

# 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

## (5) MongoDB baza podataka

---

#5

WA

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-orijentiranom 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](#)

# 1. MongoDB

MongoDB je dokumentno-orijentirana (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 računalo 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.



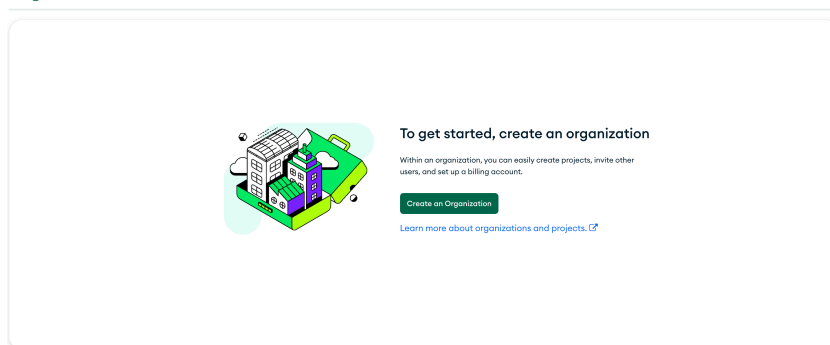
Atlas značajno pojednostavljuje upravljanje i održavanje MongoDB baze podataka. Developer se može fokusirati na razvoj aplikacije, dok se MongoDB Atlas brine o infrastrukturi i sigurnosti baze podataka, kao i o automatskom skaliranju i replikaciji podataka.

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



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 Atlasu 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`.

FIPU-WA > PROJECTS

## Create a Project

[Name Your Project](#) [Add Members](#)

### Name Your Project

Project names have to be unique within the organization (and other restrictions).

FIPU-WA-Project

### Add Tags (Optional)

Use tags to efficiently label and categorize your projects. A project can have a maximum of 50 tags. You can modify tags for the project later. [Learn more](#)

Key	Value	Actions
Select a key or enter your own	Select a value or enter your own	
+ Add tag		

0 TAGS

[Cancel](#) [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 odabrati `AWS`, `Azure` ili `GCP`. Mi ćemo odabrati `AWS` → `Frankfurt (eu-central-1)`.

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

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

FIPU-WA > PROJECTS

## Create a Project

[Name Your Project](#) [Add Members](#)

### Name Your Project

Project names have to be unique within the organization (and other restrictions).

FIPU-WA-Project

### Add Tags (Optional)

Use tags to efficiently label and categorize your projects. A project can have a maximum of 50 tags. You can modify tags for the project later. [Learn more](#)

Key	Value	Actions
Select a key or enter your own	Select a value or enter your own	
+ Add tag		

0 TAGS

[Cancel](#) [Next](#)

