

Web aplikacije (WA)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



#1

WA

(1) Uvod u HTTP, Node i Express

Web aplikacije su sofisticirana programska rješenja koja se pokreću na web poslužitelju, a korisnici im pristupaju putem web preglednika. Njihova najveća prednost je široka dostupnost na gotovo svim platformama i uređajima, bez potrebe za lokalnom instalacijom. Ovaj kolegij usmjerjen je na dizajn i razvoj web aplikacija korištenjem modernih tehnologija i alata. Za razliku od kolegija Programsko inženjerstvo, ovdje ćete naučiti kako implementirati poslužiteljski sloj web aplikacije – ključni dio koji možemo zamisliti kao "mozak" aplikacije, zadužen za logiku i obradu podataka.

Posljednje ažurirano: 23.10.2024.

Sadržaj

- [Web aplikacije \(WA\)](#)
- [\(1\) Uvod u HTTP, Node i Express](#)
 - [Sadržaj](#)
- [1. Uvod](#)
 - [1.1 Kratak povijesni pregled](#)
- [2. Instalacija potrebnih alata](#)
 - [2.1 Node.js](#)
 - [2.2 VS Code](#)
 - [2.3 Git](#)
- [3. Kako započeti novi projekt?](#)
 - [3.1 Inicijalizacija novog repozitorija](#)
 - [3.2 Izrada Node projekta](#)
- [4. Postavljanje osnovnog Express poslužitelja](#)
 - [4.1 Instalacija Express.js](#)

- [4.2 Osnovni Express.js poslužitelj](#)
- [4.3 Kako definirati osnovni endpoint?](#)
- [4.4 Nodemon](#)
- [4.5 Git commit](#)
- [5. HTTP protokol](#)
 - [5.1 HTTP zahtjev \(eng. HTTP request\)](#)
 - [5.1.1 Obavezni dijelovi HTTP zahtjeva](#)
 - [Vježba 1 - HTTP zahtjev prema našem Expressu](#)
 - [5.1.2 Opcionalni dijelovi HTTP zahtjeva](#)
 - [5.2 HTTP odgovor \(eng. HTTP response\)](#)
 - [5.2.1 Obavezni dijelovi HTTP odgovora](#)
 - [5.2.2 Opcionalni dijelovi HTTP odgovora](#)
 - [Vježba 2: Kako vidjeti cijeli HTTP odgovor?](#)
 - [Samostalni zadatak za Vježbu 1](#)

1. Uvod

Web aplikacije su softverski programi koji se koriste putem internetskog preglednika, bez potrebe za instalacijom na korisničkom računalu ili uređaju. One se izvršavaju na web poslužitelju, a korisnici im pristupaju putem internetskog preglednika. Web aplikacije su dostupne na gotovo svim platformama i uređajima, što ih čini vrlo popularnim među korisnicima.

Primjeri web aplikacija:

- Gmail: web aplikacija za slanje i primanje e-pošte
- Google Docs: omogućuje stvaranje i uređivanje dokumenata u stvarnom vremenu
- Facebook: društvena mreža za povezivanje s prijateljima i obitelji
- Online trgovine: web shopovi poput Amazona ili Ebaya za kupnju proizvoda

Svaka web aplikacija sastoji se od minimalno dva dijela:

1. **Klijentski dio** (*eng. client side*): izvršava se na korisničkom uređaju (npr. računalo, pametni telefon) i koristi se za prikaz korisničkog sučelja. Napisan je u jezicima poput HTML-a, CSS-a i JavaScripta, odnosno razvojnim okvirima poput Reacta, Angulara ili Vue.js-a.
2. **Poslužiteljski dio** (*eng. server side*): izvršava se na web poslužitelju i koristi se za obradu zahtjeva korisnika, komunikaciji s bazom podataka i definiranje poslovne logike aplikacije. Napisan je u jezicima poput JavaScripta (Node.js), Pythona (Django, Flask), Rubyja (Ruby on Rails) ili Java (Spring).

1.1 Kratak povijesni pregled

Premda nije predmet ovog kolegija, **PHP** je također popularan jezik za izradu poslužiteljskog dijela web aplikacija. Glavna prednost PHP-a je generiranje dinamičkih HTML stranica na poslužiteljskoj strani, što ga čini idealnim za izradu web stranica i aplikacija. Iako je prisutan već dugi niz godina, PHP i dalje ima veliku bazu korisnika i popularan je izbor za izradu web aplikacija (posebice moderni PHP okviri poput Laravela i Symfonyja).

JavaScript je nešto mlađi programski jezik od PHP-a (svega nekoliko mjeseci), a prvi put je implementiran u Netscape Navigatoru (najpopularniji web preglednik u to vrijeme) 1995. godine. JavaScript je postao popularan zbog svoje sposobnosti stvaranja interaktivnih korisničkih sučelja na klijentskoj strani web aplikacija, što je dovelo do razvoja modernih web aplikacija poput Gmaila i Google Mapsa. Te aplikacije su imale interaktivno korisničko sučelje napisano u JavaScriptu, dok se se za poslužiteljski sloj koristili PHP i C++ jezici.

Danas gotovo 99% web stranica koristi JavaScript na klijentskoj strani za implementaciju interaktivnog ponašanja, a svaki moderni web preglednik ima ugrađen svoj JavaScript engine koji izvršava JavaScript kod.

Međutim, 2009. godine na tržište izlazi novi revolucionarni alat koji je promijenio način na koji se danas razvijaju moderne web aplikacije - **Node.js**. Node.js je JavaScript okruženje (*eng. runtime environment*) koje dozvoljava izvođenje JavaScript koda na poslužiteljskoj strani. Drugim riječima, Node.js omogućava izvršavanje JavaScript koda izvan web preglednika.

Mi ćemo se na ovom kolegiju fokusirati upravo na taj poslužiteljski sloj web aplikacija, koristeći Node.js, odnosno biblioteku **Express.js** za izradu poslužiteljskog dijela web aplikacija.



A za one koji žele više, proučite [Deno](#) - novi JavaScript *runtime environment* koji brzo dobiva na popularnosti, a razvija ga isti programer koji je razvio Node.js!

2. Instalacija potrebnih alata

2.1 Node.js

Node.js možete preuzeti sa [službene stranice](#). Preuzmite LTS verziju (Long Term Support) koja je stabilna i sigurna za produkciju. Nakon preuzimanja, pokrenite instalacijski program i slijedite upute za instalaciju.

Preporuka je preuzeti verziju LTS 20+.

Nakon što ste uspješno instalirali Node.js, možete provjeriti je li instalacija uspješna tako da otvorite terminal i upišete sljedeću naredbu:

```
node -v
```

Ako je instalacija uspješna, trebali biste vidjeti verziju Node.js-a koju ste instalirali. Na primjer:

```
v22.9.0
```

Instalacijom Node-a dobivate i **npm** (*Node Package Manager*) koji koristimo za instalaciju paketa i modula potrebnih za razvoj web aplikacija.

```
npm -v
```

2.2 VS Code

Visual Studio Code je besplatni uređivač koda koji je dostupan za Windows, macOS i Linux. Možete preuzeti Visual Studio Code sa [službene stranice](#). Nakon preuzimanja, pokrenite instalacijski program i slijedite upute za instalaciju.

Nakon što ste uspješno instalirali Visual Studio Code, možete provjeriti je li instalacija uspješna tako da otvorite terminal i upišete sljedeću naredbu:

```
code --version
```

Možete pokrenuti Visual Studio Code tako da upišete sljedeću naredbu:

```
code
```

ili jednostavno pokrenite kroz grafičko sučelje.

2.3 Git

Git je besplatni sustav za upravljanje izvornim kodom koji je dostupan za Windows, macOS i Linux. Možete preuzeti Git sa [službene stranice](#). Nakon preuzimanja, pokrenite instalacijski program i slijedite upute za instalaciju.

Iako nije nužan za sam razvoj web aplikacija, Git je koristan alat koji ćemo često koristiti za verzioniranje izvornog koda.

Nakon što ste uspješno instalirali Git, možete provjeriti je li instalacija uspješna tako da otvorite terminal i upišete sljedeću naredbu:

```
git --version
```

Ako je instalacija uspješna, trebali biste vidjeti verziju Git-a koju ste instalirali. Na primjer:

```
git version 2.47.0
```

Ako još uvijek nemate, svakako morate izraditi i [Github](#) račun. GitHub je vrlo popularna platforma gdje developeri mogu pohranjivati, dijeliti te surađivati na kodu i projektima, a bazira se na Gitu, kao pozadinskom sustavu za verzioniranje koda.

3. Kako započeti novi projekt?

Nakon što ste uspješno instalirali Node.js, Visual Studio Code i Git, možete započeti raditi na novom projektu.

3.1 Inicijalizacija novog rezitorija

Prvi korak je definiranje strukture projekta, budući da smo odlučili verzionirati izvorni kod, koristit ćemo Git za inicijalizaciju novog repozitorija. Međutim krenut ćemo od GitHuba: idemo na [Github izraditi novi repozitorij](#), a zatim ćemo ga klonirati na naše računalo. Klonirati (eng. *clone*) znači preuzeti udaljeni repozitorij na naše računalo (lokalno).

Nazovite repozitorij "**wa_vjezbe_01**" i dodajte opis po želji. Možete ga postaviti kao privatni ili javni, a svakako odaberite opciju "**Add a README file**" kako ne bi inicijalno bio prazan.

Nakon što ste izradili repozitorij, kopirajte URL repozitorija, npr. <https://github.com/lukablaskovic/wa-vjezbe-01.git>

1. Način (terminal)

Otvorite terminal i navigirajte do direktorija u kojem želite spremiti projekt. Zatim upišite sljedeću naredbu:

```
cd putanja/do/direktorija  
git clone <URL>
```

Na primjer, ako se direktorij nalazi na radnoj površini, naredba bi mogla izgledati ovako:

Windows:

```
cd C:\Users\<VAŠ USERNAME>\Desktop
```

macOS/linux:

```
cd Desktop
```

Zamijenite **<URL>** s URL-om repozitorija koji ste prethodno kopirali. Na primjer:

```
git clone https://github.com/lukablaskovic/wa-vjezbe-01.git
```

Kako biste se uvjerili da ste u pravom direktoriju, upišite naredbu:

Windows:

```
ls
```

ili

```
dir
```

macOS/linux:

```
ls
```

Ove naredbe će vam ispisati popis datoteka i direktorija u trenutnom direktoriju.

2. Način (VS Code)

Drugi način je kloniranje repozitorija direktno iz Visual Studio Codea. Otvorite Visual Studio Code i pritisnite `Ctrl + Shift + P` (Windows) ili `Cmd + Shift + P` (macOS) kako biste otvorili *Command Palette*. Upišite `Git: Clone` i pritisnite `Enter`. Zatim unesite URL repozitorija i pritisnite `Enter`.

Ako vam ne radi, uvjerite se da imate instaliran Git i da je dostupan u PATH-u. Dodatno, u VS Codeu morate biti prijavljeni na GitHub račun.

Možete se uvjeriti da je Git dostupan u PATH-u tako da otvorite terminal i upišete:

```
git --version
```

Ako nije, dobit ćete grešku neovisno o okruženju u kojem otvarate terminal. U tom slučaju, potrebno je reinstalirati Git kroz instalacijski program i odabratи opciju koja dodaje Git u PATH.

3. Način (Github Desktop)

Treći način je kloniranje repozitorija direktno iz GitHub Desktop aplikacije. Otvorite [Github Desktop](#) aplikaciju i pritisnite `Ctrl + Shift + O` (Windows) ili `Cmd + Shift + O` (macOS) kako biste otvorili *Clone Repository* prozor. Upišite URL repozitorija i pritisnite *Clone*.

GitHub desktop je odlična aplikacija za početnike jer nudi jednostavan način za upravljanje repozitorijima, ali nije nužna za rad na projektu. Sve što možete napraviti u GitHub Desktopu možete napraviti i u terminalu ili Visual Studio Codeu, ali Desktop nudi vizualni prikaz promjena i jednostavno upravljanje repozitorijima, promjenama, *branchevima*, što može biti vrlo korisno.

3.2 Izrada Node projekta

Jednom kad ste uspješno klonirali repozitorij, možete započeti s izradom Node projekta.

Otvorite terminal i navigirajte do direktorija projekta. Zatim upišite sljedeću naredbu:

```
code .
```

ili otvorite Visual Studio Code i navigirajte do direktorija projekta.

Možete i kroz GitHub Desktop i to tako da otvorite repozitorij u aplikaciji i pritisnete `Ctrl + Shift + A` (Windows) ili `Cmd + Shift + A` (macOS) kako biste otvorili repozitorij u Visual Studio Codeu.

Kada ste otvorili projekt u Visual Studio Codeu, otvorite novi terminal: `Terminal -> New Terminal`.

Zatim upišite sljedeću naredbu:

```
npm init
```

Ova naredba pokreće inicijalizaciju novog Node projekta. Slijedite upute i unesite podatke o projektu. Ako želite preskočiti neko polje, jednostavno pritisnite `Enter`.

Ako želite preskočiti cijeli upitnik i koristiti zadane postavke, dodajte `-y` opciju:

```
npm init -y
```

Wohoo! Uspješno ste inicijalizirali novi Node projekt! 🎉

Primjetit ćete da se u direktoriju projekta pojavila datoteka `package.json`. Ova datoteka sadrži informacije o projektu, poput naziva, verzije, autora, skripti i ovisnosti. Kroz kolegij ćemo detaljno vidjeti što znači svaki dio `package.json` datoteke i kako ju možemo koristiti za upravljanje projektom.

Struktura direktorija projekta trebala bi izgledati ovako (samo 1 datoteka):

```
.
```

```
└── package.json
```

```
1 directory, 1 file
```

Idemo još malo ponoviti terminal: Konzola mi je nakrcana nakon ove inicijalizacije, kako da očistim? `clear` ili `cls` (Windows) | `clear` (macOS/linux) i sve će biti čisto. 

Kako se mogu kretati kroz direktorije? `cd ime_direktorija` za ulazak u direktorij, `cd ..` za izlazak iz direktorija, `cd` za povratak u korijenski direktorij. 

Ok sad opet ne znam di sam? `pwd` (macOS/linux) ili `cd` (Windows) će vam reći trenutnu lokaciju. 

4. Postavljanje osnovnog Express poslužitelja

4.1 Instalacija Express.js

Express.js je popularni web okvir za Node.js koji omogućava brzu izradu web aplikacija na poslužiteljskom sloju. Express.js je jedan od najpopularnijih web okvira za Node.js, a koristi se za izradu **poslužiteljskog dijela** web aplikacija.



Kako instalirati Express.js? U terminalu upišite sljedeću naredbu:

```
npm install express
```

Naredba `npm install` koristi se za instalaciju paketa i modula iz Node paketnog registra. U ovom slučaju, instalirali smo [Express.js](#) paket.

[Node paketni register](#) je online baza podataka koja sadrži tisuće paketa i modula koje možemo koristiti u našim Node projektima.

Nakon što je instalacija završena, u direktoriju projekta trebali biste vidjeti nekoliko dodanih stavki:

- direktorij `node_modules` koji sadrži sve instalirane pakete i module odnosno njihov izvorni kod
- datoteku `package-lock.json` koja sadrži informacije o verzijama paketa i modula

`package-lock.json` datoteka je važna jer osigurava da svi članovi tima koriste iste verzije paketa i modula. Ova datoteka se automatski generira prilikom instalacije paketa i modula i također ju ne smijete mijenjati ručno.

Struktura direktorija projekta trebala bi izgledati ovako:

```
.
```

```
|   node_modules
```

```
|   package-lock.json
```

```
|   package.json
```

```
2 directories, 2 files
```

U sljedećem poglavlju ćemo izraditi naš prvi Express.js poslužitelj.

4.2 Osnovni Express.js poslužitelj

Krenimo napokon s implementacijom Express.js-a!  Dodat ćemo novu JavaScript datoteku proizvoljnog naziva, uobičajeno je koristiti `app.js`, `index.js` ili `server.js`.

Mi ćemo koristiti `index.js`.

Dodajte datoteku ručno, desni klik na direktorij projekta -> `New File` -> `index.js`. Ili ako želite biti terminal ninja , upišite:

```
touch index.js
```

Osnovni Express.js poslužitelj možemo definirati u svega nekoliko linija koda:

Prvo ćemo uključiti Express.js modul u našu datoteku:

```
const express = require('express');
```

Zatim ćemo stvoriti novu Express aplikaciju:

```
const app = express(); // u varijablu app pohranjujemo objekt koji predstavlja Express aplikaciju
```

OK, kako pokrećem Express.js poslužitelj? Koristimo `listen` metodu:

```
const PORT = 3000; // port na kojem će poslužitelj slušati zahtjeve

app.listen(PORT); // Express aplikacija "sluša" na portu 3000
```

Cijeli kod izgleda ovako:

```
const express = require('express');
const app = express();

const PORT = 3000;
app.listen(PORT);
```

Spremite datoteku i pokrenite Express.js poslužitelj tako da u terminalu upišete:

```
node index.js
```

Što se dogodilo? 😰

Čini se kao da nije ništa, međutim možemo u terminalu vidjeti da je proces pokrenut budući da ne možemo više upisivati nove naredbe. To znači da je Express.js poslužitelj uspješno pokrenut i da sluša zahtjeve na portu 3000.

Gdje ovo mogu vidjeti? Aplikaciju pokrećemo na vlastitom računalu, tako da se ona izvodi na adresi `127.0.0.1`, odnosno `localhost`. Dodatno, sluša na portu `3000`, tako da je puna adresa `http://localhost:3000`. Otvorite internetski preglednik i upišite ovu adresu, trebali biste vidjeti poruku "Cannot GET /" ili slično. To je u redu, jer nismo definirali nikakve rute ili putanje.

Kako zatvoriti Express.js poslužitelj? U terminalu pritisnite `ctrl + c` (Windows) ili `cmd + c` (macOS) kako biste prekinuli izvođenje programa. Ovo će zaustaviti Express.js poslužitelj i vratiti vam kontrolu nad terminalom.

Da bi bili sigurni da se naša aplikacija "vrти" možemo dodati `callback` funkciju našoj `listen` metodi:

```
app.listen(PORT, function () {
  console.log(`Server je pokrenut na http://localhost:${PORT}`);
});
```

Ova funkcija prima i `error` argument, tako da možemo uhvatiti potencijalne greške prilikom pokretanja poslužitelja:

Možemo ju zapisati i kao `arrow callback`:

```
app.listen(PORT, error => {
  if (error) {
    console.error(`Greška prilikom pokretanja poslužitelja: ${error.message}`);
  } else {
    console.log(`Server je pokrenut na http://localhost:${PORT}`);
  }
});
```

Spremite datoteku i ponovno pokrenite Express.js poslužitelj. Ovaj put ćete vidjeti poruku: "Server je pokrenut na `http://localhost:3000`" u terminalu.

I to je to! Uspješno ste izradili prvi Express.js poslužitelj! 🎉

4.3 Kako definirati osnovni endpoint?

Rute (*eng. routes*) su putanje koje korisnici mogu posjetiti u internetskom pregledniku. Na primjer, korisnik može posjetiti putanju `/` kako bi video početnu stranicu aplikacije, ili putanju `/about` kako bi video stranicu s informacijama o aplikaciji.

Nazivamo ih još i **endpoints** ili **API endpoints**.

API (*eng. Application Programming Interface*) je skup pravila i definicija koje omogućuju različitim softverskim aplikacijama da komuniciraju jedna s drugom. Express, iako se najčešće i koristi za izgradnju API servisa, može se koristiti i za izgradnju drugih vrsta softvera.

Mi ćemo u sklopu ovog kolegija koristiti Express.js za izgradnju ukupnog poslužiteljskog sloja naše web aplikacije, uključujući i API servis za komunikaciju s klijentskim dijelom aplikacije (Vue.js). Sigurno ste čuli i za **REST API** (*eng. Representational State Transfer*), međutim o tome ćemo uskoro!

Vratimo se na rute. Rekli smo da su to putanje koje korisnici mogu posjetiti u internetskom pregledniku i koje odgovaraju na određene zahtjeve korisnika.

Sigurno ste dosad imali priliku vidjeti rute u internetskim preglednicima, npr. <https://moodle.srce.hr/2024-2025/> gdje je 2024-2025 ruta koja odgovara akademskoj godini, odnosno prepostavljamo da će nas odvesti na stranicu s informacijama o akademskoj godini 2024/2025 (u pozadini: korisnik je zatražio informacije o akademskoj godini 2024/2025, a poslužitelj pokušava vratiti taj resurs).

Definirat ćemo osnovnu rutu / koja će korisnicima prikazati poruku "Hello, world!". Koristit ćemo get metodu koja obrađuje **HTTP GET** zahtjev.

```
app.get('/'); // definiramo rutu/endpoint
```

Zatim ćemo dodati callback funkciju koja će se izvršiti kada korisnik pošalje zahtjev na tu rutu.

Ova *callback* funkcija najčešće prima dva argumenta: req (request) i res (response).

req objekt sadrži informacije o zahtjevu korisnika, dok res objekt koristimo za slanje odgovora korisniku (možemo ih nazvati bilo kako ali ovo je konvencija i dobro je se držati).

Postoji i treći argument - next koji koristimo za preusmjeravanje zahtjeva na sljedeću funkciju u lancu middlewarea, ali o tome ćemo kasnije.

Osnovna metoda res objekta je send koja služi za slanje jednostavnog odgovora korisniku. Osim nje, postoji još mnogo metoda response objekta: poput json koja šalje podatke u obliku JSON-a ili.sendFile koja šalje datoteku.

```
app.get('/', function (req, res) {
  res.send('Hello, world!');
});
```

ili arrow callback:

```
app.get('/', (req, res) => {
  res.send('Hello, world!'); // šaljemo odgovor korisniku
});
```

To je to!

Cijeli kod izgleda ovako:

```
const express = require('express');
const app = express();

const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Hello, world!');
});

app.listen(PORT, error => {
  if (error) {
```

```
    console.error(`Greška prilikom pokretanja poslužitelja: ${error.message}`);
} else {
  console.log(`Server je pokrenut na http://localhost:${PORT}`);
}
});
```

Obavezno spremite datoteku i ponovo pokrenite Express.js poslužitelj. Otvorite internetski preglednik i posjetite adresu `http://localhost:3000`. Trebali biste vidjeti poruku "Hello, world!".

Međutim, što smo ustvari dobili nazad? Otvorimo konzolu u pregledniku (F12) i vidjet ćemo da smo dobili HTML stranicu s porukom "`Hello, world!`".

```
<html>
  <head></head>
  <body>
    <text>Hello, world!</text>
  </body>
</html>
```

Kada budemo učili o zaglavljima HTTP zahtjeva, vidjet ćemo zašto je ovo tako. Za sad, zapamtite da će `send` metoda poslati odgovor korisniku u obliku HTML stranice.

4.4 Nodemon

Primijetite da je potrebno svaki put ručno zaustaviti i ponovno pokrenuti Express.js aplikaciju kada napravimo promjene u kodu. Ono što smo prilikom razvoja Vue.js aplikacija uzimali zdravo za gotovo, ovdje si moramo ručno podesiti. Iz tog razloga koristimo `nodemon` - paket koji automatski prati promjene u kodu i ponovno pokreće Express.js aplikaciju.

Kako instalirati `nodemon` kroz `npm`?

```
npm install -g nodemon
```

Opcija `-g` označava globalnu instalaciju, što znači da će `nodemon` biti dostupan u cijelom sustavu (našem računalu). Ovo je korisno jer možemo koristiti `nodemon` za pokretanje bilo koje Node.js aplikacije, a ne samo Express.js aplikacija. Ako vam `nodemon` ne radi globalno nakon instalacije, pokušajte restartirati računalo.

Rekli smo da u `package.json` datoteci definiramo aplikacije koje naš paket koristi. Kako naša aplikacija nema direktnе koristi od `nodemon` paketa, već samo mi kao developeri, možemo koristiti `--save-dev` opciju prilikom instalacije koja će dodati `nodemon` paket u `devDependencies` dio `package.json` datoteke (odnosno pakete koji su potrebni samo prilikom razvoja aplikacije).

```
npm install --save-dev nodemon
```

Vaša `package.json` datoteka sada bi trebala izgledati ovako:

```
{
  "name": "wa_vjezbe_01",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
```

```

    "test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [],
"author": "",
"license": "ISC",
"description": "",
"dependencies": {
  "express": "^4.21.1"
},
"devDependencies": {
  "nodemon": "^3.1.7"
}
}

```

Kako koristiti `nodemon`? Umjesto `node` naredbe, koristimo `nodemon` naredbu. Simple as that! 🚀

`nodemon index.js`

Sada kada napravimo promjene u kodu, `nodemon` će automatski prepoznati promjene i ponovno pokrenuti Express.js aplikaciju. To nam štedi vrijeme i olakšava razvoj aplikacije.

4.5 Git commit

Napravili smo dosta promjena u kodu, vrijeme je za prvi `commit`! 🎉

Primjećujemo da su se u lijevom izborniku VS Codea evidentirale promjene u našem projektu (vjerojatno njih 500+). Zašto se ovo dešava ako smo samo inicijalizirali node projekt i dodali jednu datoteku i napisali nekoliko linija koda? 😲

Odgovor je jednostavan: `node_modules` direktorij. Ovaj direktorij sadrži sve instalirane pakete i module potrebne za uspješno izvođenje naše aplikacije. Ovaj direktorij je velik i sadrži tisuće datoteka, što znači da će se pojaviti puno promjena u našem projektu. Međutim, `node_modules` direktorij nije potreban za izvođenje naše aplikacije jer možemo ponovno instalirati sve pakete i module koristeći `npm install` naredbu.

`npm install` metoda čita `package.json` datoteku i instalira sve pakete i module navedene u `dependencies` i `devDependencies` dijelovima datoteke. Ovo je vrlo korisno jer omogućava drugim developerima da lako instaliraju sve potrebne pakete i module za izvođenje naše aplikacije.

Kako bismo izbjegli dodavanje `node_modules` direktorija u repozitorij, dodajemo ga u `.gitignore` datoteku. Ova datoteka sadrži popis datoteka i direktorija koje ne želimo dodati u repozitorij. Dodajte `node_modules` direktorij u `.gitignore` datoteku:

Datoteka: `.gitignore`

`node_modules`

Struktura našeg projekta sada izgleda ovako:

```
.  
├── index.js  
├── node_modules  
├── package-lock.json  
└── package.json
```

2 directories, 3 files

Primjetit ćete da se promjene u `node_modules` direktoriju više ne pojavljuju u lijevom izborniku VS Codea i da se broj smanjio na nekoliko promjena.

Sada smo spremni napraviti naš prvi `commit`!

1. Način (kroz terminal):

Prvo provjerimo stanje indeksa:

```
git status
```

Ova naredba će ispisati sve promjene u projektu. Za sada nismo definirali što dodajemo u indeks, pa će nas tražiti da dodajemo datoteke s naredbom `git add`.

Možemo dodati sve datoteke u indeks tako da kao argument navedemo `.`:

```
git add .
```

Pozvat ćemo opet `git status` kako bismo provjerili jesu li sve datoteke dodane u indeks:

```
git status
```

Uvjerite se da nema datoteke `node_modules` u popisu datoteka koje će se dodati u indeks.

```
Changes to be committed:  
(use "git restore --staged <file>..." to unstage)  
  new file:  .gitignore  
  new file:  index.js  
  new file:  package-lock.json  
  new file:  package.json
```

Sada možemo pohraniti promjene kroz `commit` naredbu:

Dodajemo obaveznu poruku uz `-m` opciju:

```
git commit -m "Inicijalni commit"
```

Ako je sve prošlo u redu, dobit ćete poruku s popisom promjena:

```
[main 465f003] Inicijalni commit
 4 files changed, 1184 insertions(+)
create mode 100644 1. Uvod u Node i Express/wa_vjezbe_01/.gitignore
create mode 100644 1. Uvod u Node i Express/wa_vjezbe_01/index.js
create mode 100644 1. Uvod u Node i Express/wa_vjezbe_01/package-lock.json
create mode 100644 1. Uvod u Node i Express/wa_vjezbe_01/package.json
```

Ono što još trebamo napraviti je pohraniti promjene na udaljeni repozitorij. Ovo radimo kroz `push` naredbu:

```
git push
```

2. Način (kroz VS Code):

Otvorite Source Control tab u lijevom izborniku VS Codea. Prikazat će se sve promjene u projektu. Unesite poruku i jednostavno pritisnite Commit ikonu kako biste pohranili promjene (ovo je ekvivalentno `git add` i `git commit` naredbama). Zatim odaberite Sync Changes kako biste pohranili promjene na udaljeni repozitorij (ovo je ekvivalentno `git push` naredbi).

3. Način (kroz GitHub Desktop):

Otvorite GitHub Desktop aplikaciju i pronađite vaš repozitorij. Vidjet ćete vizualni prikaz promjena u projektu na tabu Changes.

Možete dodati opis promjena i pritisnuti Commit to main kako biste pohranili promjene (ovo je ekvivalentno `git commit` naredbi). Zatim pritisnite Push origin kako biste pohranili promjene na udaljeni repozitorij (ovo je ekvivalentno `git push` naredbi).

5. HTTP protokol

HTTP (eng. *Hypertext Transfer Protocol*) je protokol koji se koristi za **prijenos podataka na webu**. HTTP definira skup pravila i definicija koje omogućuju web preglednicima i poslužiteljima da komuniciraju jedni s drugima. HTTP protokol uključuje **zahtjeve** (eng. requests) koje klijenti šalju poslužiteljima, te **odgovore** (eng. responses) koje poslužitelji šalju klijentima.

HTTP koristi različite **metode** (eng. *HTTP method*) za različite vrste zahtjeva. Najčešće korištene HTTP metode su:

- **GET** - koristi se za dohvaćanje podataka
- **POST** - koristi se za slanje podataka
- **PUT** - koristi se za ažuriranje podataka
- **DELETE** - koristi se za brisanje podataka
- **PATCH** - koristi se za djelomično ažuriranje podataka

Ove metode koriste se za različite vrste zahtjeva. Na primjer, korisnik može poslati `GET` zahtjev kako bi dohvatio podatke s poslužitelja, ili `POST` zahtjev kako bi poslao podatke poslužitelju. Sve ove metode koriste se u web razvoju za komunikaciju između klijenta i poslužitelja. U nastavku ćemo obraditi svaku metodu posebno i pokazati kako ih implementirati u Express.js aplikaciji.

Međutim, prije nego što krenemo, važno je naučiti od čega se sastoje HTTP zahtjevi i odgovori.

HTTP prati klasičnu **klijent-poslužitelj** arhitekturu (*eng. client-server architecture*). Ukratko, to znači da klijent šalje zahtjev poslužitelju, a poslužitelj šalje odgovor klijentu. Preciznije, klijent otvara **TCP/IP** vezu s poslužiteljem, šalje HTTP zahtjev i onda čeka sve dok poslužitelj ne pošalje odgovor.

HTTP je **stateless** protokol, što znači da svaki zahtjev poslužitelju ne zna ništa o prethodnim zahtjevima. Na primjer, kada korisnik posjeti stranicu, poslužitelj ne zna ništa o prethodnim posjetama korisnika. Ovo je korisno jer omogućava poslužitelju da bude brži i efikasniji, međutim postoje tehnike kojima možemo na klijentskoj strani zapamtiti prethodne interakcije, npr. kroz kolačiće (*eng. cookies*) ili lokalno pohranjivanje (*eng. local storage*) te na taj način imati neki oblik stanja koji šaljemo s klijenta na poslužitelj.

Dakle, za sad je važno zapamtiti da klijent šalje HTTP zahtjeve poslužitelju, čeka odgovor i zatim prikazuje odgovor krajnjem korisniku. Naravno, to ne mora biti i vrlo često i nije (1 <-> 1) komunikacija, već klijent može slati različite zahtjeve na različite poslužitelje. No mi ćemo u sklopu ovog kolegija raditi samo s jednim poslužiteljem i jednim klijentom.

5.1 HTTP zahtjev (eng. HTTP request)

HTTP zahtjev predstavlja zahtjev klijenta poslužitelju, npr. klijent (web preglednik) zahtjeva određeni web resurs (npr. HTML stranicu) od poslužitelja.

HTTP zahtjev sastoji se od nekoliko dijelova od kojih su neki **obavezni**, a neki **opcionali**:

5.1.1 Obavezni dijelovi HTTP zahtjeva

Kako bi klijent poslao najjednostavniji mogući HTTP zahtjev, potrebno je navesti kome šaljemo zahtjev (*eng. Host Header*) te što želimo (*eng. Request Line*).

| Obavezni dijelovi HTTP zahtjeva | Opis | Primjer |
|---------------------------------|--|---|
| Request Line | Sastoji se od HTTP metode , traženog URI i HTTP verzije | <code>GET /index.html HTTP/1.1</code> |
| Host Header | Navodi se naziv domene ili IP adresu poslužitelja | <code>Host: www.example.com</code> |

Međutim, **Host Header** je ustvari jedini obavezni dio zahtjeva, ali to u pravilu ne želimo raditi. Idemo demonstrirati programom `curl` kako izgleda najjednostavniji HTTP zahtjev. Ovaj program je u pravilu dostupan na svakom OS-u, a koristi se za slanje HTTP zahtjeva iz terminala. Možete provjeriti imate li ga instaliranog s naredbom `curl --version`.

Idemo poslati najjednostavniji mogući HTTP zahtjev prema `http://www.google.com`:

```
curl http://www.google.com
```

Uočite što smo dobili - HTML stranicu koja definira Googleovu početnu stranicu. `curl` je automatski odabrao `GET` metodu, ali metodu možemo navesti i eksplicitno opcijom `-X`:

```
curl -X GET http://www.google.com
```

Koji smo **URI** (*eng. Uniform Resource Identifier*) dohvatili u ovom slučaju? URI predstavlja jedinstveni identifikator elektroničkog resursa. URI se često koristi kao sinonim za URL (*eng. Uniform Resource Locator*), međutim URI je općenitiji pojam koji uključuje i URL i URN (*eng. Uniform Resource Name*). Točnije, URL i URN su podskup URI-a.

U ovoj skripti će se često koristiti termin URI.

Dakle što je ovdje URI? `http://www.google.com`

Sve navedeno, ali što onda dohvaćamo? Odgovor je osnovni endpoint definiran putanjom `/`. Vidimo da je Google definirao osnovni endpoint kao početnu stranicu, to je jasno, ali sad već možemo i prepostaviti kako se zove datoteka koju dohvaćamo - `index.html`. Endpoint ili ruta `/` je u pravilu početna stranica web stranice, a datoteka `index.html` je osnovna HTML stranica koja se prikazuje korisniku.

```
curl -X GET http://www.google.com/index.html
```

Radi! 

Što ako probamo dohvatiti nešto što ne postoji? Na primjer, `http://www.google.com/about_me.html`

```
curl -X GET http://www.google.com/about_me.html
```

Vidimo da kao odgovor dobivamo HTML stranicu s porukom `"404. That's an error. The requested URL was not found on this server. That's all we know."`. Ako otvorimo u web pregledniku, ona izgleda ovako:

Dakle, **Request Line** se sastoji od HTTP **metode**, traženog **URI** i HTTP **verzije**.

| Dijelovi Request Line komponente | Opis | Primjer |
|----------------------------------|---|--|
| HTTP metoda | Akcija koju klijent želi izvršiti (npr. GET, POST, PUT) | <code>GET</code> |
| URI zahtjeva | Specifični resurs na poslužitelju koji klijent traži | <code>/over/here?name=something</code> |
| HTTP verzija | Verzija HTTP-a koja se koristi u zahtjevu | <code>HTTP/1.1</code> |

Verziju HTTP-a možemo navesti i eksplicitno, međutim u pravilu se automatski koristi `HTTP/1.1`.

Vježba 1 - HTTP zahtjev prema našem Expressu

Pokrenite Express poslužitelj koji smo izradili i pošaljite HTTP zahtjev prema njemu koristeći `curl` program. Koji odgovor očekujete?

5.1.2 Opcionalni dijelovi HTTP zahtjeva

Osim obaveznih dijelova HTTP zahtjeva, postoje i opcionalni dijelovi koji se koriste za slanje dodatnih informacija poslužitelju. Konkretno, možemo poslati **HTTP zaglavlja** (*eng. HTTP headers*) i **HTTP tijelo** (*eng. HTTP body*).

| Opcionalni dijelovi HTTP zahtjeva | Opis | Primjer |
|--|--|---|
| Opcionalna zaglavlj zahtjeva (eng. Request Headers) | Ključ-vrijednost parovi koji pružaju dodatne informacije o zahtjevu (zamislimo ih kao metapodatke) | <code>Content-Type: application/json Authorization: Bearer <token> Accept: text/html</code> |
| Tijelo zahtjeva (eng. Request Body) | Stvarni podaci koje šaljemo, često u JSON formatu, a tipično se koristi u metodama poput POST, PUT, itd. | <code>{ "username": "Pero", "password": "password123" }</code> |

Zaglavla ćemo raditi detaljnije na nekim drugim vježbama, za sada morate znati samo da postoje i da se koriste za slanje dodatnih informacija poslužitelju.

Tijelo se koristi u metodama poput POST, PUT, DELETE, PATCH, itd. gdje šaljemo podatke poslužitelju. Na primjer, kada se korisnik registrira na web stranici, šaljemo podatke kao što su **korisničko ime, lozinka, e-mail**, itd. u tijelu zahtjeva. Tijelo se može poslati u različitim formatima podataka, za sada nas zanima **JSON format**.

Kako ćemo definirati tijelo zahtjeva iznad kao JSON?

```
{
  "korisniko_ime": "pero_peric",
  "lozinka": "password123",
  "email": "pperic@gmail.com"
}
```

Recimo da naš poslužitelj ima definirani endpoint `/registracija` koji očekuje ove podatke. Dakle, korisnik ne traži nikakav resurs od poslužitelja pa niti HTML stranicu, već isključivo šalje podatke poslužitelju i očekuje nekakav odgovor (ne resurs). Ovakav endpoint definiramo `POST` metodom koja nam dozvoljava slanje podataka kroz tijelo zahtjeva, za razliku od `GET` metode.

Koristeći `curl` program, možemo poslati tijelo zahtjeva kroz opciju `-d`:

```
curl -X POST http://www.nas-super-server.com/registracija -d '{"korisniko_ime": "pero_peric",  
"lozinka": "password123", "email": "pperic@gmail.com"}'
```

Naravno, ovo neće raditi.

Više o HTTP zahtjevima možete pročitati na [MDN web dokumentaciji](#).

5.2 HTTP odgovor (eng. HTTP response)

HTTP odgovor predstavlja odgovor poslužitelja klijentu, npr. poslužitelj šalje HTML stranicu ili JSON podatke klijentu. HTTP odgovor sastoji se od nekoliko dijelova od kojih su, kao i kod zahtjeva, neki **obavezni**, a neki **opcionalni**:

5.2.1 Obavezni dijelovi HTTP odgovora

Kako bi poslužitelj poslao najjednostavniji mogući HTTP odgovor, potrebno je navesti **HTTP verziju**, **statusni kod** i **statusni tekst** kao i obavezna zaglavla odgovora (eng. *Response headers*).

| Obavezni dijelovi HTTP odgovora | Opis | Primjer |
|--|--|---|
| Status Line | Sadrži HTTP verziju , statusni kod (eng. <i>status code</i>) i reason phrase | <code>HTTP/1.1 200 OK</code> <code>HTTP/1.1 404 Not Found</code> |
| Obavezna zaglavla odgovora (eng. <i>Response headers</i>) | Pruža obavezne metapodatke o odgovoru (npr. Content-Type) | <code>Content-Type: application/json</code> <code>Date: Mon, 21 Oct 2024 10:32:45 GMT</code> |

Kod **Status Line** komponente, najzanimljiviji nam je **statusni kod**. Vjerojatno smo se svi do sad susreli sa statusnim kodovima koje nam je vratio neki poslužitelj.

Primjerice, poznati statusni kod `404` označava da traženi resurs nije pronađen, odnosno da je korisnik poslao zahtjev za resurs koji ne postoji.

Statusni kod `503` označava grešku na poslužitelju, odnosno da poslužitelj trenutno nije dostupan.

U grubo, brojevi ovih kodova označavaju različite situacije koje se mogu dogoditi prilikom slanja zahtjeva poslužitelju:

- `1xx` (100 - 199) - Informacijski odgovori (eng. *Informational responses*): Poslužitelj je primio zahtjev te ga i dalje obrađuje
- `2xx` (200 - 299) - Odgovori uspjeha (eng. *Successful responses*): Zahtjev klijenta uspješno primljen i obrađen
- `3xx` (300 - 399) - Odgovori preusmjeravanja (eng. *Redirection messages*): Ova skupina kodova govori klijentu da mora poduzeti dodatne radnje kako bi dovršio zahtjev
- `4xx` (400 - 499) - Greške na strani klijenta (eng. *Client error responses*): Sadrži statusne kodove koji se odnose na greške nastale na klijentskoj strani
- `5xx` (500 - 599) - Greške na strani poslužitelja (eng. *Server error responses*): Sadrži statusne kodove koji se odnose na greške nastale na poslužiteljskoj strani

Više o statusnim kodovima uskoro, a možete ih pročitati i sami na [MDN web dokumentaciji](#).

reason phrase odnosi se na kratki opis statusnog koda, npr. `OK` za statusni kod `200` ili `Not Found` za statusni kod `404`. Ovaj dio u pravilu nikad ne želimo mijenjati.

5.2.2 Opcionalni dijelovi HTTP odgovora

Osim obaveznih dijelova HTTP odgovora, postoje i opcionalni dijelovi koji se koriste za slanje dodatnih informacija klijentu. Konkretno, možemo poslati **HTTP zaglavla** (eng. *HTTP headers*) i **HTTP tijelo** (eng. *HTTP body*).

| Component | Description | Example |
|---|---|---|
| Tijelo odgovora (eng. Response body) | Stvarni podaci koji se vraćaju korisniku, npr. u JSON ili XML formatima | { "message": "Success", "data": { "id": 1, "name": "John" } } |
| Opcionalna zaglavljva odgovora (eng. Response headers) | Pruža opcionalne metapodatke o odgovoru (npr. Set-Cookie) | Set-Cookie: sessionId=abc123 Cache-Control: no-cache |

Vježba 2: Kako vidjeti cijeli HTTP odgovor?

Pošaljite ponovo zahtjev programom `curl` na Express poslužitelj koji smo definirali.

```
curl http://localhost:3000
```

Kao odgovor dobili smo tijelo s porukom `"Hello, world!"`. Koji statusni kod očekujete? 🤔

Provjerit ćemo obavezna zaglavja i statusni kod koji smo dobili kao odgovor kako bi vidjeli što Express radi u pozadini. Možemo koristiti opciju `-i` kako bismo vidjeli ukupan HTTP odgovor:

```
curl -i http://localhost:3000
```

Možemo vidjeti da cijeli HTTP odgovor ustvari izgleda ovako:

```
HTTP/1.1 200 OK

X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 13
ETag: W/"d-1DpwLQbzRZmu4fjajvn3KWAX1pk"
Date: Mon, 21 Oct 2024 13:37:34 GMT
Connection: keep-alive
Keep-Alive: timeout=5

Hello, world!
```

- Prvi dio je **Status Line** koji sadrži HTTP **verziju**, **statusni kod** i **reason phrase**.
- Drugi dio sadrži **zaglavljva odgovora** koja pružaju metapodatke o odgovoru.
- Treći dio je **tijelo odgovora** koje sadrži stvarni sadržaj koji se šalje klijentu.

Samostalni zadatak za Vježbu 1

Izmijenite vaš Express poslužitelj tako da:

1. Nadogradite postojeću GET rutu `/` koja sad mora vratiti HTML stranicu s `<h1>` naslovom "Hello, Express!".
2. Dodate još jednu GET rutu `/about` koja će vratiti HTML stranicu s `<h1>` naslovom "Ovo je stranica o nama!".

Obje HTML stranice pohranite u direktorij `/public`.

Kako biste vratili podatke u obliku HTML stranice, koristite `res.sendFile()` metodu.

Sintaksa:

```
res.sendFile(__dirname + 'putanja_do_datoteke');
```

3. Dodajte i posljednju GET rutu `/users` koja će vratiti korisnike u JSON formatu. Korisnike pohranite u polju kao objekte s atributima `id`, `ime` i `prezime`. Dodajte barem 3 korisnika. Kako biste vratili korisnike u JSON formatu, koristite `res.json()` metodu.

Testirajte u web pregledniku i s programom `curl` sve tri rute.

Kada završite, pohranite promjene na GitHub repozitorij s komentarom "Samostalni zadatak za vježbu 1".

Web aplikacije (WA)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



Fakultet informatike u Puli

(2) Usmjeravanje na Express poslužitelju

#2

WA

Usmjeravanje (eng. routing) se odnosi na određivanje kako će krajnje rute koje definiramo na našoj poslužiteljskoj strani odgovarati na dolazne zahtjeve klijenata. U prošloj skripti smo već definirali osnovni primjer usmjeravanja za nekoliko GET ruta i posluživali smo statične datoteke i jednostavne JSON objekte. Danas ćete naučiti kako definirati složenije usmjeravanje kroz sve HTTP metode, koja su pravila usmjeravanja i dodatni parametri koje koristimo.

Posljednje ažurirano: 3.11.2024.

Sadržaj

- [Web aplikacije \(WA\)](#)
- [\(2\) Usmjeravanje na Express poslužitelju](#)
 - [Sadržaj](#)
- [1. Ponavljanje](#)
- [2. Osnovno usmjeravanje](#)
 - [2.1 `GET` metoda i parametri](#)
 - [2.2 `POST` metoda i slanje podataka
 - \[2.2.1 Kako slati `POST` zahtjeve jednostavnije?\]\(#\)](#)
 - [Vježba 1 - Naručivanje više pizze](#)
 - [Vježba 2 - Zanima nas i adresa dostave](#)
 - [2.3 `PUT` i `PATCH` metode
 - \[2.3.1 `PUT` metoda\]\(#\)
 - \[2.3.2 `PATCH` metoda\]\(#\)](#)
 - [2.4 `DELETE` metoda](#)
 - [2.5 Kada koristiti koju `HTTP` metodu?](#)

- [3. Router objekt](#)
 - [3.1 Kako koristiti Router objekt?](#)
 - [3.2 Idemo još bolje strukturirati našu aplikaciju](#)
 - [Vježba 3 - Strukturiranje narudžbi ➡️🍕](#)
- [4. Statusni kodovi u odgovorima](#)
 - [4.1 Kako koristiti statusne kodove u Expressu?](#)
 - [Vježba 4 - Korištenje statusnih kodova u pizzeriji 🍕 4 0 4](#)
- [Samostalni zadatak za Vježbu 2](#)

1. Ponavljanje

Nastavljamo s radom na Express poslužitelju, na ovim ćemo vježbama detaljnije proučiti **usmjerenje i obradu zahtjeva** u Express aplikacijama.

Usmjerenje (*eng. routing*) se odnosi na određivanje kako će krajne rute koje definiramo na našoj poslužiteljskoj strani odgovarati na dolazne zahtjeve klijenata. U prošloj skripti smo već definirali osnovni primjer usmjerenja za nekoliko ruta i posluživali smo statične datoteke i JSON objekte.

Osnovna sintaksa za definiranje ruta u Express aplikacijama je sljedeća:

```
app.METHOD(PATH, HANDLER);
```

gdje je:

- `app` je instanca Express aplikacije
- `METHOD` je HTTP metoda (npr. GET, POST, PUT, DELETE, itd.) koju želimo posluživati
- `PATH` je putanja na koju želimo reagirati (npr. `/`, `/about`, `/contact`, itd.)
- `HANDLER` je callback funkcija koja se izvršava kada se zahtjev podudara s definiranom rutom

Tako smo definirali rutu za početnu stranicu:

```
app.get('/', function (req, res) {
  res.send('Hello, world!');
});

// odnosno

app.get('/', (req, res) => {
  res.send('Hello, world!');
});
```

`PATH` koji smo ovdje koristili je `/`, što znači da će se ova ruta pokrenuti kada korisnik posjeti početnu stranicu našeg web sjedišta.

U ovom primjeru koristili smo `GET` metodu, za koju smo općenito rekli da se koristi kada korisnik želi dohvatiti neki resurs s poslužitelja, bio on HTML dokument, slika, CSS datoteka, JavaScript datoteka, JSON objekt, itd.

2. Osnovno usmjeravanje

2.1 GET metoda i parametri

U prošloj smo skripti već naučili kako koristiti `GET` metodu za dohvati resursa s poslužitelja. U ovom ćemo primjeru proširiti našu aplikaciju tako da možemo dohvatiti resurs s poslužitelja na temelju **parametara** koje korisnik prenosi u URL-u.

Osnovna sintaksa za definiranje GET rute je sljedeća:

```
app.get(PATH, (req, res) => {
    // Ovdje pišemo kod koji će se izvršiti kada korisnik posjeti PATH
});
```

Primjerice, zamislimo da radimo **aplikaciju za naručivanje pizze** 🍕. Recimo da korisnik odluči pogledati koje su pizze dostupne, želimo da dohvati sve dostupne pizze definirane na našem poslužitelju. U tom slučaju, korisnik bi mogao posjetiti URL `/pizze`.

```
app.get('/pizze', (req, res) => {
    res.send('Ovdje su sve dostupne pizze!');
});
```

Rekli smo da možemo koristiti metodu `res.json` kako bismo poslali JSON objekt korisniku. U ovom slučaju, možemo poslati listu dostupnih pizza kao JSON objekt:

No prvo moramo definirati listu dostupnih pizza:

```
const pizze = [
    { id: 1, naziv: 'Margherita', cijena: 6.5 },
    { id: 2, naziv: 'Capricciosa', cijena: 8.0 },
    { id: 3, naziv: 'Quattro formaggi', cijena: 10.0 },
    { id: 4, naziv: 'Šunka sir', cijena: 7.0 },
    { id: 5, naziv: 'Vegetariana', cijena: 9.0 }
];

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

Kada korisnik posjeti URL `/pizze`, dobit će JSON objekt s listom dostupnih pizza. Ako nemate instaliranu jednu od ekstenzija za web preglednik koje omogućuju pregled JSON objekata u pregledniku, JSON će vam se prikazivati kao običan tekst (*eng. raw*) bez formatiranja, što može biti nepregledno. Preporuka je preuzeti jednu od JSON Formatter ekstenzija za preglednik, npr. [JSON Formatter](#) za Chromium preglednike.

Što ako korisnik želi dohvatiti **samo jednu pizzu**, a ne sve? Kako ćemo definirati rutu za dohvat jedne pizze?

Možemo definirati posebnu rutu za svaku pizzu, npr. `/margherita`, `/capricciosa`, `/quattro-formaggi`, itd. Međutim, koliko je to rješenje pametno?

Možda bi mogli proći s ovim ako restoran ima 4-5 pizza, ili 15. Što ako restoran ima 50 pizza? ili 100?

Navedeno je primjer lošeg dizajna i nepotrebno ponavljanje koda. Umjesto toga, možemo koristiti **parametre** u URL-u kako bismo dohvatili jednu pizzu.

URL parametar je dio URL-a koji se koristi za prenošenje informacija između klijenta i poslužitelja. URL parametri se definiraju u URL-u s prefiksom `:`.

Primjerice, ako možemo definirati rutu `/pizze/:id` koja će dohvatiti pizzu s određenim `id` parametrom:

```
app.get('/pizze/:id', (req, res) => {
  res.json(pizze);
});
```

Kako bi sad dohvatili određenu pizzu, moramo poslati zahtjev u obliku `/pizze/1`, `/pizze/2`, `/pizze/3`, itd. Nećemo navoditi eksplisitno `"id"` u URL-U, već nam služi kao svojevrsni **placeholder**.

Pošaljite zahtjev na `/pizze/1` i provjerite rezultat.

Zašto nismo dobili podatke samo za jednu pizzu iako smo poslali `id` parametar?

► Spoiler alert! Odgovor na pitanje

Idemo sada definirati logiku koja će dohvatiti samo jednu pizzu na temelju `id` parametra. Za početak, stvari možemo odraditi na "ručni" način, tj. prolaskom kroz sve dostupne pizze i pronašlakom one koja ima traženi `id`.

`id` iz URL-a je tipa string i možemo ga jednostavno dohvatiti pomoću `req.params` objekta.

