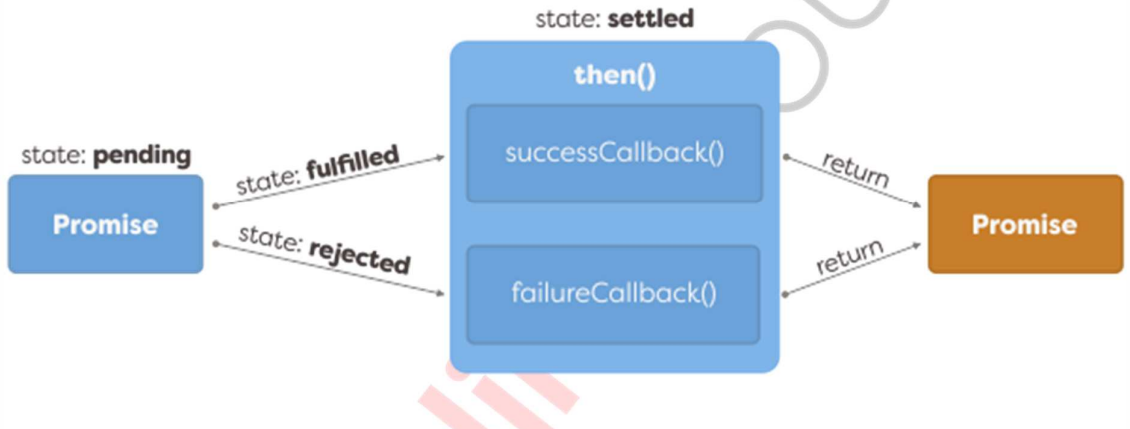
Napredno korišćenje Promisea

Prethodna lekcija ilustrovala je osnovne osobine Promisea u jeziku JavaScript. Mogli ste da se upoznate sa njihovom namenom, načinom kreiranja i konzumiranja. U ovoj lekciji nastavljamo tamo gde smo stali, pa će tako biti prikazani neki napredni scenariji rada Promiseima, pre svih, njihovo nadovezivanje i obrada grešaka. Na kraju lekcije, biće prikazan i jedan realan primer korišćenja Promisea prilikom rada sa ugrađenim funkcionalnostima web pregledača za čitanje sadržaja fajlova.

Nadovezivanje Promisea

U prethodnoj lekciji, kada je bilo reči o stanjima Promisea, prikazan je dijagram koji je ilustrovao životni tok jednog Promise objekta. Ipak, jedan deo takvog dijagrama, koji ilustruje veoma važnu osobinu Promisea, namerno je izostavljen (slika 10.1).



Slika 10.1. Metoda then() kao svoju povratnu vrednost emituje Promise objekat

Sada je dijagramu životnog toka jednog Promise objekta dodat i nešto ranije izostavljeni deo. Sa slike 10.1. može se videti da metoda `then()` kao svoju povratnu vrednost emituje Promise objekat. U pitanju je potpuno novi Promise objekat, što je na slici 10.1. jasno naznačeno korišćenjem drugačije boje za obeležavanje takvog objekta.

Činjenica da `then()` metoda kao svoju povratnu vrednost emituje novi Promise objekat omogućava da se postigne nadovezivanje Promisea. Nadovezivanje Promisea prvo će biti ilustrovano na jednom banalnom primeru:

```
let promise = new Promise(function (resolve, reject) {
    resolve(4);
});

let promise2 = promise.then(function (value) {
    console.log(value);
    return value*value;
});

promise2.then(function (value) {
    console.log("Final value is: " + value);
});
```

