



# JavaScript

Dvanaesti dio



# Pregled

- Promises
  - Kreiranje
  - Primjeri
  - Prototype metodi
    - Then
    - Catch
    - Finally
  - Metodi
    - Resolve
    - Reject
    - All
    - Race



# Obnavljanje i napomene

- Test 3
- Domaci 4 i 5
- JSON
  - Parse
  - stringify
- Promise
  - Čemu služe, zašto ih uvodimo
  - States



# Promises, kreiranje

- Za kreiranje **Promise** objekta koristi se **new** keyword i konstruktor. Već smo pomenuli, ovaj konstruktor uzima kao argument funkciju, tkz. **executor** function. Ova funkcija ima dva parametra, a oba su funkcije. Prva (**resolve**) se poziva kada se asinhrona operacija izvrši uspješno i vrati rezultat kao vrijednost. Druga (**reject**) se poziva kada se zadatak ne izvrši uspješno, a vraća razlog zašto zadatak nije izvršen, što je najčešće error objekat.
- Da obezbijedite funkciju koja ima funkcionalnosti Promise objekta, jednostavno iz funkcije vratite Promise objekat

```
const myFirstPromise = new Promise((resolve, reject) => {  
  // do something asynchronous which eventually calls either:  
  //  
  //   resolve(someValue); // fulfilled  
  // or  
  //   reject("failure reason"); // rejected  
});
```

```
function myAsyncFunction(url) {  
  return new Promise((resolve, reject) => {  
    const xhr = new XMLHttpRequest();  
    xhr.open("GET", url);  
    xhr.onload = () => resolve(xhr.responseText);  
    xhr.onerror = () => reject(xhr.statusText);  
    xhr.send();  
  });  
}
```



# Promises, svojstva i metodi

- Svojstva (properties)
  - **length** - uvijek 1 (broj argumenata promise konstruktora)
  - **prototype** - predstavlja prototype Promise konstruktora
    - Svi objekti imaju property koji se zove **prototype**, a o prototype ćemo puno detaljnije govoriti u dijelu za OOP
    - Koristi sledeće metode:
      - then()
      - catch()
      - finally()
- Metode
  - Promise.all(iterable)
  - Promise.race(iterable)
  - Promise.reject(reason)
  - Promise.resolve(value)



# Promise, primjer

```
let myFirstPromise = new Promise((resolve, reject) => {
  // We call resolve(...) when what we were doing asynchronously was successful, and reject(...) when it
  // In this example, we use setTimeout(...) to simulate async code.
  // In reality, you will probably be using something like XHR or an HTML5 API.
  setTimeout(function(){
    resolve("Success!"); // Yay! Everything went well!
  }, 250);
});

myFirstPromise.then((successMessage) => {
  // successMessage is whatever we passed in the resolve(...) function above.
  // It doesn't have to be a string, but if it is only a success message, it probably will be.
  console.log("Yay! " + successMessage);
});
```



# Promises, primjer-1

- Metod `insertAdjacentHTML()` dodaje tekst kao HTML na određenu poziciju
  - Poziva se nad html elementom na osnovu koga se definiše pozicija umetanja HTMLa.
  - Sintaksa `node.insertAdjacentHTML(position, text)`
  - Dozvoljene pozicije:
    - `afterbegin`
      - Dodaje se kao first child u node
    - `afterend`
      - Dodaje se kao next sibling za node
    - `beforebegin`
      - Dodaje se kao previous sibling za node
    - `beforeend`
      - Dodaje se kao last child za node
  - Tekst je html kod
- Pogledajmo primjer (code-1)



## Promise, primjer-2

- Tip objekta Blob() predstavlja raw podatke, najčešće se koristi za fajlove
  - [Pogledajte](#) i ovo
- Metod URL.createObjectURL(object)
  - Object može da bude File, Blob ili MediaSource
  - Vraće DOMString (UTF-16 string) sadrži URL objekta koji može biti iskorišćen kao referenca ka sadržaju source objekta object
  - [Pogledajte još detalja](#)
- Pogledajmo još jedan primjer (code-2)





