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 datotekama, Asinkroni Pristupi i Agregacija podataka

Ispravna upotreba i upravljanje podacima ključni su aspekti razvoja web aplikacija. U ovom poglavlju, fokusirat ćemo se na dvije važne teme: upravljanje datotekama na poslužiteljskoj strani i agregaciju podataka putem *query* parametara. Odabir načina pohrane ovisi o funkcionalnim zahtjevima sustava, strukturi podataka te potrebama u pogledu sigurnosti i skalabilnosti. U praksi se podaci najčešće pohranjuju na udaljenim bazama podataka, čime se omogućuje centralizirano upravljanje te jednostavan i siguran pristup. Međutim, postoje situacije u kojima je prikladno koristiti datoteke za pohranu podataka, osobito za manje količine podataka ili specifične formate koji nam se ne uklapaju u strukturu baze podataka. U tom slučaju moramo naučiti kako ispravno upravljati datotekama na poslužiteljskoj strani.

Posljednje ažurirano: 10.12.2025.

Sadržaj

- Web aplikacije (WA)
- (4) Upravljanje datotekama, Asinkroni Pristupi i Agregacija podataka
  - Sadržaj
- 1. Gdje pohranjujemo podatke u web aplikacijama?
- 2. Podaci na poslužiteljskoj strani
  - 2.1 Čitanje tekstualnih datoteka kroz `fs` modul
    - 2.1.1 Asinkroni pristup čitanju datoteke
    - 2.1.2 Apsolutna i Relativna putanja do datoteke
    - 2.1.3 Modul `path` za upravljanje putanjama
    - 2.1.4 *Callback* vs *Promise* pristup asinkronom programiranju
  - 2.2 Pohrana u datoteke kroz `fs` modul

        - 2.2.1 Pohrana `String` sadržaja u datoteku
        - 2.2.2 Čitanje i pohrana `JSON` podataka u datoteku
- 3. Agregacija podataka kroz parametre upita (Query Parameters)
    - 3.1 Query parametri: Filtriranje podataka
    - 3.2 Query parametri: Sortiranje podataka
    - 3.3 Kombiniranje parametra rute i query parametara
- Samostalni zadatak za Vježbu 4

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 učinkovita i sigurna 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 (*eng. cookies*), lokalne memorije (*eng. local storage*), sesijske memorije (*eng. session storage*), ili drugih tehnologija (npr. IndexedDB, WebSQL) 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.

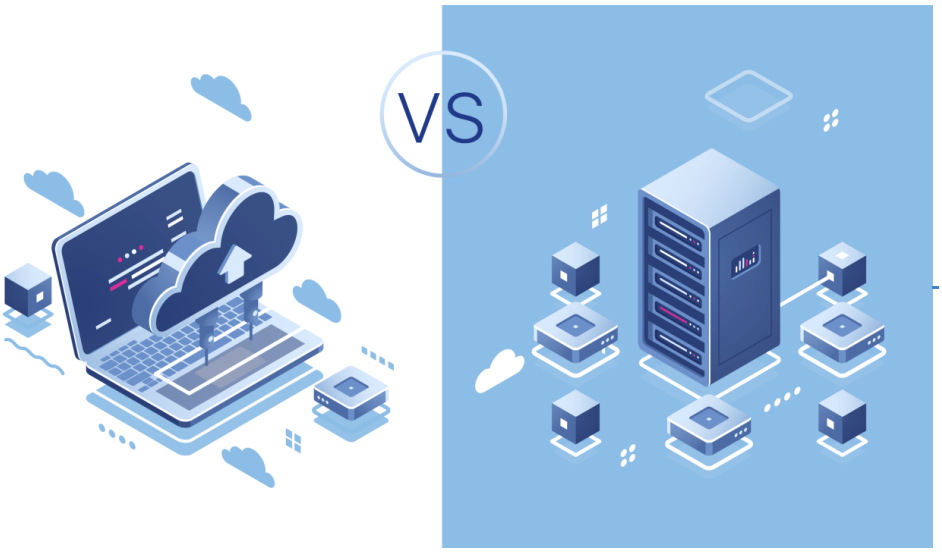
Podaci koji se pohranjuju na **klijentskoj strani** obično se koriste (samim time i pohranjuju) u sljedeće svrhe:

- **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 velikih 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 u pristupu podacima. Baze podataka mogu se u grubo podijeliti na 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)



> Slika 1: Podaci se mogu pohranjivati na klijentskoj i poslužiteljskoj strani. Vrlo je važno razumijeti razliku između ova dva pristupa i naučiti koje podatke je prikladno pohranjivati na kojoj strani i na koji način.

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.

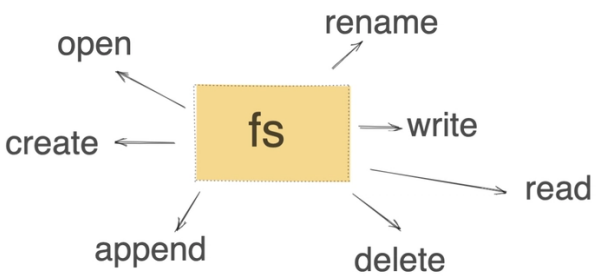
Za vrijeme razvoja prethodnih primjera, osim *in-memory* pohrane podataka, iskoristili smo i lokalne datoteke - ručno smo zapisivali neke podatke u `js` datoteke te ih koristili kao vanjske resurse. Ovo je također jedan od načina pohrane podataka - **spremanje podataka u datoteke na poslužitelju**.

Naravno, podatke je ovim pristupom moguće pohraniti u različite formate (npr. JSON, XML, CSV i dr.). Iako se na prvi pogled čini kao razumna opcija za upravljanje podacima, pokazat ćemo zbog čega takav način pohrane nije prikladan za stvarne web aplikacije. Ipak, valja naglasiti da određene web aplikacije, kako na poslužiteljskoj tako i na klijentskoj strani, podatke doista pohranjuju u datoteke. Pri tome je nužno biti oprezan: objasnit ćemo što je primjereno pohranjivati u datoteke, a što nije.

2.1 Čitanje tekstualnih datoteka kroz `fs` modul

Krenimo s primjerom **čitanja tekstualnih 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*).

