JavaScript

Jedanaesti dio

Pregled

- JSON
  - Uvod
  - Sintaksa
  - JSON vs XML
  - Parse
  - Stringify
- Fetch, uvod
- Promises
  - Uvod
  - Motivacija
  - Promise objekat
  - Sintaksa
  - Opis i skica
- Pitanja

Obnavljanje

- Dokle smo stigli, pogledajmo zajedno
  - Link
- Objekat XMLHttpRequest
  - Kreiranje
  - onreadystatechange
- XML, struktura

JSON, uvod

- Gotovo identična uloga kao i XML
  - Glavna uloga čuvanje podataka na razumljiv način
- JSON je tekst zapisan kao JS objekat
- Kada se podaci razmjenjuju između pretraživača i servera, ti podaci moraju da budu tekstualni
  - JSON je tekst, a svaki JS objekat možemo da konvertujemo u JSON i pošaljemo JSON serveru
  - Takođe, bilo koji JSON možemo da konvertujemo u JS objekat
  - Na ovaj način možemo da radimo sa podacima kao JS objektima, bez komplikovanog parsiranja i prevoda
- Sa JSON podacima moguće je:
  - Slanje
  - Primanje
  - Čuvanje
- JSON (JavaScript Object Notation) jednostavan, čitljiv, format za razmjenu podataka
  - Originalno definisan od osnivača JSa (Douglas Crockford)
- Napomena: JSON je tekst, što znači da je nezavisan od programskih jezika
  - Koristi JS sintaksu, ali JSON format je samo tekst

JSON, uvod (1)

- Zašto JSON
  - Kako je JSON format tekst, vrlo se jednostavno šalje do i sa servera
  - Koristi se kao format za čuvanje podataka kod bilo kog programskog jezika
- JS ima ugrađene funkcije koje konvertuju string, napisan u JSON formatu, u JS objekat
  - JSON.parse()
- Kada stignu podaci sa servera, u JSON formatu, ti podaci mogu da se koriste kao bilo koji JS objekat
- JSON sintaksa je podskup JS sintakse

JSON, sintaksa

- Izvedena iz notacije objekata kod JSa
  - Podaci se čuvaju u paru key:value
  - Podaci se odvajaju zarezom
  - {} čuvaju objekat
  - [] čuvaju niz
- Key (name) se postavlja u “”
  - Kod JSa to nije obavezno, kod JSONa jeste !
  - Može li jednostruki navodnik
  - Pogledajmo
- Kod JSON-a, vrijednosti moraju biti jedan od sledećih tipova podataka
  - **String(moraju da koriste double quotes), broj, objekat (JSON objekat), niz, boolean, null**
  - Tipovi kao što su function, date i undefined nisu podržani u JSONu

JSON vs XML

- Kako bi ovo izgledalo u XML formatu

JSON Example

```
{"employees":[
    { "firstName":"John", "lastName":"Doe" },
    { "firstName":"Anna", "lastName":"Smith" },
    { "firstName":"Peter", "lastName":"Jones" }
]}
```

JSON vs XML, nastavak

- JSON je kao XML jer
  - JSON i XML čitljivi za ljude (self descibing)
  - JSON i XML imaju hijerarhiju (values within values)
  - JSON i XML se mogu parsirati i koristiti za više programskih jezika
  - JSON i XML mogu biti preuzeti sa XHR
- Po čemu se JSON razlikuje od XMLa
  - JSON ne koristi end tag
  - JSON je kraći
  - JSON se brže čita i upisuje
  - JSON može da sadrži niz
- Najveća razlika XML mora da bude parsiran pomoću XML parsera dok JSON može biti parsiran standardom (ugrađenom) JS funkcijom. Ova razlika nam jasno govori da je JSON bolji nego XML
- Kod AJAX aplikacija, JSON je brži i jednostavniji od XMLa
  - Kod XMLa: pokupiti XML dokument, primijeniti XML DOM da se prođe kroz XML dokument, izvuci vrijednosti i sačuvaj ih u varijablu
  - Kod JSON-a: pokupiti JSON string, JSON.parse za parsiranje JSON stringa

JSON Parse

- Kada podaci stignu sa server, uvijek stignu u string formatu
- Da bi ste jednostavno pretvorili JSON string u JS objekat koristi se ugrađena JS funkcija
  - JSON.parse()
- Isprobajmo primjer
- **Pogledajmo sada code-4**
  - Kako da štampamo ime trećeg ljubimca?
- Pogledajmo i primjer sa datumom
  - Primjer
- Pogledajmo primjer sa funkcijom
  - Primjer
  - Ovo vrlo opasno koristiti, moguće ali nije dobra praksa
  - Eval metod izvršava JS kod koji je zapisan u vidu stringa

