Web aplikacije (WA)

Nositelj: doc. dr. sc. Nikola Tanković
Asistent: Luka Blašković, mag. inf.

Ustanova: Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli

(4) Upravljanje podacima na poslužiteljskoj strani

Učinkovita pohrana podataka od presudne je važnosti za osiguravanje visoke kvalitete i pouzdanosti svake web aplikacije. Način na koji se podaci pohranjuju ovisi o specifičnim potrebama aplikacije, vrsti podataka te zahtjevima za sigurnost i skalabilnost. Kod web aplikacija podaci se najčešće čuvaju na udaljenim bazama podataka, čime se osigurava jednostavan pristup i pouzdano upravljanje. Kroz sljedeća 2 poglavlja koja se bave pohranom podataka, naučit ćete kako ispravno spremati podatke u web aplikaciju, bilo da se radi o lokalnoj ili udaljenoj pohrani na poslužiteljskoj strani.

Posljednje ažurirano: 19.11.2024.

Sadržaj

1. Gdje pohranjujemo podatke u web aplikacijama?

Kada govorimo o pohrani podataka u web aplikacijama, važno je odmah razjasniti razliku između **klijentske** i **poslužiteljske** pohrane podataka. Web aplikacije u produkcijskom okruženju obično pohranjuju podatke na **obje razine**, kako bi se osigurala brza i učinkovita komunikacija između klijenta i poslužitelja.

Klijentska pohrana podataka (*eng. client-side storage*) odnosi se na spremanje podataka na korisničkom uređaju, obično unutar web preglednika, u obliku kolačića (cookies), lokalne memorije (*eng. local storage*), sesijske memorije (*eng. session storage*), ili drugih tehnologija (npr. IndexedDB) koje omogućuju privremeno ili trajno pohranjivanje podataka. Kod mobilnih aplikacija, klijentska pohrana može uključivati pohranu na prijenosnim uređajima (poput mobilnih telefona i tableta) putem tehnologija specifičnih za mobilne platforme.

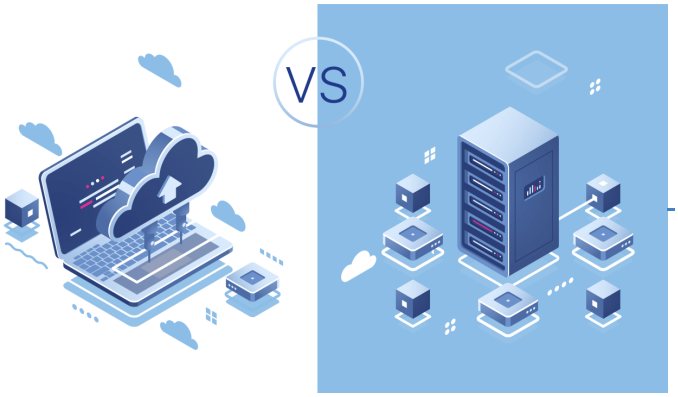
Podaci koji se pohranjuju na **klijentskoj strani** obično se koriste (samim time i pohranjuju) u sljedeće svrhe:

- personalizacija korisničkog iskustva (npr. boja pozadine, postavke jezika, odabrana paleta boja/tema, itd.)
- čuvanje korisničkih postavki (npr. preferirani način prikaza podataka, odabrane opcije, itd.)
- praćenje korisničkih aktivnosti (npr. praćenje kretanja korisnika kroz web stranicu, praćenje klikova na određene elemente)
- održavanje prethodne aktivnosti (npr. povijest pretraživanja, popis proizvoda u košarici, itd.)
- pohrana određenih podataka u svrhu optimizacije performansi (npr. predmemoriranje podataka, spremanje rezultata pretrage, itd.)

Poslužiteljska pohrana podataka (*eng. server-side storage*) odnosi se na pohranu podataka na udaljenom poslužitelju, obično u obliku baze podataka. Poslužiteljska pohrana omogućuje centralizirano upravljanje podacima, skalabilnost, sigurnost i pouzdanost. Baze podataka mogu biti relacijske (SQL) ili nerelacijske (NoSQL), ovisno o specifičnim potrebama aplikacije i karakteristikama pohranjenih podataka.

Prednosti pohrane na poslužiteljskoj strani uključuju:

- centralizirano upravljanje podacima (jednostavno pretraživanje, ažuriranje i brisanje podataka)
- visoka razina sigurnosti (pristup podacima kontroliran je na razini poslužitelja, što je *must-have* za osjetljive podatke)
- mogućnost skaliranja (u slučaju povećanja opterećenja, moguće je dodati nove poslužitelje ili resurse)



> **Ilustracija**: Pohrana podataka u web aplikacijama (klijentska i poslužiteljska pohrana)

2. Podaci na poslužiteljskoj strani

U primjerima do sad, odnosno na web poslužiteljima koje smo definirali (*naručivanje pizze*, *web shop odjeće*, *nekretnine*), podatke smo pohranjivali *in-memory*, odnosno u JS objekte. Međutim, ovo ne možemo nazivati stvarnim pohranjivanjem podataka, jer se podaci zapisuju privremeno i **nestaju prilikom gašenja poslužitelja**. Drugim riječima, pohranjuju se u RAM (radnu memoriju) poslužitelja, a ne na trajnom mediju.

Možemo zaključiti zašto ovakav pristup nije prikladan za stvarne web aplikacije, već isključivo za demonstracijske primjere, prototipove ili kao privremeno rješenja za vrijeme razvoja i testiranja.

Za vrijeme razvoja prethodnih primjera, osim *in-memory* pohrane podataka, iskoristili smo i lokalne datoteke - ručno smo zapisivali neke podatke u `js` datoteke te ih koristili kao vanjske resurse. Ovo je također jedan od načina pohrane podataka - **spremanje podataka u datoteke na poslužitelju**.

Naravno, podatke je na ovaj način moguće spremati u različitim formatima (npr. JSON, XML, CSV, itd.). Iako se na prvu čini kao solidna opcija za pohranu podataka, vidjet ćemo zašto ovaj pristup nije prikladan za stvarne web aplikacije. Ipak, neke web aplikacije na poslužiteljskoj (kao i klijentskoj) strani pohranjuju podatke u datoteke, međutim treba biti oprezan, vidjet ćete što je prikladno za pohranu u datoteke, a što nije.

2.1 Čitanje datoteka kroz `fs` modul

Krenimo s primjerom **čitanja podataka iz datoteka na poslužiteljskoj strani**. Za potrebe ovog primjera, koristit ćemo Node.js okruženje i ugrađeni `fs` modul (File System) koji omogućuje čitanje i pisanje u datoteke datotečnog sustava (*eng. file system*). Kako smo već prešli na `ES6` sintakse, držat ćemo se istog pristupa i prilikom korištenja `fs` modula.

Idemo definirati osnovni Express poslužitelj:

```
import express from 'express';

const app = express();

app.get('/', (req, res) => {
  res.status(200).send('Vrijeme je za čitanje datoteka!');
});

app.listen(3000, () => {
  console.log('Poslužitelj je pokrenut na portu 3000');
});
```

Uključit ćemo i `fs` modul (nije ga potrebno instalirati jer je ugrađen u Node.js):

```
import fs from 'fs';
```

Općenito, pohranu i čitanje podataka u datoteke možemo podijeliti na dva osnovna pristupa:

1. **Asinkroni pristup**

2. **Sinkroni pristup**

JavaScript je jednodretveni jezik (*eng. single-threaded*), što znači da se kod izvršava redom, u jednoj sekvencijalnoj niti (dretvi). Međutim, mehanizmi poput **asinkronog programiranja** i event loopa omogućuju nam da izvršavamo više operacija istovremeno, **bez blokiranja glavne dretve**. Na ovaj način, JavaScript se izvršava konkurentno, premda daje iluziju paralelnog izvršavanja. Blokiranjem glavne dretve, aplikacija bi postala neodaziva, odnosno korisniku bi se jednostavno "zamrznula".

U praksi, **asinkrono programiranje** koristimo za izvođenje operacija koje zahtijevaju vremenski zahtjevne operacije (npr. dohvaćanje podataka s udaljenog poslužitelja). Međutim, pisanje i čitanje datotečnog sustava također može biti vremenski zahtjevno, stoga je **preporučljivo koristiti asinkrone metode za pisanje i čitanje datoteka**.

2.1.1 Asinkroni pristup čitanju datoteke

Krenimo s primjerom asinkronog čitanja datoteke. Izradit ćemo datoteku `story.txt` i ručno pohraniti u nju neku kratku priču. Koristeći `fs` modul, čitat ćemo sadržaj datoteke i ispisivati ga u konzolu. Datoteku možete pronaći u direktoriju `app/data` repozitorija ovih vježbi.