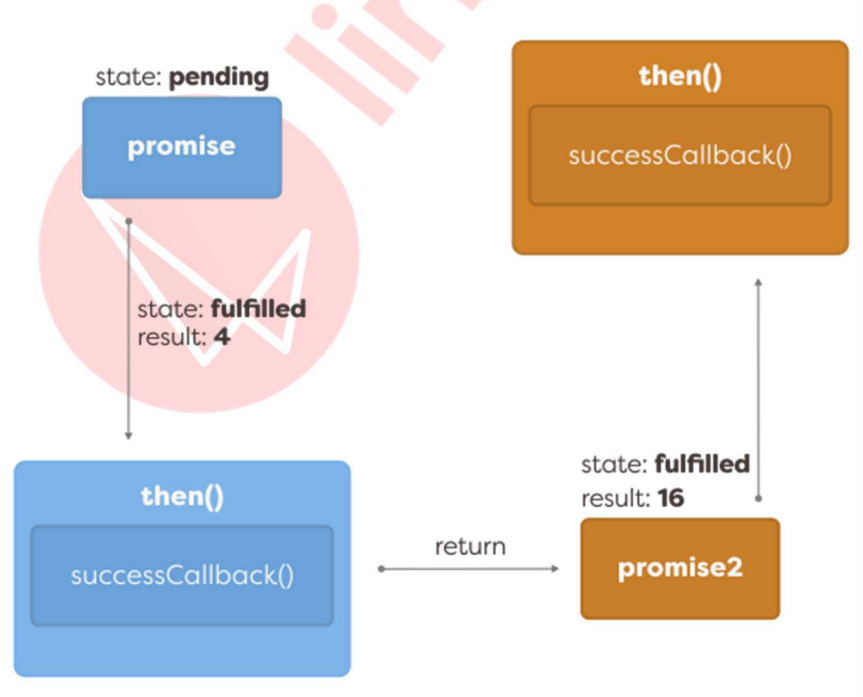
Prikazani primer, iako bez neke posebne upotrebne vrednosti, odlično ilustruje efekat nadovezivanja Promisea. U primeru je prvo kreiran jedan Promise objekat. Njegova logika je vrlo jednostavna. On odmah emituje dojavu o uspešnosti izvršavanja logike Promisea i tom prilikom se kao povratna vrednost prosleđuje broj 4. Ovakav Promise obrađuje se u sledećoj naredbi, na sličan način kao i dosad. Ipak, postoje određene razlike u odnosu na prethodne primere. S obzirom na to da `then()` metoda emituje novi Promise objekat, referenca na takav objekat je smeštena unutar promenljive `promise2`. Druga razlika odnosi se na logiku callback funkcije. Unutar callback funkcije, sada se obavlja obrada dobijene povratne vrednosti (vrednost se množi sama sobom), a zatim se takva nova vrednost emituje kao povratna.

Sve ovo stvoriće sledeći efekat: pozivanje metode `then()` nad originalnim Promise objektom stvoriće novi Promise objekat, koji će odmah nakon završetka callback funkcije ući u stanje *fulfilled*, što će za efekat imati pozivanje callback funkcije koja je nad takvim objektom definisana. Takva callback funkcija će automatski da dobije vrednost koju je callback funkcija prethodnog Promisea emitovala kao povratnu. Upravo je to uređeno u prikazanom primeru. Poslednja naredba ilustruje nadovezivanje Promise objekata. Nad promenljivom `promise2` pozvana je metoda `then()` i definisana finalna callback funkcija. Kao što je rečeno, ona će automatski dobiti povratnu vrednost prethodnog Promisea, te će stoga kompletan primer da unutar konzole ispiše:

```
4
Final value is: 16
```

Vrednost 4 je ona koja se štampa iz callback funkcije prvog Promise objekta, neposredno pre nego što se transformiše i prosledi kao povratna. Poruka *Final value is: 16* štampa se iz callback funkcije drugog Promise objekta.

Tok upravo prikazanog primera ilustruje i slika 10.2.