# Prototype, uvod

- Čuva predefinisana svojstva, uključujući i metode, određenog tipa podatka
  - Pogledajmo npr. `Array.prototype`
    - Kako da provjerimo za stringove i promise prototype
- Koristi se često kod nasljeđivanja
- Srž funkcionisanja OOP kod JSa
- Mnogo detaljnije ćemo govoriti o prototype kada budemo radili OOP



# Promise, prototype metode

- Tri metoda
  - **.then**(onFulfilled, onRejected) - povezuje fulfillment i rejection handlers na promise, i vraće novi promise koji se resolve-uje u povratnu vrijednost callback-a, ili u originalnu sređenu vrijednost ako promise nije obrađen (npr. ako onFulfillment ili onRejection nije funkcija)
  - **.catch**(onRejected) - povezuje callback **onRejected** handler na promise i vraće novi promise koji se resolve-uje u vrijednost koju vrati onReject callback ako je pozvan, ili u originalnu ostvarenu (fulfillment) vrijednost ako je promise u fulfilled
  - **.finally**(onFinally) - povezuje onFinally handler i vraće novi promise koji se resolve-uje kada se originalni promise resolve-uje. Handler se pozove kada se promise sredi u fulfilled ili rejected



# Promise.prototype.then()

- Metod `.then()` vraće Promise. Sadrži do dva argumenta: callback funkcije za success ili failure slučajeve Promise-a

```
var promise1 = new Promise(function(resolve, reject) {  
    resolve('Success!');  
});
```

```
promise1.then(function(value) {  
    console.log(value);  
    // expected output: "Success!"  
});
```

- Ako se oba argumenta izostave, ili proslijedi nešto što nije callback, metod `then()` neće imati handler-e ali neće vratiti grešku. Ako Promise koji se pozove sa `razriješi` sa `then()` koji nema handler-e, kreiraće se novi Promise bez handlera sa finalnim stanjem Promise-a nad kojim je pozvan `then()` metod



# Promise.prototype.then(), nastavak(1)

- Sintaksa
  - `p.then(onFulfilled [, onRejected])`
    - Parametri
      - `onFulfilled` - funkcija koja se poziva ako je promise fulfilled. Funkcija ima jedan argument, fulfillment value. Ako nije funkcija, interno se prevede u **Identity** funkciju (vrati ono što primi kao argument)
      - `onRejected` - funkcija koja se poziva ako se promise neuspješno izvrši. Funkcija ima jedan argument, rejection razlog. Ako nije funkcija, interno se prevede u **Identity** funkciju (vrati ono što primi kao argument)



## Promise.prototype.then(), nastavan(2)

- Vraće:
  - Promise sa statusom pending. Onda se handler funkcija (onFulfilled ili onRejected) pozove asinhrono (čim se call stack isprazni). Nakon što se handler funkcija pozove, ako je handler funkcija:
    - Vraća vrijednost ako je promise vraćen sa then() razriješen (resolved) sa return vrijednošću koja je vrijednost
    - Ne vrati ništa ako je promise vraćen sa then() razriješen sa undefined vrijednošću
    - Vraća grešku ako je promise vraćen sa then() odbijen (rejected) sa thrown error kao vrijednošću
    - Vraća razriješen/odbijen promise, ako je promise vraćen sa then() razriješen/odbijen sa promise vrijednošću
    - Vraća još jedan pending promise object, ako će promise biti razriješen/odbijen, vraćen sa then(), kasnije sa razriješenim/odbijenim promisom vraćenim sa handlerom. Takođe, vrijednost promise-a vraćena sa then() biće ista kao vrijednost promise-a vraćena od hanldera



# Promise.prototype.then(), primjeri

- Pogledajmo primjere (code-3)