```
app.get('/pizze/:id', (req, res) => {
  const id_pizza = req.params.id; // dohvaćamo id parametar iz URL-a

  for (pizza of pizze) {
    if (pizza.id == id_pizza) {
      // ako smo pronašli podudaranje u id-u
      res.json(pizza); // vrati objekt pizze kao rezultat
    }
  }
});
```

Sada kada pošaljemo zahtjev na `/pizze/1`, dobit ćemo JSON objekt s podacima o pizzi s `id`-om 1, odnosno Margheriti.

```
curl -X GET http://localhost:3000/pizze/1
```

Rezultat:

```
{
  "id": 1,
  "naziv": "Margherita",
  "cijena": 6.5
}
```

Naš endpoint `/pizze` funkcioniра и dalje i možemo ga pozvati bez parametara:

```
curl -X GET http://localhost:3000/pizze
```

Rezultat:

```
[  
  {  
    "id": 1,  
    "naziv": "Margherita",  
    "cijena": 6.5  
  },  
  {  
    "id": 2,  
    "naziv": "Capricciosa",  
    "cijena": 8  
  },  
  {  
    "id": 3,  
    "naziv": "Quattro formaggi",  
    "cijena": 10  
  },  
  {  
    "id": 4,  
    "naziv": "Šunka sir",  
    "cijena": 7  
  },  
  {  
    "id": 5,  
    "naziv": "Vegetariana",  
    "cijena": 9  
  }  
]
```

Kod možemo pojednostaviti korištenjem metode `find` koja će nam vratiti prvi element koji zadovoljava uvjet:

```
app.get('/pizze/:id', (req, res) => {  
  const id_pizza = req.params.id; // dohvaćamo id parametar iz URL-a  
  
  const pizza = pizze.find(pizza => pizza.id == id_pizza); // pronalazimo pizzu s traženim  
  // id-em  
  
  res.json(pizza);  
});
```

Što ako korisnik pošalje zahtjev za pizzu koja ne postoji? Kako ćemo riješiti taj slučaj? 🤔

► Spoiler alert! Odgovor na pitanje

```

app.get('/pizze/:id', (req, res) => {
  const id_pizza = req.params.id; // dohvaćamo id parametar iz URL-a

  const pizza = pizze.find(pizza => pizza.id == id_pizza);

  if (pizza) {
    // ako je pronađeno podudaranje, vratimo pizza objekt
    res.json(pizza);
  } else {
    // ako je rezultat undefined, vratimo poruku da pizza ne postoji
    res.json({ message: 'Pizza s traženim ID-em ne postoji.' });
  }
});

```

Sada kada pošaljemo zahtjev na `/pizze/6`, dobit ćemo poruku da pizza s traženim ID-em ne postoji.

```
curl -X GET http://localhost:3000/pizze/6
```

Rezultat:

```
{
  "message": "Pizza s traženim ID-em ne postoji."
}
```

Što ako korisnik pošalje zahtjev na `/pizze/vegetariana`? Kako ćemo riješiti taj slučaj? 🤔

► Spoiler alert! Odgovor na pitanje

Možemo koristiti metodu `isNaN` (is Not a Number) kako bismo provjerili je li `id` parametar broj:

```

app.get('/pizze/:id', (req, res) => {
  const id_pizza = req.params.id;

  if (isNaN(id_pizza)) {
    // provjeravamo je li id_pizza "Not a Number"
    res.json({ message: 'Proslijedili ste parametar id koji nije broj!' });
    return;
  }

  const pizza = pizze.find(pizza => pizza.id == id_pizza);

  if (pizza) {
    res.json(pizza);
  } else {
    res.json({ message: 'Pizza s traženim ID-em ne postoji.' });
  }
});

```

2.2 POST metoda i slanje podataka

Do sada smo koristili `GET` metodu za dohvatanje resursa s poslužitelja. Sada ćemo naučiti kako koristiti `POST` metodu za slanje podataka na poslužitelj.

`POST` metoda se koristi kada korisnik želi poslati podatke na poslužitelj, npr. kada korisnik želi **izraditi novi resurs na poslužitelju**, a podaci se šalju u **tijelu zahtjeva** (eng. *request body*).

Osnovna sintaksa za definiranje POST rute je sljedeća:

```
app.post(PATH, (req, res) => {
  // Ovdje pišemo kod koji će se izvršiti kada korisnik pošalje POST zahtjev na PATH
});
```

Vratimo se na primjer aplikacije za naručivanje pizze. Zamislimo da korisnik želi **naručiti pizzu**. Kako bismo omogućili korisniku da naruči pizzu, moramo definirati POST rutu koja će omogućiti korisniku da nekako pošalje podatke o narudžbi na poslužitelj.

Idemo napisati kostur POST rute za naručivanje pizze:

```
app.post('/naruci', (req, res) => {
  // Ovdje ćemo napisati logiku za naručivanje pizze
});
```

Ako otvorite ovu rutu u pregledniku, dobit ćete poruku "Cannot GET /naruci". To je zato što smo definirali POST rutu, a pokušavamo je otvoriti u pregledniku, što će automatski poslati GET zahtjev!

Možemo dodati jednostavnu poruku koja će korisniku reći da je narudžba uspješno zaprimljena:

```
app.post('/naruci', (req, res) => {
  res.send('Vaša narudžba je uspješno zaprimljena!');
});
```

Zahtjev možemo poslati kroz terminal aplikaciju `curl` koju smo koristili u prethodnim primjerima:

```
curl -X POST http://localhost:3000/naruci
```

Kako možemo poslati podatke o narudžbi kroz POST HTTP zahtjev? 🤔

Hoćemo li to raditi kroz parametre u URL-u?

```
//?
app.post('/naruci/:id', (req, res) => {
  res.send(`Zaprimio sam narudžbu za pizzu ${req.params.id}`);
});
```

► Spoiler alert! Odgovor na pitanje

Kako bismo poslali veličinu pizze koju želimo naručiti?

```
// ?
app.post('/naruci/:id/:velicina', (req, res) => {
  res.send(`Zaprimio sam narudžbu za ${req.params.velicina} pizza ${req.params.id}`);
});
```

Dva isječka koda iznad primjeri su jako loše prakse. Zašto?

- **URL parametri su javno vidljivi** i mogu sadržavati osjetljive informacije (kako ćemo poslati podatke o plaćanju?)
- **Kod postaje nečitljiv** i teško održiv
- **Nije skalabilno** (što ako želimo poslati još više podataka? Ili više pizza?! 🍕🍕🍕)
- **Nije standardizirano** (kako ćemo znati koji parametar odgovara kojem podatku?)

Dakle, rekli smo da podatke šaljemo u **tijelu zahtjeva** (eng. *request body*). Kako ćemo to napraviti?

U prvoj skripti smo već naučili da podaci koji se šalju u tijelu zahtjeva mogu biti u različitim formatima, npr. JSON, XML, HTML, itd. Mi ćemo u pravilu slati podatke u **JSON** formatu.

Međutim, u našem web pregledniku nemamo mogućnost slanja POST zahtjeva s tijelom zahtjeva kada direktno pristupamo URL-u neke rute poslužitelja. Možemo poslati kroz naš `curl` alat s opcijom `-d`:

```
curl -X POST http://localhost:3000/naruci -d '{"pizza": "Margherita", "velicina": "srednja"}'
```

Kako ćemo sada u našoj Express aplikaciji dohvatiti podatke koje je korisnik poslao u tijelu zahtjeva?

Podaci koje korisnik šalje u tijelu zahtjeva se nalaze u `req.body` objektu.

Primjer:

```
app.post('/naruci', (req, res) => {
  const narudzba = req.body;
  console.log('Primljeni podaci:', narudzba);
  res.send('Vaša narudžba je uspješno zaprimljena!');
});
```

Primjetit ćete da će se u konzoli ispisati poruka `"Primljeni podaci: undefined"`. Razlog zašto se ne ispisuju podaci je taj što Express ne zna kako parsirati podatke u tijelu zahtjeva. Da bismo to omogućili, moramo koristiti **middleware** koji će parsirati podatke u tijelu zahtjeva. O middleware funkcijama više u sljedećim lekcijama, međutim za sada ćemo koristiti ugrađeni middleware `express.json()` koji će parsirati podatke u JSON formatu.

Jednostavno dodajemo na početku naše aplikacije, nakon definiranja instance aplikacije:

```
app.use(express.json());
```

Pokušajte ponovo. Vidjet ćete da podaci i dalje ne dolaze kada šaljemo kroz `curl`. Razlog je taj što `curl` ne šalje podatke u JSON formatu po *defaultu*, već to moramo specificirati u **zaglavljiju** našeg HTTP zahtjeva.

Zaglavljia možemo specificirati pomoću opcije `-H`, a dodat ćemo zaglavljje `Content-Type: application/json`:

```
curl -X POST http://localhost:3000/naruci -H "Content-Type: application/json" -d '{"pizza": "Margherita", "velicina": "srednja"}'
```

Ako ste upisali točno naredbu, trebali biste vidjeti ispis u konzoli:

```
Primljeni podaci: { pizza: 'Margherita', velicina: 'srednja' }
```

Sada kada imamo podatke o narudžbi, možemo ih koristiti u našoj aplikaciji. Na primjer, možemo poslati korisniku poruku s informacijama o narudžbi:

```
app.post('/naruci', (req, res) => {
  const narudzba = req.body;
  console.log('Primljeni podaci:', narudzba);
  res.send(`Vaša narudžba za ${narudzba.pizza} (${narudzba.velicina}) je uspješno
  zaprimljena!`);
});
```

Što ako korisnik ne pošalje podatke o pizzi ili veličini pizze? Kako ćemo riješiti taj slučaj? 🤔

Možemo izvući ključeve JavaScript objekta kroz metodu `Object.keys` i provjeriti jesu li svi ključevi prisutni:

```
app.post('/naruci', (req, res) => {
  const narudzba = req.body;
  const kljucevi = Object.keys(narudzba);

  if (!(kljucevi.includes('pizza') && kljucevi.includes('velicina'))) {
    res.send('Niste poslali sve potrebne podatke za narudžbu!');
    return;
  }

  res.send(`Vaša narudžba za ${narudzba.pizza} (${narudzba.velicina}) je uspješno
  zaprimljena!`);
});
```

Sada kada pošaljemo zahtjev bez podataka:

```
curl -X POST http://localhost:3000/naruci -H "Content-Type: application/json" -d '{}'
```

Ili s pogrešnim podacima:

```
curl -X POST http://localhost:3000/naruci -H "Content-Type: application/json" -d '{"pizza": "Margherita", "cijena": 6.5}'
```

2.2.1 Kako slati `POST` zahtjeve jednostavnije?

Kako ne bismo morali svaki put pisati `curl` naredbe za slanje POST zahtjeva, možemo koristiti alate koji nam omogućuje puno jednostavnije slanje HTTP zahtjeva s tijelom zahtjeva, zaglavljima i drugim opcijama.

Jedan od takvih alata je [Postman](#), koji je dostupan za sve platforme i omogućuje nam jednostavno slanje HTTP zahtjeva, testiranje API-ja, automatsko generiranje dokumentacije, itd.



Preuzmite Postman s [ovog linka](#). Potrebno je izraditi račun, ali je besplatan za korištenje.

Jednom kada se prijavite, morate napraviti novi radni prostor (*workspace*). Kliknite na `New Workspace` i unesite naziv radnog prostora. Možete ga nazvati `Web aplikacije - Vježbe`.

Odaberite '+' i dodajte novu kolekciju koju možete nazvati `WA2` te dodajte novi zahtjev u kolekciju odabirom "Add a request". Nazovite zahtjev `Jelovnik` i odaberite GET zahtjev (po defaultu je GET).

Vidjet ćete razno-razne opcije koje možete koristiti za slanje zahtjeva, kao što su **URL, HTTP metoda, zaglavljia, tijelo zahtjeva, autorizacija** itd.

Uočite da se unutar zaglavja već nalazi postavljeno čak 7 različitih zaglavja, dakle Postman automatski postavlja neka zaglavja za nas.

Pošaljite zahtjev na endpoint `/pizze` i vidjet ćete rezultat u obliku JSON objekta s dostupnim pizzama. Morate unijeti puni URL u formatu:

```
http://localhost:3000/pizze
```

Ako je sve OK, ispod će vam se prikazati JSON objekt unutar **Body** taba, ali možete vidjeti i **zaglavija koja su došla s odgovorom**.

Postoji puno alternative Postmanu, npr. [Insomnia](#), [Paw](#), [Thunder Client](#), [HTTPie](#), od kojih se neki izvode na webu, a neki lokalno na računalu.

Zgodno je preuzeti i **Thunder Client** koji je dostupan kao ekstenzija za Visual Studio Code.

Otvorite Thunder Client ekstenziju i odaberite `New Request`. Unesite URL `http://localhost:3000/pizze` i odaberite metodu `GET`. Kliknite na `Send Request` i vidjet ćete isti rezultat kao i u Postmanu.

`POST` zahtjev možete poslati na isti način, samo odaberite metodu `POST` i unesite URL `http://localhost:3000/naruci`. U tijelo zahtjeva unesite JSON objekt s podacima o narudžbi:

```
{
  "pizza": "Capricciosa",
  "velicina": "jumbo"
}
```

Trebali biste dobiti poruku: `Vaša narudžba za Capricciosa (jumbo) je uspješno zaprimljena!`.

Vježba 1 - Naručivanje više pizze

Nadogradite Express poslužitelj na način da pohranjujete podatke o narudžbama "in-memory", odnosno u varijablu koja će se resetirati svaki put kada se poslužitelj ponovno pokrene.

Nadogradite POST rutu `/naruci` tako da očekuje od korisnika **polje objekata** s podacima o narudžbi. Svaki objekt mora sadržavati ključeve `pizza`, `velicina` i `kolicina`.

```
[
  {
    "pizza": "Capricciosa",
    "velicina": "jumbo",
    "kolicina": 1
  },
  {
    "pizza": "Vegetariana",
    "velicina": "srednja",
    "kolicina": 2
  }
]
```

Ako neki od ključeva nedostaje, vratite korisniku poruku da nije poslao sve potrebne podatke.

Provjerite je li korisnik naručio pizzu koja postoji u vašem jelovniku. Ako korisnik naruči pizzu koja ne postoji, vratite korisniku poruku da jedna ili više pizza koju je naručio ne postoji

Ako korisnik pošalje podatke u ispravnom formatu, dodajte narudžbu u listu narudžbi i vratite korisniku poruku da je narudžba za pizze (izlistajte naručene nazive pizza) uspješno zaprimljena.

Vježba 2 - Zanima nas i adresa dostave

Nadogradite POST rutu `/naruci` tako da očekuje od korisnika dodatne podatke o narudžbi, kao što su `prezime`, `adresa` i `broj_telefona`.

Na jednak način kao u vježbi 1, provjerite jesu li svi potrebni podaci poslani i jesu li sve pizze koje je korisnik naručio prisutne u vašem jelovniku.

Primjer JSON objekta koji se šalje:

```
{
  "narudzba": [
    {
      "pizza": "Capricciosa",
      "velicina": "jumbo",
      "kolicina": 1
    },
    {
      "pizza": "Vegetariana",
      "velicina": "srednja",
      "kolicina": 2
    }
  ],
  "klijent": [
    "prezime": "Perić",
    "adresa": "Alda Negrija 6",
    "broj_telefona": "0912345678"
  ]
}
```

Ako korisnik pošalje podatke u ispravnom formatu, dodajte narudžbu u listu narudžbi i vratite korisniku `JSON` poruku sa sljedećim podacima:

```
message: "Vaša narudžba za pizza_1_naziv (pizza_1_velicina) i pizza_2_naziv (pizza_2_naziv)  
je uspješno zaprimljena!",  
prezime: "Perić",  
adresa: "Alda Negrija 6",  
ukupna_cijena: izračunajte ukupnu cijenu narudžbe
```

2.3 PUT i PATCH metode

Sljedeće metode koje ćemo naučiti su `PUT` i `PATCH` metode. Obe metode se koriste za **ažuriranje resursa** na poslužitelju. Međutim, razlika između njih je u tome što `PUT` metoda **zamjenjuje cijeli resurs** novim podacima, dok `PATCH` metoda **ažurira samo određene dijelove resursa**.

2.3.1 PUT metoda

Krenimo s metodom `PUT`. Zahtjev s ovom HTTP metodom se koristi za ažuriranje cijelog resursa na poslužitelju. Kada klijent pošalje ovakav zahtjev, želi zamijeniti cijeli resurs novim podacima koje šalje u **tijelu zahtjeva**.

Ključni elementi:

- **zamjenjuje cijeli resurs:** Kada šaljete `PUT` zahtjev, poslužitelj očekuje da ćete uključiti **sve informacije** za taj resurs, čak i onda kada želite zamijeniti samo manji dio resursa (npr. nekoliko polja u objektu).
- **može se koristiti za stvaranje novog resursa:** Ako šaljete `PUT` zahtjev s podacima o resursu koji ne postoji, ovaj zahtjev se može koristiti za stvaranje novog resursa. Zašto? Zato što se u `URI` navodi identifikator resursa.

Primjer: Recimo da želite ažurirati podatke o pizzi s `id`-om 1. Slanjem `PUT` zahtjeva na `/pizze/1` poslužitelj očekuje da ćete poslati **sve podatke** o pizzi, uključujući `id`, `naziv`, `cijena`, itd.

```
const pizze = [  
  { id: 1, naziv: 'Margherita', cijena: 6.5 },  
  { id: 2, naziv: 'Capricciosa', cijena: 8.0 },  
  { id: 3, naziv: 'Quattro formaggi', cijena: 10.0 },  
  { id: 4, naziv: 'Šunka sir', cijena: 7.0 },  
  { id: 5, naziv: 'Vegetariana', cijena: 9.0 }  
];
```

Zahtjev bi dakle izgledao ovako:

```
curl -X PUT http://localhost:3000/pizze/1 -H "Content-Type: application/json" -d '{"id": 1,  
"naziv": "Margherita", "cijena": 7.0}'
```

Primijetite da smo ažurirali samo cijenu Margherite, ali smo morali poslati sve podatke o pizzi.

Rekli smo da možemo koristiti `PUT` metodu i za stvaranje novog resursa, s obzirom da se u `URI` navodi identifikator resursa, a u tijelu zahtjeva šaljemo sve podatke o resursu.

Primjer:

```
curl -X PUT http://localhost:3000/pizze/6 -H "Content-Type: application/json" -d '{"id": 6, "naziv": "Quattro stagioni", "cijena": 8.0}'
```

Ako možemo koristiti `PUT` metodu za stvaranje novog resursa, zašto onda koristimo `POST` metodu? 😊

U pravilu želimo koristiti `POST` metodu za stvaranje novog resursa. Zašto? Iako je moguće koristiti `PUT` metodu, primijetite da smo morali poslati sve podatke o resursu, uključujući `id`. Ako korisnik šalje podatke o resursu, ne bi trebao znati `id` resursa, **već bi ga trebao generirati poslužitelj**.

Kako bi izgledao `POST` zahtjev za dodavanje nove pizze u naš jelovnik? Uočite da ne šaljemo `id` pizze, već samo `naziv` i `cijenu`. Također pogledajte `URI` zahtjeva.

```
curl -X POST http://localhost:3000/pizze -H "Content-Type: application/json" -d '{"naziv": "Quattro stagioni", "cijena": 8.0}'
```

U Expressu možemo jednostavno definirati `PUT` rutu sljedećom sintaksom:

```
app.put(PATH, (req, res) => {
  // Ovdje pišemo kod koji će se izvršiti kada korisnik pošalje PUT zahtjev na PATH
});
```

Dakle sintaksa je ista kao i za `GET` i `POST` rute, samo što koristimo `app.put` umjesto `app.get` ili `app.post`.

Primjer metode `PUT` za ažuriranje podataka o pizzi:

```
app.put('/pizze/:id', (req, res) => {
  const id_pizza = req.params.id;
  const nova_pizza = req.body;
  nova_pizza.id = id_pizza; // dodajemo id pizze u objekt, u slučaju da ga klijent nije poslao u tijelu zahtjeva

  const index = pizze.findIndex(pizza => pizza.id == id_pizza);

  if (index !== -1) {
    pizze[index] = nova_pizza;
    res.json(pizze[index]);
  } else {
    res.json({ message: 'Pizza s traženim ID-em ne postoji.' });
  }
});
```

2.3.2 `PATCH` metoda

`PATCH` metoda se koristi za **ažuriranje dijelova resursa** na poslužitelju. Za razliku od `PUT` metode koja zamjenjuje cijeli resurs, `PATCH` metoda se koristi kada želimo ažurirati samo **određene dijelove resursa**.

Primjer: Ako želimo ažurirati samo cijenu pizze s `id`-om 1, koristit ćemo `PATCH` metodu:

```
curl -X PATCH http://localhost:3000/pizze/1 -H "Content-Type: application/json" -d
'{"cijena": 7.0}'
```

Metodu `PATCH` ne želimo koristiti za stvaranje novog resursa, jer ne želimo stvoriti resurs s nepotpunim podacima. Primjerice, ako korisnik pošalje `PATCH` zahtjev na `/pizze/6`, a zaboravi poslati `naziv` pizze, ne želimo stvoriti novu pizzu s nepotpunim podacima.

U Expressu možemo definirati `PATCH` rutu na sljedeći način:

```
app.patch(PATH, (req, res) => {
  // Ovdje pišemo kod koji će se izvršiti kada korisnik pošalje PATCH zahtjev na PATH
});
```

Primjer metode `PATCH` za ažuriranje podataka o pizzi:

```
app.patch('/pizze/:id', (req, res) => {
  const id_pizza = req.params.id;
  const nova_pizza = req.body;

  const index = pizze.findIndex(pizza => pizza.id == id_pizza);

  if (index !== -1) {
    for (const key in nova_pizza) {
      pizze[index][key] = nova_pizza[key];
    }

    // ili
    // pizze[index] = { ...pizze[index], ...nova_pizza }; // spread operator

    res.json(pizze[index]);
  } else {
    res.json({ message: 'Pizza s traženim ID-em ne postoji.' });
  }
});
```

2.4 `DELETE` metoda

Metoda `DELETE` se koristi za **brisanje resursa** na poslužitelju. Kada klijent pošalje ovakav zahtjev, poslužitelj briše resurs s identifikatorom koji je naveden u `URI` zahtjeva.

Primjer: Ako želimo obrisati pizzu s `id`-om 1, koristit ćemo `DELETE` metodu:

```
curl -X DELETE http://localhost:3000/pizze/1
```

U Expressu možemo definirati `DELETE` rutu na sljedeći način:

```
app.delete(PATH, (req, res) => {
  // Ovdje pišemo kod koji će se izvršiti kada korisnik pošalje DELETE zahtjev na PATH
});
```

Primjer metode `DELETE` za brisanje podataka o pizzi:

```
app.delete('/pizze/:id', (req, res) => {
  const id_pizza = req.params.id;

  const index = pizze.findIndex(pizza => pizza.id == id_pizza);

  if (index !== -1) {
    pizze.splice(index, 1);
    res.json({ message: 'Pizza uspješno obrisana.' });
  } else {
    res.json({ message: 'Pizza s traženim ID-em ne postoji.' });
  }
});
```

2.5 Kada koristiti koju `HTTP` metodu?

Naučili smo kako koristiti sljedeće `HTTP` metode:

- `GET` - dohvati resurs (npr. `GET /pizze` ili `GET /pizze/1` ili `GET /narudzbe`)
- `POST` - stvori novi resurs (npr. `POST /pizze` ili `POST /narudzbe` ili `POST /login` s podacima za autentifikaciju)
- `PUT` - zamijeni cijeli resurs (npr. `PUT /pizze/1` ili `PUT /korisnici/1` s cijelim podacima o resursu)
- `PATCH` - ažuriraj dio resursa (npr. `PATCH /pizze/1` ili `PATCH /korisnici/1` s parcijalnim podacima o resursu)
- `DELETE` - obriši resurs (npr. `DELETE /pizze/1` ili `DELETE /korisnici/1` bez tijela zahtjeva)

Postoje još metode koje nismo spomenuli, kao što su `HEAD`, `OPTIONS`, `TRACE`, `CONNECT` itd. Međutim, ove metode su manje uobičajene i koriste se u specifičnim situacijama. Vi ih ne morate znati koristiti.

Iako je moguće koristiti bilo koju metodu za gotovo bilo koju akciju, ipak postoje pravila i dobre prakse koje se koriste u razvoju web aplikacija. Evo nekoliko smjernica:

- `GET` metodu koristimo za dohvat resursa s poslužitelja. Ova metoda ne bi trebala imati nikakve druge efekte osim dohvata podataka. Primjerice, ako korisnik posjeti URL u pregledniku, očekujemo da će dobiti odgovor s podacima, ali ne očekujemo da će se nešto promijeniti na poslužitelju (npr. ažurirati podaci u bazi podataka).
- `POST` metodu koristimo za stvaranje novog resursa na poslužitelju. Ova metoda se koristi kada korisnik želi poslati podatke na poslužitelj, npr. kada korisnik želi stvoriti novu pizzu u našem jelovniku. Međutim, metodu koristimo i za druge akcije, poput autentifikacije korisnika kada korisnik želi u tijelu zahtjeva poslati korisničko ime i lozinku (prisjetimo se da je kod GET zahtjeva sve vidljivo u URL-u).
- `PUT` metodu koristimo za zamjenu cijelog resursa novim podacima. Ova metoda se koristi kada korisnik želi zamijeniti cijeli resurs novim podacima. Primjerice, kada korisnik želi ažurirati podatke o pizzi, ali mora poslati sve podatke o pizzi, uključujući `id` pa i one podatke koji se ne mijenjaju.
- `PATCH` metodu koristimo za ažuriranje dijelova resursa. Ova metoda se koristi kada korisnik želi ažurirati samo određene dijelove resursa. Primjerice, kada korisnik želi ažurirati samo cijenu pizze, a ne i naziv pizze.

- **DELETE** metodu koristimo za brisanje resursa. Ova metoda se koristi kada korisnik želi obrisati resurs s poslužitelja. Primjerice, kada korisnik želi obrisati pizzu iz našeg jelovnika.

3. Router objekt

Prilikom razvoja ozbiljnijeg poslužitelja, vjerojatno ćemo morati definirati mnoštvo različitih ruta. Možemo vidjeti da naša `index.js` datoteka postaje sve veća i veća kako dodajemo nove rute.

Na primjer, za jednostavno dohvaćanje pizze i pizze po ID-u, potrebne su nam dvije rute:

```
app.get('/pizze', (req, res) => {
  // implementacija
});

app.get('/pizze/:id', (req, res) => {
  // implementacija
});
```

Što ako imamo još više ruta? Na primjer, rute za naručivanje pizze, ažuriranje podataka o pizzi, brisanje pizze, itd. Naša datoteka `index.js` postaje sve veća i teže ju je održavati.

Kako bismo olakšali organizaciju koda, poželjno je koristiti `Router` objekt koji nam omogućuje grupiranje ruta i definiranje ruta u zasebnim datotekama.

`Router` objekt jedna je od ključnih komponenti Expressa koja nam omogućuje grupiranje srodnih ruta. Na primjer, sve rute vezane uz pizze možemo grupirati u jedan `Router` objekt, ili sve rute vezane uz korisnike u drugi `Router` objekt.

3.1 Kako koristiti Router objekt?

Naš trenutni poslužitelj sastoji se od sljedećih datoteka:

```
.
├── index.js
├── node_modules
├── package-lock.json
└── package.json
```

Praktično je organizirati naše rute u zasebne datoteke. Na primjer, možemo imati datoteku `pizze.js` u kojoj ćemo definirati sve rute vezane uz pizze, ili datoteku `narudzbe.js` gdje ćemo definirati sve rute vezane uz narudžbe.

Dodatno, te datoteke možemo pohraniti u zajednički direktorij `routes` ili `router`.

Dodajmo direktorij `routes` u naš projekt i datoteku `pizze.js` unutar tog direktorija:

```
.
```

```
├── index.js
├── node_modules
├── package-lock.json
├── package.json
└── routes
    └── pizze.js
```

Unutar `pizze.js` datoteke moramo uključiti ponovo Express modul, ali i definirati `Router` objekt:

```
const express = require('express');
const router = express.Router();
```

Kako bi naš `Router` objekt bio dostupan u `index.js` datoteci, moramo ga izvesti:

```
const express = require('express');
const router = express.Router();

module.exports = router;
```

Međutim, dok nismo napisali puno koda, nije loše da se napokon prebacimo na novu ES6 sintaksu koju ste vjerojatno već pisali u VUE.js aplikacijama. **ECMAScript** (JavaScript ES) je standardizacija JavaScripta, a **ES6** je šesta verzija standarda koja je donijela puno novih značajki, uključujući i modernu sintaksu za organizaciju i strukturiranje modula.

U ES6 sintaksi, umjesto `module.exports` koristimo `export default`:

`export default` sintaksa omogućava nam izvoz jednog objekta, funkcije ili varijable iz modula. Kada koristimo `export default`, možemo uvesti taj objekt, funkciju ili varijablu u drugom modulu koristeći `import` sintaksu bez vitičastih zagrada (odnosno bez navođenja imena objekta kojeg uvozimo).

Prije svega, moramo ažurirati našu `package.json` datoteku kako bismo koristili ES6 sintaksu. Dodajte sljedeći redak u `package.json` datoteku:

```
"type": "module",
```

Sada možemo koristiti ES6 sintaksu u našem projektu. Idemo prvo ispraviti `index.js` datoteku.

U `index.js` datoteci, umjesto `require` koristimo `import` sintaksu:

```
import object from 'module'; // umjesto const object = require('module');

// odnosno

import express from 'express'; // umjesto const express = require('express');
```

Ako imamo više izvoza iz jednog modula, možemo ih uvesti koristeći vitičaste zagrade:

```
import { object1, object2 } from 'module'; // umjesto const { object1, object2 } =
require('module');
```

Vratimo se na `pizze.js`, gdje ćemo također koristiti ES6 sintaksu:

```
import express from 'express';
const router = express.Router();

export default router;
```

Kako ovaj `Router` objekt možemo zamisliti kao malu aplikaciju unutar naše glavne aplikacije, možemo dodati rute na isti način kao što smo to radili u `index.js` datoteci, ali ćemo umjesto `app` koristiti `router`:

Idemo dodati rutu za dohvatzanje svih pizza:

```
import express from 'express';
const router = express.Router();

const pizze = [
  { id: 1, naziv: 'Margerita', cijena: 7.0 },
  { id: 2, naziv: 'Capricciosa', cijena: 9.0 },
  { id: 3, naziv: 'Šunka sir', cijena: 8.0 },
  { id: 4, naziv: 'Vegetariana', cijena: 12.0 },
  { id: 5, naziv: 'Quattro formaggi', cijena: 15.0 }
];

router.get('/', (req, res) => {
  // ruta za dohvatzanje svih pizza, pišemo router.get umjesto app.get
  res.json(pizze);
});

export default router;
```

Na isti način kopirajte i rutu za dohvatzanje pizze po ID-u.

Jednom kad to imamo, možemo uvesti `Router` objekt u našu `index.js` datoteku s proizvoljnim imenom:

```
import pizzeRouter from './routes/pizze.js';
```

Zatim samo moramo reći našoj aplikaciji da koristi taj `Router` objekt:

```
app.use(pizzeRouter);
```

To je to! Testirajte dohvaćanje svih pizza i pizze po ID-u koristeći Postman ili Thunder Client.

3.2 Idemo još bolje strukturirati našu aplikaciju

Kako bismo još bolje strukturirali naš poslužitelj, možemo napraviti još nekoliko stvari.

Prvo, kopirajmo preostale `/pizze` rute iz `index.js` datoteke u `pizze.js` datoteku.

Dakle, u `pizze.js` datoteci imamo sljedeće rute:

```
router.get('/pizze');
router.get('/pizze/:id');
router.put('/pizze/:id');
router.patch('/pizze/:id');
app.delete('/pizze/:id');
```

Što je redundantno u ovim rutama? 🤔

► Spoiler alert! Odgovor na pitanje

To ćemo definirati u `index.js` datoteci kada koristimo `pizzeRouter`:

```
app.use('/pizze', pizzeRouter);
```

Sada naše rute u `pizze.js` datoteci mijenjamo na sljedeći način:

```
router.get('/');
router.get('/:id');
router.put('/:id');
router.patch('/:id');
router.delete('/:id');
```

Vježba 3 - Strukturiranje narudžbi ➡️🍕

Strukturirajte narudžbe na jednak način kao što smo to napravili za pizze. Definirajte `narudzbe.js` datoteku unutar `routes` direktorija i prebacite polje narudžbi i sve rute vezane uz narudžbe u tu datoteku.

Kako se radi o resursu `narudzbe`, prefiks `/narudzbe` dodajte samo jednom, na početku svih ruta. Dakle `URI` za dodavanje narudžbe trebao bi izgledati ovako: `/narudzbe`, a ne `/narudzbe/naruci` ili samo `/naruci`.

Kada završite, uvezite `narudzbeRouter` u `index.js` datoteku i koristite ga u aplikaciji. Vaša `index.js` datoteka trebala bi izgledati ovako:

```
import express from 'express';
import pizzeRouter from './routes/pizze.js';
import narudzbeRouter from './routes/narudzbe.js';

const app = express();

const PORT = 3000;

app.use(express.json());

app.use('/pizze', pizzeRouter);
app.use('/narudzbe', narudzbeRouter);

app.listen(PORT, error => {
  if (error) {
    console.error(`Greška prilikom pokretanja poslužitelja: ${error.message}`);
  } else {
    console.log(`Server dela na http://localhost:${PORT}`);
  }
});
```

```
    }  
});
```

4. Statusni kodovi u odgovorima

Statusni kodovi (eng. *HTTP status codes*) su brojevi koji se koriste u **HTTP odgovorima** kako bi klijentu dali informaciju u kojem je stanju zahtjev koji je poslao. Drugim riječima, ako klijent pošalje zahtjev koji rezultira greškom, poslužitelj uz odgovarajuću poruku vraća i statusni kod koji označava vrstu greške.

Ako se podsjetimo statusnih kodova iz prve skripte, rekli smo da ih možemo podijeliti u sljedeće kategorije:

- **1xx** (100 - 199) - Informacijski odgovori (eng. *Informational responses*): Poslužitelj je primio zahtjev te ga i dalje obrađuje
- **2xx** (200 - 299) - Odgovori uspjeha (eng. *Successful responses*): Zahtjev klijenta uspješno primljen i obrađen
- **3xx** (300 - 399) - Odgovori preusmjeravanja (eng. *Redirection messages*): Ova skupina kodova govori klijentu da mora poduzeti dodatne radnje kako bi dovršio zahtjev
- **4xx** (400 - 499) - Greške na strani klijenta (eng. *Client error responses*): Sadrži statusne kodove koji se odnose na greške nastale na klijentskoj strani
- **5xx** (500 - 599) - Greške na strani poslužitelja (eng. *Server error responses*): Sadrži statusne kodove koji se odnose na greške nastale na poslužiteljskoj strani

Statusni kodovi neizbjegljivi su dio svakog HTTP standarda, a njihovom primjenom standardiziramo komunikaciju između klijenta i poslužitelja. Na taj način, klijent može interpretirati odgovor poslužitelja i ovisno o statusnom kodu poduzeti odgovarajuće radnje.

Na primjer, ako pošaljemo klijentu JSON poruku `message: "Pizza nije pronađena"` ili `message: "Greška prilikom obrade narudžbe"`, potrebno je posebno tumačiti te poruke na klijentskoj strani. Međutim, to ne želimo raditi, jer bi svaki programer mogao interpretirati poruke na svoj način.

Statusni kodovi su standardizirani i svaki statusni kod ima svoje značenje. Na primjer, statusni kod `404` označava da resurs nije pronađen (prvi slučaj), dok statusni kod `500` označava općenitu grešku na poslužitelju (drugi slučaj).

4.1 Kako koristiti statusne kodove u Expressu?

U Expressu možemo dati statusne kodove u odgovorima koristeći `res.status()` metodu. Ova metoda postavlja statusni kod odgovora na poslužitelju.

Primjer postavljanja statusnog koda `200` (*OK*) u odgovoru:

```
app.get('/pizze', (req, res) => {  
  res.status(200); // postavljanje statusnog koda 200 koji označava uspješan odgovor (OK)  
});
```

Na metodu `res.status()` možemo nadovezati metodu `res.send()` ili `res.json()` kako bismo poslali podatkovni odgovor klijentu:

```
app.get('/pizze', (req, res) => {
  res.status(200).json(pizze); // poslati sve pizze kao JSON odgovor s statusnim kodom 200
});
```

Što ako **poslužitelj** ne može pronaći resurs **koji je korisnik zatražio**? U tom slučaju, možemo poslati statusni kod 404 (*Not Found*):

```
app.get('/pizze/:id', (req, res) => {
  const id_pizza = req.params.id;
  const pizza = pizze.find(pizza => pizza.id == id_pizza);

  if (pizza) {
    res.status(200).json(pizza);
  } else {
    res.status(404).json({ message: 'Pizza nije pronađena.' });
  }
});
```

Koji ćemo statusni kod poslati klijentu ako korisnik pošalje zahtjev s neispravnim podacima? Na primjer, ako korisnik pošalje kao parametar `id` slovo umjesto broja? U tom slučaju, možemo poslati statusni kod 400 (*Bad Request*):

```
router.get('/:id', (req, res) => {
  const id_pizza = req.params.id;

  if (isNaN(id_pizza)) {
    return res.status(400).json({ message: 'ID pizze mora biti broj.' }); // poslati statusni
  kod 400 ako ID pizze nije broj
  }

  const pizza = pizze.find(pizza => pizza.id == id_pizza);

  if (pizza) {
    return res.status(200).json(pizza); // poslati statusni kod 200 ako je pizza pronađena
  } else {
    return res.status(404).json({ message: 'Pizza nije pronađena.' }); // poslati statusni
  kod 404 ako pizza nije pronađena
  }
});
```

Statusnih kodova ima mnogo, a svaki od njih ima svoje značenje. Možete pronaći **popis i definicije svih statusnih kodova** na [ovoj poveznici](#).

Međutim, u praksi se ne najčešće ne koriste svi statusni kodovi, već nekolicina njih. Evo nekoliko najčešće korištenih statusnih kodova:

- 200 - OK: Zahtjev je uspješno primljen i obrađen (npr. GET zahtjev za dohvata svih pizza)
- 201 - Created: Resurs je uspješno stvoren (npr. nakon slanja POST zahtjeva)
- 400 - Bad Request: Zahtjev nije moguće obraditi zbog neispravnih podataka (npr. korisnik je poslao neispravan ID pizze prilikom narudžbe)

- 404 - Not Found: Resurs nije pronađen (npr. korisnik je poslao ID pizze koja ne postoji)
- 500 - Internal Server Error: Opća greška na poslužitelju (npr. greška prilikom obrade narudžbe, najvjerojatnije zbog greške u kodu na poslužitelju)

Postoji puno varijacija 4xx, 5xx i 2xx statusnih kodova, pa tako imamo:

- 401 - Unauthorized: Korisnik nije autoriziran za pristup resursu (npr. korisnik nema prava pristupa resursu jer nije prijavljen)
- 204 - No Content: Zahtjev je uspješno primljen i obrađen, ali nema sadržaja za prikazati (npr. nakon brisanja resursa)
- 403 - Forbidden: Korisnik nema prava pristupa resursu (npr. korisnik nema prava pristupa resursu jer nije administrator)
- 301 - Moved Permanently: Resurs je trajno premješten na novu lokaciju (npr. kada se mijenja URL resursa)
- 503 - Service Unavailable: Poslužitelj nije dostupan (npr. poslužitelj je preopterećen)
- 409 - Conflict: Zahtjev nije moguće obraditi zbog konflikta (npr. korisnik pokušava ažurirati resurs koji je već ažuriran, npr. kod PUT/PATCH zahtjeva)

Vježba 4 - Korištenje statusnih kodova u pizzeriji



Dodajte **statusne kodove** u odgovore vašeg poslužitelja za sve rute vezane uz pizze i narudžbe.

Pokušajte koristiti što semantički ispravnije statusne kodove. Na primjer, ako korisnik pokuša dohvatiti pizzu koja ne postoji, pošaljite statusni kod 404 (*Not Found*), ali ako korisnik pošalje neispravan ID pizze, pošaljite statusni kod 400 (*Bad Request*).

Dodatno, dodajte 3 nove rute u vašu pizzeriju:

- dohvaćanje svih narudžbi
- dohvaćanje narudžbe po ID-u
- brisanje narudžbe po ID-u

Kada završite, testirajte sve rute koristeći Postman ili Thunder Client, a zatim provjerite statusne kodove u odgovorima koje ste dobili.

Samostalni zadatak za Vježbu 2

Definirajte novi Express projekt u kojem ćete implementirati jednostavni poslužitelj za agenciju za nekretnine.

Osmislite dizajn poslužitelja, a podatke spremajte u polje objekata, odnosno *in-memory*. **Podaci o nekretninama** trebaju sadržavati sljedeće informacije:

- ID nekretnine
- Naziv nekretnine
- Opis nekretnine
- Cijena nekretnine
- Lokacija nekretnine

- `Broj soba`
- `Površina nekretnine`
- `Cijena nekretnine`

Implementirajte sljedeće rute:

- dohvati sve nekretnine
- dohvati nekretninu po ID-u
- dodaj novu nekretninu
- ažuriraj nekretninu potpuno
- ažuriraj nekretninu djelomično
- obriši nekretninu
- pošalji novu ponudu

Ponude spremajte na jednak način u polje objekata, a **svaka ponuda mora sadržavati**:

- `ID ponude`
- `ID nekretnine`
- `Ime kupca`
- `Prezime kupca`
- `Ponuđena cijena`
- `Broj telefona kupca`

Dodajte slične **provjere** kao u pizzeriji, primjerice:

- provjerite jesu li ID-evi brojevi, ako ne vratite odgovarajući statusni kod i poruku
- provjerite jesu li svi podaci poslani u tijelu zahtjeva, ako nisu vratite odgovarajući statusni kod i poruku
- provjerite jesu li svi podaci ispravni, npr. cijena nekretnine ne može biti negativna, broj soba ne može biti negativan, itd.
- prilikom izrade ponude, provjerite postoji li uopće nekretnina s navedenim ID-em

Rute za **nekretnine i ponude grupirajte u zasebne Router objekte** i organizirajte ih u zasebnim datotekama unutar `routes` direktorija. Koristite statusne kodove u odgovorima.

Za testiranje koristite Postman ili Thunder Client.

Web aplikacije (WA)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



(3) Komunikacija s klijentskom stranom

#3

WA

Do sad smo vidjeli kako izraditi Express poslužitelj i osnovne rute koje odgovaraju na HTTP zahtjeve. Dosad smo HTTP zahtjeve slali pomoću web preglednika i HTTP klijenata. Naučili smo dijelove HTTP zahtjeva i odgovora, kako slati i očekivati JSON podatke kroz odgovor, parametre zahtjeva i statusne kodove. Vidjeli smo i kako strukturirati aplikaciju kroz Express Router objekt. U ovoj skripti napokon ćemo spojiti naš Express poslužitelj s klijentskom stranom. Preciznije, realizirat ćemo komunikaciju s Vue razvojnim poslužiteljem kroz Axios HTTP klijent.

Posljednje ažurirano: 11.11.2024.

Sadržaj

- [Web aplikacije \(WA\)](#)
- [\(3\) Komunikacija s klijentskom stranom](#)
 - [Sadržaj](#)
- [1. Postavljanje Express poslužitelja](#)
 - [1.1 Definiranje osnovnih ruta](#)
 - [1.2 Router za proizvode](#)
 - [1.3 Router za narudžbe](#)
- [2. Izrada jednostavne vue3 aplikacije s Tailwind CSS](#)
 - [2.1 Konfiguracija projekta](#)
 - [2.2 Struktura projekta i testiranje](#)
 - [2.3 Komponenta za prikaz proizvoda](#)
- [3. Axios HTTP klijent](#)
 - [3.1 Slanje GET zahtjeva](#)
 - [3.1.1 CORS policy](#)
 - [3.1.2 Prikazivanje proizvoda na frontendu](#)

- [3.1.3 Dodavanje podataka na backendu](#)
 - [3.2 Slanje POST zahtjeva](#)
- [Samostalni zadatak za Vježbu 3](#)

1. Postavljanje Express poslužitelja

Krenimo s definiranjem osnovnog Express poslužitelja. Izradit ćemo jednostavni web shop jedne **male trgovine koja prodaje odjeću i obuću**.

Do sad smo već naučili kako izraditi node projekt i definirati Express poslužitelj uz relevantne rute na primjeru aplikacije za naručivanje pizze. U ovom primjeru ćemo izraditi potpuno novi poslužitelj. Kako ćemo imati i klijentsku stranu, definirat ćemo zasebne direktorije za klijentsku (`vue`) i serversku stranu (`server`). Sve možemo definirati unutar direktorija `webshop`.

```
webshop
└── server
    └── vue
```

Unutar `server` direktorija ćemo izraditi novi node projekt, instalirati Express i definirati osnovni poslužitelj.

```
import express from 'express';

const app = express();

app.use(express.json());

const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Webshop API');
});

app.listen(PORT, error => {
  if (error) {
    console.error(`Greška prilikom pokretanja poslužitelja: ${error.message}`);
  } else {
    console.log(`Server dela na http://localhost:${PORT}`);
  }
});
```

1.1 Definiranje osnovnih ruta

Želimo da korisnik može pregledati sve proizvode u trgovini, pogledati detalje o pojedinom proizvodu te napraviti narudžbu.

Prema tome, prvo ćemo osmisiliti backend dizajn naše aplikacije:

- GET `/proizvodi` - dohvata sve proizvode
- GET `/proizvodi/:id` - dohvata proizvod s određenim ID-om
- POST `/narudzba` - stvara novu narudžbu

So far, so good. Sada ćemo definirati boilerplate za rute za svaku od ovih akcija:

```

app.get('/proizvodi', (req, res) => {
  res.send('Dohvati sve proizvode');
});

app.get('/proizvodi/:id', (req, res) => {
  const id_req = req.params.id;
  res.send(`Dohvati proizvod s ID: ${id_req}`);
});

app.post('/narudzba', (req, res) => {
  res.send('Napravi novu narudžbu');
});

```

Proizvode ćemo spremati *in-memory*, tj. u polju objekata. Možemo definirati konstruktor `Proizvod` koji će nam pomoći pri stvaranju novih proizvoda.

Definirat ćemo i polje `proizvodi` te u njega pohraniti nekoliko instanciranih `Proizvod` objekata.

```

function Proizvod(id, naziv, cijena, velicine) {
  this.id = id;
  this.naziv = naziv;
  this.cijena = cijena;
  this.velicine = velicine;
}

const proizvodi = [
  new Proizvod(1, 'Obična crna majica', 100, ['XS', 'S', 'M', 'L']),
  new Proizvod(2, "Levi's 501 traperice", 110, ['S', 'M', 'L']),
  new Proizvod(3, 'Zimska kapa', 40, 'onesize'),
  new Proizvod(4, 'Čarape Adidas', 20, ['34-36', '37-39', '40-42']),
  new Proizvod(5, 'Tenisice Nike', 200, ['38', '39', '40', '41', '42', '43', '44', '45'])
];

```

Sada možemo definirati logiku za dohvaćanje **svih proizvoda** i **proizvoda s određenim ID-om**:

```

app.get('/proizvodi', (req, res) => {
  res.status(200).json(proizvodi);
});

app.get('/proizvodi/:id', (req, res) => {
  const id_req = req.params.id;

  if (isNaN(id_req)) {
    res.status(400).send('ID mora biti broj');
    return;
  }

  const proizvod = proizvodi.find(proizvod => proizvod.id == id_req);

  if (proizvod) {
    res.status(200).json(proizvod);
  } else {

```

```
    res.status(404).send('Proizvod nije pronađen');
  }
});
```

Testirajte poslužitelj koristeći HTTP klijent ili web preglednik.

1.2 Router za proizvode

Nakon što ste definirali i testirali obje rute, definirat ćemo **Express Router** u datoteci `proizvodi.js`. Struktura projekta će izgledati ovako:

```
webshop
├── server
│   ├── index.js
│   ├── node_modules
│   ├── package-lock.json
│   ├── package.json
│   └── routes
│       └── proizvodi.js
└── vue
```

Definirajmo Express Router u datoteci `proizvodi.js` i prebacimo logiku iz `index.js` datoteke kako smo naučili na prethodnim vježbama.

```
// routes/proizvodi.js

import express from 'express';

const router = express.Router();

function Proizvod(id, naziv, cijena, velicine) {
    this.id = id;
    this.naziv = naziv;
    this.cijena = cijena;
    this.velicine = velicine;
}

const proizvodi = [
    new Proizvod(1, 'Obična crna majica', 100, ['XS', 'S', 'M', 'L']),
    new Proizvod(2, "Levi's 501 traperice", 110, ['S', 'M', 'L']),
    new Proizvod(3, 'Zimska kapa', 40, 'onesize'),
    new Proizvod(4, 'Čarape Adidas', 20, ['34-36', '37-39', '40-42']),
    new Proizvod(5, 'Tenisice Nike', 200, ['38', '39', '40', '41', '42', '43', '44', '45'])
];

router.get('/', (req, res) => {
    res.status(200).json(proizvodi);
});

router.get('/:id', (req, res) => {
    const id_req = req.params.id;

    if (isNaN(id_req)) {
        res.status(400).send('ID mora biti broj');
        return;
    }
```

```
const proizvod = proizvodi.find(proizvod => proizvod.id == id_req);
if (proizvod) {
  res.status(200).json(proizvod);
} else {
  res.status(404).send('Proizvod nije pronađen');
}
});

export default router;
```

Kako smo uklonili prefikse `/proizvodi` i `/proizvodi/:id`, moramo prefiks definirati prilikom uključivanja router objekta u `index.js` datoteku.

