NoSQL and MongoDB

NoSQL vs SQL, MongoDB, Mongoose

Table of Contents

Have a Question?

#js-web

Relational and NoSQL Databases

Differences and Examples

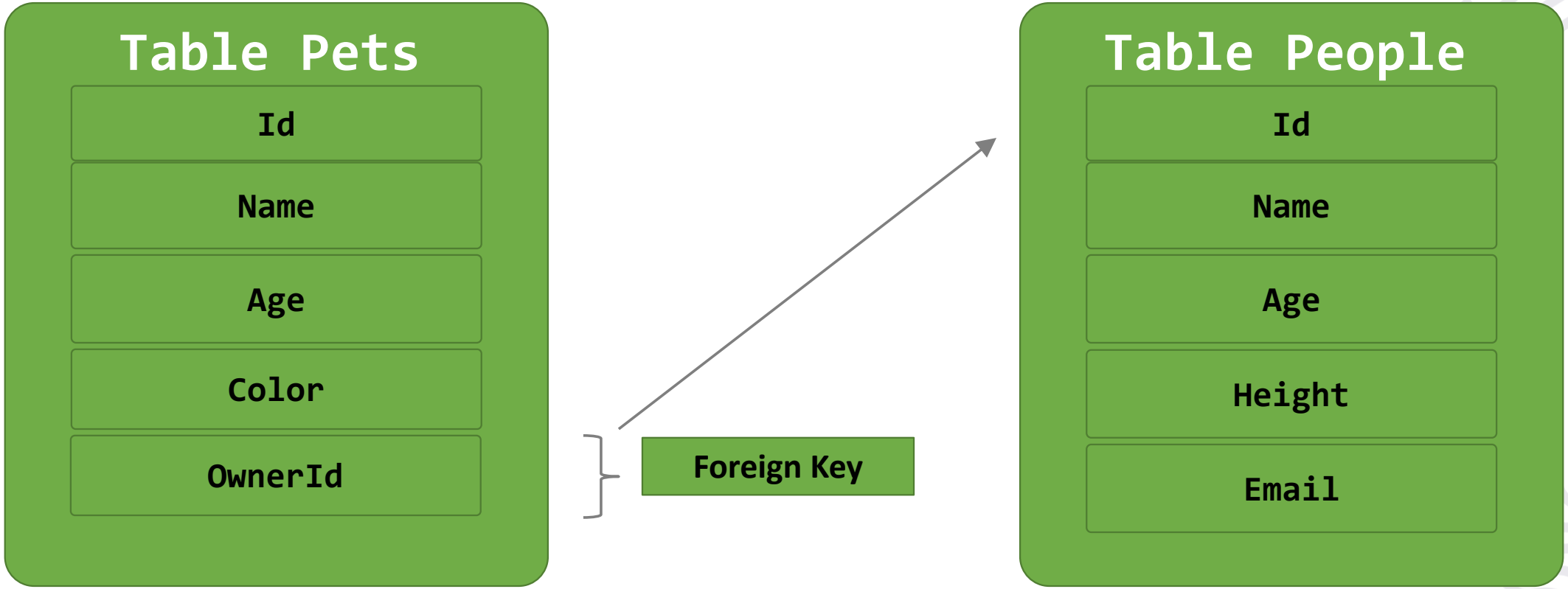
Relational Database

- Organize data into one or more **tables** of **columns** and **rows**
- Unique **key** identifying each **row** of data
- Almost all relational databases use **SQL** to **extract** data

```
SELECT * FROM Students
```

- **Relations** between tables are done using **Foreign Keys (FK)**
- Such databases are **Oracle**, **MySQL**, **SQL Server**, etc..

Relational Database – Example



Non-relational Database (NoSQL)

- Key-value **stores**

```
{
    "_id": ObjectId("59d3fe7ed81452db0933a871"),
    "email": "peter@gmail.com",
    "age": 22
}
```

- **SQL** query is **not** used in NoSQL systems
- More **scalable** and **provide** superior **performance**
- Such databases are **MongoDB**, **Cassandra**, **Redis**, etc..

MongoDB Overview

Installation, Configuration, Startup

Install MongoDB

- Download from: https://www.mongodb.com/download-center
- When **installed**, MongoDB needs a **driver**
  - One to use with Node.js, .NET, Java, etc..
  - Install MongoDB **driver** for Node.js

```
npm install mongodb -g
```

Configure MongoDB

- Additional configurations are **needed**
  - Go to installation folder and **run** a command prompt as an **administrator**
  - Type the following command

Usually in **C:\Program Files\MongoDB\Server\3.4\bin**

```
<path to mongod.exe> mongod --dbpath <path to store data>
```

  - Additional information at https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/

Working with MongoDB Shell Client

- Start the shell from **another** CLI
  - Type the command **mongo**

```
show dbs
```

```
use mytestdb
```

```
db.mycollection.insert({"name":"George"})
```

```
db.mycollection.find({"name":" George"})
```

```
db.mycollection.find({})
```

