async i await

Kako bi se asinhrono programiranje dodatno uprostilo, specifikacijom ECMAScript 2017 u jezik je uvršćen još jedan skup funkcionalnosti. Reč je o ključnim rečima `async` i `await`, koje dodatno pojednostavljuju rad sa Promiseima. Lekcija pred vama biće posvećena ovim ključnim rečima i njihovom korišćenju za realizaciju asinhronog programiranja u JavaScriptu.

async funkcije

Priču o ključnim rečima `async` i `await` započećemo ključnom rečju `async`. U pitanju je ključna reč koja se postavlja ispred deklaracije funkcije, čime se takva funkcija pretvara u *async funkciju*.

Prva funkcija koja će biti pretvorena u *async funkciju* biće vrlo jednostavna:

```
function sayHello(){
    return "Hello World";
}
```

Funkcija `sayHello()` nema ulaznih parametara i emituje tekst *Hello World* kao svoju povratnu vrednost. Stoga, ukoliko je pozovemo, njena povratna vrednost će biti `string` *Hello World*:

```
sayHello(); //Hello World
```

Postavljanjem ključne reči `async` ispred ovakve funkcije, ona postaje *async funkcija*:

```
async function sayHello(){
    return "Hello World";
}
```

Upravo prikazana funkcija jeste prva async funkcija koju smo kreirali u ovoj lekciji. Ipak, po čemu se ona razlikuje od identične funkcije koja ispred svoje deklaracije nema ključnu reč `async`? Odgovor na ovo pitanje je vrlo lako dati:

```
async function sayHello(){
    return "Hello World";
}

console.log(sayHello());
```

Sada je async funkcija `sayHello()` pozvana i njena povratna vrednost ispisana unutar konzole:

```
Promise {<resolved>: "Hello World"}
```

Na osnovu ispisa unutar konzole, lako se može uvideti osnovna razlika između *običnih* i *async funkcija* – **async funkcije kao svoju povratnu vrednost emituju Promise objekat**.

Async funkcije kao svoju povratnu vrednost uvek emituju Promise objekat. Povratna vrednost koja se iz takvih funkcija emituje korišćenjem ključne reči `return` pakuje se unutar Promise objekta, odnosno postaje rezultat jednog razrešenog (*engl. resolved*) Promisea.

Kako bi se konzumirao rezultat neke async funkcije, s obzirom na činjenicu da one povratnu vrednost pakuju unutar Promise objekta, dovoljno je uraditi sledeće:

```
async function sayHello() {
    return "Hello World";
}

sayHello().then(function(result) {
    console.log(result);
});
```

Nad povratnom vrednošću funkcije `sayHello()`, sada je pozvana nama dobro poznata metoda `then()`, te je na taj način obavljena konzumacija rezultata Promisea. Unutar konzole se dobija:

```
Hello World
```

Ukoliko se u primer uključe i Arrow funkcije, konzumiranje async funkcije se može obaviti još kompaktnije:

```
sayHello().then((result) => console.log(result));
```

Async funkcije su upravo ono što i njihov naziv kaže – **asinhrone funkcije**. To praktično znači da njihov poziv ne blokira izvršavanje koda koji se nalazi nakon njihovog poziva:

```
async function sayHello() {
    return "Hello World";
}

sayHello().then((result) => console.log(result));

console.log("some other code");
```

Sada je nakon poziva asinhrone funkcije postavljena i jedna dodatna naredba. Unutar konzole se dobija:

```
some other code
Hello World
```

Jasno se može videti da se prvo izvršava naredba koja je definisana nakon poziva async funkcije. Ovo je moguće zato što se callback koji je pridružen završetku async funkcije izvršava tek kada je stek poziva potpuno slobodan.

Pitanje

async funkcije kreiraju se upotrebom ključne reči:

- **async**
- await
- function
- static

Objašnjenje:

async je ključna reč koja se postavlja ispred deklaracije funkcije, čime se takva funkcija pretvara u async funkciju.