Za **asinkrono čitanje datoteke**, koristimo metodu `fs.readFile()`:

Sintaksa:

```
fs.readFile(path, options, callback);
```

gdje su:

- `path` - putanja do datoteke (**obavezno**)
- `options` - specifikacija enkodiranja datoteke (opcionalno)
  - `encoding` - encoding datoteke (npr. `'utf8'`)
  - `flag` - opcionalni znak kojim se označava način pristupa datoteci (npr. `'r'` za čitanje)
- `callback` - callback funkcija koja se poziva nakon što se datoteka pročita (**obavezno**)

`callback` funkcija prima dva argumenta:

1. `err` - greška (ako postoji)
2. `data` - sadržaj datoteke (ako je pročitan)

Primjer čitanja datoteke `story.txt`:

```
// relativna putanja do datoteke 'story.txt'
fs.readFile('./data/story.txt', 'utf8', (err, data) => {
  // čitanje datoteke 'story.txt' u utf8 formatu
  if (err) {
    // ako se dogodila greška
    console.error('Greška prilikom čitanja datoteke:', err); // ispisuje grešku
    return;
  }

  console.log('Sadržaj datoteke:', data); // ispisuje sadržaj datoteke
});
```

U ovom primjeru, čitamo datoteku `story.txt` u utf-8 formatu. `utf-8` format je najčešće korišteni format za čitanje i pisanje tekstualnih datoteka u digitalnoj formi budući da podržava sve znakove Unicode standarda. Gotovo svaka web stranica, dokument ili programski kod napisan je u `utf-8` formatu.

Ako kod samo zaljepimo unutar poslužitelja, datoteka `story.txt` će se pročitati asinkrono čim se poslužitelj pokrene. Ukoliko datoteka ne postoji, bit će ispisana greška.

Možemo vidjeti ispis u konzoli:

```
Sadržaj datoteke: Već trideset i tri godine jedan stari ribar i njegova žena živjeli su
siromašno.

Trideset i tri godine stari ribar i njegova žena živjeli su siromašno u staroj i trošnoj
kolibi od gline na obali sinjeg mora. Dane su provodili usamljeno i skromno. Starac je
svaki dan išao loviti ribu kako bio on i žena imali što jesti, a starica je ostajala u
kolibi, prela i kuhala ručak.

“Živio na žalu sinjeg mora
Starac ribar sa staricom svojom;
U staroj su kolibi od gline
Proživjeli tri’es’t i tri ljeta.
Starac mrežom lovio je ribu,
A starica prela svoju pređu”

...
```

2.1.2 Apsolutna i Relativna putanja do datoteke

Prije nego nastavimo, važno je razumjeti razliku između **apsolutne** i **relativne** putanje do datoteke (*eng. file path*).

Apsolutna putanja (*eng. absolute path*) je putanja koja **počinje od korijenskog direktorija datotečnog sustava**. Na primjer, u Unix/Linux sustavima, korijenski direktorij je `/`, dok je u Windows sustavima to `C:\` (pretpostavka).

> Apsolutna putanja uvijek **počinje s korijenskim direktorijem** i **sadrži sve direktorije i datoteke koje se nalaze između korijenskog direktorija i ciljne datoteke**.

Primjer apsolutne putanje do datoteke `story.txt` na Windows sustavu:

```
C:\Users\Username\Documents\GitHub\WA4 - Pohrana podataka\data\story.txt
```

> **VAŽNO**: Windows sustavi koriste `\` kao separator direktorija, dok Unix/Linux sustavi koriste `/`.

Datoteku `story.txt` možemo pročitati koristeći apsolutnu putanju:

```
fs.readFile('C:\\Users\\Username\\Documents\\GitHub\\WA4 - Pohrana
podataka\\data\\story.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Greška prilikom čitanja datoteke:', err);
    return;
  }

  console.log('Sadržaj datoteke:', data);
});
```

Međutim, apsolutna putanja je specifična za svakog korisnika i njegov datotečni sustav. Također, teško je čitljiva i često je podložna greškama prilikom pisanja.

Osim toga, vidimo da smo u kodu koristili dvostruke kosine (`\\`) kao separator direktorija. Ovo je specifično za Windows sustave budući da jedna kosa crta (`\`) predstavlja **escape znak** u JavaScriptu. Kako bismo izbjegli ovu konflikt, koristimo dvostruke kose crte. Primjer, escape znak za novi red je `\n` pa samim tim `\\` predstavlja jednu kosa crtu unutar stringa.

> **Relativna putanja** (*eng. relative path*) je putanja koja **počinje od trenutnog radnog direktorija**. Relativna putanja **ne počinje s korijenskim direktorijem** i sadrži samo direktorije i datoteke koji se nalaze **između trenutnog radnog direktorija i ciljne datoteke**.

Trenutni radni direktorij možemo dobiti pomoću globalne varijable `__dirname` u CommonJS modulu ili putem `import.meta.url` u ES modulima. Ova varijabla sadrži putanju do trenutnog direktorija u kojem se nalazi trenutni modul, npr. `index.js` u našem slučaju.

Primjer relativne putanje do datoteke `story.txt`:

```
./data/story.txt
```

Važno je naglasiti da se relativna putanja **ne mijenja** ovisno o korisniku ili operacijskom sustavu. Međutim, **moramo biti oprezni prilikom pokretanja aplikacije iz različitih direktorija**.

Na primjer, ako se definicija poslužitelja `index.js` nalazi u direktoriju `app`, a datoteka `story.txt` u direktoriju `data` koji se također nalazi unutar direktorija `app`:

```
app
├── data
│   └── story.txt
├── index.js
├── node_modules
├── package-lock.json
└── package.json
```

relativna putanja do datoteke `story.txt` bit će:

```
./data/story.txt
```

> Točkom `.` označavamo **trenutni direktorij**, a zatim nizom direktorija i datoteka definiramo putanju do ciljne datoteke.

Međutim, ako se datoteka `story.txt` nalazi u direktoriju `data` koji se nalazi u korijenskom direktoriju projekta, npr:

```
WA4 - Pohrana podataka
├── data
│   └── story.txt
├── app
    ├── index.js
    ├── node_modules
    ├── package-lock.json
    └── package.json
```

tada će relativna putanja biti:

```
../data/story.txt
```

> Dvije točke `..` označavaju **roditeljski direktorij** (*eng. parent directory*), a zatim nizom direktorija i datoteka definiramo putanju do ciljne datoteke.

Trebamo paziti u kojem se direktoriju nalazi instanca terminala kako bismo mogli koristiti relativne putanje bez problema. Trenutnu putanju u direktoriju možemo provjeriti koristeći `pwd` naredbu u terminalu.

```
pwd
```

Kako bi pokrenuli sljedeći kod bez greške, odnosno kako bi se datoteka `story.txt` ispravno pročitala, moramo se **s terminalom nalaziti u direktoriju gdje se nalazi** `index.js` datoteka. Dakle unutar: `app/`.

```
fs.readFile('./data/story.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Greška prilikom čitanja datoteke:', err);
    return;
  }

  console.log('Sadržaj datoteke:', data);
});
```

`./data/story.txt` znači:

- `./` - trenutni direktorij (gdje se nalazi `index.js`)
- `data/` - direktorij `data` unutar trenutnog direktorija

- `story.txt` - datoteka `story.txt` unutar direktorija `data`

Međutim, ako se s terminalom nalazimo u korijenskom direktoriju projekta (`WA4 - Pohrana podataka`), te pokušamo pokrenuti poslužitelj, **dobit ćemo grešku prilikom čitanja datoteke**.

Primjerice, ako pokrećemo poslužitelj s: `node app/index.js`, pa i ako pokrećemo poslužitelj putem VS Code Run naredbe (problem je što ona koristi korijenski direktorij projekta), datoteka `story.txt` **neće biti pronađena**. Međutim, poslužitelj će se pokrenuti bez problema.

```
Poslužitelj je pokrenut na portu 3000
Greška prilikom čitanja datoteke: [Error: ENOENT: no such file or directory, open
'./data/story.txt'] {
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: './data/story.txt'
}
```

Dakle, ako se nalazimo u korijenskom direktoriju projekta, trebali bismo izmjeniti putanju do datoteke u:

```
fs.readFile('./app/data/story.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Greška prilikom čitanja datoteke:', err);
    return;
  }

  console.log('Sadržaj datoteke:', data);
});
```

> Sada radi, međutim ako terminalom opet uđemo u direktorij `app/`, kod će opet baciti grešku. Dakle, **relativne putanje ovise o trenutnom radnom direktoriju**.