```
// index.js

import proizvodiRouter from './routes/proizvodi.js';

app.use('/proizvodi', proizvodiRouter);
```

Testirajte poslužitelj koristeći HTTP klijent ili web preglednik.

1.3 Router za narudžbe

Definirat ćemo još jedan router za narudžbe, za sada samo s jednom rutom koja stvara novu narudžbu.

```
// routes/narudzbe.js

import express from 'express';

const router = express.Router();

router.post('/', (req, res) => {
  res.send('Napravi novu narudžbu');
});

export default router;
```

Na jednak način kao i za proizvode, uključite i router za narudžbe u `index.js` datoteku.

Možemo definirati konstruktor `Narudzba` koji će nam pomoći pri stvaranju novih narudžbi.

```
// routes/narudzbe.js

function Narudzba(id, naruceni_proizvodi) {
  this.id = id;
  this.naruceni_proizvodi = naruceni_proizvodi;
}
```

Atribut `naruceni_proizvodi` će biti polje objekata koji sadrže: **ID proizvoda, veličinu i naručenu količinu**.

Primjer:

```
// routes/narudzbe.js

const narudzba = new Narudzba(1, [
  { id: 1, velicina: 'M', narucena_kolicina: 2 },
  { id: 3, velicina: 'onesize', narucena_kolicina: 1 }
]);
```

Možemo još dodati u konstruktor metodu `ukupnaCijena` koja će izračunati ukupnu cijenu narudžbe.

Kako smo narudžbu definirali kao polje objekata gdje svaki objekt sadrži ID proizvoda, veličinu i naručenu količinu, moramo za svaki naručeni proizvod pronaći odgovarajući proizvod iz polja `proizvodi`, pomnožiti cijenu proizvoda s naručenom količinom i zbrojiti sve proizvode. Iskoristit ćemo metodu `Array.reduce` za zbrajanje za bržu implementaciju.

```
// routes/narudzbe.js

function Narudzba(id, naruceni_proizvodi) {
  this.id = id;
  this.naruceni_proizvodi = naruceni_proizvodi;
  this.ukupnaCijena = function () {
```

```

let ukupno = this.naruceni_proizvodi.reduce((suma, trenutni_proizvod) => {
    let proizvod_obj = proizvodi.find(proizvod => proizvod.id == trenutni_proizvod.id);
    return suma + proizvod_obj.cijena * trenutni_proizvod.narucena_kolicina;
}, 0); // 0 je početna vrijednost sume
return ukupno;
};

}

const narudzba = new Narudzba(1, [
{ id: 1, velicina: 'M', narucena_kolicina: 2 },
{ id: 3, velicina: 'onesize', narucena_kolicina: 1 }
]);

// Testiramo metodu ukupnaCijena za narudžbu koja sadrži 2 majice i 1 kapu
console.log(narudzba.ukupnaCijena()); // 240

```

Obzirom da sad koristimo `ES6` sintaksu (definirali smo u prošloj skripti kroz `package.json`), nije loše ostati dosljedan pa možemo funkciju konstruktore pretvoriti u klase. Koristimo ključnu riječ `class` i metodu `constructor` za inicijalizaciju objekta, dok metodu `ukupnaCijena` možemo definirati kao `get` metodu klase jer nema potrebe da se poziva kao zasebna funkcija ili da se nalazi unutar konstruktora.

```

// routes/narudzbe.js
class Proizvod {
    constructor(id, naziv, cijena, velicine) {
        this.id = id;
        this.naziv = naziv;
        this.cijena = cijena;
        this.velicine = velicine;
    }
}

const proizvodi = [
    new Proizvod(1, 'Obična crna majica', 100, ['XS', 'S', 'M', 'L']),
    new Proizvod(2, "Levi's 501 traperice", 110, ['S', 'M', 'L']),
    new Proizvod(3, 'Zimska kapa', 40, 'onesize'),
    new Proizvod(4, 'Čarape Adidas', 20, ['34-36', '37-39', '40-42']),
    new Proizvod(5, 'Tenisice Nike', 200, ['38', '39', '40', '41', '42', '43', '44', '45'])
];

class Narudzba {
    constructor(id, naruceni_proizvodi) {
        this.id = id;
        this.naruceni_proizvodi = naruceni_proizvodi;
    }

    get ukupnaCijena() {
        let ukupno = this.naruceni_proizvodi.reduce((suma, currProizvod) => {
            let pronadeni_proizvod = proizvodi.find(p => p.id == currProizvod.id);
            console.log(pronadeni_proizvod);
            return suma + pronadeni_proizvod.cijena * currProizvod.narucena_kolicina;
        }, 0);
        return ukupno;
    }
}

```

```

}

// dummy narudžba
const narudzba = new Narudzba(1, [
  { id: 1, velicina: 'M', narucena_kolicina: 2 },
  { id: 3, velicina: 'onesize', narucena_kolicina: 1 }
]);

console.log(narudzba.ukupnaCijena()); // 240

```

No, sada se javlja problem ponavljanja koda u router objektima. Kako bismo to riješili, možemo izdvojiti konstruktor i polje za spremanje proizvoda i narudžbi u zasebnu datoteku `data.js`.

```

// server/data.js

class Proizvod {
  constructor(id, naziv, cijena, velicine) {
    this.id = id;
    this.naziv = naziv;
    this.cijena = cijena;
    this.velicine = velicine;
  }
}

const proizvodi = [
  new Proizvod(1, 'Obična crna majica', 100, ['XS', 'S', 'M', 'L']),
  new Proizvod(2, "Levi's 501 traperice", 110, ['S', 'M', 'L']),
  new Proizvod(3, 'Zimska kapa', 40, 'onesize'),
  new Proizvod(4, 'Čarape Adidas', 20, ['34-36', '37-39', '40-42']),
  new Proizvod(5, 'Tenisice Nike', 200, ['38', '39', '40', '41', '42', '43', '44', '45'])
];

```

Sada moramo još samo izvesti ove podatke iz `data.js` datoteke.

```

// server/data.js

export { Proizvod, proizvodi };

```

Za kraj ćemo još samo dodati POST rutu za izradu i pohranu nove narudžbe u polje narudžbi `narudzbe`.

```

// narudzbe.js

let narudzbe = [];

router.post('/', (req, res) => {
  let podaci = req.body;
  let naruceni_proizvodi = podaci.naruceni_proizvodi;

  if (!Array.isArray(naruceni_proizvodi) || naruceni_proizvodi.length == 0) {
    return res.status(400).json({ message: 'Nema podataka' });
  }
}

```

```
let latest_id = narudzbe.length ? narudzbe.at(-1).id + 1 : 1; // one-liner za generiranje  
ID-a  
  
let narudzba_obj = new Narudzba(latest_id, naruceni_proizvodi);  
  
narudzbe.push(narudzba_obj);  
  
return res.status(201).json(podaci); // vraćamo poslane podatke o narudžbi i statusni kod  
201 (Created) budući da smo stvorili novi resurs  
});
```

To je to za sada! Prebacujemo se na izradu klijentske strane (aka. frontend).

2. Izrada jednostavne Vue3 aplikacije s Tailwind CSS

2.1 Konfiguracija projekta

Definirali smo osnovni Express poslužitelj s rutama za proizvode i narudžbe. Sada ćemo izraditi jednostavnu [Vue.js](#) aplikaciju koja će komunicirati s poslužiteljem.

[Tailwind CSS](#) popularni je CSS razvojni okvir koji omogućuje brzo i jednostavno stiliziranje web stranica upotrebom predefiniranih CSS klasa. Njegova glavna prednost je što ne koristi gotove komponente već se stilovi definiraju direktno u HTML-u, a samim time omogućuje **veću fleksibilnost i prilagodbu**.

Koristit ćemo Tailwind 3 za potrebe naše aplikacije. **Napomena**, Tailwind CSS zahtjeva Node.js verziju 12.13.0 ili noviju.

Vue aplikaciju možemo instalirati i zasebno pa dodati Tailwind CSS, ali brže i jednostavnije je kroz [vite](#) i postojeći [Vue Tailwind template](#).

Upute za instalaciju nalaze se na sljedećoj poveznicu: <https://tailwindcss.com/docs/guides/vite#vue>

- **1. korak:** otvorite poveznicu iznad i provjerite da ste odabrali `using vue`. Zatim kopirajte naredbu za inicijalizaciju projekta s predloškom:

```
npm create vite@latest my-project -- --template vue
```

- **2. korak:** unutar direktorija projekta, pokrenite `npm install` kako biste instalirali sve potrebne pakete:

```
cd my-project
```

```
npm install
```

Nakon toga možete (ali i ne morate) prebaciti sadržaj gdje se nalazi vue projekt u `webshop/vue` direktorij, obzirom da smo sada dobili novi direktorij `my-project`.

- **3. korak:** instalirajte `Tailwind CSS`, `PostCSS` i `Autoprefixer` pakete koji su potrebni za ispravan rad Tailwinda:

```
npm install -D tailwindcss postcss autoprefixer
```

```
npx tailwindcss init -p
```

Uočite da smo dobili dvije nove datoteke u korijenskom direktoriju projekta: `tailwind.config.js` i `postcss.config.js`.

- **4. korak:** konfigurirajte `tailwind.config.js` datoteku prema uputama na poveznici:

Zamjenjujemo sadržaj datoteke s:

```
// tailwind.config.js

/** @type {import('tailwindcss').Config} */
export default {
  content: ['./index.html', './src/**/*.{vue,js,ts,jsx,tsx}'],
  theme: {
    extend: {}
  },
  plugins: []
};
```

- **5. korak:** unutar `./src/style.css` postavljamo sljedeće 3 direktive (brišemo sve ostalo):

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Uspješno te instalirali Vue aplikaciju s Tailwind CSS-om. Sada ćemo malo srediti strukturu projekta, a zatim pokrenuti projekt i testirati funkcionira li sve.

2.2 Struktura projekta i testiranje

Nakon što ste sve instalirali, trebali biste imati sljedeću strukturu `vue` projekta:

```
vue
├── node_modules
└── public
└── src
    ├── assets
    ├── components
    ├── App.vue
    ├── main.js
    └── style.css
└── index.html
└── package-lock.json
└── package.json
└── postcss.config.js
└── README.md
└── tailwind.config.js
└── vite.config.js
```

Kako biste se uvjerili da je Tailwind instaliran, otvorite `./src/App.vue` datoteku i dodajte sljedeći `template` kod:

```
<template>
  <h1 class="text-3xl font-bold underline">Hello world!</h1>
</template>
```

Pokrenite aplikaciju koristeći naredbu:

```
npm run dev
```

Trebali biste vidjeti poruku `Hello world!` s velikim podebljanim fontom i podcrtanim tekstrom budući da je tako definirano Tailwind CSS klasama `text-3xl`, `font-bold` i `underline`.

Ako vidite, znači da je sve uspješno instalirano i konfiguirano. Možete još za sigurnost pokušati izmijeniti neku klasu, npr. `text-3xl` u `text-5xl` i vidjeti hoće li se promjena odraziti na stranici.

Općenito, klase za Tailwind CSS ne želite mijenjati direktno u CSS kodu, već koristiti i **kombinirati predefinirane klase** koje dolaze s Tailwindom. Nema ih puno smisla niti učiti napamet iako su u pravilu intuitivno imenovane. **Preporuka je naučiti služiti se [TailwindCSS dokumentacijom](#).**

2.3 Komponenta za prikaz proizvoda

Izradit ćemo komponentu `Proizvod.vue` koja će prikazivati **određeni proizvod s njegovim detaljima**. U `./src/components` direktoriju izradite novu datoteku `Proizvod.vue`.

Kako frontend dizajn korisničkog sučelja nije predmet ovog predmeta, upotrijebit ćemo gotovi template stiliziran pomoću Tailwinda te raditi na funkcionalnostima komponente. Ako hoćete, možete uređivati stilove prema vlastitim željama i/ili izraditi vlastiti dizajn.

Template možete kopirati iz `/snippets/ProductTemplate.html` koji se nalazi u ovom repozitoriju.

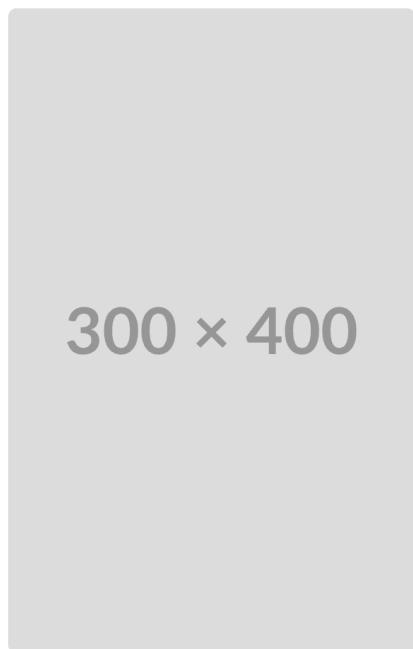
Nakon što ste kopirali template i izradili komponentu, morate omotati html kod u `template` tag.

```
<!-- Proizvod.vue -->
<template>
  <!-- HTML kod ovdje -->
</template>
```

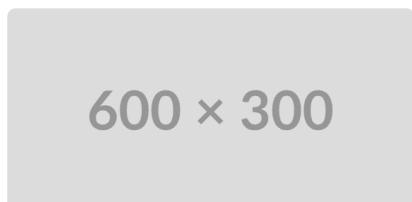
Zatim ćemo u `App.vue` uključiti ovu komponentu i iscrtati ju na stranici.

```
<template>
  <div>
    <!-- Uključujemo komponentu -->
    <ProductView />
  </div>
</template>
<script setup>
import ProductView from './components/ProductView.vue';
</script>
```

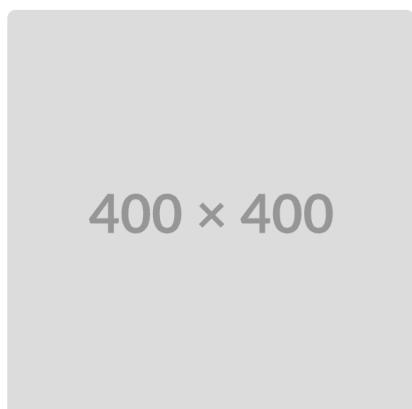
Sada možete pokrenuti aplikaciju i vidjeti kako izgleda komponenta `Proizvod.vue`.



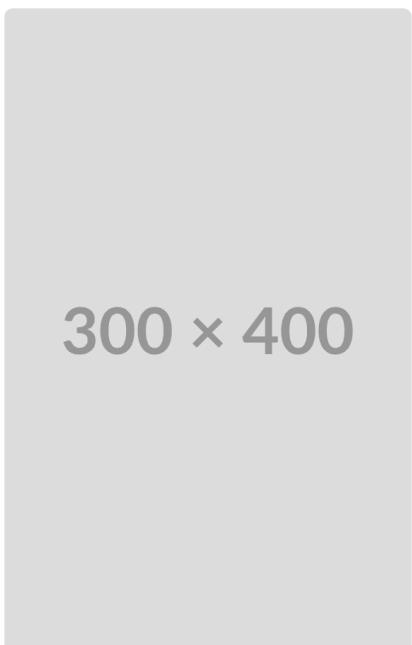
300 × 400



600 × 300



400 × 400



300 × 400

Naslov proizvoda

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Karakteristike

- Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium
- doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto
- beatae vitae dicta sunt explicabo
- Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit

Detalji

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore

100€

Boje



Veličina



Dodaj u košaricu

3. Axios HTTP klijent

Za komunikaciju s Express poslužiteljem imamo na raspolaganju više opcija. Moguće je koristiti i `fetch` API koji smo upoznali na Skriptnim jezicima i Programskom inženjerstvu, međutim kroz neke vanjske biblioteke možemo definirati konciznu i čitljivu sintaksu za slanje HTTP zahtjeva te rukovanje odgovorima.

Jedna od takvih biblioteka je i [Axios](#) koji ćemo koristiti na ovom kolegiju. **Axios** je HTTP klijent za Node i web preglednika koji se bazira na sintaksi `Promise` objekata.

Axios pojednostavljuje slanje HTTP zahtjeva na vanjske API servise, omogućava detaljnu konfiguraciju zahtjeva, upravljanje odgovorima i greškama itd.

Instalirajte Axios paket koristeći npm:

```
npm install axios
```

Jednom kad ste ga instalirali, možete ga koristiti u svim `vue` i `js` datotekama.

3.1 Slanje GET zahtjeva

Krenimo sa slanjem jednostavnog GET zahtjeva za dohvaćanje svih proizvoda iz naše trgovine. Možete otvoriti `ProductView.vue` datoteku i dodati sljedeći kod unutar `script` taga:

```
// ProductView.vue

<script setup>
import axios from 'axios';

axios
  .get('http://localhost:3000/proizvodi')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('Greška u dohvatu podataka:', error);
  });
</script>
```

Vidimo da Axios koristi `Promise` objekte za rukovanje odgovorima i greškama. Prisjetimo se ukratko kako `Promise` objekti funkcioniraju. `Promise` predstavlja eventualni rezultat asinkronog procesa i može biti u jednom od tri stanja: `pending`, `fulfilled` ili `rejected`.

- `pending`: inicijalno stanje, očekuje se rezultat
- `fulfilled`: proces je završen uspješno
- `rejected`: proces je završen s greškom

Kada se proces završi, `Promise` objekt prelazi u jedno od dva završna stanja: `fulfilled` ili `rejected`. Ukoliko je proces završen uspješno, `then` metoda se poziva s rezultatom, dok se u slučaju greške poziva `catch` metoda.

Sintaksa:

```
const myPromise = new Promise((resolve, reject) => {
  // asinkroni proces
  if (uspjeh) {
    resolve('Uspješno');
  } else {
    reject('Greška');
  }
});
```

Kada pozovemo `Promise`, rukujemo njime pomoću `then` i `catch` metoda:

```
myPromise
  .then(result => {
    // ako je rezultat uspješan (resolve)
    console.log(result); // ispisuje se "Uspješno"
  })
  .catch(error => {
    // ako je došlo do greške (reject)
    console.error(error); // ispisuje se "Greška"
});
```

Rekli smo da možemo koristiti i `async/await` sintaksu za rukovanje `Promise` objektima. **Asinkrone funkcije** su funkcije koje vraćaju `Promise` objekt. Ukoliko se u funkciji koristi `await` ključna riječ, **funkcija se zaustavlja** dok se ne razriješi `Promise` objekt.

```
let result = await myPromise;

console.log(result); // ispisuje se "Uspješno"
```

Međutim sad ako želimo rukovati greškama, moramo koristiti `try/catch` blok.

```
try {
  let result = await myPromise;
  console.log(result); // ispisuje se "Uspješno"
} catch (error) {
  console.error(error); // ispisuje se "Greška"
}
```

Vratimo se na Axios. U gornjem primjeru, Axios šalje GET zahtjev na adresu

`http://localhost:3000/proizvodi` i **očekuje odgovor**.

- Ukoliko je odgovor uspješan, ispisuje se odgovor u konzoli.
- Ukoliko dođe do greške, ispisuje se poruka o grešci.

Pokrenite Express aplikaciju i pokušajte poslati ovaj GET zahtjev na način da ćete samo osvježiti stranicu. Nakon toga otvorite web konzolu i provjerite ispis podataka.

3.1.1 CORS policy

Na našu žalost, ispisa nema već se u konzoli pojavljuje greška.

```
✖ Access to XMLHttpRequest at 'http://localhost:3000/proizvodi' from origin 'http://localhost:5173' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. :5173/#:1
✖ ▶ Error fetching data: ▶ AxiosError {message: 'Network Error', name: 'AxiosError', code: 'ERR_NETWORK', config: {...}, request: XMLHttpRequest, ...} ProductView.vue:283
✖ ▶ GET http://localhost:3000/proizvodi net::ERR_FAILED 200 (OK) ProductView.vue:278 ⓘ
```

Prvo što možemo uočiti jest da se zahtjev definitivno poslao, budući da vidimo nekakav ispis u konzoli. Također možemo vidjeti URL na kojem je poslan zahtjev: `http://localhost:3000/proizvodi`.

Greška kaže: `Access to XMLHttpRequest at 'http://localhost:3000/proizvodi' from origin 'http://localhost:5173' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.`

U prijevodu, slanje zahtjeva i pristup resursu su blokirani jer poslužitelj ne dopušta pristup resursu iz vanjskog izvora (naša vue aplikacija). Ovo je sigurnosna mjera koja se zove **CORS policy** (Cross-Origin Resource Sharing).

Statusni kod `200` znači da je zahtjev uspješno poslan, ali vidimo da je odgovor blokiran i grešku pod statusnim kodom: `ERR_NETWORK`.

Ono što moramo napraviti je omogućiti pristup resursima iz vanjskog izvora na Express poslužitelju.

U Express poslužitelju, instalirajte sljedeću biblioteku:

```
npm install cors
```

Zatim uključite `cors` u `index.js` datoteci:

```
// index.js

import express from 'express';
import cors from 'cors';

const app = express();
app.use(cors());
```

Linija `app.use(cors())` omogućuje pristup svim resursima iz vanjskih izvora (nikako pustiti ovo kad se radi o producijskom okruženju). Sada bi trebali moći poslati zahtjev i dobiti odgovor.

Napomena: CORS policy je sigurnosna mjera koja se koristi kako bi se spriječilo izvršavanje zlonamjernog koda na strani klijenta. U producijskom okruženju, uvijek je preporučljivo postaviti odgovarajuće postavke za CORS policy.

Sada možemo vidjeti da je odgovor uspješno primljen i da se podaci ispisuju u konzoli.

```
▼ (5) [{}]
  ▷ 0: {id: 1, naziv: 'Obična crna majica', cijena: 100, velicine: Array(4)}
  ▷ 1: {id: 2, naziv: 'Levi's 501 traperice', cijena: 110, velicine: Array(3)}
  ▷ 2: {id: 3, naziv: 'Zimska kapa', cijena: 40, velicine: 'onesize'}
  ▷ 3: {id: 4, naziv: 'Čarape Adidas', cijena: 20, velicine: Array(3)}
  ▷ 4: {id: 5, naziv: 'Tenisice Nike', cijena: 200, velicine: Array(8)}
  length: 5
  [[Prototype]]: Array(0)
```

Možete otvoriti i `Network` tab u Developer alatima vašeg web preglednika i vidjeti detalje svih zahtjeva i odgovora. Ovdje ćete pronaći puno korisnih informacija o svakom zahtjevu, uključujući i zaglavlja, tijelo zahtjeva i odgovora, statusni kod itd.

Međutim vidjet ćete i puno zahtjeva koji se pošalju/primaju prilikom učitavanja stranice. Ovo je zato što se u pozadini izvršava razvojni Vue.js poslužitelj koji nam servira mnoge datoteke i resurse kako bi se aplikacija ispravno prikazala. Koga više zanima ova tema, možete pročitati [kako funkciraju SSA](#) (Single-page Application) aplikacije, poput Vue.js-a.

Osim toga, možete pronaći i zahtjev koji smo poslali na naš Express poslužitelj i vidjeti neke detalje o njemu:

The screenshot shows the Network tab in the Chrome DevTools. A request for 'proizvodi' is selected. The Headers section shows the following details:

| Request URL | http://localhost:3000/proizvodi |
|-----------------------------|-------------------------------------|
| Request Method | GET |
| Status Code | 200 OK |
| Remote Address | [::1]:3000 |
| Referrer Policy | strict-origin-when-cross-origin |
| Access-Control-Allow-Origin | * |
| Content-Length | 407 |
| Content-Type | application/json; charset=utf-8 |
| Date | Sun, 10 Nov 2024 20:45:17 GMT |
| Etag | W/"197-qAD6ElI3UvVCy0wJBXADmNfeme0" |
| X-Powered-By | Express |

Primjerice, uočite generalne informacije o zahtjevu:

- **Request URL:** `http://localhost:3000/proizvodi`
- **Request Method:** `GET`
- **Status Code:** `200 OK`

Ako otvorite `Preview` ili `Response` možete vidjeti tijelo odgovora koje je poslano s poslužitelja, odnosno podatke o proizvodima.

Testirajte slanje GET zahtjeva za dohvaćanje pojedinog proizvoda.

3.1.2 Prikazivanje proizvoda na frontendu

Ideja je da upotrijebimo podatke koje smo dobili iz Express poslužitelja i prikažemo ih korisniku. Prvo moramo naravno podatke spremiti u neku varijablu, a zatim ih prikazati na stranici.

U `ProductView.vue` datoteci, definirajmo varijablu `proizvodi` koja će sadržavati podatke o proizvodima.

```
// ProductView.vue

<script setup>
import axios from 'axios';

let proizvodi = [];

axios
  .get('http://localhost:3000/proizvodi/1')
  .then(response => {
    proizvodi = response.data;
  })
  .catch(error => {
    console.error('Greška u dohvatu podataka:', error);
  });
</script>
```

Međutim ovaj pristup nije dobar budući da Vue komponente nemaju ugrađenu reaktivnost za promjene varijabli poput `proizvodi`. Potrebno je definirati reaktivnu varijablu koristeći `ref` ili `reactive` funkcije.

Reactivity API u Vue 3 omogućuje nam da pratimo promjene varijabli na razne načine. Do sada je vjerojatno većina vas učila Vue2 koji koristi `data` objekt za definiranje reaktivnih varijabli. U Vue 3, `data` objekt je zamijenjen s `setup` funkcijom koja vraća objekt s reaktivnim varijablama. Više informacija o tome na sljedećoj [poveznici](#).

`ref` funkciju koristimo za definiranje **reaktivne reference** na primitivne tipove podataka** (string, number, boolean) ili **objekte**.

- vrijednost vraćamo pomoću `value` svojstva ako ga čitamo unutar skriptnog dijela komponente, u template dijelu možemo izostaviti `value` svojstvo.
- ako se promijeni vrijednost reaktivne reference, Vue će automatski osježiti komponentu.

`reactive` funkciju koristimo za definiranje **reaktivnih objekata** koji sadrže više svojstava.

- iako možemo koristiti i `ref` za objekte, `reactive` je bolji izbor jer omogućuje reaktivnost za dublja svojstva objekta (npr. kada definiramo config aplikacije, korisničke postavke itd.)
- promjene unutar objekta će se automatski pratiti i osježiti komponentu, a za pristupanje vrijednostima ne koristimo `value` svojstvo.

Kako ne bi previše zakomplificirali stvari, možemo koristiti `ref` funkciju za definiranje reaktivne varijable `proizvod`. Oprez, ako koristite `reactive` funkciju morate paziti da ne pregazite cijeli objekt dohvaćenim podacima, već samo pojedinačne attribute (npr. greška bi bila: `proizvod = response.data`).

```
<script setup>
import { ref } from 'vue';
```

```

import axios from 'axios';

let proizvod = ref(null);

axios
  .get('http://localhost:3000/proizvodi/1')
  .then(response => {
    console.log(response.data);
    proizvod.value = response.data;
  })
  .catch(error => {
    console.error('Greška u dohvatu podataka:', error);
  });

console.log('proizvod', proizvod); // prazan Reference objekt budući da se asinkroni zahtjev
još nije izvršio
</script>

```

Problem je što se `console.log` izvršava prije nego što se asinkroni zahtjev izvrši. Kako bismo riješili ovaj problem, možemo koristiti `await` ključnu riječ unutar `setup` funkcije, međutim kako želimo da se asinkroni zahtjev izvrši samo jednom, kada korisnik učita stranicu, koristit ćemo `onMounted` hook.

`onMounted` je Lifecycle Hook u Vue.js kojim se može definirati **callback funkcija koja će se izvršiti kada se komponenta učita** (montira).

Kako se radi o asinkronoj callback funkciji, moramo koristiti `async` ključnu riječ ispred definicije funkcije.

```

<script setup>
import { ref, onMounted } from 'vue';
import axios from 'axios';

let proizvod = ref(null);

// asinkroni callback (hook)
onMounted(async () => {
  try {
    const response = await axios.get('http://localhost:3000/proizvodi/1');
    proizvod.value = response.data; // postavljanje podataka u reaktivnu varijablu
  } catch (error) {
    console.error('Greška u dohvatu podataka:', error);
  }
});
</script>

```

Jednom kad imamo podatke o proizvodu, možemo ih prikazati na stranici. U `template` tagu možemo prikazati podatke pomoću `{} {}` interpolacije odnosno `v-model` direktive ako se radi o atributima HTML elemenata.

```

<template>
  ...
  <a href="#" aria-current="page" class="font-medium text-gray-500 hover:text-gray-600">{{ proizvod.naziv }}</a>
  ...
  <h1 class="text-2xl font-bold tracking-tight text-gray-900 sm:text-3xl">{{ proizvod.naziv }}</h1>
</template>

```

Vidimo da stvari rade, ali dobivamo grešku u konzoli `cannot read properties of null`. Ovo je zato što se komponenta renderira prije nego što se podaci o proizvodu dohvate s poslužitelja, a početna vrijednost reaktivne varijable `proizvod` je `null`.

Kako bismo to riješili, možemo koristiti `v-if` direktivu koja će prikazati element samo ako je on istinit. U ovom slučaju, prikazat ćemo elemente samo ako postoji dohvaćeni podaci o proizvodu.

```

<template>
  <div v-if="proizvod">
    <a href="#" aria-current="page" class="font-medium text-gray-500 hover:text-gray-600">{{ proizvod.naziv }}</a>
    <h1 class="text-2xl font-bold tracking-tight text-gray-900 sm:text-3xl">{{ proizvod.naziv }}</h1>
  </div>
</template>

```

- ili možemo definirati početne vrijednosti za reaktivnu varijablu `proizvod` koristeći `ref` funkciju.

```

<script setup>
let proizvod = ref({
  id: 0,
  naziv: '',
  cijena: 0,
  velicine: []
});

```

Nadopunit ćemo i preostale podatke o proizvodu: `cijena` i `velicine`.

```

<template>
  ...
  <div class="mt-4 lg:row-span-3 lg:mt-0">
    <h2 class="sr-only">Product information</h2>
    <p class="text-3xl tracking-tight text-gray-900">{{ proizvod.cijena }}€</p>
  ...
</template>

```

Za veličine nemamo jednostavne inline tekst elemente, već ćemo koristiti `v-for` direktivu za ponavljanje HTML elemenata koji prikazuju pravokutnik s dostupnim veličinama.

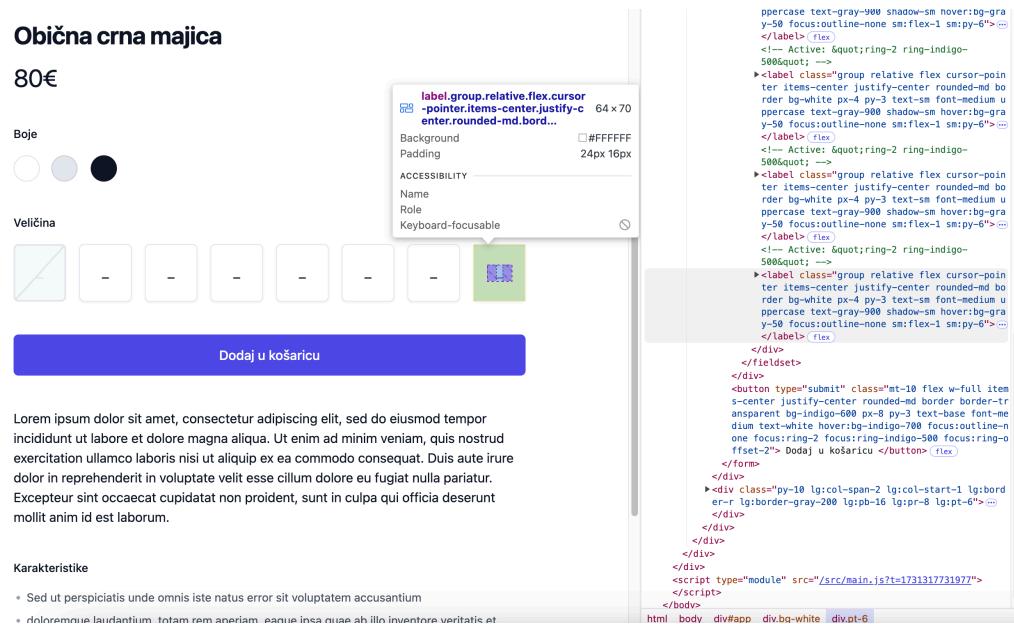
Koristeći Find (**CTRL/CMD + f**) pronađite HTML sekciju s veličinama:

```
<div class="mt-10">
  <div class="flex items-center justify-between">
    <h3 class="text-sm font-medium text-gray-900">Veličina</h3>
  </div>

  <fieldset aria-label="Choose a size" class="mt-4">
    <div class="grid grid-cols-4 gap-4 sm:grid-cols-8 lg:grid-cols-4">
      <!-- Active: "ring-2 ring-indigo-500" -->
      <label
        class="group relative flex cursor-not-allowed items-center justify-center rounded-md border border-gray-50 px-4 py-3">
        <input type="radio" name="size-choice" value="" disabled class="sr-only" />
        <span></span>
        <span aria-hidden="true" class="pointer-events-none absolute -inset-px rounded-md border-2 border-gray-200">
          <svg
            class="absolute inset-0 h-full w-full stroke-2 border-gray-200"
            viewBox="0 0 100 100"
            preserveAspectRatio="none"
            stroke="currentColor"
          >
            <line x1="100" y1="100" x2="100" y2="0" vector-effect="non-scaling-stroke" />
          </svg>
        </span>
      </label>
      <!-- Active: "ring-2 ring-indigo-500" -->
      <label
        class="group relative flex cursor-pointer items-center justify-center rounded-md border border-white px-4 py-3 text-sm">
        <input type="radio" name="size-choice" value="" class="sr-only" />
        <span></span>
        <!-- Active: "border", Not Active: "border-2"
        Checked: "border-indigo-500", Not Checked: "border-transparent" -->
        <span class="pointer-events-none absolute -inset-px rounded-md" aria-hidden="true"></span>
      </label>
      <!-- Active: "ring-2 ring-indigo-500" -->
      <label
        class="group relative flex cursor-pointer items-center justify-center rounded-md border border-white px-4 py-3 text-sm">
        <input type="radio" name="size-choice" value="S" class="sr-only" />
        <span></span>
        <!-- Active: "border", Not Active: "border-2"
        Checked: "border-indigo-500", Not Checked: "border-transparent" -->
        <span class="pointer-events-none absolute -inset-px rounded-md" aria-hidden="true"></span>
      </label>
      <!-- Active: "ring-2 ring-indigo-500" -->
```

Ako analizirate HTML strukturu, uočit ćete da je svaka veličina definirana unutar `label` oznake, a sama oznaka je unutar `span` elementa. Trenutno su za sve prikazane `_` oznake, a za prvu imamo i dodatni `svg` element koji prikazuje prekriženu veličinu (nije dostupna).

Ako vam nije potpuno jasno, vrlo je korisno kroz Developer alate preglednika analizirati HTML strukturu i pronaći odgovarajuće elemente.



Na ovaj način možete jednostavno uočiti strukturu koja definira neki podatak.

Budući da se radi o dinamičkim podacima, odnosno veličine se mogu mijenjati ovisno o proizvodu, a imamo i ponavljajući HTML kod koji definira jednu veličinu, moramo koristiti `v-for` direktivu za iscrtavanje ponavljajućih HTML elemenata.

Sintaksa v-for:

{{element}}

- ili ako koristimo element unutar HTML oznake:

```
<div v-for="element in list" :key="element" :value = "element"></div>
```

OPREZ: razlikuje se od sintakse JavaScript `for` petlje koja koristi `of` ključnu riječ. Ovo više nalikuje na `for` petlju u Pythonu.

U našem slučaju, izbrisat ćemo sve osim jedne ponavljajuće veličine (preskačemo i prvu jer je ta prekrižena) i pišemo `v-for` za iscrtavanje za svaku veličinu u objektu koji definira proizvod. Dodatno, obzirom da `label` sadrži dosta CSS klase, možda je bolje da sve omotamo jednostavnim `div` elementom i tu definiramo `v-for` direktivu.

```
<div v-for="velicina in proizvod.velicine" :key="velicina">
  <!-- v-for direktiva za iscrtavanje velicina -->
  <label
    class="group relative flex cursor-pointer items-center justify-center rounded-md border
    bg-white px-4 py-3 text-sm font-medium uppercase text-gray-900 shadow-sm hover:bg-gray-50
    focus:outline-none sm:flex-1 sm:py-6"
  >
    <input type="radio" name="size-choice" value="_" class="sr-only" />
    <span>{{ velicina }}</span>
    <!--
      Active: "border", Not Active: "border-2"
      Checked: "border-indigo-500", Not Checked: "border-transparent"
    -->
    <span class="pointer-events-none absolute -inset-px rounded-md" aria-hidden="true">
  </span>
  </label>
  <!-- Active: "ring-2 ring-indigo-500" -->
</div>
```

3.1.3 Dodavanje podataka na backendu

Sada kada smo prikazali sve podatke, dodat ćemo još nekoliko podataka za naš proizvod na Express poslužitelju kako bi upotpunili prikaz u našoj aplikaciji.

Dodat ćemo `slike` i `opis` proizvoda.

Proširit ćemo definiciju konstruktora koji definira proizvod u `data.js` datoteci.

```
// server/data.js

class Proizvod {
  constructor(id, naziv, cijena, velicine) {
    this.id = id;
    this.naziv = naziv;
    this.cijena = cijena;
    this.velicine = velicine;
  }
}
```

Za slike ćemo pronaći nekoliko javnih URL-ova slika proizvoda na internetu, a opis možemo izmisliti.

```
// server/data.js

class Proizvod {
  constructor(id, naziv, cijena, velicine, opis, slike) {
    this.id = id;
    this.naziv = naziv;
    this.cijena = cijena;
    this.velicine = velicine;
    this.opis = opis;
    this.slike = slike;
  }
}

const proizvodi = [
  new Proizvod(1, 'Obična crna majica', 80, ['XS', 'S', 'M', 'L']), // dodajte opis i polje
  poveznica na slike
  new Proizvod(2, "Levi's 501 traperice", 110, ['S', 'M', 'L']),
  new Proizvod(3, 'Zimska kapa', 40, 'onesize'),
  new Proizvod(4, 'Čarape Adidas', 20, ['34-36', '37-39', '40-42']),
  new Proizvod(5, 'Tenisice Nike', 200, ['38', '39', '40', '41', '42', '43', '44', '45'])
];

export { proizvodi, Proizvod };
```

U `vue` aplikaciji, odnosno komponenti `ProductView.vue` možemo dodati prikaz slika i opisa proizvoda. Pronađite odgovarajuće HTML elemente i dodajte ih u template na isti način. Dodat ćemo slike ručno, bez korištenja `v-for` direktive budući da su različitih dimenzija.

```
<template>
```

```

...
<!-- Image gallery -->
<div class="mx-auto mt-6 max-w-2xl sm:px-6 lg:grid lg:max-w-7xl lg:grid-cols-3 lg:gap-x-8
lg:px-8">
  <div class="aspect-h-4 aspect-w-3 hidden overflow-hidden rounded-lg lg:block">
    
  </div>
  <div class="hidden lg:grid lg:grid-cols-1 lg:gap-y-8">
    <div class="aspect-h-2 aspect-w-3 overflow-hidden rounded-lg">
      
    </div>
    <div class="aspect-h-2 aspect-w-3 overflow-hidden rounded-lg">
      
    </div>
  </div>
  <div class="aspect-h-2 aspect-w-3 overflow-hidden rounded-lg">
    
  </div>
</div>
...
<div class="space-y-6">
  <p class="text-base text-gray-900">{{ proizvod.opis }}</p>
</div>
...
</template>

```

Uspješno smo pročitali sve podatke o proizvodu i prikazali ih na klijentskoj strani! Čestitke!



Obična crna majica

Otkrijte ultimativni spoj udobnosti i stila s našom premium crnom majicom kratkih rukava. Ova majica nije samo odjevni predmet; ona je vaš tih saveznik u svakodnevnim avanturama, idealna za sve prilike i sve stilove. Izrađena je od visokokvalitetnog, prozračnog pamuka koji miluje vašu kožu poput nježnog povjetara. Njena izrada pažljivo prati svaki šav, jamčeći trajnost i oblik kroz bezbrojna pranja, dok njezin besprjekorni kroj ističe vašu figuru, bez obzira na tjelesnu građu.

Karakteristike

- Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium
- doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto
- beatae vitae dicta sunt explicabo
- Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit

80€

Boje



Veličina



3.2 Slanje POST zahtjeva

Za kraj ćemo još samo pokazati kako slati POST zahtjeve na Express poslužitelju. U ovom slučaju, ćemo zahtjev za dodavanje novog proizvoda u našu trgovinu.

Jedina razlika je što POST zahtjev ima dodatno tijelo koje sadrži podatke koje želimo poslati na poslužitelj. U Axiosu, tijelo se definira kao drugi argument funkcije `axios.post`.

Sintaksa je sljedeća:

```
axios.post(url, data, config);
```

gdje su:

- `url`: URL adresa na koju šaljemo zahtjev
- `data`: podaci (tijelo zahtjeva) koje šaljemo na poslužitelj
- `config`: dodatne postavke zahtjeva (npr. zaglavla, autentifikacija itd.)

Narudžbu možemo poslati na sljedeći način:

```
<script setup>
axios.post('http://localhost:3000/narudzbe', podaci);
</script>
```

- gdje su `podaci` objekt koji sadrži podatke o narudžbi

Na poslužitelju smo definirali da se naša narudžba sastoji od `id` i `naruceni_proizvodi` gdje su `naruceni_proizvodi` polje objekata koji sadrže `id` proizvoda i `narucena_kolicina` proizvoda. Međutim, logično je da se `id` narudžbe generira automatski na poslužitelju, što smo i implementirali ranije.

Dakle, naše tijelo zahtjeva može izgledati ovako:

```
let podaci = ref({
  naruceni_proizvodi: [
    { id: 1, narucena_kolicina: 2 },
    { id: 3, narucena_kolicina: 1 }
  ]
});
```

Sada ćemo poslati POST zahtjev s hardkodiranim podacima o narudžbi, čisto da vidite kako se to radi. Naravno, podaci se inače šalju nakon što korisnik doda sve proizvode u košaricu, uz ostale podatke o narudžbi i odradi plaćanje.

U Vue3 koristimo [Composition API](#) pa definiramo podatke i metode drugačije unutar skriptnog dijela. Već smo vidjeli kako definiramo podatke, a metode možemo definirati kao varijable koje pohranjuju složene funkcione izraze.

Dummy podatke smo definirali iznad, a metodu za slanje narudžbe možemo definirati na sljedeći način:

```
// ProductView.vue
<script setup>
const posaljiNarudzbu = async () => {
  try {
    let response = await axios.post('http://localhost:3000/narudzbe', podaci.value); // 
    axios.post(url, data)
    console.log(response);
  } catch (error) {
    console.error('Greška u dohvatu podataka: ', error);
  }
};
</script>
```

Metodu možemo pozvati na kraju `onMounted` hooka ili pritiskom na gumb "Dodaj u košaricu" koristeći direktivu `@click`.

