



Univerzitet u Beogradu - Elektrotehnički fakultet

NodeJS i rad sa bazama



Predavač: Dražen Drašković

Beograd, septembar 2025. godine



Dve vrste baza podataka

- Nerelacione baze podataka
 - napravljene zbog eksplozije količine i vrsta podataka, kao i broja istovremenih zahteva nad bazom i nepredviđene dužine trajanja transakcije
 - najpoznatije nerelacione sastoje se od parova ključ-vrednost, kolonske baze, baze zasnovane na dokumenatima, grafovske baze
- Relacione baze podataka
 - organizacija podataka zasnovana na relacionom modelu entiteta i odnosa
 - između tabela stvaraju se veze
 - svaka tabela mora imati primarni ključ (PK), a u vezama učestvuju ti ključevi kao strani ključevi (FK)



Ključ-vrednost baze (1)

- Najjednostavniji tip NoSQL baza
- Svaki zapis se čuva kao par sastavljen od ključa i vrednosti
- Pristupa se samo preko ključa
- Omogućava skladištenje podataka bez prethodno definisane šeme
- Primaju i ažuriraju podatke zasnovane samo na ograničenom skupu vrednosti, dok je za potrebe upita nad drugim vrednostima neophodno kreirati sopstvene indekse
- Ažuriranje ovakvih sistema zahteva više prolaza kroz bazu: prvi kojim se pronade zapis, drugi kojim se ažurira zapis, treći kojim se ažurira indeks
- Ključ-vrednost baze omogućavaju horizontalno skaliranje



Ključ-vrednost baze (2)

- Predstavnic: DynamoDB (Amazon), Redis, Oracle NoSQL
- Kod DynamoDB tabele nemaju fiksnu šemu, dok njeni objekti mogu imati proizvoljan broj atributa
- Prednost: raste paralelno sa podacima, performanse su bolje ako se dodaju novi serveri
- Tabela je kolekcija stavki, a svaka stavka je kolekcija ključ-vrednost parova, koji predstavljaju attribute
- Primarni ključ predstavljen putem heša (ili heš sa opsegom)
- Heš ključ koristi jedan atribut od koga se kreira heš indeks
- Kako nalazimo stavku? Moramo znati tačnu vrednost ključa



Ključ-vrednost baze (3)

DynamoDB tabela Korisnici

UserId	Ime	Prezime	e-mail	Pol
markom	Marko	Marković	marko@domen.com	muški
petarp	Petar	Petrović	petar@domen.com	muški
jovanaj	Jovana	Jovanović		

Ključevi

Stavke

Atributi



Kolonske baze (1)

- Podacima se pristupa preko kolona, a ne po vrednostima
- Obrada po kolonama daje bolje performanse
- Za čuvanje podataka koriste tzv. kolonske familije ili multidimenzionalne sortirane mape
- Kolone imaju ime i pamte veći broj vrednosti po vrsti (svaka identifikovana vremenskom oznakom)
- Do svake vrednosti može se doći preko: ključ-vrednosti, ključ-kolone i vremenske oznake
- Primeri: Cassandra (distribuirana baza za velike količine brzorastućih podataka), MariaDB, CrateDB, ClickHouse, Hypertable, Apache HBase, itd.



Kolonske baze (2)





Grafovske baze (1)

- Zasnovane na teoriji grafova, uzimajući u obzir veze između podataka i tipove podataka
- Grafovi se sastoje od čvorova (entiteta), koji mogu da budu međusobno povezani neodređenim brojem veza (grana) i svojstava koji su atributi čvorova ili veza
- Cilj ovih baza: čuvanje podataka bez ograničenja koja nameću unapred definisani modeli; podaci se organizuju na način koji diktiraju priroda podataka i njihovih međusobnih veza