2.1.3 `Callback` vs `Promise` pristup

Rekli smo da ćemo operacije s datotekama obavljati asinkrono, budući da one mogu potrajati i ne želimo zaustaviti rad poslužitelja dok se operacija ne završi. Idemo nadograditi naš poslužitelj na način da ćemo definirati endpoint `/story` koji će čitati datoteku `story.txt` i vraćati njen sadržaj kao odgovor.

```
import express from 'express';
import fs from 'fs';

const app = express();

app.get('/story', (req, res) => {
  fs.readFile('./data/story.txt', 'utf8', (err, data) => {
    if (err) {
      console.error('Greška prilikom čitanja datoteke:', err);
      return;
    }
```

```
    console.log('Sadržaj datoteke:', data);
    res.status(200).send(data);
  });
});

app.listen(3000, () => {
  console.log('Poslužitelj je pokrenut na portu 3000');
});
```

Međutim, nije uobičajeno da se kod koji se odnosi na čitanje datoteke nalazi unutar funkcije koja definira rutu, odnosno endpoint. Idemo ga prebaciti u zasebnu funkciju.

Česta greška 1:

Prebacit ćemo kod koji se odnosi na čitanje datoteke u zasebnu funkciju `read_story()`. Zatim ćemo definirati rutu `/story` koja će slati JSON odgovor rezultat poziva ove funkcije. Funkcija `read_story()` definira prazan string `story_text` koji će se popuniti sadržajem datoteke, a zatim se isti vraća kao rezultat funkcije. **Ovo je pogrešan pristup!**

```
function read_story() {
  let story_text = '';
  fs.readFile('./data/story.txt', 'utf8', (err, data) => {
    if (err) {
      console.error('Greška prilikom čitanja datoteke:', err);
      return;
    }

    console.log('Sadržaj datoteke:', data);
    story_text = data;
  });
  return story_text;
}

app.get('/story', (req, res) => {
  res.status(200).send(read_story());
});
```

Zašto ovo ne radi? 🤔

- `fs.readFile` je **asinkrona funkcija**. Kada se pozove `read_story()`, instancira se proces čitanja datoteke, međutim funkcija odmah vrati prazan string `story_text` prije nego što se datoteka pročita budući da je to radnja koja traje dulje. Kada se datoteka pročita, `story_text` se popuni sadržajem datoteke, međutim funkcija je već završila i vratila prazan string.
- `story_text` se nadopunjuje unutar callback funkcije koja se poziva **nakon što se datoteka pročita**. Međutim, prošao je voz, JavaScript je sekvencijalno izvršio kod u nastavku i vratio prazan string.
- mi ustvari ovdje pokušavamo upravljati asinkronim kodom na sinkroni način, što nije moguće.

Česta greška 2:

U redu, nećemo se predati. Pokušat ćemo riješiti problem tako da ćemo ustvari pohraniti rezultat izvršavanja funkcije `readFile` u varijablu `story_text`, a zatim vratiti tu varijablu kao rezultat funkcije `read_story()`. U endpointu ćemo prvo podatke definirati u varijablu, a zatim je poslati kao odgovor. **Ovo je isto pogrešan pristup!**

```
function read_story() {
  let story_text = fs.readFile('./data/story.txt', 'utf8', (err, data) => {
    if (err) {
      console.error('Greška prilikom čitanja datoteke:', err);
      return;
    }
    console.log('Sadržaj datoteke:', data);
    story_text = data;
  });
  return story_text;
}

app.get('/story', (req, res) => {
  let data = read_story();
  res.status(200).send(data);
});
```

Zašto ovo ne radi? 🤔

- iz istog razloga kao i prije, `fs.readFile` je asinkrona funkcija, a mi pokušavamo vratiti rezultat prije nego što se datoteka pročita. Drugim riječima, opet pokušavamo upravljati asinkronim kodom na sinkroni način.

Problem je moguće riješiti na 2 načina, **ovisno kako odaberemo obrađivati asinkrone operacije**:

> 1. Način: **Callback pattern**

Callback pattern u JavaScriptu predstavlja rješenje za upravljanje asinkronim operacijama koje sa bazira na pozivanju callback funkcija nakon što se operacija završi. Već ste naučili da je `callback` jednostavno funkcija koja se prosljeđuje kao argument drugoj funkciji, a koja se poziva nakon što se izvrši određena operacija (u nekom kasnijem vremenskom trenutku).

Kako radi callback pattern?

1. Prosljeđujemo callback funkciju kao argument drugoj funkciji
2. Funkcija koja prima callback funkciju izvršava isti callback jednom kad odradi svoj posao, odnosno kad se zadovolji neki uvjet
3. Navedeno dozvoljava "non-blocking", asinkrono programiranje

Sinkroni primjer:

```
function pozdrav(ime, callback) {
  console.log(`Pozdrav, ${ime}!`);
  callback(); // poziv callback funkcije nakon što se ispiše pozdravna poruka
}
```

```
function dovidenja() {
  console.log('Doviđenja!');
}

// pozivamo funkciju 'pozdrav' s callback funkcijom 'dovidenja'

pozdrav('Ivana', dovidenja);

// Ispisuje:

// Pozdrav, Ivana!
// Doviđenja!
```

Asinkroni primjer:

```
function fetch_data(callback) {
  console.log('Dohvaćam podatke s udaljenog poslužitelja...');

  setTimeout(() => {
    const podaci = { racun: 'HR1234567890', stanje: 5000 };
    callback(podaci); // poziv callback funkcije nakon što se dohvate podaci
  }, 2000); // simulacija čekanja 2 sekunde na dohvat podataka
}

function handle_data(podaci) {
  console.log('Podaci su dohvaćeni:', podaci);
}

// pozivamo funkciju 'simuliraj_dohvat_podataka' s callback funkcijom 'prikazi_podatke'

fetch_data(handle_data);

// Ispisuje:

// Dohvaćam podatke s udaljenog poslužitelja...
// nakon 2 sekunde...
// Podaci su dohvaćeni: { racun: "HR1234567890" , stanje: 5000 };
```

Idemo isto primijeniti na naš primjer čitanja datoteke:

Kojoj funkciji ćemo u primjeru iznad proslijediti callback argument? 🤔

▶ Spoiler alert! Odgovor na pitanje

```
function read_story(callback) {
  fs.readFile('./data/story.txt', 'utf8', callback); // ovdje prosljeđujemo callback
funkciju iz argumenta
}

app.get('/story', (req, res) => {
```

```
  read_story((err, data) => {
    // kao argument prosljeđujemo cijelu implementaciju callback funkcije
    if (err) {
      res.status(500).send('Greška prilikom čitanja priče');
    } else {
      res.send(data);
    }
  });
});
```

Callback funkcija je definirana arrow sintaksom, i izgleda ovako:

```
(err, data) => {
  if (err) {
    res.status(500).send('Greška prilikom čitanja priče');
  } else {
    res.send(data);
  }
};
```

Dakle, kod koji šalje odgovor klijentu nalazi se unutar callback funkcije koja se poziva nakon što se datoteka pročita. Na ovaj način, osiguravamo da se odgovor šalje tek nakon što se datoteka pročita, odnosno nakon što se završi asinkrona operacija. Bez obzira što implementacija callback funkcije možda izgleda kao da se izvršava odmah nakon poziva `read_story()`, ona se zapravo izvršava nakon što se datoteka pročita.

> 2. Način: **Promise pattern**

Kako bismo izbjegli "callback hell" (duboko gniježđenje callback funkcija), možemo koristiti `Promise` pattern. Sintaksa iznad možda izgleda neintuitivno, a kod postaje teško čitljiv i održiv s više callback funkcija. `Promise` pattern je moderniji pristup i omogućuje nam da se rješavamo callback funkcija i pišemo čišći i čitljiviji kod.

Međutim, kako bismo koristili `Promise` pattern, koristit ćemo ekstenziju `fs` modula - `fs.promises`. Ova ekstenzija omogućuje nam da koristimo `Promise` pattern za čitanje, kao i za pisanje u datoteke. Naravno, samim time možemo koristiti `async/await` sintaksu kako bi riješili `then` i `catch` lanca.

```
import fs from 'fs/promises';

app.get('/story', (req, res) => {
  fs.readFile('data/story.txt', 'utf8')
    .then(data => {
      // uspješno čitanje datoteke
      res.status(200).send(data);
    })
    .catch(error => {
      // greška prilikom čitanja datoteke
      console.error('Error reading file:', error);
      res.status(500).send('Error reading story file.');
    });
});
```

Vidimo da sad možemo koristiti `then` i `catch` lanac, što može biti čitljivije i čišće od korištenja callback funkcija. Međutim, najbolji način je sintaksu prenijeti u zasebnu funkciju i koristiti alternativnu `async/await` sintaksu.