Koristeći `fs` modul, možemo čitati različite formate tekstualnih datoteka, uključujući običan tekst, JSON, CSV, XML, HTML i dr.



> Slika 2: `fs` modul omogućuje rad s datotekama na poslužiteljskoj strani.

Definirajmo 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');
});
```

> Napomena, određeni *bundleri* (npr. Vite, Webpack, esbuild) omogućuju uvoz tekstualnih datoteka ili podataka direktno kroz `import` sintaksu. Navedeno nije zadano JavaScript ponašanje te zahtijeva dodatnu konfiguraciju tih bundlera.

Uključit ćemo i `fs` modul (nije ga potrebno ručno instalirati):

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

> Napomena: Više o asinkronom programiranju i konkurentnom izvođenju kôda učit ćete na kolegiju Raspodijeljeni sustavi na Diplomskom studiju. Za sada je dovoljno razumjeti osnovne koncepte asinkronog programiranja i kako ih primijeniti u praksi.

U praksi, **asinkrono programiranje** koristimo za izvođenje operacija koje zahtijevaju vremenski zahtjevne operacije (npr. dohvaćanje podataka s udaljenog poslužitelja). Međutim, pisanje i čitanje podataka iz datotečnog sustava također može biti vremenski zahtjevno, stoga je **preporučljivo koristiti asinkrone metode za pisanje i čitanje datoteka**. Zašto? Ukratko, ne želimo da naš poslužitelj čeka ili se "zamrzne" dok se datoteka čita ili piše ili dok se ne dovrši neka druga vremenski zahtjevna operacija.

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 (priča u prilogu na Merlinu/GitHubu). 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);
```

- `path` - relativna ili apsolutna putanja do datoteke (**obavezno**)
- `options` - specifikacija enkodiranja datoteke (opcionalno)
    - `encoding` - encoding datoteke (npr. `'utf8'`)
    - `flag` - opcionalni *char* 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 je nastala prilikom čitanja datoteke)
2. `data` - sadržaj datoteke (ako je sadržaj datoteke uspješno 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
});
```

`utf-8` encoding standard:

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. Danas je gotovo svaka web stranica, dokument ili programski kôd pohranjen u `utf-8` standardu.

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

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 (*eng. root*) direktorija datotečnog sustava**. Na primjer, u Unix/Linux sustavima, korijenski direktorij je `/`, dok je u Windows sustavima to najčešće `C:\`, ali može biti i neki drugi disk (npr. `D:\`, `E:\`, itd.) ovisno o konfiguraciji sustava.

Bash naredbom `pwd` (*print working directory*) možemo dobiti apsolutnu putanju do **trenutnog radnog direktorija** u kojem se nalazimo:

```
→ pwd

# Na Linux OS-u
/home/username/Documents/GitHub/WA4 - Pohrana podataka/app
# Na Windows OS-u:
C:\Users\Username\Documents\GitHub\WA4 - Pohrana podataka\app
# Na Mac OS-u:
/Users/username/Documents/GitHub/WA4 - Pohrana podataka/app
```

Apsolutna putanja uvijek **započinje s korijenskim direktorijem** i **sadrži sve direktorije i datoteke koje se nalaze između korijenskog direktorija i ciljane datoteke/direktorija**.

- `Apsolutna putanja` = `korijenski direktorij` + `svi direktoriji na putu` + `ciljna datoteka/direktorij`

Primjer apsolutne putanje do datoteke `story.txt`

```
# Na MacOS-u (Linux bi samo umjesto 'Users' imao 'home')
/Users/lukablaskovic/Github/FIPU-WA/WA4 - Upravljanje podacima i agregacija
podataka/app/data/story.txt

# Na Windows OS-u
C:\Users\LukaBlaskovic\Github\FIPU-WA\WA4 - Upravljanje podacima i agregacija
podataka\app\data\story.txt
```

> Uočite: Windows sustavi koriste `\` (backslash) kao **separator direktorija**, dok Unix/Linux sustavi koriste `/` (forward slash).

Datoteku `story.txt` možemo pročitati na sljedeći način koristeći apsolutnu putanju:

```
// apsolutna putanja do datoteke 'story.txt' na Windows OS-u pohranjena u string varijablu
u JavaScriptu
fs.readFile('C:\\Users\\Username\\Documents\\GitHub\\WA4 - Upravljanje podacima i
agregacija podataka\\data\\story.txt', 'utf8', (err, data) => {
    if (err) {
        console.error('Greška prilikom čitanja datoteke:', err);
        return;
    }

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

Navedeno je **loša praksa** budućći da je apsolutna putanja specifična za svakog korisnika i njegov datotečni sustav. Također, teško je čitljiva i često je podložna greškama kod ručnog unosa, pogotovo ako je putanja dugačka i sadrži razne direktorije.

Primjer: Kada bi netko klonirao ovaj repozitorij na svoje računalo i pokušao pokrenuti gornji kôd, došlo bi do greške budući da apsolutna putanja ne bi odgovarala njegovom datotečnom sustavu.

Osim toga, vidimo da smo u kôdu koristili dvostruke kose crte (`\\`) 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. Lista čestih *escape* znakova u JavaScriptu dostupna je ovdje.

> **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 (ipak, ovo je bolje raditi s `path` modulom). Ove varijable sadrži putanju do trenutnog direktorija u kojem se nalazi trenutni modul, npr. `index.js` u našem slučaju.

Primjer relativne putanje do datoteke `story.txt` ako se nalazimo u direktoriju: `/Users/lukablaskovic/Github/FIPU-WA/WA4 - Upravljanje podacima i agregacija podataka/app`:

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

Moramo biti oprezni prilikom pokretanja aplikacije iz različitih direktorija. Relativna putanja je relativna u **odnosu na trenutni radni direktorij** iz kojeg pokrećemo aplikaciju.

Na primjer, ako se datoteka `index.js` nalazi u direktoriju `app`, a datoteka `story.txt` u poddirektoriju `data` unutar istog direktorija `app`:

```
app <-- radni direktorij
├── data
│   └── story.txt <-- ciljna datoteka
├── index.js
├── node_modules
├── package-lock.json
└── package.json
```

Relativna putanja od `index.js` do datoteke `story.txt` bit će:

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

> Zapamti: Točkom `.` označavamo **trenutni direktorij**, a zatim nizom direktorija definiramo relativnu putanju do ciljne datoteke.