# Promise.prototype.catch()

- Metod `catch()` vraće Promise i samo se bavi sa rejected stanjem. Ponaša se kao `Promise.prototype.then(undefined, onRejected)` jer u suštini `Promise.prototype.catch(onRejected)` interno poziva `Promise.prototype.then(undefined, onRejected)`. Ovo znači da treba proslijediti `onRejected` funkciju iako želimo fallback na *undefined* rezultujuću vrijednost, npr. `obj.catch(() => {})`

```
var promise1 = new Promise(function(resolve, reject) {  
  throw 'Uh-oh!';  
});  
  
promise1.catch(function(error) {  
  console.log(error);  
});  
// expected output: Uh-oh!
```



# Promise.prototype.catch(), nastavak(1)

- Sintaksa
  - `p.catch(onRejected)`
    - Parametri
      - `onRejected` - funkcija koja se poziva kada je Promise odbijen. Ova funkcija ima jedan argument i to `reason(rejection reason)`. Promise vraćen sa `catch()` je rejected ako `onRejected` baca grešku ili vraće promise koji je sam po sebi rejected, inače vraće resloved
    - Vraće
      - Interno poziva `Promise.prototype.then()` na objekat nad kim je pozvana, proslijedi argumente `undefined` i `onRejected`, a onda vrati vrijednost tog poziva (vidjeli smo prije šta `then()` metod vraće)





# Promises.prototype.catch(), primjeri

- Pogledajmo primjere (code-4)



# Promises.prototype.finally()

- Vraće Promise. Kada se riješi promise, bez obzira da li je to fulfilled ili rejected, određena callback funkcija se pozove. Ovo omogućava da kod mora da se izvrši kada se Promise riješi. Ovim je omogućeno izbjegavanje dupliranja koda u then() ili catch()
- Sintaksa
  - `p.finally(onFinally)`
    - Parametri
      - Funkcija koja se poziva kada se riješi(resolved ili rejected) promise
    - Vraće
      - Promise čiji finally handler je setovan na specifičnu funkciju onFinally



## Promise.prototype.finally(), nastavak

- Ovaj metod može biti od koristi ako je potrebna neka obrada nakon što se promise riješi, bez obzira na ishod
- Vrlo slično kao da smo odradili .then(onFinally, onFinally) sa sledećim razlikama
  - Ne morate dva puta pozivati istu funkciju kod .finally
  - Finally callback ne prihvata nijedan argument, jer se ne zna pouzdano u koje stanje će promise da završi. Ova slučaj je dobar kada nas ne zanima rejection razlog ili fulfillment vrijednost
  - `Promise.resolve(2).then(() => {}, () =>{})` vraće `undefined`, `Promise.resolve(2).finally(() => {})` vraće `2`
  - Slično za `reject`
  - Napomena: **throw** u finally callback-u će odbiti novi promise sa porukom definisanom u `throw`



# Promise.prototype.finally(), primjer

- Pogledajmo primjer (code-5)
- Zanimljivo isprobati za vježbu
  - [JSON placeholder](#)
  - [JSON server](#)
  - Pogledajmo ove podatke
    - [Link](#)
    - Za listu APIa (dosta besplatnih, pogledajte [GitHub repo](#))



# Promise.resolve()

- **Promise.resolve(value)** metod vraće Promise objekat koji je razriješen sa zatom vrijednošću value.
- Ako je vrijednost
  - promise, taj promise bude vraćen,
  - ako je vrijednost thenable (podržava metod then()) vraćeni promise će da ‘prati’ thenable vrijednost prilivši se eventualnom stanju,
  - inače vraćeni promise je fulfilled sa vrijednošću value
- Ne pozivajte Promise.resolve na thenable koji se sam po sebi razrješava. To bi proizvelo beskonačnu rekurziju.

```
var promise1 = Promise.resolve(123);
```

```
promise1.then(function(value) {  
  console.log(value);  
  // expected output: 123  
});
```