Za početak ćemo samo primijeniti `async/await` sintaksu na prethodni primjer:

```
app.get('/story', async (req, res) => {
  try {
    // pokušaj izvršiti asinkronu operaciju
    const data = await fs.readFile('data/story.txt', 'utf8'); // pročitaj datoteku
'story.txt'
    res.status(200).send(data); // uspješan rezultat čitanja datoteke vrati u HTTP
odgovoru
  } catch (error) {
    // uhvati grešku
    console.error('Error reading file:', error);
    res.status(500).send('Error reading story file.'); // greška prilikom čitanja datoteke
  }
});
```

Kod za čitanje možemo prebaciti u zasebnu asinkronu funkciju:

```
async function read_story() {
  try {
    const data = await fs.readFile('data/story.txt', 'utf8'); // await budući da je
fs.readFile asinkrona funkcija
    return data;
  } catch (error) {
    console.error('Error reading file:', error);
    return null;
  }
}

app.get('/story', (req, res) => {
  const data = await read_story(); // await budući da je read_story također asinkrona
funkcija
  if (data) {
    res.status(200).send(data);
  } else {
    res.status(500).send('Error reading story file.');
  }
});
```

Vidimo grešku, zašto? 🤔

▶ Spoiler alert! Odgovor na pitanje

Ispravno:

```
app.get('/story', async (req, res) => {
  const data = await read_story(); // await budući da je read_story također asinkrona
funkcija
  if (data) {
    res.status(200).send(data);
  } else {
    res.status(500).send('Error reading story file.');
  }
});
```

> Možete odabrati koji pristup je vama draži, međutim `Promise` pattern i `async/await` sintaksa su moderniji pristupi i češće se koriste u praksi.

2.2 Pohrana u datoteke kroz `fs` modul

Rekli smo da pohrana u datoteke, kao i čitanje, može biti vremenski zahtjevno, stoga je preporučljivo koristiti asinkrone metode.

Za asinkronu pohranu u datoteku, koristimo metodu fs.writeFile():

Sintaksa:

```
fs.writeFile(path, data, options, callback);
```

gdje su:

- `path` - putanja do datoteke (**obavezno**)
- `data` - podaci koje želimo zapisati u datoteku (**obavezno**)
- `options` - specifikacija enkodiranja datoteke (opcionalno)
  - `encoding` - encoding datoteke (npr. `'utf8'`)
  - `flag` - opcionalni znak kojim se označava način pristupa datoteci (npr. `'w'` za pohranu (*default*))
- `callback` - callback funkcija koja se poziva nakon što se datoteka pročita (**obavezno**)

`callback` funkcija prima dva argumenta:

1. `err` - greška (ako postoji)
2. `data` - sadržaj datoteke (ako je pročitan)

Jednako kao i kod čitanja, moguće je koristiti `Callback` i `Promise` pattern za pohranu u datoteke. Međutim ponovo, `Promise` pattern i `async/await` sintaksa su moderniji pristupi.

Primjer pohrane u datoteku kroz `Callback` pattern:

```
app.get('/write', (req, res) => {
  const data = 'Ovo je tekst koji želimo zapisati u datoteku.';
  fs.writeFile('data/write.txt', data, 'utf8', err => {
    if (err) {
      console.error('Greška prilikom pohrane u datoteku:', err);
      res.status(500).send('Greška prilikom pohrane u datoteku.');
    } else {
      console.log('Podaci uspješno zapisani u datoteku.');
      res.status(200).send('Podaci uspješno zapisani u datoteku.');
    }
  });
});
```

Vidjet ćete novu datoteku `write.txt` u direktoriju `data` s tekstom: `Ovo je tekst koji želimo zapisati u datoteku.`.

Isto možemo postići i kroz `Promise` pattern odnosno `fs/promises` ekstenziju:

```
app.get('/write', async (req, res) => {
  const data = 'Ovo je tekst koji želimo zapisati u datoteku.';
  try {
    await fs.writeFile('data/write.txt', data, 'utf8');
    console.log('Podaci uspješno zapisani u datoteku.');
    res.status(200).send('Podaci uspješno zapisani u datoteku.');
  } catch (error) {
    console.error('Greška prilikom pohrane u datoteku:', error);
    res.status(500).send('Greška prilikom pohrane u datoteku.');
  }
});
```

Ili kroz zasebnu asinkronu funkciju:

```
async function write_data(data) {
  try {
    await fs.writeFile('data/write.txt', data, 'utf8');
    console.log('Podaci uspješno zapisani u datoteku.');
    return true;
  } catch (error) {
    console.error('Greška prilikom pohrane u datoteku:', error);
    return false;
  }
}

app.get('/write', async (req, res) => {
  const data = 'Ovo je tekst koji želimo zapisati u datoteku.';
  const success = await write_data(data);
  if (success) {
    res.status(200).send('Podaci uspješno zapisani u datoteku.');
  } else {
    res.status(500).send('Greška prilikom pohrane u datoteku.');
```

```
  }
});
```

Uočite jednu stvar koja nam ovdje ne odgovara. Implementacija je dobra i funkcionira, međutim mi šaljemo GET zahtjev za pohranu u datoteku. To naravno nije dobra praksa jer GET zahtjevi ne smiju mijenjati stanje na poslužitelju (također, ne šaljemo podatke već samo signal da želimo zapisati u datoteku, a zapisujemo tekst koji je hardkodiran).

U praksi, pohrana u datoteku obično se obavlja kroz `POST zahtjev ako se radi o kreiranju novih podataka ili` PUT `i` PATCH` zahtjev ako se radi o ažuriranju postojećih podataka.

Ako pogledati sintaksu iznad, možete vidjeti u opcijama `flag` parametar. Ovaj parametar označava način pristupa datoteci. Po *defaultu*, koristi se `w` flag koji označava zamjenu sadržaja datoteke novim sadržajem. Međutim, možemo koristiti i druge flagove:

- `r` - čitanje datoteke (*default* kod `fs.readFile`)
- `w` - pohrana u datoteku (*default* kod `fs.writeFile`), zamjena sadržaja datoteke novim sadržajem (najviše odgovara HTTP metodi `PUT`)
- `a` - dodavanje sadržaja na kraj datoteke, operacija append (najviše odgovara HTTP metodi `POST`)
- `r+` - čitanje i pohrana u datoteku, možemo koristiti kada želimo čitati i pisati istu datoteku simultano (najviše odgovara HTTP metodi `PATCH`)

U nastavku ćemo prikazati primjere pohrane u datoteku kroz oba pristupa (Callback i Promise), definirat ćemo i flagove za svaki primjer.

2.2.1 Pohrana `String` sadržaja u datoteku

U ovom primjeru, pohranit ćemo string sadržaj u datoteku `text.txt` kroz `Callback` pattern:

```
import fs from 'fs';

app.get('/write-callback', (req, res) => {
  const string = 'Ovo je tekst koji smo pohranili asinkrono u datoteku kroz Callback
pattern i w flag.';
  // flag je `w`, dakle svaki put ćemo zamijeniti sadržaj datoteke
  fs.writeFile('data/text.txt', string, { encoding: 'utf8', flag: 'w' }, err => {
    if (err) {
      console.error('Greška prilikom pohrane u datoteku:', err);
      res.status(500).send('Greška prilikom pohrane u datoteku.');
    } else {
      console.log('Podaci uspješno zapisani u datoteku.');
      res.status(200).send('Podaci uspješno zapisani u datoteku.');
    }
  });
});
```

Možemo dodavati i na kraj datoteke kroz `Promise` pattern.

```
imoprt fs from 'fs/promises';
```

```
app.get('/append-promise', async (req, res) => {
  const string = 'Ovo je tekst koji smo pohranili asinkrono u datoteku kroz Promise
pattern i a flag.';
  // flag je `a`, dakle svakim pozivom ćemo dodati sadržaj na kraj datoteke
  try {
    await fs.writeFile('data/text.txt', string, { encoding: 'utf8', flag: 'a' });
    console.log('Podaci uspješno zapisani u datoteku.');
    res.status(200).send('Podaci uspješno zapisani u datoteku.');
  } catch (error) {
    console.error('Greška prilikom pohrane u datoteku:', error);
    res.status(500).send('Greška prilikom pohrane u datoteku.');
  }
});
```

Vidimo da se tekst dodaje na kraj datoteke, a ne zamjenjuje (razmaci se ne dodaju automatski).