```
<button
  type="submit"
  @click="posaljiNarudzbu"
  class="mt-10 flex w-full items-center justify-center rounded-md border border-transparent
  bg-indigo-600 px-8 py-3 text-base font-medium text-white hover:bg-indigo-700 focus:outline-
  none focus:ring-2 focus:ring-indigo-500 focus:ring-offset-2"
>
  Dodaj u košaricu
</button>
```

To je to! Ako ste dodali ispis na poslužitelju, možete vidjeti da se narudžba uspješno poslala, proizvodi su pronađeni te je izračunata ukupna cijena narudžbe.

Ako pogledamo karticu `Network` u Developer alatima, možemo pronaći zahtjev na `/narudzbe` i vidjeti da smo dobili statusni kod `201 Created` i nazad dobili podatke o narudžbi, upravo kako smo i definirali na poslužitelju.

Samostalni zadatak za Vježbu 3

Vaš zadatak je nadograditi aplikaciju `webshop` dodavanjem novih funkcionalnosti.

Kako već imate iskustva u razvoju klijentske strane aplikacije, morate implementirati sljedeće funkcionalnosti:

1. Instalirajte i podesite `vue-router` za navigaciju između stranica. Aplikacija mora imati dvije stranice: početnu stranicu (`/proizvodi`) koja će prikazivati sve proizvode kao kartice s osnovnim detaljima (vi ih odaberite i dizajnirajte kartice) i stranicu za prikaz pojedinog proizvoda koju imate već definiranu (`/proizvodi/:id`).
2. Implementirajte funkcionalnost gdje korisnik može stisnuti na određenu karticu proizvoda i biti preusmjeren na stranicu s detaljima proizvoda.
3. Jednom kad korisnik pristupi stranici s detaljima (`/proizvodi/:id`), mora se poslati GET zahtjev na Express poslužitelj za dohvati podataka o jednom proizvodu s odgovarajućim `id` parametrom. Naravno, na početnoj stranici `/proizvodi` morate u `onMounted` hook u kojem ćete poslati GET zahtjev za dohvat svih proizvoda.

Nadogradite Express poslužitelj na način da ćete za svaki proizvod dodati još nekoliko atributa:

- `dostupne_boje` (npr. crna, bijela, plava, crvena)
- `karakteristike` (npr. materijal, težina, informacije o održavanju)

Definirajte nekoliko proizvoda i nadopunite ih s novim atributima u `data.js` datoteci.

Nadogradite klijentsku stranu na način da ćete prikazati sve nove podatke o proizvodu na stranici s detaljima.

Kada korisnik odabere proizvod i pritisne gumb "Dodaj u košaricu", nemojte poslati narudžbu kao što je to sad slučaj, već spremite podatke u vanjsku `.js` datoteku, npr. `narudzbe.js` ili `kosarica.js` ili u `localStorage` / `sessionStorage`. Na ovaj način pohranjujete podatke o narudžbi lokalno na strani klijenta (naučit ćemo kako to raditi bolje).

Nakon što korisnik doda proizvod, preusmjerite ga na početnu stranicu. Na početnoj stranici prikažite broj proizvoda u košarici, a ispod broja proizvoda dodajte još jedan gumb "Naruči proizvode" koji će poslati POST zahtjev `/narudzbe` na Express poslužitelj s podacima o narudžbi u tijelu zahtjeva.

Rješenje učitajte na svoj GitHub repozitorij, a poveznicu na Merlin.

Web aplikacije (WA)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



Fakultet informatike u Puli

(4) Upravljanje podacima na poslužiteljskoj strani

#4

WA

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

- [Web aplikacije \(WA\)](#)
- [\(4\) Upravljanje podacima na poslužiteljskoj strani](#)
 - [Sadržaj](#)
- [1. Gdje pohranjujemo podatke u web aplikacijama?](#)
- [2. Podaci na poslužiteljskoj strani](#)
 - [2.1 Čitanje datoteka kroz `fs` modul](#)
 - [2.1.1 Asinkroni pristup čitanju datoteke](#)
 - [2.1.2 Apsolutna i Relativna putanja do datoteke](#)
 - [2.1.3 `Callback` vs `Promise` pristup](#)
 - [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 Query parametre](#)
 - [3.1 Filtriranje podataka](#)
 - [3.2 Sortiranje podataka](#)
- [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 poslužiteljske** pohrane podataka. Web aplikacije u producijskom 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.

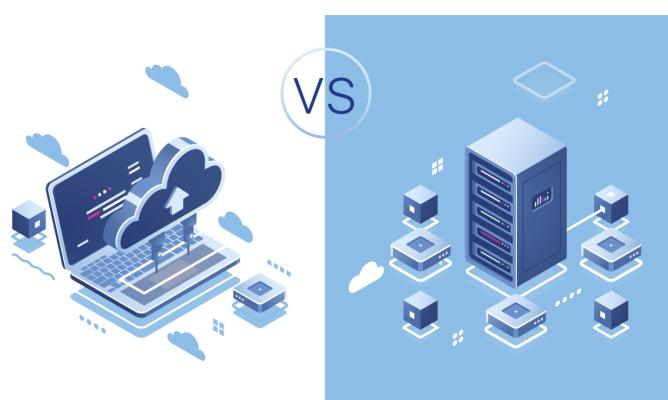
Podaci koji se pohranjuju na **klijentskoj strani** obično se koriste (samim time i pohranjuju) u sljedeće svrhe:

- personalizacija korisničkog iskustva (npr. boja pozadine, postavke jezika, odabrana paleta boja/tema, itd.)
- čuvanje korisničkih postavki (npr. preferirani način prikaza podataka, odabране 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 sekvensijalnoj 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:\` (prepostavka).

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, absolutna 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 dvostrukе 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 dvostrukе 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 sekvensijalno 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);
  });
  return story_text;
}
```

```

    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 funkciju
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 'dovidjenja'

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 proslijedujemo callback funkciju
    // iz argumenta
}

app.get('/story', (req, res) => {
    read_story((err, data) => {
        // kao argument proslijedujemo 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 rješili `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 odabratи koji pristup je vama dražи, međutim `Promise` pattern i `async/await` sintaksa su moderniji pristupи 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 tekstrom: `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 funkcioniра, 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, pohranu 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` - pohranu 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_per));
        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 zapetljani 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,
deserializiraj 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 izbjegći 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.

The screenshot shows the Postman interface with a GET request to `http://localhost:3000/students?fakultet=FIPU`. In the 'Query' tab, there is a checked checkbox labeled 'fakultet' with the value 'FIPU'. In the 'Response' tab, the status is 200 OK, size is 189 Bytes, and time is 15 ms. The response body is a JSON array of three student objects, each with 'fakultet' set to 'FIPU':

```
1 [
2   {
3     "ime": "Pero",
4     "prezime": "Perić",
5     "godine": 20,
6     "fakultet": "FIPU"
7   },
8   {
9     "ime": "Ana",
10    "prezime": "Anić",
11    "godine": 18,
12    "fakultet": "FIPU"
13  },
14  {
15    "ime": "Ivo",
16    "prezime": "Ivić",
17    "godine": 22,
18    "fakultet": "FIPU"
19  }
20 ]
```

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

The screenshot shows a Postman interface. The URL is set to `http://localhost:3000/students`. In the 'Query' tab, there are two parameters: `fakultet` with value `FIPU` and `godine` with value `20`. The 'Send' button is at the top right. To the right, the 'Response' tab displays the following JSON data:

```

1  [
2    {
3      "ime": "Pero",
4      "prezime": "Perić",
5      "godine": 20,
6      "fakultet": "FIPU"
7    },
8    {
9      "ime": "Ana",
10     "prezime": "Anić",
11     "godine": 18,
12     "fakultet": "FIPU"
13   },
14   {
15     "ime": "Ivo",
16     "prezime": "Ivić",
17     "godine": 22,
18     "fakultet": "FIPU"
19   },
20   {
21     "ime": "Mara",
22     "prezime": "Marić",
23     "godine": 21,
24     "fakultet": "FET"
25   },
26   {
27     "ime": "Jure",
28     "prezime": "Jurić",
29     "godine": 19,
30     "fakultet": "FET"
31   },
32   {
33     "ime": "Iva",
34     "prezime": "Ivić",
35     "godine": 23,
36     "fakultet": "FET"
37   }
38 ]

```

Moguće je definirati i više `query` parametara, npr. `godine`, `prezime`, `ime` i slično. Ukoliko želimo filtrirati studente po više kriterijima, 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; // dohvati smo query parametar 'fakultet'
  let godine_query = req.query.godine; // dohvati smo 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 **opcionali**. 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 dohvati **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; // dohvati query parametar 'fakultet'
  let godine_query = req.query.godine; // dohvati 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 prepostavku 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.');
  }
});
```

The screenshot shows the Postman interface with a GET request to `http://localhost:3000/students/Peru/Perić`. In the 'Query' tab, there are three parameters: `fakultet` (value: FIPU), `godine` (value: 20), and `parameter` (value: value). The 'Response' tab shows the JSON output of the student object:

```
1 {
2   "ime": "Peru",
3   "prezime": "Perić",
4   "godine": 20,
5   "fakultet": "FIPU"
6 }
```

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, dohvativamo `query` parametar i sortirajmo studente koristeći metodu `Array.sort()`:

Radi jednostavnosti, izostavit ćemo logiku za filtriranje:

```
app.get('/students', async (req, res) => {
  let sortiraj_po_godinama = req.query.sortiraj_po_godinama; // dohvativamo 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**:

| Header | Value |
|--------------|------------------|
| Content-Type | application/json |
| Status | 200 OK |
| Size | 376 Bytes |
| Time | 9 ms |

Response

```

1 [
2   {
3     "ime": "Ana",
4     "prezime": "Anić",
5     "godine": 18,
6     "fakultet": "FIPU"
7   },
8   {
9     "ime": "Jure",
10    "prezime": "Juric",
11    "godine": 19,
12    "fakultet": "FET"
13 },
14 {
15   "ime": "Pero",
16   "prezime": "Perić",
17   "godine": 20,
18   "fakultet": "FIPU"
19 },
20 {
21   "ime": "Mara",
22   "prezime": "Marić",
23   "godine": 21,
24   "fakultet": "FET"
25 },
26 {
27   "ime": "Ivo",
28   "prezime": "Ivić",
29   "godine": 22,
30   "fakultet": "FIPU"
31 },
32 {
33   "ime": "Iva",
34   "prezime": "Ivić",
35   "godine": 23,
36   "fakultet": "FET"
37 }
38 ]

```

Sortiranje po godinama **silazno**:

```

1 [
2   {
3     "ime": "Iva",
4     "prezime": "Ivić",
5     "godine": 23,
6     "fakultet": "FET"
7   },
8   {
9     "ime": "Ivo",
10    "prezime": "Ivić",
11    "godine": 22,
12    "fakultet": "FIPU"
13 },
14 {
15   "ime": "Mara",
16   "prezime": "Marić",
17   "godine": 21,
18   "fakultet": "FET"
19 },
20 {
21   "ime": "Pero",
22   "prezime": "Perić",
23   "godine": 20,
24   "fakultet": "FIPU"
25 },
26 {
27   "ime": "Dure",
28   "prezime": "Jurić",
29   "godine": 19,
30   "fakultet": "FET"
31 },
32 {
33   "ime": "Ana",
34   "prezime": "Anić",
35   "godine": 18,
36   "fakultet": "FIPU"
37 }
38 ]

```

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`):

```
1 {  
2   "ime": "Ivo",  
3   "prezime": "Ivić",  
4   "godine": 22,  
5   "fakultet": "FIPU"  
6 }
```

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

```
1 Student nije pronađen.
```

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

Implementirajte osnovne funkcionalnosti za dohvati, dodavanje i dohvati pojedinog zaposlenika. Zaposlenik treba imati sljedeće atributе:

- `id` - jedinstveni identifikator zaposlenika (generira se na poslužitelju)
- `ime` - ime zaposlenika
- `prezime` - prezime zaposlenika
- `godine_staza` - 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 osnovnu 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/deserializacije 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_staza_min` - filtriranje svih zaposlenika po minimalnom broju godina staža

- `godine_staza_max` - filtriranje svih zaposlenika po maksimalnom broju godina staža
-

Web aplikacije (WA)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



Fakultet informatike u Puli

#5

WA

(5) MongoDB baza podataka

Zadnji put vidjeli smo kako pohranjivati podatke na poslužitelju u datoteke te smo naveli zašto to može biti problematično za veće količine podataka i podatke kojima korisnici naše aplikacije direktno pristupaju. MongoDB je popularna NoSQL baza podataka koja se bazira na dokumentno-orientiranom modelu za pohranu podataka. Umjesto tablica i redaka kao u tradicionalnim relacijskim bazama podataka, MongoDB koristi zbirke (kolekcije) i dokumente. Na ovaj način, podaci su strukturirani u formatu sličnom JSON-u, što omogućuje fleksibilnu i intuitivnu organizaciju podataka. Naučit ćemo kako se izraditi MongoDB Atlas cluster u Cloudu, kako se povezati na njega putem našeg Express poslužitelja te kako izvršavati osnovne CRUD operacije nad podacima. Skripta je dosta opširna, za početak je važno da pohvatate osnovne koncepte i metode rada s MongoDB bazom podataka, a kasnije koristite skriptu kao svojevrsnu dokumentaciju.

Posljednje ažurirano: 6.12.2024.

Sadržaj

- [Web aplikacije \(WA\)](#)
- [\(5\) MongoDB baza podataka](#)
 - [Sadržaj](#)
- [1. MongoDB](#)
 - [1.1 MongoDB Atlas](#)
- [2. Povezivanje na cluster](#)
 - [2.1 Priprema Express poslužitelja](#)
 - [2.2 Connection string](#)
 - [2.3 db.js](#)
 - [2.4 dotenv modul](#)
- [3. CRUD operacije](#)
 - [3.1 GET operacija](#)

- [Mongo metoda: `collection\(\).find\(\)`](#)
- [3.1.1 GET `/pizze`](#)
- [3.1.2 GET `/pizze/:naziv`](#)
 - [Mongo metoda: `collection\(\).findOne\(\)`](#)
- [3.2 POST operacija](#)
 - [3.2.1 POST `/pizze`](#)
 - [Mongo metoda: `collection\(\).insertOne\(\)`](#)
 - [3.2.2 POST `/narudzbe`](#)
 - [Validacija zahtjeva na poslužitelju](#)
- [3.3 PUT i PATCH operacije](#)
 - [3.3.1 PATCH `/pizze/:naziv`](#)
 - [Mongo metoda: `collection\(\).updateOne\(\)`](#)
 - [MongoDB Update operatori](#)
 - [3.3.2 PATCH `/narudzbe/:id`](#)
 - [3.3.3 PUT `/pizze`](#)
 - [Mongo metoda: `collection\(\).insertMany\(\)`](#)
- [3.4 DELETE operacija](#)
 - [3.4.1 DELETE `/pizze/:naziv`](#)
 - [Mongo metoda: `collection\(\).deleteOne\(\)`](#)
 - [Mongo metoda: `collection\(\).deleteMany\(\)`](#)
- [4. Agregacija podataka](#)
 - [4.1 Filtriranje podataka](#)
 - [4.1.1 GET `/pizze?query`](#)
 - [4.2 Ažuriranje svih podataka gdje je uvjet zadovoljen](#)
 - [Mongo metoda: `collection\(\).updateMany\(\)`](#)
 - [4.3 Sortiranje podataka](#)
 - [4.3.1 GET `/pizze?sort`](#)
 - [4.4 Složena agregacija podataka metodom `aggregate\(\)`](#)
- [5. MongoDB - TL;DR](#)
 - [5.1 Spajanje na bazu podataka](#)
 - [5.2 CRUD operacije](#)
 - [5.3 MongoDB operatori](#)
 - [5.3.1 Operatori ažuriranja \(eng. Update operators\)](#)
 - [5.3.2 Operatori usporedbe \(eng. Comparison operators\)](#)
 - [5.3.3 Logički operatori \(eng. Logical operators\)](#)

- [Samostalni zadatak za Vježbu 5](#)
 - [Nadogradnja pizzerija aplikacije \(1 bod\)](#)
 - [Dodavanje naručivanja \(1 bod\)](#)

1. MongoDB

MongoDB je dokumentno-orientirana (eng. document-oriented) baza podataka koja se koristi za pohranu podataka u formatu sličnom JSON-u. MongoDB razvija tvrtka MongoDB Inc. i dostupna je kao [source-available](#) softver. MongoDB je popularna baza podataka zbog svoje skalabilnosti, fleksibilnosti i jednostavnosti korištenja.

Općenito, baze podataka možemo podijeliti na relacijske i nerelacijske (NoSQL).

1. **Relacijske baze podataka** (eng. *Relational database*) pohranjuju podatke u tabličnom formatu koristeći **redove** i **stupce**, a odnosi između podataka definiraju se pomoću **ključeva**. Primjeri relacijskih baza podataka uključuju MySQL, PostgreSQL, SQLite, Oracle.
2. **Nerelacijske baze podataka** (eng. *NoSQL database*) pohranjuju podatke u formatu koji nije tabličan. Nerelacijske baze podataka koriste različite modele za pohranu podataka, kao što su **dokumenti**, **ključ-vrijednost**, **stupci** ili **grafovi**. Primjeri nerelacijskih baza podataka uključuju MongoDB, Cassandra, Redis, Neo4j.

Postoje [prednosti i nedostaci](#) oba pristupa, a odabir baze podataka ovisi o specifičnim zahtjevima projekta. Općenito, nerelacijske baze podatke pružaju veću fleksibilnost jer ne zahtijevaju unaprijed definiranu shemu. To ih čini idealnim za aplikacije koje rade s velikim količinama nestrukturiranih podataka ili polustrukturiranih podataka.

Dokumenti u MongoDB bazi podataka pohranjeni su u [BSON](#) formatu (*Binary JSON*), koji je binarna reprezentacija JSON formata.

Dva osnovna gradivna elementa MongoDB baze podataka su **dokumenti** i **kolekcije**:

- **Dokument** (eng. *Document*) je ustvari **jedan zapis** (eng. *record*), koji se prikazuje strukturom koja sadrži ključ-vrijednost parove, baš kao i JSON objekt.
- **Kolekcija** (eng. *Collection*) je **skup dokumenata**. Kolekcije u MongoDB bazi podataka su ekvivalent tablicama u relacijskim bazama podataka i služe za **grupiranje srodnih dokumenata**.



1.1 MongoDB Atlas

MongoDB moguće je koristiti na više načina, ovisno o potrebama projekta na kojem radimo. Moguće ga je preuzeti i instalirati na računalno lokalno, međutim mi to nećemo raditi za potrebe ovog kolegija, već ćemo umjesto toga koristiti Cloud uslugu MongoDB Atlas.

MongoDB Atlas je **cloud usluga** koja omogućuje jednostavno stvaranje, upravljanje i skaliranje MongoDB baza podataka u **oblaku**. Usluga je dostupna na <https://www.mongodb.com/docs/atlas/> i omogućuje brzo postavljanje MongoDB baze podataka bez potrebe za instalacijom i konfiguracijom lokalnog MongoDB poslužitelja.



Atlas značajno pojednostavljuje upravljanje i održavanje MongoDB baze podataka. Developer se može fokusirati na razvoj aplikacije, dok se MongoDB Atlas brine o infrastrukturi i sigurnosti baze podataka, kao i o automatskom skaliranju i replikaciji podataka.

Ova usluga se plaća, ali [postoji i besplatan plan](#) za male aplikacije i učenje. Za potrebe vašeg projekta i ovog kolegija, dovoljno je koristiti upravo besplatan plan.

Prvi korak je registracija MongoDB Atlas računa. Registrirajte se na

<https://www.mongodb.com/cloud/atlas/register> i slijedite upute za registraciju. Preporuka je koristiti Google račun za prijavu (može biti i studentski mail).

1. Jednom kad se prijavite, **morate stvoriti novu organizaciju**. Organizacija je najviša razina u MongoDB Atlasu i služi za grupiranje projekata i timova. Izradu organizacije možete započeti klikom na `create Organization` unutar `/preferences/organizations` stranice.

Organizations

A screenshot of the MongoDB Atlas 'Create an organization' page. It features a 3D icon of a city skyline. The main text reads 'To get started, create an organization' and 'Within an organization, you can easily create projects, invite other users, and set up a billing account.' Below this is a 'Create an Organization' button and a link 'Learn more about organizations and projects.'.

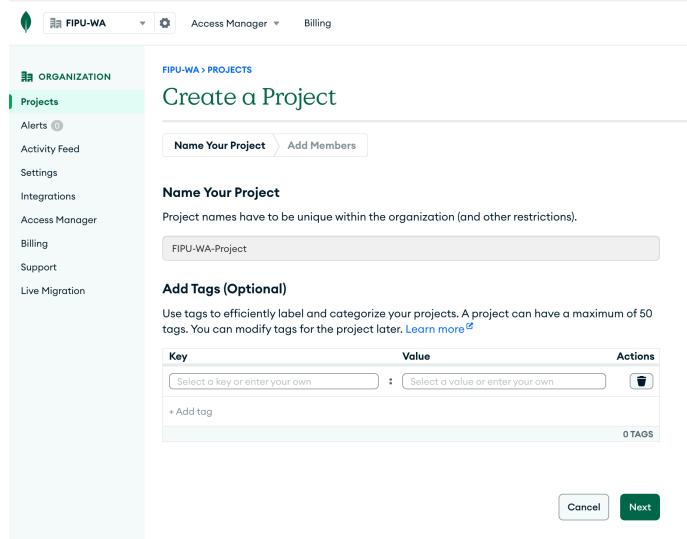
Dostupno na <https://cloud.mongodb.com/v2#/preferences/organizations>

Aplikaciju nazovite `FIPU`, `FIPU-WA` ili nešto u tom stilu, odaberite `MongoDB Atlas` i kliknite `Next`.

Možete dodati i članove vaše organizacije, za sada preskočite ovaj korsak i kliknite na `Create Organization`.

1. **Nakon što ste stvorili organizaciju, morate stvoriti projekt.** Projekt je druga razina u hijerarhiji MongoDB Atlasa i služi za grupiranje baza podataka i podjelu resursa između timova i različitih aplikacija.

Mi ćemo izraditi samo jedan projekt, možete ga nazvati `FIPU-WA-Project`. Kliknite na `New Project`, unesite naziv projekta i odaberite `Next`.



Preskočite dodavanje članova projekta i kliknite na `Create Project`.

1. Nakon što ste stvorili projekt, možete stvoriti cluster. Cluster je ustvari MongoDB baza podataka koja se izvršava u oblaku. Radi se ustvari o skupini MongoDB poslužitelja koji rade zajedno kako bi osigurali visoku dostupnost i pouzdanost baze podataka.

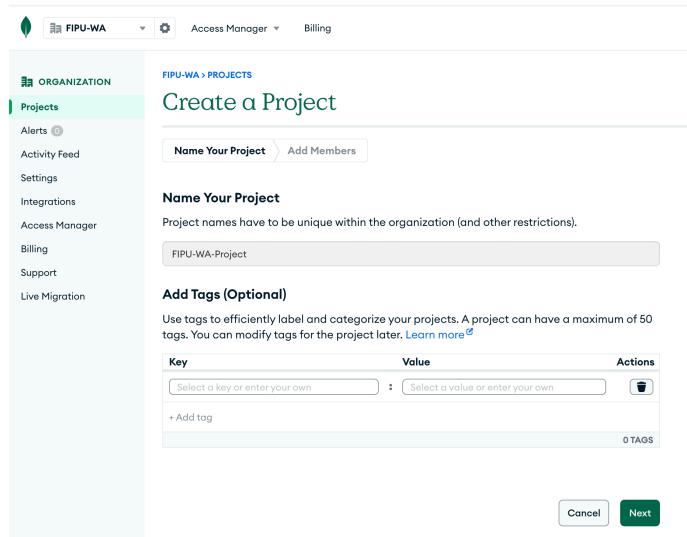
Odaberite `Create a cluster` → `M0 Cluster` (besplatan plan).

MongoDB Atlas za vas rješava sve tehničke detalje oko postavljanja i konfiguracije, uključujući infrastrukturu gdje će se baza podataka *deployati*. Međutim možete izabrati poslužitelja i regiju koja je fizički najbliža vašoj lokaciji.

Od poslužitelja, moguće je odabratи `AWS`, `Azure` ili `GCP`. Mi ćemo odabratи `AWS` → `Frankfurt (eu-central-1)`.

Dodijelite i neki naziv *clusteru*, npr. `FIPU-WA-Cluster` i kliknite na `Create Deployment`.

Možete i odabratи opciju `Preload sample dataset` kako bi se u vašu bazu podataka učitao uzorak podataka s kojim možete raditi.



Nakon što se *cluster* izradi, morat ćete izraditi novog korisnika koji će se koristiti za pristup bazi podataka. Automatski će se unijeti vaša IP adresa, korisničko ime i generirati lozinka.

Spremite lozinku jer će vam trebati za spajanje na bazu podataka.

Connect to FIPU-WA-Cluster

1 Set up connection security 2 Choose a connection method 3 Connect

You need to secure your MongoDB Atlas cluster before you can use it. Set which users and IP addresses can access your cluster now. [Read more ↗](#)

1. Add a connection IP address

✓ Your current IP address (89.164.116.219) has been added to enable local connectivity. Only an IP address you add to your Access List will be able to connect to your project's clusters. Add more later in [Network Access ↗](#).

2. Create a database user

This first user will have [atlasAdmin ↗](#) permissions for this project.

We autogenerated a username and password. You can use this or create your own.

ⓘ You'll need your database user's credentials in the next step. Copy the database user password.

Username Password [SHOW](#) [Copy](#)

[Create Database User](#)

[Close](#) [Choose a connection method](#)

2. Povezivanje na cluster

Jednom kad ste uspješno napravili *cluster* u MongoDB Atlasu, možete se povezati na njega na više načina:

- [MongoDB Compass](#) aplikacija (Desktop GUI za MongoDB; omogućuje jednostavan pregled i manipulaciju podacima u bazi)
- [MongoDB Shell](#) (CLI za MongoDB; omogućuje izvršavanje naredbi nad bazom podataka i pregled podataka)
- [MongoDB Node.js native driver](#) (Node.js biblioteka za povezivanje na MongoDB bazu podataka; ovo ćemo koristiti u nastavku skripte)
- [MongoDB for VS Code](#) (VS Code ekstenzija za MongoDB; omogućuje pregled podataka u bazi iz VS Code, vrlo praktično u razvoju)

Mi ćemo u nastavku koristiti **MongoDB native driver za Node.js** kako bismo se povezali na bazu podataka unutar našeg Express poslužitelja.

- Moguće je (i preporučljivo) koristiti i druge alate za povezivanje, kako biste imali bolji uvid u podatke u bazi i kako biste mogli brže i jednostavnije raditi s podacima na više razina apstrakcije.

Ako ste sve odradili kako treba, trebali biste vidjeti podatke o vašem *clusteru* u MongoDB Atlasu. Odabirom na `Browse Collections` možete vidjeti i kolekcije koje su automatski kreirane u vašoj bazi podataka ako ste odabrali opciju `Preload sample dataset`.

The screenshot shows the MongoDB Compass interface for the 'FIPU-WA-Cluster' database. The 'Collections' tab is selected, displaying the 'sample_mflix.comments' collection. The collection has 4079 documents and a total size of 1.79MB. The interface includes a search bar, filters, and a results table showing two document snippets. The first snippet is:

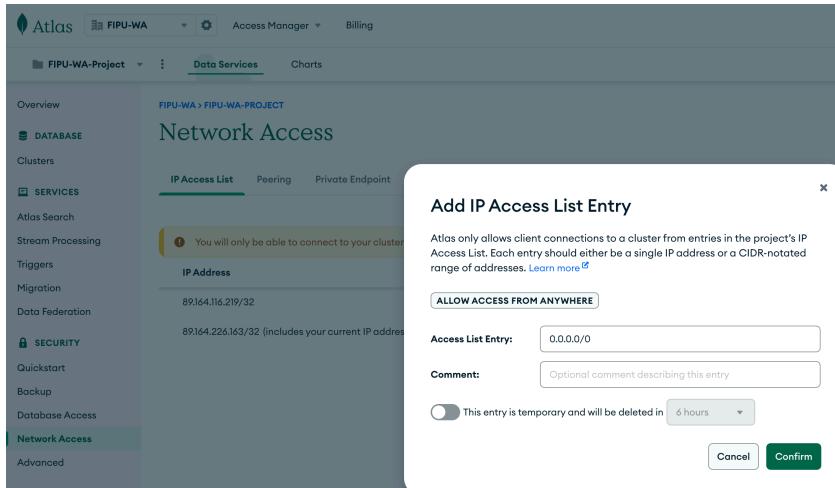
```
_id: ObjectId('5a9427648b0beebe69579e7')
name: "Mercedes Tyler"
email: "mercedes_tyler@fakegmail.com"
movie_id: ObjectId('573a1390f29313cabcd4323')
text: "Etius veritatis vero facilis quareat fuga temporibus. Praesentium exped_"
date: 2002-08-18T04:56:07.000+00:00
```

The second snippet is:

```
_id: ObjectId('5a9427648b0beebe69579f5')
name: "John Bishop"
email: "john_bishop@fakegmail.com"
movie_id: ObjectId('573a1390f29313cabcd446f')
text: "Id error ab at molestias dolorum incidunt. Non deserunt praesentium do_"
date: 1975-01-21T00:31:22.000+00:00
```

Prije nego krenemo s povezivanjem na Atlas, potrebno je unutar `Security/Network Access` dodati IP adresu u whitelistu kako bi se mogli povezati na bazu podataka s našeg računala. Ovo je dodatna sigurnosna mjera kako bi se spriječilo neovlašteno povezivanje na bazu podataka.

Međutim, kako se dinamička IP adresa našeg računala povremeno mijenja, nije loše privremeno (isključivo u procesu razvoja i učenja), omogućiti pristup sa svih IP adresa. Ovo možete učiniti tako da dodate zapis `0.0.0.0/0`.



Naravno, Atlasu vaše aplikacije se i dalje pristupa preko *connection stringa*.

2.1 Priprema Express poslužitelja

Prije nego krenemo s povezivanjem na bazu podataka, pripremit ćemo osnovni Express poslužitelj. Vraćamo se na poslužitelj za naručivanje pizze iz prethodnih vježbi 

Napravite novi direktorij i definirajte osnovni Express poslužitelj u `index.js` datoteci:

```
import express from 'express';

const app = express();

app.use(express.json());

app.get('/', (req, res) => {
  res.send('Pizza app');
});

const PORT = 3000;
app.listen(PORT, error => {
  if (error) {
    console.log('Greška prilikom pokretanja servera', error);
  }
  console.log(`Pizza poslužitelj dela na http://localhost:${PORT}`);
});
```

2.2 Connection string

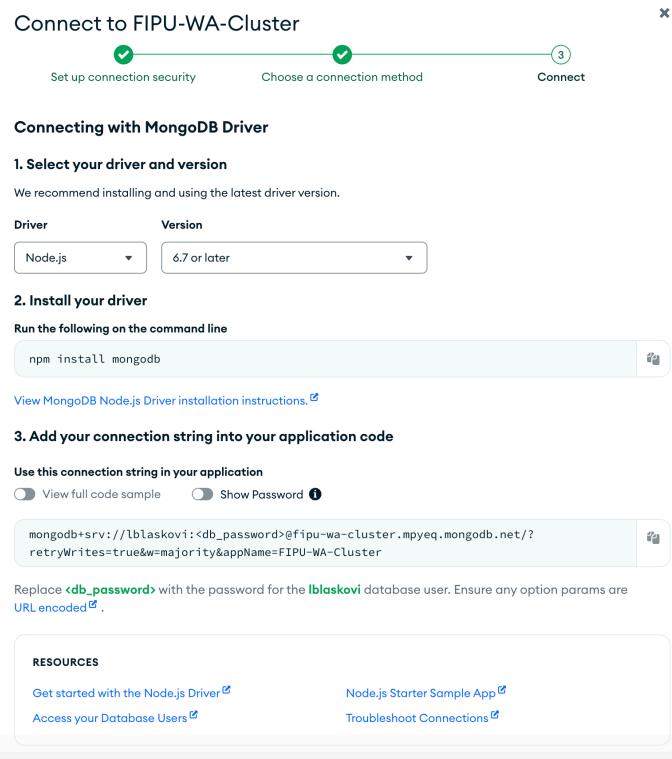
Povezivanje koristeći MongoDB native driver za Node.js realizira se kroz tzv. **Connection string**. *Connection string* je niz znakova koji sadrži informacije potrebne za povezivanje na vaš konkretni *cluster* u MongoDB Atlasu.

Odaberite svoj *cluster* u MongoDB Atlasu i kliknite na `connect` gumb. Odaberite `Drivers`.

Odaberite `Node.js` kao driver i najnoviju verziju drivera. Mi ćemo koristiti `6.7 or later`.

Napomena: Moguće je koristiti i `Mongoose` driver za povezivanje na MongoDB bazu. Mongoose je ORM (Object-Relational Mapping) biblioteka za MongoDB i omogućuje definiranje sheme i modela za Mongo bazu. Više o Mongoose biblioteci možete pročitati na <https://mongoosejs.com/>. Rad s ovom bibliotekom je izvan opsega ovog kolegija.

Kopirajte vaš **Connection string** na sigurno mjesto, tamo gdje ste kopirali i generirani password



Connection string predložak izgleda otprilike ovako:

```
mongodb+srv://<username>:<password>@<cluster>.cluster.mpyeq.mongodb.net//?  
retryWrites=true&w=majority&appName=<appname>
```

Sastoji se od:

- **protokola:** `mongodb+srv://`
- **credentials:** `<username>:<password>`
- **hostname/IP adresa i port:** `<cluster>.cluster.mpyeq.mongodb.net`
- **dodatnih opcija:** `?retryWrites=true&w=majority&appName=<appname>`



Važno! Connection string je privatni podatak i ne smije se dijeliti s drugima (**sadrži sve podatke potrebne za spajanje na vaš Mongo cluster**). Ukoliko ga dijelite, osigurajte se da ste ga uklonili iz javno dostupnih repozitorija ili datoteka. U nastavku ćemo vidjeti kako možemo koristiti `.env` datoteku za pohranu osjetljivih podataka te ju dodati u `.gitignore` kako bi se spriječilo slanje osjetljivih podataka na GitHub udaljeni repozitorij.

2.3 db.js

Prije nego što se povežemo na bazu podataka, moramo instalirati MongoDB driver za Node.js. Instalirajte `mongodb` paket koristeći `npm`:

```
npm install mongodb
```

Do sad smo naučili nekoliko dobrih praksi u razvoju poslužitelja:

- ne želimo sve "trpati" u `index.js` datoteku, već stvaramo **modularnu strukturu aplikacije** kroz Router objekte.
- u `index` datoteci koristimo `app.use()` metodu za povezivanje Router objekata na određene rute.

Jednako tako, i kod povezivanja na bazu podataka, **dobra praksa je da izdvojimo logiku povezivanja u zasebnu datoteku**. Stvorite novu datoteku `db.js` u kojoj ćemo definirati logiku povezivanja na bazu podataka. Glavninu logike možete pronaći prilikom generiranja connection stringa u MongoDB Atlasu, međutim ona je zapisana u *commonjs* sintaksi, mi ćemo ju pojednostaviti kroz *ES6* sintaksu.

Ideja je da možemo koristiti `db.js` datoteku kao modul u našem Express poslužitelju, kako bismo se u svakoj datoteci (npr. u Router objektima) mogli spojiti na bazu podataka.

```
touch db.js
```

Uključit ćemo `MongoClient` klasu iz `mongodb` paketa:

```
import { MongoClient } from 'mongodb';
```

Pohranjujemo *Connection string* u varijablu (uobičajeno je izdvojiti naziv cluster-a u zasebnu varijablu):

- naravno, zaliđepite vaš *Connection string* iz MongoDB Atlasa

```
const mongoURI = 'mongodb+srv://<username>:<password>@<cluster>.cluster.mpyeq.mongodb.net/?retryWrites=true&w=majority&appName=<appname>';
const db_name = 'sample_mflix'; // naziv predefinirane baze podataka
```

Zatim definiramo asinkronu funkciju `connectToDatabase` koja će se koristiti za povezivanje na bazu podataka:

Definirat ćemo `client` varijablu koja će sadržavati **instancu MongoClient klase**:

Sintaksa:

```
const client = new MongoClient(url: string, options?: MongoClientOptions);
```

U opcijama možemo definirati objekt s dodatnim opcijama, za sada ćemo to ostaviti prazno.

Popis svih opcija možete pronaći na [sljedećoj poveznici](#).

Jednom kad definirate klijent, povezujemo se metodom `connect()`.

- Kod ćemo omotati `try-catch` blokom kako bismo uhvatili eventualne greške prilikom spajanja na bazu podataka.

- U slučaju greške, ispisujemo poruku u konzolu i bacamo grešku (koristimo `throw` naredbu).
- `throw` naredba prekida izvršavanje trenutne funkcije i vraća grešku.
- Grešku koju baca `throw` naredba možemo uhvatiti koristeći `catch` blok kasnije u kodu.
- U varijablu `db` spremamo referencu na bazu podataka koju smo odabrali (u našem slučaju `sample_mflix`).

```
async function connectToDatabase() {
  try {
    const client = new MongoClient(mongoURI); // stvaramo novi klijent
    await client.connect(); // spajamo se na klijent
    console.log('Uspješno spajanje na bazu podataka');
    let db = client.db(db_name); // odabiremo bazu podataka
    return db;
  } catch (error) {
    console.error('Greška prilikom spajanja na bazu podataka', error);
    throw error;
  }
}
```

Izvesti ćemo funkciju `connectToDatabase` kako bismo ju mogli koristiti u drugim datotekama:

```
export { connectToDatabase };
```

Unutar `index.js` datoteke, importat ćemo funkciju `connectToDatabase` i pozvati ju nakon definiranja instance poslužitelja:

```
import { connectToDatabase } from './db.js';

const app = express();

let db = await connectToDatabase();
```

Ponovno pokrenite Express poslužitelj i provjerite konzolu. Ako se uspješno spojite na bazu podataka, trebali biste vidjeti poruku `Uspješno spajanje na bazu podataka`.

2.4 `dotenv` modul

Kako bismo spriječili slanje osjetljivih podataka na GitHub, koristit ćemo `.env` datoteku za **pohranu osjetljivih podataka**.

Općenito, *environment* varijable su varijable okoline koje se koriste za pohranu osjetljivih podataka kao što su lozinke, API ključevi, database credentials, ili bilo koje druge postavke koje mogu varirati ovisno o okolini (npr. razvojna, testna, produkcijska okolina).

- U našem kontekstu, želimo spriječiti pohranu *Connection stringa* MongoDB baze podataka na GitHub.

U Node.js aplikacijama, možemo koristiti `dotenv` paket za učitavanje *environment* varijabli iz `.env` datoteke.

Instalirajte `dotenv` paket koristeći `npm`:

```
npm install dotenv
```

Stvorite `.env` datoteku u korijenskom direktoriju vašeg projekta:

```
touch .env
```

Unutar `.env` datoteke, definirajte vaše osjetljive podatke. Uobičajeno je environment variable pisati velikim slovima i koristiti `_` za razdvajanje riječi:

Svi podaci s desne strane znaka `=` su stringovi, **ne trebate koristiti navodnike**.

```
MONGO_URI=mongodb+srv://<username>:<password>@<cluster>.cluster.mpyeq.mongodb.net//?
retryWrites=true&w=majority&appName=<appname>
MONGO_DB_NAME=sample_mflix
```

Nakon što ste pohranili osjetljive podatke u `.env` datoteku, možete ih učitati u vašu aplikaciju koristeći `dotenv` paket.

U `db.js` datoteci, uvezat ćemo `dotenv` paket i učitati osjetljive podatke iz `.env` datoteke:

```
import { config } from 'dotenv';

config(); // učitava osjetljive podatke iz .env datoteke
```

Varijablama sad pristupamo unutar objekta `process.env`

Testirajmo:

```
console.log(process.env.MONGO_URI);
console.log(process.env.MONGO_DB_NAME);
```

Ako ne radi, pokušajte pokrenuti novu instancu terminala.

Sad možemo zamijeniti `mongoURI` i `db_name` varijable s `process.env.MONGO_URI` i `process.env.MONGO_DB_NAME`:

```
import { MongoClient } from 'mongodb';

import { config } from 'dotenv';

config(); // učitava osjetljive podatke iz .env datoteke

let mongoURI = process.env.MONGO_URI;
let db_name = process.env.MONGO_DB_NAME;

async function connectToDatabase() {
  try {
    const client = new MongoClient(mongoURI); // stvaramo novi klijent
    await client.connect(); // spajamo se na klijent
    console.log('Uspješno spajanje na bazu podataka');
    let db = client.db(db_name); // odabiremo bazu podataka
```

```
    return db;
  } catch (error) {
    console.error('Greška prilikom spajanja na bazu podataka', error);
    throw error;
  }
}
export { connectToDatabase };
```

Na kraju, ne smijemo zaboraviti dodati `.env` datoteku u `.gitignore` kako bismo spriječili njeno slanje na GitHub.

Osim `.env` datoteke, možete dodati i `node_modules` direktorij kako biste spriječili pohranu svih paketa našeg projekta. Ovo je korisno jer ne želimo slati pakete na GitHub, budući da ih možemo ponovno instalirati koristeći `npm install` ako su definirani u `package.json` datoteci.

Datoteku `.gitignore` dodajete u korijenskom direktoriju vašeg projekta, sa sljedećim sadržajem:

```
node_modules
.env
```

Struktura projekta sad bi trebala izgledati otprilike ovako:

```
.
├── .env
├── .gitignore
├── db.js
├── index.js
├── node_modules
├── package-lock.json
└── package.json
```

3. CRUD operacije

CRUD (*Create, Read, Update, Delete*) su osnovne operacije koje se izvršavaju nad podacima u bazi podataka.

U MongoDB bazi podataka, CRUD operacije se izvršavaju nad **dokumentima u kolekcijama**.

1. **Create** (*stvaranje*) - dodavanje novog dokumenta u kolekciju
2. **Read** (*čitanje*) - dohvaćanje podataka iz kolekcije
3. **Update** (*ažuriranje*) - ažuriranje postojećeg dokumenta u kolekciji
4. **Delete** (*brisanje*) - brisanje dokumenta iz kolekcije

Vidimo da su CRUD operacije analogue HTTP metodama.

Ovisno o kompleksnosti strukture projekta, CRUD operacije moguće je pisati direktno unutar definicije ruta u Express poslužitelju, ili ih možemo izdvojiti u zasebne datoteke kako bismo imali bolju organizaciju koda. **Za početak ćemo ih pisati direktno unutar definicije ruta.**

3.1 GET operacija

Prisjetimo se 2 osnovne GET rute koje smo definirali u Express poslužitelju za dohvaćanje svih pizza i pojedinačne pizze:

```
app.get('/pizze', (req, res) => {
  res.status(200).json(pizze);
});

app.get('/pizze/:id', (req, res) => {
  const id = req.params.id;
  const pizza = pizze.find(pizza => pizza.id === id); // Oprez, ovo je metoda Array.find()
  koja dohvaća prvi element koji zadovoljava callback predikat
  res.status(200).json(pizza);
});
```

Podatke smo prethodno definirali *in-memory*, ali i unutar JSON datoteke, a sada ćemo ih pohraniti u MongoDB Atlas.

Prije nego to napravimo, pokušat ćemo dohvatiti postojeće podatke iz predefinirane baze podataka `sample_mflix`. Prvi korak je definirati kolekciju iz koje ćemo dohvatiti podatke.

U MongoDB Atlasu, kliknite na `Browse Collections` za definirani `cluster` i odaberite kolekciju iz koje ćemo dohvatiti podatke. Recimo, iz kolekcije `users` (`sample_mflix.users`).

Zapamti! `cluster = FIPU-WA-Cluster`, baza podataka = `sample_mflix`, kolekcija = `users`

Kolekciju dohvaćamo koristeći `db.collection()` metodu, gdje je `db` referenca na bazu podataka koju smo dobili kao rezultat funkcije `connectToDatabase()`.

```
let allUsers = db.collection('users');
```

Mongo metoda: `collection().find()`

Možemo dohvatiti sve dokumente iz kolekcije koristeći `collection().find()` metodu (ekvivalentno SQL upitu `SELECT * FROM users`).

Važno! Ovo metoda, različita je od metode `Array.find()` koju smo koristili u prethodnim primjerima. Ova metoda vraća **Cursor** objekt kad se poziva nad MongoDB kolekcijom, a ne *in-memory* poljem.

Sintaksa:

```
db.collection.find(filter, options);
```

gdje su opcionalni parametri:

- `filter` - opcionalni objekt koji **sadrži kriterije pretrage** (npr. `{ name: 'John' }`), ekvivalentno `WHERE name = 'John'` SQL izrazu; ako se ne navede - vraćaju se svi dokumenti. Postoji puno kriterija pretrage, više o tome u nastavku
- `options` - opcionalni objekt koji **sadrži dodatne opcije** (npr. `{ projection: { name: 1, age: 1 } }`), ekvivalentno `SELECT name, age FROM ...` SQL izrazu. U nastavku više o ovom argumentu, za sada ćemo ga ostaviti praznim.

Dohvatit ćemo sve korisnike iz kolekcije `users`:

```
let allUsers = await users.find(); // dohvaća sve dokumente iz kolekcije
```

`find()` metoda vraća `Cursor` objekt - **pokazivač na rezultate upita**. Da bismo dohvatili same rezultate, koristimo `Iterator.toArray()` metodu.

```
let allUsers = await users.find().toArray(); // dohvaća sve dokumente iz kolekcije kao Array
```

Ovaj kod možemo ubaciti u GET rutu `/users`:

```
app.get('/users', async (req, res) => {
  let users_collection = db.collection('users'); // pohranjujemo referencu na kolekciju
  let allUsers = await users_collection.find().toArray(); // dohvaćamo sve korisnike iz
  // kolekcije i pretvaramo Cursor objekt u Array
  res.status(200).json(allUsers);
});
```

Pošaljite zahtjev na `http://localhost:3000/users` i provjerite jesu li podaci uspješno dohvaćeni iz baze podataka.

3.1.1 GET /pizze

Sada ćemo izraditi kolekciju s podacima o pizzama i implementirati odgovarajuće GET rute za dohvaćanje **svih pizza i pojedinačne pizze**.

Otvorite sučelje vašeg Atlas Clustera i odaberite `Browse Collections`. Kliknite na `+ Create Database` i nazovite bazu podataka `pizza_db`, `pizzeria` ili sl.

Definirajte prvu kolekciju i nazovite ju `pizze`.

Jednom kad to napravite, vidjet ćete praznu kolekciju `pizze`. Kliknite na `Insert Document` - unijet ćemo nekoliko dokumenata.

The screenshot shows the MongoDB Compass interface. At the top, it displays 'DATABASES: 2' and 'COLLECTIONS: 7'. Below this, there's a sidebar with '+ Create Database' and a search bar for namespaces. The main area shows the 'pizza_db.pizze' collection. It has a storage size of 4KB, logical data size of 0B, and 0 total documents. The indexes total size is 4KB. There are tabs for 'Find', 'Indexes', 'Schema Anti-Patterns', 'Aggregation', and 'Search Indexes'. A button for 'Generate queries from natural language in Compass' is also present. At the bottom, there's a 'QUERY RESULTS: 0' section with a 'Filter' dropdown and a text input 'Type a query: { field: 'value' }'. Below that are 'Reset', 'Apply', and 'Options' buttons. A prominent 'INSERT DOCUMENT' button is located at the top right of the main collection view.

Prvo što ćete uočiti je da je **podatak ID već unesen**, u MongoDB bazi podataka svaki dokument mora imati jedinstveni identifikator, a on se označava s `_id` poljem te je tipa `ObjectId`.

Podatke na web sučelju Atlasa možete dodavati na dva načina:

- pisanjem `JSON` strukture (ustvari je `BSON`)
- kroz sučelje za unos podataka

```
{ "_id": { "$oid": "674d808cbbb8072b29ae839f" } }
```

Na postojeći zapis dodajemo polja `naziv` i `cijena`.

```
{
  "_id": { "$oid": "674d808cbbb8072b29ae839f" },
  "naziv": "Capricciosa",
  "cijena": 11
}
```

Preko sučelja izgleda ovako:

The screenshot shows the 'Insert Document' dialog in MongoDB Compass. It starts with the heading 'Insert Document' and 'To collection pizze'. Below that is a code editor containing the following JSON document:

```
1   _id: ObjectId('674d808cbbb8072b29ae839f')
2   naziv: "Capricciosa"
3   cijena: 11
```

On the right side of the dialog, there's a schema viewer showing the types for each field:

| | |
|---------------------|-----------------------|
| <code>_id</code> | <code>ObjectId</code> |
| <code>naziv</code> | <code>String</code> |
| <code>cijena</code> | <code>Int32</code> |

At the bottom right are two buttons: 'Cancel' and 'Insert'.

Dodajte sljedeće pizze u kolekciju:

```
{
  "naziv": "Margherita",
  "cijena": 9
},
{
  "naziv": "Quattro Stagioni",
  "cijena": 13
},
{
  "naziv": "Pepperoni",
  "cijena": 11
}
```

```

    "naziv": "Quattro Formaggi",
    "cijena": 15
},
{
    "naziv": "Vegetariana",
    "cijena": 12
},
{
    "naziv": "Šunka sir",
    "cijena": 10
}

```

Postupak je moguće i malo ubrzati kloniranjem postojećeg zapisa (Mongo će automatski generirati novi `_id` za svaki!):

The screenshot shows the MongoDB Compass interface with three documents in the 'pizza' collection:

- `_id: ObjectId('674d8243bbb8072b29ae839f')`
naziv : "Capricciosa"
cijena : 11
- `_id: ObjectId('674d820d0bbb8072b29ae83a1')`
naziv : "Margherita"
cijena : 9
- `_id: ObjectId('674d8243bbb8072b29ae83a2')`
naziv : "Quattro Stagioni"
cijena : 13

A 'Clone document' button is visible at the bottom right.

Nakon što ste dodali pizze, možemo ih dohvatiti na isti način kao prethodno korisnike, koristeći metodu `collection().find()`

```

app.get('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze'); // referenca na kolekciju 'pizze'
  let allPizze = await pizze_collection.find().toArray(); // pretvorba u Array
  res.status(200).json(allPizze);
});

```

Napomena, potrebno je još unutar `.env` datoteke promijeniti vrijednost varijable `MONGO_DB_NAME` na `pizza_db`, ili kako ste već nazvali bazu.

```
MONGO_DB_NAME=pizza_db
```

Testirajte dohvaćanje svih pizza na ruti: <http://localhost:3000/pizze>

The screenshot shows a Postman request to `http://localhost:3000/pizze` with the following details:

- Method:** GET
- URL:** `http://localhost:3000/pizze`
- Response Status:** 200 OK
- Size:** 422 Bytes
- Time:** 46 ms
- Response Body:** A JSON array of 32 pizza documents, each with a unique `_id` and different values for `naziv` and `cijena`.

Ako ste dobili statusni kod 200 i podatke o pizzama, uspješno ste dohvatali podatke iz baze podataka.

3.1.2 GET /pizze/:naziv

U NoSQL bazama podataka nemamo strogo definiranu shemu (*eng. Database schema*) kao u relacijskim bazama podataka pa je moguće "na licu mjesta" mijenjati strukturu dokumenata.

Samim tim, nemamo niti strogo definirane ključeve, poput **Primary key** u relacijskim bazama podataka.

Međutim, postoji nešto što nalikuje ključevima, a to su indeksi. **Indeksi (eng. Index) su struktura podataka koja omogućuje brže pretraživanje podataka u bazi podataka.**

Bez indeksa, NoSQL baze podataka morale bi pretraživati svaki dokument u kolekciji kako bi pronašle odgovarajući dokument. **Indeksi omogućuju brže pretraživanje podataka jer se podaci pretražuju prema indeksu koji pokazuju na grupe podataka, a ne prema samim dokumentima** (koga zanima više, googlati B i B+ stabla). Samim tim, sve metode pretraživanja, filtriranja i sortiranja podataka su brže.

U MongoDB bazi podataka, indeksi se mogu ručno izraditi, a neki se i automatski stvaraju, npr. za ključ `_id`, koji je **jedinstveni identifikator svakog dokumenta**. Ovaj indeks omogućuje brže pretraživanje podataka prema `_id` ključu, što je i *defaultna vrijednost* kod metode `collection().find()`.

Kada otvorite određenu kolekciju na Atlasu, pronađite sekciju `Indexes`

The screenshot shows the MongoDB Atlas interface for the `pizza_db.pizze` collection. At the top, it displays storage details: Storage Size: 4KB, Logical Data Size: 0B, Total Documents: 0, and Indexes Total Size: 4KB. Below this, there are tabs for `Find`, **Indexes** (which is selected), `Schema Anti-Patterns`, `Aggregation`, and `Search Indexes`. A green button labeled `CREATE INDEX` is visible. A yellow banner at the bottom says "We're sorry, an unexpected error has occurred." with a `TRY AGAIN` button. The main table lists the index `_id` with the following details:

| Name, Definition, and Type | Size | Usage | Properties | Action |
|---|---------|--------------------------------|------------|--------|
| <code>_id</code> <code>_id</code> REGULAR | 36.0 KB | <1/min since Mon Dec 2 2024 | | |

Uočite postojeći indeks na `_id` polje, koji je automatski dodan prilikom dodavanja prvog dokumenta u kolekciju.

Do sad smo definirali GET rutu za dohvaćanje pojedine pizze po ID-u. Međutim, tada su nam ID-evi bili jednostavni brojevi koje smo ručno definirali i bili su sekvencijalnog slijeda `0, 1, 2, 3, 4, 5...`. Dodavanjem zapisa, jednostavno smo dohvatili posljednji ID i dodali `+1`.

Ovdje to nije moguće jer su nam ID-evi kompleksni `objectId` objekti koje MongoDB automatski dodaje prilikom dodavanja novog zapisa. Samim tim, nešto je komplikiranije definirati endpoint `/pizze/:id`.

Kako se radi o aplikaciji za pizzeriju, možemo se složiti da su **pizze u meniju također jedinstveni podaci** pa možemo iskoristiti `naziv` ključ kao ključ po kojem ćemo pretraživati/dohvaćati.

Međutim, rekli smo da je pretraživanje po `_id` polju brže jer je indeksirano. Kako ćemo onda pretraživati po `naziv` polju?

► Spoiler alert! Odgovor na pitanje

Dodavanje indeksa možemo odraditi putem Atlas web sučelja ili direktno u kodu. Za sada ćemo direktno preko web sučelja.

Pizzu po nazivu želimo dohvatiti unutar GET rute `/pizze/:naziv`

Koristeći metodu `collection().find()` možemo definirati filter pretrage:

```
collection().find({ naziv: 'naziv_pizze' });
```

Mongo metoda: `collection().findOne()`

Možemo koristiti i metodu `collection().findOne()` koja vraća samo prvi dokument koji zadovoljava kriterije pretrage (`filter`).

Metoda u principu radi poput `Array.find()` metode, ali ne pišemo callback funkciju, već `filter` objekt.

Sintaksa:

```
collection().findOne(filter); // vraća samo 1 dokument
```

```
collection().findOne({ naziv: 'naziv_pizze' }); // vrati prvi dokument koji ima naziv 'naziv_pizze'
```

Ako koristimo metodu `findOne()`, uvjek dobivamo samo jedan dokument pa ne moramo koristiti `toArray()` metodu.

Dodajemo parametar rute `naziv` koji ćemo koristiti za pretragu:

```
app.get('/pizze/:naziv', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let naziv_param = req.params.naziv;
  let pizza = await pizze_collection.find({ naziv: naziv_param }).toArray();
  // ili
  let pizza = await pizze_collection.findOne({ naziv: naziv_param }); // samo 1 rezultat, ne koristimo metodu Iterator.toArray()
  res.status(200).json(pizza);
});
```

Testirajte, npr. na `http://localhost:3000/pizze/Margherita`.

- Kod radi, ali nismo još dodali indeks.

Postoji mnogo vrsta indeksa u Mongu, mi ćemo za sada dodati tzv. **Single Field Text Index** na `naziv` polje koji će optimizirati sljedeću pretragu:

```
db.pizze.find({ naziv: 'Capricciosa' });
```

Indeksi se definiraju *ključ-vrijednost* sintaksom:

```
<naziv_polja>: <tip_indeksa>
```

U našem slučaju:

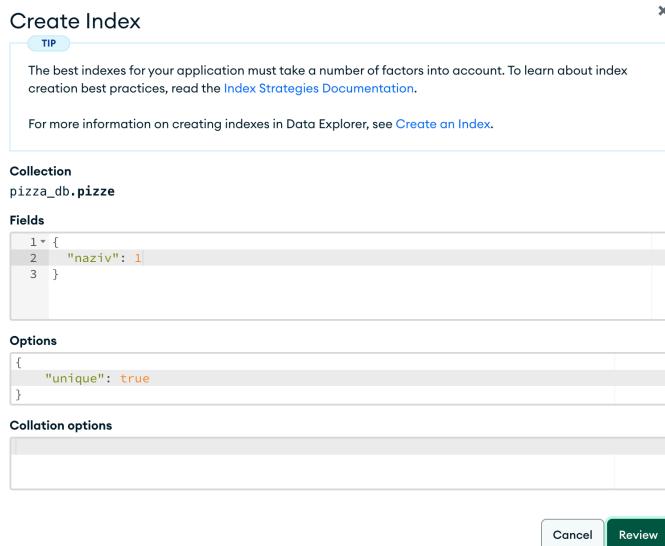
```
"naziv" : 1,
```

- Naziv polja/ključa je `naziv`, a tip indeksa je `1` što označava uzlazni indeks, dok `-1` označava silazni indeks.

Možemo dodati i `unique` **svojstvo indeksa unutar opcija**, kako bismo osigurali da su svi nazivi pizza jedinstveni:

```
{
  "unique": true
}
```

Dodajemo indeks preko Atlas web sučelja:



Ako izostavite JSON objekt (vitičaste zagrade) kod Options, dobit ćete grešku.

Možemo vidjeti nadodani indeks i automatski dodijeljeni naziv `naziv_1` gdje `_1` označava uzlazni indeks.

| Name, Definition, and Type | Size | Usage | Properties | Action |
|----------------------------|---------|-----------------------------|------------|--------|
| <code>_id</code> | 36.0 kB | <1/min since Mon Dec 2 2024 | | |
| <code>naziv_1</code> | 20.0 kB | <1/min since Mon Dec 2 2024 | UNIQUE | ⚡️ 🚫 |

Testirajte kod, stvari ostaju iste, ali sada je pretraga po nazivu optimizirana (premda to ne uočavamo na malom broju podataka i malom broju GET zahtjeva).

3.2 POST operacija

Dodat ćemo mogućnost dodavanja novih pizza u kolekciju `pizze`, a nakon toga i stvaranje narudžbe u kolekciju `narudzbe`.

3.2.1 POST /pizze

Definirajmo prvo kostur endpointa:

```
app.post('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let novaPizza = req.body;
  res.status(201).json(); // 201 jer smo kreirali novi resurs
});
```

Mongo metoda: `collection().insertOne()`

Novi dokument (točno jedan) u kolekciju dodajemo pomoću `collection().insertOne()` metode:

Sintaksa:

```
db.collection('naziv_kolekcije').insertOne(object);
```

Povratna vrijednost ove metode je objekt koji sadrži:

- `acknowledged` - boolean vrijednost koja označava je li operacija uspješno izvršena (`true`) ili nije (`false`)
- `insertedId` - ID novododanog dokumenta (`objectId`)

Naš objekt je u tijelu HTTP zahtjeva, koji sad mora izgledati ovako:

```
{
  "naziv": "Slavonska",
  "cijena": 14
}
```

Moramo paziti na 3 stvari prilikom definiranja HTTP zahtjeva:

- da sadrži sve potrebne ključeve (`naziv`, `cijena`)
- da sadrži jedinstveni ključ `naziv` jer smo tako definirali indeksom `naziv_1`
- da je u JSON formatu

```
app.post('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let novaPizza = req.body;
  let result = await pizze_collection.insertOne(novaPizza);
  res.status(201).json(result.insertedId); // Vraćamo klijentu ID novododanog dokumenta
});
```

Provjerite je li se dodao dokument s novom pizzom u Atlasu.

Ako pokušate dodati istu pizzu, dobit ćete grešku jer smo to spriječili indeksom (ovu zabranu zamislite kao `SQL UNIQUE` ograničenje ili `BEFORE INSERT TRIGGER`)

Međutim, greška nije obrađena pa se naš Express poslužitelj ruši...

Ispis greške u konzoli:

```
ErrorResponse: {
  index: 0,
  code: 11000,
  errmsg: 'E11000 duplicate key error collection: pizza_db.pizze index: naziv_1 dup key: {
naziv: "Slavonska" }',
  keyPattern: { naziv: 1 },
  keyValue: { naziv: 'Slavonska' }
},
index: 0,
code: 11000,
keyPattern: { naziv: 1 },
keyValue: { naziv: 'Slavonska' },
[Symbol(errorLabels)]: Set(0) {}
}
```

Grešku možemo pročitati unutar `result`, preciznije u `result.errorResponse` objektu.

Kako kod "pukne" na liniji `await pizze_collection.insertOne(novaPizza);`, moramo dodati `try-catch` blok kako bismo uhvatili grešku i poslali odgovarajući status klijentu.