JSON stringify

- Kada šaljemo podatke ka serveru, ti podaci moraju biti zapisani u vidu stringa
  - JSON.stringify() ugrađeni JS metod koji konvertuje JS objekat u string
- Pogledajmo primjer
  - Primjer
  - Šta ako funkciju ne konvertujemo u string
  - Šta ako umjesto funkcije za vrijednost **age** property-a stavimo undefined vrijednost
- Pogledajmo još jedan primjer
  - Primjer
- Za još detalja vezanih za JSON pročitati
  - JSON Objects
  - JSON Arrays
  - JSON PHP
  - JSON HTML
  - JSONP

Fetch

- Fetch API služi kao interfejs za preuzimanje resursa.
- Funkcioniše slično kao XMLHttpRequest, ali novi API obezbjeđuje više opcija, kao i veću fleksibilnost
- Međutim, da biste razumjeli kako funkcioniše fetch, prije toga treba uvesti još pojmova.
- Jedan od ključnih je **Promise**

Promises, uvod

- Objekat koji predstavlja da li je asinhrona operacija završena ili se desila neka greška
- Promise je returned objekat na koji se kači callback, umjesto prosleđivanja callback-a kroz funkciju
- Pretpostavimo da imamo funkciju **createAudioFileAsync()** koja asinhrono generiše fajl za zvuk, sa zadatom konfiguracijom i dvije callback funkcije, jedna se poziva kada se audio fajl uspješno kreira, a druga ako se pojavi greška

```
function successCallback(result) {
  console.log("Audio file ready at URL: " + result);
}

function failureCallback(error) {
  console.log("Error generating audio file: " + error);
}

createAudioFileAsync(audioSettings, successCallback, failureCallback);
```

Promises, uvod (1)

- Međutim, danas se koristi moderaniji način za ovo i to što funkcija vrati tkz. Promise objekat na koji možemo da zakačimo callback funkcije
- Ako prethodnu funkciju prepišemo da vraće Promise objekat, dobijamo sledeće

```
createAudioFileAsync(audioSettings).then(successCallback, failureCallback);
```

- Ovo je skraćena verzija za

```
const promise = createAudioFileAsync(audioSettings);
promise.then(successCallback, failureCallback);
```

Promises, garancije

- Za razliku od prvog načina prosleđivanja, kod modernog načina, imamo neke garancije
  - Callback se nikad neće pozvati prije nego što se završi trenutno izvršavanje JS event loop-a
  - Callbacks dodati sa then() čak i nakon success ili failure asionhrone operacije će biti pozvani kao što je već naglašeno
  - Više callback funkcija može biti dodato pozivanjem then() više puta. Svaki callback se izvršava jedan za drugim u redosledu u kom su dodati
- Jedna od ključnih stvari kod Promise-a je što mogu biti pozvani u chain-u

Promises, chaining

- Često je potrebno izvršiti dvije ili više asinhronih operacija zaredom, što znači da je nam je često potreban rezultat prethodne funkcije koja vraće Promise
- Pogledajmo primjer

```
const promise = doSomething();
const promise2 = promise.then(successCallback, failureCallback);
```

identično

```
const promise2 = doSomething().then(successCallback, failureCallback);
```

- Promjenljiva **promise2** predstavlja ne samo da je završena funkcija doSomething() koja vraće Promise (znamo da vraće Promise jer jedino Promise ima metod then()) već i callback successCallback ili failureCallback koje smo proslijedili, a i one same mogu da budu funkcije koje vraćaju Promise. Kada je ovo slučaj, bilo koji callback dodat u **promise2** stavlja se u red iza Promise-a koji vraće successCallback ili failureCallback

Promises, old style vs modern style

- Pogledajmo kako se pisao nekad kod kada nije bilo Promise objekta

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('Got the final result: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

```
doSomething().then(function(result) {
  return doSomethingElse(result);
})
.then(function(newResult) {
  return doThirdThing(newResult);
})
.then(function(finalResult) {
  console.log('Got the final result: ' + finalResult);
})
.catch(failureCallback);
```

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => {
  console.log(`Got the final result: ${finalResult}`);
})
.catch(failureCallback);
```

Promises, old style vs modern style (nastavak)

- Napomene:
  - **catch**(failureCallback) je skraćena verzija za **then**(null, failureCallback)
  - Uvijek vratite rezultat, inače callback neće uhvatiti rezultat prethodnog Promise-a
- Moguće je posle catch zakačiti then(). Pogledajmo primjer. Šta će da se štampa

```
new Promise((resolve, reject) => {
    console.log('Initial');

    resolve();
})
.then(() => {
    throw new Error('Something failed');

    console.log('Do this');
})
.catch(() => {
    console.log('Do that');
})
.then(() => {
    console.log('Do this, no matter what happened before');
});
```

Promises, error propagation

- Moguće je prekinuti lanac then() poziva. Pogledajmo primjer

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => console.log(`Got the final result: ${finalResult}`))
.catch(failureCallback);
```

nekad

```
try {
  const result = syncDoSomething();
  const newResult = syncDoSomethingElse(result);
  const finalResult = syncDoThirdThing(newResult);
  console.log(`Got the final result: ${finalResult}`);
} catch(error) {
  failureCallback(error);
}
```