Slika 10.2. Tok izvršavanja upravo prikazanog primera

Upravo prikazani primer nadovezivanja Promisea sa razlogom je razdvojen u nekoliko naredbi, uz upotrebu koda za smeštanje povratnih vrednosti u zasebne promenljive. Ipak, u praksi se mnogo češće kompletan lanac Promisea definiše jednom naredbom. Tako se prikazani primer kompaktnije može napisati ovako:

```
new Promise(function (resolve, reject) {
    resolve(4);
}).then(function (value) {
    console.log(value);
    return value * value;
}).then(function (value) {
    console.log("Final value is: " + value);
});
```

Sada je kod napisan u nešto kompaktnijem formatu, koji više asocira na nadovezivanje koje se u praksi postiže.

Ukoliko se u primer uključe i Arrow funkcije koje su ilustrovane u jednoj od prethodnih lekcija ovoga kursa, dobija se još kompaktniji kod:

```
new Promise(function (resolve, reject) {
    resolve(4);
}).then( (value) => {
    console.log(value);
    return value * value;
}).then( (value) => {
    console.log("Final value is: " + value);
});
```

Pitanje

Metoda `then()` kao svoju povratnu vrednost emituje:

- **Promise**
- Object
- String
- null

Objašnjenje:

Metoda za konzumiranje Promisea – then(), kao svoju povratnu vrednost emituje novi Promise objekat.

Nadovezivanje asinhronih operacija

Prethodni primer ilustrovao je veoma važnu osobinu Promisea – metoda `then()` kao svoju povratnu vrednost emituje novi Promise objekat, što na kraju omogućava nadovezivanje Promisea. Ipak, prethodni primer je ilustrovao najjednostavniji scenario nadovezivanja Promisea. Kada callback funkcija, koja se definiše kao `then()` parametar, kao svoju povratnu vrati neku prostu vrednost, obavlja se ono što i u prethodnom primeru. Ipak, problem može nastati ukoliko pokušavamo da na ovaj način nadovežemo neke asinhrone operacije:

```
new Promise(function (resolve, reject) {
    resolve(4);
}).then(function (value) {

    setTimeout(() => value = value * value, 2000);
    return value;

}).then(function (value) {
    console.log("Final value: " + value);
});
```

Sada je izmenjena logika callback funkcije prve `then()` metode. Nova vrednost se izračunava tek nakon dve sekunde. Unutar konzole se dobija:

```
Final value: 4
```

Na osnovu ispisa se može zaključiti da je finalna vrednost koja se dobija u poslednjoj callback funkciji identična onoj koja je emitovana iz prvog Promise objekta. Potpuno očekivano, nije se čekalo na obavljanje naše vremenski zahtevne operacije u prvoj callback funkciji. Razlog je i više nego jednostavan – čim se u callback funkciji pozove naredba `return`, dolazi do emitovanja novog Promise objekta, koji je automatski u *fulfilled* stanju.

Kako bismo poslednju callback funkciju naterali da čeka na završetak neke vremenski zahtevne operacije, neophodno je iz prethodne callback funkcije vratiti Promise objekat:

```
new Promise(function (resolve, reject) {
    resolve(4);
}).then(function (value) {

    return new Promise(function(resolve, reject){
        setTimeout(() => resolve (value * value), 2000);
    });

}).then(function (value) {
    console.log("Final value: " + value);
});
```

Sada je logika prve callback funkcije optimizovana tako da vraća Promise objekat. Unutar takvog Promise objekta je spakovana vremenski zahtevna operacija, a, nakon njenog isteka, unutar kreiranog Promisea poziva se `resolve()` metoda i njoj se prosleđuje novoizračunata vrednost (`value * value`).