```
app.post('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let novaPizza = req.body;
  let result = {}; // inicijaliziramo prazan objekt
  try {
    result = await pizze_collection.insertOne(novaPizza);
  } catch (error) {
    console.log(error.errorResponse);
  }
  res.status(201).json(result); // vraćamo klijentu cijeli result objekt
});
```

Vidimo ispis `result.errorResponse` u konzoli, **pitanje**: Zašto se klijentu nije vratio objekt `result`, ako smo tako naveli u posljednjoj liniji?

► Spoiler alert! Odgovor na pitanje

Gotovo nikada u programiranju web poslužitelja ne želimo koristiti strukturu endpointa kao što je implementirano iznad:

- ne želimo definirati inicijalno prazan `result` objekt (općenito kad definiramo inicijalno praznu varijablu, vjerojatno nešto radimo krivo)
- ne želimo vraćati korisniku cijeli `result` objekt, već samo informacije koje su mu potrebne
- ispravno je premjestiti slanja HTTP odgovora unutar rezolucija `try-catch` bloka

Ispravno je sljedeće:

```
app.post('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let novaPizza = req.body;
  try {
    let result = await pizze_collection.insertOne(novaPizza);
    res.status(201).json({ insertedId: result.insertedId }); // Kad šaljemo JSON, moramo
    podatak spremiti u neki ključ
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse }); // 400 jer je korisnik poslao
    neispravne podatke
  }
});
```

Testirajte dodavanje nove pizze putem HTTP klijenta, kao i dodavanje iste pizze dvaput.

Greška se sada obrađuje i klijentu se šalje cijeli objekt greške (koji onda klijent obrađuje na svojoj strani):

The screenshot shows a POST request to `http://localhost:3000/pizze`. The request body contains a JSON object with fields `naziv` and `cijena`. The response status is 400 Bad Request, with a size of 206 Bytes and a time of 74 ms. The response body is a JSON object containing an `error` field with details about a duplicate key error.

```
1  {
2    "error": {
3      "index": 0,
4      "code": 11000,
5      "errmsg": "E11000 duplicate key error collection: pizza_db.pizze index: naziv_1 dup key: {
6        \"naziv\": \"$Slavonska\" }",
7      "keyPattern": {
8        "naziv": 1
9      },
10     "keyValue": {
11       "naziv": "Slavonska"
12     }
13   }
```

Međutim ako dodamo novu pizzu `Fantasia`:

```
{
  "naziv": "Fantasia",
  "cijena": 12.5
}
```

The screenshot shows a POST request to `http://localhost:3000/pizze`. The request body contains a JSON object with fields `naziv` and `cijena`. The response status is 201 Created, with a size of 41 Bytes and a time of 73 ms. The response body is a JSON object containing an `insertedId` field.

```
1  {
2    "insertedId": "674d9ec10d1918579e32b79b"
3  }
```

Provjerite zapise u Atlasu.

3.2.2 POST /narudzbe

Vrlo slično možemo dodati i novu narudžbu u kolekciju `narudzbe`. Prvo ćemo izraditi kolekciju `narudzbe` u Atlasu (iako nije nužno, MongoDB će automatski stvoriti kolekciju ako ne postoji).

Endpoint možemo definirati identično kao i `POST /pizze` budući da dodajemo točno 1 zapis, samo ćemo promijeniti naziv kolekcije.

Tijelo zahtjeva definiramo direktno na klijentskoj strani, odnosno u HTTP klijentu:

```
app.post('/narudzbe', async (req, res) => {
  let narudzbe_collection = db.collection('narudzbe');
  let novaNarudzba = req.body;
  try {
    let result = await narudzbe_collection.insertOne(novaNarudzba);
    res.status(201).json({ insertedId: result.insertedId });
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
  }
});
```

Što uopće moramo definirati u tijelu zahtjeva?

Što će se desiti ako u tijelu pošaljemo samo sljedeće?

```
{
  "kupac": "Marko Marić"
}
```

► Spoiler alert! Odgovor na pitanje

Validacija zahtjeva na poslužitelju

Definirat ćemo jednostavnu validaciju na način koji smo i do sad radili, koristeći čisti JavaScript.

Najlakše je započeti definicijom JSON strukture koju očekujemo: Kupac je jedan, ali može naručiti više pizza. Za svaku pizzu osim naziva, moramo navesti i veličinu. Međutim, možemo naručiti dvije iste pizze, ali različitih veličina i količina.

Primjer strukture JSON tijela zahtjeva:

```
{
  "kupac": "Marko Marić",
  "narucene_pizze": [
    {
      "naziv": "Capricciosa",
      "kolicina": 2,
      "velicina": "srednja"
    },
    {
      "naziv": "Vegetariana",
      "kolicina": 1,
      "velicina": "velika"
    }
  ]
}
```

```
{
  "naziv": "Capricciosa",
  "količina": 1,
  "veličina": "mala"
},
{
  "naziv": "Šunka sir",
  "količina": 3,
  "veličina": "srednja"
}
]
}
```

Osim toga, moramo proslijediti i adresu za dostavu te broj telefona.

```
{
  "kupac": "Marko Marić",
  "adresa": "Vodnjanska 12, 52100 Pula",
  "broj_telefona": "098 123 456",
  "narucene_pizze": [
    {
      "naziv": "Capricciosa",
      "količina": 2,
      "veličina": "srednja"
    },
    {
      "naziv": "Vegetariana",
      "količina": 1,
      "veličina": "velika"
    },
    {
      "naziv": "Capricciosa",
      "količina": 1,
      "veličina": "mala"
    },
    {
      "naziv": "Šunka sir",
      "količina": 3,
      "veličina": "srednja"
    }
  ]
}
```

Kako ovo sada validirati?

Možemo u `Array` obaveznih ključeva dodati ključeve koje očekujemo: `kupac`, `adresa`, `broj_telefona` i `narucene_pizze`.

Nakon toga, za svaki ključ iz tog polja, u *callback* funkciji provjeravamo postoji li taj ključ u tijelu zahtjeva.

- prvo pretvaramo objekt `novaNarudzba` u Array njezinih ključeva
- zatim provjeravamo za svaki ključ iz `obavezniKljucevi` postoji li u `novaNarudzba`

```

app.post('/narudzbe', async (req, res) => {
  let narudzbe_collection = db.collection('narudzbe');
  let novaNarudzba = req.body;

  let obavezniKljucevi = ['kupac', 'adresa', 'broj_telefona', 'narucene_pizze'];

  // pretvaramo objekt novaNarudzba u Array ključeva, pa provjeravamo sa Array.includes()
  if (!obavezniKljucevi.every(kljuc => Object.keys(novaNarudzba).includes(kljuc))) {
    return res.status(400).json({ error: 'Nedostaju obavezni ključevi' });
  }

  try {
    let result = await narudzbe_collection.insertOne(novaNarudzba);
    res.status(201).json({ insertedId: result.insertedId });
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
  }
});

```

Međutim, provjeru iznad je moguće i skratiti koristeći novi operator `in` koji provjerava je li navedeni ključ prisutan u objektu:

```
key in object;
```

Radi se o modernoj ES6 JavaScript sintaksi koja jako nalikuje Python sintaksi.

Važno! Ovaj operator može se koristiti na ovaj način samo za objekte, ne polja!

Iz toga razloga ne moramo pretvarati objekt u Array ključeva, već možemo direktno provjeriti ključeve:

```

if (!obavezniKljucevi.every(kljuc => kljuc in novaNarudzba)) {
  return res.status(400).json({ error: 'Nedostaju obavezni ključevi' });
}

```

Još moramo provjeriti svaku naručenu pizzu iz polja `narucene_pizze`:

Možemo iterirati kroz polje `narucene_pizze` i za svaku pizzu provjeriti jesu li navedeni ključevi: `naziv`, `količina` i `veličina`. Idemo istom logikom i ove ključeve pohraniti u varijablu:

```

app.post('/narudzbe', async (req, res) => {
  let narudzbe_collection = db.collection('narudzbe');
  let novaNarudzba = req.body;

  let obavezniKljucevi = ['kupac', 'adresa', 'broj_telefona', 'narucene_pizze'];
  // ključevi koje ćemo provjeravati za svaku pizzu (stavku narudžbe)
  let obavezniKljuceviStavke = ['naziv', 'količina', 'velicina'];

  if (!obavezniKljucevi.every(kljuc => kljuc in novaNarudzba)) {
    return res.status(400).json({ error: 'Nedostaju obavezni ključevi' });
  }
}

```

```

// za svaku stavku narudžbe provjeravamo obavezne ključeve na isti način
for (let stavka of novaNarudzba.narucene_pizze) {
    if (!obavezniKljuceviStavke.every(kljuc => kljuc in stavka)) {
        return res.status(400).json({ error: 'Nedostaju obavezni ključevi u stavci narudžbe' });
    }
}

try {
    let result = await narudzbe_collection.insertOne(novaNarudzba);
    res.status(201).json({ insertedId: result.insertedId });
} catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
}
});

```

Ili možemo ugnijezditi još jednu `every` metodu kako bi izbjegli `for` petlju:

```

app.post('/narudzbe', async (req, res) => {
    let narudzbe_collection = db.collection('narudzbe');
    let novaNarudzba = req.body;

    let obavezniKljucevi = ['kupac', 'adresa', 'broj_telefona', 'narucene_pizze'];
    let obavezniKljuceviStavke = ['naziv', 'kolicina', 'veličina'];

    if (!obavezniKljucevi.every(kljuc => kljuc in novaNarudzba)) {
        return res.status(400).json({ error: 'Nedostaju obavezni ključevi' });
    }

    // za svaku stavku narudžbe provjeravamo obavezne ključeve, ovaj put ugniježđenom `every` metodom
    if (!novaNarudzba.narucene_pizze.every(stavka => obavezniKljuceviStavke.every(kljuc => kljuc in stavka))) {
        return res.status(400).json({ error: 'Nedostaju obavezni ključevi u stavci narudžbe' });
    }

    try {
        let result = await narudzbe_collection.insertOne(novaNarudzba);
        res.status(201).json({ insertedId: result.insertedId });
    } catch (error) {
        console.log(error.errorResponse);
        res.status(400).json({ error: error.errorResponse });
    }
});

```

Testirajte kod, maknite neki od obveznih ključeva iz tijela zahtjeva i provjerite je li validacija ispravna.

```

1 {
2   "kupac": "Marko Marić",
3   "adresa": "Vodnjanska 12, 52100 Pula",
4   "broj_telefona": "098 123 456",
5   "narucene_pizze": [
6     {
7       "naziv": "Capricciosa",
8       "kolicina": 2,
9       "velicina": "srednja"
10      },
11      {
12        "naziv": "Vegetariana",
13        "kolicina": 1,
14        "velicina": "velika"
15      },
16      {
17        "naziv": "Capricciosa",
18        "kolicina": 1
19      },
20      {
21        "naziv": "Sunka sir",
22        "kolicina": 3,
23        "velicina": "srednja"
24      }
25    ]
26  }

```

Status: 400 Bad Request Size: 59 Bytes Time: 31 ms

Response Headers 6 Cookies Results Docs

```

1 {
2   "error": "Nedostaju obavezni ključevi u stavci narudžbe"
3 }

```

Potpuna validacija ključeva u tijelu zahtjeva za endpoint **POST /narudzbe**.

Provjerite na Atlasu je li nova narudžba dodana.

```

_id: ObjectId('674db5a3a96e615617af91d3')
kupac : "Marko Marić"
adresa : "Vodnjanska 12, 52100 Pula"
broj_telefona : "098 123 456"
▼ narucene_pizze: Array (4)
  ▶ 0: Object
    naziv : "Capricciosa"
    kolicina : 2
    velicina : "srednja"
  ▶ 1: Object
    naziv : "Vegetariana"
    kolicina : 1
    velicina : "velika"
  ▶ 2: Object
    naziv : "Capricciosa"
    kolicina : 1
    velicina : "mala"
  ▶ 3: Object

```

U prikazu vidimo klasične JSON označke (Array, Object), ali i `objectId` označke koje MongoDB automatski dodaje.

Nadogradit ćemo validaciju podataka dodatnim provjerama. Za svaku naručenu pizzu (stavku narudžbe), želimo provjeriti:

1. postoji li pizza u bazi podataka?
2. je li `količina` tipa `integer` i veća od 0?
3. je li `veličina` jedna od triju veličina: `mala`, `srednja`, `velika`?

Dodat ćemo prvo 2. i 3. provjeru, budući da smo to već radili u prethodnim primjerima.

```

app.post('/narudzbe', async (req, res) => {
  let narudzbe_collection = db.collection('narudzbe');
  let novaNarudzba = req.body;

  let obavezniKljucevi = ['kupac', 'adresa', 'broj_telefona', 'narucene_pizze'];
  let obavezniKljuceviStavke = ['naziv', 'kolicina', 'velicina'];

  if (!obavezniKljucevi.every(kljuc => kljuc in novaNarudzba)) {
    return res.status(400).json({ error: 'Nedostaju obavezni ključevi' });
  }
}

```

```

    if (!novaNarudzba.narucene_pizze.every(stavka => obavezniKljuceviStavke.every(kljuc =>
      kljuc in stavka))) {
        return res.status(400).json({ error: 'Nedostaju obavezni ključevi u stavci narudžbe' });
    }

    // dodajemo dodatne provjere za svaku stavku narudžbe
    // negacija uvjeta: budući da 'every' vraća true ako je za svaki element polja uvjet
    ispunjen
    if (
      !novaNarudzba.narucene_pizze.every(stavka => {
        // provjeravamo 3 uvjeta: količina je integer i veća od 0, veličina je jedna od triju
        veličina
        return Number.isInteger(stavka.količina) && stavka.količina > 0 && ['mala', 'srednja',
        'velika'].includes(stavka.veličina);
      })
    ) {
      return res.status(400).json({ error: 'Neispravni podaci u stavci narudžbe' });
    }

    try {
      let result = await narudzbe_collection.insertOne(novaNarudzba);
      res.status(201).json({ insertedId: result.insertedId });
    } catch (error) {
      console.log(error.errorResponse);
      res.status(400).json({ error: error.errorResponse });
    }
  );
}

```

Dodat ćemo još provjeru **postoji li pizza u bazi podataka**. To ćemo napraviti tako da za svaku pizzu iz polja `narucene_pizze` provjerimo postoji li pizza s tim nazivom u kolekciji `pizze`.

Prvo dohvaćamo kolekciju `pizze` iz baze:

```
let pizze_collection = db.collection('pizze');
```

Raspakiramo u `Array` sve dokumente iz kolekcije:

```
let dostupne_pizze = await pizze_collection.find().toArray();
```

Dakle, ako je korisnik proslijedio barem jednu pizzu koje nema u bazi podataka, trebamo mu vratiti grešku.

HINT: "Barem jednu" → koristite `Array.some()` metodu

```

if (!novaNarudzba.narucene_pizze.every(stavka => dostupne_pizze.some(pizza => pizza.naziv ===
stavka.naziv))) {
  return res.status(400).json({ error: 'Odabrali ste pizzu koju nemamo u ponudi' });
}

```

POST <http://localhost:3000/narudzbe>

Body 1

```

1 {
2   "kupac": "Marko Marić",
3   "adresa": "Vodnjanska 12, 52100 Pula",
4   "broj_telefona": "098 123 456",
5   "narucene_pizze": [
6     {
7       "naziv": "Nutella",
8       "kolicina": 2,
9       "velicina": "srednja"
10    },
11    {
12      "naziv": "Vegetariana",
13      "kolicina": 1,
14      "velicina": "velika"
15    },
16    {
17      "naziv": "Capricciosa",
18      "kolicina": 1,
19      "velicina": "mala"
20    },
21    {
22      "naziv": "Šunka sir",
23      "kolicina": 3,
24      "velicina": "srednja"
25    }
26  ]
27 }

```

Status: **400 Bad Request** Size: **51 Bytes** Time: **44 ms**

Response

```

1 {
2   "error": "Odabrali ste pizzu koju nemamo u ponudi"
3 }

```

Slanje zahtjeva s pizzom Nutella| koja nije u bazi podataka

3.3 PUT i PATCH operacije

Do sad ste naučili da se PUT i PATCH metode koriste za ažuriranje podataka. Razlika između njih je u tome što **PUT metoda zamjenjuje cijeli dokument novim**, dok **PATCH metoda ažurira samo određene dijelove dokumenta**.

U kontekstu naše pizzerije, implementirat ćemo PATCH metodu za ažuriranje statusa narudžbe i cijene pizze. PUT metodu koristit ćemo za zamjenu cijelog menija novim.

3.3.1 PATCH /pizze/:naziv

Prvo ćemo definirati PATCH metodu za ažuriranje cijene pizze. Kako smo već definirali GET metodu za dohvaćanje pizze po nazivu, možemo koristiti istu logiku.

```
app.get('/pizze/:naziv', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let naziv_param = req.params.naziv;
  let pizza = await pizze_collection.findOne({ naziv: naziv_param });
  res.status(200).json(pizza);
});
```

Mongo metoda: collection().updateOne()

Za ažuriranje točno jednog dokumenta koristimo collection().updateOne() metodu. Ova metoda očekuje maksimalno **2 obavezna parametra**:

- `filter` - **obavezni parametar**, definira objekt kojim opisujemo koji podatak želimo ažurirati.
 - Npr. isto kao i kod `collection().find()` metode, možemo direktno navesti `{ naziv: 'Capricciosa' }`, (ekvivalentno SQL izrazu `WHERE naziv = 'Capricciosa'`).
- `update` - **obavezni parametar**, kojim definiramo što želimo ažurirati. Ovaj parametar je **objekt koji sadrži ključeve koje želimo ažurirati i nove vrijednosti tih ključeva**.
 - Npr. `{ $set: { cijena: 15 } }`, (ekvivalentno SQL izrazu `SET cijena = 15`). Operator koji pišemo na mjestu ključa zove se **update operator**.
- `options` - **opcionalni parametar**, koji definira dodatne opcije ažuriranja.
 - Npr. `{ upsert: true }`, što znači da će se novi dokument dodati ako ne postoji dokument koji zadovoljava `filter`.

MongoDB Update operatori

Update operatori ključeva (eng. Field Update Operators) su sljedeći:

- `$set` - postavlja vrijednost ključa na novu vrijednost
- `$unset` - briše ključ iz dokumenta
- `$inc` - povećava vrijednost ključa za određeni integer
- `$mul` - množi vrijednost ključa s određenim brojem

- `$min` - postavlja vrijednost ključa na novu vrijednost samo ako je postojeća vrijednost manja od nove
- `$max` - postavlja vrijednost ključa na novu vrijednost samo ako je postojeća vrijednost veća od nove
- `$rename` - preimenuje ključ u dokumentu
- `$currentDate` - postavlja vrijednost ključa na trenutni datum
- postoji ih još...

Ispred update operatora uvijek ide znak `$`

Sintaksa:

```
db.collection('naziv_kolekcije').updateOne({ filter }, { update }); // gdje su filter i
update objekti
```

Primjeri korištenja *update operatora* i *updateOne()* metode:

1. Želimo zamijeniti cijenu pizze `capricciosa` s novom cijenom `15`:

```
// update operator
{
  $set: {
    cijena: 15;
  }
}

// updateOne() metoda
collection().updateOne({ naziv: 'Capricciosa' }, { $set: { cijena: 15 } });
```

2. Želimo zamijeniti naziv pizze `Capricciosa` s novim nazivom `Capricciosa Supreme`:

```
// update operator
{
  $set: {
    naziv: 'Capricciosa Supreme';
  }
}

// updateOne() metoda
collection().updateOne({ naziv: 'Capricciosa' }, { $set: { naziv: 'Capricciosa Supreme' } });
```

3. Želimo povećati cijenu pizze `Capricciosa` za `2` eura:

```
// update operator
{
  $inc: {
    cijena: 2;
  }
}

// updateOne() metoda
collection().updateOne({ naziv: 'Capricciosa' }, { $inc: { cijena: 2 } });

```

4. Želimo obrisati cijenu pizze `Capricciosa`:

```
// update operator
{
  $unset: {
    cijena: '';
  }
}

// updateOne() metoda
collection().updateOne({ naziv: 'Capricciosa' }, { $unset: { cijena: '' } });

```

5. Želimo postaviti cijenu pizze `Capricciosa` na `10` eura, ali samo ako je trenutna cijena manja od `10` eura:

```
// update operator
{
  $min: {
    cijena: 10;
  }
}

// updateOne() metoda
collection().updateOne({ naziv: 'Capricciosa' }, { $min: { cijena: 10 } });

```

6. Želimo postaviti cijenu pizze `Margherita` na `20` eura, ali samo ako je trenutna cijena veća od `20` eura:

```
// update operator
{
  $max: {
    cijena: 20;
  }
}

// updateOne() metoda
collection().updateOne({ naziv: 'Margherita' }, { $max: { cijena: 20 } });

```

7. Želimo preimenovati ključ `cijena` u `cijena_eur` za pizzu `Quattro Stagioni`:

```
// update operator
{
  $rename: {
    cijena: 'cijena_eur';
  }
}

// updateOne() metoda
collection().updateOne({ naziv: 'Quattro Stagioni' }, { $rename: { cijena: 'cijena_eur' } });
});
```

8. Želimo postaviti ključ `datum_dodavanja` na trenutni datum za pizzu `Vegetariana`:

```
// update operator
{
  $currentDate: {
    datum_dodavanja: {
      $type: 'date';
    }
  }
}

// updateOne() metoda
collection().updateOne({ naziv: 'Vegetariana' }, { $currentDate: { datum_dodavanja: { $type: 'date' } } });
});
```

Dakle, endpoint za ažuriranje cijene pizze ćemo definirati koristeći `$set` update operator (ujedno i najčešće korišteni operator):

```
app.patch('/pizze/:naziv', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let naziv_param = req.params.naziv;
  let novaCijena = req.body.cijena;

  try {
    let result = await pizze_collection.updateOne({ naziv: naziv_param }, { $set: { cijena: novaCijena } });
    res.status(200).json({ modifiedCount: result.modifiedCount });
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
  }
});
```

Primjer slanja zahtjeva. Povećat ćemo cijenu pizze `Capricciosa` na `13` eura:

PATCH http://localhost:3000/pizze/Margherita

Body 1

```
1 {  
2   "cijena": 13  
3 }
```

Status: 200 OK Size: 19 Bytes Time: 120 ms

Kao odgovor dobivamo broj ažuriranih dokumenata (`modifiedCount : 1`).

Ovaj podatak možemo iskoristit kako bi se uvjerili u ispravnost ažuriranja te informaciju proslijediti klijentu, ali i dodati provjeru ako pizza nije pronađena (`modifiedCount == 0`)

```
app.patch('/pizze/:naziv', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let naziv_param = req.params.naziv;
  let novaCijena = req.body.cijena;

  try {
    let result = await pizze_collection.updateOne({ naziv: naziv_param }, { $set: { cijena: novaCijena } });

    if (result.modifiedCount === 0) {
      return res.status(404).json({ error: 'Pizza nije pronađena' });
    }

    res.status(200).json({ modifiedCount: result.modifiedCount });
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
  }
});
```

3.3.2 PATCH /narudzbe/:id

Na isti način ćemo definirati PATCH metodu za ažuriranje statusa narudžbe.

Prvo ćemo definirati jednostavni `GET /narudzbe` za dohvaćanje svih narudžbi.

```
app.get('/narudzbe', async (req, res) => {
  let narudzbe_collection = db.collection('narudzbe');
  let narudzbe = await narudzbe_collection.find().toArray();

  if (narudzbe.length === 0) {
    return res.status(404).json({ error: 'Nema narudžbi' });
  }

  res.status(200).json(narudzbe);
});
```

Kod filtera `collection().findOne()` metode koristimo `ObjectID` konstruktor kako bi pretvorili string ID iz URL parametra u MongoDB `ObjectID` tip podatka.

Ovaj konstruktor je potrebno učitati na početku datoteke:

```
import { ObjectId } from 'mongodb';
```

Dakle, metoda za dohvaćanje jedne narudžbe po ID-u bila bi sljedeća:

```
app.get('/narudzbe/:id', async (req, res) => {
  let narudzbe_collection = db.collection('narudzbe');
  let id_param = req.params.id;
  let narudzba = await narudzbe_collection.findOne({ _id: new ObjectId(id_param) }); // instanciramo objekt ObjectId

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

  res.status(200).json(narudzba);
});
```

The screenshot shows a Postman request for a GET request to `http://localhost:3000/narudzbe/674dbb61cd85b5d3bd083714`. The response status is `200 OK`, size is `377 Bytes`, and time is `43 ms`. The response body is a JSON object representing a pizza order with the following details:

```
{
  "_id": "674dbb61cd85b5d3bd083714",
  "kupac": "Marko Marić",
  "adresa": "Vodnjanska 12, 52100 Pula",
  "broj_telefona": "098 123 456",
  "narucene_pizze": [
    {
      "naziv": "Margherita",
      "kolicina": 2,
      "velicina": "srednja"
    },
    {
      "naziv": "Vegetariana",
      "kolicina": 1,
      "velicina": "velika"
    },
    {
      "naziv": "Capricciosa",
      "kolicina": 1,
      "velicina": "mala"
    },
    {
      "naziv": "Šunka sir",
      "kolicina": 3,
      "velicina": "srednja"
    }
  ]
}
```

ID smo kopirali ručno iz prethodnog odgovora (možemo i direktno iz Atlasa)

Kako bismo sad ažurirali status narudžbe, koristit ćemo `PATCH` metodu i `updateOne()` metodu sa `$set` operatorom. Bez obzira što ovog polja trenutno nema u narudžbi, on će se automatski dodati.

```
app.patch('/narudzbe/:id', async (req, res) => {
  let narudzbe_collection = db.collection('narudzbe');
  let id_param = req.params.id;
  let noviStatus = req.body.status; // npr. 'isporučeno', 'u pripremi', 'otkazano'

  try {
    let result = await narudzbe_collection.updateOne({ _id: new ObjectId(id_param) }, { $set: { status: noviStatus } });

    if (result.modifiedCount === 0) {
      return res.status(404).json({ error: 'Narudžba nije pronađena' });
    }

    res.status(200).json({ modifiedCount: result.modifiedCount });
  } catch (error) {
```

```

    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
}
});

```

3.3.3 PUT /pizze

Što ako želimo zamijeniti cijeli meni s pizzama odjednom. Primjerice, imamo korisničko sučelje na strani klijenta gdje možemo dodati, ažurirati i obrisati pizze iz menija. **Rezultat tih akcija je novi meni koji sadrži sve pizze** (odnosno `JSON` koji sadrži sve pizze).

Takav JSON izgledao bi otprilike ovako:

```
[
{
  "naziv": "Margherita",
  "cijena": 10.5
},
{
  "naziv": "Napolitana",
  "cijena": 12.5
},
{
  "naziv": "Funghi",
  "cijena": 11.5
},
{
  "naziv": "Capricciosa",
  "cijena": 13.5
},
{
  "naziv": "Vegetariana",
  "cijena": 14.5
},
{
  "naziv": "Šunka sir",
  "cijena": 15.5
},
{
  "naziv": "Quattro Stagioni",
  "cijena": 16.5
},
{
  "naziv": "Fantasia",
  "cijena": 17.5
}
]
```

Kada bi pizze dodavali ručno, trebali bi za svaki zapis pozvati postojeći endpoint `POST /pizze`.

Na frontendu bi to, koristeći `Axios` biblioteku, izgledalo ovako:

```
// gdje su pizze polje objekata prikazano iznad
for (let pizza of pizze) {
  axios
    .post('http://localhost:3000/pizze', pizza)
    .then(response => console.log(response.data))
    .catch(error => console.error(error));
}
```

Definirat ćemo endpoint `PUT /pizze` koji će zamijeniti cijeli meni s pizzama novim menijem odjednom.

Mongo metoda: `collection().insertMany()`

Metoda `collection().insertMany()` koristi se za dodavanje više dokumenata odjednom u kolekciju. Ova metoda očekuje **1 obavezni parametar**:

- `documents` - **obavezni parametar**, polje objekata koje želimo dodati u kolekciju.
 - Npr. `[{ naziv: 'Margherita', cijena: 10.5 }, { naziv: 'Napolitana', cijena: 12.5 }]`. Navedeno je ekvivalentno SQL izrazu: `INSERT INTO pizze (naziv, cijena) VALUES ('Margherita', 10.5), ('Napolitana', 12.5);`.
- `options` - **opcionalni parametar**, koji definira dodatne opcije dodavanja.
 - Npr. `{ ordered: false }`, što znači da će se svi dokumenti dodati. Po *defaultu*, ova metoda prestaje dodavati ako najde na grešku, parametrom `ordered: false` to se može spriječiti.

Sintaksa:

```
db.collection('naziv_kolekcije').insertMany([ dokument1, dokument2, ... ]); // samo
'documents' parametar
```

Primjer korištenja `insertMany()` metode endpoint `PUT /pizze`:

```
app.put('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let noviMeni = req.body;

  try {
    let result = await pizze_collection.insertMany(noviMeni); // dodajemo novi meni (polje
objekata)
    res.status(200).json({ insertedCount: result.insertedCount });
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
  }
});
```

Međutim ova metoda će sad samo dodati nove pizze u kolekciju, **a ne zamijeniti cijeli meni**.

Ovo možemo riješiti na dva načina:

1. **obrisati cijelu kolekciju** te zatim pozvati `collection().insertMany()` metodu koja će stvoriti novu kolekciju i ubaciti sve pizze

2. **obrisati sve pizze** iz kolekcije metodom `collection().deleteMany()` te zatim pozvati `insertMany()` metodu

Kako bismo obrisali cijelu kolekciju, možemo koristiti metodu `collection().drop()`:

```
await pizze_collection.drop();
```



```
// 1. način
app.put('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let noviMeni = req.body;

  try {
    await pizze_collection.drop(); // brišemo cijelu kolekciju
    let result = await pizze_collection.insertMany(noviMeni);
    res.status(200).json({ insertedCount: result.insertedCount });
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
  }
});
```

Ili koristimo metodu `deleteMany()`, bez da brišemo cijelu kolekciju (što je sigurnije):

```
// 2. način
app.put('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let noviMeni = req.body;

  try {
    await pizze_collection.deleteMany({}); // brišemo sve pizze iz kolekcije
    let result = await pizze_collection.insertMany(noviMeni);
    res.status(200).json({ insertedCount: result.insertedCount });
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
  }
});
```

The screenshot shows a REST API testing interface. The URL is `http://localhost:3000/pizza`. The method is `PUT`. The `Body` tab is selected, showing a JSON array of 8 pizza items. The response tab shows a status of `200 OK`, size of `19 Bytes`, and time of `100 ms`. The response body contains the message `"insertedCount": 8`.

```
PUT http://localhost:3000/pizza Send
Query Headers 2 Auth Body 1 Tests Pre Run
JSON XML Text Form Form-encode GraphQL Binary
JSON Content Format
1 [
2   {
3     "naziv": "Margherita",
4     "cijena": 10.5
5   },
6   {
7     "naziv": "Napolitana",
8     "cijena": 12.5
9   },
10  {
11    "naziv": "Funghi",
12    "cijena": 11.5
13  },
14  {
15    "naziv": "Capricciosa",
16    "cijena": 13.5
17  },
18  {
19    "naziv": "Vegetariana",
20    "cijena": 14.5
21  },
22  {
23    "naziv": "Sunka sir",
24    "cijena": 15.5
25  },
26  {
27    "naziv": "Quattro Stagioni",
28    "cijena": 16.5
29  },
30  {
31    "naziv": "Fantasia",
32    "cijena": 17.5
33  }
34 ]
```

| Response | |
|--------------------------------------|---------|
| Headers 6 | Cookies |
| Results | Docs |
| 1 { 2 "insertedCount": 8 3 } | |

Primjer slanja zahtjeva za zamjenu cijelog menija s pizzama

3.4 DELETE operacija

Za kraj CRUD operacija, pogledat ćemo još jednu metodu - `DELETE`. Ova metoda koristi se za brisanje podataka iz baze podataka.

Moguće je brisati pojedinačni podatak, više podataka prema nekom filteru ili cijelu kolekciju (kao što ste već vidjeli iznad).

Pokazat ćemo primjer brisanja pizze iz menija prema nazivu.

3.4.1 DELETE `/pizze/:naziv`

Mongo metoda: `collection().deleteOne()`

Koristit ćemo metodu `deleteOne()` koja briše točno jedan dokument iz kolekcije (**prvi koji pronađe**). Ova metoda očekuje **1 obavezni parametar**:

- `filter` - **obavezni parametar**, definira filter objekt koji opisuje podatak koji želimo obrisati, isto kao i kod `collection().find()` metode. Npr. `{ naziv: 'Capricciosa' }`, (ekvivalentno SQL izrazu `WHERE naziv = 'Capricciosa'`).

Sintaksa:

```
db.collection('naziv_kolekcije').deleteOne({ filter });
```

```
app.delete('/pizze/:naziv', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let naziv_param = req.params.naziv;

  try {
    let result = await pizze_collection.deleteOne({ naziv: naziv_param }); // brišemo pizzu
    prema nazivu
    res.status(200).json({ deletedCount: result.deletedCount });
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
  }
});
```

Primjer brisanja pizze `Capricciosa` iz menija:

The screenshot shows a Postman request configuration. The method is set to `DELETE` and the URL is `http://localhost:3000/pizze/Margherita`. The `Body` tab is selected, showing a JSON object with a single key `naziv` set to `Margherita`. The response tab shows a status of `200 OK`, a size of `18 Bytes`, and a time of `75 ms`. The response body is a JSON object with one key `deletedCount` set to `1`.

Mongo metoda: `collection().deleteMany()`

Ako želimo obrisati više dokumenata iz kolekcije, koristimo metodu `deleteMany()`. Ova metoda očekuje **1 obavezni parametar**:

- `filter` - **obavezni parametar**, definira koji dokumenti želimo obrisati, isto kao i kod `collection().find()` metode.
 - Npr. `{ cijena: { $gte: 15 } }`, (ekvivalentno SQL izrazu `WHERE cijena >= 15`). Ako navedemo samo `{}`, obrisat će se svi dokumenti iz kolekcije.

Sintaksa:

```
db.collection('naziv_kolekcije').deleteMany({ filter }); //
```

```
app.delete('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');

  try {
    let result = await pizze_collection.deleteMany({}); // brišemo sve pizze iz kolekcije
    res.status(200).json({ deletedCount: result.deletedCount });
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
  }
});
```

4. Agregacija podataka

Agregacija podataka odnosi se na obradu podataka na temelju nekog kriterija. Krenut ćemo s primjerom filtriranja podataka koji smo već koristili u prethodnim primjerima. Međutim, nadograđujemo stvari na način da ćemo se prisjetiti `query` parametra HTTP zahtjeva te ih kombinirati s `MongoDB` upitima.

4.1 Filtriranje podataka

Prisjetimo se `query` parametra:

- rekli smo da ih definiramo unutar URL-a nakon znaka `?`
- svaki `query` parametar sastoji se od ključa i vrijednosti, npr. `?ključ1=vrijednost1`
- više `query` parametara odvajamo znakom `&`, npr. `?ključ1=vrijednost1&ključ2=vrijednost2`

Primjer URL-a s `query` parametrima:

```
http://localhost:3000/pizze?cijena=10&naziv=Capricciosa
```

Kod definicije endpointa, **ove parametre ne navodimo direktno u URL-u**, već ih dohvaćamo iz `req.query` objekta.

Vidjeli smo kako filtrirati, točnije pretraživati, određeni dokument u kolekciji koristeći `collection().find()` metodu:

```
let pizze = await pizze_collection.find({ naziv: 'Capricciosa' }).toArray(); // pronađi pizzu
čiji je naziv 'Capricciosa'
```

Rekli smo da, ako se radi uvijek o jednom dokumentu, koristimo `findOne()` metodu:

```
let pizza = await pizze_collection.findOne({ naziv: 'Capricciosa' }); // pronađi pizzu čiji je naziv 'Capricciosa'
```

Kako bismo pronašli sve pizze čija je cijena jednaka 10, pišemo sljedeći kod:

```
let pizze = await pizze_collection.find({ cijena: 10 }).toArray(); // pronađi sve pizze čija je cijena 10
```

Međutim, kako bismo pronašli sve pizze gdje je cijena minimalno 10, ili maksimalno 15.

Koristimo sljedeće MongoDB operatore usporedbe (eng. Comparison Operators):

- `$gte` - veće ili jednako (`greater than or equal`)
- `$lte` - manje ili jednako (`less than or equal`)
- `$gt` - veće (`greater than`)
- `$lt` - manje (`less than`)
- `$eq` - jednako (`equal`)
- `$ne` - nije jednako (`not equal`)

Kao i kod operatora usporedbe, i ovdje ispred ide znak \$

Primjeri korištenja comparison operatora i find() metode:

1. Želimo pronaći sve pizze čija je cijena veća ili jednaka 10 :

```
// comparison operator
{
  cijena: {
    $gte: 10;
  }
}

// find() metoda
collection().find({ cijena: { $gte: 10 } });
```

2. Želimo pronaći sve pizze čija je cijena manja ili jednaka 15 :

```
// comparison operator
{
  cijena: {
    $lte: 15;
  }
}

// find() metoda
collection().find({ cijena: { $lte: 15 } });
```

3. Želimo pronaći sve pizze čija je cijena veća od 10 i manja od 15 :

```
// comparison operator
{
  cijena: {
    $gt: 10,
    $lt: 15;
  }
}

// find() metoda
collection().find({ cijena: { $gt: 10, $lt: 15 } });
```

4. Želimo pronaći sve pizze čija je cijena različita od 5:

```
// comparison operator
{
  cijena: {
    $ne: 5;
  }
}

// find() metoda
collection().find({ cijena: { $ne: 5 } });
```

4.1.1 GET /pizze?query

Kombinirat ćemo ova dva pristupa (query parametre i MongoDB operatore usporedbe) kako bismo filtrirali pizze prema cijeni.

Očekuje se da će korisnik poslati `query` parametar `cijena` u URL-u, npr. `http://localhost:3000/pizze?cijena=10`.

```
app.get('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let cijena_query = req.query.cijena;

  try {
    let pizze = await pizze_collection.find({ cijena: Number(cijena_query) }).toArray(); // provjerava se točno podudaranje cijene
    res.status(200).json(pizze);
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
  }
});
```

U redu, ali ako sad pošaljemo zahtjev bez query parametra `cijena`, dobit ćemo prazan Array. Moramo obraditi uvjet gdje korisnik nije poslao `query` parametar.

```
app.get('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');
```

```

let cijena_query = req.query.cijena;

if (!cijena_query) {
  let pizze = await pizze_collection.find().toArray(); // dohvaćamo sve pizze
  return res.status(200).json(pizze);
}

try {
  let pizze = await pizze_collection.find({ cijena: Number(cijena_query) }).toArray(); // provjerava se točno podudaranje cijene
  res.status(200).json(pizze);
} catch (error) {
  console.log(error.errorResponse);
  res.status(400).json({ error: error.errorResponse });
}
});
```

Status: 200 OK Size: 71 Bytes Time: 50 ms

| Response | Headers 6 | Cookies | Results | Docs |
|--|-----------|---------|---------|------|
| <pre> 1 [2 { 3 "_id": "674de678d0113a49fff63e85", 4 "naziv": "Napolitana", 5 "cijena": 12.5 6 } 7]</pre> | | | | |

Primjer slanja zahtjeva s query parametrom `cijena=10`

Ako bismo htjeli pronaći sve pizze čija je cijena veća ili jednaka `10`, koristili bismo `$gte` operator:

```

let pizze = await pizze_collection.find({ cijena: { $gte: Number(cijena_query) } })
.toArray(); // dohvaćamo pizze čija je cijena veća ili jednaka od cijena_query
```

Status: 200 OK Size: 427 Bytes Time: 62 ms

| Response | Headers 6 | Cookies | Results | Docs |
|--|-----------|---------|---------|------|
| <pre> 1 [2 { 3 "_id": "674de678d0113a49fff63e85", 4 "naziv": "Napolitana", 5 "cijena": 12.5 6 }, 7 { 8 "_id": "674de678d0113a49fff63e87", 9 "naziv": "Capricciosa", 10 "cijena": 13.5 11 }, 12 { 13 "_id": "674de678d0113a49fff63e88", 14 "naziv": "Vegetariana", 15 "cijena": 14.5 16 }, 17 { 18 "_id": "674de678d0113a49fff63e89", 19 "naziv": "Šunka sir", 20 "cijena": 15.5 21 }, 22 { 23 "_id": "674de678d0113a49fff63e8a", 24 "naziv": "Quattro Stagioni", 25 "cijena": 16.5 26 }, 27 { 28 "_id": "674de678d0113a49fff63e8b", 29 "naziv": "Fantasia", 30 "cijena": 17.5 31 } 32]</pre> | | | | |

4.2 Ažuriranje svih podataka gdje je uvjet zadovoljen

Što ako želimo koristiti agregaciju podataka za ažuriranje svih dokumenata u kolekciji gdje je uvjet zadovoljen, a ne sam za njihovo dohvaćanje?

Kako bismo povećali cijenu svih pizza čija je cijena manja od `15` za `2` eura?

Mongo metoda: `collection().updateMany()`

Koristit ćemo metodu `updateMany()` koja radi na isti način kao i `updateOne()`, ali ažurira sve dokumente koji zadovoljavaju uvjet. Ova metoda prima **2 obavezna parametra**:

- `filter` - **obavezni parametar**, definira filter objekt koji predstavlja one dokumente želimo ažurirati, isto kao i kod `collection().find()` metode.
 - Npr. `{ cijena: { $lt: 15 } }`, (ekvivalentno SQL izrazu `WHERE cijena < 15`).
- `update` - **obavezni parametar**, kojim definiramo što želimo ažurirati. Ovaj parametar je JSON objekt koji sadrži ključeve koje želimo ažurirati i nove vrijednosti tih ključeva.
 - Npr. `{ $inc: { cijena: 2 } }`, (ekvivalentno SQL izrazu `SET cijena = cijena + 2`).
- `options` - **opcionalni parametar**, koji definira dodatne opcije ažuriranja.
 - Npr. `{ upsert: true }`, što znači da će se novi dokument dodati iako ne postoji dokument koji zadovoljava `filter`.

Sintaksa:

```
db.collection('naziv_kolekcije').updateMany({ filter }, { update }); // {filter}, {update}  
obavezni parametri
```

Primjerice: želimo povećati cijenu svih pizza čija je cijena manja od `15` za `2` eura:

```
let result = await pizze_collection.updateMany({ cijena: { $lt: 15 } }, { $inc: { cijena: 2 } })  
// {filter}, {update}
```

- za filtriranje smo koristili `$lt` **operator usporedbe**
- za ažuriranje smo koristili `$inc` **update operator**

Ili, recimo da želimo postaviti cijenu svih pizza čija je cijena veća od `15` na `20` eura:

```
let result = await pizze_collection.updateMany({ cijena: { $gt: 15 } }, { $set: { cijena: 20 } })  
// {filter}, {update}
```

- za filtriranje smo koristili `$gt` **operator usporedbe**
- za ažuriranje smo koristili `$set` **update operator**

Prema tome, definirat ćemo endpoint koji će povećati cijenu svih pizza čija je cijena manja od `15` za `2` eura. Kako ažuriramo djelomične podatke (samo tamo gdje je uvjet zadovoljen, a ne cijeli dokument), koristit ćemo `PATCH` metodu.

```

app.patch('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');

  try {
    let result = await pizze_collection.updateMany({ cijena: { $lt: 15 } }, { $inc: { cijena: 2 } });
    // povećaj cijenu svih pizza čija je cijena manja od 15 za 2 eura
    res.status(200).json({ modifiedCount: result.modifiedCount });
  } catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
  }
});

```

4.3 Sortiranje podataka

Sortiranje podataka možemo obaviti koristeći `collection().sort()` metodu. Ova metoda očekuje **1 obavezni parametar**:

- `sort` - **obavezni parametar**, definira po kojem kriteriju želimo sortirati podatke. Npr. `{ cijena: 1 }`, (sortira po cijeni od najmanje prema najvećoj). Ako želimo sortirati od najveće prema najmanje, koristimo `{ cijena: -1 }`.

Primjer korištenja `sort()` metode:

```

let pizze = await pizze_collection.find().sort({ cijena: 1 }).toArray(); // sortira pizze po
cijeni od najmanje prema najvećoj

```

Možemo navesti i više polja po kojima želimo sortirati:

Primjer, želimo sortirati po nazivu pizze od A do Z, a zatim po cijeni od najveće prema najmanjoj:

```

let pizze = await pizze_collection.find().sort({ naziv: 1, cijena: -1 }).toArray(); // 
sortira pizze po nazivu od A do Z, a zatim po cijeni od najveće prema najmanjoj

```

4.3.1 GET /pizze?sort

Dodat ćemo ova dva uvjeta kao `query` parametre u URL-u:

- `cijena` - za sortiranje po cijeni
- `naziv` - za sortiranje po nazivu

Vrijednosti parametra mogu biti upravo `1` ili `-1`.

```

app.get('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze');
  let cijena_query = req.query.cijena;
  let naziv_query = req.query.naziv;

  try {
    let pizze = await pizze_collection
      .find()

```

```
        .sort({ cijena: Number(cijena_query), naziv: Number(naziv_query) })
        .toArray(); // sortira pizze po cijeni i nazivu
    res.status(200).json(pizze);
} catch (error) {
    console.log(error.errorResponse);
    res.status(400).json({ error: error.errorResponse });
}
});
```

Primjer sortiranja po nazivu od A do Z, a zatim po cijeni od najveće prema najmanjoj:

GET <http://localhost:3000/pizze/?naziv=1&cijena=-1>

Send

Query Headers 2 Auth Body Tests Pre Run

Query Parameters

naziv

cijena

parameter

Status: 200 OK Size: 493 Bytes Time: 41 ms

Response Headers 6 Cookies Results Docs

```
1 [
2 {
3   "_id": "674de678d0113a49fff63e8b",
4   "naziv": "Fantasia",
5   "cijena": 17.5
6 },
7 {
8   "_id": "674de678d0113a49fff63e8a",
9   "naziv": "Quattro Stagioni",
10  "cijena": 16.5
11 },
12 {
13   "_id": "674de678d0113a49fff63e89",
14   "naziv": "Sunka sir",
15   "cijena": 15.5
16 },
17 {
18   "_id": "674de678d0113a49fff63e88",
19   "naziv": "Vegetariana",
20   "cijena": 14.5
21 },
22 {
23   "_id": "674de678d0113a49fff63e87",
24   "naziv": "Capricciosa",
25   "cijena": 13.5
26 },
27 {
28   "_id": "674de678d0113a49fff63e85",
29   "naziv": "Napolitana",
30   "cijena": 12.5
31 },
32 {
33   "_id": "674de678d0113a49fff63e86",
34   "naziv": "Funghi",
35   "cijena": 11.5
36 }
37 ]
```

Sortiranje po cijeni ide nakon, tako da će rezultat biti sortiran po cijeni od najveće prema najmanjoj.

4.4 Složena agregacija podataka metodom `aggregate()`

Za kraj ćemo pogledati kako možemo koristiti `aggregate()` metodu za složenije agregacije podataka, primjerice kada želimo odraditi više operacija nad podacima prije nego što ih dohvativimo.

Ova metoda dozvoljava složene operacije, poput filtriranja, sortiranja, grupiranja, računanja itd.

Sintaksa:

```
db.collection('naziv_kolekcije').aggregate([ { operacija1 }, { operacija2 }, {operacija3} ... ]);
```

- gdje operacija može biti bilo koja MongoDB operacija. Ove operacije često nazivamo i `pipeline` operacijama ili `pipeline stages`.

U MongoDB, pipeline operacije se izvršavaju redom, gdje je **izlaz jedne operacije ulaz sljedeće operacije**.

Operacije, kao i sve do sad u MongoDB, koriste JSON sintaksu.

1. `$match` - filtrira dokumente prema nekom uvjetu, kao i kod `find()` metode.