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

2

3

Set up connection securityChoose a connection methodConnect

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

laskovi

Password

\*\*\*\*\*

SHOW

Copy

Create Database User

Close

Choose a connection method

## 2. Povezivanje na cluster

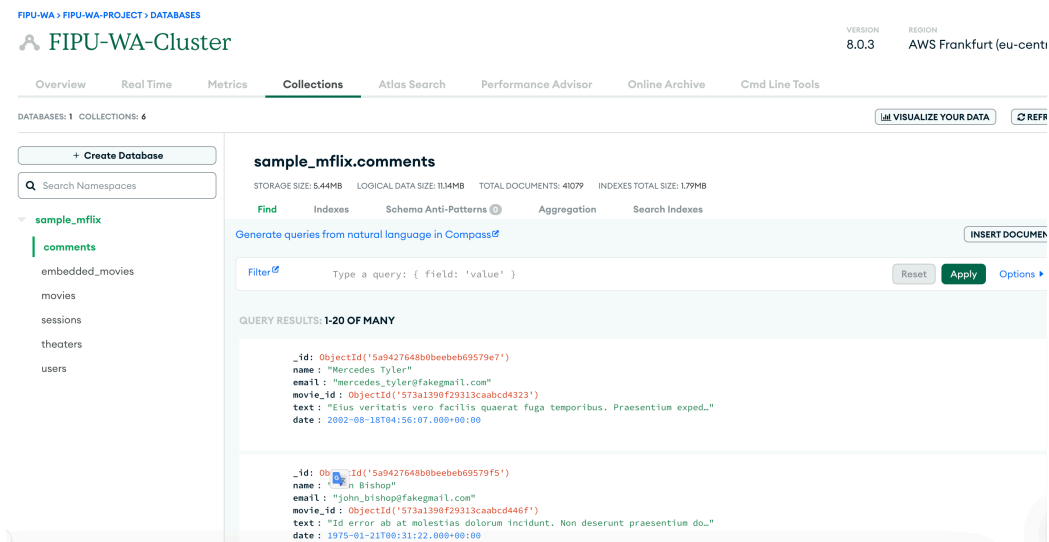
Jednom kad ste uspješno napravili *cluster* u MongoDB Atlasu, možete se povezati na njega na više načina:

- [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.

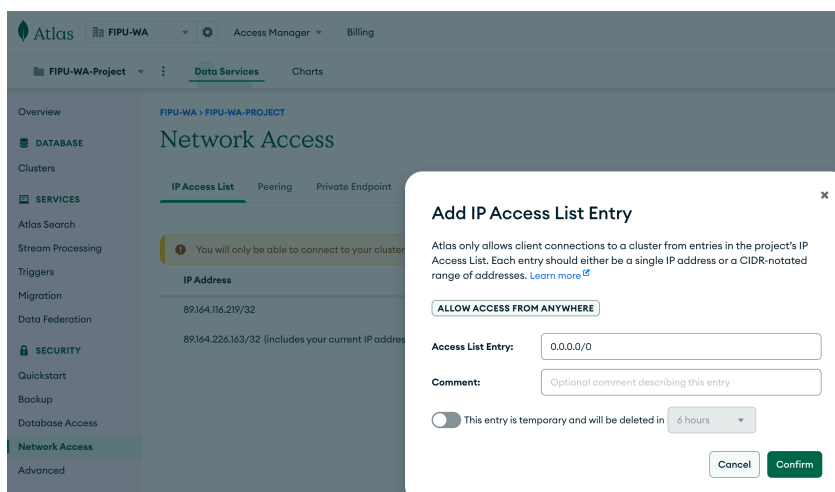
- Moguće je (i preporučljivo) koristiti i druge alate za povezivanje, kako biste imali bolji uvid u podatke u bazi i kako biste mogli brže i jednostavnije raditi s podacima na više razina apstrakcije.

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**.



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 konkretan *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 baze. 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**

Connect to FIPU-WA-Cluster

Set up connection security
Choose a connection method
Connect

Connecting with MongoDB Driver

1. Select your driver and version

We recommend installing and using the latest driver version.

Driver
Version

Node.js
6.7 or later

2. Install your driver

Run the following on the command line

```
npm install mongodb
```

View MongoDB Node.js Driver installation instructions.

3. Add your connection string into your application code

Use this connection string in your application

View full code sample
Show Password

```
mongodb+srv://lbaskovi:<db_password>@fipu-wa-cluster.mpyeq.mongodb.net/?
retryWrites=true&w=majority&appName=FIPU-WA-Cluster
```

Replace <db\_password> with the password for the lbaskovi database user. Ensure any option params are URL encoded.

RESOURCES

Get started with the Node.js Driver
Node.js Starter Sample App

Access your Database Users
Troubleshoot Connections

*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>`

mongodb+srv://	user:pass	@	sample.host:27017	/	?maxPoolSize=20&w=majority
protocol	credentials		hostname/IP and port of instance(s)		connection options

**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 clustera u zasebnu varijablu):

- naravno, zalijepite 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 varijable 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 analogne 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.

