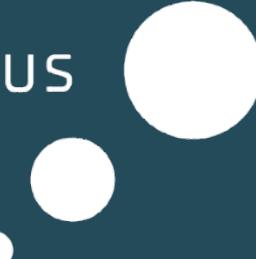


# SQL og NoSQL

## Opsamling



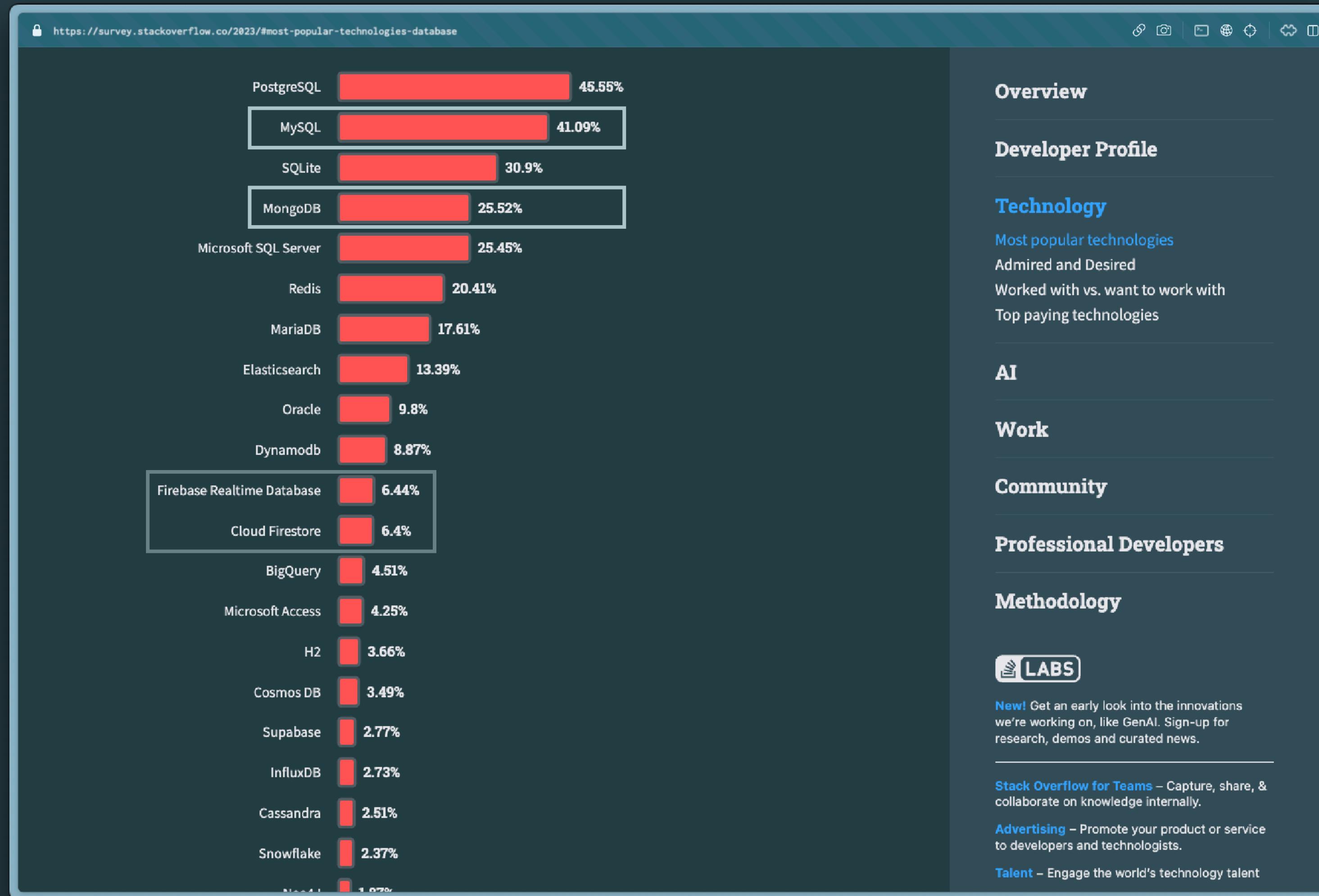
1. Opsamling
  1. SQL og NoSQL
  2. ORM og ODM
  3. Schema Design og Validering
  4. MongoDB Client vs Mongoose
  5. Indeksering
2. Eksamensopgave og vejledning



# Agenda



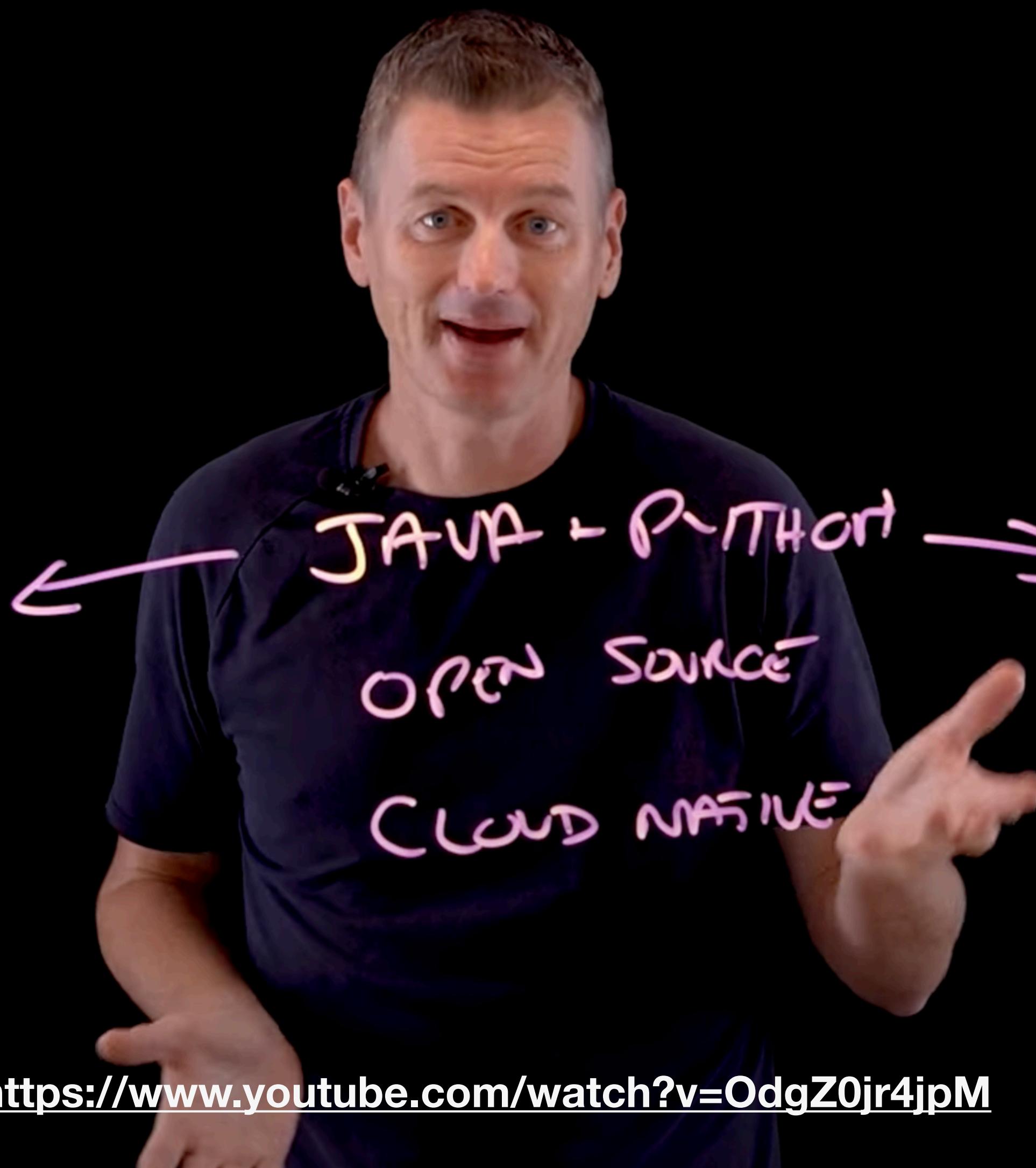
# Most Popular Databases



<https://survey.stackoverflow.co/2023/#section-most-popular-technologies-databases>