Promises, async/await (sintaksa)

- Uvodimo kratko **async/await**

```
async function foo() {
  try {
    const result = await doSomething();
    const newResult = await doSomethingElse(result);
    const finalResult = await doThirdThing(newResult);
    console.log(`Got the final result: ${finalResult}`);
  } catch(error) {
    failureCallback(error);
  }
}
```

Promise objekat

- Predstavlja eventualno završenu (ili failed) asinhronu operaciju i rezultat
- Pogledajmo primjer

```
var promise1 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve('foo');
  }, 300);
});

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

console.log(promise1);
// expected output: [object Promise]
```

Promise, sintaksa

- Pogledajmo sintaksu

```
new Promise( /* executor */ function(resolve, reject) { ... }
```

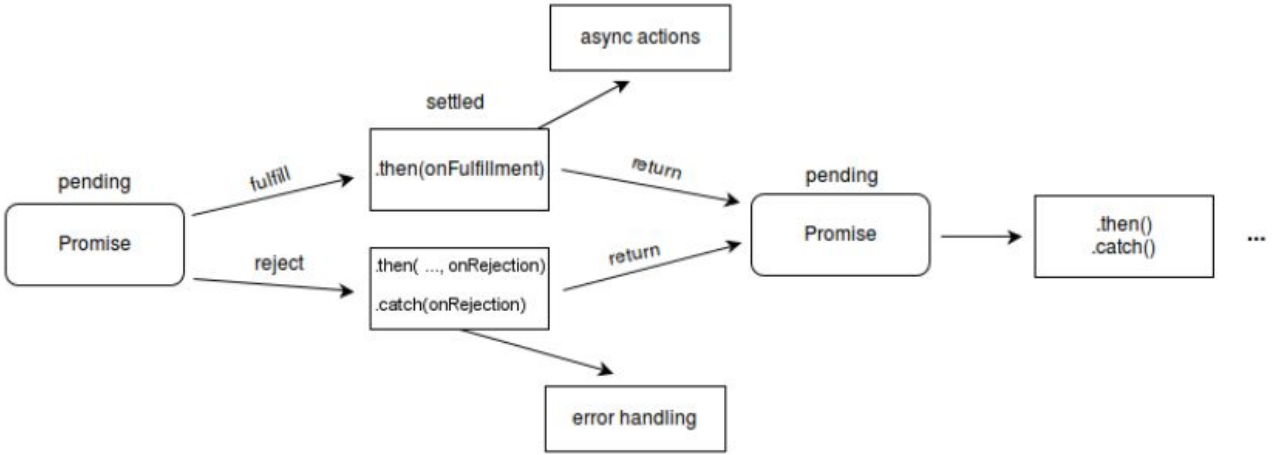
- **executor** predstavlja funkcija koja ima **resolve** i **reject** argumente
  - Ova funkcija se odmah poziva pri kreiranju Promise objekta
  - **executor** funkcija se poziva prije nego Promise konstruktor vrati kreirani objekat
  - Funkcije **resolve** i **reject** funkcije, kada se pozovu, riješe (resolve) ili odbiju (reject) promise
  - Funkcija executor inicira neki asinhroni posao, a onda nakon što se završi, pozove se funkcija resolve da riješi promise ili reject da odbije promise (ako se pojavi greška).
  - Ako se desi greška u executor funkciji, promise se obdija i vrijednost koju vrati ova funkcija postaje ignorisana

Promise, opis

- **Promise** možete posmatrati kao proxy za vrijednosti koje nisu obavezno poznate kada se promise kreira. Omogućava povezivanja handler-a sa asinhronim akcijama koje su eventulano uspješne vrijednosti ili neuspješni razlozi (error message). Ovo omogućava asinhronim metodoma da vraća vrijednosti kao sinhrone metode: umjesto da vraćaju odmah rezultat, asinhrone metode vraćaju promise da bi u budućnosti u određenom momentu obezbijedili vrijednost
- Promise se nalazi u jednom od stanja
    - Pending - inicijalno stanje, operacija nije gotova
    - Fulfilled - operacija je uspješno završena
    - Rejected - operacija je neuspješna
- Stanje pending može da bude ispunjeno (fulfilled) sa vrijednošću ili odbijeno sa razlogom (greška). Kada se jedna od ovih opcija desi odgovarajuću handleri se poziva sa metodom **then()**
    - Ako je promise već fulfilled ili rejected kada se zakači odgovarajući handler, handler se poziva

Promise, skica

- Promise.prototype.then() i Promise.prototype.catch() vraćaju promise, tako da mogu da se povežu u lanac



Promise, detalji

- Microtask queue
- Još jedan članak
- Napredno

Pitanja