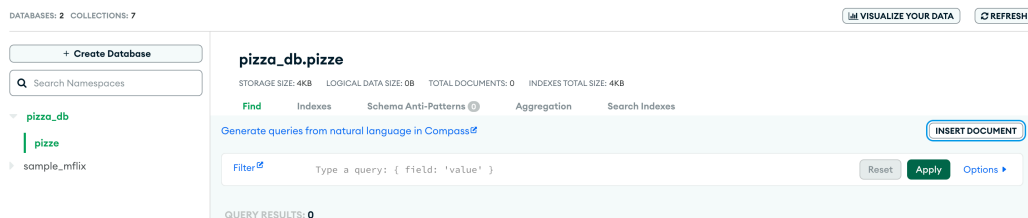
### 3.1.1 GET `/pizze`

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.



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:

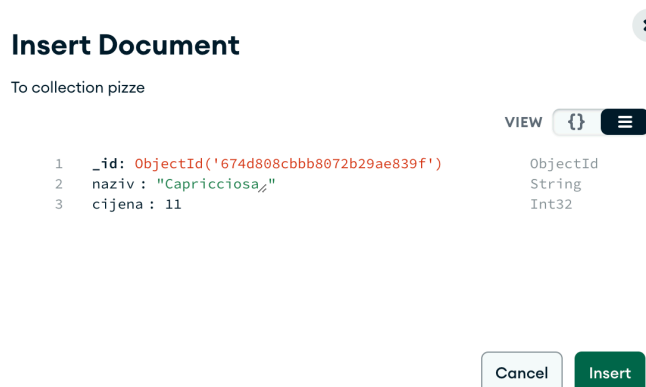
- pisanjem `JSON` strukture (ustvari je `BSON`)
- kroz sučelje za unos podataka

```
{ "_id": { "$oid": "674d808cbbb8072b29ae839f" } }
```

Na postojeći zapis dodajemo polja `naziv` i `cijena`.

```
{
  "_id": { "$oid": "674d808cbbb8072b29ae839f" },
  "naziv": "Capricciosa",
  "cijena": 11
}
```

Preko sučelja izgleda ovako:



Dodajte sljedeće pizze u kolekciju:

```
{
  "naziv": "Margherita",
  "cijena": 9
}
```

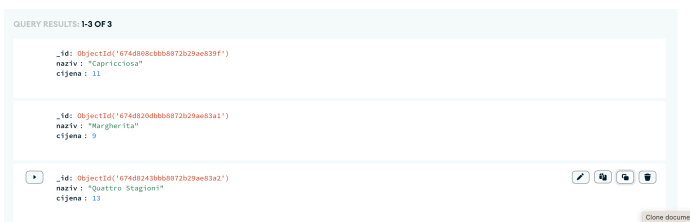


```

},
{
  "naziv": "Quattro Stagioni",
  "cijena": 13
},
{
  "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!):



Nakon što ste dodali pizze, možemo ih dohvatiti na isti način kao prethodno korisnike, koristeći metodu `collection().find()`