# MySQL

TABLE  
1995  
SCHEMA  
RIGID



# MongoDB

DOCUMENT  
2007  
JSON  
FLEXIBLE

# Differences between SQL and NoSQL

	SQL Databases	NoSQL Databases
<b>Data Storage Model</b>	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
<b>Development History</b>	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
<b>Examples</b>	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune
<b>Primary Purpose</b>	General purpose	Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
<b>Schemas</b>	Rigid	Flexible
<b>Scaling</b>	Vertical (scale-up with a larger server)	Horizontal (scale-out across commodity servers)
<b>Multi-Record ACID Transactions</b>	Supported	Most do not support multi-record ACID transactions. However, some – like MongoDB – do.
<b>Joins</b>	Typically required	Typically not required
<b>Data to Object Mapping</b>	Requires ORM (object-relational mapping)	Many do not require ORMs. MongoDB documents map directly to data structures in most popular programming languages.

<https://www.mongodb.com/nosql-explained/nosql-vs-sql>

# Features: SQL vs NoSQL

Feature	SQL Databases	NoSQL Databases
Data Structure	Structured	Flexible (Document, Key-Value, Graph, Wide-Column)
Data Types	Structured	Structured, Semi-structured, Unstructured
Scalability	Vertical (increasing resources on a single server)	Horizontal (adding more servers to a cluster)
ACID Compliance	Yes	No (may support eventual consistency)
Query Language	Structured Query Language (SQL)	Less structured query languages or no specific query language
Applications	E-commerce, Financial Systems, Enterprise Applications	Web Applications, Social Media Platforms, Mobile Apps
Strengths	Data organization, Complex queries, ACID compliance	Flexibility, Scalability, Handling unstructured data
Weaknesses	Rigid schema, Slow performance for large data	Less mature technology, Less standardized query syntax

[!\[\]\(d1526d19d632be0d0d4b68f470ac314b\_img.jpg\) Export to Sheets](#)

<https://g.co/bard/share/9a4c7d0aa722>

# Features: MySQL vs MongoDB

Feature	MySQL	MongoDB
Data Structure	Relational (tables, rows, columns)	Document (JSON-like documents)
Schema	Rigid, requires predefined schema	No schema, flexible data structure
Data Types	Structured	Structured, Semi-structured, Unstructured (JSON, XML, binary data)
Scalability	Vertical (increasing resources on a single server)	Horizontal (adding more servers to a cluster)
ACID Compliance	Yes	Eventual consistency
Query Language	Structured Query Language (SQL)	MongoDB Query Language (MongoDBQL)
Applications	E-commerce, Financial Systems, Enterprise Applications	Web Applications, Social Media Platforms, Mobile Apps, Real-time applications
Strengths	Data organization, Complex queries, ACID compliance, Mature technology	Flexibility, Scalability, Handling unstructured data, Real-time data handling
Weaknesses	Rigid schema, Slow performance for large data	Less mature technology, Less standardized query syntax, May not support complex queries

Export to Sheets

<https://g.co/bard/share/1233c9c2da05>

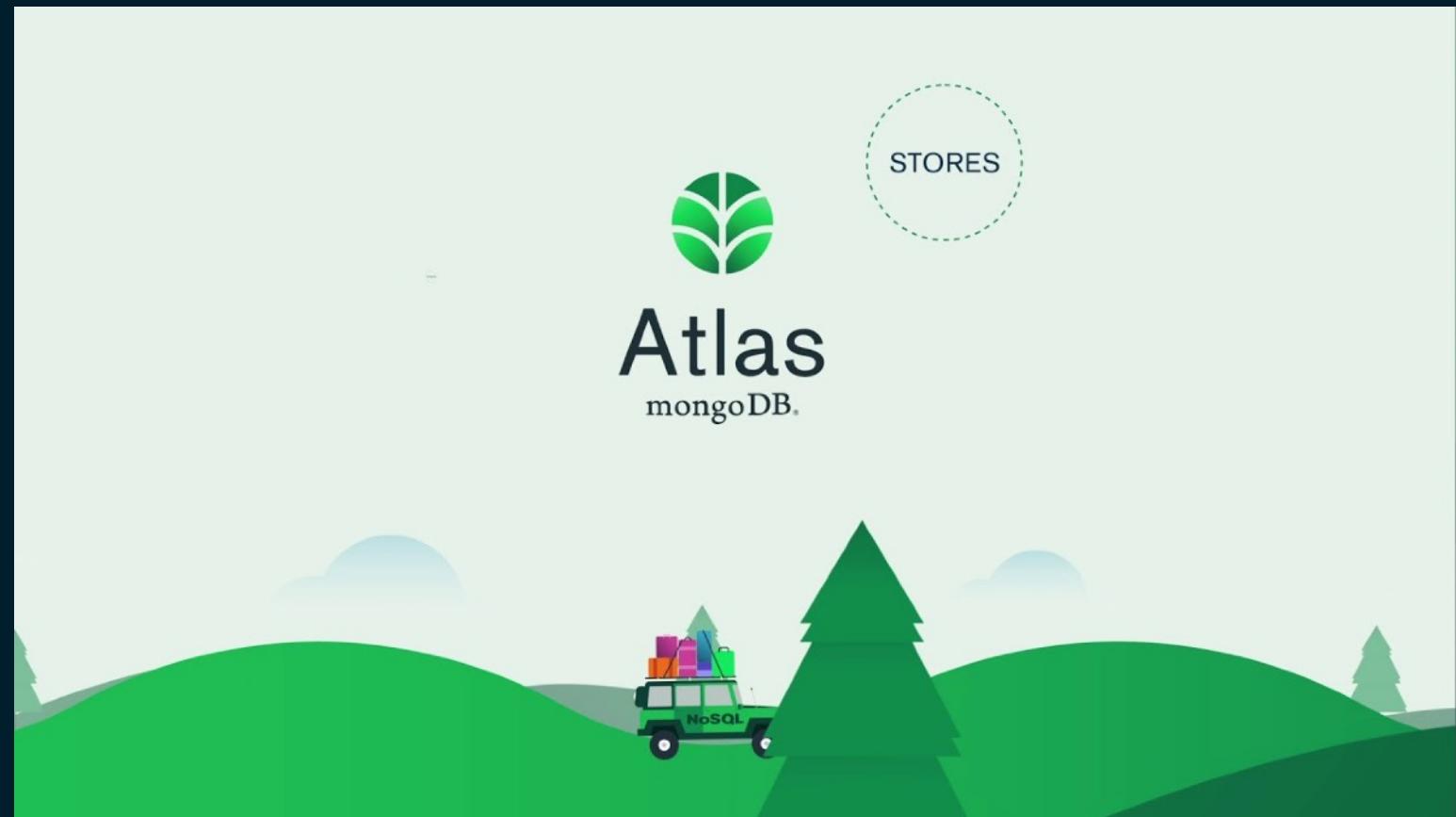
# Best For: MySQL vs MongoDB

Feature	MySQL	MongoDB
Best for	Applications with structured data, complex queries, and strict data integrity	Applications with unstructured or evolving data, high scalability, and real-time data handling