  - Additional information at
    - https://docs.mongodb.com/manual/reference/mongo-shell/

Working with MongoDB GUI

- Choose one of the many
- For example
  - Robo 3T- https://robomongo.org/download
  - NoSQLBooster- https://nosqlbooster.com

Working with MongoDB from Node.js – Example

```
const mongodb = require('mongodb');
const MongoClient = mongodb.MongoClient;
const connectionStr = 'mongodb://localhost:27017';
const client = new MongoClient(connectionStr);
client.connect(function(err) {
  const db = client.db('testdb');
  const people = db.collection('people');

  people.insert({ 'name': 'Pavel' }, (err, result) => {
    people.find({ name: 'Ivan' }).toArray((err, data) => {
      console.log(data);
    });
  });
});
```

Mongoose Overview

Installation, Models, Schema

Mongoose Overview

- Mongoose is an object-document **model** module in Node.js for MongoDB
  - It **provides** a straight-forward, **schema-based** solution to **model** your application data
  - Includes build-in type **casting** and **validation**
  - **Extends** the native **queries** (much **easier** to use)
  - To **install** type in CMD

```
npm install mongoose --g
```

Working with Mongoose in Node.js

- Load the following module

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

- Connecting to the database

```
mongoose.connect('mongodb://localhost:27017/unidb')
```

MongoDB Hosting

- Host a **database** in the largest MongoDB **cloud** service
- Go to '**mLab**' and register - https://mlab.com/
- You can **store** up to 500 MB of **content**

Mongoose Models

Constructor, Virtual Properties, Validation

Mongoose Models

- Mongoose **supports** models
  - Fixed **types** of documents
    - Used like object **constructors**
  - Needs a **mongoose.Schema** call

```
const modelSchema = new mongoose.Schema({
  propString: String,
  propNumber: Number,
  propObject: {},
  propArray: [],
  propBool: Boolean
});

const Model = mongoose.model('Model', modelSchema);
```

Model Methods

- Since mongoose models are just JavaScript **object constructors,** they can have **methods**
  - And these methods can be **added** to a schema
    - Use a **different** syntax than plain JS

```
const studentSchema = new mongoose.Schema({…});

studentSchema.methods.getInfo = function() {
   return `I am ${this.firstName} ${this.lastName}`;
};
```

Avoid **arrow** functions

Model Virtual Properties

- Yet, not all properties **need** to be **persisted** to the **database**
- Mongoose provides a way to **create** properties, that are accessible on all models, but are **not persisted** to the database
  - And they have both **getters** and **setters**

```
studentSchema.virtual('fullName').get(function () {
  return this.firstName + ' ' + this.lastName
});
```

Property Validation

- With Mongoose developers can **define** custom **validation** on their **properties**
  - Validate records when trying to **save**

```
studentSchema.path('firstName')
  .validate(function () {
     return this.firstName.length >= 2
     && this.firstName.length <= 10
}, 'First name must be between 2 and 10 symbols long!')
```

Error message as **second** param

Exporting Modules

- Having all model definitions in the **main** module is **no** good
  - That is the reason Node.js has **modules** in the first place
  - In folder **models**, file **Student.js**

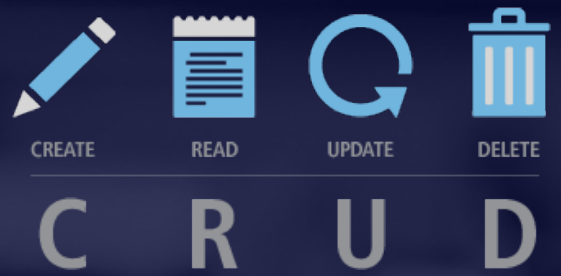
```
const mongoose = require('mongoose');
const studentSchema = new mongoose.Schema({
    firstName: { type: String, required: true },
    lastName: { type: String, required: true },
    facultyNumber: { type: String, required: true, unique: true },
    age: { type: Number }
});

module.exports = mongoose.model('Student');
```

Using Modules

- We can put each **model** in a different **module**, and **load** all models at start
  - Where it is needed

```
const Student = require('./models/Student');
```



CRUD with Mongoose

Create, Read, Update, Delete

CRUD with Mongoose

- Mongoose supports **all** CRUD operations
  - Create (Persist data)

```
new Student({}).save(callback)
```

  - Read (Extract data)

```
Student.find({})
```

CRUD with Mongoose

- Update (Modify data)

```
Student
  .findById(id, callback)
Student
  .findByIdAndUpdate(id, {$set: {prop: newVal}}, callback)
Student
  .update({_id: id, {$set: {prop: newVal}}, callback)
```

- Delete (Remove data)

```
Student.findByIdAndRemove(id, callback)
Student.remove({name: studentName})
```