# Promise.resolve(), nastavak(1)

- Sintaksa
  - Promise.resolve(value)
  - Promise.resolve(promise)
  - Promise.resolve(thenable)
  - Parametri
    - Value - argument koji se razriješi kao vrijednost, a može biti i promise ili thenable
  - Vraće
    - Promise koji se razriješi u zadatu vrijednost, ili promise proslijeđen kao vrijednost ako je value promise objekat



# Promise.resolve(), primjeri

- Pogledajmo par primjera (code-6)



# Promise.reject()

- Metod Promise.reject(reason) vraće Promise objekat koji je rejected (odbijen) iz određen razloga(reason)

- Sintaksa

- Promise.reject(reason)

- Parametri

- Reason -vraće zašto je promise odbijen

- Vraće

- Promise koji je odbijen iz zadatog razloga

```
function resolved(result) {  
  console.log('Resolved');  
}
```

```
function rejected(result) {  
  console.log(result);  
}
```

```
Promise.reject(new Error('fail')).then(resolved, rejected);  
// expected output: Error: fail
```






## Promise.all() metod

- Metod Promise.all(iterable) metod vraće jedan promise koji se resolvuje kada se svi promisi (iterable) resolvuju ili kada iterable argument ne sadrži promise. U slučaju rejection, vrati se greška prvog promise-a koji se ne izvrši ispravno

```
var promise1 = Promise.resolve(3);
var promise2 = 42;
var promise3 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then(function(values) {
  console.log(values);
});
// expected output: Array [3, 42, "foo"]
```



# Promise.all(), nastavak (1)

- Sintaksa
  - **Promise.all(iterable)**
    - Parametri
      - Iterable - objekat kao što je niz ili string (oni su iterabilni)
    - Return
      - Ako je iterable prazan, već razriješen Promise (resolved), sinhrono
      - Ako iterable ne sadrži promise, asinhrono razriješen promise
      - U ostalim slučajevima pending Promise. Onda, vraćeni promise se razriješi ili odbije asinhrono (čim call stack postane prazan) kada svi promisi u iterable budu riješeni, ili ako bar jedan ne bude.



# Promise.all(), primjeri

- Pogledajmo par primjera (code-7)



## Promise.race() metod

- Metod Promise.race(iterable) metod vraće promise koji se resolve-uje ili reject-uje čim se jedan od promise-a u iterable resolve-uje ili reject-uje sa vrijednošću ili razlogom tog promise-a.

```
var promise1 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 500, 'one');
});

var promise2 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 100, 'two');
});

Promise.race([promise1, promise2]).then(function(value) {
  console.log(value);
  // Both resolve, but promise2 is faster
});
// expected output: "two"
```



# Promise.race(), nastavak(1)

- Sintaksa
  - Promise.race(iterable)
    - Parametri
      - Iterable - iterabilni objekat, kao što je niz
    - Vraće
      - Promise u stanju pending koji asinhrono vraće vrijednost prvog promise-a iz iterable koji se riješi bez obzira na ishod



# Promise.race(), primjeri

- Pogledajmo primjere (code-8)



# Promises, kreiranje wrappera oko starog callback API-a

- Promise može biti kreiran od nule primjenom konstruktora.
- U idealnom svijetu, sve asinhronne funkcije trebaju da vrate Promise. Međutim, neki API i dalje očekuju callbacks **success** i/ili **failure**

```
setTimeout(() => saySomething("10 seconds passed"), 10000);
```

- Miješanje old-style callback-a i promise-a je problematično. U primjeru gore, ako se desi greška nad `saySomething()`, nema ništa da je obradi. Međutim, ovo je moguće realizovati kroz wrappovanje u Promise

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));  
  
wait(10000).then(() => saySomething("10 seconds")).catch(failureCallback);
```



Pitanja