Međutim, ako se datoteka `story.txt` nalazi u direktoriju `data` koji se nalazi u direktoriju `WA4 - Upravljanje podacima i agregacija podataka`, a datoteka `index.js` u direktoriju `app`, tada struktura direktorija izgleda ovako:

```
WA4 - Upravljanje podacima i agregacija podataka
├── data
│   └── story.txt <-- ciljna datoteka
├── app <-- radni direktorij
    ├── index.js
    ├── node_modules
    ├── package-lock.json
    └── package.json
```

Tada će relativna putanja do datoteke `story.txt` (u odnosu na datoteku `index.js`) 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 i u kojem se radnom direktoriju nalazi instanca terminala kako bismo mogli koristiti relativne putanje bez problema. Trenutnu putanju u direktoriju možemo provjeriti koristeći `pwd` naredbu u terminalu.

Oznake `.` i `..` su vrlo korisne kod definiranja relativnih putanja, stoga ih je važno zapamtiti, a predstavljaju **pokazivače** na **trenutni** i **roditeljski** direktorij.

> Napomena: Pokazivače na roditeljski direktorij moguće je i ponavljati, kako bismo došli do željenog direktorija. Na primjer, `../../data/story.txt` označava da se iz trenutnog direktorija trebamo vratiti **dva direktorija unatrag** (u roditeljski direktorij roditeljskog direktorija), a zatim ući u direktorij `data` iz tog (*grandparent* direktorija) i pristupiti datoteci `story.txt`.

> Studenti koji žele ponoviti rad s datotekama i direktorijima u terminalu, preporučuje se skripta iz kolegija Operacijski sustavi: OS1 - Uvod u operacijske sustave.

Kako bi pokrenuli sljedeći kôd bez greške, odnosno kako bi se datoteka `story.txt` ispravno pročitala, **moramo terminal pozicionirati u direktorij gdje se nalazi** `index.js` datoteka; dakle unutar: `app` direktorija.

```
→ cd app
→ node index.js
```

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

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

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

```
→ cd .. # prebacujemo se u korijenski direktorij projekta
→ node app/index.js # upisujemo ispravnu putanju do index.js datoteke
# svejedno greška!
```

Preciznije, ako poslužitelj pokrećemo naredbom node `app/index.js`, datoteka `story.txt` neće biti pronađena. Unatoč tome, poslužitelj će se normalno pokrenuti.

Problem nije u relativnoj putanji koju smo koristili naredbom `node` za pokretanje poslužitelja, već u samoj relativnoj putanji do datoteke `story.txt` unutar `fs.readFile()` metode. Budući da se nalazimo u korijenskom direktoriju projekta, relativna putanja `./data/story.txt` traži datoteku `story.txt` unutar direktorija `WA4 - Upravljanje podacima i agregacija podataka/data/`, **koja ne postoji na toj lokaciji**.

```
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 na sljedeći način:

```
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/`, kôd će opet baciti grešku. Dakle, **relativne putanje ovise o trenutnom radnom direktoriju aktivne terminal sesije** (zato ih i nazivmo relativnima).

2.1.3 Modul `path` za upravljanje putanjama

Kako bismo minimizirali probleme s relativnim putanjama, možemo koristiti ugrađeni Node.js modul `path` koji nam omogućuje jednostavno upravljanje putanjama do datoteka i direktorija na ***platform-independent***.

`path` je također ugrađeni modul u Node.js, stoga ga nije potrebno ručno instalirati. Uključujemo ga na sljedeći način:

```
import path from 'path';
```

Koristeći `path` modul, možemo generirati apsolutnu putanju do datoteke `story.txt`, pomoću metode `path.join()` koja spaja različite dijelove putanje u jednu ispravnu putanju. Ovo je korisno zbog različitih operacijskih sustava koji koriste različite separatore direktorija (`/` vs `\`) ali i različite definicije korijenskog direktorija.

Sintaksa:

```
path.join([...paths])

# ili

path.join(path1, path2, ..., pathN)
```

- `paths` - niz dijelova putanje koje želimo spojiti u jednu ispravnu putanju

Primjer:

```
const storyPath = path.join(korijenski_direktorij, 'data', 'story.txt');
```

Rekli smo da `__dirname` varijabla sadrži apsolutnu putanju do **trenutnog** direktorija (ne korijenskog!). Ipak, u ES modulima nije dostupna, stoga možemo koristiti ekvivalent `import.meta.url` za dobivanje apsolutne putanje do trenutnog modula.

Sintaksa:

```
__dirname # nije dostupan u ES modulima
import.meta.url # dostupan u ES modulima
```

Dakle, sljedeće je greška:

```
const storyPath = path.join(import.meta.url, 'data', 'story.txt'); // Pogrešna putanja!
```

> Napomena: Ova funkcija ne vraća čisti Path string, već URL string koji počinje s `file:`. Ovo možemo riješiti korištenjem `fileURLToPath` funkcije iz `url` modula.

Ovakvo definirana putanja glasila bi: `file:/Users/lukablaskovic/Github/FIPU-WA/WA4%20-%20Upravljanje%20podacima%20i%20agregacija%20podataka/app/index.js/data/story.txt`

- `import.meta.url` vraća apsolutnu putanju do trenutnog modula, ali s prefiksom `file:` koji označava da se radi o datoteci na disku.
- Nadalje, nemojte da vas zbuni `%20` unutar putanje - to je URL enkodirani znak za razmak (space) budući da URL ne može sadržavati razmake.
- Konačno, pokušavamo dodati `/data/story.txt` na kraj putanje do `index.js` datoteke, što nije ispravno budući da `data` direktorij nije unutar `app/index.js` datoteke.