Create Example

```
const mongoose = require('mongoose');
const connectionStr = 'mongodb://localhost:27017/unidb';
const studentSchema = new mongoose.Schema({
  name: { type: String, required: true, minlength: 3 },
  age: { type: Number }
});

const Student = mongoose.model('Student', studentSchema);
mongoose.connect(connectionStr).then(() => {
  new Student({ name: 'Petar', age: 21 })
    .save()
    .then(student => {
      console.log(student._id)
    });
});
```

You can also use Student.create()

Read Example

```
Student
    .find({})
    .then(students => console.log(students))
    .catch(err => console.error(err))
Student
    .find({name: 'Petar'})
    .then(students => console.log(students))
Student
    .findOne({name: 'Petar'})
    .then(student => console.log(student))
```

Always handle errors

Update Example

```
Student
    .findById('57fb9fe1853ab747b0f692d1')
    .then((student) => {
      student.firstName = 'Stamat'
      student.save()
    });
Student
    .findByIdAndUpdate('57fb9fe90cd76e4e2c59e1a2', {
      $set: { name: 'Stamat' }
    });
Student
    .update(
      { firstName: 'Kiril' },
      { $set: { name: 'Petar' } },
      { multi: true })
```

Update multiple entities

Remove & Count Example

```
Student
    .findByIdAndRemove('57fb9fe1853ab747b0f692d1')
Student
    .remove({ name: 'Stamat' })
Student
    .count()
    .then(console.log)
Student
    .count({ age: { $gt: 19 } })
    .then(console.log)
```

Remove by criteria

Get the count by criteria

Mongoose Queries

Chaining

Mongoose Queries

- Mongoose defines **all** queries of the native MongoDB driver in a more **clear** and **useful** way
  - Instead of

```
{
  $or: [
    {conditionOne: true},
    {conditionTwo: true}
  ]
}
```

  - Do

```
.where({ conditionOne: true }).or({ conditionTwo: true })
```

Mongoose Queries Example

- Mongoose supports **many** queries
  - For equality/non-equality

```
Student.findOne({'lastName':'Petrov'})
```

```
Student.find({}).where('age').gt(7).lt(14)
```

```
Student.find({}).where('facultyNumber').equals('12399')
```

  - Selection of some properties

```
Student.findOne({'lastName':'Kirilov'}).select('name age')
```

Mongoose Queries Example 2

- Sorting

```
Student.find({}).sort({age:-1})
```

- Limit & skip

```
Student.find({}).sort({age:-1}).skip(10).limit(10)
```

- Different methods could be **stacked** one upon the other

```
Student.find({}).where('firstName').equals('gosho').where('age').gt
(18).lt(65).sort({age:-1}).skip(10).limit(10)
```

Model Population

Reference Documents in Other Collections

Population Definition

- Population is the process of **automatically replacing** the **specified paths** in the document with document(s) from **other** collection(s)
- We may populate a single document, multiple documents, plain object, multiple plain objects, or all objects returned from a query

Example

- We create **two models** that **reference** each other

```
const studentSchema = new mongoose.Schema({
  name: String,
  age: Number,
  facultyNumber: String
  teacher: { type: Schema.Types.ObjectId, ref: 'Teacher' }
  subjects: [{ type: Schema.Types.ObjectId, ref: 'Subject' }]
});
const subjectSchema = new mongoose.Schema({
  title: String,
  students: [{ type: Schema.Types.ObjectId, ref: 'Student' }]
});
const Student = mongoose.model('Student', studentSchema);
const Subject = mongoose.model('Subject', subjectSchema);
```

Population

- To load all the data **referenced** with the entity use **populate()**

```
Student.findOne({ name: 'Peter' })
  .populate('subjects')
  .then(student => {
     console.log(student.subjects)
   })
```

Will return an array of **objects** and **NOT** Id's

- You can also load **multiple** paths

```
Student.findOne({ name: 'Peter' })
   .populate('subjects')
   .populate('teacher')
   .then(student => {
      console.log(student.teacher)
      console.log(student.subjects)
   })
```

Query Conditions

- Populate based on a **condition**

```
Subject.
  find({}).
  populate({
    path: 'students',
    match: { age: { $gte: 19 }},
    select: 'name facultyNumber',
    options: { limit: 3 }
  })
```

- More on populate here - mongoosejs.com/docs/populate.html

Summary

- NoSQL databases provide **superior** performance
- Mongoose gives us a **schema-based** solution

```
const modelSchema = new mongoose.Schema({
   propString: String
});
```

- Mongoose supports all **CRUD** operations
- Chaining queries with Mongoose is possible

```
Student.find({}).where('firstName').equals('gosho
')
.where('age').gt(18).lt(65).sort({age:1}).skip(10
)
.limit(10)
```

Questions?

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © Kingsland University – https://kingslanduniversity.com

KINGSLAND UNIVERSITY

THANK YOU