 Export to Sheets

<https://g.co/bard/share/1233c9c2da05>

# Why NoSQL?



[https://www.youtube.com/watch?v=0X43QfCfyk0&t=35s&ab\\_channel=MongoDB](https://www.youtube.com/watch?v=0X43QfCfyk0&t=35s&ab_channel=MongoDB)

- **Flexibility:** NoSQL databases do not require a predefined schema, which means that the data structure can be easily changed as the application evolves. This makes them well-suited for applications that handle unstructured or semi-structured data.
- **Scalability:** NoSQL databases are typically horizontally scalable, which means that they can be easily added to more servers to handle increasing data and workloads. This makes them ideal for applications that need to support large amounts of data or that experience sudden spikes in traffic.
- **Performance:** NoSQL databases can often outperform SQL databases for certain types of workloads, such as read-heavy applications or applications that require low latency. This is because they are designed to store and retrieve data efficiently, without the overhead of maintaining a rigid schema.
- **Ease of use:** NoSQL databases can be easier to use than SQL databases, especially for developers who are not familiar with traditional relational databases. They often have simpler APIs and data models, which can make it easier to develop and maintain applications.

# SQL vs MQL



**MongoDB**

Docs Menu

## Terminology and Concepts

The following table presents the various SQL terminology and concepts and the corresponding MongoDB terminology and concepts.

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	<code>\$lookup</code> , embedded documents
primary key	primary key
Specify any unique column or column combination as primary key.	In MongoDB, the primary key is automatically set to the <code>_id</code> field.

**Share Feedback**

<https://www.mongodb.com/docs/manual/reference/sql-comparison/>

## SQL Schema Statements

```
CREATE TABLE people (
    id MEDIUMINT NOT NULL
        AUTO_INCREMENT,
    user_id Varchar(30),
    age Number,
    status char(1),
    PRIMARY KEY (id)
)
```

## MongoDB Schema Statements

Implicitly created on first `insertOne()` or `insertMany()` operation. The primary key `_id` is automatically added if `_id` field is not specified.

```
db.people.insertOne( {
    user_id: "abc123",
    age: 55,
    status: "A"
} )
```

## SQL INSERT Statements

```
INSERT INTO people(user_id,  
                  age,  
                  status)  
VALUES ("bcd001",  
       45,  
      "A")
```

## MongoDB insertOne() Statements

```
db.people.insertOne(  
  { user_id: "bcd001", age: 45, status: "A" })
```

## SQL SELECT Statements

```
SELECT *  
FROM people
```

```
SELECT id,  
       user_id,  
       status  
FROM people
```

```
SELECT user_id, status  
FROM people
```

```
SELECT *  
FROM people  
WHERE status = "A"
```

## MongoDB find() Statements

```
db.people.find()
```

```
db.people.find(  
    { },  
    { user_id: 1, status: 1 }  
)
```

```
db.people.find(  
    { },  
    { user_id: 1, status: 1, _id: 0 }  
)
```

```
db.people.find(  
    { status: "A" }  
)
```

## SQL Update Statements

```
UPDATE people  
SET status = "C"  
WHERE age > 25
```

```
UPDATE people  
SET age = age + 3  
WHERE status = "A"
```

## MongoDB updateMany() Statements

```
db.people.updateMany(  
  { age: { $gt: 25 } },  
  { $set: { status: "C" } }  
)
```

```
db.people.updateMany(  
  { status: "A" } ,  
  { $inc: { age: 3 } }  
)
```

# Mapping

## Object Relational Mapping (ORM)

&

## Object Document Mapping (ODM)

```
index.php
<script>
</head>
<body>
<main>
<section class="wrapper--page">
<nav>
<ul>
<li id="left--item">MF</li>
<li class="right--items">A href="#section--about">Over mij</a>
<li class="right--items">A href="#section--skills">Skills</a>
<li class="right--items">A href="#section--work">Work</a>
</ul>
</nav>
<section id="intro--section">
<article id="intro--section--text">

<h1 class="section--header">
    Hey, ik ben Arnold!
</h1>
<p>Ik ben een front-end developer en student applicatiewetenschappen.</p>
<p><a href="#work--section" class="pink--button scroll" style="background-color: #ff99cc; color: white; padding: 10px 20px; border-radius: 5px; text-decoration: none; font-weight: bold; font-size: 1em; border: 2px solid #ff99cc; transition: background-color 0.3s ease, color 0.3s ease; ">Work</a></p>

<h2 class="section--header">
    Mijn Skills.
</h2>
<section id="skills--section--wrap">
</section>

```

# Web App

The screenshot shows a web application titled "CRUD App". It displays a grid of six user profiles. Each profile includes a thumbnail, name, title, email, and two buttons: "UPDATE" and "DELETE". Below the grid is a modal window titled "Create a new User" with fields for name, title, mail, and image.

	Name	Title	Mail	Image
1	Peter Lind	Senior Lecturer	petl@kea.dk	<a href="https://">https://</a>
2	Rasmus Cederdorff	Senior Lecturer	race@dev.dk	<a href="https://">https://</a>
3	Lars Bogetoft	Head of Education	larb@eaaa.dk	<a href="https://">https://</a>
4	Edith Terte	Lecturer	edan@kea.dk	<a href="https://">https://</a>
5	Frederikke Bender	Head of Education	fbe@kea.dk	<a href="https://">https://</a>
6	Murat Kilic	Senior Lecturer	mki@eaaa.dk	<a href="https://">https://</a>
7	Anne Andersen	Head of Education	anki@mail.dk	<a href="https://">https://</a>