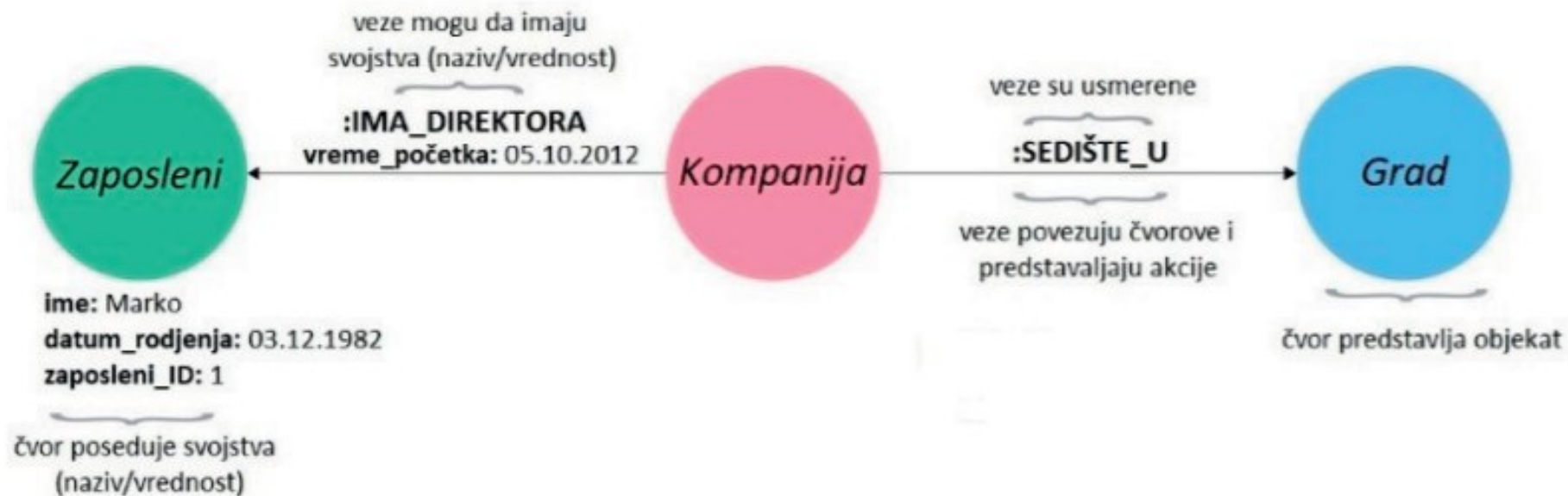


Grafovske baze (2)

- Organizacija podataka u bazi pokazuje kako se svaki pojedinačni entitet povezuje, odnosno u kakvoj je relaciji sa drugim entitetima.
- Problem samoreferencirajućih podataka (koji postoji kod relacionih) prevaziđen je ovde brzim prolaskom kroz čvorove i njihove veze.
- Posebna prednost ovih baza: čuvanje podataka u obliku u kom su uneti, korišćenje pokazivača za navigaciju i prolazak kroz graf.
- Primeri: AllegroGraph, GraphDB Lite, Amazon Neptune, AnzoGraph, ArangoDB, InfiniteGraph, InfoGrid, HyperGraphDB, itd.



Grafovske baze - primer





Baze zasnovane na dokumentima (1)

- Baza koja se skladišti u dokumentima, nalik na JSON (*JavaScript Object Notation*) format
- Svaki dokument opisuje jedan objekat i sadrži jedno ili više polja. Svako polje sadrži jednu vrednost nekog od predefinisanih tipova: tekstualnog, binarnog, decimalnog, datumskog ili tipa niza.
- Značaj ovih baza: podaci se unose bez pripreme, a imamo brz i jednostavan pristup podacima
- Najpopularniji predstavnik: Mongo DB
- MongoDB se sastoji od kolekcija (Collections), a u kolekcije se dodaju dokumenti.
- Ono što su kod relacionih tabele, to su Mongo kolekcije, ono što su zapisi unutar tabele, to su dokumenti u MongoDB.



Baze zasnovane na dokumentima (2)

- Ključna razlika:
 - relacione: zapisi jedne tabele moraju imati istu strukturu
 - nerelacione: dokumenti jedne kolekcije mogu imati različita polja
 - polja u različitim dokumentima mogu biti različitih tipova podataka, čak i kada modeliraju istu vrstu objekata
 - zaključak: kod MongoDB moguće kreirati veze između dokumenata koje su i 1:1 ili 1:N



Primeri kod MongoDB

```
{
  _id: "marko",
  ime: "Marko Marković",
  adrese: [
    {
      ulica: "Petra Lekovića 74",
      grad: "Užice",
      država: "Srbija"
    },
    {
      ulica: "Kostolačka 123",
      grad: "Beograd",
      država: "Srbija"
    }
  ]
}
```

*Veza 1:N ostvarena uz pomoć
ugnježdenih dokumenata*

```
{
  _id: "orm",
  naziv: "O'Reilly Media",
  godina_osnivanja: 1980,
  država: "SAD",
  knjige: [123456789, 234567890]
}

{
  _id: 123456789,
  naslov: "MongoDB: The Definitive  
Guide",
  autor: [ "Kristina Chodorow", "Mike  
Dirolf" ],
  broj_strana: 216,
  jezik: "Engleski"
}

{
  _id: 234567890,
  naslov: "50 Tips and Tricks for MongoDB  
Developer",
  autor: "Kristina Chodorow",
  broj_strana: 68,
  jezik: "Engleski"
}
```

*Veza 1:N ostvarena
referenciranjem dokumenata*



MongoDB

- Pruža bogat skup opcija indeksiranja radi optimizovanja različitih vrsta upita
- Osim standardnih indeksa nad jednim poljem, ovaj proizvod nudi i sledeće vrste indeksa:
 - složene (*Compound*)
 - tekstualne (*Text*)
 - geoprostorne (*Geospatial*)
 - indekse za polja koja sadrže nizove (*Multikey*)
 - indekse koji omogućavaju upravljanje životnim vekom podataka (*Time to Live Indexes*)
 - jedinstvene indekse (*Unique Indexes*)
- Sposobnost skladištenja fajlova bilo koje vrste i veličine
- *Aggregation Framework* – analiza podataka u samoj bazi



MEAN i MERN

- *MEAN* i *MERN* full stack tehnologije u osnovi imaju *Mongo DB* kao nerelacionu bazu, ali bi ta komponenta mogla da bude zamenjena i relacionom (kada je potrebno).
- *NodeJS* treba da instalira i učitava potrebnu biblioteku, kao što se u svim veb aplikacijama preko drajvera aplikacija konektuje na samu bazu.
- Instalacija:
 - `npm install mongodb --save`
 - `npm install mysql --save`
- Korišćenje:
 - `var mongo = require('mongodb');`
 - `var mysql = require('mysql');`