```

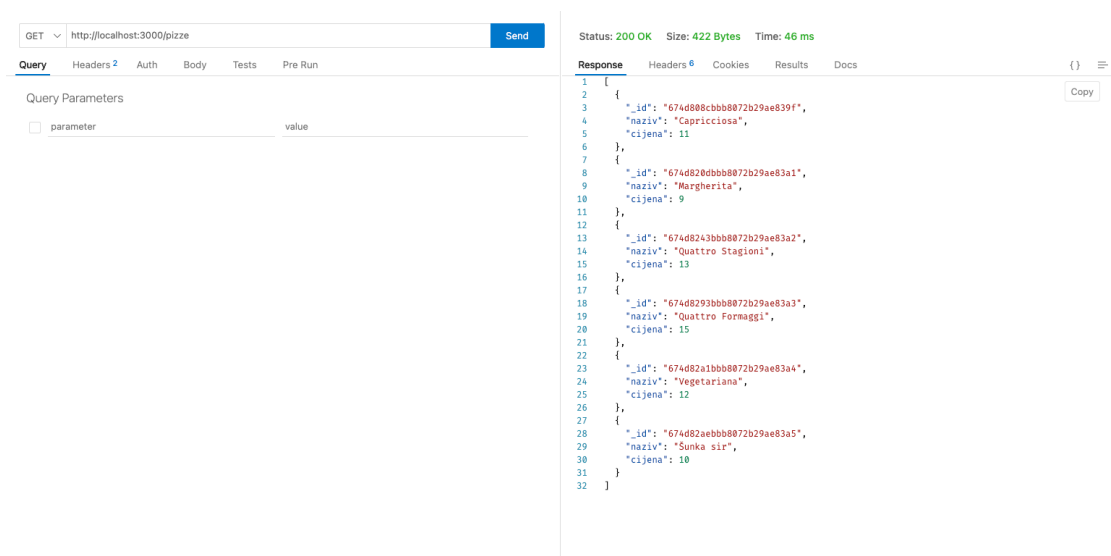
app.get('/pizze', async (req, res) => {
  let pizze_collection = db.collection('pizze'); // referenca na kolekciju 'pizze'
  let allPizze = await pizze_collection.find().toArray(); // pretvorba u Array
  res.status(200).json(allPizze);
});

```

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`



Ako ste dobili statusni kod 200 i podatke o pizzama, uspješno ste dohvatili 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.

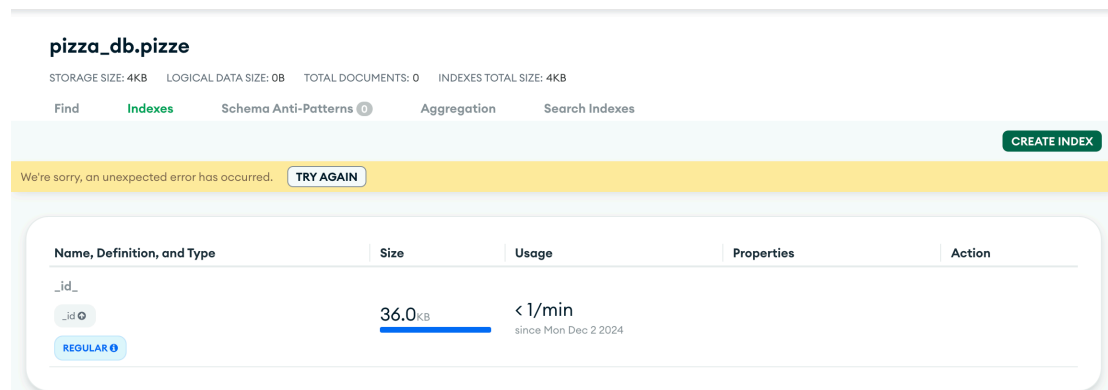
Samim tim, nemamo niti strogo definirane ključeve, poput **Primary key** u relacijskim bazama podataka.

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`



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 kompliciranije 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()`, uvijek 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:

## Create Index

TIP

The best indexes for your application must take a number of factors into account. To learn about index creation best practices, read the [Index Strategies Documentation](#).

For more information on creating indexes in Data Explorer, see [Create an Index](#).

### Collection

pizza\_db.pizze

### Fields

1	{
2	"naziv": 1
3	}

### Options

{	
"unique": true	
}	

### Collation options

--	--

Cancel

Review

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> REGULAR	36.0 B	<1/min since Mon Dec 2 2024		
<code>naziv_1</code> REGULAR	20.0 B	<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`)

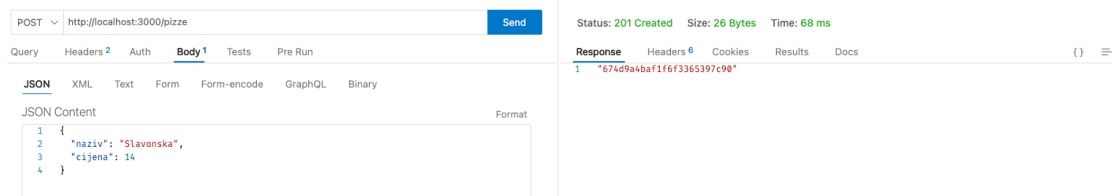
Naš objekt je u tijelu HTTP zahtjeva, koji sad mora izgledati ovako:

```
{
  "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.errorMessage);
    res.status(400).json({ error: error.errorMessage }); // 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):

POST http://localhost:3000/pizze

Status: 400 Bad Request Size: 206 Bytes Time: 74 ms

Response

```
{
  "error": {
    "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"
    }
  }
}
```

Međutim ako dodamo novu pizzu **Fantasia**:

```
{
  "naziv": "Fantasia",
  "cijena": 12.5
}
```

POST http://localhost:3000/pizze

Status: 201 Created Size: 41 Bytes Time: 73 ms

Response

```
{
  "insertedId": "674d9ec10d1918579e32b79b"
}
```

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.errorMessage);
    res.status(400).json({ error: error.errorMessage });
  }
});
```

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 pize, ali različitih veličina i količina.

*Primjer strukture JSON tijela zahtjeva:*

```
{
  "kupac": "Marko Marić",
  "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"
  }
]
}

```

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', 'količina', 'veličina'];

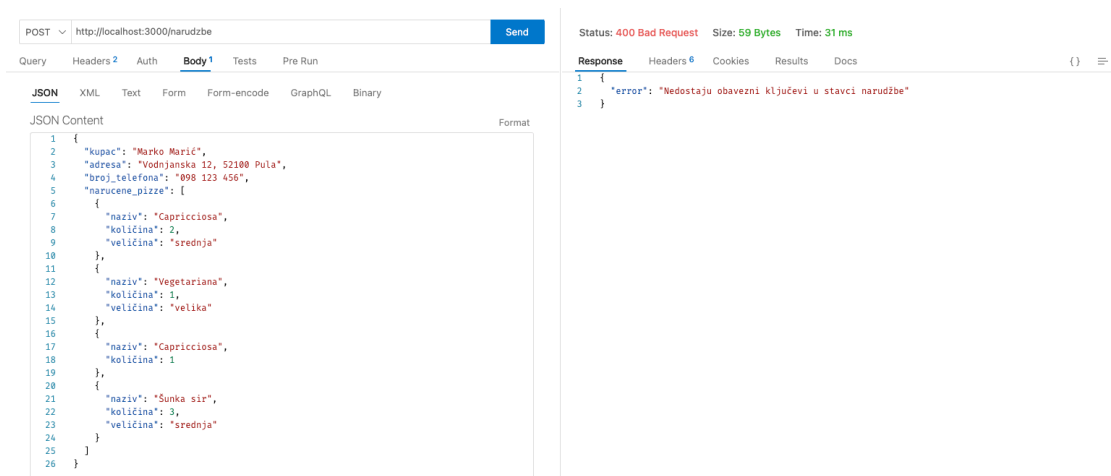
  if (!obavezniKljucevi.every(kljuc => kljuc in novaNarudzba)) {
    return res.status(400).json({ error: 'Nedostaju obavezni ključevi' });
  }

  // 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 obaveznih ključeva iz tijela zahtjeva i provjerite je li validacija ispravna.



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"
      količina : 2
      veličina : "srednja"
    1: Object
      naziv : "Vegetariana"
      količina : 1
      veličina : "velika"
    2: Object
      naziv : "Capricciosa"
      količina : 1
      veličina : "mala"
    3: Object
```