2.2.2 Čitanje i pohrana `JSON` podataka u datoteku

U ovom primjeru, pohranit ćemo JSON podatke u datoteku `data.json` kroz `Callback` pattern i *defaultne* opcije:

```
let student_pero = {
  ime: 'Pero',
  prezime: 'Perić',
  godine: 20,
  fakultet: 'FIPU'
};
```

Podsjetnik kako izgleda JSON objekt koji ćemo pohraniti:

```
{
  "ime": "Pero",
  "prezime": "Perić",
  "godine": 20,
  "fakultet": "FIPU"
}
```

Međutim, potrebno je odraditi konverziju JSON objekta u string prije pohrane u datoteku (proces serijalizacije):

Podsjetnik: **Serijalizacija/Deserijalizacija**:

- **Serijalizacija** (*eng. serialization*) je proces pretvaranja objekta u niz bajtova kako bi se mogao pohraniti u memoriju, bazi podataka ili datoteci. U našem slučaju, serijalizacija je pretvaranje JavaScript objekta `student_pero` u JSON string. Za to koristimo funkciju `JSON.stringify()`.
- **Deserijalizacija** (*eng. deserialization*) je proces pretvaranja niza bajtova u objekt. U našem slučaju, deserijalizacija je pretvaranje JSON stringa u JavaScript objekt. Za to koristimo funkciju `JSON.parse()`.

```
import fs from 'fs';
app.get('/write-json-callback', (req, res) => {
  // flag je defaultni `w`, dakle svaki put ćemo zamijeniti sadržaj datoteke.
Serijalizacija kroz JSON.stringify()
  fs.writeFile('data/data.json', JSON.stringify(student_pero), err => {
    if (err) {
      console.error('Greška prilikom pohrane u datoteku:', err);
      res.status(500).send('Greška prilikom pohrane u datoteku.');
    } else {
      console.log('Podaci uspješno zapisani u datoteku.');
      res.status(200).send('Podaci uspješno zapisani u datoteku.');
    }
  });
});
```

Isto možemo postići i kroz `Promise` pattern:

```
import fs from 'fs/promises';

app.get('/write-json-promise', async (req, res) => {
  // flag je defaultni `w`, dakle svaki put ćemo zamijeniti sadržaj datoteke.
Serijalizacija kroz JSON.stringify()
  try {
    await fs.writeFile('data/data.json', JSON.stringify(student_pero));
    console.log('Podaci uspješno zapisani u datoteku.');
    res.status(200).send('Podaci uspješno zapisani u datoteku.');
  } catch (error) {
    console.error('Greška prilikom pohrane u datoteku:', error);
    res.status(500).send('Greška prilikom pohrane u datoteku.');
  }
});
```

Kako se radi o pohrani u datoteku, moramo zamijeniti kod iznad `POST` metodom, dok ćemo JSON direktno preuzeti iz tijela zahtjeva:

```
import fs from 'fs/promises';

app.post('/student', async (req, res) => {
  const student = req.body;

  if (Object.keys(student).length === 0) {
    return res.status(400).send('Niste poslali podatke.');
  }

  try {
    await fs.writeFile('data/data.json', JSON.stringify(student));
    console.log('Podaci uspješno zapisani u datoteku.');
    res.status(200).send('Podaci uspješno zapisani u datoteku.');
  } catch (error) {
```

```
    console.error('Greška prilikom pohrane u datoteku:', error);
    res.status(500).send('Greška prilikom pohrane u datoteku.');
  }
});
```

Dakle kod iznad zamjenjuje cijeli resurs. Ako bismo dodavali podatke na kraj datoteke, koristili bismo `a` flag. Međutim, u tom slučaju pravilno je koristiti `PUT` metodu budući da se radi o ažuriranju postojećeg resursa `data.json`.

```
import fs from 'fs/promises';

// endpoint ima isti naziv, promijenili smo samo metodu u PUT
app.put('/student', async (req, res) => {
  const student = req.body;

  if (Object.keys(student).length === 0) {
    return res.status(400).send('Niste poslali podatke.');
  }

  try {
    await fs.writeFile('data/data.json', JSON.stringify(student), { flag: 'a' });
    console.log('Podaci uspješno zapisani u datoteku.');
    res.status(200).send('Podaci uspješno zapisani u datoteku.');
  } catch (error) {
    console.error('Greška prilikom pohrane u datoteku:', error);
    res.status(500).send('Greška prilikom pohrane u datoteku.');
  }
});
```

Radi, međutim vidimo da se podaci dodaju na kraj datoteke, bez zareza koji bi odvojio dva JSON objekta.

Jedan od načina na koji možemo riješiti ovaj problem je da:

- prvo pročitamo datoteku,
- deserijaliziramo JSON podatke,
- dodamo novi podatak,
- a zatim serijaliziramo i
- pohranimo natrag u datoteku.

Ispraznite JSON datoteku i pošaljite `POST` zahtjev s JSON tijelom:

```
[
  {
    "ime": "Pero",
    "prezime": "Perić",
    "godine": 20,
    "fakultet": "FIPU"
  }
]
```

Sada kada deserijaliziramo JSON podatke, dobit ćemo polje objekata, a ne jedan objekt. Upravo to i želimo kako bismo mogli pozvati `push()` metodu nad poljem objekata.

```
import fs from 'fs/promises';

app.put('/student', async (req, res) => {
  const student = req.body;

  if (Object.keys(student).length === 0) {
    return res.status(400).send('Niste poslali podatke.');
  }

  try {
    // pročitaj datoteku
    const data = await fs.readFile('data/data.json', 'utf8');
    // deserijaliziraj JSON podatke
    const students = JSON.parse(data);
    // dodaj novog studenta
    students.push(student);
    // serijaliziraj i pohrani
    await fs.writeFile('data/data.json', JSON.stringify(students));
    console.log('Podaci uspješno zapisani u datoteku.');
    res.status(200).send('Podaci uspješno zapisani u datoteku.');
  } catch (error) {
    console.error('Greška prilikom pohrane u datoteku:', error);
    res.status(500).send('Greška prilikom pohrane u datoteku.');
  }
});
```

Koristeći kod iznad, poslat ćemo `PUT` zahtjev s novim studentom, a on će se dodati na kraj polja objekata u datoteci `data.json`.

Tijelo `PUT` zahtjeva:

```
{
  "ime": "Ana",
  "prezime": "Anić",
  "godine": 18,
  "fakultet": "FIPU"
}
```

Vidimo da smo dobili dosta zapetljan kod, gdje moramo prvo čitati, a nakon tog dodavati, serijalizirati i pohranjivati objekte. Stvari možemo pojednostaviti još jednom ekstenzijom, ovaj put `fs-extra`. Ova ekstenzija nudi mnoge korisne metode koje olakšavaju rad s datotekama, uključujući gotove metode za čitanje i pisanje JSON podataka.

Ovaj modul moramo naknadno instalirati:

```
npm install fs-extra
```

Iskoristit ćemo funkcije `readJson()` i `writeJson()` koje su dostupne u `fs-extra` modulu te napisati istu `PUT` metodu:

```
import fs from 'fs-extra';

app.put('/student', async (req, res) => {
  const student = req.body;

  if (Object.keys(student).length === 0) {
    return res.status(400).send('Niste poslali podatke.');
  }

  try {
    const students = await fs.readJson('data/data.json'); // pročitaj datoteku,
deserijaliziraj JSON podatke i pohrani u varijablu
    students.push(student); // dodaj novog studenta u polje
    await fs.writeJson('data/data.json', students); // serijaliziraj i pohrani u datoteku

    console.log('Podaci uspješno zapisani u datoteku.');
    res.status(200).send('Podaci uspješno zapisani u datoteku.');
  } catch (error) {
    console.error('Greška prilikom pohrane u datoteku:', error);
    res.status(500).send('Greška prilikom pohrane u datoteku.');
  }
});
```

Koristeći `fs-extra` modul, možemo pojednostaviti kod i izbjeći ručno čitanje i pisanje JSON podataka, odnosno serijalizaciju i deserijalizaciju.

Tek sad kad smo se namučili s čitanjem i pisanjem u datoteke, možemo se vratiti na našu priču **zašto možda nije najbolje rješenje koristiti datoteke za pohranu podataka**.

Vidjeli smo da pohrana i čitanje datoteka nije tako jednostavna operacija, premda se tako naizgled čini. U praksi, datoteke se koriste za pohranu podataka koji se **rijetko mijenjaju**, kao što su konfiguracijske datoteke, datoteke s logovima, datoteke s podacima koje je potrebno čuvati između restarta aplikacije i slično.

Problemi **skalabilnosti** su očiti. Što je potrebno promijeniti strukturu podataka našeg studenta u primjeru iznad? Što ako imamo veliki broj datoteka, kako ćemo ih ažurirati? Što ako naša baza korisnika toliko naraste da postane neučinkovito sve pohranjivati u datoteke, kako ćemo dijeliti datoteke između više instanci aplikacije/poslužitelja?

Što ako želimo pretraživati podatke, filtrirati, sortirati, spajati, grupirati? Sve ove operacije su moguće, ali su puno jednostavnije i efikasnije kroz **baze podataka**.

Jedan od većih problema je i **konkurentnost** i **sigurnost**. Što ako više korisnika istovremeno pokuša čitati i pisati u istu datoteku? Kako ćemo osigurati da se podaci ne izgube, ne prepišu, ne završe u nekom nevaljalom stanju?

> Ovo su se pitanja kojima se bave developeri koji aktivno rade na razvoju baza podataka. **DBMS** (eng. Database Management System) su sustavi koji su razvijeni upravo iz ovih razloga; kako bi olakšali pohranu, upravljanje, pretraživanje, ažuriranje i brisanje podataka na siguran i efikasan način, uz osiguranje konzistentnosti i integriteta podataka.