```
{
  $match: { cijena: { $lt: 15 } } // pronađi sve pizze čija je cijena manja od 15
}

{
  $match: { naziv: 'Capricciosa' } // pronađi pizzu čiji je naziv 'Capricciosa'
}

{
  $match: { cijena: { $gte: 10, $lte: 15 } } // pronađi sve pizze čija je cijena između 10 i
15
}
```

2. **\$group** - grupira dokumente **prema nekom polju** i specificiranoj funkciji agregacije (npr. `$sum`, `$avg`, `$min`, `$max`).

```
{
  $group: {
    _id: '$category', // grupiraj po kategoriji pizze (npr. 's mesom', 'vegetarijanske',
'slatke')
    broj_pizza: { $sum: 1 } // za svaku kategoriju, izračunaj broj pizza
  }
}

{
  $group: {
    _id: '$category', // grupiraj po kategoriji pizze
    prosjecna_cijena: { $avg: '$cijena' } // za svaku kategoriju, izračunaj prosječnu cijenu
pizza
  }
}
```

3. **\$sort** - sortira dokumente **prema nekom polju**.

```
{
  $sort: { cijena: 1 } // sortiraj pizze po cijeni od najmanje prema najvećoj
}

{
  $sort: { naziv: -1 } // sortiraj pizze po nazivu od Z do A
```

4. **\$limit** - ograničava broj rezultata

```
{
  $limit: 5; // ograniči rezultate na prvih 5
}
```

5. **\$skip** - preskače određeni broj rezultata

```
{  
  $skip: 5; // preskoči prvih 5 rezultata  
}
```

6. `$lookup` - spaja dokumente iz druge kolekcije koristeći *left outer join*

```
{  
  $lookup: {  
    from: 'kolekcija2', // ime druge kolekcije  
    localField: 'id', // polje iz trenutne kolekcije  
    foreignField: 'id', // polje iz druge kolekcije  
    as: 'ime_polja' // ime polja u kojem će se spremiti rezultati  
  }  
}
```

I tako dalje, ima ih jako puno. Cijeli popis možete pronaći na [sljedećoj poveznici](#).

Kako ovo koristiti u praksi?

Primjerice, ako želimo pronaći sve pizze čija je cijena manja od 15 i sortirati ih po cijeni od najmanje prema najvećoj, pišemo sljedeći `aggregate` upit:

```
let pizze = await pizze_collection.aggregate([{$match: {cijena: {$lt: 15}}}, {$sort: {cijena: 1}}]);
```

Ako želimo pronaći sve pizze čija je cijena manja od 15, sortirati ih po cijeni od najmanje prema najvećoj i ograničiti rezultate na prvih 5 :

```
let pizze = await pizze_collection.aggregate([{$match: {cijena: {$lt: 15}}}, {$sort: {cijena: 1}}, {$limit: 5}]);
```

Ako želimo pronaći sve pizze čija je cijena manja od 15, sortirati ih po cijeni od najmanje prema najvećoj, ograničiti rezultate na prvih 5, ali preskočiti prva 2 :

```
let pizze = await pizze_collection.aggregate([{$match: {cijena: {$lt: 15}}}, {$sort: {cijena: 1}}, {$skip: 2}, {$limit: 5}]);
```

5. MongoDB - TL;DR

MongoDB je dokumentno-orientirana baza podataka koja koristi JSON-like dokumente za pohranu podataka.

Implementacija Drivera za Node.js je `mongodb` paket. **Implementacija je ogromna** i ima jako puno razrađenih metoda, operatora i ostalih funkcionalnosti.

Dokumentacija: <https://www.mongodb.com/docs/>

Važno je razumjeti osnovni princip rada svih metoda u MongoDB-u, **a to je korištenje JSON strukture** za definiranje filtera, ažuriranja, sortiranja, grupiranja i ostalih operacija. U usporedbi s relacijskom bazom, gdje pišemo SQL upite, u MongoDB-u koristimo isključivo gotove metode s **JSON strukturom kao parametrima**.

U ovom tutorijalu smo pokrili osnovne CRUD operacije, agregaciju podataka, složene upite, sortiranje, grupiranje, ažuriranje i brisanje podataka, međutim ima tu još toga jako puno.

5.1 Spajanje na bazu podataka

Praktično definirati unutar vanjske datoteke, npr. `db.js`:

```
// db.js
import { MongoClient } from 'mongodb';
import { config } from 'dotenv';

config();

let mongoURI = process.env.MONGO_URI;
let db_name = process.env.MONGO_DB_NAME;

async function connectToDatabase() {
  try {
    const client = new MongoClient(mongoURI);
    await client.connect();
    console.log('Uspješno spajanje na bazu podataka');
    let db = client.db(db_name);

    return db;
  } catch (error) {
    console.error('Greška prilikom spajanja na bazu podataka', error);
    throw error;
  }
}
export { connectToDatabase };
```

Zatim možemo definirati `db` objekt unutar bilo koje datoteke, najčešće je to `index.js`:

```
// index.js

import express from 'express';
import { connectToDatabase } from './db.js';

const app = express();

let db = await connectToDatabase();

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

- Kolekciju dohvaćamo koristeći `db.collection('naziv_kolekcije')` metodu.
- Kolekciju možemo napraviti koristeći `db.createCollection('naziv_kolekcije')`
- Koristeći `db.listCollections()` možemo dohvatiti sve kolekcije u bazi podataka
- Koristeći `db.dropCollection('naziv_kolekcije')` možemo obrisati kolekciju
- Indekse možemo raditi i u kodu, koristeći `db.collection('naziv_kolekcije').createIndex({ kljuc: vrijednost })`
- Možemo dohvatiti sve indekse koristeći `db.collection.getIndexes()`
- Isto tako, možemo obrisati indeks koristeći `db.collection.dropIndex({ kljuc: vrijednost })`

5.2 CRUD operacije

- **C - Create**
 - `collection().insertOne(document)` - **dodavanje jednog dokumenta** `document` u kolekciju
 - `collection().insertMany(documents)` - **dodavanje više dokumenata** `documents` u kolekciju
- **R - Read**
 - `collection().find(filter, projection)` - **dohvaćanje svih dokumenata koji zadovoljavaju filter**, vraća `Cursor`. `projection` je opcionalni parametar koji definira koja polja želimo dohvatiti
 - `collection().findOne(filter, projection)` - **dohvaćanje prvog dokumenta koji zadovoljava filter**, vraća `Promise`. `projection` je opcionalni parametar koji definira koja polja želimo dohvatiti
 - `cursor.toArray()` - pretvaranje `cursor` objekta u polje dokumenata, vraća `Promise`
 - `aggregate([pipeline])` - **složena agregacija podataka**, gdje `pipeline` predstavlja niz operacija koje želimo izvršiti
 - `collection().countDocuments(filter)` - **brojanje dokumenata koji zadovoljavaju filter**, vraća `Promise`
- **U - Update**
 - `collection().updateOne(filter, update)` - **ažuriranje prvog dokumenta koji zadovoljava filter** s novim podacima `update`
 - `collection().updateMany(filter, update)` - **ažuriranje svih dokumenata koji zadovoljavaju filter** s novim podacima `update`
 - `collection().replaceOne(filter, replacement)` - **zamjena prvog dokumenta koji zadovoljava filter** s novim dokumentom `replacement`
- **D - Delete**
 - `collection().deleteOne(filter)` - **brisanje prvog dokumenta koji zadovoljava filter**
 - `collection().deleteMany(filter)` - **brisanje svih dokumenata koji zadovoljavaju filter**

5.3 MongoDB operatori

MongoDB sadrži implementiranu veliku količinu operatora za razne operacije, poput usporedbe, logičkih operacija, ažuriranja, grupiranja, sortiranja itd.

5.3.1 Operatori ažuriranja (eng. Update operators)

| Update operator | Sintaksa | Primjer | Objašnjenje |
|----------------------------|--|---|--|
| <code>\$set</code> | <code>{ \$set: { key: value } }</code> | <code>{ \$set: { age: 30 } }</code> | Postavlja vrijednost ključa <code>key</code> u dokumentu na vrijednost <code>value</code> . Ako ključ ne postoji, dodaje ključ. |
| <code>\$unset</code> | <code>{ \$unset: { key: "" } }</code> | <code>{ \$unset: { age: "" } }</code> | Briše vrijednost ključa <code>key</code> u dokumentu. |
| <code>\$inc</code> | <code>{ \$inc: { key: value } }</code> | <code>{ \$inc: { age: 1 } }</code> | Inkrementira vrijednost ključa <code>key</code> za definiranu vrijednost <code>value</code> . |
| <code>\$mul</code> | <code>{ \$mul: { key: value } }</code> | <code>{ \$mul: { price: 1.1 } }</code> | Množi vrijednost ključa za definiranu vrijednost. |
| <code>\$rename</code> | <code>{ \$rename: { oldKey: newKey } }</code> | <code>{ \$rename: { name: "fullName" } }</code> | Preimenuje ključ <code>oldKey</code> u ključ <code>newKey</code> . |
| <code>\$min</code> | <code>{ \$min: { key: value } }</code> | <code>{ \$min: { age: 18 } }</code> | Postavlja vrijednost ključa <code>key</code> na novu vrijednost <code>value</code> samo ako je postojeća vrijednost manja od nove. |
| <code>\$max</code> | <code>{ \$max: { key: value } }</code> | <code>{ \$max: { age: 65 } }</code> | Postavlja vrijednost ključa <code>key</code> na novu vrijednost <code>value</code> samo ako je postojeća vrijednost veća od nove. |
| <code>\$currentDate</code> | <code>{ \$currentDate: { key: type } }</code> | <code>{ \$currentDate: { lastModified: { \$type: "timestamp" } } }</code> | Postavlja vrijednost ključa <code>key</code> na trenutni datum (timestamp). |
| <code>\$push</code> | <code>{ \$push: { key: value } }</code> | <code>{ \$push: { tags: "newTag" } }</code> | Ako je ključ <code>key</code> polje, dodaje u njega vrijednost <code>value</code> . Ako polje ne postoji, dodaje ga. |
| <code>\$pop</code> | <code>{ \$pop: { key: 1 or -1 } }</code> | <code>{ \$pop: { tags: -1 } }</code> | Briše prvi (-1) ili zadnji (1) element unutar polja. |
| <code>\$pull</code> | <code>{ \$pull: { key: condition } }</code> | <code>{ \$pull: { tags: "oldTag" } }</code> | Briše sve elemente polja koji su istiniti za dani <code>condition</code> . |
| <code>\$addToSet</code> | <code>{ \$addToSet: { key: value } }</code> | <code>{ \$addToSet: { tags: "uniqueTag" } }</code> | Dodaje vrijednost <code>value</code> u polje samo ako vrijednost već ne postoji. |
| <code>\$each</code> | <code>{ \$push: { key: { \$each: values } } }</code> | <code>{ \$push: { tags: { \$each: ["tag1", "tag2"] } } }</code> | Dodaje više vrijednosti <code>values</code> u polje. Često se koristi u kombinaciji s <code>\$push</code> . |

5.3.2 Operatori usporedbe (eng. Comparison operators)

| Comparison operator | Sintaksa | Primjer | Objašnjenje |
|---------------------|---|---|--|
| <code>\$eq</code> | <code>{ key: { \$eq: value } }</code> | <code>{ age: { \$eq: 25 } }</code> | Podudara dokumente gdje je vrijednost ključa <code>key</code> jednaka vrijednosti <code>value</code> . |
| <code>\$ne</code> | <code>{ key: { \$ne: value } }</code> | <code>{ age: { \$ne: 25 } }</code> | Podudara dokumente gdje vrijednost ključa <code>key</code> nije jednaka vrijednosti <code>value</code> . |
| <code>\$gt</code> | <code>{ key: { \$gt: value } }</code> | <code>{ age: { \$gt: 25 } }</code> | Podudara dokumente gdje je vrijednost ključa <code>key</code> veća od vrijednosti <code>value</code> |
| <code>\$gte</code> | <code>{ key: { \$gte: value } }</code> | <code>{ age: { \$gte: 25 } }</code> | Podudara dokumente gdje je vrijednost ključa <code>key</code> veća ili jednaka od vrijednosti <code>value</code> . |
| <code>\$lt</code> | <code>{ key: { \$lt: value } }</code> | <code>{ age: { \$lt: 25 } }</code> | Podudara dokumente gdje je vrijednost ključa <code>key</code> manja od vrijednosti <code>value</code> |
| <code>\$lte</code> | <code>{ key: { \$lte: value } }</code> | <code>{ age: { \$lte: 25 } }</code> | Podudara dokumente gdje je vrijednost ključa <code>key</code> manja ili jednaka vrijednosti <code>value</code> . |
| <code>\$in</code> | <code>{ key: { \$in: [value1, value2] } }</code> | <code>{ age: { \$in: [25, 30, 35] } }</code> | Podudara dokumente gdje je vrijednost ključa <code>key</code> unutar danog polja s vrijednostima. |
| <code>\$nin</code> | <code>{ key: { \$nin: [value1, value2] } }</code> | <code>{ age: { \$nin: [25, 30, 35] } }</code> | Podudara dokumente gdje vrijednost ključa <code>key</code> nije unutar danog polja s vrijednostima. |

5.3.3 Logički operatori (eng. Logical operators)

| Logical operator | Sintaksa | Primjer | Objašnjenje |
|------------------|---|---|--|
| \$and | { \$and: [{ condition1 }, { condition2 }] } | { \$and: [{ age: { \$gt: 20 } }, { age: { \$lt: 30 } }] } | Spaja više uvjeta, samo dokumenti koji su istiniti za sve uvjete će bit vraćeni. |
| \$or | { \$or: [{ condition1 }, { condition2 }] } | { \$or: [{ age: { \$lt: 20 } }, { age: { \$gt: 30 } }] } | Spaja više uvjeta, dokumenti koji su istiniti za barem jedan uvjet će bit vraćeni. |
| \$not | { key: { \$not: { condition } } } | { age: { \$not: { \$gte: 30 } } } | Negira uvjet, vraća samo one dokumente za koje uvjet ne vrijedi. |
| \$nor | { \$nor: [{ condition1 }, { condition2 }] } | { \$nor: [{ age: { \$lt: 20 } }, { age: { \$gt: 30 } }] } | Spaja više uvjeta, vraća samo one dokumente gdje koji ne zadovoljavaju niti jedan. |
| \$exists | { key: { \$exists: boolean } } | { age: { \$exists: true } } | Podudara dokumente gdje specificirani ključ <code>key</code> postoji odnosno ne postoji <code>boolean</code> |
| \$type | { key: { \$type: value } } | { age: { \$type: "int" } } | Podudara dokumente gdje je specificirani ključ <code>key</code> određenog tipa podataka <code>value</code> |

Samostalni zadatak za Vježbu 5

Nadogradnja pizzerija aplikacije (1 bod)

Prvi dio samostalnog zadatka odnosi se na nadogradnju postojeće aplikacije. Potrebno je definirati jednostavan VUE.js 3 frontend nalik onom iz skripte WA3. Početna stranica mora prikazivati sve dostupne pizze, uključujući sliku, naziv, cijenu i sastojke.

Dovoljno je da Vue aplikacija sadrži samo 1 endpoint - `/pizze`. Inspiracija za dizajn može se pronaći na [Tivoli pizzeria webu](#).

Dizajn možete implementirati u CSS frameworku po izboru (Bootstrap, Tailwind, Bulma, itd.).

Preuzmite Express poslužitelj koji smo definirali na [WA5 – prvi dio](#) i nadogradite ga na sljedeće načine:

- Implementirajte GET `/pizze` endpoint koji će vraćati sve dostupne pizze iz MongoDB baze podataka
- U MongoDB, nadogradite kolekciju `pizze` s novim poljima: `slika` i `sastojci`. U sastojke spremite polje sastojaka (stringova) koje pizza sadrži.
- Implementirajte POST `/pizze` endpoint koji će dodavati nove pizze u kolekciju
- Implementirajte validaciju podataka koje korisnik šalje na `/pizze` za prethodni endpoint. Morate provjeriti jesu li sadržani svi i točno navedeni ključevi. Provjerite je li cijena broj i svaki sastojak u sastojcima string.
- Koristeći `Axios` paket, pozovite GET `/pizze` prilikom učitavanja stranice i prikažite podatke grafički.

Dodavanje naručivanja (1 bod)

Drugi dio samostalnog zadatka odnosi se na dodavanje mogućnosti naručivanja pizza. Potrebno je definirati novu kolekciju `pizza_narudzbe` u MongoDB bazi podataka. Kolekcija mora sadržavati sljedeće ključeve:

- `ime` - ime osobe koja naručuje
- `adresa` - adresa dostave
- `telefon` - broj telefona
- `pizza_stavke` - polje stavki narudžbe (naručene pizze):
Svaka stavka mora sadržavati sljedeće ključeve:
 - `naziv` - naziv pizze koja se naručuje
 - `kolicina` - količina naručene pizze (može i `float`, npr. `0.5`)
 - `velicina` - naručena veličina pizze (`'mala'`, `'srednja'`, `'velika'`)
- `ukupna_cijena` - ukupna cijena narudžbe (računa se na poslužitelju, **ne šalje klijent**)

Implementirajte sljedeće funkcionalnosti:

- Implementirajte POST `/narudzba` endpoint koji će dodavati nove narudžbe u kolekciju `pizza_narudzbe`
- Implementirajte validaciju podataka koje korisnik šalje na `/narudzba` za prethodni endpoint. Morate provjeriti jesu li sadržani i točno navedeni svi ključevi. Provjerite je li telefon broj ili string koji se sastoji samo od brojeva i je li svaka stavka u polju stavki ispravno definirana (naziv, količina, veličina).
- Na poslužitelju izračunajte vrijednost ključa `ukupna_cijena` na temelju naručenih pizza. Cijenu pizze dobivate dohvaćanjem određene pizze u kolekciji `pizze`

- Nadogradite Vue aplikaciju na način da ćete na dnu stranice dodati button `naruči pizze` gdje ćete poslati zahtjev na endpoint `/narudzba` s podacima o narudžbi. Ako korisnik pošalje neispravne podatke, vratite odgovarajuću grešku i statusni kod na poslužitelju i prikažite lijepo grafički korisniku tu informaciju na frontendu.
-

Web aplikacije (WA)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



Fakultet informatike u Puli

(6) Middleware funkcije

#6

WA

Middleware funkcije predstavljaju komponente koje posreduju između dolaznog HTTP zahtjeva i odaziva poslužitelja. Validacija podataka dolaznih zahtjeva i autorizacija zahtjeva predstavljaju dvije od najčešćih primjena *middleware* funkcija. Validacijom podataka osiguravamo da su podaci koje korisnik šalje ispravni, odnosno da zadovoljavaju određene kriterije njihovim sadržajem, strukturom, duljinom ili tipom podataka. Kroz skriptu ćemo osim validacije na razini rute i aplikacijskoj razini, proći i kroz biblioteku `express-validator` koja olakšava validaciju podataka dolaznih zahtjeva primjenom gotovih *middleware* funkcija.

Posljednje ažurirano: 7.1.2025.

Sadržaj

- [Web aplikacije \(WA\)](#)
- [\(6\) Middleware funkcije](#)
 - [Sadržaj](#)
- [1. Što su middleware funkcije?](#)
 - [1.1 Middleware na razini definicije rute](#)
 - [1.2 Strukturiranje programa u više datoteka](#)
 - [1.3 Middleware na aplikacijskoj razini](#)
- [2. express-validator biblioteka](#)
 - [2.1 Učitavanje modula](#)
 - [2.2 Obrada validacijskih grešaka](#)
 - [2.3 Kombiniranje vlastitih middlewarea s express-validator](#)
 - [2.4 Validacijski lanac](#)
 - [2.4.1 Validacija emaila](#)
 - [2.4.2 Provjera minimalne/maksimalne duljine lozinke](#)
 - [2.4.3 Provjera sadržaja](#)

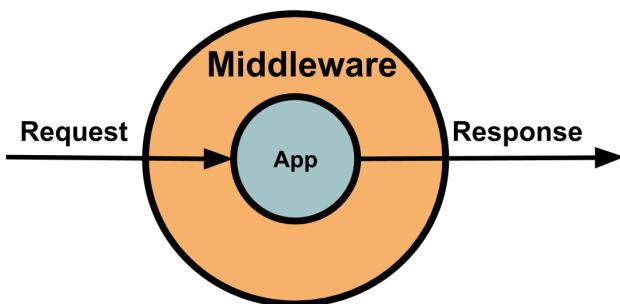
- [2.4.4 Min/Max vrijednosti](#)
- [2.4.5 Provjera je li vrijednost Boolean](#)
- [2.4.6 Provjera specifičnih vrijednosti](#)
- [2.4.7 Složena provjera lozinke regularnim izrazom](#)
- [2.4.8 Grananje lanca provjere](#)
- [2.4.9 Obrada polja u tijelu zahtjeva](#)
- [2.5 Često korišteni validatori](#)
- [2.6 Sanitizacija podataka](#)
- [2.7 Sprječavanje reflektiranog XSS napada](#)
- [Samostalni zadatak za Vježbu 6](#)

1. Što su *middleware* funkcije?

Middleware funkcije (eng. *Middleware functions*) su funkcije koje se izvršavaju u različitim fazama obrade HTTP zahtjeva, tj. *request-response* ciklusa. U Express.js razvojnom okruženju, u pravilu se koriste u trenutku kad HTTP zahtjev stigne na poslužitelj, a prije konkretne obrade zahtjeva (eng. *route handler*) definirane u implementaciji rute odnosno endpointa. Međutim, mogu se koristiti i na aplikacijskoj razini (eng. *Application level middleware*) ili na razini rutera (eng. *Router level middleware*).

Middleware funkcije se koriste za:

- izvođenje koda koji se ponavlja u više različitih ruta
- izvođenje koda prije ili nakon obrade zahtjeva
- validaciju podataka dolaznih zahtjeva
- autorizaciju zahtjeva
- logiranje zahtjeva
- obradu grešaka itd.



1.1 *Middleware* na razini definicije rute

Najčešći oblik korištenja middleware funkcija je na razini definicije rute. U tom slučaju, middleware funkcija se definira kao argument metode `app.METHOD()`:

Osnovna sintaksa:

```
app.METHOD(path, [middleware], callback);
```

odnosno:

```
app.METHOD(path, [middleware_1, middleware_2, ..., middleware_n], callback);
```

gdje su:

- `app` - instanca Express aplikacije
- `METHOD` - HTTP metoda
- `path` - putanja na koju se odnosi ruta

- `middleware` - middleware funkcija ili niz od N middleware funkcija
- `callback` - funkcija koja se izvršava kad se zahtjev "poklopi" s definiranom rutom

Middleware funkcije navodimo u **uglatim zagradama nakon putanje**.

Ako koristimo više middleware funkcija, **svaka od njih se izvršava redom**, a navodimo ih kao niz elemenata, identično kao elemente u polju.

Primjer definicije rute s middleware funkcijom:

```
app.get('/korisnici', middleware_fn, (req, res) => {
  // Obrada zahtjeva
});
```

- `middleware_fn` - middleware funkcija koja se izvršava prije obrade zahtjeva

Middleware funkcije imaju minimalno 3 parametra, i to:

- `req` - **objekt** dolaznog HTTP zahtjeva
- `res` - **objekt** HTTP odgovora koji se šalje korisniku
- `next` - **funkcija** koja se poziva kako bi se prešlo na sljedeću middleware funkciju ili na obradu zahtjeva tj. *route handler*

Dakle, middleware funkcije imaju pristup *request* (`req`) i *response* (`res`) objektima, jednako kao i *route handler* funkcija tj. callback funkcija rute.

Osnovna sintaksa middleware funkcije s 3 parametra:

```
const middleware_fn = (req, res, next) => {
  // Izvođenje koda
  next(); // pozivanjem funkcije next() prelazimo na sljedeću middleware funkciju ili na
  obradu zahtjeva
};
```

odnosno:

```
function middleware_fn(req, res, next) {
  // Izvođenje koda
  next(); // pozivanjem funkcije next() prelazimo na sljedeću middleware funkciju ili na
  obradu zahtjeva
}
```

Primjer: Definirat ćemo jednostavni Express poslužitelj koji obrađuje zahtjeve na putanji `GET /korisnici`.

- Korisnike ćemo definirati kao niz *in-memory* objekata s ključevima `id`, `ime` i `prezime`.

```
import express from 'express';

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

let PORT = 3000;
```

```

app.listen(PORT, error => {
  if (error) {
    console.error(`Greška prilikom pokretanja poslužitelja: ${error.message}`);
  } else {
    console.log(`Poslužitelj dela na http://localhost:${PORT}`);
  }
});

```

Dodajemo rute za dohvati svih korisnika (GET /korisnici) i pojedinog korisnika (GET /korisnici/:id):

```

let korisnici = [
  { id: 983498354, ime: 'Ana', prezime: 'Anić', email: 'aanic@gmail.com' },
  { id: 983498355, ime: 'Ivan', prezime: 'Ivić', email: 'iivic@gmail.com' },
  { id: 983498356, ime: 'Sanja', prezime: 'Sanjić', email: 'ssanjic123@gmail.com' }
];

// dohvati svih korisnika
app.get('/korisnici', async (req, res) => {
  if (korisnici) {
    return res.status(200).json(korisnici);
  }
  return res.status(404).json({ message: 'Nema korisnika' });
});

// dohvati pojedinog korisnika
app.get('/korisnici/:id', async (req, res) => {
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {
    return res.status(200).json(korisnik);
  }
  return res.status(404).json({ message: 'Korisnik nije pronađen' });
});

```

U redu, do sad nismo koristili *middleware* funkcije niti imamo potrebu za njima u kodu iznad.

- Međutim, što ako želimo dodati još jednu rutu koja će ažurirati email adresu pojedinog korisnika (PATCH /korisnici/:id)?

```

// ažuriranje email adrese pojedinog korisnika
app.patch('/korisnici/:id', async (req, res) => {
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {
    korisnik.email = req.body.email;
    console.log(korisnici);
    return res.status(200).json(korisnik);
  }
  return res.status(404).json({ message: 'Korisnik nije pronađen' });
});

```

Primjerice: želimo ažurirati email Sanje Sanjić na "saaaanja123@gmail.com". Kako bismo to učinili, koristimo HTTP `PATCH` metodu i šaljemo sljedeći zahtjev:

```
PATCH http://localhost:3000/korisnici/983498356
Content-Type: application/json

{
  "email": "saaaanja123@gmail.com"
}
```

Lagano možemo uočavati potrebu za korištenjem *middleware* funkcija na razini definicije rute. Potreba se javlja prilikom validacije **tijela dolaznog HTTP zahtjeva**, odnosno želimo provjeriti je li korisnik poslao ispravnu JSON strukturu (objekt) s ključem `email` te je li vrijednost ključa `email` tipa string, a naposljeku i je li email adresa ispravna.

Do sad smo isto provjeravali u samoj callback funkciji rute, recimo na sljedeći način:

```
app.patch('/korisnici/:id', async (req, res) => {
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {
    // postoji li ključ email i je li tipa string
    if (req.body.email && typeof req.body.email === 'string') {
      // trebali bi dodati još provjera za ispravnost strukture email adrese
      korisnik.email = req.body.email;
      console.log(korisnici);
      return res.status(200).json(korisnik);
    }
    return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
  }
  return res.status(404).json({ message: 'Korisnik nije pronađen' });
});
```

Što ako još želimo provjeriti ispravnost email adrese?

- Praktično bi bilo to implementirati u vanjskoj funkciji, koristiti neku biblioteku ili regularni izraz.
- U svakom slučaju, to je posao koji može obaviti *middleware* funkcija budući da **kod postaje sve složeniji** s **previše `if` grananja**.

Idemo vidjeti kako bismo ove provjere implementirali u *middleware* funkciji. Znamo da one imaju pristup `request` (`req`) i `response` (`res`) objektima.

Middleware funkciju možemo nazvati `validacijaEmaila`:

```
// middleware funkcija
const validacijaEmaila = (req, res, next) => {
  //implementacija
  next();
};
```

Jednostavno preslikamo istu provjeru od ranije:

```
// middleware funkcija
const validacijaEmaila = (req, res, next) => {
  if (req.body.email && typeof req.body.email === 'string') {
    // ako postoji ključ email i tipa je string
    next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
  }
  // u suprotnom?
};
```

- U suprotnom, tj. ako uvjet nije zadovoljen, želimo poslati korisniku odgovor s statusom 400 i porukom "Neispravna struktura tijela zahtjeva"

```
// middleware funkcija
const validacijaEmaila = (req, res, next) => {
  if (req.body.email && typeof req.body.email === 'string') {
    // ako postoji ključ email i tipa je string
    next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
  }
  // u suprotnom
  return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
};
```

Jednom kad smo definirali *middleware* funkciju, dodajemo ju kao **drugi argument** metode `app.patch()`, a prethodnu provjeru uklanjamo iz callback funkcije rute:

- ako ruta ima samo jedan *middleware*, možemo i izostaviti uglate zagrade [...]

```
// dodajemo validacijaEmaila kao drugi argument
app.patch('/korisnici/:id', [validacijaEmaila], async (req, res) => {
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {
    korisnik.email = req.body.email;
    console.log(korisnici);
    return res.status(200).json(korisnik);
  }
  return res.status(404).json({ message: 'Korisnik nije pronađen' });
});
```

Važno! Middleware `validacijaEmaila` će se izvršiti prije obrade zahtjeva u callback funkciji rute. Ako uvjeti nisu zadovoljeni, *middleware* će poslati odgovor korisniku sa statusom 400 i porukom "Neispravna struktura tijela zahtjeva", dok se callback funkcija nikada neće izvršiti

Druga velika prednost korištenja *middleware* funkcija je **ponovna upotrebljivost koda** (eng. *reusability*).

- Naime, često je slučaj da više ruta zahtjeva iste provjere, i to istim redoslijedom.
- U tom slučaju, umjesto da kopiramo isti kod u svaku rutu, možemo ga jednostavno izdvojiti u zasebnu *middleware* funkciju i koristiti ju u svakoj ruti koja zahtjeva tu provjeru.

Sada imamo sljedeće rute:

- GET /korisnici - dohvati svih korisnika

- `GET /korisnici/:id` - dohvati pojedinog korisnika
- `PATCH /korisnici/:id` - ažuriranje email adrese pojedinog korisnika

Ako pogledamo implementacije, vidimo da u svakoj ruti koristimo `parseInt(req.params.id)` kako bismo dobili brojčanu vrijednost `id` parametra rute te zatim pretražujemo korisnika po tom `id`-u.

Ovo je odličan primjer gdje možemo koristiti *middleware* funkciju!

Nazvat ćemo ju `pretragaKorisnika`

Prve dvije linije *middlewarea* `pretragaKorisnika` su identične kao i u metodama `GET /korisnici/:id` i `PATCH /korisnici/:id`:

```
const pretragaKorisnika = (req, res, next) => {
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
};
```

Ako korisnik postoji, želimo nastaviti s izvođenjem sljedeće *middleware* funkcije ili s obradom zahtjeva u callbacku, u suprotnom želimo poslati korisniku odgovor s statusom `404` i porukom "Korisnik nije pronađen"

```
const pretragaKorisnika = (req, res, next) => {
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {
    next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
  }
  return res.status(404).json({ message: 'Korisnik nije pronađen' });
};
```

Dodatno, kako su `req` i `res` objekti globalni na razini definicije rute, možemo jednostavno dodati svojstvo `korisnik` u `req` objekt kako bismo ga mogli koristiti u svim drugim *middlewareima* ili u callback funkciji rute



```
const pretragaKorisnika = (req, res, next) => {
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {
    req.korisnik = korisnik; // dodajemo svojstvo korisnik na req objekt
    next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
  }
  return res.status(404).json({ message: 'Korisnik nije pronađen' });
};
```

Sada možemo refaktorirati rute `GET /korisnici/:id` i `PATCH /korisnici/:id`. Prvo ćemo rutu `GET /korisnici/:id`

Pogledajmo trenutačnu implementaciju:

```

app.get('/korisnici/:id', async (req, res) => {
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {
    return res.status(200).json(korisnik);
  }
  return res.status(404).json({ message: 'Korisnik nije pronađen' });
});

```

Vidimo da možemo izbaciti gotovo sve! Ostaje nam samo slanje `korisnik` objekta sa statusom `200`.

Dodajemo *middleware* `pretragaKorisnika`:

```

app.get('/korisnici/:id', [pretragaKorisnika], async (req, res) => {
  // implementacija
});

```

Čitaj:

- "Prije obrade zahtjeva, izvrši *middleware* `pretragaKorisnika`.
- Ako *middleware* prođe (tj. vrati `next()`), nastavi s obradom zahtjeva odnosno izvrši callback funkciju rute."

Dakle, samo vraćamo korisnika koji se sad nalazi u `req.korisnik`:

```

app.get('/korisnici/:id', [pretragaKorisnika], async (req, res) => {
  return res.status(200).json(req.korisnik);
});

```

To je to! Idemo refaktorirati i rutu `PATCH /korisnici/:id`.

Pogledajmo trenutačnu implementaciju:

```

app.patch('/korisnici/:id', [validacijaEmaila], async (req, res) => {
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {
    korisnik.email = req.body.email;
    console.log(korisnici);
    return res.status(200).json(korisnik);
  }
  return res.status(404).json({ message: 'Korisnik nije pronađen' });
});

```

Ruta već sadrži *middleware* `validacijaEmaila`. Međutim, mi moramo **prvo provjeriti ispravnost `id`-a, odnosno provjeriti postojanje korisnika.**

- To ćemo jednostavno učiniti dodavanjem *middleware-a* `pretragaKorisnika` prije `validacijaEmaila`:

```
// dodajemo middleware pretragaKorisnika prije validacijaEmaila
app.patch('/korisnici/:id', [pretragaKorisnika, validacijaEmaila], async (req, res) => {
  // implementacija
});
```

Sada možemo izbaciti sve provjere iz callback funkcije rute:

```
app.patch('/korisnici/:id', [pretragaKorisnika, validacijaEmaila], async (req, res) => {
  req.korisnik.email = req.body.email; // ostavljamo samo promjenu emaila
  console.log(korisnici); // možemo pustiti i ispis strukture
  return res.status(200).json(req.korisnik); // vraćamo korisnika
});
```

To je to! 😊 Uočite koliko *middleware* funkcije čine kod čitljivijim!

Međutim, prije nego nastavimo, uočite sljedeće:

- slanjem zahtjeva na `GET /korisnici/:id` dobivamo korisnika s određenim `:id`-em, što je OK ali dobivamo i sljedeću grešku u konzoli:

```
Error [ERR_HTTP_HEADERS_SENT]: Cannot set headers after they are sent to the client
  at ServerResponse.setHeader (node:_http_outgoing:699:11)
  at ServerResponse.header (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/response.js:794:10)
  at ServerResponse.send (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/response.js:174:12)
  at ServerResponse.json (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/response.js:278:15)
  at pretragaKorisnika (file:///Users/lukablaskovic/Github/FIPU-WA/WA6%20-
%20Middleware%20funkcije/app/index.js:39:26)
  at Layer.handle [as handle_request] (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/layer.js:95:5)
  at next (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/route.js:149:13)
  at Route.dispatch (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/route.js:119:3)
  at Layer.handle [as handle_request] (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/layer.js:95:5)
  at /Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/index.js:284:15
```

- slanjem zahtjeva na `PATCH /korisnici/:id` odradit ćemo izmjenu email adrese, ali dobivamo dvaput istu grešku u konzoli:

```
Error [ERR_HTTP_HEADERS_SENT]: Cannot set headers after they are sent to the client
  at ServerResponse.setHeader (node:_http_outgoing:699:11)
  at ServerResponse.header (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/response.js:794:10)
  at ServerResponse.send (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/response.js:174:12)
  at ServerResponse.json (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/response.js:278:15)
```

```

    at validacijaEmaila (file:///Users/lukablaskovic/Github/FIPU-WA/WA6%20-
%20Middleware%20funkcije/app/index.js:28:26)
    at Layer.handle [as handle_request] (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/layer.js:95:5)
    at next (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/route.js:149:13)
    at pretragaKorisnika (file:///Users/lukablaskovic/Github/FIPU-WA/WA6%20-
%20Middleware%20funkcije/app/index.js:37:5)
    at Layer.handle [as handle_request] (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/layer.js:95:5)
    at next (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/route.js:149:13)
    at pretragaKorisnika (file:///Users/lukablaskovic/Github/FIPU-WA/WA6%20-
%20Middleware%20funkcije/app/index.js:37:5)
    at Layer.handle [as handle_request] (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/layer.js:95:5)
    at next (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/route.js:149:13)
Error [ERR_HTTP_HEADERS_SENT]: Cannot set headers after they are sent to the client
    at ServerResponse.setHeader (node:_http_outgoing:699:11)
    at ServerResponse.header (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/response.js:794:10)
    at ServerResponse.send (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/response.js:174:12)
    at ServerResponse.json (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/response.js:278:15)
    at pretragaKorisnika (file:///Users/lukablaskovic/Github/FIPU-WA/WA6%20-
%20Middleware%20funkcije/app/index.js:39:26)
    at Layer.handle [as handle_request] (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/layer.js:95:5)
    at next (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/route.js:149:13)
    at Route.dispatch (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/route.js:119:3)
    at Layer.handle [as handle_request] (/Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/layer.js:95:5)
    at /Users/lukablaskovic/Github/FIPU-WA/WA6 - Middleware
funkcije/app/node_modules/express/lib/router/index.js:284:15

```

Zašto dobivamo ove greške? 🤔

► Spoiler alert! Odgovor na pitanje

Dodat ćemo ispise na početku svake *middleware* funkcije kako bismo pratili redoslijed njihova izvršavanja:

```

const validacijaEmaila = (req, res, next) => {
  console.log('Middleware: validacijaEmaila');
  if (req.body.email && typeof req.body.email === 'string') {
    next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
  }
  return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
};

const pretragaKorisnika = (req, res, next) => {
  console.log('Middleware: pretragaKorisnika');
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {

```

```

    req.korisnik = korisnik;
    next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
}
return res.status(404).json({ message: 'Korisnik nije pronađen' });
};

```

Kako bismo sigurno prekinuli izvršavanje trenutne *middleware* funkcije, dodajemo `return` ispred `next()`:

```

const validacijaEmaila = (req, res, next) => {
  console.log('Middleware: validacijaEmaila');
  if (req.body.email && typeof req.body.email === 'string') {
    return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
  }
  return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
};

const pretragaKorisnika = (req, res, next) => {
  console.log('Middleware: pretragaKorisnika');
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {
    req.korisnik = korisnik;
    return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
  }
  return res.status(404).json({ message: 'Korisnik nije pronađen' });
};

```

- ili koristimo `else` uvjetni izraz kada šaljemo statusni kod `4xx`:

```

const validacijaEmaila = (req, res, next) => {
  console.log('Middleware: validacijaEmaila');
  if (req.body.email && typeof req.body.email === 'string') {
    return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
  } else {
    return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
  }
};

const pretragaKorisnika = (req, res, next) => {
  console.log('Middleware: pretragaKorisnika');
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {
    req.korisnik = korisnik;
    return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
  } else {
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
  }
};

```

1.2 Strukturiranje programa u više datoteke

Rekli smo da je jedna od prednosti korištenja *middleware* funkcija ponovna upotrebljivost koda. Međutim, vidite da već sad `index.js` datoteka postaje nečitljiva zbog miješanja definicija ruta i *middleware* funkcija. Uobičajena praksa je odvojiti *middleware* funkcije u zasebne datoteke, jednako kao što smo radili i za definicije rute koristeći `express.Router()`.

Napraviti ćemo dva nova direktorija, jedan za rute i jedan za *middleware* funkcije:

```
mkdir routes  
mkdir middleware
```

Obzirom da u pravilu želimo koristiti istu skupinu ruta s istim *middleware* funkcijama, možemo jednako nazvati datoteke u direktoriju `routes` i `middleware`: nazvat ćemo ih `korisnici.js`.

Naša struktura poslužitelja sada izgleda ovako:

```
 .  
  +-- middleware  
  |    +-- korisnici.js  
  +-- routes  
  |    +-- korisnici.js  
  +-- index.js  
  +-- node_modules  
  +-- package.json  
  +-- package-lock.json
```

Prvo ćemo jednostavno prebaciti definicije *middleware* funkcija iz `index.js` u `middleware/korisnici.js`:

```
// middleware/korisnici.js

const validacijaEmaila = (req, res, next) => {
  console.log('Middleware: validacijaEmaila');
  if (req.body.email && typeof req.body.email === 'string') {
    return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
  } else {
    return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
  }
};

const pretragaKorisnika = (req, res, next) => {
  console.log('Middleware: pretragaKorisnika');
  const id_route_param = parseInt(req.params.id);
  const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
  if (korisnik) {
    req.korisnik = korisnik;
    return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
  } else {
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
  }
};
```

```

};

// izvoz middleware funkcija
export { validacijaEmaila, pretragaKorisnika };

```

Moramo još prebaciti i naše podatke:

```

// middleware/korisnici.js

let korisnici = [
  { id: 983498354, ime: 'Ana', prezime: 'Anić', email: 'aanic@gmail.com' },
  { id: 983498355, ime: 'Ivan', prezime: 'Ivić', email: 'iivic@gmail.com' },
  { id: 983498356, ime: 'Sanja', prezime: 'Sanjić', email: 'ssanjic123@gmail.com' }
];

```

Prebacujemo definicije ruta iz `index.js` u `routes/korisnici.js`:

```

// routes/korisnici.js

import express from 'express';
// uključujemo middleware funkcije iz middleware/korisnici.js
import { validacijaEmaila, pretragaKorisnika } from '../middleware/korisnici.js';

const router = express.Router();

let korisnici = [
  { id: 983498354, ime: 'Ana', prezime: 'Anić', email: 'aanic@gmail.com' },
  { id: 983498355, ime: 'Ivan', prezime: 'Ivić', email: 'iivic@gmail.com' },
  { id: 983498356, ime: 'Sanja', prezime: 'Sanjić', email: 'ssanjic123@gmail.com' }
];

router.get('/', async (req, res) => {
  if (korisnici) {
    return res.status(200).json(korisnici);
  }
  return res.status(404).json({ message: 'Nema korisnika' });
});

router.get('/:id', [pretragaKorisnika], async (req, res) => {
  return res.status(200).json(req.korisnik);
});

router.patch('/:id', [pretragaKorisnika, validacijaEmaila], async (req, res) => {
  req.korisnik.email = req.body.email;
  console.log(korisnici);
  return res.status(200).json(req.korisnik);
});

export default router;

```

U `index.js` datoteci uključujemo router i dodajemo odgovarajući prefiks:

```
// index.js
```

```
import express from 'express';
import korisniciRouter from './routes/korisnici.js';

const app = express();
app.use(express.json()); // ova naredba obavezno ide prije dodavanja routera
app.use('/korisnici', korisniciRouter);

let PORT = 3000;

app.listen(PORT, error => {
  if (error) {
    console.error(`Greška prilikom pokretanja poslužitelja: ${error.message}`);
  } else {
    console.log(`Poslužitelj dela na http://localhost:${PORT}`);
  }
});
```

1.3 Middleware na aplikacijskoj razini

Pokazali smo kako definirati *middleware* funkcije na razini definicije rute, unutar metoda `app.METHOD(URL, [middleware_1, middleware_2, ... middleware_N], callback)`.

Međutim, **middleware funkcije možemo definirati i na razini aplikacije**, tj. na razini objekta `app`.

`app` objekt konvencionalno se koristi za konfiguraciju ukupne Express.js aplikacije, a instancira se pozivanjem funkcije `express() → const app = express();`.

Primjerice, ako želimo da se neka *middleware* funkcija izvrši prije svake rute, neovisno je li to `GET /korisnici`, `GET /korisnici/:id` ili `PATCH /korisnici/:id` ili pak `GET /pizze` itd, možemo ju na razini aplikacijskog objekta (*eng. Application-level middleware*).

Primjer: Možemo definirati *middleware* `timestamp` koja će ispisati u konzolu **trenutni datum i vrijeme** svaki put kad se zaprimi zahtjev na poslužitelju:

```
// index.js

const timer = (req, res, next) => {
  console.log(`Trenutno vrijeme: ${new Date().toLocaleString()}`);
  next();
};

// koristimo timer middleware na aplikacijskoj razini
app.use(timer);
```

Međutim, uključivanje ovog *middlewarea* moramo definirati **prije** uključivanja routera, **inače će se odnositi samo na rute koje slijede nakon njega**:

```
// index.js

import express from 'express';
import korisniciRouter from './routes/korisnici.js';

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

const timer = (req, res, next) => {
  console.log(`Trenutno vrijeme: ${new Date().toLocaleString()}`);
  next();
};

// koristimo timer middleware na aplikacijskoj razini
app.use(timer);

app.use('/korisnici', korisniciRouter);

let PORT = 3000;

app.listen(PORT, error => {
  if (error) {
```

```
    console.error(`Greška prilikom pokretanja poslužitelja: ${error.message}`);
} else {
  console.log(`Poslužitelj dela na http://localhost:${PORT}`);
}
});
```

Pogledajte malo bolje kod. Uočavate li još negdje *middleware* koji smo do sad uvijek koristili? 😊

► Spoiler alert! Odgovor na pitanje

Pokušajte poslati zahtjev na bilo koju rutu i uočite ispis trenutnog vremena u konzoli.

Dobra praksa, pogotovo u produkcijskom okruženju, jest definirati *middleware* na razini aplikacije koji ispisuje *logove* o svakom zahtjevu koji stigne na poslužitelj. Ovo je korisno za praćenje i analizu ponašanja poslužitelja, kao i za *debugging*.

Primjerice, želimo ispisati trenutni datum, vrijeme, metodu HTTP zahtjeva i URL zahtjeva:

```
[1/6/2025, 12:30:40 PM] : GET /korisnici
```

Vrijeme znamo izračunati, HTTP metoda se nalazi u `req.method`, a URL zahtjeva u `req.originalUrl`.

Rješenje:

```
// index.js

const requestLogger = (req, res, next) => {
  const date = new Date().toLocaleString();
  const method = req.method;
  const url = req.originalUrl;
  console.log(`[${date}] : ${method} ${url}`);
  next();
};

app.use(requestLogger);
```

Testirajmo slanjem zahtjeva na `GET http://localhost:3000/korisnici/983498356`

Rezultat:

```
Poslužitelj dela na http://localhost:3000
[1/6/2025, 12:33:31 PM] : GET /korisnici/983498356
Middleware: pretragaKorisnika
```

ili slanjem zahtjeva na: `PATCH http://localhost:3000/korisnici/983498356` s tijelom zahtjeva:

```
{
  "email": "sanja.sanjić@gmail.com"
}
```

Rezultat:

```
[1/6/2025, 12:34:49 PM] : PATCH /korisnici/983498356
Middleware: pretragaKorisnika
Middleware: validacijaEmaila
```

Osim pozivanja *middlewarea* na **aplikacijskog razini na svim rutama**, možemo ga pozvati i na definiranoj ruti za sve HTTP metode.

- *Primjerice*: ako imamo skupinu ruta URL-a `/admin`. Želimo u terminalu naglasiti da je pristigao zahtjev na `/admin` rutu, neovisno o metodi HTTP zahtjeva.

 Koristimo funkciju `app.all()` odnosno `router.all()`:

```
// index.js

const adminLogger = (req, res, next) => {
  console.log('Oprez! Pristigao zahtjev na /admin rutu');
  // u pravilu ovdje moramo provjeriti autorizacijski token, što ćemo vidjeti kasnije
  next();
};

app.all('/admin', adminLogger); // na svim /admin rutama pozovi adminLogger middleware
// odnosno
router.all('/admin', adminLogger);
```

Primjerice: Ako pošaljemo zahtjev na `GET http://localhost:3000/admin`, u konzoli ćemo dobiti ispis:

```
Poslužitelj dela na http://localhost:3000
Oprez! Pristigao zahtjev na /admin rutu
[1/6/2025, 12:50:04 PM] : PATCH /admin
```

Kada definiramo middleware na razini aplikacije, ponekad želimo uključiti i 4. neobavezni parametar (`err`) kako bismo mogli uhvatiti greške koje se dogode u *middleware* funkciji. Ovaj parametar se koristi za hvatanje grešaka koje se dogode u *middleware* funkciji.

Primjer:

```
// index.js

const errorHandler = (err, req, res, next) => {
  console.log(err);
  res.status(500).json({ message: 'Greška na poslužitelju' });
};

app.use(errorHandler);
```

Kada će se izvršiti ovaj *middleware*? 😊

► Spoiler alert! Odgovor na pitanje

Možemo uvijek provjeriti simulacijom greške u nekoj ruti:

```
// index.js

app.get('/error', (req, res) => {
  throw new Error('Simulirana greška na poslužitelju');
});
```

Middleware funkcije na razini rutera (*eng. Router level middleware*) definiramo na **identičan način kao i na razini aplikacije/rute**, samo što ih dodajemo kao drugi argument metode `router.METHOD()`, gdje je `router` instanca `express.Router()`.

2. express-validator biblioteka

[express-validator](#) biblioteka nudi **skup gotovih middleware funkcija za validaciju podataka** u zahtjevima. Biblioteka zahtjeva Node.js 14+ verziju i Express.js 4.17.1+ verziju.

`express-validator` biblioteka kroz svoje *middleware* funkcije nudi dvije vrste provjera:

1. **Validacija** (eng. Validation): **provjera ispravnosti podataka** u zahtjevu
2. **Sanitizacija** (eng. Sanitization): **čišćenje podataka u zahtjevu** u sigurno stanje

Instalirajmo biblioteku:

```
npm install express-validator
```

2.1 Učitavanje modula

Učitajmo modul `express-validator`:

```
// index.js

import { body, validationResult, query, param } from 'express-validator';
```

- `body()` - funkcija koja definira **provjere za tijelo zahtjeva**
- `validationResult(req)` - funkcija koja **izračunava rezultate provjera zahtjeva**
- `query()` - funkcija koja definira **provjere za query parametre**
- `param()` - funkcija koja definira **provjere za route parametre**
- `check()` - funkcija koja definira **provjere za bilo koji dio zahtjeva**

Primjerice: definirat ćemo super jednostavni endpoint `GET /hello` koji očekuje query parametar `ime`:

```
app.get('/hello', (req, res) => {
  res.send('Hello, ' + req.query.ime);
});
```

Ako pošaljemo zahtjev bez query parametra `name`, dobit ćemo odgovor `"Hello, undefined"`.

Validator dodajemo na isti način kao i prethodno manualno definirane *middleware* funkcije, a to je kao drugi argument metode `app.METHOD()`.

- to je zato što su validatori ustvari predefinirane *middleware* funkcije

Koristimo `query` funkciju za provjeru query parametra `ime`:

Sintaksa:

```
query('key');
```

U našem slučaju je to:

```
query('ime');
```

Validator za provjeru da li vrijednost nije prazna `notEmpty()`.