Direktorij gdje se nalazi određena (na danoj putanji) može se dobiti pomoću `path.dirname()` metode:

```
const currentDir = path.dirname(import.meta.url);
// Ispisuje: file:/Users/lukablaskovic/Github/FIPU-WA/WA4%20-%20Upravljanje%20podacima%20i%20agregacija%20podataka/app
```

Pretvorbu `file:` URL-a u čisti string putanje možemo napraviti pomoću `fileURLToPath` funkcije iz `url` modula:

Sintaksa:

```
import { fileURLToPath } from 'url';

const currentDirPath = fileURLToPath(currentDir);
// Ispisuje: /Users/lukablaskovic/Github/FIPU-WA/WA4 - Upravljanje podacima i agregacija podataka/app
```

Ovo sada možemo upotrijebiti za generiranje apsolutne putanje do datoteke `story.txt` na sljedeći način:

```
const storyPath = path.join(currentDirPath, 'data', 'story.txt');
// Ispisuje: /Users/lukablaskovic/Github/FIPU-WA/WA4 - Upravljanje podacima i agregacija podataka/app/data/story.txt
```

Ipak, našim mukama ovdje nije kraj. Ne postoji jedinstveni način kako dobiti **korijenski direktorij projekta** u Node.js aplikaciji, tako da ostaje problem definiranja apsolutne putanje do datoteke `story.txt` ako se nalazimo u različitim direktorijima.

Najbolje rješenje je samostalno definirati korijenski direktorij projekta, npr. pomoću `path.resolve()` metode koja vraća apsolutnu putanju do određenog direktorija, a ako ne definiramo argument, vraća apsolutnu putanju do trenutnog radnog direktorija.

Sintaksa:

```
path.resolve([...paths])

# ili
path.resolve(path1, path2, ..., pathN)

# ili samo

path.resolve() // vraća apsolutnu putanju do trenutnog radnog direktorija
```

Primjer:

```
const rootDir = path.resolve(); // apsolutna putanja do trenutnog radnog direktorija
const storyPath = path.join(rootDir, 'data', 'story.txt');
```

Ako bismo htijeli dobiti direktorij iznad trenutnog radnog direktorija, možemo koristiti `..` pokazivač:

```
const parentDir = path.resolve('..'); // apsolutna putanja do roditeljskog direktorija (u odnosu na trenutni radni direktorij)
const storyPath = path.join(parentDir, 'app', 'data', 'story.txt'); // ali onda ovdje dodajemo i 'app' direktorij
```

Ako se prebacimo u direktorij `/Users/lukablaskovic/Github/FIPU-WA/WA4 - Upravljanje podacima i agregacija podataka`, i pokušamo pokrenuti poslužitelj naredbom `node app/index.js`, **ponovno dobivamo grešku** budući da naredba `path.resolve('..')` računa putanju iznad trenutnog radnog direktorija, a ne iznad direktorija gdje se nalazi `index.js` datoteka.

Zaključno: treba pripaziti u radu s relativnim i apsolutnim putanjama do datoteka, te koristiti `path` modul kako bismo minimizirali probleme s različitim operacijskim sustavima. **Relativna putanja** relativna je u odnosu na **trenutni radni direktorij** iz kojeg pokrećemo aplikaciju pa trebamo biti oprezni gdje se nalazimo u datotečnom sustavu prilikom pokretanja aplikacije, dok je **apsolutna putanja** uvijek ista bez obzira na trenutni radni direktorij - ali može stvoriti probleme prilikom pokretanja *developerskih* okruženja na različitim računalima.

2.1.4 *Callback* vs *Promise* pristup asinkronom programiranju

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.

> Preporuka je ponoviti koncept asinkronog programiranja: PJS5 - DOM, JSON i Asinkrono programiranje - 3. poglavlje u skripti.

```
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 kôd 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 kôd koji se odnosi na čitanje datoteke u zasebnu funkciju `read_story()`, a zatim ćemo definirati endpoint `/story` koja će slati JSON odgovor rezultat poziva ove funkcije natrag korisniku. 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 iako na prvi pogled izgleda ispravno!**

```
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 vrijeme. Jednom 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 kôd u nastavku te funkcija `read_story()` je već vratila prazan string.

Mi ustvari ovdje pokušavamo upravljati asinkronim kôdom na sinkroni način, što je pogrešno.

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 poziv funkcije `read_story()` spremiti u varijablu `data`, a zatim poslati kao odgovor klijentu.

```
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 `read_story()` prije nego što se datoteka pročita. Funkcija `readFile` ne vraća sadržaj datoteke te ju ne možemo pohraniti u varijablu na ovaj 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" (*non-blocking IO*), 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 izmijeniti i primjer s čitanjem datoteke `story.txt` koristeći *asinkroni callback* pattern.

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, kôd 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 kôd 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 kôd.

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
    }
});
```

Kôd 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 biblioteke `fs`:

```
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 pogledate 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 - ne zamjenjuje se postojeći sadržaj.

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 zadane 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):

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()`.

> Napomena: Koncepti serijalizacije i deserijalizacije podataka primjenjuju se i izvan JavaScripta te su ključni za razumijevanje razmjene podataka između različitih sustava putem mreže. Primjerice, u programskom jeziku Python serijalizacija u JSON format izvodi se pomoću ugrađenog modula `json`. Tijekom deserijalizacije JSON objekt pretvara se u Python rječnik (*dictionary*), dok se JSON polja prevode u Python liste (*list*).

```
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 kôd iznad `POST` metodom, dok ćemo JSON direktno preuzeti iz tijela HTTP 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 kôd 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 uočite 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 kôd 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 kôd, gdje moramo prvo čitati, a nakon tog dodavati, serijalizirati i pohranjivati objekte. Stvari možemo pojednostaviti još jednom ekstenzijom - `fs-extra`. Ova ekstenzija nudi mnoge korisne metode koje olakšavaju rad s datotekama, uključujući gotove metode za čitanje i pisanje JSON podataka.