U prikazu vidimo klasične JSON oznake (Array, Object), ali i `ObjectId` oznake 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', 'količina', 'veličina'];

  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 prosljedio 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' });
}

```

OST

http://localhost:3000/narudzbe

Send

Query

Headers2

Auth

Body1

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

```
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       "količina": 2,
9       "veličina": "srednja"
10    },
11    {
12      "naziv": "Vegetariana",
13      "količina": 1,
14      "veličina": "velika"
15    },
16    {
17      "naziv": "Capricciosa",
18      "količina": 1,
19      "veličina": "mala"
20    },
21    {
22      "naziv": "Šunka sir",
23      "količina": 3,
24      "veličina": "srednja"
25    }
26  ]
27 }
```

Status: 400 Bad Request

Size: 51 Bytes

Time: 44 ms

Response

Headers6

Cookies

Results

Docs

```
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 pize. `PUT` metodu koristit ćemo za zamjenu cijelog menija novim.

### 3.3.1 PATCH `/pizze/:naziv`

Prvo ćemo definirati PATCH metodu za ažuriranje cijene pize. Kako smo već definirali GET metodu za dohvaćanje pize 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';
    }
  }
}

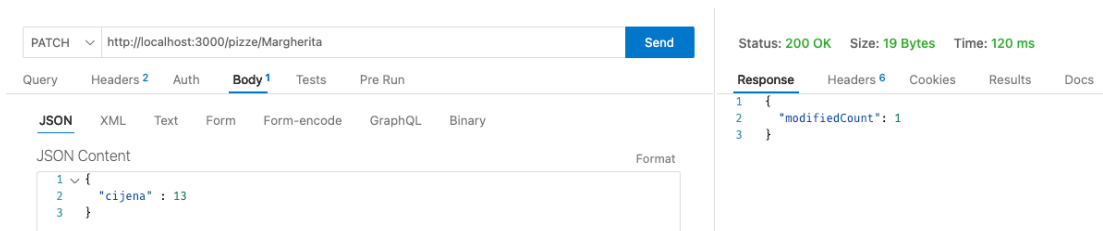
// updateOne() metoda
collection().updateOne({ naziv: 'Vegetariana' }, { $currentDate: { datum_dodavanja: { $type: 'date' } } });
```

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:



Kao odgovor dobivamo broj ažuriranih dokumenata (`modifiedCount : 1`).

Ovaj podatak možemo iskoristiti kako bi se uvjerali 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 REST client interface. The top bar indicates a GET request to `http://localhost:3000/narudzbe/674dbb61cd85b5d3bd083714` with a status of 200 OK, size of 377 Bytes, and time of 43 ms. The 'Query' tab is active, showing no parameters. The 'Response' tab displays the following JSON data:

```
{
  "_id": "674dbb61cd85b5d3bd083714",
  "kupac": "Marko Marić",
  "adresa": "Vodnjanska 12, 52100 Pula",
  "broj_telefona": "098 123 456",
  "narucene_pizze": [
    {
      "naziv": "Margherita",
      "količina": 2,
      "veličina": "srednja"
    },
    {
      "naziv": "Vegetariana",
      "količina": 1,
      "veličina": "velika"
    },
    {
      "naziv": "Capricciosa",
      "količina": 1,
      "veličina": "mala"
    },
    {
      "naziv": "Šunka sir",
      "količina": 3,
      "veličina": "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 naiđe 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.errorMessage);
    res.status(400).json({ error: error.errorMessage });
  }
});
```

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.errorMessage);
    res.status(400).json({ error: error.errorMessage });
  }
});
```



PUT

http://localhost:3000/pizze

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": "Šunka 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 }
```

Status: 200 OK Size: 19 Bytes Time: 100 ms

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 pronade**). 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 REST client interface. On the left, the 'Query' tab is active, displaying a DELETE request to `http://localhost:3000/pizze/Margherita`. The 'Body' tab is also visible. On the right, the 'Response' tab shows the result of the request: a 200 OK status, 18 bytes size, and 75 ms time. The response body is a JSON object: `{ "deletedCount": 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.errorMessage);
    res.status(400).json({ error: error.errorMessage });
  }
});
```