NodeJS + Mongoose

RAD SA NERELACIONOM BAZOM



Konekcija sa mongoose

- Moramo da znamo putanju do mongodb i naziv baze
- Pokrenuti Mongo konekciju! (iz MongoDB Compass ili Robo 3T)
- Primer:

```
const mongoose = require('mongoose');
async function connectDB() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza', {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('Uspesna konekcija na bazu preko Mongoose-a!');
  } catch (err) {
    console.error('Greška prilikom konekcije:', err.message);
  }
}
connectDB();
```



Konekcija sa MongoDB driver

- Primer:

```
const { MongoClient } = require('mongodb');
const url = 'mongodb://localhost:27017';
const dbName = 'mojabaza';

async function connectDB() {
  const client = new MongoClient(url);

  try {
    await client.connect();
    console.log('Uspesna konekcija na bazu preko MongoClient-a!');
    const db = client.db(dbName);
    // možeš odmah raditi db.collection('users').find()
  } catch (err) {
    console.error('Greška prilikom konekcije:', err.message);
  } finally {
    await client.close();
  }
}

connectDB();
```



Kreiranje kolekcije pomoću scheme i modela

```
const mongoose = require('mongoose');
// 1. Konekcija na MongoDB
async function connectDB() {
  try {
    await
      mongoose.connect('mongodb://localhost:27017/mojabaza', {
        useNewUrlParser: true,
        useUnifiedTopology: true,
      });
    console.log('Uspesna konekcija na bazu!');
  } catch (err) {
    console.error('Greška prilikom konekcije:', err.message);
  }
}

// 2. Definicija šeme i modela (npr. za korisnike)
const korisnikSchema = new mongoose.Schema({
  ime: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  datumKreiranja: { type: Date, default: Date.now }
});
```

```
const Korisnik = mongoose.model('Korisnik', korisnikSchema);

// 3. Kreiranje kolekcije implicitno preko modela
async function init() {
  await connectDB();

  try {
    // Ova naredba će kreirati kolekciju ako ne postoji
    await Korisnik.init();
    console.log('Kolekcija je uspesno kreirana preko Mongoose!');
  } catch (err) {
    console.error('Greška prilikom kreiranja kolekcije:',
      err.message);
  } finally {
    await mongoose.disconnect();
  }
}

init();
```



Ubacivanje podataka u kolekciju sa *save()*

- Mongoose-u nema direktno insertOne i insertMany funkcija, kao u prirodnom MongoDB driveru, ali postoje ekvivalentne metode:
 - new Model(doc).save() → ekvivalent insertOne
 - Model.insertMany(docs) → ekvivalent insertMan

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

const korisnikSchema = new mongoose.Schema({
  ime: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  datumKreiranja: { type: Date, default: Date.now }
});

const Korisnik = mongoose.model('Korisnik', korisnikSchema);

async function main() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza', {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('Uspesna konekcija na bazu!');
```

```
// insertOne ekvivalent
const noviKorisnik = new Korisnik({
  ime: 'Petar Petrović',
  email: 'petar@example.com'
});

const sacuvan = await noviKorisnik.save();
console.log('Ubacen jedan korisnik:', sacuvan);

} catch (err) {
  console.error('Greška:', err.message);
} finally {
  await mongoose.disconnect();
}

main();
```



Ubacivanje podataka u kolekciju sa *insertMany()*

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

const korisnikSchema = new mongoose.Schema({
  ime: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  datumKreiranja: { type: Date, default: Date.now }
});

const Korisnik = mongoose.model('Korisnik', korisnikSchema);

async function main() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza', {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('Uspesna konekcija na bazu!');
```

```
// insertMany ekvivalent
const korisnici = [
  { ime: 'Ana Anić', email: 'ana@etf.rs' },
  { ime: 'Marko Marković', email: 'marko@etf.rs' }
];

const ubaceni = await Korisnik.insertMany(korisnici);
console.log('Ubaceno više korisnika:', ubaceni);

} catch (err) {
  console.error('Greška:', err.message);
} finally {
  await mongoose.disconnect();
}

main();
```



Dohvatanje podataka - findOne

```
const mongoose = require('mongoose');
const korisnikSchema = new mongoose.Schema({
  ime: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  datumKreiranja: { type: Date, default: Date.now }
});
const Korisnik = mongoose.model('Korisnik', korisnikSchema);

async function main() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza');

    // Dohvati jednog korisnika na osnovu email-a
    const korisnik = await Korisnik.findOne({ email: 'ana@example.com' });

    if (korisnik) {
      console.log('Pronadjen korisnik:', korisnik);
    } else {
      console.log('Nije pronadjen korisnik sa datim email-om.');
    }

  } catch (err) {
    console.error('Greška:', err.message);
  } finally {
    await mongoose.disconnect();
  }
}
main();
```



