će ponašati normalno i izvršiti povratnu funkciju na svom završetku ili ako se pojavi greška. Ako joj ne prosledite povratnu funkciju kao argument, funkcija će odmah vratiti objekat `Promise`. To rešenje efikasno kombinuje povratne funkcije s obećanjima što programeru omogućava da u vreme pozivanja funkcije odlučuje koji će interfejs izabrati, bez potrebe da unapred promisifikuje funkciju. Mnoge biblioteke, kao što su `mongoose` i `sequelize`, podržavaju ovo rešenje.

Sada ćemo razmotriti jednostavnu implementaciju tog rešenja. Recimo da želimo da implementiramo prazan modul koji asinhrono izvršava operacije deljenja:

```
module.exports = function asyncDivision (dividend, divisor, cb) {
  return new Promise((resolve, reject) => {                    // [1]

    process.nextTick(() => {
      const result = dividend / divisor;
      if (isNaN(result) || !Number.isFinite(result)) {
        const error = new Error('Pogrešni operandi');
        if (cb) { cb(error); }                                 // [2]
        return reject(error);
      }
      if (cb) { cb(null, result); }                            // [3]
      resolve(result);
    });

  });
};
```

Kôd ovog modula je lako razumljiv, ali ipak ima nekoliko detalja koje vredi istaći:

- Prvo, vraćamo novo obećanje koje pravimo pomoću konstruktora klase `Promise`. Celokupnu logiku definišemo unutar funkcije koju prosleđujemo kao argument konstruktoru.
- U slučaju grške, odbacujemo obećanje, ali ako je u vreme pozivanja prosleđena povratna funkcije, izvršavamo je da bismo grešku prosledili dalje.
- Pošto izračunamo rezultat, razrešavamo obećanje, ali i u ovom slučaju, ako je zadata povratna funkcija, rezultat prosleđujemo i njoj.

Pogledajmo sada kako bismo koristili ovaj modul, i s povratnim funkcijama i s obećanjima:

```
// upotreba povratnih funkcija
asyncDivision(10, 2, (error, result) => {
  if (error) {
    return console.error(error);
  }
  console.log(result);
});

// upotreba obećanja
asyncDivision(22, 11)
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

Trebalo bi da bude jasno da ako ulože vrlo mali napor, programeri koji će koristiti naš novi modul moći će lako da biraju stil koji najboje odgovara njihovim potrebama, bez obaveze da uvode dodatnu spoljašnju funkciju za promisifikovanje kad god požele da koriste obećanja.

# Generatori

Specifikacija ES2015 uvodi još jedan mehanizam koji se, između ostalog, može iskoristiti za pojednostavjivanje načina upravljanja asinhronim izvršavanjem naših Node.js aplikacija. Reč je o **generatorima**, poznatim i kao **polu-korutine** (engl. *semi-coroutines*). To je oblik generalizacije podrutina, gde može postojati više različitih ulaznih tačaka u istu rutinu. Standardna funkcija može zapravo imati samo jednu ulaznu tačku, koja odgovara mestu pozivanja same funkcije. Generator je sličan funkciji, smo što se njegovo izvršavanje može privremeno zaustaviti (pomoću naredbe `yield`) a onda kasnije nastaviti. Generatori su naročito korisni za implementiranje iteratora, a trebalo bi da vam oni budu poznati pošto smo već ranije razmatrali kako se iteratori mogu iskoristiti za implementiranje važnih modela za upravljanje asinhronim načinima izvršavanja koda, kao što su sekvencijalno ili ograničeno paralelno izvršavanje.