Modul `fs-extra` možemo instalirati kroz npm:

```
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 {
        // pročitaj datoteku, deserijaliziraj JSON podatke i pohrani u varijablu
        const students = await fs.readJson('data/data.json');
        students.push(student);
        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 kôd i izbjeći ručno čitanje i pisanje JSON podataka, odnosno automatiziramo procese serijalizacije i deserijalizacije.

Rekapitulacija:

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 na prvu pomisao tako 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 sl.

Problemi **skalabilnosti** su očiti. Što ako 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, sigurnije i učinkovitije 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 (*eng. overwrite*), ne završe u nekom nevaljalom stanju (*eng. corrupted data state*)?

> 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 učinkovit način, uz osiguranje konzistentnosti i integriteta podataka. O DBMS sustavima i relacijskim bazama podataka detaljno ste učili na kolegijima Baze podataka 1 i Baze podataka 2.

Ipak, ponekad struktura baze podataka koju koristi naša aplikacija jednostavno nije primjerena za pohranu određenih vrsta podataka. Primjerice, konfiguracijske datoteke, datoteke s logovima, *cache* podaci, datoteke s privremenim podacima i druge slične datoteke kratkog životnog ciklusa. **U takvim slučajevima, korištenje datoteka može biti opravdano i praktično rješenje**.

Ono što ne želite nikad spremati u datoteke su **osjetljivi podaci** poput lozinka, osobnih podataka korisnika, financijskih informacija i sl. Datoteke ne nude gotovo nikakvu razinu sigurnosti i zaštite podataka koje nude moderne baze podataka. Također, podatkovne strukture s jasno definiranom shemom i relacijama između podataka su puno bolje podržane u bazama podataka nego u datotekama.

Što se tiče **binarnih datoteka** (slike, videozapisi, audio zapisi i sl.), postoje specijalizirani sustavi za pohranu i upravljanje takvim vrstama podataka, poput **sustava za upravljanje sadržajem** (*eng. Content Management Systems - CMS*) ili **sustava za pohranu objekata** (*eng. Object Storage Systems*). Ovi sustavi su optimizirani za rukovanje velikim količinama binarnih podataka i nude značajke poput verzioniranja, replikacije, sigurnosnih kopija i skalabilnosti koje nisu dostupne u običnim datotekama. Ipak, njihova implementacija jest složenija, a deployment u stvarno okruženje skuplji - pohrana ovakovog sadržaja na poslužitelju može biti privremeno rješenje, ali nije dobro dugoročno rješenje.

3. Agregacija podataka kroz parametre upita (Query Parameters)

Ipak, prije nego se krenemo baviti bazom podataka (na sljedećim vježbama), moramo naučiti kako agregirati podatke na poslužiteljskoj strani kroz **parametre upita**, poznatije kao ***query parameters***.

Query ili *search* parametri su dio URL-a koji služi za prenošenje **dodatnih informacija o resursu** koji se traži ili ponekad o **radnji koju je potrebno izvršiti**. *query* parametri se dodaju na URL nakon znaka `?` i odvajaju se znakom `&`. Svaki *query* parametar sastoji se od imena i vrijednosti, odvojenih znakom `=`.

Sigurno smo svi bar jednom vidjeli URL s *query* parametrom, npr.:

```
https://www.youtube.com/watch?v=dQw4w9WgXcQ
```

- **query parametar** je ovdje `v`, a vrijednost `dQw4w9WgXcQ` predstavlja jedinstveni identifikator videa na

YouTube platformi.

Ipak, zadnjih godina YouTube je počeo koristiti i skraćenu domenu koja video identificira samo kroz **route parametar**:

```
https://youtu.be/dQw4w9WgXcQ
```

Query parametri su često korišteni za **filtriranje**, **sortiranje**, **paginaciju** i druge operacije nad podacima koje se dohvaćaju s poslužitelja. Međutim, kao što vidite iz primjera YouTube-a, query parametri se mogu koristiti i za **identifikaciju resursa**.

Query parametri nisu obavezni dio URL-a, za razliku od **parametara rute** koji su definirani unutar same rute (npr. `/users/:userId`).

Sintaksa:

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

http://localhost:3000/route?key1=value1&key2=value2
```

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** (*eng. subset*), npr. filtrirati po nekom kriteriju, sortirati, paginirati stranice i sl.

3.1 Query parametri: 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:

```
// data/students.json
[
    { "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`?

U tom slučaju **ne želimo raditi novi endpoint**, već možemo nadograditi postojeći koristeći *query* parametre.

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

Ključ (uz `?`) nam ovdje može biti `fakultet`, a vrijednost (uz `=`) `FIPU`.

Primjer:

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

Glavnina URL-a ostaje ista, samo dodajemo *query* parametar `fakultet` s vrijednošću `FIPU`.

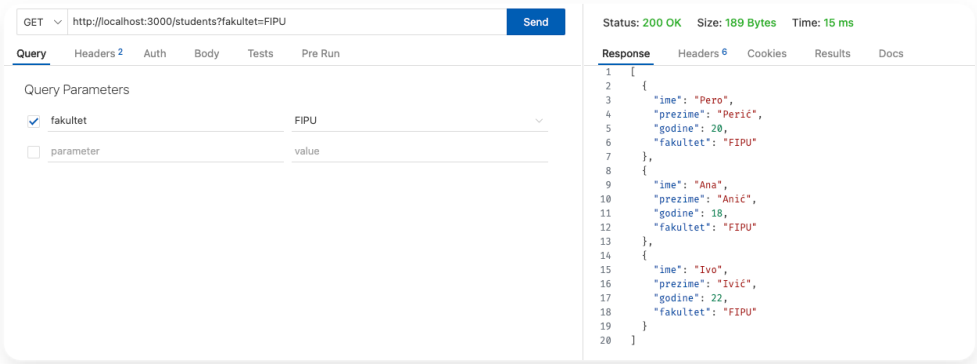
> **Uočite**: `req.query` je objekt koji sadrži sve ***query* parametre** poslane u URL-u. Nemojte ovo miješati s `req.params` objektom koji predstavlja drugu vrstu parametara - **parametre rute**.