Dohvatanje podataka - find

```
const mongoose = require('mongoose');
const korisnikSchema = new mongoose.Schema({
  ime: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  datumKreiranja: { type: Date, default: Date.now }
});

const Korisnik = mongoose.model('Korisnik', korisnikSchema);

async function main() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza');

    // Dohvati sve korisnike
    const sviKorisnici = await Korisnik.find();
    console.log('Lista svih korisnika:', sviKorisnici);

    // Dohvati sve korisnike sa određenim uslovom
    const samoAna = await Korisnik.find({ ime: 'Ana Anić' });
    console.log('Samo Ana:', samoAna);

  } catch (err) {
    console.error('Greška:', err.message);
  } finally {
    await mongoose.disconnect();
  }
}

main();
```



Dohvatanje podataka - različiti filteri (1)

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

const korisnikSchema = new mongoose.Schema({
  ime: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  godine: { type: Number, required: true },
  aktivan: { type: Boolean, default: true },
  datumKreiranja: { type: Date, default: Date.now }
});

const Korisnik = mongoose.model('Korisnik', korisnikSchema);

async function main() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza');

    // 1. Dohvati korisnike sa tačnim imenom
    const ana = await Korisnik.find({ ime: 'Ana Anić' });
    console.log('Ana:', ana);

    // 2. Dohvati korisnike starije od 25 godina
    const stariji = await Korisnik.find({ godine: { $gt: 25 } });
    console.log('Stariji od 25:', stariji);

    // 3. Dohvati korisnike čije godine su između 20 i 30
    const mladi = await Korisnik.find({ godine: { $gte: 20, $lte: 30 } });
    console.log('Godine između 20 i 30:', mladi);
  }
}
```

Operatori u filterima	
\$gt	veće od
\$gte	veće ili jednako
\$lt	manje od
\$lte	manje ili jednako
\$ne	nije jednako
\$in	vrednost je u listi
\$nin	vrednost nije u listi
Regex	pretraga stringa po obrascu



Dohvatanje podataka - različiti filteri (2)

```
// 4. Dohvati korisnike koji su aktivni
const aktivni = await Korisnik.find({ aktivan: true });
console.log('Aktivni korisnici:', aktivni);

// 5. Kombinovani uslovi (npr. aktivni i stariji od 25)
const aktivniStariji = await Korisnik.find({ aktivan: true, godine: { $gt: 25 } });
console.log('Aktivni stariji od 25:', aktivniStariji);

// 6. Pretraga sa regularnim izrazom (ime koje počinje na "A")
const regexPretraga = await Korisnik.find({ ime: /^A/ });
console.log('Imena koja počinju na A:', regexPretraga);

// 7. Dohvati samo određena polja (projection)
const samoImeEmail = await Korisnik.find({}, 'ime email');
console.log('Samo ime i email:', samoImeEmail);

} catch (err) {
  console.error('Greška:', err.message);
} finally {
  await mongoose.disconnect();
}
}

main();
```



Sortiranje i paginacija

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

const korisnikSchema = new mongoose.Schema({
  ime: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  godine: { type: Number, required: true },
  aktivan: { type: Boolean, default: true },
  datumKreiranja: { type: Date, default: Date.now }
});

const Korisnik = mongoose.model('Korisnik', korisnikSchema);

async function main() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza');

    // Primer paginacije
    const page = 2; // željena stranica
    const limit = 3; // broj rezultata po stranici
    const skip = (page - 1) * limit;

    const korisnici = await Korisnik.find({ aktivan: true })
      .sort({ godine: -1 }) // sortiraj po godinama (opadajuće)
      .skip(skip) // preskoči rezultate sa prethodnih stranica
      .limit(limit) // uzmi samo "limit" rezultata
      .select('ime godine email'); // vrati samo ova polja

    console.log(`Stranica ${page}:`, korisnici);
```

```
// Ukupan broj aktivnih korisnika (za izračunat broja stranica)
const totalCount = await Korisnik.countDocuments({ aktivan: true });
const totalPages = Math.ceil(totalCount / limit);
console.log(`Ukupno stranica: ${totalPages}`);

} catch (err) {
  console.error('Greška:', err.message);
} finally {
  await mongoose.disconnect();
}

main();
```

- **sort({ godine: -1 })** - sortiranje po godinama, od najvećih ka najmanjim (1 za rastuće, -1 za opadajuće)
- **skip((page - 1) * limit)** - preskoči rezultate iz prethodnih stranica
- **limit(max)** - ograniči broj rezultata po stranici
- **countDocuments()** - izračunava ukupan broj dokumenata da bi znao koliko ima stranica



Brisanje zapisa (dokumenta) - *deleteOne*

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

const korisnikSchema = new mongoose.Schema({
  ime: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  godine: { type: Number, required: true },
  aktivan: { type: Boolean, default: true },
  datumKreiranja: { type: Date, default: Date.now }
});

const Korisnik = mongoose.model('Korisnik', korisnikSchema);

async function main() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza');

    //Brisanje jednog korisnika prema email adresi
    const resultOne = await Korisnik.deleteOne({ email: "pera@example.com" });
    console.log("Rezultat deleteOne:", resultOne);
    // { acknowledged: true, deletedCount: 1 }
```



Brisanje više zapisa - *deleteMany*

```
//Brisanje svih korisnika mlađih od 18 godina
const resultMany = await Korisnik.deleteMany({ godine: { $lt: 18 } });
console.log("Rezultat deleteMany:", resultMany);
// { acknowledged: true, deletedCount: N }