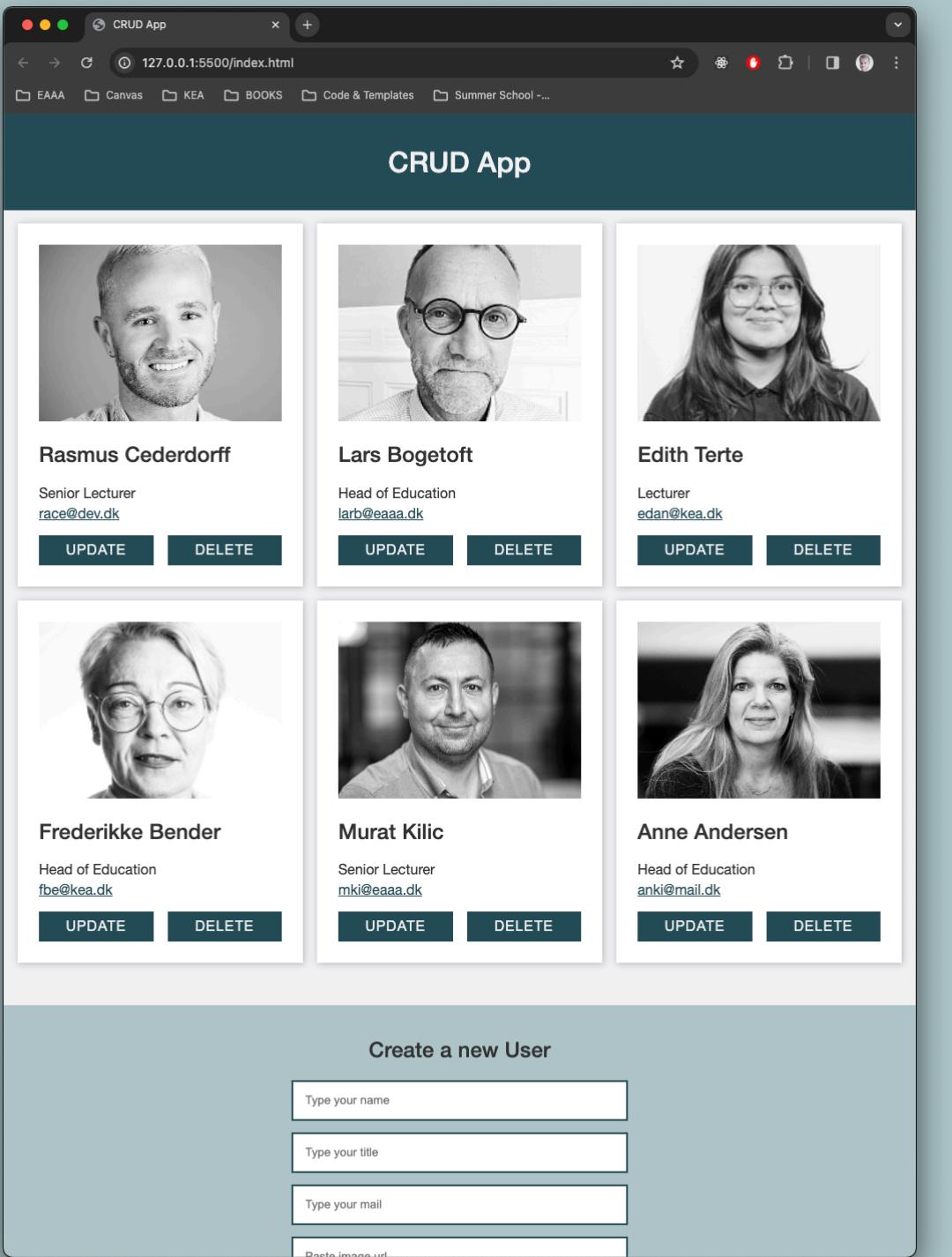
# Database

A screenshot of a database table with columns: id, name, mail, title, and image. The data corresponds to the users listed in the web application.

	id	name	mail	title	image
1	Peter Lind	petl@kea.dk	Senior Lecturer	<a href="https://">https://</a>	
2	Rasmus Cederdorff	race@dev.dk	Senior Lecturer	<a href="https://">https://</a>	
3	Lars Bogetoft	larb@eaaa.dk	Head of Education	<a href="https://">https://</a>	
4	Edith Terte	edan@kea.dk	Lecturer	<a href="https://">https://</a>	
5	Frederikke Bender	fbe@kea.dk	Head of Education	<a href="https://">https://</a>	
6	Murat Kilic	mki@eaaa.dk	Senior Lecturer	<a href="https://">https://</a>	
7	Anne Kirketerp	anki@eaaa.dk	Head of Education	<a href="https://">https://</a>	

Hvordan “mapper” vi det her?

# Frontend



Web App  
(Client)

# REST API med Node.js



REST API  
(Udveksler JSON)

The screenshot shows a code editor with the file "app.js" open. The code defines a REST API for managing users. It includes routes for reading all users, reading one user by ID, creating a new user, updating an existing user, and deleting a user. The API interacts with a MySQL database using the "db" object. The word "RESPONSE" is overlaid on the right side of the JSON, and an arrow points from the code to the MySQL database on the right.

```
15 // READ all users
16 app.get("/users", async (request, response) => {
17   const query = "SELECT * FROM users"; // SQL query
18   const [users] = await db.execute(query); // Execute the query
19   response.json(users); // Send the results as JSON
20 });
21
22 // READ one user
23 app.get("/users/:id", async (request, response) => {
24   const id = request.params.id; // grabs the id from the url
25   const query = "SELECT * FROM users WHERE id=?"; // sql query
26   const values = [id]; // values to insert into query
27   const [results] = await db.execute(query, values); // execute query
28   response.json(results[0]); // send response
29 });
30
31 // CREATE user
32 app.post("/users", async (request, response) => {
33   const user = request.body; // grab the user from the request
34   const query = "INSERT INTO users(name, mail, title, image"
35   const values = [user.name, user.mail, user.title, user.image];
36   const [result] = await db.execute(query, values); // execute query
37   response.json(result); // send response
38 });
39
40 // UPDATE user
41 app.put("/users/:id", async (request, response) => {
42   const id = request.params.id; // grabs the id from the url
43   const user = request.body; // grab the user from the request
44   const query = "UPDATE users SET name=?, mail=?, title=?,"
45   const values = [user.name, user.mail, user.title, user.image];
46   const [result] = await db.execute(query, values); // execute query
47   response.json(result); // send response
48 });
49
50 // DELETE user
```

Backend App  
(Server)

# Database

	id	name	mail	title
1	Peter Lind	petl@kea.dk	Senior Lecturer	
2	Rasmus Cederdorff	race@dev.dk	Senior Lecturer	
3	Lars Bogetoft	larb@eaaa.dk	Head of Education	
4	Edith Terte	edan@kea.dk	Lecturer	
5	Frederikke Bender	fbe@kea.dk	Head of Education	
6	Murat Kilic	mki@eaaa.dk	Senior Lecturer	
7	Anne Kirketerp	anki@eaaa.dk	Head of Education	

SQL

# BACKEND

A screenshot of a web browser window titled "localhost:3000/users". The content shows a JSON array of user objects. Each user has an "id", "name", "mail", "title", and "image" field. The "image" fields contain URLs for profile pictures. The browser has tabs for "Raw" and "Parsed" views.

```
[{"id": 1, "name": "Rasmus Cederdorff", "mail": "race@dev.dk", "title": "Senior Lecturer", "image": "https://share.cederdorff.com/images/race.jpg"}, {"id": 2, "name": "Lars Bogetoft", "mail": "larb@aaa.dk", "title": "Head of Education", "image": "https://kea.dk/sfir/w200-clx1/images/user-profile/chefer/larb.jpg"}, {"id": 3, "name": "Edith Terte", "mail": "edan@kea.dk", "title": "Lecturer", "image": "https://media.licdn.com/dms/image/C4E03AQ6nx7oUPqo_g/profile-displayphoto-shrink_800_800/0/164370886591?e=1697673600&v=beta&t=Qp4GcxVLJfsZ14t-if6YJ601u7bH2oLwWgVxb-X5Nt4"}, {"id": 4, "name": "Frederikke Bender", "mail": "fbe@kea.dk", "title": "Head of Education", "image": "https://kea.dk/sfir/w200-clx1/images/user-profile/chefer/fbe.jpg"}, {"id": 5, "name": "Murat Kılıç", "mail": "mki@aaa.dk", "title": "Senior Lecturer", "image": "https://www.eaaa.dk/media/llyavasj/murat-kilic.jpg?width=800&height=450&rnd=133401946552600000"}, {"id": 6, "name": "Anne Andersen", "mail": "anki@mail.dk", "title": "Head of Education", "image": "https://www.eaaa.dk/media/5buh1xeo/anne-kirketerp.jpg?width=800&height=450&rnd=133403878321500000"}]
```