Jednostavno vežemo na rezultat funkcije `query()`:

```
query('ime').notEmpty();
```

To je to! Sad ga još samo dodajemo u našu rutu:

```
//index.js

app.get('/hello', [query('ime').notEmpty()], (req, res) => {
  res.send('Hello, ' + req.query.ime);
});
```

Ako pokušate ponovno poslati zahtjev bez, i dalje ćete dobiti odgovor `"Hello, undefined"`.

Razlog tomu je što `express-validator` ne izvještava automatski klijenta o greškama. Dodavanjem dodatnih validatora, moramo ručno definirati strukturu JSON odgovora u slučaju greške.

2.2 Obrada validacijskih grešaka

Kako bismo dobili rezultate provjere, koristimo funkciju `validationResult(req)` koja prima `req` objekt i **vraća rezultate provjere u slučaju da dode do greške**.

```
const errors = validationResult(req); // sprema greške svih validacija koje su provele
// middleware funkcije, ako ih ima!
```

Dodajemo u našu rutu i ispisom provjeravamo sadržaj:

```
//index.js

app.get('/hello', [query('ime').notEmpty()], (req, res) => {
  const errors = validationResult(req); // spremanje grešaka
  console.log(errors);
  res.send('Hello, ' + req.query.ime);
});
```

Ako nema grešaka, npr. ako pošaljemo zahtjev: `GET http://localhost:3000/hello?ime=Ana`, dobivamo sljedeći ispis:

```
Result { formatter: [Function: formatter], errors: [] }
```

Ako pošaljemo zahtjev bez query parametra, npr. `GET http://localhost:3000/hello`, dobivamo detaljan ispis s detaljima o pogrešci:

```

Result {
  formatter: [Function: formatter],
  errors: [
    {
      type: 'field',
      value: '',
      msg: 'Invalid value',
      path: 'ime',
      location: 'query'
    }
  ]
}

```

Kako čitamo ispis? "Greška je nastala u `query` parametru naziva `ime`, jer je njegova vrijednost `value` prazna."

 Funkcijom `isEmpty()` možemo **provjeriti je li vrijednost prazna**.

Ako greške ne postoje (tj. `errors.isEmpty() == true`), šaljemo odgovor `OK` klijentu, u suprotnom šaljemo odgovor s detaljima o grešci koji je dostupan u `errors.array()` uz status `Bad Request`.

```

// index.js

app.get('/hello', [query('ime').notEmpty()], (req, res) => {
  const errors = validationResult(req);
  // ako nema greške
  if (errors.isEmpty()) {
    return res.send('Hello, ' + req.query.ime);
  }
  return res.status(400).json({ errors: errors.array() });
});

```

2.3 Kombiniranje vlastitih *middlewarea* s *express-validator*

Moguće je kombinirati vlastite *middleware* funkcije s *express-validator* validatorima.

- Primjerice, ako želimo provjeriti da li je korisnik s određenim `id`-om pronađen, a zatim provjeriti da li je email ispravan, možemo iskoristiti `pretragaKorisnika` *middleware* koji se nalazi u `middleware/korisnici.js`, a ostatak provjere odraditi kroz *express-validator* biblioteku.

Primjer: Nadogradit ćemo rutu `PATCH /korisnici:id` tako da provjerava ispravnost email adrese.

Prvi korak je izbaciti postojeći vlastiti middleware za provjeru email adrese:

```
// routes/korisnici.js

// uklanjamo validacijaEmaila middleware
router.patch('/:id', [pretragaKorisnika], async (req, res) => {
  req.korisnik.email = req.body.email;
  console.log(korisnici);
  return res.status(200).json(req.korisnik);
});
```

Želimo provjeriti sljedeće:

- da li je ključ `email` proslijeđen u **tijelu zahtjeva**, dakle koristimo `body('email')` a ne `query('email')`
 - da li je vrijednost ključa `email` ispravno strukturirana
- Funkcijom `isEmail()` možemo brzo provjeriti je li vrijednost email adrese ispravna.
- dodajemo provjeru kao drugi *middleware* u nizu, nakon `pretragaKorisnika`

```
// routes/korisnici.js

router.patch('/:id', [pretragaKorisnika, body('email').isEmail()], async (req, res) => {
  req.korisnik.email = req.body.email;
  console.log(korisnici);
  return res.status(200).json(req.korisnik);
});
```

Na kraju još dodajemo obradu grešaka te vraćamo klijentu odgovarajuće JSON odgovore:

```
// routes/korisnici.js

router.patch('/:id', [pretragaKorisnika, body('email').isEmail()], async (req, res) => {
  const errors = validationResult(req);
  // ako nema greške
  if (errors.isEmpty()) {
    req.korisnik.email = req.body.email;
    console.log(korisnici);
    return res.status(200).json(req.korisnik);
  }
  return res.status(400).json({ errors: errors.array() });
});
```

Primjerice: ako pokušamo proslijediti neispravnu email adresu, npr. `PATCH http://localhost:3000/korisnici/983498356` s tijelom zahtjeva:

```
{
  "email": "sssssanja123@gmail.com"
}
```

Dobivamo natrag JSON odgovor s detaljima o grešci:

```
{
  "errors": [
    {
      "type": "field",
      "value": "sssssanja123@gmail.com",
      "msg": "Invalid value",
      "path": "email",
      "location": "body"
    }
  ]
}
```

Ako pokušamo definirati pogrešan ključ u tijelu zahtjeva, npr. `PATCH http://localhost:3000/korisnici/983498356` s tijelom zahtjeva:

```
{
  "email123": "sssssanja123@gmail.com"
}
```

Dobivamo odgovarajuću grešku i za to:

```
{  
  "errors": [  
    {  
      "type": "field",  
      "msg": "Invalid value",  
      "path": "email",  
      "location": "body"  
    }  
  ]  
}
```

Pa i ako pošaljemo prazno tijelo zahtjeva, dobit ćemo grešku u tijelu odgovora.

2.4 Validacijski lanac

U `express-validator` biblioteci ima **mnoštvo validatora**, a nudi i mogućnost **kombiniranja više validatora u jedan lanac provjere** (eng. *Validation Chain*), koji se izvršava redom, definiranjem [lanca metoda](#).

- bez obzira što postoji lanac provjera, ovdje se radi o jednoj *middleware* funkciji

Primjerice: želimo osim ispravnosti emaila provjeriti i sadrži li email nastavak `@unipu.hr`.

Isto možemo postići kombinacijom validatora `isEmail()` i `contains()`

```
// routes/korisnici.js

router.patch('/:id', [pretragaKorisnika, body('email').isEmail().contains('@unipu.hr')],  
async (req, res) => {  
  const errors = validationResult(req);  
  // ako nema grešaka  
  if (errors.isEmpty()) {  
    req.korisnik.email = req.body.email; // ažuriramo email  
    console.log(korisnici);  
    return res.status(200).json(req.korisnik);  
  }  
  return res.status(400).json({ errors: errors.array() });  
});
```

Na svaki validator možemo dodati i poruku koja će se prikazati u slučaju greške:

Poruku definiramo metodom `withMessage()`:

```
body('email').isEmail().withMessage('Email adresa nije  
ispravna').contains('@unipu.hr').withMessage('Email adresa mora biti s @unipu.hr');
```

2.4.1 Validacija emaila

Koristimo `isEmail()` validator:

```
body('email').isEmail().withMessage('Molimo upišite ispravnu email adresu');
```

2.4.2 Provjera minimalne/maksimalne duljine lozinke

Koristimo `isLength()` validator:

Minimalnu duljinu navodimo kao argument metode, slično kao kod MongoDB upita:

```
body('password').isLength({ min: 6 }).withMessage('Lozinka mora imati minimalno 6 znakova');  
  
// ili  
  
body('password').isLength({ min: 6, max: 20 }).withMessage('Lozinka mora imati između 6 i 20  
znakova');
```

2.4.3 Provjera sadržaja

`isAlphanumeric()` validator provjerava sadrži li vrijednost samo slova i brojeve:

```
body('username').isAlphanumeric().withMessage('Korisničko ime mora sadržavati samo slova i brojeve');
```

`isAlpha()` validator provjerava sadrži li vrijednost samo slova:

```
body('name').isAlpha().withMessage('Ime mora sadržavati samo slova');
```

2.4.4 Min/Max vrijednosti

Koristimo `isInt()` validator za provjeru je li vrijednost tipa integer, opcionalno možemo definirati raspon kao i kod `isLength()`:

```
body('age').isInt({ min: 18, max: 99 }).withMessage('Dob mora biti između 18 i 99 godina');
```

Koristimo `isFloat()` validator za provjeru je li vrijednost tipa float:

```
body('price').isFloat({ min: 0 }).withMessage('Cijena mora biti pozitivan broj');
```

2.4.5 Provjera je li vrijednost Boolean

Koristimo `isBoolean()` validator:

```
body('active').isBoolean().withMessage('Aktivnost mora biti tipa boolean');
```

2.4.6 Provjera specifičnih vrijednosti

Koristimo `isIn()` validator za provjeru je li vrijednost sadržana u nekom nizu:

```
body('role').isIn(['admin', 'user']).withMessage('Uloga mora biti admin ili user');
```

2.4.7 Složena provjera lozinke regularnim izrazom

Koristimo `matches()` validator:

- pišemo regularni izraz koji definira pravila za lozinku
- npr. lozinka mora sadržavati barem jedno slovo i jedan broj, duljine minimalno 8 znakova

```
body('password')
.matches('/^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d]{8,}$/)
.withMessage('Lozinka mora sadržavati barem jedno slovo i jedan broj');
```

2.4.8 Grananje lanca provjere

 Možemo koristiti i `check()` funkciju koja će pretražiti parametar definiran nazivom bez obzira gdje se nalazi, bilo to:

- u **tijelu zahtjeva** (`req.body`)
- u **query** parametrima (`req.query`)
- u **route** parametrima (`req.params`)
- u **zaglavljima** (`req.headers`)
- u **kolačićima** (`req.cookies`)

Ako se naziv parametra ponavlja na više mesta, npr. parametar `password` postoji i u tijelu zahtjeva i u query parametrima (naravno nije pametno), `check()` će svejedno odraditi validaciju za sve vrijednosti.

Primjer validacijskog grananja za registraciju korisnika gdje želimo provjeriti sljedeće:

- korisnik obavezno mora unijeti ime
- korisnik obavezno mora unijeti ispravnu email adresu
- lozinka mora imati minimalno 6 znakova
- potvrda lozinke mora biti jednaka lozinki

```
const { check, validationResult } = require('express-validator');

app.post(
  '/register',
  [
    // ne navodimo lokaciju jer će check() pretražiti sve parametre
    check('name').notEmpty().withMessage('Ime je obavezno'), // zaseban middleware (1)
    check('email').isEmail().withMessage('Email je u krivom formatu'), // zaseban middleware
    (2) check('password').isLength({ min: 6 }).withMessage('Lozinka mora imati barem 6 znakova'),
    // zaseban middleware (3)
    check('confirmPassword') //zaseban middleware (4)
      .custom((value, { req }) => value === req.body.password)
      .withMessage('Lozinke se ne podudaraju!')
  ],
  (req, res) => {
    // callback funkcija
    const errors = validationResult(req);
    // >>> implementacija registracije ovdje <<<
    // ako nema pogrešaka:
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }
    res.send('Registracija uspješna!');
  }
);
```

Napomena: U primjeru iznad imamo 4 *middleware* funkcije (bez obzira što imaju ulančane metode). Ukupno je 4 *middleware* jer polje gdje se definiraju ima ukupno 4 elementa.

2.4.9 Obrada polja u tijelu zahtjeva

Što ako klijent proslijedi **polje elemenata** u tijelu zahtjeva?

- Stvari ostaju iste! `express-validator` će provjeriti svaki element polja 

Na primjer: klijent pošalje zahtjev s nekim `ID`-evima:

```
{  
  "ids": [5, 4, 11, 4, 123]  
}
```

Validacija provjerava je li svaki element polja `ids` tipa integer:

```
body('ids').isInt().withMessage('Svaki element polja mora biti tipa integer');
```

Međutim, `express-validator` će sve **dolazne podatke tretirati kao stringove**, samim time, ako proslijedimo string `"123"`, validacija će proći.

- Proslijedimo li niz `[5, 4, 11, 4, "abc"]`, validacija **neće proći**.

2.5 Često korišteni validatori

| Validator | Sintaksa | Primjer |
|-----------------------------|------------------------------|--|
| Obavezno polje | notEmpty() | check('ime').notEmpty().withMessage('Ime je obavezno') |
| Je prazno | isEmpty() | check('kljuc').isEmpty().withMessage('Kljuc mora biti prazan') |
| Ključ postoji | exists() | check('kljuc').exists().withMessage('Kljuc mora postojati') |
| Validacija emaila | isEmail() | check('email').isEmail().withMessage('Pogrešan email format') |
| Min. duljina | isLength({ min: x }) | check('password').isLength({ min: 6 }).withMessage('Mora biti minimalno 6 znakova') |
| Max. duljina | isLength({ max: x }) | check('username').isLength({ max: 12 }).withMessage('Mora biti maksimalno 12 znakova') |
| Alfanumerički znak | isAlphanumeric() | check('username').isAlphanumeric().withMessage('Samo slova i brojevi!') |
| Točna duljina | isLength({ min: x, max: x }) | check('zip').isLength({ min: 5, max: 5 }).withMessage('Mora biti točno 5 znakova') |
| Jednako | equals('vrijednost') | check('role').equals('admin').withMessage('Mora biti admin') |
| Min/Max vrijednost | toInt({ min: x, max: y }) | check('age').toInt({ min: 18, max: 65 }).withMessage('Samo vrijednosti između 18 i 65') |
| Integer check | toInt() | check('age').toInt().withMessage('Mora biti integer') |
| Decimal check | isDecimal() | check('price').isDecimal().withMessage('Mora biti decimalni broj') |
| Boolean check | isBoolean() | check('isActive').isBoolean().withMessage('Mora biti Boolean vrijednost') |
| String check | isString() | check('name').isString().withMessage('Mora biti string') |
| Inclusion | isIn(['a', 'b']) | check('role').isIn(['admin', 'user']).withMessage('Kriva uloga!') |
| Sadržavanje podskupa | contains('nešto') | check('username').contains('admin').withMessage('Mora sadržavati admin') |
| Exclusion | not().isIn(['a', 'b']) | check('username').not().isIn(['root', 'admin']) |
| Custom Regex | matches('/regex/') | check('username').matches(/^[a-zA-Z]+\$/).withMessage('Dozvoljena samo velika i mala slova') |
| Validacija URL-a | isURL() | check('website').isURL().withMessage('Pogrešan URL!') |
| Validacija kreditne kartice | isCreditCard() | check('card').isCreditCard().withMessage('Pogrešan broj kreditne kartice') |
| Validacija IBAN-a | isIBAN() | check('iban').isIBAN().withMessage('Pogrešan IBAN') |
| ISO Date | isISO8601() | check('date').isISO8601().withMessage('Netočan format datuma') |
| Custom Validator | custom(fn) | check('field').custom(value => value > 0).withMessage('Vrijednost mora biti pozitivna') |
| Poklapanje lozinke | custom((value, { req })) | check('confirm').custom((lozinka, { req }) => lozinka === req.body.prosljedena_lozinka) |
| Trim | trim() | check('username').trim().notEmpty().withMessage('Polje je obavezno!') |
| Array check | isArray() | check('roles').isArray().withMessage('Mora biti polje') |
| Object check | isObject() | check('user').isObject().withMessage('Mora biti objekt') |

Sve validatore `express-validator` biblioteke možete pronaći na [službenoj dokumentaciji](#). Naravno, nije ih potrebno sve znati napamet, već ove koji se najčešće koriste.

2.6 Sanitizacija podataka

Sanitizacija podataka (*eng. Sanitization*) je proces čišćenja podataka u zahtjevu na način da se oni dovedu u sigurno stanje.

- *Primjerice:* ako korisnik unese email adresu s velikim slovima, možemo ju pretvoriti u mala slova prije nego krenemo s validacijom

`express-validator` biblioteka nudi **niz middlewarea** koji se koriste na isti način kao i validatori.

Pretvorba email adrese u mala slova

```
body('email').normalizeEmail(all_lowercase: true);

// npr. email: 'Sanja.sanjic@gmail.com' -> 'sanja.sanjic@gmail.com'
```

Uklanjanje praznih znakova

```
body('username').trim();

// npr. ' Sanja ' -> 'Sanja'
```

Pretvorba stringa u broj

```
body('age').toInt();

// npr. '25' -> 25
```

Brisanje znakova koji nisu definirani

```
body('username').whitelist('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890');

// npr. 'Sanja123!' -> 'Sanja123'
```

Brisanje znakova koji su definirani

```
body('username').blacklist('!@#$%^&*()_+');

// npr. 'Sanja123!$$$' -> 'Sanja123'
```

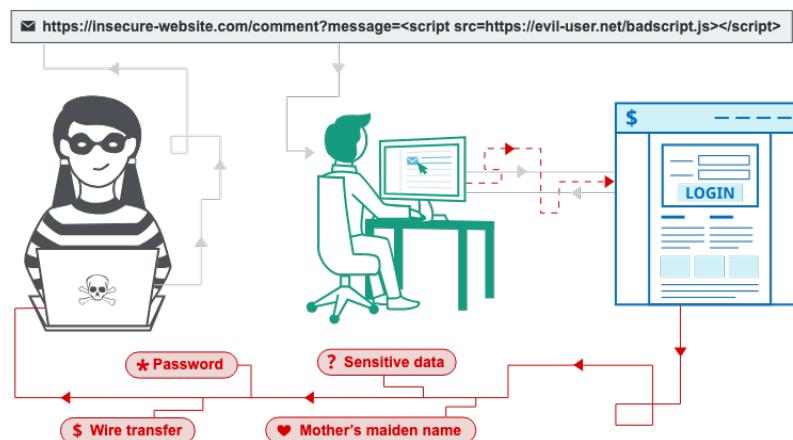
2.7 Sprječavanje reflektiranog XSS napada

XSS (*eng. Cross-Site Scripting*) napadi su vrlo česti i opasni. Postoji više kategorija XSS napada, a jedan od najčešćih je **reflektirani XSS napad** (*eng. Reflected XSS attack*).

Napad izgleda ovako:

- korisnik šalje HTTP zahtjev na poslužitelj s malicioznim JavaScript kodom, najčešće u URL-u
- maliciozni kod, najčešće obuhvaćen u HTML `<script>` tagu, izvršava se na korisničkoj strani

- u usporedbi sa **pohranjenim XSS napadom** (eng. *Stored XSS attack*), reflektirani XSS napad je **jednokratan i ne ostavlja tragove u bazi podataka niti na poslužitelju**



Uzet ćemo za primjer našu rutu `GET /hello` koja očekuje query parametar `ime`.

```
// index.js

app.get('/hello', [query('ime').notEmpty()], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  res.send('Hello, ' + req.query.ime);
});
```

Ako pošaljemo zahtjev: `GET http://localhost:3000/hello?ime=Pero`, dobit ćemo odgovor "Hello, Pero".

- Ako pošaljemo prazan zahtjev, dobit ćemo grešku jer smo to pokrili s `notEmpty()` validatorom.

Možemo nadograditi rutu tako da još sanitiziramo query parametar koristeći `trim()` middleware kako bi uklonili prazne znakove s početka i kraja stringa te možemo provjeriti je li korisnik poslao samo slova koristeći `isAlpha()` validator.

Sljedeći primjer ima samo 1 middleware, međutim možemo ih odvojiti i u zasebne middleware funkcije:

```
// 1 middleware
app.get('/hello', [query('ime').notEmpty().trim().isAlpha()], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  res.send('Hello, ' + req.query.ime);
});

// 3 middlewarea
app.get('/hello', [query('ime').notEmpty(), query('ime').trim(), query('ime').isAlpha()],
  (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
})
```

```
    res.send('Hello, ' + req.query.ime);
});
```

Možemo dodati i odgovarajuće poruke za greške:

```
// index.js

app.get('/hello', [query('ime').notEmpty().withMessage('Ime je obavezno'),
query('ime').trim(), query('ime').isAlpha().withMessage('Ime mora sadržavati samo slova')],
(req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  res.send('Hello, ' + req.query.ime);
});
```

Međutim, što da nemamo provjeru `isAlpha()` i korisnik pošalje maliciozni kod u query parametru?

- *Banalni primjer: Maliciozni korisnik pošalje skriptni tag u query parametru* koji sadrži

```
alert('Hakirani ste! Molimo da pošaljete novac na adresu...');
```

Primjer takvog HTTP zahtjeva izgledao bi ovako:

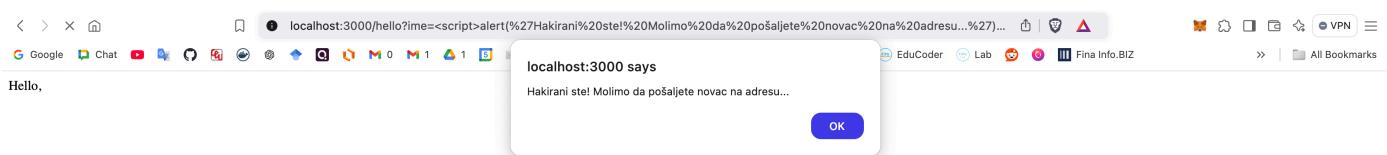
```
GET http://localhost:3000/hello?ime=<script>alert('Hakirani ste! Molimo da pošaljete novac na adresu...')</script>
```

Ako maknete `isAlpha()` validator, dobit ćete odgovor s "malicioznim kodom", odnosno **skripta će se izvršiti na korisničkoj strani**:

```
// index.js

app.get('/hello', [query('ime').notEmpty().withMessage('Ime je obavezno'),
query('ime').trim()], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  res.send('Hello, ' + req.query.ime);
});
```

Ako pošaljete GET zahtjev u web pregledniku, dobit ćete `alert` poruku.



✓ Jedan od *middlewarea* koji se može koristiti za sprječavanje reflektiranog XSS napada je `escape()` middleware.

```
query('ime').escape();
```

Ovaj *middleware* će zamijeniti HTML znakove, npr. <, >, &, ', " s njihovim ekvivalentima <, >, &, ', ".

```
// index.js

app.get('/hello', [query('ime').notEmpty().withMessage('Ime je obavezno'),
query('ime').trim(), query('ime').escape()], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  res.send('Hello, ' + req.query.ime);
});
```

Primjer odgovora (neće se izvršiti skripta i XSS napad je spriječen):

```
Hello, &lt;script&gt;alert(&#x27;Hakirani ste! Molimo da pošaljete novac na
adresu...&#x27;)&lt;&#x2F;script&gt;
```

Samostalni zadatak za Vježbu 6

Izradite novi poslužitelj `movie-server` na proizvoljnem portu te implementirajte sljedeće rute:

1. `GET /movies` - vraća listu filmova u JSON formatu
2. `GET /movies/:id` - vraća podatke o filmu s određenim `id`-om
3. `POST /movies` - dodaje novi film u listu filmova (*in-memory*)
4. `PATCH /movies/:id` - ažurira podatke o filmu s određenim `id`-om
5. `GET /actors` - vraća listu glumaca u JSON formatu
6. `GET /actors/:id` - vraća podatke o glumcu s određenim `id`-om
7. `POST /actors` - dodaje novog glumca u listu glumaca (*in-memory*)
8. `PATCH /actors/:id` - ažurira podatke o glumcu s određenim `id`-om

Podaci za filmove:

```
[  
  {  
    "id": 4222334,  
    "title": "The Shawshank Redemption",  
    "year": 1994,  
    "genre": "Drama",  
    "director": "Frank Darabont"  
  },  
  {  
    "id": 5211223,  
    "title": "The Godfather",  
    "year": 1972,  
    "genre": "Crime",  
    "director": "Francis Ford Coppola"  
  },  
  {  
    "id": 4123123,  
    "title": "The Dark Knight",  
    "year": 2008,  
    "genre": "Action",  
    "director": "Christopher Nolan"  
  }  
]
```

Podaci za glumce:

```
[  
  {  
    "id": 123,  
    "name": "Morgan Freeman",  
    "birthYear": 1937,  
    "movies": [4222334]  
  },  
  {
```

```
{
  "id": 234,
  "name": "Marlon Brando",
  "birthYear": 1924,
  "movies": [5211223]
},
{
  "id": 345,
  "name": "Al Pacino",
  "birthYear": 1940,
  "movies": [5211223]
}
]
```

Implementirajte *middleware* koji će se upotrebljavati za pretraživanje filmova i glumaca po `id`-u. Kada korisnik pošalje zahtjev na rutu koja ima route parametar `id` na resursu `/movies`, *middleware* će provjeriti postoji li taj film u listi filmova. Napravite isto i za glumce, dodatnim *middlewareom*. Odvojite rute u zasebne router instance te implementacije *middlewareova* u zasebne datoteke unutar `middleware` direktorija.

Dodajte novi *middleware* na razini Express aplikacije koji će logirati svaki dolazni zahtjev na konzolu u sljedećem formatu:

```
[movie-server] [2024-06-01 12:00:00] GET /movies
```

Za svaki zahtjev morate logirati:

- naziv aplikacije
- trenutni datum i vrijeme
- HTTP metodu zahtjeva
- URL zahtjeva

Instalirajte `express-validator` biblioteku te implementirajte sljedeće validacije za odgovarajuće rute:

- `POST /movies` - validirajte jesu li poslani `title`, `year`, `genre` i `director`
- `PATCH /movies/:id` - validirajte jesu li poslani `title`, `year`, `genre` ili `director`
- `POST /actors` - validirajte jesu li poslani `name` i `birthYear`
- `PATCH /actors/:id` - validirajte jesu li poslani `name` ili `birthYear`
- `GET /movies/:id` - validirajte je li `id` tipa integer
- `GET /actors/:id` - validirajte je li `id` tipa integer
- `GET /movies` - dodajte 2 query parametra `min_year` i `max_year` te validirajte jesu li oba tipa integer. Ako su poslani, provjerite jesu li `min_year` i `max_year` u ispravnom rasponu (npr. `min_year < max_year`). Ako je poslan samo jedan parametar, provjerite je li tipa integer.
- `GET /actors` - dodajte route parametar `name` te provjerite je li tipa string. Uklonite prazne znakove s početka i kraja stringa koristeći odgovarajući *middleware*.

Obradite greške za svaku rutu slanjem objekta s greškama koje generira `express-validator` biblioteka.

Osigurajte sve rute od reflektiranog XSS napada koristeći odgovarajući *middleware*.

Web aplikacije (WA)

Nositelj: doc. dr. sc. Nikola Tanković

Asistent: Luka Blašković, mag. inf.

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



Fakultet informatike u Puli

(7) Autentifikacija i autorizacija zahtjeva

#7

WA

Autentifikacija i autorizacija su ključni koncepti području sigurnosti informacijskih sustava, a predstavljaju srodne, ali različite procese. Autentifikacija je proces provjere identiteta korisnika ili sustava, a za cilj ima utvrditi je li osoba ili sustav onaj za koga se predstavlja. Autorizacija je proces određivanja prava pristupa korisnika ili sustava određenim resursima ili funkcionalnostima. U ovom poglavlju, naučit ćemo kako implementirati ove dva ključna procesa svake web aplikacije. Konkretno, naučit ćemo kako autentificirati korisnika na poslužitelju te poslati autorizacijski token (JSON Web Token) klijentu.

Posljednje ažurirano: 14.1.2025.

Sadržaj

- [Web aplikacije \(WA\)](#)
- [\(7\) Autentifikacija i autorizacija zahtjeva](#)
 - [Sadržaj](#)
- [1. Autentifikacija vs Autorizacija](#)
 - [1.1 Autentifikacija korisnika](#)
 - [1.2 Enkripcija vs Hashiranje](#)
 - [1.3 `bcrypt` paket](#)
 - [1.4 Registracija korisnika](#)
 - [1.5 Provjera podudaranja hash vrijednosti \(autentifikacija\)](#)
- [2. Autorizacija kroz JWT](#)
 - [2.1 Što je ustvari token?](#)
 - [2.2 Kako iskoristiti JWT token za autorizaciju?](#)
 - [2.3 Provjera valjanosti JWT tokena](#)
 - [2.4 Implementacija funkcija za generiranje i provjeru JWT tokena](#)

- [1. Korak \(Registracija korisnika\)](#)
- [2. Korak \(Prijava korisnika s klijentske strane\)](#)
- [3. Korak \(Prijava korisnika na poslužiteljskoj strani\)](#)
- [4. Korak \(Generiranje JWT tokena\)](#)
- [5. Korak \(Pohrana JWT tokena na klijentskoj strani i slanje na poslužitelj\)](#)
- [6. Korak \(Provjera valjanosti JWT tokena na poslužiteljskoj strani\)](#)
- [2.5 Autorizacijski middleware](#)
- [2.6 Rok trajanja JWT tokena](#)
- [Samostalni zadatak za Vježbu 7](#)

1. Autentifikacija vs Autorizacija

Autentifikacija (eng. *authentication*) je proces provjere identiteta korisnika ili sustava. Cilj autentifikacije je utvrditi je li osoba ili sustav onaj za koga se predstavlja.

Kako funkcioniра autentifikacija u web aplikacijama?

1. **Prikupljanje vjerodajnica (eng. credentials):** Korisnik unosi vjerodajnice (npr. korisničko ime, lozinku, biometrijske podatke, PIN, itd.) putem određenog sučelja (npr. web obrasca).
2. **Provjera vjerodajnica:** Poslužitelj provjerava unesene podatke uspoređujući ih s onima pohranjenima u bazi podataka (npr. podudaranje korisničkog imena i lozinke).
3. **Rezultat provjere:** Ako su uneseni podaci ispravni, korisnik je autentificiran, i dobiva **autorizacijski pristup**

Autorizacija (eng. *authorization*) je proces određivanja prava pristupa **autentificiranog** korisnika. Dakle, autorizacija dolazi nakon uspješne autentifikacije.

Kako funkcioniра autorizacija u web aplikacijama?

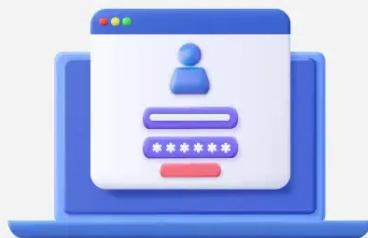
1. **Dodjela prava pristupa:** Poslužitelj upravlja pravima pristupa korisnika (npr. u obliku definiranih uloga, dozvola, itd.)
2. **Provjera prava pristupa:** Kada korisnik pokuša pristupi određenom resursu ili funkcionalnosti, poslužitelj provjerava je li taj korisnik ovlašten za taj pristup
3. **Rezultat autorizacije:** Ako je korisnik ovlašten, poslužitelj mu omogućuje pristup traženom resursu ili funkcionalnosti, inače mu vraća grešku

Sljedeća tablica ukratko objašnjava razliku između autentifikacije i autorizacije:

| Autentifikacija | Autorizacija |
|--|---|
| Provjerava tko je korisnik. | Određuje što korisnik smije raditi. |
| Izvodi se na početku (prijava). | Izvodi se svaki put kad korisnik traži pristup resursu. |
| Rezultat: "Jesi li ti stvarno ta osoba?" | Rezultat: "Smiješ li ovo raditi?" |

Working of Authentication and Authorization

Authentication



Confirms users are who they say they are.

Authorization



Gives users permission to access a resource

U prvom dijelu ove lekcije, naučit ćemo kako autentificirati korisnika na poslužitelju.

1.1 Autentifikacija korisnika

Krenimo definicijom osnovnog Express.js poslužitelja:

```
import express from 'express';
import cors from 'cors';

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

PORT = 3000;

app.get('/', (req, res) => {
  res.send('Spremni za autentifikaciju!');
});

app.listen(PORT, () => {
  console.log(`Poslužitelj dela na portu ${PORT}`);
});
```

Rekli smo da autentifikacija uključuje prikupljanje vjerodajnica korisnika. U web aplikacijama, u pravilu se koriste **korisničko ime/email i lozinka** ako se radi o tradicionalnim web aplikacijama.

Međutim, u **modernim web aplikacijama**, danas su postali uobičajeni napredniji oblici autentifikacije kao što:

- **Biometrijski podaci** (npr. otisak prsta, prepoznavanje lica, itd.)
- **Multi-faktorska autentifikacija** (kombinira više metoda autentifikacije, npr. kroz SMS, e-mail, itd.)
- **Autentifikacija bez lozinke** (npr. slanje autentifikacijskog koda na e-mail, one-time password, push notifikacije, itd.)

- **Autentifikacija bazirana na certifikatima** (npr. SSL/TLS certifikati)
- **Single Sign-On (SSO)** autentifikacija (prijava putem sigurnih poslužitelja treće strane, npr. Google, GitHub, Facebook, itd.)
- **OAuth2 autentifikacija** (autentifikacija putem OAuth2 protokola)

Većina ovih metoda autentifikacije zahtijeva dodatne biblioteke i servise, te su izvan opsega ove lekcije.

Mi ćemo naučiti kako implementirati "from-scratch" autentifikaciju korisnika putem **jednostavnih vjerodajnica**, kao što su korisničko ime i lozinka.

Definirat ćemo POST rutu putem koje će korisnik poslati svoje vjerodajnice. Do sad smo naučili da rute nazivamo prema resursima kojima pristupamo, međutim ovdje možemo uvesti iznimku jer ne pristupamo konkretnom resursu, već provodimo proces autentifikacije. Rutu možemo nazvati `/login`:

Napomena! Korisničko ime i lozinka su osjetljivi podaci te ih nikada ne želimo slati preko route odnosno query parametara, već želimo ove podatke **slati u tijelu HTTP zahtjeva!**

```
app.post('/login', (req, res) => {
  const { username, password } = req.body; // pristupamo korisničkom imenu i lozinci iz tijela zahtjeva
});
```

U grubo, ideja je sljedeća:

- Korisnik će poslati POST zahtjev na `/login` rutu s korisničkim imenom i lozinkom
- Poslužitelj će provjeriti jesu li korisničko ime i lozinka ispravni
- Ako jesu, korisnik je autentificiran, a poslužitelj će mu poslati potvrdu

Prvo ćemo pohraniti korisnike u *in-memory* listu korisnika:

```
const users = [
  { id: 1, username: 'johnDoe', password: 'password' },
  { id: 2, username: 'janeBB', password: 'password123' },
  { id: 3, username: 'admin', password: 'super_secret_password' }
];
```

Opisani endpoint bi izgledao ovako:

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username && user.password === password);

  if (user) {
    res.send('Uspješno ste autentificirani!');
  } else {
    res.status(401).send('Neuspješna autentifikacija!');
  }
});
```

Koje probleme ovdje možemo uočiti?

1. **Lozinke su pohranjene u plain textu.** Ovo je vrlo loša praksa jer ako maliciozni korisnik pristupi bazi podataka (*eng. Data leak*), može vidjeti sve lozinke korisnika. Dodatno, ako se korisnik prijavljuje preko nesigurne mreže, lozinka može biti presretnuta.
2. **Nema nikakvog mehanizma zaštite od *brute-force* napada.** Maliciozni korisnik može pokušati beskonačno puta prijaviti se s različitim kombinacijama korisničkog imena i lozinke.
3. Pohrana lozinki bez ikakvog enkripcijskog mehanizma je **neprihvatljiva** i **ilegalna** u većini zemalja, posebno u EU. Rizici su preveliki, a [kazne su visoke](#).
4. **Nema mehanizma za *session management*.** Kako će klijent znati da je autentificiran, ako poslužitelj vratí samo poruku "Uspješno ste autentificirani!"? Bolje pitanje je: **kako će se poslužitelj sjetiti da je korisnik autentificiran**, ako je svaki zahtjev novi zahtjev (HTTP je *stateless* protokol)?

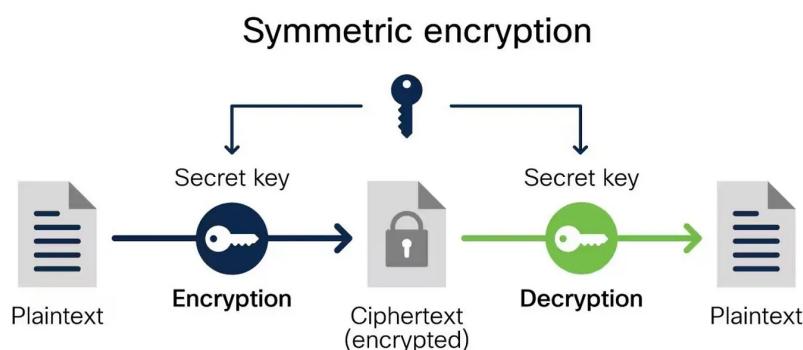
1.2 Enkripcija vs Hashiranje

Enkripcija (eng. *Encryption*) je proces pretvaranja podataka ili poruka u kodirani oblik kako bi se osigurala njihova privatnost i zaštita od neovlaštenog pristupa. Kodiranjem s originalni podaci, koje često zovem "običan" ili "čisti" tekst, pretvaraju u nečitki oblik koji nazivamo "šifrirani" tekst, odnosno *ciphertext*.

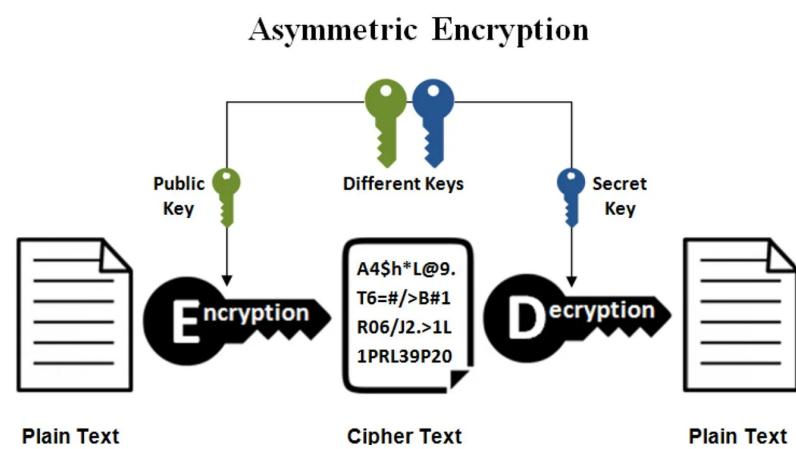
Samo osobe koje posjeduju odgovarajući **ključ** za dešifriranje mogu ponovno dobiti originalne podatke.

Postoje dvije vrste enkripcije:

1. **Simetrična enkripcija:** Koristi **isti ključ za enkripciju i dekripciju podataka**.
2. **Asimetrična enkripcija:** Koristi dva različita, ali povezana ključa: **javni ključ** za enkripciju i **privatni ključ** za dekripciju podataka.



Primjer simetrične enkripcije: koristi isti ključ za enkripciju i dekripciju podataka.

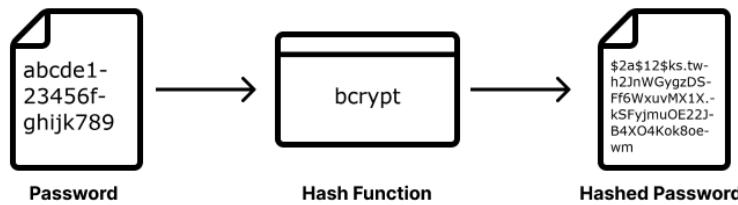


Primjer asimetrične enkripcije: koristi dva različita, ali povezana ključa: javni ključ za enkripciju i privatni ključ za dekripciju podataka.

Hashiranje (eng. *Hashing*) je proces pretvaranja ulaznih podataka u fiksni niz znakova pomoću matematičke funkcije koju nazivamo **hash funkcijom**. Hash funkcija je funkcija koja prima **ulazne podatke proizvoljne duljine** i vraća **izlazni niz fiksne duljine**.

Glavna razlika između enkripcije i hashiranja je da **hashiranje nije reverzibilno**. To znači da ne možemo dobiti originalne podatke iz hashiranog niza. Hashiranje se koristi za **sigurno pohranjivanje lozinki** uz dodatne sigurnosne mehanizme kao što su dodani *salt* (nasumični niz znakova koji se veže uz lozinku).

Password Hashing



Primjer hashiranja: ulazni podaci se pretvaraju u fiksni niz znakova pomoću hash funkcije.

Dakle, lozinka koja se jednom **hashira** ne može se **dehashirati**!

- **hash funkcije su matematički algoritmi** koji "generiraju" jedinstveni niz znakova za svaki ulazni niz, ali se iste funkcije ne mogu koristiti za "rekonstrukciju" originalnog niza
- to je zato što hash funkcije **gube određene informacije** prilikom generiranja hash vrijednosti
- bez obzira što isti ulaz uvijek daje isti izlaz (deterministička funkcija), **nije postoji inverz funkcija** za vraćanje izlaza u originalni niz

Kako bismo onda provjerili je li korisnik unio ispravnu lozinku? 🤔

► Spoiler alert! Odgovor na pitanje

Primjer:

1. Pohranjeni hash u bazi podataka: `5f4dcc3b5aa765d61d8327deb882cf99`
2. Unesena lozinka pri prijava: `lozinka123`
3. Hash unesene lozinke: `HASH(lozinka123)=5f4dcc3b5aa765d61d8327deb882cf99`
4. Usporedba hashova: `5f4dcc3b5aa765d61d8327deb882cf99 == 5f4dcc3b5aa765d61d8327deb882cf99`
(lozinke (odnosno **njihovi hashovi**) se podudaraju)

Prema tome, što ćemo koristiti za pohranu lozinki korisnika? **Enkripciju ili hashiranje?**

Odgovor je **hashiranje!** U sljedećoj tablici usporedit ćemo ove dvije tehnike i navesti prednosti i nedostatke svake.

| Značajka | Hashiranje | Enkripcija |
|----------------------------|---|---|
| Smjer | Jednosmjerno – nema povratka na izvornu lozinku | Dvosmjerno – enkriptirani podatak se može dekriptirati |
| Svrha | Provjera lozinki bez potrebe za čuvanjem plain teksta | Čuvanje podataka u enkriptiranom obliku s mogućnošću dekripcije |
| Primjena | Lozinke za prijavu – uspoređuje se hash lozinka | Pohrana podataka koji se kasnije trebaju dekriptirati, primjerice u HTTPS komunikaciji |
| Potrebni ključevi | Nije potrebno upravljati ključevima | Potrebno je upravljati enkripcijskim ključevima |
| Otpornost na napade | Otpornost na brute force uz soljenje ključa | Ranjivo na kompromitaciju ključa |
| Upravljanje | Jednostavno – samo se hash funkcija i <i>salt</i> parametri pohranjuju | Puno složenije – ključevi se moraju pravilno čuvati |
| Tipična upotreba | Pohrana lozinki za provjeru autentičnosti | Pohrana podataka koji se trebaju dekriptirati (npr. poruke), blockchain tehnologije |
| Prednosti | Jednosmjernost, jednostavnost, otpornost na napade | Omogućava povrat podataka u izvornom obliku |
| Nedostaci | Ne omogućava povrat lozinke, samo usporedbu hash vrijednosti | Ako ključ procuri, svi podaci su ugroženi |
| Primjeri algoritama | SHA-256, bcrypt, Argon2 | AES, RSA, DES |

U praksi, **hashiranje** je **sigurniji** i **jednostavniji** način pohrane lozinki korisnika.

1.3 bcrypt paket

Mi ćemo koristiti **bcrypt** algoritam za hashiranje lozinki korisnika. Bcrypt algoritam razvili su Niels Provos i David Mazières 1999. godine, a danas je jedan od popularnijih algoritama za hashiranje lozinki. Koga zanima više o bcrypt algoritmu, može pročitati članak na [Wikipediji](#).

Instalirat ćemo `bcrypt` paket pomoću npm-a:

```
npm install bcrypt
```

Uključimo `bcrypt` u našu aplikaciju:

```
import bcrypt from 'bcrypt';
```

Za hashiranje lozinke koristimo metodu `hash`:

Moguće je koristiti asinkroni i sinkroni način rada s `bcrypt` paketom. Preporučuje se korištenje asinkronog načina rada jer je sigurniji i ne blokira izvođenje poslužitelja (*non-blocking*).

Metoda `hash` prima 3 argumenta:

- `plainPassword`: lozinka koju želimo hashirati u obliku običnog teksta
- `saltRounds`: broj rundi za generiranje *salt* vrijednosti
- `callback`: funkcija koja se poziva nakon što se hashiranje završi. Callback funkcija prima 2 argumenta: `err` i `hash`. Ako se dogodi greška, `err` će biti različit od `null`, a inače će `hash` sadržavati hashiranu lozinku.

Sintaksa:

```
bcrypt.hash(plainPassword, saltRounds, (err, hash) => {});
```

Broj rundi za generiranje *salt* vrijednosti određuje koliko će se puta izvršiti hash funkcija. Veći broj rundi znači da će hashiranje trajati dulje, ali će biti sigurnije. **Preporučuje se korištenje vrijednosti između 10 i 12.**

Primjerice, ako je `saltRounds = 3`, hashiranje će izgledati ovako:

1. Generira se *salt* vrijednost (nasumični niz znakova) **tri puta** kako bi se dobila konačna *salt* vrijednost (npr.
`salt = salt1 + salt2 + salt3`)
2. Nakon što je generirana konačna *salt* vrijednost, `bcrypt` koristi tu vrijednost zajedno s unesenom lozinkom kako bi generirao hash. Tijekom ovog procesa, lozinka i *salt* vrijednost prolaze kroz određeni broj iteracija (određenih sa `saltRounds` parametrom) kako bi se proizveo sigurni hash.
3. Jednom kada je hash generiran, poziva se callback funkcija s `hash` vrijednošću.

U callbacku možemo definirati jednostavnu obradu greške:

```

let plainPassword = 'lozinka123';
let saltRounds = 10;

bcrypt.hash(plainPassword, saltRounds, (err, hash) => {
  if (err) {
    console.error(`Došlo je do greške prilikom hashiranja lozinke: ${err}`);
    return;
  } else {
    console.log(`Hashirana lozinka: ${hash}`);
  }
});

```

Ovaj kod će ispisati hashiranu lozinku u konzoli.

```
Hashirana lozinka: $2b$10$iyK8.vxPtPG8bArU9ucKjOF2tDqzEkFmaquk0yQRuNKKRG7/YBcyy
```

Slično kao kod rada s datotekama, osim callback pristupa možemo koristiti i Promise pristup s `bcrypt` paketom. Tada metoda `hash` vraća Promise objekt koji možemo raspakirati pomoću `then` i `catch` metoda, odnosno `async` i `await` sintakse.

```

bcrypt
  .hash(plainPassword, saltRounds)
  .then(hash => {
    console.log(`Hashirana lozinka: ${hash}`);
  })
  .catch(err => {
    console.error(`Došlo je do greške prilikom hashiranja lozinke: ${err}`);
  });

```

Odnosno:

```

try {
  let hash = await bcrypt.hash(plainPassword, saltRounds);
  console.log(`Hashirana lozinka: ${hash}`);
} catch (err) {
  console.error(`Došlo je do greške prilikom hashiranja lozinke: ${err}`);
}

```

Sve skupa možemo zapakirati u asinkronu funkciju `hashPassword` koja će primati 2 argumenta: `plainPassword` i `saltRounds`.

```

async function hashPassword(plainPassword, saltRounds) {
  try {
    let hash = await bcrypt.hash(plainPassword, saltRounds);
    return hash;
  } catch (err) {
    console.error(`Došlo je do greške prilikom hashiranja lozinke: ${err}`);
    return null;
  }
}

```

Ova funkcija će vratiti hashiranu lozinku ako je proces uspješan, inače će vratiti `null`.

1.4 Registracija korisnika

Sada kada znamo kako hashirati lozinke korisnika, možemo implementirati funkcionalnost registracije korisnika.

Dodati ćemo rutu POST `/register` koja će primati korisničko ime i lozinku korisnika.

- dodajemo korisnika u *in-memory* listu korisnika, ali prije toga hashiramo lozinku funkcijom `hashPassword`

```
const users = [];

app.post('/register', async (req, res) => {
  const { username, password } = req.body;

  const hashed_password = await hashPassword(password, 10);

  if (!hashed_password) {
    // ako se iz nekog razloga dogodi greška prilikom hashiranja lozinke
    res.status(500).send('Došlo je do greške prilikom hashiranja lozinka!');
    return;
  }

  const novi_korisnik = { id: users.length + 1, username, password: hashed_password };
  users.push(novi_korisnik);

  return res.status(201).json({ message: 'Korisnik uspješno registriran', user: novi_korisnik });
});
```

Pošaljite zahtjev preko HTTP klijenta, trebali biste dobiti odgovor u ovom obliku:

```
{
  "message": "Korisnik uspješno registriran",
  "user": {
    "id": 1,
    "username": "peroPeric123",
    "password": "$2b$10$kAPhPJRYnYZNVh.YmC3NwuaUjRPuwO.MQizgCP5kNdO/FrAa7ZXcu"
  }
}
```

Ovime smo završili prvi korak u autentifikaciji korisnika.

1.5 Provjera podudaranja hash vrijednosti (autentifikacija)

Recimo da se 5 korisnika registriralo u našoj aplikaciji, uključujući korisnika `peroPeric123` iz primjera iznad.

```
let users = [
  { id: 1, username: 'peroPeric123', password:
    '$2b$10$kAPhPJRYnYZNVh.YmC3NwuaUjRPuwO.MQizgCP5kNdO/FrAa7ZXcu' },
  { id: 2, username: 'maraMara', password:
    '$2b$10$fNvGAkcfgSLVqGUbMGOKOU4lu3Ub bcmKyJ0aVULyK1oYOWe5MpWie' },
  { id: 3, username: 'ivanIvanko555', password:
    '$2b$10$ZKe8aSUUEBNzQlhPigzFKOBne/4v6AzEckXZ.I7.j.TXfFQRYIt8G' },
  { id: 4, username: 'anaAnic', password:
    '$2b$10$H2HR4nlPbhRFW/5YKtIuC.b5rRsPz2EE7dYz561W44/8rxJ2RrfVW' },
  { id: 5, username: 'justStanko', password:
    '$2b$10$wXcmTomNSfS9Ivafuy6/iuant3GQgxSXSwf1ZNx9d6iwuSi/d1HMK' }
];
```

Rekli smo da je matematički nemoguće dehashirati hash vrijednost i dobiti originalnu lozinku.

Prema tome, morat ćemo svaki put ponoviti proces hashiranja korisničke lozinke i usporediti dobiveni hash s onim koji je pohranjen u bazi podataka.

Međutim, potrebno je osim hashiranja ponovnog hashiranja lozinke, **provesti točan broj rundi soljenja** ključa kako bi se dobila identična hash vrijednost. Duljina izvođenja hash funkcije ovisi o broju rundi *soljenja* ključa, o duljini lozinke, ali i o samom algoritmu koji se koristi.

Za provjeru **podudaranja hash vrijednosti sa tekstuualnom vrijednosti**, koristimo metodu `compare`:

```
bcrypt.compare(plainText, hashedValue, callback);
```

Ova metoda uspoređuje `plainText` (običan tekst) s `hashedValue` (hash vrijednost) i vraća `true` ako se podudaraju, inače vraća `false`.

Rezultat funkcije je boolean vrijednost, ovisno o podudaranju hash vrijednosti s unesenom lozinkom.

Još jedanput, razlika s enkripcijom je što se **ne može dehashirati** hash vrijednost. Odnosno, ne možemo utvrditi podudaranje ako nemamo originalnu lozinku.

Primjer:

Obzirom da se lozinke podudaraju, očekujemo ispis Lozinke se podudaraju!.

Na isti način možemo logiku pohraniti u funkciju `checkPassword` koja prima 2 argumenta: `plainPassword` i `hashedPassword`.