} catch (err) {
  console.error("Greška:", err.message);
} finally {
  await mongoose.disconnect();
}
}
```

- `deleteOne(filter)` – briše samo prvi dokument koji ispunjava uslov
- `deleteMany(filter)` – briše sve dokumente koji ispunjavaju uslov
- Rezultat u oba slučaja vraća objekat sa poljima:
 - `acknowledged: true` - znači da je server potvrdio operaciju
 - `deleteCount` - broj obrisanih dokumenata



Brisanje cele kolekcije - *drop*

```
const mongoose = require('mongoose');
const korisnikSchema = new mongoose.Schema({
  ime: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  godine: { type: Number, required: true },
  aktivan: { type: Boolean, default: true },
  datumKreiranja: { type: Date, default: Date.now }
});
const Korisnik = mongoose.model('Korisnik', korisnikSchema);

async function main() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza');

    // Brisanje cele kolekcije "korisnici"
    await Korisnik.collection.drop();
    console.log("Kolekcija 'korisnici' je obrisana!");
  } catch (err) {
    if (err.code === 26) {
      console.log("Kolekcija ne postoji, pa nije mogla biti obrisana.");
    } else {
      console.error("Greška:", err.message);
    }
  }
}
```

```
  } finally {
    await mongoose.disconnect();
  }
}

main();
```

- **Korisnik.collection.drop()** - briše kompletnu kolekciju sa svim dokumentima i indeksima.
- Ako kolekcija ne postoji, MongoDB baca grešku sa kodom 26, pa je zato dobro uhvatiti i taj slučaj.
- Za razliku od **deleteMany({})** koji briše sve dokumente, ali kolekcija i dalje postoji, **drop()** briše i samu kolekciju.



Ažuriranje zapisa - *updateOne*

- Kada ažuriramo jedan dokument, koristimo funkciju **updateOne**, koja kao parametre očekuje dve vrednosti: upit-objekat koji menjamo (param. 1) i objekat koji sadrži nove vrednosti (param. 2)

```
const mongoose = require('mongoose');
const korisnikSchema = new mongoose.Schema({
  ime: String,
  email: String,
  godine: Number,
  aktivan: Boolean
});
const Korisnik = mongoose.model('Korisnik', korisnikSchema);

async function main() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza');

    //Ažurira jednog korisnika po email adresi
    const result = await Korisnik.updateOne(
      { email: "pera@example.com" },          // filter
      { $set: { aktivan: false, godine: 30 } } // update
    );

    console.log("Rezultat updateOne:", result);
  } catch (err) {
    console.error("Greška:", err.message);
  } finally {
    await mongoose.disconnect();
  }
}

main();
```



Ažuriranje više zapisa - *updateMany*

- Koristimo metodu **updateMany** i ponovo operator **\$set**

```
async function updateAllInactive() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza');

    // Ažurira sve korisnike starije od 25 godina, postavlja aktivan = false
    const result = await Korisnik.updateMany(
      { godine: { $gt: 25 } },          // filter
      { $set: { aktivan: false } }     // update
    );

    console.log("Rezultat updateMany:", result);
  } catch (err) {
    console.error("Greška:", err.message);
  } finally {
    await mongoose.disconnect();
  }
}

updateAllInactive();
```

- **updateOne()** - menja prvi dokument koji odgovara filteru.
- **updateMany()** - menja sve dokumente koji odgovaraju filteru.
- Rezultat koji se vraća sadrži:
 - **acknowledged: true**
 - **matchedCount** (koliko dokumenata je pronađeno)
 - **modifiedCount** (koliko dokumenata je zaista izmenjeno)



Upiti sa limit i sort funkcijama

- Kada unutar SELECT želimo da dobijemo samo nekoliko dokumenata (zapisa), koristimo LIMIT

```
const mongoose = require('mongoose');
const korisnikSchema = new mongoose.Schema({
  ime: String,
  email: String,
  godine: Number,
  aktivan: Boolean
});
const Korisnik = mongoose.model('Korisnik', korisnikSchema);

async function main() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza');
    const korisnici = await Korisnik.find({ aktivan: true })
      .limit(3) // ograniči rezultat na najviše 3 dokumenta
      .sort({ godine: -1 }); // opciono: sortiraj po godinama opadajuće

    console.log('Najviše 3 aktivna korisnika:', korisnici);
  } catch (err) {
    console.error('Greška:', err.message);
  } finally {
    await mongoose.disconnect();
  }
}
main();
```



Upiti sa limit i skip funkcijama za paginaciju