## 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 operatori
{
  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 });
}
});
```

Query Parameters

Parameter	Value
<input checked="" type="checkbox"/> cijena	12.5
<input type="checkbox"/> parameter	value

Response

```
[
  {
    "_id": "674de678d0113a49fff63e85",
    "naziv": "Napolitana",
    "cijena": 12.5
  }
]
```

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
```

Query Parameters

Parameter	Value
<input checked="" type="checkbox"/> cijena	12.5
<input type="checkbox"/> parameter	value

Response

```
[
  {
    "_id": "674de678d0113a49fff63e85",
    "naziv": "Napolitana",
    "cijena": 12.5
  },
  {
    "_id": "674de678d0113a49fff63e87",
    "naziv": "Capricciosa",
    "cijena": 13.5
  },
  {
    "_id": "674de678d0113a49fff63e88",
    "naziv": "Vegetariana",
    "cijena": 14.5
  },
  {
    "_id": "674de678d0113a49fff63e89",
    "naziv": "Sunka sir",
    "cijena": 15.5
  },
  {
    "_id": "674de678d0113a49fff63e8a",
    "naziv": "Quattro Stagioni",
    "cijena": 16.5
  },
  {
    "_id": "674de678d0113a49fff63e8b",
    "naziv": "Fantasia",
    "cijena": 17.5
  }
]
```

## 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

Status: 200 OK

Size: 493 Bytes

Time: 41 ms

Query

Headers 2

Auth

Body

Tests

Pre Run

Query Parameters

☒ naziv
 

1

☒ cijena
 

-1

☐ parameter
 

value

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 dohvatimo.

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.

- `$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-orijentirana 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 ( <code>-1</code> ) ili zadnji ( <code>1</code> ) 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
<code>\$and</code>	<pre>{ \$and: [ {   condition1 }, {   condition2 } ] }</pre>	<pre>{ \$and: [{ age: {   \$gt: 20 } }, { age: {   \$lt: 30 } }] }</pre>	Spaja više uvjeta, samo dokumenti koji su istiniti za sve uvjete će bit vraćeni.
<code>\$or</code>	<pre>{ \$or: [ {   condition1 }, {   condition2 } ] }</pre>	<pre>{ \$or: [{ age: { \$lt:   20 } }, { age: { \$gt:   30 } }] }</pre>	Spaja više uvjeta, dokumenti koji su istiniti za barem jedan uvjet će bit vraćeni.
<code>\$not</code>	<pre>{ key: { \$not: {   condition } } }</pre>	<pre>{ age: { \$not: {   \$gte: 30 } } }</pre>	Negira uvjet, vraća samo one dokumente za koje uvjet ne vrijedi.
<code>\$nor</code>	<pre>{ \$nor: [ {   condition1 }, {   condition2 } ] }</pre>	<pre>{ \$nor: [{ age: {   \$lt: 20 } }, { age: {   \$gt: 30 } }] }</pre>	Spaja više uvjeta, vraća samo one dokumente gdje koji ne zadovoljavaju niti jedan.
<code>\$exists</code>	<pre>{ key: {   \$exists: boolean } }</pre>	<pre>{ age: { \$exists:   true } }</pre>	Podudara dokumente gdje specificirani ključ <code>key</code> postoji odnosno ne postoji <code>boolean</code>
<code>\$type</code>	<pre>{ key: { \$type:   value } }</pre>	<pre>{ age: { \$type: "int" } }</pre>	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 nadgradite ga na sljedeće načine:

- Implementirajte GET `/pizze` endpoint koji će vraćati sve dostupne pizze iz MongoDB baze podataka
- U MongoDB, nadgradite 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.