REQUEST



RESPONSE

A screenshot of a terminal window titled "node-express-rest-users". It shows a Node.js application running on port 3000. The code in "app.js" contains an "app.get('/users')" route that executes a SQL query to fetch all users from a database and returns the results as JSON. The terminal also shows the application restarting.

```
JS app.js > ⚡ app.put("/users/:id") callback > [?] query
15 // READ all users
16 app.get("/users", async (request, response) => {
17   const query = "SELECT * FROM users"; // SQL qu
18   const [users] = await db.execute(query); // Ex
19   response.json(users); // Send the results as J
20 });
21
PROBLEMS OUTPUT TERMINAL ...
Restarting 'app.js'
App listening on http://localhost:3000
```

A screenshot of a MySQL database table named "mysql-users". The table has columns for "id", "name", and "mail". It contains five rows of data corresponding to the users listed in the JSON response.

	id	name	mail
1	1	Maria Louise Bendixen	mlbe@aaa.dk
2	2	Rasmus Cederdorff	race@aaa.dk
3	3	Anne Kirketerp	anki@aaa.dk
4	4	Line Skjødt	lskj@aaa.dk
5	5	Dan Okkels Brendstrup	dob@aaa.dk

CLIENT

SERVER

# Object Relational Mapping

## ORM

id	name	mail	title	image
1	Peter Lind	petl@kea.dk	Senior Lecturer	<a href="https://share.cederdorff.com/images/pe...">https://share.cederdorff.com/images/pe...</a>
2	Rasmus Cederdorff	race@dev.dk	Senior Lecturer	<a href="https://share.cederdorff.com/images/ra...">https://share.cederdorff.com/images/ra...</a>
3	Lars Bogetoft	larb@eaaa.dk	Head of Education	<a href="https://kea.dk/slir/w200-c1x1/images/u...">https://kea.dk/slir/w200-c1x1/images/u...</a>
4	Edith Terte	edan@kea.dk	Lecturer	<a href="https://media.lidcn.com/dms/image/C4E0...">https://media.lidcn.com/dms/image/C4E0...</a>
5	Frederikke Bender	fbe@kea.dk	Head of Education	<a href="https://kea.dk/slir/w200-c1x1/images/u...">https://kea.dk/slir/w200-c1x1/images/u...</a>
6	Murat Kilic	mki@eaaa.dk	Senior Lecturer	<a href="https://www.eaaa.dk/media/llyavasj/mur...">https://www.eaaa.dk/media/llyavasj/mur...</a>
7	Anne Kirketerp	anki@eaaa.dk	Head of Education	<a href="https://www.eaaa.dk/media/5buh1xeo/ann...">https://www.eaaa.dk/media/5buh1xeo/ann...</a>

```
const user = {  
  id: 6,  
  name: "Murat Kilic",  
  mail: "mki@eaaa.dk",  
  title: "Senior Lecturer",  
  image: "https://www.eaaa.dk/media/llyavasj/mur...";  
};
```

From database row to object

# Object Relational Mapping

## ORM

id	name	mail	title	image
1	Peter Lind	petl@kea.dk	Senior Lecturer	<a href="https://share.cederdorff.com/images/pe...">https://share.cederdorff.com/images/pe...</a>
2	Rasmus Cederdorff	race@dev.dk	Senior Lecturer	<a href="https://share.cederdorff.com/images/ra...">https://share.cederdorff.com/images/ra...</a>
3	Lars Bogetoft	larb@eaaa.dk	Head of Education	<a href="https://kea.dk/slir/w200-c1x1/images/u...">https://kea.dk/slir/w200-c1x1/images/u...</a>
4	Edith Terte	edan@kea.dk	Lecturer	<a href="https://media.licdn.com/dms/image/C4E0...">https://media.licdn.com/dms/image/C4E0...</a>
5	Frederikke Bender	fbe@kea.dk	Head of Education	<a href="https://kea.dk/slir/w200-c1x1/images/u...">https://kea.dk/slir/w200-c1x1/images/u...</a>
6	Murat Kilic	mki@eaaa.dk	Senior Lecturer	<a href="https://www.eaaa.dk/media/llyavasj/mur...">https://www.eaaa.dk/media/llyavasj/mur...</a>
7	Anne Kirketerp	anki@eaaa.dk	Head of Education	<a href="https://www.eaaa.dk/media/5buh1xeo/ann...">https://www.eaaa.dk/media/5buh1xeo/ann...</a>

From database table to array of objects

```
const users = [
  {
    id: 2,
    name: "Rasmus Cederdorff",
    mail: "race@dev.dk",
    title: "Senior Lecturer",
    image: "https://share.cederdorff.com/images/race.jpg"
  },
  {
    id: 3,
    name: "Lars Bogetoft",
    mail: "larb@eaaa.dk",
    title: "Head of Education",
    image: "https://kea.dk/slir/w200-c1x1/images/user-profile/chefer/larb.jpg"
  },
  {
    id: 4,
    name: "Edith Terte",
    mail: "edan@kea.dk",
    title: "Lecturer",
    image: "https://media.licdn.com/dms/image/C4E03AQE6nx7oUPqo\_g/profile-displayphoto-shrink\_800\_800"
  },
  {
    id: 5,
    name: "Frederikke Bender",
    mail: "fbe@kea.dk",
    title: "Head of Education",
    image: "https://kea.dk/slir/w200-c1x1/images/user-profile/fbe.jpg"
  },
  {
    id: 6,
    name: "Murat Kilic",
    mail: "mki@eaaa.dk",
    title: "Senior Lecturer",
    image: "https://www.eaaa.dk/media/llyavasj/murat-kilic.jpg?width=800&height=450&rnd=133401946552600"
  },
  {
    id: 7,
    name: "Anne Andersen",
    mail: "anki@mail.dk",
    title: "Head of Education",
    image: "https://www.eaaa.dk/media/5buh1xeo/anne-kirketerp.jpg?width=800&height=450&rnd=133403878321"
  }
];
```

# Object Relational Mapping

ORM

- **Simplifies Database Interaction:** Allows using objects in code to interact with a relational database.
- **Maps Objects to Tables:** Links object-oriented language structures to relational database tables.
- **Abstraction Layer:** Abstracts database operations, reducing manual SQL coding.
- **Automates Queries:** Generates and executes SQL queries based on object interactions.
- **Saves Development Time:** Reduces the need for low-level database code, streamlining development.

# Object Relational Mapping

## Pros

- **Simplifies the development of database applications:** ORMs can simplify the development of database applications by abstracting away database-specific details. This makes it easier for developers to focus on the business logic of their applications.
- **Increases productivity:** ORMs can increase productivity by automating repetitive tasks, such as creating and updating tables and columns. This can free up developers to focus on more complex tasks.
- **Improves maintainability:** ORMs can improve the maintainability of database applications by making it easier to change the database structure. This is because ORMs can generate code compatible with the new database structure.