```
const mongoose = require('mongoose');
const korisnikSchema = new mongoose.Schema({
  ime: String,
  email: String,
  godine: Number,
  aktivan: Boolean
});
const Korisnik = mongoose.model('Korisnik', korisnikSchema);
async function main() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza');
    const page = 2;      // druga stranica
    const limit = 3;      // broj dokumenata po stranici
    const skip = (page - 1) * limit;
    const korisnici = await Korisnik.find({ aktivan: true })
      .sort({ godine: -1 }) // sortiranje po godinama opadajuće
      .skip(skip)           // preskoči dokumente sa prethodnih stranica
      .limit(limit);        // uzmi samo "limit" dokumenata
    console.log(`Stranica ${page}:`, korisnici);
  } catch (err) {
    console.error('Greška:', err.message);
  } finally {
    await mongoose.disconnect();
  }
}
main();
```

- **skip(n)** - preskače n dokumenata (koristi se za prelazak na sledeću stranicu).
- **limit(n)** - vraća maksimalno n dokumenata po stranici.
- Kombinacija **sort + skip + limit** daje standardnu paginaciju sa opcionalnim sortiranje.



Spajanje - korak 1) Definišemo šeme

- U Mongoose-u se koristi referenciranje dokumenata i metoda populate(), koja nam omogućava da „spojimo“ dokument iz druge kolekcije.
- Primer sa dve kolekcije: **Korisnici** i **Komentari**
- **ref: 'Komentar'** govori Mongoose-u da je ovo **referenca** na dokument u kolekciji **Komentar**.

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

```
const KomentarSchema = new mongoose.Schema({  
  text: String,  
  datum: { type: Date, default: Date.now }  
});
```

```
const KorisnikSchema = new mongoose.Schema({  
  ime: String,  
  email: String,  
  komentari: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Komentar' }] // reference  
});
```

```
const Komentar = mongoose.model('Komentar', KomentarSchema);
```

```
const Korisnik = mongoose.model('Korisnik', KorisnikSchema);
```



Spajanje - korak 2) Ubacivanje primera podataka

```
async function main() {
  await mongoose.connect('mongodb://localhost:27017/mojabaza');

  // Kreiraj komentare
  const k1 = await new Komentar({ text: 'Prvi komentar' }).save();
  const k2 = await new Komentar({ text: 'Drugi komentar' }).save();

  // Kreiraj korisnika i poveži komentare
  const korisnik = new Korisnik({
    ime: 'Petar',
    email: 'petar@example.com',
    komentari: [k1._id, k2._id]
  });

  await korisnik.save();
  console.log('Korisnik sa komentarima kreiran!');
  await mongoose.disconnect();
}

main();
```



Spajanje - korak 3) Dohvatanje podataka sa „join“

- **Mongoose ne koristi striktni SQL join, već referenciranje i populate.**
- **.populate('komentari')** - automatski zamenjuje `_id` polja sa pravim dokumentima iz kolekcije Komentar.
- **\$lookup** se koristi u MongoDB agregacionom radnom okviru, dok *populate* pruža jednostavniji API za klasične „relacije“ u aplikacijama.

```
async function fetchKorisnik() {  
  await mongoose.connect('mongodb://localhost:27017/mojabaza');  
  
  const korisnik = await Korisnik.findOne({ ime: 'Petar' }).populate('komentari');  
  console.log('✂ Korisnik sa komentarima:', korisnik);  
  
  await mongoose.disconnect();  
}  
  
fetchKorisnik();
```



Primer agregacije sa \$lookup

- Šeme iste kao u koraku 1) prethodnog primera.

```
async function fetchKorisniciSaKomentarima() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mojabaza');
    const rezultat = await Korisnik.aggregate([
      {
        $lookup: {
          from: "komentars",          // ime kolekcije u bazi
          localField: "komentari",    // polje u Korisnik dokumentu koje sadrži _id-jeve
          foreignField: "_id",        // polje u kolekciji komentara koje se spaja
          as: "komentariInfo"         // novo polje koje će sadržati spajanje
        }
      },
      {
        $project: { ime: 1, email: 1, komentariInfo: 1 } // opcionalno: izaberemo polja
      }
    ]);
    console.log("Korisnici sa komentarima:", JSON.stringify(rezultat, null, 2));
  } catch (err) {
    console.error("Greška:", err.message);
  } finally {
    await mongoose.disconnect();
  }
}

fetchKorisniciSaKomentarima();
```

- **from** - ime kolekcije sa kojom spajamo (*Komentar* kolekcija u MongoDB-u se automatski pluralizuje u *komentars*).
- **localField** - polje u trenutnoj kolekciji koje sadrži reference (*komentari*).
- **foreignField** - polje u drugoj kolekciji koje se koristi za spajanje (*_id*).
- **as** - ime polja u izlaznom dokumentu, gde će se smestiti spojeni podaci.
- Rezultat je niz dokumenata, gde svaki korisnik sada ima polje *komentariInfo* koje sadrži stvarne komentare.



NodeJS + MySQL

RAD SA RELACIONOM BAZOM



Kreiranje konekcije pomoću mysql2

- Koristi se paket mysql2, koji podržava *Promise* / *async* / *await*

```
// Instalirati: npm install mysql2
const mysql = require('mysql2/promise'); // koristimo promise verziju
async function main() {
  let con;
  try {
    con = await mysql.createConnection({ //vraća Promise, pa se koristi await
      host: "localhost",
      user: "korime",
      password: "lozinka",
      database: "mojaBaza" // opciono
    });
    console.log("Uspesno ste se konektovali!");
    // Primer jednostavnog upita
    const [rows, fields] = await con.execute('SELECT NOW() AS sada'); //vraća [rows, fields], lakše za obradu rezultata
    console.log("Trenutno vreme:", rows[0].sada);
  } catch (err) {
    console.error("Greška:", err.message);
  } finally {
    if (con) await con.end(); //zatvara konekciju (uvek u finally bloku)
  }
}
main();
```



SELECT upiti u MySQL

- Svaka naredba zove se upit, i nad konektovanom bazom može da se izvrši (MySQL ima podršku za CRUD upite). Nakon izvršenog upita, dobijamo rezultat (MySQL result kao povratnu vrednost).