```
async function checkPassword(plainPassword, hashedPassword) {
  try {
    let result = await bcrypt.compare(plainPassword, hashedPassword);
    return result;
  } catch (err) {
    console.error(`Došlo je do greške prilikom usporedbe hash vrijednosti: ${err}`);
    return false;
  }
}
```

Funkciju `checkPassword` ćemo pozvati u ruti za autentifikaciju koju smo definirali na početku:

- prvo pronalazimo korisnika u listi korisnika

```
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username);

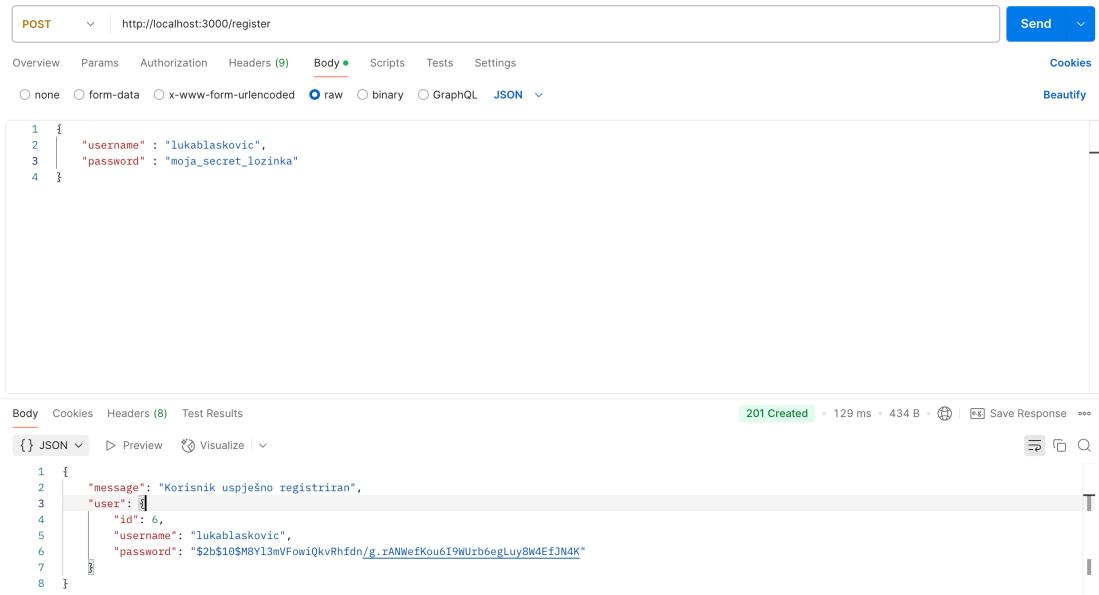
  if (!user) {
    return res.status(404).send('Ne postoji korisnik!');
  }

  const lozinkaIspravna = await checkPassword(password, user.password);

  if (lozinkaIspravna) {
    return res.send('Uspješno ste autentificirani!');
  } else {
    return res.status(401).send('Neuspješna autentifikacija!'); // 401 - Unauthorized
  }
});
```

To je to! 🚀 Testirat ćemo rute:

1. Registrirajte korisnika preko `/register` rute
2. Pokušajte autentificirati korisnika koji ne postoji preko `/login` rute
3. Autentificirajte korisnika preko `/login` rute pogrešnom lozinkom
4. Autentificirajte korisnika preko `/login` rute ispravnom lozinkom



POST http://localhost:3000/register

Body (JSON)

```

1 {
2   "username": "lukablasovic",
3   "password": "moja_secret_lozinka"
4 }

```

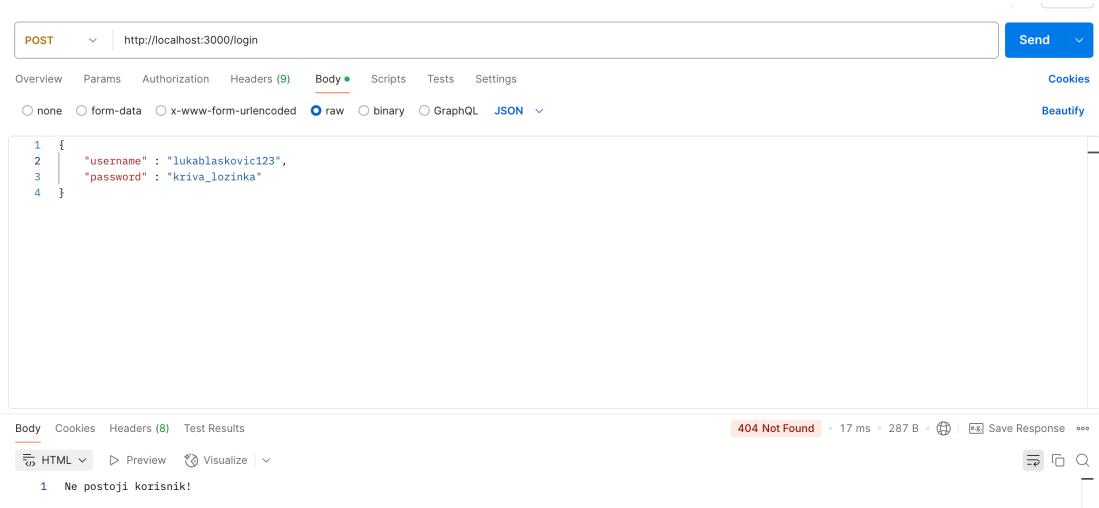
201 Created

```

1 {
2   "message": "Korisnik uspješno registriran",
3   "user": [
4     {
5       "id": 6,
6       "username": "lukablasovic",
7       "password": "$2b$10$MBY13mVfowiQkvRhfdn/g.rANWeFkou6I9WUrb6egLuy8N4EfJN4K"
8     }
9 ]

```

Registracija korisnika slanjem zahtjeva na `POST /register` u Postmanu s korisničkim imenom i lozinkom



POST http://localhost:3000/login

Body (JSON)

```

1 {
2   "username": "lukablasovic123",
3   "password": "kriva_lozinka"
4 }

```

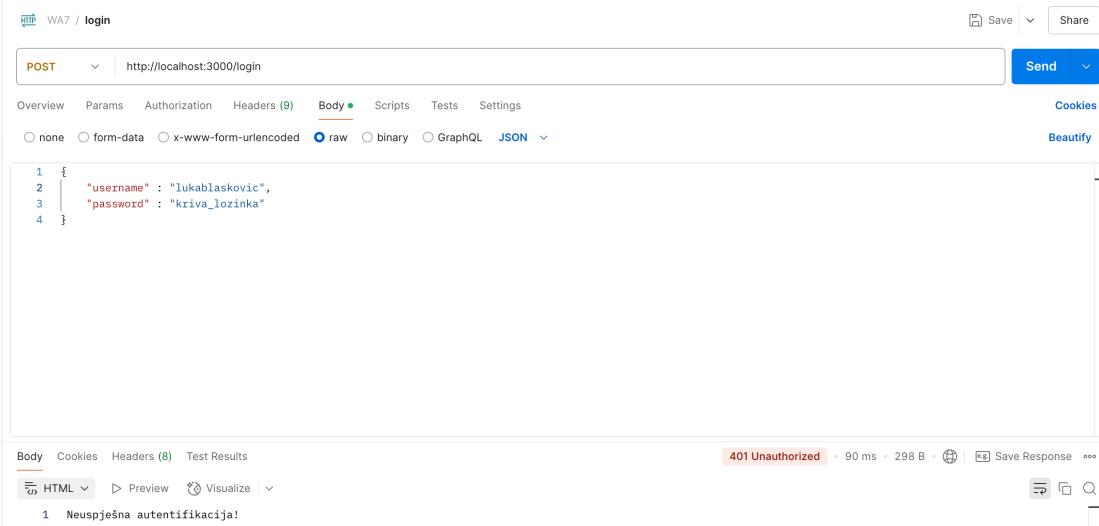
404 Not Found

```

1 Ne postoji korisnik!

```

Pokušaj autentifikacije nepostojećeg korisnika slanjem zahtjeva na `POST /login` u Postmanu



http://localhost:3000/login

Body (JSON)

```

1 {
2   "username": "lukablasovic",
3   "password": "kriva_lozinka"
4 }

```

401 Unauthorized

```

1 Neuspješna autentifikacija!

```

Pokušaj autentifikacije postojećeg korisnika s pogrešnom lozinkom slanjem zahtjeva na `POST /login` u Postmanu

The screenshot shows the Postman interface. At the top, it says "POST" and "http://localhost:3000/login". Below that, under "Body", there are tabs for "none", "form-data", "x-www-form-urlencoded", "raw", "binary", "GraphQL", and "JSON". The "JSON" tab is selected, and the body content is:

```

1  {
2   "username" : "lukablasovic",
3   "password" : "moja_secret_lozinka"
4 }

```

At the bottom of the interface, it shows a "200 OK" status with a response time of 85 ms and a size of 290 B. The response body contains the message "Uspješno ste autentificirani!" followed by a small yellow star icon.

Uspješna autentifikacija korisnika slanjem zahtjeva na `POST /login` u Postmanu

Uspješno smo implementirali autentifikaciju korisnika putem hashiranja lozinki! 🎉

Važno je napomenuti da je ovo samo osnovna implementacija autentifikacije korisnika. U praksi, autentifikacija korisnika može biti puno složenija i uključivati dodatne sigurnosne mehanizme koje smo naveli ranije.

Dodatno, važno je nadodati da iz sigurnosnih razloga nije dobra praksa slati detaljnu poruku o grešci korisniku u slučaju neuspješne autentifikacije. Umjesto toga, preporučuje se slanje **generičke poruke o grešci** kako bi se spriječilo otkrivanje informacije što je pogrešno (korisničko ime, email ili lozinka) i zašto je došlo do greške.

Iz tog razloga, sljedeći kod:

```

app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username);

  if (!user) {
    return res.status(404).send('Ne postoji korisnik!');
  }

  const lozinkaIspravna = await checkPassword(password, user.password);

  if (lozinkaIspravna) {
    return res.send('Uspješno ste autentificirani!');
  } else {
    return res.status(401).send('Neuspješna autentifikacija!'); // 401 - Unauthorized
  }
});

```

- bolje je napisati ovako:

```

app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username);

```

```
if (!user) {
    return res.status(401).send('Neuspješna autentifikacija!'); // 401 - Unauthorized (ne
otkrivamo da korisnik ne postoji)
}

const lozinkaIspravna = await checkPassword(password, user.password);

if (lozinkaIspravna) {
    return res.send('Uspješno ste autentificirani!');
} else {
    return res.status(401).send('Neuspješna autentifikacija!'); // 401 - Unauthorized (ne
otkrivamo da je lozinka pogrešna)
}
});
```

Vidimo da u drugom primjeru, **vraćamo istu poruku o grešci** bez obzira u kojem dijelu procesa autentifikacije je došlo do greške!

2. Autorizacija kroz JWT

Rekli smo da je **autorizacija** proces davanja prava korisniku da pristupi određenim resursima ili funkcionalnostima

U kontekstu web aplikacija, autorizaciju ćemo provoditi kroz **JWT** (*eng. JSON Web Token*).

JSON Web Token (JWT) je kompaktan, siguran i samodostatni način razmjene informacija između dviju strana u obliku JSON objekata. Koristi se prvenstveno za autorizaciju u modernim aplikacijama.

JWT je službeno definiran kao standard 2015. godine kada je objavljen u RCF 7519 dokumentu (<https://tools.ietf.org/html/rfc7519>), međutim u razvoju je od ranih 2010-ih godina.

Prije uvođenja JWT-a kao autorizacijskog standarda, koristili smo **session-based** autentifikaciju koja se bazirala na tzv. kolačićima (*eng. cookies*).

Sustavi temeljeni na sesijama funkcioniraju otprilike ovako:

- server je odgovoran za pohranu sesije korisnika u internoj memoriji ili bazi podataka
- klijent je dobivao kolačić (*eng. cookie*) s jedinstvenim identifikatorom sesije
- svaki put kada korisnik pristupi resursu, kolačić se šalje na poslužitelj
- poslužitelj provjerava je li sesija valjana i korisnik ima pristup resursu

Glavni nedostatak ovakvog pristupa je potreba za **pohranom sesije na poslužitelju**. Ovo može biti problematično u **distribuiranim sustavima** gdje je potrebno održavati **stanje sesije** između više poslužitelja. Samim time, ovakvo rješenje je **teško skalirati**.

JWT kao alternativa koristi **token-based** autentifikaciju. Ovaj pristup je **distribuiran i samodostatan**.

To znači da se **svi podaci potrebni za autorizaciju zahtjeva nalaze u samom tokenu!** 

2.1 Što je ustvari token?

JWT token je ništa drugo nego specijalni niz znakova koji se sastoji od 3 dijela:

1. **Header**: JSON objekt koji sadrži informacije o **tipu tokena i korištenom algoritmu za enkripciju**
2. **Payload**: JSON objekt koji sadrži **korisničke podatke** (npr. korisničko ime, email) koje želimo "pohraniti u token", ali i **dodatne informacije** (npr. rok trajanja tokena)
3. **Signature**: enkriptirani dio tokena (**kriptografski potpis**) koji se koristi za **provjeru integriteta podataka**. Ovaj dio se generira na temelju **(1) headera i (2) payloada i enkripcijskog ključa**.

Otvorite jwt.io web stranicu. Ovdje možete pronaći koristan za vizualizaciju i dekodiranje JWT tokena.



Debugger Libraries Introduction Ask
Crafted by  Auth0 by Okta

| Encoded | Decoded |
|--|---|
| <pre>eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c</pre> | <p>HEADER:</p> <pre>{ "alg": "HS256", "typ": "JWT" }</pre> <p>PAYOUT:</p> <pre>{ "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }</pre> <p>VERIFY SIGNATURE</p> <pre>HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) □ secret base64 encoded</pre> |

 Signature Verified
SHARE JWT

Primjer JWT tokena s tri dijela: header, payload i signature

Uočite spomenuta 3 dijela tokena, svaki je označen različitom bojom, a desno ima svoj dekodirani oblik.

1. **Header:** označen **crvenom** bojom (u ovom slučaju, koristi se **HS256** algoritam za enkripciju, i tip tokena je **JWT**)
2. **Payload:** označen **rozom** bojom (sadrži korisničke podatke (**sub**, **name**) i **iat** - **issued at** - vrijeme izdavanja tokena)
3. **Signature:** označen **plavom** bojom (enkriptirani dio tokena koji se sastoji od **headera**, **payloada** i **enkripcijskog ključa**)

Signature kao enkripcijski algoritam koristi **HMACSHA256** (*Hash-based Message Authentication Code using SHA-256*), koji ustvari kombinira **SHA-256** hash funkciju s **HMAC** algoritmom i sigurnim ključem za enkripciju.

U donjem lijevom kutu možemo vidjeti oznaku **Signature Verified** što znači da je token valjan i da je **integritet podataka sačuvan**.

Svaki dio JWT tokena odvojen je točkom (**.**). Ovo je važno jer nam omogućava da token dekodiramo i provjerimo njegovu valjanost.

Struktura JWT tokena:

```
header.payload.signature
```

Primjerice:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

Sljedeća tablica prikazuje kodirani i dekodirani dio JWT tokena za svaki od tri dijela:

| Dio JWT tokena | Kodirani dio (eng. Encoded) | Dekodirani dio (eng. Decoded) |
|----------------|--|---|
| Header | eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 | {"alg": "HS256", "typ": "JWT"} |
| Payload | eyJzdWIoiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpvag4gRG9lIiwiWF0IjoxNTE2MjM5MDIyfQ | {"sub": "1234567890", "name": "John Doe", "iat": 1516239022} |
| Signature | SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c | HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) |

Ako pokušamo promijeniti bilo koji dio tokena, **signature** će se promijeniti i token više neće biti valjan i dobit ćemo grešku "Invalid Signature".

Dodatno, vidimo da se prikaz dekodiranog **payloada** također promijenio.

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIo
iIxMjM0NTY3ODkwIiwibmFtZ
SI6Ikpvag4gRG9lIiwiWF0
IjoxNTE2MjM5MDIyfQ.SflK
xwRJSMeKKF2QT4fwpMeJf36
POk6yJV_adQssw5c
```

Warning: Looks like your JWT payload is not a valid JSON object. JWT payloads must be top level JSON objects as per <https://tools.ietf.org/html/rfc7519#section-7.2>

Decoded

```
HEADER:
```

| | |
|---|-----------------|
| { | "alg": "HS256", |
| | "typ": "JWT" |
| } | |

```
PAYOUT:
```

| | |
|---|----------------------|
| " | |
| { | "sub": "1234567890", |
| | "name": "John Doe", |
| | "iat": 1516239022 |
| } | |
| | " |
| | " |

```
VERIFY SIGNATURE
```

| | |
|--------------------------------|--|
| HMACSHA256(| |
| base64UrlEncode(header) + ".", | |
| base64UrlEncode(payload), | |
| your-256-bit-secret | |
|) | <input type="checkbox"/> secret base64 encoded |

SHARE JWT

⊗ Invalid Signature

Promijenili smo samo jedan znak u **payload** dijelu tokena, što je rezultiralo promjenom **signature** dijela tokena i greškom "Invalid Signature".

Možemo slobodno promijeniti vrijednosti unutar **payloada**, stavit ćemo recimo `username` i `email` korisnika, a izbrisat ćemo `iat` i `sub` vrijednosti.

```
{
  "username": "peroPeric123",
  "email": "peroPeric@gmail.com"
}
```

Koji dijelovi kodiranog tokena će se sada promijeniti i zašto? 🤔

► Spoiler alert! Odgovor na pitanje

| Encoded | Decoded | | | | | | |
|--|--|---------|--|----------|--|------------------|--|
| <pre>eyJhbGciOiJIUzI1NiIsInR5cCI 6IkpxVCJ9.eyJ1c2VybmcFtZSI6I nBlcm9QZXJpYzEyMyIsImVtYWls IjoicHBlcmt9QZXJpY0BnbWFpbC5 jb20ifQ.oyypVUgQaFNftKCPjos TCGQK9ytP_qz_fppACfcAS8E</pre> | <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">HEADER:</td> </tr> <tr> <td style="padding: 5px;">{ "alg": "HS256", "typ": "JWT" }</td> </tr> <tr> <td style="padding: 5px;">PAYLOAD:</td> </tr> <tr> <td style="padding: 5px;">{ "username": "peroPeric123", "email": "peroPeric@gmail.com" }</td> </tr> <tr> <td style="padding: 5px;">VERIFY SIGNATURE</td> </tr> <tr> <td style="padding: 5px;">HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) <input type="checkbox"/> secret base64 encoded</td> </tr> </table> | HEADER: | { "alg": "HS256", "typ": "JWT" } | PAYLOAD: | { "username": "peroPeric123", "email": "peroPeric@gmail.com" } | VERIFY SIGNATURE | HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) <input type="checkbox"/> secret base64 encoded |
| HEADER: | | | | | | | |
| { "alg": "HS256", "typ": "JWT" } | | | | | | | |
| PAYLOAD: | | | | | | | |
| { "username": "peroPeric123", "email": "peroPeric@gmail.com" } | | | | | | | |
| VERIFY SIGNATURE | | | | | | | |
| HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) <input type="checkbox"/> secret base64 encoded | | | | | | | |

 Signature Verified

SHARE JWT

Promijenili smo **payload** token i došlo je do promjene **signature** dijela tokena budući da se generira na temelju **headera i payloada**

Do sad nismo koristili nikakav enkripcijski ključ za generiranje **signature** dijela tokena. U praksi ga je potrebno koristiti kako bi se osigurala sigurnost tokena.

Preporučuje se korištenje **256-bitnog ključa** za generiranje **signature** dijela tokena. Ključeve je moguće generirati pomoću raznih alata preko interneta, a možemo koristiti i `crypto` modul u Node.js-u.

```
import crypto from 'crypto';

console.log(crypto.randomBytes(32).toString('hex')); // generira 256-bitni ključ (32 x 8 = 256)
```

Ako pozovete više puta ovaj kod, svaki put ćete dobiti novi nasumični 256-bitni ključ.

Primjer generiranog ključa:

```
1b6bded687b99a58817fd80b41ca72e4dfa68087da8dac7c0a945735e525057d
```

Ključ možemo kopirati u odgovarajuće polje.

Primjetite da se sad generira potpuno različiti **signature** dio tokena.

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.e
yJc2VybmcFtZSI6InBlcm9QZXJpYzEyMyIsImV
tYWlsIjoiCHB1cm9QZXJpY0BnbWFpbC5jb20if
Q.eGgt9ak7c-
SE7HnsAciey92uimRDNOH1znQiw7cJP7Y
```

Decoded EDIT THE PAYLOAD AND SECRET

| |
|--|
| HEADER: ALGORITHM & TOKEN TYPE |
| { "alg": "HS256", "typ": "JWT" } |
| PAYOUT: DATA |
| { "username": "peroPeric123", "email": "peroPeric@gmail.com" } |
| VERIFY SIGNATURE |
| HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), [1b6bded687b99a58817fd]) <input type="checkbox"/> secret base64 encoded |

Signature Verified

SHARE JWT

Generirali smo novi **signature** dio tokena pomoću 256-bitnog ključa

Dodatno, moguće je "dekodirati" ključ pomoću `base64` enkodiranja te na taj način osigurati dodatan sloj sigurnosti. `base64` nije enkripcijski algoritam, već deterministička reverzibilna funkcija koja transformira niz binarnih podataka u niz printabilnih znakova.

2.2 Kako iskoristiti JWT token za autorizaciju?

Kako bismo koristili JWT token za autorizaciju, potrebno je:

1. **Generirati novi JWT token** prilikom **uspješne autentifikacije** korisnika
2. **Poslati JWT token** korisniku kao odgovor u HTTP zahtjevu
3. Na korisničkoj strani, korisnik će **pohraniti JWT token** u lokalnu memoriju web preglednika koristeći `localStorage` ili `sessionStorage`.
 - `localStorage`: podaci se pohranjuju **bez vremenskog ograničenja**, odnosno ostaju pohranjeni i nakon zatvaranja preglednika ili taba, ali se **brišu čišćenjem postavki** u web pregledniku
 - `sessionStorage`: podaci se pohranjuju **samo za vrijeme trajanja sesije**, odnosno brišu se nakon zatvaranja taba ili preglednika, ili brisanjem postavki u web pregledniku
4. **Svaki put kada korisnik pristupi zaštićenom resursu**, klijentska strana **mora poslati pohranjeni JWT token** u zaglavlu HTTP zahtjeva!
5. **Poslužitelj provjerava valjanost JWT tokena i dopušta pristup resursu** ako je token valjan, inače vraća autorizacijsku grešku.

Kako bismo generirali i potvrdili ispravnost JWT tokena na poslužiteljskoj strani, potrebno je koristiti `jsonwebtoken` paket.

Instalirajmo `jsonwebtoken` paket pomoću npm-a:

```
npm install jsonwebtoken
```

Uključimo `jsonwebtoken` u našu aplikaciju:

```
import jwt from 'jsonwebtoken';
```

JWT token generirat ćemo pomoću metode `sign`:

```
jwt.sign(payload, secretOrPrivateKey, [options, callback]);
```

gdje su:

- `payload`: JSON objekt koji sadrži korisničke podatke koje želimo pohraniti u token
- `secretOrPrivateKey`: tajni ključ koji se koristi za generiranje **signature** dijela tokena
- `options`: dodatne opcije (opcionalno) za generiranje tokena (npr. rok trajanja tokena)

Koje informacije želimo pohraniti u **payload** dijelu tokena? U pravilu, to su korisničko ime, email, ID korisnika, rola korisnika, itd. Može biti **sve od navedenog ili samo dio**. Ono što svakako nije uobičajeno, je pohranjivati **osjetljive podatke** kao što su lozinke.

Što mislite, zašto nije dobra praksa pohranjivati osjetljive podatke u JWT token, kao što su lozinke? 😬

► Spoiler alert! Odgovor na pitanje

Primjer generiranja JWT tokena:

```
let payload = { username: 'markoMaric', email: 'markooo@gmail.com' };

// random 256-bitni ključ
let secret_key = '1b6bded687b99a58817fd80b41ca72e4dfa68087da8dac7c0a945735e525057d';

let jwt_token = jwt.sign(payload, secret_key);
console.log(jwt_token);
```

Ako zalijepite ovaj kod direktno u poslužitelj, dobit ćete generirani JWT token.

Primjer generiranog JWT tokena s podacima iznad:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlc2VybmFtZSI6Im1hcmtvTWFyaWMiLCJlbWFpbCI6Im1hcmtvb29A
Z21haWwuY29tIiwiaWF0IjoxNzM2ODA1ODE1fQ.WftGGMyGh5vymH0eRz14oEpf7fPlv7Q5z0L8zoEiNdI
```

Ako zalijepite ovaj token na jwt.io stranicu, možete vidjeti dekodirane podatke u **payload** dijelu tokena.

Dekodirani **payload** dijelovi JWT tokena, **token nije valjan** zbog pogrešnog enkripcijskog ključa

Međutim, vidimo da dobivamo grešku: "invalid signature". Zašto? 🤔

► Spoiler alert! Odgovor na pitanje

Zaključujemo sljedeće:

- **payload** dio tokena možemo dekodirati i vidjeti sadržaj
- **signature** dio tokena ne možemo dekodirati jer ne znamo enkripcijski ključ

Kada bi na poslužitelj stigao ovaj token, poslužitelj bi dekodirao pogrešan **signature** dio tokena i token bi bio **označen kao nevaljan!**

- to implicira da je klijent promijenio **signature** dio tokena i time narušio integritet tokena.

VAŽNO! Jedini način kako klijent može generirati ispravan **signature** dio tokena bez pomoći poslužitelja je ako sazna **enkripcijski ključ**.

Iz tog razloga, enkripcijski ključ je potrebno pohraniti i čuvati na poslužitelju, u **varijablama okruženja** (eng. *environment variables*).

Instalirat ćemo `dotenv` paket kako bismo mogli koristiti varijable okruženja u našoj aplikaciji:

```
npm install dotenv
```

Uključimo `dotenv` u naš poslužitelj:

```
import dotenv from 'dotenv';

dotenv.config();
```

Izradit ćemo `.env` datoteku u korijenskom direktoriju projekta i pohraniti enkripcijski ključ u njoj. Uobičajeno ga je nazvati `JWT_SECRET`, ali naravno, može se zvati kako god.

```
JWT_SECRET=1b6bded687b99a58817fd80b41ca72e4dfa68087da8dac7c0a945735e525057d
```

Sada možemo koristiti ovaj enkripcijski ključ u našoj aplikaciji:

```
const JWT_SECRET = process.env.JWT_SECRET;

let jwt_token = jwt.sign(payload, JWT_SECRET); // koristimo enkripcijski ključ iz varijable
                                               okruženja
console.log(jwt_token);
```

Ako unesemo ispravan enkripcijski ključ na jwt.io stranici, dobit ćemo potvrdu da je token valjan.

The screenshot shows the jwt.io interface. On the left, under 'Encoded', is a long string of characters representing the JWT token. On the right, under 'Decoded', are the parsed components:

- HEADER: ALGORITHM & TOKEN TYPE**:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```
- PAYOUT: DATA**:

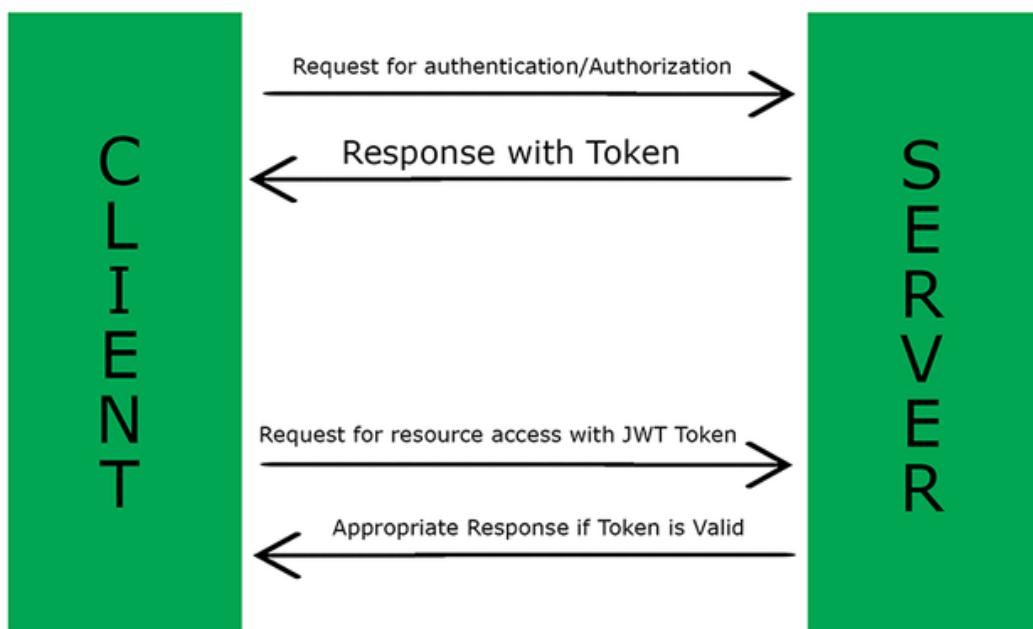
```
{  
  "username": "markoMaric",  
  "email": "markooo@gmail.com",  
  "iat": 1736885815  
}
```
- VERIFY SIGNATURE**:

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  dac7c0a945735e525057d  
)  secret base64 encoded
```

At the bottom, there is a green button labeled 'Signature Verified' and a blue button labeled 'SHARE JWT'.

Dekodirani **payload** dijelovi JWT tokena s ispravnim **signature** dijelom, **token je valjan**

Ključna razlika u usporedbi s pohranom sesija i pristupom baziranim na kolačićima je što JWT token **sadrži sve informacije potrebne za autorizaciju** i nema potrebe za pohranom sesije na poslužitelju, već poslužitelj svaki put provjerava **valjanost tokena**.



Komunikacija između klijenta i poslužitelja koristeći JWT token za autorizaciju

2.3 Provjera valjanosti JWT tokena

Kako bismo provjerili valjanost JWT tokena na poslužiteljskoj strani, koristimo metodu `verify`:

```
jwt.verify(token, secretOrPublicKey, [options, callback]);
```

gdje su:

- `token`: JWT token koji želimo provjeriti (stoji s klijentske strane)
- `secretOrPublicKey`: tajni ključ ili javni ključ koji se koristi za provjeru **signature** dijela tokena
- `options`: dodatne opcije (opcionalno) za provjeru tokena (npr. rok trajanja tokena)
- `callback`: funkcija koja se poziva nakon provjere tokena

Callback funkcija prima dva argumenta: `err` i `decoded` gdje je `decoded` dekodirani **payload** dio tokena, a `err` je greška ako dođe do problema prilikom provjere tokena.

Primjer provjere valjanosti JWT tokena:

```
let token =  
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlc2VybmcFtZSI6Im1hcmtvTWFyaWMiLCJlbWFpbCI6Im1hcmtvb29  
AZ21haWwuY29tIiwiaWF0IjoxNzM2ODA2NjEwfQ.LVqWWsZBn9fxVrbeEEoKFL1VRnfnfJ2wFElpgjf2oBM';  
  
// err je greška u slučaju da token nije valjan, decoded je dekodirani payload u slučaju da  
je token valjan  
jwt.verify(token, JWT_SECRET, (err, decoded) => {  
  if (err) {  
    console.error(`Došlo je do greške prilikom verifikacije tokena: ${err}`);  
    return;  
  }  
  
  console.log('Token je valjan!');  
  console.log(decoded);  
});
```

U konzoli dobivamo ispis:

```
Token je valjan!  
{ username: 'markoMaric', email: 'markooo@gmail.com', iat: 1736806610 }
```

Ako promijenimo samo jedan znak u tokenu, dobit ćemo grešku:

```
Došlo je do greške prilikom verifikacije tokena: JsonWebTokenError: invalid token
```

2.4 Implementacija funkcija za generiranje i provjeru JWT tokena

U praksi, korisno je implementirati funkcije za generiranje i provjeru JWT tokena i smjestiti ih u zasebnu datoteku, kako bi se olakšalo njihovo korištenje u aplikaciji

Definirajte novu datoteku `auth.js` u kojoj ćemo smjestiti sljedeće funkcije:

- `hashPassword` - funkcija za hashiranje lozinke
- `checkPassword` - funkcija za provjeru podudaranja lozinke i hash vrijednosti
- `generateJWT` - funkcija za generiranje JWT tokena **u slučaju uspješne autentifikacije**
- `verifyJWT` - funkcija za provjeru valjanosti JWT tokena **prilikom pristupa zaštićenim resursima**

Uključujemo biblioteke koje koristimo:

```
// auth.js

import bcrypt from 'bcrypt';
import dotenv from 'dotenv';
import jwt from 'jsonwebtoken';

dotenv.config();

const JWT_SECRET = process.env.JWT_SECRET;
```

Funkcija za hashiranje lozinke koja koristi `bcrypt` paket:

```
// auth.js

async function hashPassword(plainPassword, saltRounds) {
  try {
    let hash = await bcrypt.hash(plainPassword, saltRounds); // hashiranje lozinke
    return hash;
  } catch (err) {
    console.error(`Došlo je do greške prilikom hashiranja lozinke: ${err}`);
    return null;
  }
}
```

Funkcija za provjeru podudaranja lozinke i hash vrijednosti:

```
// auth.js

async function checkPassword(plainPassword, hashedPassword) {
  try {
    let result = await bcrypt.compare(plainPassword, hashedPassword); // usporedba lozinke i hash vrijednosti
    return result;
  } catch (err) {
    console.error(`Došlo je do greške prilikom usporedbe hash vrijednosti: ${err}`);
    return false;
  }
}
```

Na isti način ćemo ukomponirati kod za generiranje JWT tokena u funkciju `generateJWT`:

```
// auth.js

async function generateJWT(payload) {
  try {
    let token = jwt.sign(payload, JWT_SECRET); // generiranje JWT tokena s payloadom i enkripcijskim ključem
    return token;
  } catch (err) {
    console.error(`Došlo je do greške prilikom generiranja JWT tokena: ${err}`);
    return null;
  }
}
```

I na kraju, funkcija za provjeru valjanosti JWT tokena:

```
// auth.js

async function verifyJWT(token) {
  try {
    let decoded = jwt.verify(token, JWT_SECRET); // provjera valjanosti JWT tokena
    return decoded;
  } catch (err) {
    console.error(`Došlo je do greške prilikom verifikacije JWT tokena: ${err}`);
    return null;
  }
}
```

Sada možemo koristiti ove funkcije u našoj aplikaciji:

```
// index.js

import { hashPassword, checkPassword, generateJWT, verifyJWT } from './auth.js';
```

1. Korak (Registracija korisnika)

Prilikom registracije korisnika, koristimo funkciju `hashPassword` za hashiranje lozinke:

```
// index.js

app.post('/register', async (req, res) => {
  const { username, password } = req.body;

  let hashedPassword = await hashPassword(password, 10); // hashiranje lozinke

  // dodajemo korisnika u listu korisnika
  users.push({ username, password: hashedPassword });

  res.status(200).send('Korisnik je uspješno registriran!');
});
```

The screenshot shows a POST request to `/register` at `http://localhost:3000/register`. The request body is a JSON object:

```
1 {
2   "username": "lukablaskovic",
3   "password": "lozinka123"
4 }
```

The response is a 200 OK status with the message "Korisnik je uspješno registriran!".

Korak 1: Registracija korisnika, šaljemo POST zahtjev na `/register` rutu

2. Korak (Prijava korisnika s klijentske strane)

Nakon uspješne registracije, korisniku vraćamo potvrdu ali ne moramo vraćati hashiranu lozinku. **Želimo korisnika i hashiranu lozinku spremiti u bazu podataka.**

Radi jednostavnosti, sada ćemo ga spremiti u listu korisnika.

```
console.log(users);
```

Ispis u konzoli

```
[  
  {  
    username: 'lukablaskovic',  
    password: '$2b$10$ziHeJiULEION1DyeA5EAXOfvHnXhfGHycBJw8iyVGRa3iPA32ojhq'  
  }  
]
```

Klijentska strana obrađuje ovaj odgovor i preusmjerava korisnika na formu za prijavu.

Klijentska strana šalje POST zahtjev na `/login` rutu. U ovom koraku, korisnik unosi korisničko ime i lozinku.

3. Korak (Prijava korisnika na poslužiteljskoj strani)

Na poslužitelju, koristimo funkciju `checkPassword` za provjeru podudaranja lozinke i hash vrijednosti na početku rute `/login`.

Naravno, prvo provjeravamo postoji li korisnik u listi korisnika.

Vraćamo istu grešku ako korisnik ne postoji ili ako lozinka nije ispravna.

```
// index.js

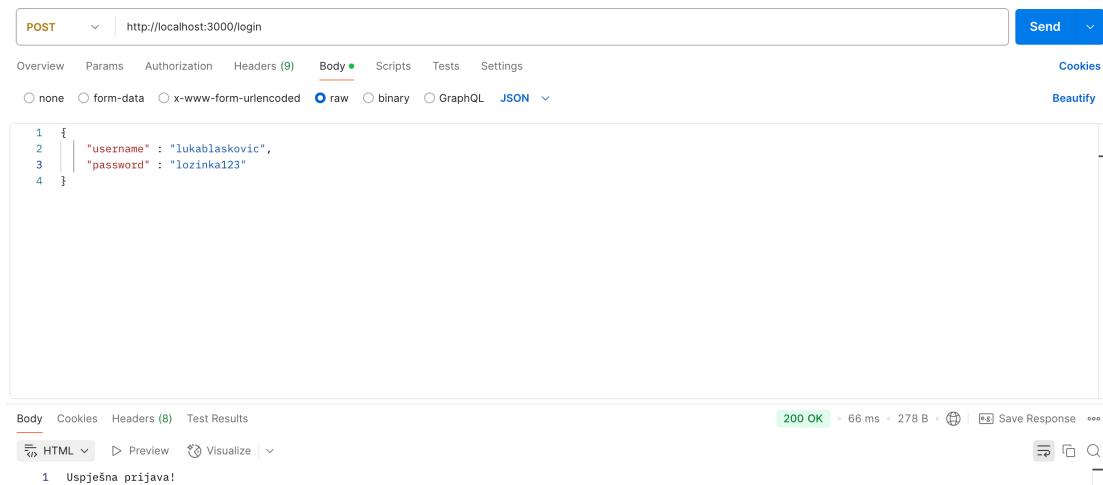
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username);

  if (!user) {
    return res.status(400).send('Greška prilikom prijave!');
  }

  let result = await checkPassword(password, user.password); // usporedba lozinke i hash vrijednosti

  if (!result) {
    return res.status(400).send('Greška prilikom prijave!');
  }
});
```



Korak 3: Uspješna prijava korisnika, šaljemo POST zahtjev na `/login` rutu s istim podacima

4. Korak (Generiranje JWT tokena)

Ako je prijava uspješna, šaljemo korisniku JWT token kao odgovor.

```
// index.js

app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = users.find(user => user.username === username);

  if (!user) {
```

```

    return res.status(400).send('Greška prilikom prijave!');
}

let result = await checkPassword(password, user.password); // usporedba lozinke i hash
vrijednosti

if (!result) {
    return res.status(400).send('Greška prilikom prijave!');
}
// ako je prijava uspješna, generiramo JWT token
let token = await generateJWT({ id: user.id, username: user.username }); // generiranje JWT
tokena
// šaljemo JWT token korisniku
res.status(200).json({ jwt_token: token });
});

```

The screenshot shows a Postman interface. The method is set to POST, the URL is <http://localhost:3000/login>, and the Body tab is selected with the JSON option chosen. The request body contains the following JSON:

```

1  {
2   |   "username" : "lukablasovic",
3   |   "password" : "lozinka123"
4 }

```

The response status is 200 OK, with a response time of 86 ms and a size of 425 B. The response body is a JSON object containing a single key 'jwt_token' with a long string value representing a JWT token.

Korak 4: Uspješna prijava korisnika, šaljemo JWT token kao odgovor

5. Korak (Pohrana JWT tokena na klijentskoj strani i slanje na poslužitelj)

Korisnik sprema JWT token u lokalnu memoriju web preglednika (npr. `localStorage`).

Definirat ćemo neke resurse koji se odnose na korisnika i koji su zaštićeni, npr. možemo definirati resurs `/objave` gdje će korisnik moći pregledati samo svoje objave.

```
// index.js

let objave = [
  { id: 1, naslov: 'Prva objava', sadrzaj: 'Ovo je prva objava', autor: 'lukablaskovic' },
  { id: 2, naslov: 'Druga objava', sadrzaj: 'Ovo je druga objava', autor: 'markoMaric' },
  { id: 3, naslov: 'Treća objava', sadrzaj: 'Ovo je treća objava', autor: 'peroPeric' },
  { id: 4, naslov: 'Četvrta objava', sadrzaj: 'Ovo je četvrta objava', autor: 'lukablaskovic' }
];
// index.js

app.get('/objave', async (req, res) => {
  res.json(objave); // ali samo one koje se odnose na autoriziranog korisnika?
});
```

Rekli smo da JWT token želimo poslati u zaglavju HTTP zahtjeva: `Authorization`. Kao vrijednost ovog zaglavja, koristimo `Bearer` prefiks i sam JWT token nakon jednog razmaka.

Bearer token predstavlja **autentifikacijski tip** koji koristi JWT token za autorizaciju.

Dakle, zaglavje mora biti:

```
Authorization: Bearer <JWT token>
```

odnosno:

```
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Imx1a2FibGFza292aWMiLCJpYXQiOjE3MzY4MDgwNT19.C8BeyReQFYz6l0L7vsvjN5HIrre3EnDuYIZIppZ0tDA
```

U Postmanu je moguće odabrati tip autorizacije `Bearer Token` i zalijepiti JWT token u polje.

The screenshot shows the Postman interface with the following details:

- Authorization Tab:** The `Authorization` tab is selected, and the `Auth Type` dropdown is set to `Bearer Token`.
- Token Field:** A large text area displays a long JWT token: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Imx1a2FibGFza292aWMiLCJpYXQiOjE3MzY4MDgwNT19.C8BeyReQFYz6l0L7vsvjN5HIrre3EnDuYIZIppZ0tDA`.
- Body Tab:** The `Body` tab is active, showing a JSON object with one key-value pair:


```
{
  "jwt_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Imx1a2FibGFza292aWMiLCJpYXQiOjE3MzY4MDgwNT19.C8BeyReQFYz6l0L7vsvjN5HIrre3EnDuYIZIppZ0tDA"
}
```
- Response Header:** The response header shows `200 OK`, `86 ms`, `425 B`, and other details.

Korak 5: Zalijepimo Bearer Token u Postmanu pod `Authorization` zaglavje

6. Korak (Provjera valjanosti JWT tokena na poslužiteljskoj strani)

Ako se vratimo na rutu `/objave`, sada možemo provjeriti valjanost JWT tokena dohvaćanjem zaglavja kroz `req.headers.authorization`.

- koristimo metodu `split(' ')` kako bismo odvojili `Bearer` prefiks od samog JWT tokena
- zatim se indeksiramo na sam JWT token (indeks 1)
- dekodiramo JWT token pomoću funkcije `verifyJWT` iz `auth.js` datoteke

Ako je JWT token valjan, `verifyJWT` će vratiti dekodirani **payload** dio tokena, u suprotnom će vratiti `null`.

```
// index.js

app.get('/objave', async (req, res) => {
  let token = req.headers.authorization.split(' ')[1]; // dohvaćanje JWT tokena iz zaglavlja

  let decoded = await verifyJWT(token); // provjera valjanosti JWT tokena

  if (!decoded) {
    return res.status(401).send('Nevaljan JWT token!');
  }

  // filtriramo objave prema autoru ako je JWT token valjan, odnosno ako je korisnik
  // autoriziran
  let userObjave = objave.filter(objava => objava.autor === decoded.username); // dohvaćamo
  // podatke iz dekodiranog payloada (decoded)

  res.json(userObjave);
});
```

Ako je JWT token valjan, **korisnik će dobiti samo one objave koje su njegove** jer smo tako definirali u funkciji `filter`.

The screenshot shows the Postman application interface. A GET request is made to `http://localhost:3000/objave`. In the Authorization tab, the type is set to "Bearer Token" and the token value is `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.eyJ...`. The response status is 200 OK, and the JSON body contains two objects representing posts:

```

1  [
2   {
3     "id": 1,
4     "naslov": "Prva objava",
5     "sadrzaj": "Ovo je prva objava",
6     "autor": "lukablasovic"
7   },
8   {
9     "id": 4,
10    "naslov": "Četvrta objava",
11    "sadrzaj": "Ovo je četvrta objava",
12    "autor": "lukablasovic"
13  }
14 ]

```

2.5 Autorizacijski middleware

U praksi, korisno je definirati **autorizacijski middleware** koji će se izvršiti prije svakog pristupa zaštićenim resursima kako ne bi morali svaki put provjeravati valjanost JWT tokena u samoj ruti.

Autorizacijski middleware će omogućiti da se **provjera valjanosti JWT tokena izvrši prije nego što se izvrši sama ruta**, što je u ovom slučaju poželjno ponašanje.

Definirajmo novi middleware `authMiddleware` u `auth.js` datoteci:

```
const authMiddleware = async (req, res, next) => {
  // implementacija
  next(); // nastavljamo dalje
};
```

Što ćemo staviti u ovaj middleware? **Sve što je potrebno za provjeru JWT tokena**

- pristupamo JWT tokenu iz zaglavlja (`req.headers.authorization`)
- dekodiramo JWT token pomoću funkcije `verifyJWT` iz `auth.js` datoteke
- ako je token valjan, spremamo dekodirani **payload** dio tokena u `req.authorised_user` objekt
- ako token nije valjan, vraćamo grešku

```
// auth.js

const authMiddleware = async (req, res, next) => {
  let token = req.headers.authorization.split(' ')[1]; // dohvaćanje JWT tokena iz zaglavlja

  let decoded = await verifyJWT(token); // provjera valjanosti JWT tokena

  if (!decoded) {
    return res.status(401).send('Nevaljan JWT token!');
  }

  req.authorised_user = decoded; // spremamo dekodirani payload u req objekt
  next(); // nastavljamo dalje
};
```

Sada možemo upotrijebiti ovaj middleware u ruti `/objave` i skratiti kod:

```
// index.js

app.get('/objave', [authMiddleware], async (req, res) => {
  let userObjave = objave.filter(objave => objave.autor === req.authorised_user.username); // dohvaćamo podatke iz dekodiranog payloada (req.authorised_user)

  res.json(userObjave);
});
```

Ovaj middleware možemo upotrijebiti na svim rutama koje su "zaštićene", odnosno koje zahtijevaju autorizaciju korisnika.

2.6 Rok trajanja JWT tokena

U praksi, korisno je definirati **rok trajanja JWT tokena** kako bi se spriječilo zloupotrebu tokena. Na primjer, ako maliciozni korisnik ukrade JWT token na klijentskoj strani, može ga koristiti za pristup zaštićenim resursima sve dok token ne istekne. Ako je token beskonačnog trajanja, može se koristiti zauvijek.

U tu svrhu, nije loše definirati **rok trajanja tokena** u **payload** dijelu tokena.

Prilikom generiranja JWT tokena, možemo definirati **rok trajanja** tokena u sekundama pomoću opcije `expiresIn` u `options` objektu:

```
let token = jwt.sign(payload, JWT_SECRET, { expiresIn: '1h' }); // token traje 1 sat
```

Ovaj token će trajati **1 sat** od trenutka generiranja. Nakon toga, funkcija `verify` će vratiti grešku `"TokenExpiredError"`. U tom slučaju potrebno je na klijentskoj strani preusmjeriti korisnika na formu za prijavu, a poslužitelj neće dozvoliti pristup zaštićenim resursima.

Samostalni zadatak za Vježbu 7

Nadogradite aplikaciju iz vježbe [TaskManager](#) tako da sadrži autentifikaciju i autorizaciju korisnika pomoću JWT tokena.

1. Implementirajte registraciju korisnika na poslužiteljskoj strani, a na klijentskoj strani omogućite korisniku unos korisničkog imena i lozinke.
2. Na poslužiteljskoj strani pohranite korisnika u Mongo bazu, a lozinku obvezno hashirajte prije pohrane.
3. Za svaki zadatak u bazi podataka dodajte ključ `userId` koji će sadržavati ID korisnika koji je izradio taj zadatak (`userId` je podatak koji generira sam MongoDB)
4. Na poslužiteljskoj strani implementirajte rutu za prijavu korisnika, gdje korisnik unosi korisničko ime i lozinku a dobiva potpisani JWT token koji traje 24 sata.
5. Implementirajte autorizacijski middleware koji će se izvršiti prije rute za dohvaćanje svih zadataka. Nakon provjere ispravnosti JWT tokena, moraju biti vraćeni oni zadaci koji se odnose na autoriziranog korisnika. Ako token nije valjan ili ne postoje zadaci za tog korisnika, vratite odgovarajuću grešku.
6. Na klijentskoj strani implementirajte pohranu JWT tokena u `localStorage` i slanje tokena u zaglavljtu HTTP zahtjeva na svaku rutu koja zahtijeva autorizaciju, npr. dohvaćanje svih zadataka.
7. Na poslužiteljskoj strani upotrijebite autorizacijski middleware i na ruti za dodavanje novog zadatka, gdje će se: prvo provjeriti valjanost JWT tokena, zatim pronaći korisnik čiji se ID nalazi u tokenu (možete i prema korisničkom imenu) te napisljetu dodati zadatak u bazu podataka.

Primjer: Ako korisnik 'anaAnic' dodaje novi zadatak, uz podatke o zadatku potrebno je poslati i header s JWT tokenom koji je generiran za korisnika 'anaAnic'. Na poslužitelju se provjerava valjanost tokena, pronalazi korisnik 'anaAnic' i dodaje zadatak u bazu podataka s ključem `userId` koji sadrži ID korisnika 'anaAnic'.

```
{  
  id: 1,  
  userId: "64b67f9dc3a1d3c7e6a25f1b",  
  naslov: "Naučiti JWT",  
  opis: "Naučiti kako koristiti JWT token za autorizaciju korisnika",  
  zavrsen: false  
  tags: ["hitno", "faks"]  
}
```