# Object Relational Mapping

## Cons

- **May reduce performance:** ORMs can reduce the performance of database applications by introducing an extra layer between the application and the database. This is because ORMs need to convert data between the application's data model and the database's data model.
- **May make it harder to understand the database:** ORMs can make it harder to understand the database by abstracting away database-specific details. This is because developers need to understand how the ORM works to understand how the data is stored in the database.
- **May be difficult to learn:** ORMs can be difficult to learn, especially for developers who don't have experience with databases. This is because ORMs introduce a new paradigm for interacting with databases.



Sequelize

v6 - stable ▾

API References ▾

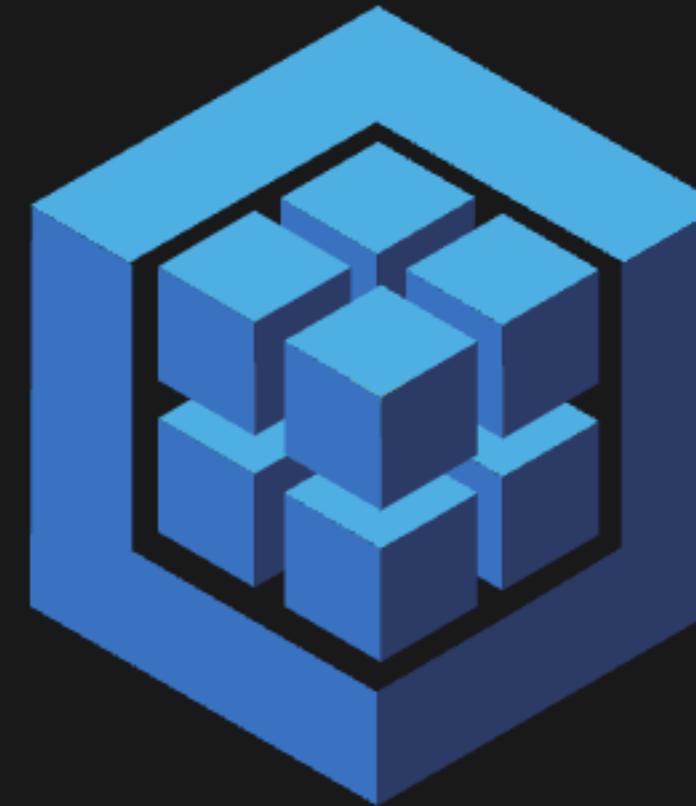
Slack

GitHub

Security



Search



# Sequelize

Sequelize is a modern TypeScript and Node.js ORM for Oracle, Postgres, MySQL, MariaDB, SQLite and SQL Server, and more. Featuring solid transaction support, relations, eager and lazy loading, read replication and more.

Getting Started

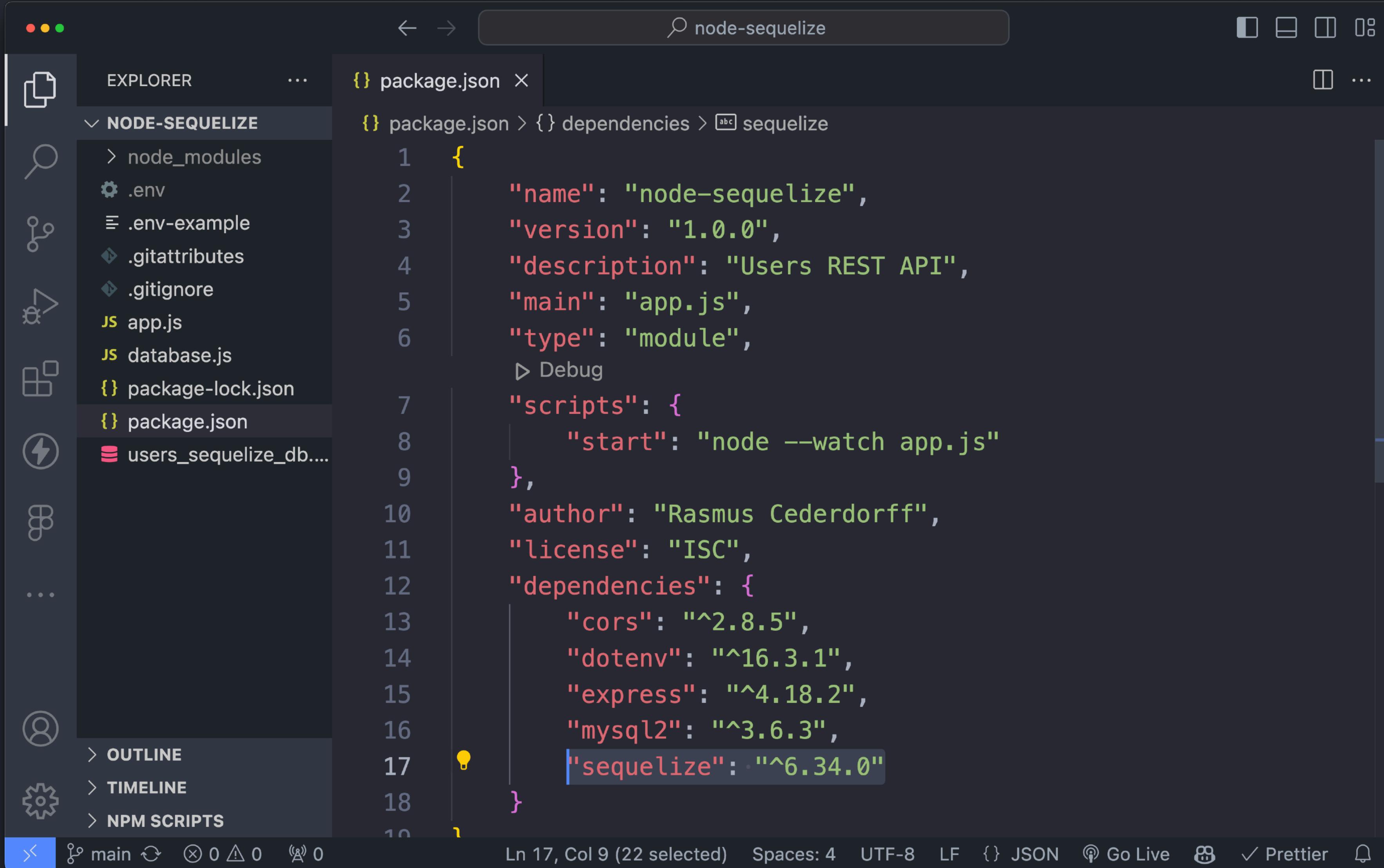
API Reference

Upgrade to v6

Support us

... npm package

# Sequelize is a dependency (node module)



The screenshot shows a dark-themed instance of Visual Studio Code (VS Code) with the following details:

- File Explorer:** On the left, the "NODE-SEQUELIZE" folder is expanded, showing files like `node_modules`, `.env`, `.env-example`, `.gitattributes`, `.gitignore`, `app.js`, `database.js`, `package-lock.json`, `package.json`, and `users_sequelize_db....`.
- Search Bar:** At the top center, the search bar contains the text `node-sequelize`.
- Code Editor:** The main editor area displays the `package.json` file content. The `dependencies` section includes the entry `"sequelize": "^6.34.0"`.
- Status Bar:** At the bottom, the status bar shows the file path `Ln 17, Col 9 (22 selected)`, encoding `Spaces: 4`, and other settings.

```
{} package.json X
{} package.json > {} dependencies > sequelize
1  {
2    "name": "node-sequelize",
3    "version": "1.0.0",
4    "description": "Users REST API",
5    "main": "app.js",
6    "type": "module",
7    "scripts": {
8      "start": "node --watch app.js"
9    },
10   "author": "Rasmus Cederdorff",
11   "license": "ISC",
12   "dependencies": {
13     "cors": "^2.8.5",
14     "dotenv": "^16.3.1",
15     "express": "^4.18.2",
16     "mysql2": "^3.6.3",
17     "sequelize": "^6.34.0"
18   }
19 }
```