```
// u definiciji endpointa NIŠTA NE MIJENJAMO!
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 metodu `Array.filter()` nad poljem studenata:

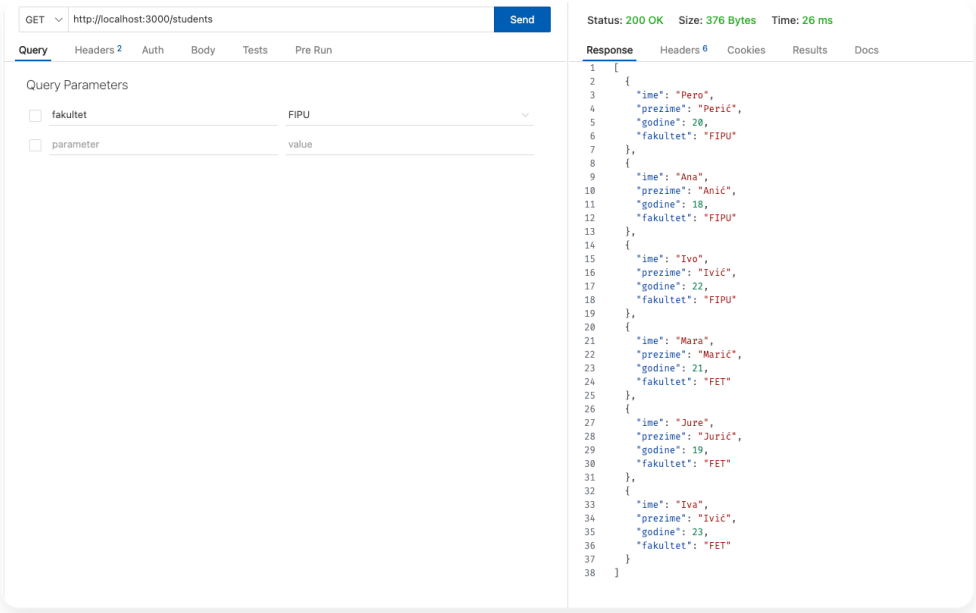
```
app.get('/students', async (req, res) => {
    let fakultet_query = req.query.fakultet; // dohvatimo query parametar 'fakultet'
    try {
        const data = await fs.readFile('data/students.json', 'utf8');
        const students = JSON.parse(data);

        // ako postoji query parametar 'fakultet', filtriraj studente
        if (fakultet_query) {
            const filtered_students = students.filter(student => student.fakultet ===
fakultet_query);
            res.status(200).send(filtered_students);
            // ako ne postoji query parametar, vrati sve studente
        } 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 Postman. HTTP klijenti nude opciju unosa query parametara kao *ključ-vrijednost* parova pa ih možemo unijeti i na taj način ili direktnim unosom u URL string.



> Slika 3: Filtriranje studenata po fakultetu kroz *query* parametar `fakultet` u Postmanu. Uočite kako Postman automatski detektira query parametre pod karticom zahtjeva *query*.

Ako uklonimo *query* parametar, dobit ćemo sve studente.



> Slika 4: Uklanjanjem *query* parametra, endpoint vraća 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.

> **Važna napomena**: Prema *restful* API principima, query parametri se koriste prvenstveno za **agregaciju podataka** (filtriranje, sortiranje, paginacija i sl.). U pravilu ne želimo koristiti *query* parametre za identifikaciju pojedinog resursa (za to smo rekli da koristimo parametre rute). Zamislite poslužitelj koji vraća podatke telefonskog imenika, parametar rute bi ovdje mogao biti **broj telefona** (`/imenik/:broj_telefona`) - vraća jedan zapis. Međutim, **prezime** može biti *query* parametar koji nam može poslužiti za sažimanje rezultata prema prezimenu (filtriranje), npr. `/imenik?prezime=Horvat` - vraća više zapisa.

Recimo, želimo studente s fakulteta `FIPU` i koji imaju `20` godina:

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

- *query* parametar `fakultet` ima vrijednost `FIPU` (`?fakultet=FIPU`)
- *query* parametar `godine` ima vrijednost `20` (`&godine=20`)

Sada moramo uzeti u obzir četiri različita slučaja:

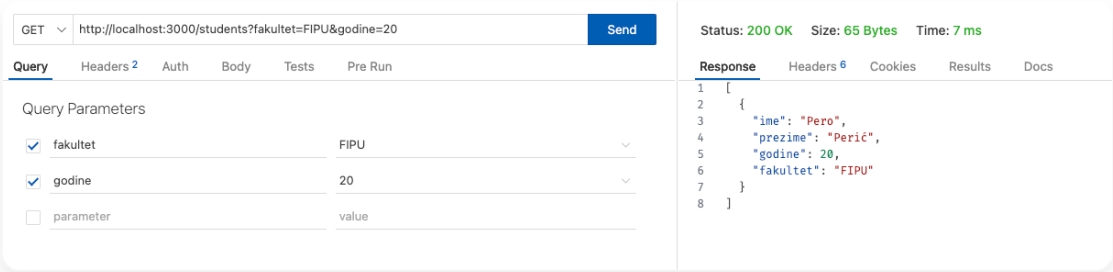
1. Kada su prisutni oba *query* parametra (`fakultet` i `godine`), npr. `/students?fakultet=FIPU&godine=20`
2. Kada je prisutan samo `fakultet` parametar, npr. `/students?fakultet=FET`
3. Kada je prisutan samo `godine` parametar, npr. `/students?godine=22`
4. Kada nema *query* parametara

Endpoint *callback* mora ispravno obraditi svaki od navedenih HTTP zahtjeva.

U kôdu moramo samo dohvatiti dodatni parametar i nadograditi `Array.filter` *callback* funkciju:

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

```
    let godine_query = req.query.godine;