Ključna reč await

U nekim situacijama može se javiti potreba da se unutar async funkcija pokrene neka druga, asinhrona operacija i da se tom prilikom sačeka na rezultat njenog izvršavanja. U takvim situacijama se može upotrebiti ključna reč `await`.

Dakle, `await` je ključna reč koja se postavlja ispred poziva funkcija koje kao svoju povratnu vrednost emituju Promise objekat. Upotreba ključne reči `await` eksplicitno govori JavaScript kodu da čeka na završetak funkcionalnosti koja je definisana unutar Promisea.

Ključna reč `await` može se naći isključivo unutar async funkcije i ona zamrzava izvršavanje takve funkcije sve dok se asinhrona logika nakon ključne reči `await` na završi. Ipak, bitno je znati da ključna reč `await` stopira samo izvršavanje koda unutar async funkcije, dok se sav ostali kod nesmetano izvršava.

Efekat ključne reči `await` biće ilustrovan na sledećem primeru: zamislite da nešto ranije prikazana `sayHello()` funkcija mora da prvo dobavi ime nekog korisnika, a tek onda da mu kaže, na primer, *Hello Ben*. Dobijanje imena može biti primer vremenski zahtevne operacije ukoliko se takvo ime, na primer, dobija preko mreže. Prvo će biti prikazana funkcija za dobijanje imena:

```
function getName() {

    return new Promise((resolve, reject) => {
        setTimeout(() => resolve("Ben"), 3000);
    });

}
```

Funkcija `getName()` simulira vremenski zahtevnu operaciju – njoj je potrebno tri sekunde kako bi došla do imena nekog korisnika. Ukoliko bi se koristili pristupi koje smo dosad naučili, mogli bismo da napišemo nešto ovako:

```
async function sayHello() {

    let name;

    getName().then(function(result){
        name = result;
    });
```

```
    return "Hello " + name;

}

sayHello().then((result) => console.log(result));
```

Unutar async funkcije `sayHello()` sada je dodat kod za konzumiranje Promisea, koji emituje funkciju za čitanje imena. Kao što ste mogli videti, ona će nakon tri sekunde doći do traženog imena, pa će se aktivirati i callback funkcija metode `then()`. Ipak, kada se ovakav primer pokrene, unutar konzole se dobija:

```
Hello undefined
```

Ukoliko malo bolje proučite prikazani kod, lako se može razumeti da je dobijen rezultat potpuno očekivan. Naime, unutar async funkcije definisan je kod koji konzumira rezultat jednog Promisea. Iz prethodnih lekcija je poznato da se konzumiranje rezultata Promisea obavlja asinhrono. Stoga se ne čeka na dobijanje rezultata ovakvog Promisea i odmah se prelazi na emitovanje povratne vrednosti. Upravo zbog toga promenljiva `name` ima vrednost `undefined`, pa se unutar konzole dobija tekst *Hello undefined*.

Kako bi se JavaScriptu stavilo do znanja da treba da čeka na završetak neke vremenski zahtevne operacije, unutar async funkcija koristi se ključna reč `await`. Tako je prikazani primer moguće konvertovati na sledeći način:

```
function getName() {

    return new Promise((resolve, reject) => {
        setTimeout(() => resolve("Ben"), 3000);
    });

}

async function sayHello() {

    let name;
    name = await getName();
    return "Hello " + name;

}

sayHello().then((result) => console.log(result));
```

Sada je unutar async funkcije iskorišćena ključna reč `await` i to ispred poziva funkcije koja kao svoju povratnu vrednost vraća Promise objekat. Na ovaj način će se logika funkcije `sayHello()` stopirati sve dok se Promise ne razreši. Stoga će unutar konzole biti ispisan sledeći tekst:

```
Hello Ben
```

Bitno je razumeti da ključna reč `await` stopira logiku async funkcije, ali ne i bilo kog drugog koda koji se nalazi u kontekstu iz koga se takva funkcija poziva:

```
function getName() {

    return new Promise((resolve, reject) => {
        setTimeout(() => resolve("Ben"), 3000);
    });

}

async function sayHello() {

    let name;
    name = await getName();
    return "Hello " + name;

}

sayHello().then((result) => console.log(result));

console.log("some other code...");
```

Sada, nakon pozivanja async funkcije, postavljena je još jedna naredba. Unutar konzole se dobija:

```
some other code...
Hello Ben
```

Jasno se može videti da se naredbe nakon poziva async funkcije izvršavaju nesmetano. Drugim rečima, one ne čekaju na završetak async funkcije.