# Define models to handle the DB interaction

```
// ===== Define Models ===== //
// Define user model
const User = sequelize.define("user", {
  // User model attributes
  name: {
    type: DataTypes.STRING,
    allowNull: false // Name is required
  },
  title: {
    type: DataTypes.STRING
  },
  mail: {
    type: DataTypes.STRING,
    allowNull: false // Email is required
  },
  image: {
    type: DataTypes.TEXT // URL to image
  }
});

// Sample users
// Sample user 1
const user1 = await User.create({
  name: "Rasmus Cederdorff",
  title: "Senior Lecturer",
  mail: "race@eaaa.dk",
  image: "https://share.cederdorff.com/images/race.jpg"
});

// Sample user 2
const user2 = await User.create({
  name: "Anne Kirketerp",
  title: "Head of Department",
  mail: "anki@eaaa.dk",
  image: "https://www.eaaa.dk/media/5buh1xeo/anne-kirketerp.jpg?width=1000"
});

// Sample user 3
const user3 = await User.create({
  name: "Murat Kilic",
  title: "Senior Lecturer",
  mail: "mki@eaaa.dk",
  image: "https://www.eaaa.dk/media/llyavasj/murat-kilic.jpg?width=1000"
});
```

# Creates users table with entities

```
// ===== Define Models ===== //
// Define user model
const User = sequelize.define("user", {
  // User model attributes
  name: {
    type: DataTypes.STRING,
    allowNull: false // Name is required
  },
  title: {
    type: DataTypes.STRING
  },
  mail: {
    type: DataTypes.STRING,
    allowNull: false // Email is required
  },
  image: {
    type: DataTypes.TEXT // URL to image
  }
});
```

id	name	title	mail	image	createdAt	updatedAt
1	Rasmus Ced...	Senior Lect...	race@eaaa.dk	https://share.cederdo...	2023-11-12 19:0...	2023-11-12 19:0...
2	Anne Kirke...	Head of Dep...	anki@eaaa.dk	https://www.eaaa.dk/m...	2023-11-12 19:0...	2023-11-12 19:0...
3	Murat Kilic	Senior Lect...	mki@eaaa.dk	https://www.eaaa.dk/m...	2023-11-12 19:0...	2023-11-12 19:0...

# Methods (SQL Queries)

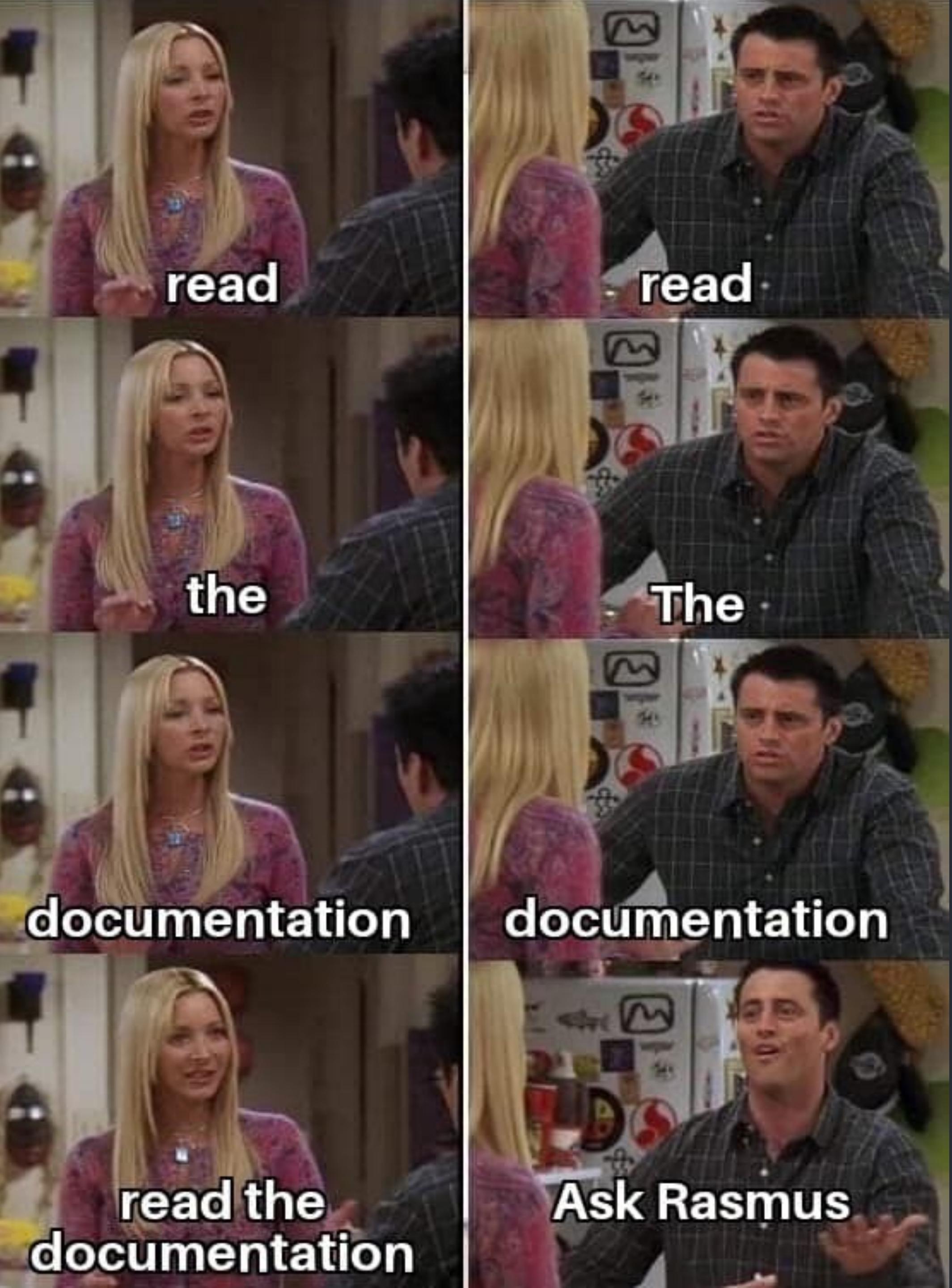
```
const users = await User.findAll(); // SELECT * FROM users;
```

```
const users = await User.findByPk(id); // SELECT * FROM users WHERE id = id;
```

```
const newUser = await User.create(user); // INSERT INTO users (name, title, mail, image) VALUES
```

```
const [result] = await User.update(user, { where: { id: id } }); // UPDATE users SET name =
```

```
const result = await User.destroy({ where: { id: id } }); // DELETE FROM users WHERE id = id
```



# Useful Methods (see the docs)

- Core Concepts of Sequelize: <https://sequelize.org/docs/v6/category/core-concepts/>
- Model Basics: <https://sequelize.org/docs/v6/core-concepts/model-basics/>
- Model Querying - Basics: <https://sequelize.org/docs/v6/core-concepts/model-querying-basics/>
- Model Querying - Finders: <https://sequelize.org/docs/v6/core-concepts/model-querying-finders/>
- Associations: <https://sequelize.org/docs/v6/core-concepts/assocs/>