    try {
        const data = await fs.readFile('data/students.json', 'utf8');
        const students = JSON.parse(data);
        // slučaj 1: oba query parametra su prisutna
        if (fakultet_query && godine_query) {
            const filtered_students = students.filter(student => student.fakultet ===
fakultet_query && student.godine === parseInt(godine_query));
            return res.status(200).send(filtered_students);
            // slučaj 2: prisutan je samo fakultet query parametar
        } else if (fakultet_query) {
            const filtered_students = students.filter(student => student.fakultet ===
fakultet_query);
            return res.status(200).send(filtered_students);
            // slučaj 3: prisutan je samo godine query parametar
        } else if (godine_query) {
            const filtered_students = students.filter(student => student.godine ===
parseInt(godine_query));
            return res.status(200).send(filtered_students);
            // slučaj 4: nema query parametara
        } else {
            return res.status(200).send(students);
        }
    } catch (error) {
        console.error('Greška prilikom čitanja datoteke:', error);
        res.status(500).send('Greška prilikom čitanja datoteke.');
    }
});
```



> Slika 5: Filtriranje studenata s dva query parametra: `fakultet` i `godine`

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, ali možemo ih slati i više ili nijedan.
- *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.

Prilikom izgradnje poslužitelja, uobičajena praksa je prvo definirati definicija ruta s eventualnim parametrima rute, a zatim ih nadograđivati s *query* parametrima za agregaciju podataka.

3.2 Query parametri: 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`.

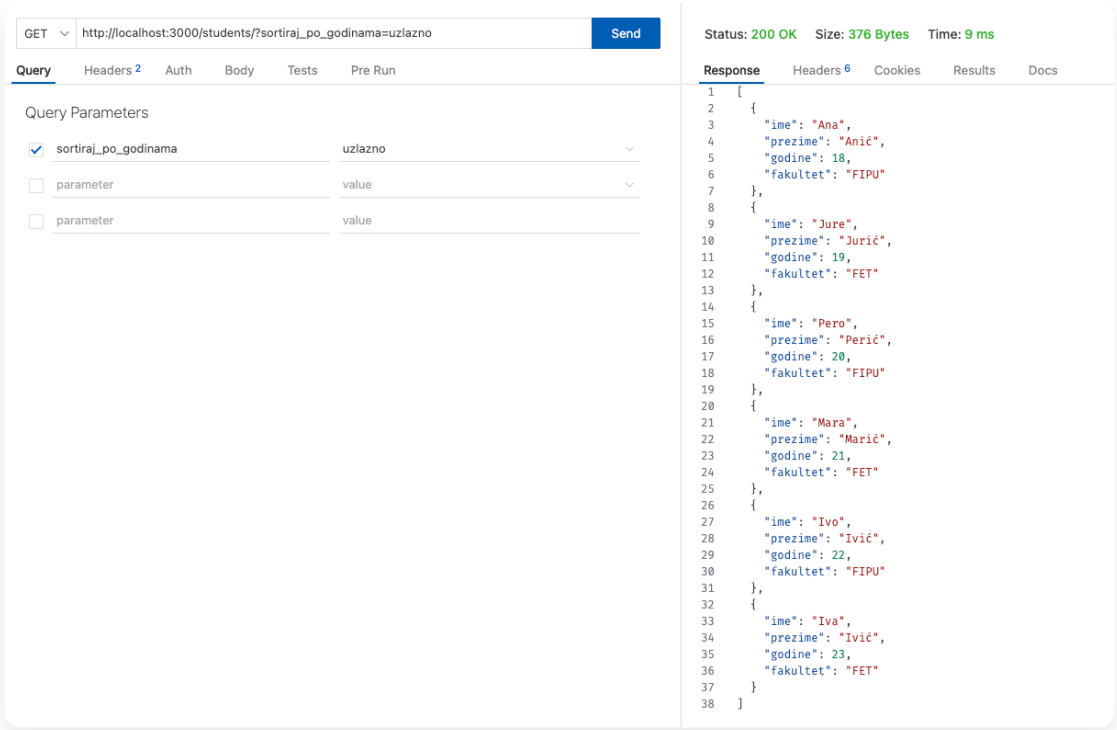
```
http://localhost:3000/students?sortiraj_po_godinama=uzlazno
```

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

Radi jednostavnosti, izostavit ćemo logiku za filtriranje (iako ona može i ostati):

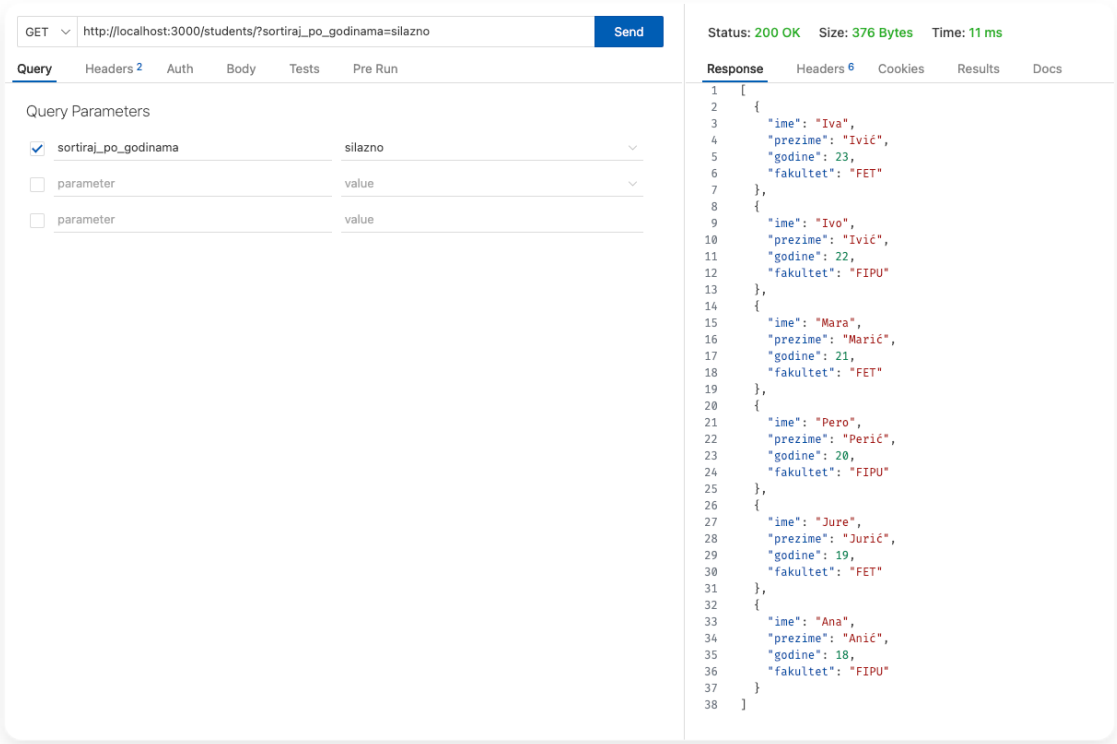
```
app.get('/students', async (req, res) => {
    let sortiraj_po_godinama = req.query.sortiraj_po_godinama; // dohvatimo vrijednost
query parametar 'sortiraj_po_godinama'
    try {
        const data = await fs.readFile('data/students.json', 'utf8');
        const students = JSON.parse(data);
        // ako je prisutan query parametar 'sortiraj_po_godinama', sortiraj studente
        if (sortiraj_po_godinama) {
            if (sortiraj_po_godinama === 'uzlazno') {
                // sortiranje uzlazno: od najmanjeg prema najvećem
                students.sort((a, b) => a.godine - b.godine);
                // sortiranje silazno: od najvećeg prema najmanjem
            } 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**:



> Slika 6: Sortiranje studenata po godinama uzlazno kroz *query* parametar u Postmanu

Sortiranje po godinama **silazno**:



> Slika 7: Sortiranje studenata po godinama silazno kroz *query* parametar u Postmanu

3.3 Kombiniranje parametra rute i query parametara

Kao što smo rekli, *route* i *query* parametri se mogu kombinirati. Navedeno ćemo prikazati na dva primjera:

1. Zamislite da izrađujemo poslužitelj za aplikaciju za naručivanje hrane. Dva resursa mogu biti `restorani` i `meni`.

- Rutu za dohvat svih restorana možemo definirati kao `/restorani`
- Rutu za dohvat menija određenog restorana možemo definirati kao `/restorani/:restoranId/meni`

Primjer:

```
app.get("/restorani", (req, res) => {
    return res.json(restorani);
});

app.get("/restorani/:restoranId/meni", (req, res) => {
    const restoranId = req.params.restoranId;

    if (!restoranId) {
        return res.status(400).send('Niste poslali restoranId parametar.');
    }

    const meni = dohvatiMeniZaRestoran(restoranId);

    if (!meni) {
        return res.status(404).send('Restoran nije pronađen ili nema meni.');
    }

    return res.json(meni);
```

Parametar rute možemo kombinirati s *query* parametrima za agregaciju podataka. Recimo, želimo filtrirati meni prema vrsti jela (predjelo, glavno jelo, desert). U tom slučaju možemo definirati *query* parametar `vrsta_jela`.

```
app.get('/restorani/:restoranId/meni', (req, res) => {
    const restoranId = req.params.restoranId; // dohvatimo parametar rute 'restoranId'
    const vrsta_jela = req.query.vrsta_jela; // dohvatimo query parametar 'vrsta_jela'

    if (!restoranId) {
        return res.status(400).send('Niste poslali restoranId parametar.');
    }

    let meni = dohvatiMeniZaRestoran(restoranId);

    if (!meni) {
        return res.status(404).send('Restoran nije pronađen ili nema meni.');
    }

    // ako je prisutan query parametar 'vrsta_jela', filtriraj meni
    if (vrsta_jela) {
        meni = meni.filter(jelo => jelo.vrsta === vrsta_jela);
    }

    return res.json(meni);
});
```

Primjer HTTP poziva za endpoint iznad:

- `/restorani/12`: dohvatit će cijeli meni restorana s ID-jem `12`
- `/restorani/12/meni?vrsta_jela=desert`: dohvatit će samo deserte iz menija restorana s ID-jem `12`
- `/restorani/12/meni?vrsta_jela=glavno`: dohvatit će samo glavna jela iz menija restorana s ID-jem `12`

2. Drugi primjer može biti poslužitelj za upravljanje narudžbama gdje imamo rutu za dohvat svih narudžbi `/narudzbe` i route parametar koji dohvaća pojedinu narudžbu po ID-u `/narudzbe/:narudzbaId`.

- Rutu za dohvat svih narudžbi možemo definirati kao `/narudzbe`
- Rutu za dohvat pojedine narudžbe možemo definirati kao `/narudzbe/:narudzbaId`

```
app.get('/narudzbe', (req, res) => {
    return res.json(narudzbe);
});

app.get('/narudzbe/:narudzbaId', (req, res) => {
    const narudzbaId = req.params.narudzbaId;

    if (!narudzbaId) {
        return res.status(400).send('Niste poslali narudzbaId parametar.');
    }

    const narudzba = dohvatiNarudzbuPoId(narudzbaId);

    if (!narudzba) {
        return res.status(404).send('Narudžba nije pronađena.');
    }

    return res.json(narudzba);
});
```

Recimo da želimo dohvatiti određene elemente narudžbe između ponuđenih, npr. samo `artikli` ili samo `cijena`, ili `artikli`, `cijena`, `datum` i `nacin_placanja`. Navedene stavke možemo izlistati kroz *query* parametar `sadrzi`.

```
app.get('/narudzbe/:narudzbaId', (req, res) => {
    const narudzbaId = req.params.narudzbaId;
    const sadrzi = req.query.sadrzi; // dohvatimo query parametar 'sadrzi'

    if (!narudzbaId) {
        return res.status(400).send('Niste poslali narudzbaId parametar.');
    }

    let narudzba = dohvatiNarudzbuPoId(narudzbaId);

    if (!narudzba) {
```

```
        return res.status(404).send('Narudžba nije pronađena.');
    }

    // ako je prisutan query parametar 'sadrzi', filtriraj narudžbu
    if (sadrzi) {
        const kljucevi = sadrzi.split(','); // podijeli vrijednost na ključeve
        narudzba = kljucevi.reduce((filtered, key) => {
            if (narudzba.hasOwnProperty(key)) {
                filtered[key] = narudzba[key];
            }
            return filtered;
        }, {});
    }

    return res.json(narudzba);
});
```

Primjer HTTP poziva za endpoint iznad:

- `/narudzbe/45`: dohvatit će cijelu narudžbu s ID-jem `45`
- `/narudzbe/45?sadrzi=artikli,cijena`: dohvatit će samo artikle i cijenu iz narudžbe s ID-jem `45`
- `/narudzbe/45?sadrzi=datum,nacin_placanja`: dohvatit će samo datum i način plaćanja iz narudžbe s ID-jem `45`

Samostalni zadatak za Vježbu 4

Izradite novi Express poslužitelj i definirajte jednostavni *restful 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