Obrada izuzetaka prilikom korišćenja async/await pristupa

Potpuno je realno očekivati da unutar async funkcije može doći do pojave greške. U takvoj situaciji Promise, koji se implicitno stvara kao povratna vrednost takvih funkcija, dobija stanje *rejected*. Stoga je pojavu greške moguće obraditi korišćenjem pristupa koji su ilustrovani u prethodnim lekcijama:

```
async function sayHello() {
      throw new Error("Unknown error.");
      return "Hello World";
}

sayHello().then((result) => console.log(result), (error) =>
console.log(error));
```

Sada su prilikom pozivanja async funkcije definisane dve callback funkcije kao parametri `then()` metode. Druga će se aktivirati u slučaju pojave izuzetka ili greške unutar async funkcije:

```
Error: Unknown error.
```

Identično je bilo moguće postići i korišćenjem metode `catch()`, u slučaju da smo zainteresovani samo za dojavu o izuzetku:

```
async function sayHello() {
    throw new Error("Unknown error.");
    return "Hello World";
}

sayHello().catch((error) => console.log(error));
```

Efekat oba pristupa je identičan.

Segment u kome se `async/await` pristup posebno razlikuje od direktnog korišćenja Promisea jeste obrada grešaka unutar samih async funkcija prilikom korišćenja ključne reči `await`. S obzirom na to da ključna reč `await` zapravo omogućava da se asinhroni kod piše kao sinhroni, takva osobina se odnosi i na obradu izuzetaka. Drugim rečima, obrada izuzetaka koji mogu nastati unutar funkcija na čiji rezultat se čeka upotrebom ključne reči `await` može se postići na tradicionalni način, korišćenjem `try...catch` blokova:

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

async function sayHello() {
    let name;

    try {
        name = await getName();
    } catch(e){
        return e;
    }

    return "Hello " + name;
}

sayHello().then((result) => console.log(result));
```

Prikazani kod predstavlja modifikovani već prikazani primer. Sada se Promise unutar `getName()` metode nakon tri sekunde razrešava neuspehom. Takvo odbijanje Promisea obrađuje se unutar async funkcije korišćenjem `try` i `catch` blokova.

Primer – transformisanje primera za čitanje sadržaja fajla upotrebom ključnih reči async/await

U prethodnoj lekciji je prikazan primer čitanja sadržaja jednog fajla koji se nalazi na udaljenom serveru, korišćenjem ugrađene metode `fetch()`. Podsetimo se kako je izgledala osnovna logika takvog primera:

```
      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";
      });
```

Prikazani kod ilustruje klasičan primer nadovezivanja Promisea. Ovakav scenario se veoma lako može zameniti upotrebom ključne reči `await` unutar jedne async funkcije. Za početak, biće kreirana takva async funkcija:

```
async function getData() {
      ...
}
```

Unutar ovakve funkcije sada će biti dodata osnovna logika za čitanje sadržaja fajla sa udaljenog servera:

```
async function getData() {
    let response = await fetch('text.txt');
    let text = await response.text();
    textArea.innerHTML = text;
}
```

Ukoliko se sećate prethodne lekcije, čitanje sadržaja nekog fajla korišćenjem `fetch()` metode je operacija koja se sastoji iz dva dela. Samo pozivanje `fetch()` metode emituje Promise koji se razrešava `Response` objektom. Takav `Response` objekat u sebi ima osnovne informacije iz zaglavlja odgovara. Ipak, tekstualni sadržaj se dobija pozivanjem metode `text()`. Opet je reč o metodi koja emituje Promise, koji se razrešava isporukom tekstualnog sadržaja fajla na definisanoj putanji. Zbog svega toga, u prikazanom primeru ispred poziva metoda `fetch()` i `text()` navodi se ključna reč `await`. Na taj način će se eksplicitno čekati na razrešenje oba Promisea koji se na ovaj način dobijaju.

Iz prikazanog je moguće videti da ključne reči `async` i `await` omogućavaju pretvaranje lanaca nadovezivanja `then()` metoda u jedan oblik koji više liči na sinhroni kod. Naime, više nema nadovezivanja i callback funkcija. Naredbe se navode jedna za drugom i izvršavaju sekvencijalno.

U primeru je neophodno realizovati i obradu eventualnih izuzetaka:

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

Sada su pozivi asinhronih funkcija na koje se čeka smešteni unutar `try` bloka i dodat je pripadajući `catch` blok za obradu eventualnih izuzetaka, do kojih može doći prilikom izvršavanja ugrađenih `fetch()` i `text()` metoda. Iz prethodne lekcije je poznato da će se `fetch()` metoda razrešiti odbačenim Promise objektom samo ukoliko nije moguće uputiti HTTP zahtev. Stoga će na taj način unutar `catch` bloka biti obrađeni slučajevi koji podrazumevaju probleme sa mrežnom konekcijom i slično. Pored toga, catch blok će se aktivirati i u ostalim slučajevima, poput nepostojećeg fajla, greške na serveru i slično. To će se dogoditi zato što je u kodu dodat uslovni blok, kojim se proverava da li je server odgovorio nekim kodom različitim od 200. Ukoliko nije, kreira se i izbacuje izuzetak, koji će takođe biti obrađen od strane `catch` bloka.

Kao i u prethodnoj lekciji, logika za učitavanje sadržaja fajla aktiviraće se prilikom klika na `button` element, a rezultat će biti smešten unutar jednog `textarea` elementa. Na kraju, tokom učitavanja fajla, biće prikazan loader.

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 () {
                getData();
            });
            async function getData() {
                try {
                    loader.style.display = "inline-block";
```

```
                    let response = await fetch('https://v-
dresevic.github.io/Advanced-JavaScript-Programming/data/text.txt');
                    if (response.status !== 200) {
                        throw new Error("Error while reading file.");
                    }
                    let text = await response.text();
                    textArea.innerHTML = text;
                } catch (err) {
                    textArea.innerHTML = 'Fetch problem: ' + err.message;
                } finally {
                    loader.style.display = "none";
                }
            }
        </script>
    </body>
</html>
```

Rezime

- specifikacijom ECMAScript 2017 u jezik su uvršćene ključne reči `async` i `await`, koje dodatno pojednostavljuju rad sa Promiseima;
- `async` je ključna reč koja se postavlja ispred deklaracije funkcije, čime se takva funkcija pretvara u async funkciju;
- async funkcije kao svoju povratnu vrednost emituju Promise objekat;
- povratna vrednost koja se iz async funkcija emituje korišćenjem ključne reči `return` pakuje se unutar Promise objekta, odnosno postaje rezultat jednog razrešenog Promisea;
- async funkcije su asinhrone funkcije, odnosno njihov poziv ne blokira izvršavanje narednih naredbi;
- `await` je ključna reč koja se postavlja ispred poziva funkcija koje kao svoju povratnu vrednost emituju Promise objekat i eksplicitno govori JavaScript kodu da čeka na završetak funkcionalnosti koja je definisana unutar Promisea;
- ključna reč `await` može se naći isključivo unutar async funkcija;
- ključna reč `await` stopira logiku async funkcije, ali ne i bilo kog drugog koda koji se nalazi u kontekstu iz koga se takva funkcija poziva;
- kada unutar async funkcije dođe do greške, izuzetak se pakuje unutar *rejected* Promisea i emituje kao povratna vrednost;
- greške async funkcija je moguće obraditi korišćenjem `then()` i `catch()` metoda, izvan takvih funkcija;
- unutar async funkcija izuzetke je moguće obrađivati upotrebom regularnih `try` i `catch` blokova;
- ključna reč `await` omogućava da se asinhronim kodom rukuje kao da je sinhroni.