U ovakvim situacijama, kada callback funkcija emituje Promise objekat, sledeća callback funkcija čeka na završetak logike takvog Promisea, što se može videti i u primeru. Stoga je ovo vrlo efektan način da se postigne nadovezivanje više asinhronih operacija.
Još jedan primer nadovezivanja asinhronih operacija emitovanjem Promise objekata iz callback funkcija podrazumevaće korišćenje funkcije `delay()`, koju smo kreirali u prethodnoj lekciji:

```
function delay(time) {
    return new Promise(function (resolve, reject) {
        setTimeout(resolve, time);
    });
}

delay(4000).then(function() {
```

```
        console.log("4 seconds passed...");
        return delay(2000);
    }).then(function() {
        console.log("2 more seconds passed...");
        return delay(3000);
    }).then(function () {
        console.log("3 more seconds passed...");
    });
```

Unutar funkcije `delay()` jedna tradicionalna funkcionalnost web pregledača koja se zasniva na callback funkcijama upakovana je u Promise objekat. Ovoga puta, takvu funkciju koristimo za nadovezivanje tri Promise objekta. Prvi Promise objekat se kreira pozivanjem funkcije `delay()` sa parametrom 4000. Zatim se iz prve callback funkcije, kao povratna vrednost, emituje drugi Promise objekat, pozivanjem funkcije `delay()` sa vrednošću 2000. Na kraju se unutar sledeće callback funkcije kreira još jedan, treći Promise objekat, pozivanjem metode `delay()` sa vrednošću 3000. Sve ovo će unutar konzole stvoriti sledeći efekat:

```
4 seconds passed...
2 more seconds passed...
3 more seconds passed...
```

Obrada grešaka Promisea

Za kraj priče o Promiseima, biće ilustrovane i različite tehnike za obradu grešaka, do kojih može doći prilikom izvršavanja logike Promisea. Neke od njih su dosad već ilustrovane, ali će sada svakako biti prikazane ponovo, s obzirom na to da počinjemo od osnovnih primera.

Emitovanje neuspešnog završetka operacije unutar Promisea

Osnovni način za signaliziranje pojave greške tokom logike Promisea jeste pozivanje metode na koju upućuje drugi parametar funkcije egzekutor:

```
let promise = new Promise(function (resolve, reject) {
      setTimeout(() => reject(new Error("Unknown error.")), 3000);
});
```

Bitno je reći da se metodi za signaliziranje greške ne mora proslediti `Error` objekat, već to može biti i vrednost bilo kog drugog tipa:

```
 let promise = new Promise(function (resolve, reject) {
    setTimeout(() => reject("Unknown error."), 3000);
});
```

U oba prikazana primera Promise će se završiti neuspehom, odnosno Promise će se ispuniti ulaskom u stanje *rejected*.

Konzumiranje rejected Promisea

Osnovni način na koji se konzumira *rejected* Promise jeste definisanje logike koja će se u takvoj situaciji aktivirati, i to njenim smeštanjem unutar callback funkcije koja se, kao drugi parametar, prosleđuje metodi `then()`:

```
promise.then(function(result) {
    console.log(result);
},
function(error) {
    console.log(error);
});
```

U ovakvoj situaciji aktiviraće se druga callback funkcija i unutar konzole će biti ispisan tekst greške.

Dojavu o grešci unutar Promisea moguće je konzumirati i korišćenjem metode `catch()` prikazane u prethodnoj lekciji:

```
promise.catch(function(error) {
    console.log(error);
});
```

Naravno, ovakav pristup se može koristiti samo ukoliko nismo zainteresovani za konzumiranje rezultata uspešnog izvršavanja Promisea.

Implicitno odbijanje Promisea

Ono što je bitno znati jeste i to da se emitovanje neuspešnog završetka operacije unutar Promisea može postići i bez direktnog pozivanja metode koja je za to namenjena. Kada se unutar Promisea izbaci bilo koji izuzetak, JavaScript izvršno okruženje automatski menja status Promisea u *rejected,* a izuzetak do koga je došlo emituje se kao povratna vrednost:

```
let promise = new Promise(function (resolve, reject) {
      throw new Error("Unknown error.");
});

promise.catch(function (error) {
      console.log(error);
});
```

Bitno je znati i to da se na ovakvu osobinu može računati samo ukoliko je reč o sinhronom kodu koji se izvršava unutar Promisea. Drugim rečima, greška koju proizvede bilo koji sinhroni kod unutar Promisea tretiraće se kao odbacivanje Promisea, promenom njegovog statusa u *rejected.* Ipak, isto ne važi i za asinhroni kod, s obzirom na to da on suštinski ne pripada kodu Promisea:

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => { throw new Error("Unknown error.") },
3000);
});

promise.catch(function (error) {
    console.log(error);
});
```

Sada se izuzetak emituje unutar asinhrone operacije koja će se aktivirati sa zadrškom od tri sekunde. Unutar konzole se dobija neobrađeni izuzetak:

```
Uncaught Error: Unknown error.
```

Obrada grešaka prilikom nadovezivanja Promisea

Još jedno veoma značajno unapređenje Promisea u odnosu na pristup koji je podrazumevao korišćenje callback funkcija ispoljava se u domenu obrade grešaka prilikom nadovezivanja Promisea. Podsetimo se jednog od prethodnih primera iz ove lekcije:

```
new Promise(function (resolve, reject) {
    resolve(4);
}).then(function (value) {

    return new Promise(function(resolve, reject){
        setTimeout(() => resolve (value * value), 2000);
    });

}).then(function (value) {
    console.log("Final value: " + value);
});
```

Primer ilustruje nadovezivanje dva Promisea. Ipak, šta će se dogoditi ukoliko se neki od njih završi neuspešno? Na ovo pitanje je vrlo lako odgovoriti:

```
new Promise(function (resolve, reject) {
    reject("Unknown error.");
}).then(function (value) {

    return new Promise(function (resolve, reject) {
        setTimeout(() => resolve(value * value), 2000);
    });

}).then(function (value) {
    console.log("Final value: " + value);
});
```

Sada se logika početnog Promise objekta završava neuspešno. Ipak, s obzirom na to da nigde nije definisana logika koja će konzumirati takvu grešku, dolazi do pojave neobrađenog izuzetka:

```
Uncaught (in promise) Unknown error.
```

Konzumiranje greške se prilikom nadovezivanja Promisea može obaviti veoma lako. Dovoljno je definisati poziv `catch()` metode na kraju lanca nadovezivanja:

```
new Promise(function (resolve, reject) {
    reject("Unknown error.");
}).then(function (value) {

    return new Promise(function (resolve, reject) {
        setTimeout(() => resolve(value * value), 2000);
    });

}).then(function (value) {
    console.log("Final value: " + value);
}).catch(function(error){
    console.log(error);
});
```

Na kraj lanca nadovezivanja sada je dodat poziv metode `catch()`, čija će callback funkcija obraditi pojavu greške unutar Promisea. Bitno je znati da je u ovakvim situacijama, odnosno u situacijama koje podrazumevaju nadovezivanje Promisea, dovoljno definisati samo jedan poziv `catch()` metode. Ona će konzumirati izuzetak koji se eventualno dogodi unutar bilo kog Promisea u lancu:

```
new Promise(function (resolve, reject) {
    resolve(4);
}).then(function (value) {

    return new Promise(function (resolve, reject) {
        setTimeout(() => resolve(value * value), 2000);
        reject("Unknown error.");
    });

}).then(function (value) {
    console.log("Final value: " + value);
}).catch(function(error){
    console.log(error);
});
```

Sada je obavljeno odbijanje drugog Promise objekta, tako što je poziv metode `reject()` premešten unutar njegovog bloka. Ipak, efekat će biti identičan, odnosno i u ovakvoj situaciji `catch()` metoda će konzumirati nastali izuzetak.

Na kraju, `catch()` metodom, pored izuzetaka unutar Promisea, obrađuju se i oni koji se mogu pojaviti unutar callback funkcija:

```
new Promise(function (resolve, reject) {
    resolve(4);
}).then(function (value) {

    return new Promise(function (resolve, reject) {
        setTimeout(() => resolve(value * value), 2000);
    });

}).then(function (value) {

    throw new Error("Unknown error.");
    console.log("Final value: " + value);