3. Agregacija podataka kroz `Query` parametre

Ipak, prije nego se krenemo baviti bazama podataka (u sljedećem poglavlju), moramo naučiti kako agregirati podatke na poslužiteljskoj strani kroz `query` parametre.

Query parametri su dio URL-a koji se koristi za prenošenje informacije o resursu koji se traži ili o akciji koja se želi izvršiti. `Query` parametri se dodaju na URL nakon znaka `?` i odvajaju se znakom `&`. Svaki `query` parametar sastoji se od imena i vrijednosti, odvojenih znakom `=`.

Sintaksa:

```
http://localhost:3000/route?key1=value1
```

gdje je:

- `?` - znak koji označava početak `query` parametara
- `key1` - ime `query` parametra
- `value1` - vrijednost `query` parametra

Dakle, ove parametre šaljemo kao dio URL-a, najčešće je to unutar `GET` zahtjeva.

Zašto `GET`? Uobičajeno je koristiti ovu vrstu parametra za slanje `GET` zahtjeva kada želimo dohvatiti određeni podskup podataka, npr. filtrirati po nekom kriteriju, sortirati, paginirati i slično.

3.1 Filtriranje podataka

Uzet ćemo primjer poslužitelja sa studentima iz prethodnog poglavlja:

```
import express from 'express';
import fs from 'fs/promises';

const app = express();
app.use(express.json());

app.get('/students', async (req, res) => {
  try {
    const data = await fs.readFile('data/students.json', 'utf8');
    const students = JSON.parse(data);
    res.status(200).send(students);
  } catch (error) {
    console.error('Greška prilikom čitanja datoteke:', error);
    res.status(500).send('Greška prilikom čitanja datoteke.');
  }
```

```
});

app.listen(3000, () => {
  console.log('Poslužitelj je pokrenut na http://localhost:3000');
});
```

U datoteku `students.json` pohranit ćemo ručno nekoliko studenata:

```
[
  { "ime": "Pero", "prezime": "Perić", "godine": 20, "fakultet": "FIPU" },
  { "ime": "Ana", "prezime": "Anić", "godine": 18, "fakultet": "FIPU" },
  { "ime": "Ivo", "prezime": "Ivić", "godine": 22, "fakultet": "FIPU" },
  { "ime": "Mara", "prezime": "Marić", "godine": 21, "fakultet": "FET" },
  { "ime": "Jure", "prezime": "Jurić", "godine": 19, "fakultet": "FET" },
  { "ime": "Iva", "prezime": "Ivić", "godine": 23, "fakultet": "FET" }
]
```

Ako pošaljemo `GET` zahtjev na `http://localhost:3000/students`, dobit ćemo sve studente u JSON odgovoru. Međutim, što ako želimo dohvatiti samo studente koji studiraju na `FIPU`? Isto možemo postići kroz `query` parametre.

Ažurirat ćemo postojeću rutu `/students` kako bismo omogućili filtriranje studenata prema fakultetu:

Ključ nam ovdje može biti `fakultet`, a vrijednost `FIPU`. Ukoliko želimo dohvatiti studente s fakulteta `FIPU`, URL bi izgledao ovako:

```
http://localhost:3000/students?fakultet=FIPU
```

Međutim, samu rutu **nećemo izmjenjivati**, već ćemo dohvaćati `query` parametre iz `req.query` objekta.

> **Uočite**, `req.query` je objekt koji sadrži sve `query` **parametre** poslane u URL-u. Nemojte ovo miješati s `req.params` objektom koji drugu vrstu parametara - **parametre rute**.

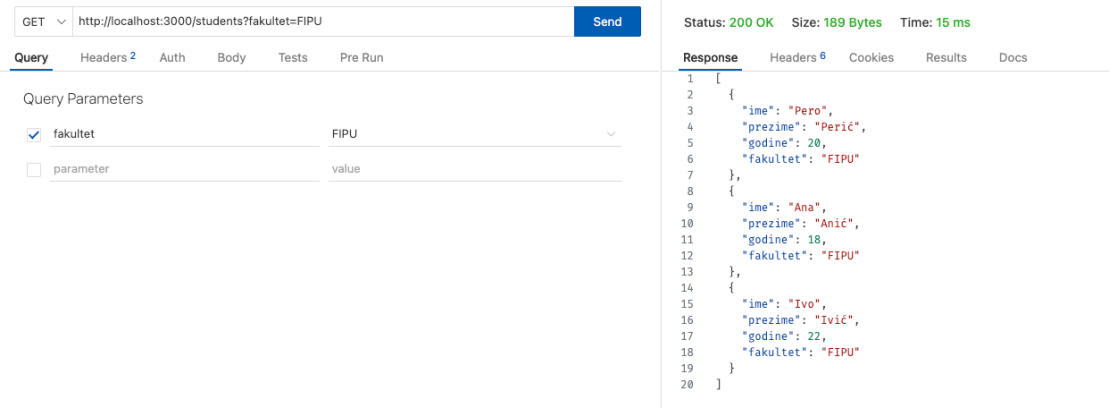
```
app.get('/students', async (req, res) => {
  let fakultet_query = req.query.fakultet; // dohvatimo query parametar 'fakultet'
  try {
    const data = await fs.readFile('data/students.json', 'utf8');
    const students = JSON.parse(data);
    res.status(200).send(students);
  } catch (error) {
    console.error('Greška prilikom čitanja datoteke:', error);
    res.status(500).send('Greška prilikom čitanja datoteke.');
  }
});
```

Vidimo da URL ostaje isti! Sada je potrebno samo odraditi filtriranje koristeći funkciju `filter()` nad poljem studenata:

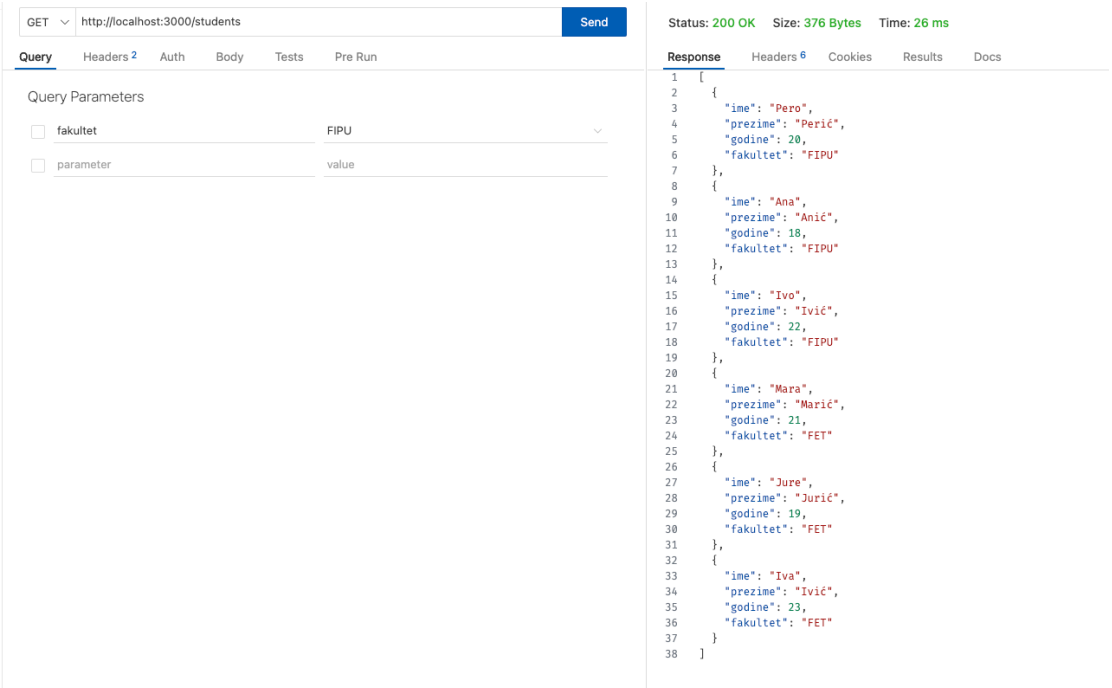
```
app.get('/students', async (req, res) => {
  let fakultet_query = req.query.fakultet; // dohvatimo query parametar 'fakultet'
```

```
  try {
    const data = await fs.readFile('data/students.json', 'utf8');
    const students = JSON.parse(data);

    if (fakultet_query) {
      const filtered_students = students.filter(student => student.fakultet ===
fakultet_query);
      res.status(200).send(filtered_students);
    } else {
      res.status(200).send(students);
    }
  } catch (error) {
    console.error('Greška prilikom čitanja datoteke:', error);
    res.status(500).send('Greška prilikom čitanja datoteke.');
  }
});
```

Možemo testirati kroz web preglednik ili Thunder Client/Postman. HTTP klijenti nude opciju unosa `query` parametara kao ključ vrijednost parova pa ih možemo unijeti i na taj način ili direktno u URL.