```
// Instalirati: npm install mysql2
const mysql = require('mysql2/promise');
async function main() {
  let con;
  try {
    //Konekcija ka bazi
    con = await mysql.createConnection({
      host: "localhost",
      user: "korime",
      password: "lozinka",
      database: "mojaBaza"
    });
    console.log("Konektovani na bazu!");
    //Jednostavan SELECT upit
    const [rows, fields] = await con.execute('SELECT * FROM korisnici');
    console.log("Rezultati upita:");
    rows.forEach((row, index) => {
      console.log(`${index + 1}. Ime: ${row.ime}, Email: ${row.email}, Godine: ${row.godine}`);
    });
  } catch (err) {
    console.error(err);
  }
}
```

- await con.execute(query, params)**
 - query => SQL upit
 - params => niz vrednosti koje se bezbedno ubacuju u upit (? služi kao placeholder)
- [rows, fields]**
 - rows => niz rezultata (objekti sa poljima iz tabele)
 - fields => metapodaci kolona (ne mora da se koristi često)



Parametrizovani SELECT upiti u MySQL

- **Parametrizovani upiti**
 - Sprečavaju SQL injection
 - Primer: 'SELECT * FROM korisnici WHERE email = ?' i [userEmail]

```
// Parametrizovani SELECT (sigurnije za korisnički unos)
const userEmail = 'pera@example.com';
const [userRows] = await con.execute('SELECT * FROM korisnici WHERE email = ?', [userEmail]);

console.log("Rezultat parametrizovanog upita:", userRows);

} catch (err) {
  console.error("Greška:", err.message);
} finally {
  if (con) await con.end();
}

main();
```





Kreiranje nove baze iz Node skripte

```
// Instalirati: npm install mysql2
const mysql = require('mysql2/promise');

async function main() {
  let con;
  try {
    // Konekcija ka MySQL serveru (bez selektovane baze)
    con = await mysql.createConnection({
      host: "localhost",
      user: "korime",
      password: "lozinka"
    });
    console.log("Konektovani na MySQL server!");

    // Kreiranje baze
    const dbName = "mojaNovaBaza";
    await con.execute(`CREATE DATABASE IF NOT EXISTS \`${dbName}\``);
    console.log(`Baza podataka "${dbName}" je kreirana (ili već postoji)!`);
  } catch (err) {
    console.error("Greška:", err.message);
  } finally {
    if (con) await con.end();
  }
}
main();
```



Kreiranje nove tabele iz Node skripte

```
// Uvoz mysql2 paketa u promise verziji

const mysql = require('mysql2/promise');
async function main() {
  // Kreiranje konekcije sa serverom
  const connection = await mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'tvoja_lozinka' // promeniti lozinku
  });
  try {
    // Kreiranje nove baze podataka (ako ne postoji)
    await connection.query(`CREATE DATABASE IF NOT EXISTS moja_baza`);
    console.log('Baza je kreirana ili već postoji.');
```

```
    // Koriscenje te baze
    await connection.query(`USE moja_baza`);
    // Kreiranje nove tabele
    const createTableQuery = `
      CREATE TABLE IF NOT EXISTS studenti (
        id INT AUTO_INCREMENT PRIMARY KEY,
        ime VARCHAR(50) NOT NULL,
        prezime VARCHAR(50) NOT NULL,
        godina_studija INT
      )
    `;
```

```
    await connection.query(createTableQuery);
    console.log('Tabela "studenti" je kreirana ili već postoji.');
```

```
  } catch (err) {
    console.error('Došlo je do greške:', err);
  } finally {
    // Zatvaranje konekcije
    await connection.end();
  }
}
// Pokretanje funkcije
main();
```



Zakasnelo dodavanje kolone

```
// Uvoz mysql2 paketa u promise verziji
const mysql = require('mysql2/promise');
async function main() {
  const connection = await mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: '',
    database: 'ime_moje_baze',
  });
  try {
    // Dodavanje nove kolone "email" u tabelu "studenti"
    const alterTableQuery = `
      ALTER TABLE studenti
      ADD COLUMN email VARCHAR(100) AFTER prezime
    `;

    await connection.query(alterTableQuery);
    console.log('Kolona "email" je uspešno dodata u tabelu "studenti".');
  } catch (err) {
    console.error('Došlo je do greške prilikom ALTER TABLE:', err);
  } finally {
    await connection.end();
  }
}
main();
```



Ubacivanje podatka u bazu (INSERT)

```
const mysql = require('mysql2/promise');

async function main() {
  const connection = await mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'lozinka',
    database: 'moja_baza'
  });

  try {
    // --- Ubacivanje jednog reda ---
    const insertOneQuery = `
      INSERT INTO studenti (ime, prezime, godina_studija, email)
      VALUES (?, ?, ?, ?)
    `;

    await connection.query(insertOneQuery, ['Marko', 'Marković', 2, 'marko@example.com']);
    console.log('Jedan red je ubačen.');
```