}).catch(function(error){
    console.log(error);
});
```

Sada se unutar callback funkcije poslednje `then()` metode obavlja emitovanje izuzetka. Iako nije reč o klasičnom odbijanju Promisea, i na ovaj način se kontrola izvršavanja prebacuje na callback funkciju `catch()` metode.

Primer – korišćenje ugrađenih funkcionalnosti za čitanje podataka fajla

Na početku priče o Promiseima rečeno je da je njihovo razumevanje veoma važno, zato što većina modernih Web API-ja, odnosno funkcionalnosti koje web pregledači izlažu našem kodu, koriste upravo Promisee. Stoga, korišćenje modernih Web API-ja ujedno podrazumeva i korišćenje Promisea. Takvu situaciju prikazaće sledeći primer, koji će ilustrovati čitanje sadržaja jednog fajla koji se nalazi na udaljenom serveru.

Čitanje sadržaja fajla koji se nalazi na nekoj web putanji može se postići korišćenjem ugrađene metode `fetch()` koju web pregledači nama izlažu na korišćenje. Metoda `fetch()` ima sledeću sintaksu:

```
let promise = fetch(url, [options])
```

Može se videti da `fetch()` metoda može da prihvati dva parametra, od čega je samo prvi parametar obavezan:

- `url` – putanja na kojoj se nalazi fajl;
- `options` – parametri za konfigurisanje zahteva koji se upućuje.

Ukoliko se drugi parametar izostavi, metoda `fetch()` se koristi za upućivanje jednostavnog GET zahteva. S obzirom na to da mi želimo da pročitamo sadržaj jednog fajla, u nastavku će ova metoda biti korišćena bez `options` parametra.

Ono što je posebno važno da vidite jeste činjenica da metoda `fetch()` kao svoju povratnu vrednost emituje Promise objekat. Stoga će pozivanje ove metode izgledati ovako:

```
fetch('text.txt').then(function (response) {
      return response.text();
});
```

Nad Promise objektom koji emituje metoda `fetch()` poziva se metoda `then()` kako bi se obradio odgovor dobijen od ove metode. Promise se ispunjava odmah nakon što udaljeni HTTP server pošalje zaglavlje odgovora. Drugim rečima, ne čeka se na kompletan sadržaj, već samo na zaglavlje. Zbog toga je korišćenje metode `fetch()` uglavnom operacija koja se sastoji iz dva dela. Unutar callback metode prvog Promisea dobija se `Response` objekat. Nad njim je u primeru pozvana metoda `text()` kako bi, pored zaglavlja, bio dobijen i tekst, koji se nalazi unutar fajla na URL adresi. Metoda `text()` kao svoju povratnu vrednost opet ima Promise objekat. Stoga, ovo omogućava da se obavi nadovezivanje Promisea:

```
fetch('text.txt').then(function (response) {
      return response.text();
}).then(function (text) {
      textArea.innerHTML = text;
});
```

Sada, unutar callback metode narednog Promise objekta, dolazi se do pročitanog teksta. Naravno, neophodno je obraditi i slučaj u kome može doći do greške:

```
fetch('text.txt').then(function (response) {
      return response.text();
}).then(function (text) {
      textArea.innerHTML = text;
}).catch(function (err) {
      textArea.innerHTML = 'Fetch problem: ' + err.message;
});
```

Bitno je znati da će inicijalni Promise biti odbijen samo ukoliko web pregledač nije u stanju da uputi HTTP zahtev, na primer, zbog mrežnih problema. Ukoliko dođe do neke druge greške, Promise neće biti odbijen. Stoga, ukoliko, na primer, traženi fajl ne postoji, neće doći do odbijanja Promisea. Kako bismo obradili i ovakve situacije, biće dodata sledeća logika:

```
      fetch('text.txt').then(function (response) {
            if(response.status !== 200){
                  throw Error("Error while reading file.");
            }
            return response.text();
      }).then(function (text) {
            textArea.innerHTML = text;
      }).catch(function (err) {
            textArea.innerHTML = 'Fetch problem: ' + err.message;
      });
```

Sada je dodat kod za proveru statusa odgovora. Za bilo koji status koji se razlikuje od 200 (OK), kreira se i izbacuje izuzetak sa odgovarajućom porukom.

U našem primeru, čitanje sadržaja fajla obavljaće se klikom na jedan `button` element. Takođe, pročitani tekst biće smeštan unutar jednog `textarea` elementa. Na kraju, primer će podrazumevati i prikaz loadera tokom trajanja učitavanja podataka, pa pored pozivanja već prikazanih metoda za konzumiranje Promisea, u finalnom primeru biće dodat i poziv metode `finally()`.

Kod kompletnog primera izgleda ovako:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1.0">
        <title>Promise finally example</title>
        <style>
            #loader {
                display: inline-block;
                width: 18px;
                height: 18px;
                display: none;
            }
            #loader:after {
                content: " ";
                display: block;
                width: 18px;
                height: 18px;
                margin: 2px;
                border-radius: 50%;
                border: 2px solid #241f1f;
                border-color: #241f1f transparent #241f1f transparent;
                animation: lds-dual-ring 1.2s linear infinite;
            }
            @keyframes lds-dual-ring {
                0% {
                    transform: rotate(0deg);
                }
                100% {
                    transform: rotate(360deg);
                }
            }
            #my-text-area {
                display: block;
                width: 100%;
                margin-top: 16px;
            }
```

```
        </style>
    </head>
    <body>
        <button id="get-text-btn">Get Data</button>
        <div id="loader"></div>
        <textarea id="my-text-area" rows="30"></textarea>
        <script>
            let button = document.getElementById("get-text-btn");
            let textArea = document.getElementById("my-text-area");
            let loader = document.getElementById("loader");
            button.addEventListener("click", function() {
                loader.style.display = "inline-block";
                fetch('https://v-dresevic.github.io/Advanced-JavaScript-
Programming/data/text1.txt').then(function (response) {
                    if(response.status !== 200){
                        throw Error("Error while reading file.");
                    }
                    return response.text();
                }).then(function (text) {
                    textArea.innerHTML = text;
                }).catch(function (err) {
                    textArea.innerHTML = 'Fetch problem: ' + err.message;
                }).finally(function(){
                    loader.style.display = "none";
                });
            });
        </script>
    </body>
</html>
```

Rezime

- metode za konzumiranje Promisea `then()`, `catch()` i `finally()` kao svoju povratnu vrednost emituju novi Promise objekat;
- činjenica da metode za konzumiranje Promisea kao svoju povratnu vrednost emituju novi Promise objekat omogućava da se postigne nadovezivanje Promisea;
- prilikom nadovezivanja Promisea, povratna vrednost callback metode prethodnog Promisea automatski se prosleđuje callback metodi narednog Promisea u lancu;
- prilikom emitovanja prostih tipova i objekata iz callback metoda Promisea, takva povratna vrednost pakuje se automatski unutar jednog Promise objekta i emituje kao povratna vrednost;
- nadovezivanje asinhronih operacija podrazumeva emitovanje Promise objekta iz callback funkcije jednog Promisea; u takvoj situaciji, naredna metoda za konzumaciju Promisea čeka na završetak Promisea koji je eksplicitno emitovan kao povratna vrednost;
- emitovanje neuspešnog završetka operacije unutar Promisea postiže se pozivanjem metode čiju referencu čuva drugi parametar funkcije egzekutor;
- Promise unutar čije logike dođe do greške ispunjava se i dobija status *rejected*
- Promise dobija status *rejected* čak i onda kada unutar njegove logike dođe do emitovanja izuzetka;
- *rejected* Promise se može konzumirati drugim parametrom `then()` metode ili korišćenjem metode `catch()`
- prilikom nadovezivanja Promisea, dovoljno je navesti jedan poziv `catch()` metode, koja će se aktivirati kada dođe do greške u bilo kom Promise objektu ili callback funkciji.