Ako maknemo `query` parametar, dobit ćemo sve studente.



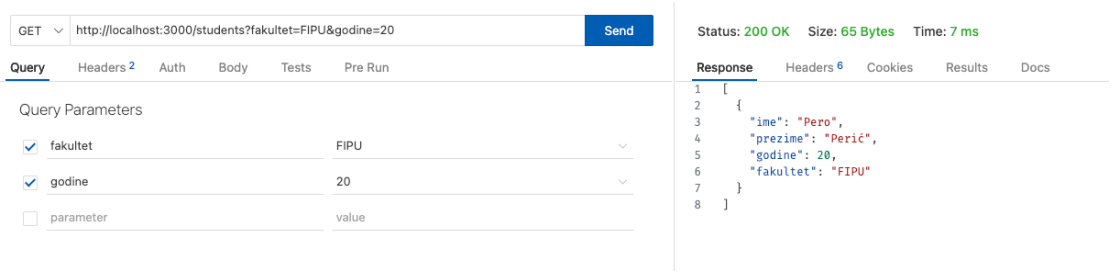
Moguće je definirati i više `query` parametara, npr. `godine`, `prezime`, `ime` i slično. Ukoliko želimo filtrirati studente po više kriterija, možemo koristiti `&` operator unutar URL-a:

Recimo, želimo studente s fakulteta `FIPU` i godinama `20`:

```
http://localhost:3000/students?fakultet=FIPU&godine=20
```

U kodu moramo samo dohvatiti dodatni parametar i nadograditi filter:

```
app.get('/students', async (req, res) => {
  let fakultet_query = req.query.fakultet; // dohvatimo query parametar 'fakultet'
  let godine_query = req.query.godine; // dohvatimo query parametar 'godine'
  try {
    const data = await fs.readFile('data/students.json', 'utf8');
    const students = JSON.parse(data);

    if (fakultet_query && godine_query) {
      const filtered_students = students.filter(student => student.fakultet ===
fakultet_query && student.godine === parseInt(godine_query));
      res.status(200).send(filtered_students);
    } else if (fakultet_query) {
      const filtered_students = students.filter(student => student.fakultet ===
fakultet_query);
      res.status(200).send(filtered_students);
    } else {
      res.status(200).send(students);
    }
  } catch (error) {
    console.error('Greška prilikom čitanja datoteke:', error);
    res.status(500).send('Greška prilikom čitanja datoteke.');
  }
});
```



To je to! Filtriranje možemo implementirati po želji puno različitih načina kroz `query` parametre.

Važno je ovdje uočiti sljedeće:

- `query` parametri su **opcionalni**. Ako ih ne pošaljemo, dobit ćemo sve studente.
- `query` parametri su **neovisni**. Ako pošaljemo samo jedan parametar, dobit ćemo filtrirane studente samo prema tom parametru.
- `query` parametre želimo koristiti isključivo za neki oblik **agregacije podataka**

- `query` parametre **ne želimo koristiti** kao zamjenu za **parametre rute**. Parametri rute su **obavezni** ako postoje i koriste se dohvat **pojedinog resursa**

Posebno se osvrnite na posljednju stavku.

Recimo, ako želimo dohvatiti pojedinog studenta, ne želimo definirati query parametar `id` ili `ime`. Takve stvari rješavamo kroz parametre ruta (:id, :ime) i dohvaćamo ih kroz `req.params` objekt. Dodatno, takve rute želimo definirati kao posljednje u nizu ruta kako bi se izbjeglo preklapanje s `query` parametrima.

Ako želimo rutu za dohvaćanje svih studenata, definiramo je kao prvu rutu, a zatim ju nadograđujemo s `query` parametrima.

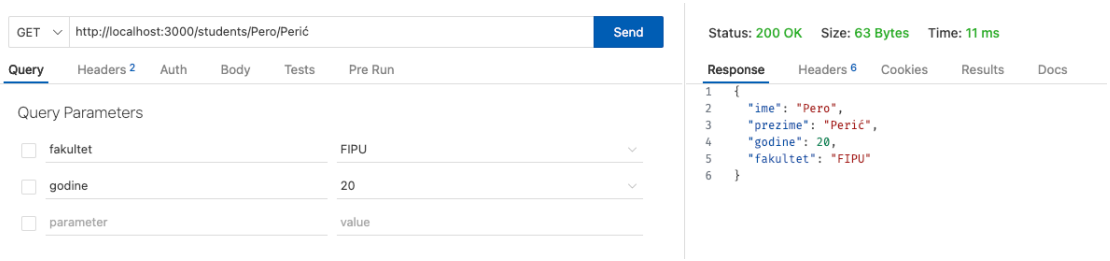
> Sljedeću rutu želimo pozivati na način: `http://localhost:3000/students?fakultet=FET` ili `http://localhost:3000/students?fakultet=FIPU&godine=20`

```
app.get('/students', async (req, res) => {
  let fakultet_query = req.query.fakultet; // dohvatimo query parametar 'fakultet'
  let godine_query = req.query.godine; // dohvatimo query parametar 'godine'
  try {
    const data = await fs.readFile('data/students.json', 'utf8');
    const students = JSON.parse(data);

    if (fakultet_query && godine_query) {
      const filtered_students = students.filter(student => student.fakultet ===
fakultet_query && student.godine === parseInt(godine_query));
      res.status(200).send(filtered_students);
    } else if (fakultet_query) {
      const filtered_students = students.filter(student => student.fakultet ===
fakultet_query);
      res.status(200).send(filtered_students);
    } else {
      res.status(200).send(students);
    }
  } catch (error) {
    console.error('Greška prilikom čitanja datoteke:', error);
    res.status(500).send('Greška prilikom čitanja datoteke.');
  }
});
```

Dohvat pojedinog studenta definiramo kao **zasebnu rutu** na sljedeći način, uz lošu pretpostavku da su ime i prezime jedinstveni:

```
app.get('/students/:ime/:prezime', async (req, res) => {
  let ime = req.params.ime;
  let prezime = req.params.prezime;
  try {
    const data = await fs.readFile('data/students.json', 'utf8');
    const students = JSON.parse(data);
    const student = students.find(student => student.ime === ime && student.prezime ===
prezime);
    if (student) {
```

```
      res.status(200).send(student);
    } else {
      res.status(404).send('Student nije pronađen.');
    }
  } catch (error) {
    console.error('Greška prilikom čitanja datoteke:', error);
    res.status(500).send('Greška prilikom čitanja datoteke.');
  }
});
```



3.2 Sortiranje podataka

`Query` parametre ne moramo koristiti samo za filtriranje podataka, možemo i za sortiranje. Uzmimo primjer gdje želimo sortirati studente po godinama uzlazno ili silazno.

U tom slučaju možemo definirati `query` parametar `sortiraj_po_godinama` koji će imati vrijednosti `uzlazno` ili `silazno`.

```
http://localhost:3000/students?sortiraj_po_godinama=uzlazno
```

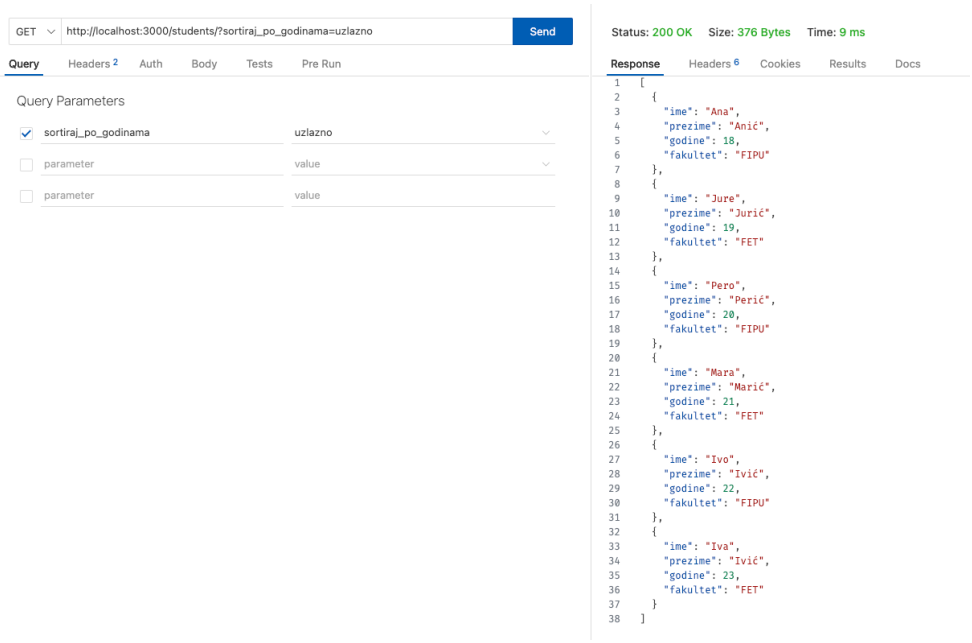
U kodu, dohvatimo `query` parametar i sortirajmo studente koristeći metodu `Arary.sort()`:

Radi jednostavnosti, izostavit ćemo logiku za filtriranje:

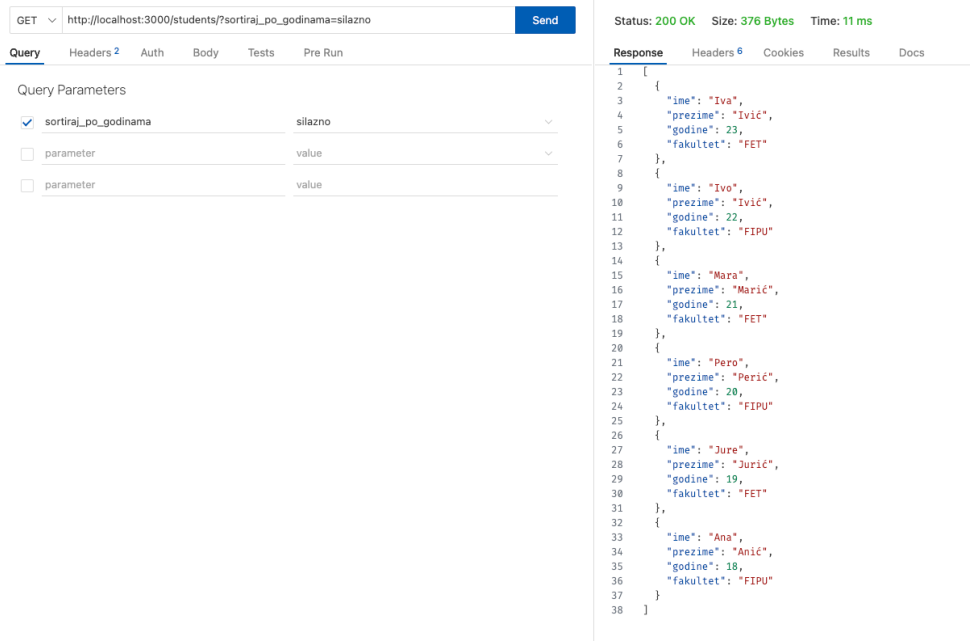
```
app.get('/students', async (req, res) => {
  let sortiraj_po_godinama = req.query.sortiraj_po_godinama; // dohvatimo query parametar
'sortiraj_po_godinama'
  try {
    const data = await fs.readFile('data/students.json', 'utf8');
    const students = JSON.parse(data);

    if (sortiraj_po_godinama) {
      if (sortiraj_po_godinama === 'uzlazno') {
        students.sort((a, b) => a.godine - b.godine);
      } else if (sortiraj_po_godinama === 'silazno') {
        students.sort((a, b) => b.godine - a.godine);
      }
    }

    res.status(200).send(students);
  } catch (error) {
    console.error('Greška prilikom čitanja datoteke:', error);
    res.status(500).send('Greška prilikom čitanja datoteke.');
```

```
  }
});
```

Sortiranje po godinama **uzlazno**:



Sortiranje po godinama **silazno**:



Za kraj, dozvoljeno je i kombiniranje `query` parametra i parametra rute. Recimo da želimo dohvatiti resurs našeg studenta po imenu i prezimenu (param `:ime/:prezime`), ali dodati dodatni filter `fakultet` putem `query` parametra:

Želimo poslati zahtjev na sljedeći način:

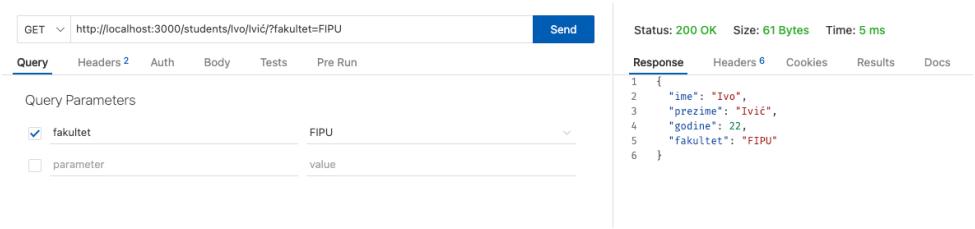
```
http://localhost:3000/students/Pero/Perić?fakultet=FIPU
```

> **Čitamo**: Dohvati određenog studenta s imenom `Pero` i prezimenom `Perić` koji studira na fakultetu `FIPU`. Bilo bi točnije dohvaćati po `id` parametru, ali za potrebe primjera koristimo ime i prezime.

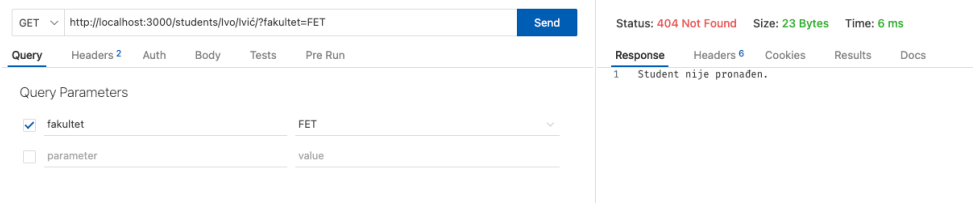
U kodu, dohvatimo `query` parametar i parametre rute:

```
app.get('/students/:ime/:prezime', async (req, res) => {
  let ime = req.params.ime; //parametar rute ime
  let prezime = req.params.prezime; // parametar rute prezime
  let fakultet_query = req.query.fakultet; // dohvatimo query parametar 'fakultet'
  try {
    const data = await fs.readFile('data/students.json', 'utf8');
    const students = JSON.parse(data);
    const student = students.find(student => student.ime === ime && student.prezime === 
prezime && student.fakultet === fakultet_query);
    if (student) {
      res.status(200).send(student);
    } else {
      res.status(404).send('Student nije pronađen.');
    }
  } catch (error) {
    console.error('Greška prilikom čitanja datoteke:', error);
    res.status(500).send('Greška prilikom čitanja datoteke.');
  }
});
```

Primjer dohvaćanja studenta s imenom `Ivo` i prezimenom `Ivić` (:ime/:prezime) koji studira na fakultetu `FIPU` (`?fakultet=FIPU`):



Primjer dohvaćanja istog resursa, ali s pogrešnim fakultetom u `query` parametru:



Samostalni zadatak za Vježbu 4

Izradite novi Express poslužitelj i definirajte jednostavni API za upravljanje podacima o zaposlenicima neke organizacije. API treba imati sljedeće rute:

- `GET /zaposlenici` - dohvat svih zaposlenika
- `GET /zaposlenici/:id` - dohvat zaposlenika po ID-u
- `POST /zaposlenici` - dodavanje novog zaposlenika

Implementirajte osnovne funkcionalnosti za dohvat, dodavanje i dohvat pojedinog zaposlenika. Zaposlenik treba imati sljedeće atribute:

- `id` - jedinstveni identifikator zaposlenika (generira se na poslužitelju)
- `ime` - ime zaposlenika
- `prezime` - prezime zaposlenika
- `godine_staža` - godine radnog staža zaposlenika
- `pozicija` - pozicija zaposlenika u organizaciji (npr. direktor, voditelj, programer, dizajner, itd.)

Pohranite prvo ručno nekoliko zaposlenika u JSON datoteku `zaposlenici.json`.

1. Definirajte osnovu validaciju podataka za sva 3 zahtjeva: provjera jesu li svi podaci poslani, jesu li ID i godine staža brojevi, jesu li ime i prezime stringovi itd. Ukoliko podaci nisu ispravni, vratite odgovarajući status i poruku greške. Ukoliko nisu pronađeni zaposlenici, vratite odgovarajući status i poruku.
2. Implementirajte mogućnost dodavanja novog zaposlenika. Zaposlenik se dodaje na kraj polja zaposlenika u datoteci. Morate koristiti `POST` metodu i poslati JSON tijelo s podacima o zaposleniku te spremati podatke u JSON datoteku kroz proces serijalizacije/deserijalizacije podataka.

Implementirajte sljedeće `query` parametre na endpointu `/zaposlenici`:

- `sortiraj_po_godinama` - sortiranje svih zaposlenika po godinama staža uzlazno ili silazno
- `pozicija` - filtriranje svih zaposlenika po poziciji u organizaciji
- `godine_staža_min` - filtriranje svih zaposlenika po minimalnom broju godina staža
- `godine_staža_max` - filtriranje svih zaposlenika po maksimalnom broju godina staža