```

  // --- Ubacivanje više redova odjednom ---
  const insertMultipleQuery = `
    INSERT INTO studenti (ime, prezime, godina_studija, email)
    VALUES ?
  `;
```

```
const values = [
  ['Ana', 'Anić', 1, 'ana@example.com'],
  ['Jovan', 'Jovanović', 3, 'jovan@example.com'],
  ['Mila', 'Milić', 2, 'mila@example.com']
];

await connection.query(insertMultipleQuery, [values]);
console.log('Više redova je ubačeno.');
```

```

} catch (err) {
  console.error('Došlo je do greške prilikom INSERT:', err);
} finally {
  await connection.end();
}

main();
```



Filter WHERE i ORDER BY sortiranje

```
const mysql = require('mysql2/promise');

async function main() {
  const connection = await mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'lozinka',
    database: 'moja_baza'
  });

  try {
    // SELECT ime i email gde je godina_studija = 2,
    // sortirano po imenu
    const selectQuery = `
      SELECT ime, email
      FROM studenti
      WHERE godina_studija = ?
      ORDER BY ime ASC
    `;
  }
}
```

```
const [rows, fields] = await connection.query(selectQuery, [2]);

console.log('Rezultati upita:');
rows.forEach(row => {
  console.log(`Ime: ${row.ime}, Email: ${row.email}`);
});

} catch (err) {
  console.error('Došlo je do greške prilikom SELECT:', err);
} finally {
  await connection.end();
}

main();
```



Brisanje podataka (DELETE) i brisanje tabele (DROP)

```
const mysql = require('mysql2/promise');
```

```
async function main() {
```

```
  const connection = await mysql.createConnection({
```

```
    host: 'localhost',
```

```
    user: 'root',
```

```
    password: 'lozinka',
```

```
    database: 'moja_baza'
```

```
  });
```

```
  try {
```

```
    // --- Brisanje jednog reda ---
```

```
    const deleteQuery = `
```

```
      DELETE FROM studenti
```

```
      WHERE ime = ? AND prezime = ?
```

```
    `;
```

```
    const [deleteResult] = await connection.query(deleteQuery, ['Marko', 'Marković']);
```

```
    console.log(`Broj obrisanih redova: ${deleteResult.affectedRows}`);
```

```
// --- Brisanje cele tabele ---
```

```
const dropTableQuery = `
```

```
  DROP TABLE IF EXISTS studenti
```

```
`;
```

```
await connection.query(dropTableQuery);
```

```
console.log('Tabela "studenti" je obrisana.');
```

```
} catch (err) {
```

```
  console.error('Došlo je do greške:', err);
```

```
} finally {
```

```
  await connection.end();
```

```
}
```

```
}
```

```
main();
```



Ažuriranje podataka (UPDATE)

```
const mysql = require('mysql2/promise');

async function main() {
  const connection = await mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'lozinka',
    database: 'moja_baza'
  });

  try {
    // --- UPDATE jednog reda ---

    const updateOneQuery = `
      UPDATE studenti
      SET email = ?
      WHERE ime = ? AND prezime = ?
    `;

    const [updateOneResult] = await connection.query(updateOneQuery, ['marko.novimejl@example.com', 'Marko', 'Jarić']);
    console.log(`Broj izmenjenih redova (jedan red): ${updateOneResult.affectedRows}`);

    // --- UPDATE više redova odjednom ---

    const updateMultipleQuery = `
      UPDATE studenti
      SET godina_studija = ?
      WHERE godina_studija = ?
    `;

    const [updateMultipleResult] =
      await connection.query(updateMultipleQuery, [3, 2]);
    console.log(`Broj izmenjenih redova (više redova):
      ${updateMultipleResult.affectedRows}`);

  } catch (err) {
    console.error('Došlo je do greške prilikom UPDATE:', err);
  } finally {
    await connection.end();
  }
}

main();
```



Spajanje tabela (JOIN)

```
const mysql = require('mysql2/promise');
async function main() {
  const connection = await mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'lozinka',
    database: 'moja_baza'
  });

  try {    // JOIN tabela studenti i predmeti po student_id
    const joinQuery = `
      SELECT s.ime, s.prezime, p.naziv AS predmet FROM studenti s
      INNER JOIN predmeti p ON s.id = p.student_id
      ORDER BY s.ime ASC
    `;

    const [rows, fields] = await connection.query(joinQuery);
    console.log('Rezultati JOIN upita:');
    rows.forEach(row => {
      console.log(`Student: ${row.ime} ${row.prezime}, Predmet: ${row.predmet}`);
    });
  } catch (err) {
    console.error('Došlo je do greške prilikom JOIN:', err);
  } finally {
    await connection.end();
  }
}
main();
```



Zaključak

- Node možemo povezivati sa kojom god želimo bazom
- Najčešće su korićene u kombinaciji sa Node:
 - MySQL (relaciona)
 - PostgreSQL (relaciona)
 - MongoDB (nerelaciona, po dokumentima)
 - Redis (nerelaciona, ključ-vrednost)
 - SQL Lite (relaciona)



Hvala na pažnji 😊

PITANJA?