# Routes: SQL vs Sequelize

```
// READ all users
app.get("/users", async (request, response) => {
  const query = "SELECT * FROM users"; // SQL query
  const [users] = await db.execute(query); // Execute the query
  response.json(users); // Send the results as JSON
});

// READ one user
app.get("/users/:id", async (request, response) => {
  const id = request.params.id; // grabs the id from the url
  const query = "SELECT * FROM users WHERE id=?"; // sql query
  const values = [id]; // values to insert into query
  const [results] = await db.execute(query, values); // execute the query
  response.json(results[0]); // send response
});

// CREATE user
app.post("/users", async (request, response) => {
  const user = request.body; // grab the user from the request body
  const query = "INSERT INTO users(name, mail, title, image) values(?, ?, ?, ?);";
  const values = [user.name, user.mail, user.title, user.image]; // values to ins
  const [result] = await db.execute(query, values); // execute the query
  response.json(result); // send response
});

// UPDATE user
app.put("/users/:id", async (request, response) => {
  const id = request.params.id; // grabs the id from the url
  const user = request.body; // grab the user from the request body
```

```
// READ all users
app.get("/users", async (request, response) => {
  const users = await User.findAll(); // SELECT * FROM users;
  response.json(users); // send the users as JSON
});

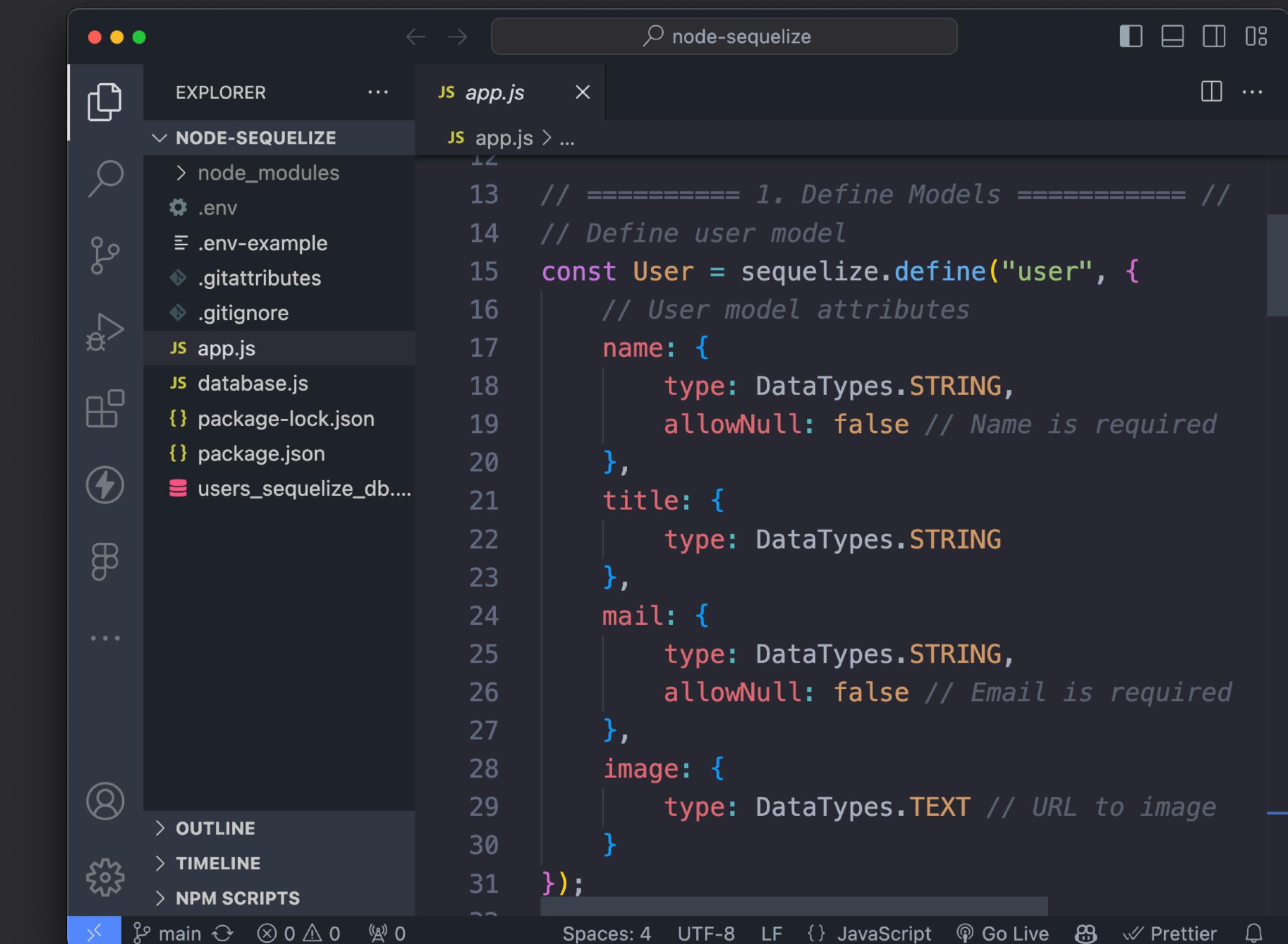
// READ one user by id
app.get("/users/:id", async (request, response) => {
  const id = request.params.id; // use id from url
  const users = await User.findByPk(id); // SELECT * FROM users WHERE id =
  response.json(users); // send the users as JSON
});

// CREATE one user
app.post("/users", async (request, response) => {
  const user = request.body; // use request body as user object
  const newUser = await User.create(user); // INSERT INTO users (name, ti
  response.json(newUser); // send the new user as JSON
});

// UPDATE one user by id
app.put("/users/:id", async (request, response) => {
  const id = request.params.id; // use id from url
  const user = request.body; // use request body as user object
  const [result] = await User.update(user, { where: { id: id } });
  if (result) {
    response.json({ message: "User updated" });
  } else {
```

# Node.js & Sequelize

Øvelse



The screenshot shows a dark-themed Node.js development environment. The Explorer pane on the left lists files and folders related to a Sequelize application, including `node_modules`, `.env`, `.env-example`, `.gitattributes`, `.gitignore`, `app.js` (which is selected), `database.js`, `package-lock.json`, `package.json`, and `users_sequelize_db....`. The Editor pane on the right displays the `app.js` file, which contains code for defining a User model using Sequelize. The code includes attributes for name, title, mail, and image, each with specific data types and allowNull settings.

```
// ===== 1. Define Models ===== //
// Define user model
const User = sequelize.define("user", {
  // User model attributes
  name: {
    type: DataTypes.STRING,
    allowNull: false // Name is required
  },
  title: {
    type: DataTypes.STRING
  },
  mail: {
    type: DataTypes.STRING,
    allowNull: false // Email is required
  },
  image: {
    type: DataTypes.TEXT // URL to image
  }
});
```

# Object Document Mapping

ODM

```
_id: ObjectId('70717a4759314d6e48596d33')
image: "https://www.eaaa.dk/media/14qpfeq4/line-skjodt.jpg?width=800&height=45..."
mail: "lslkj@eaaa.dk"
name: "Line Skjødt"
title: "Senior Lecturer & Internship Coordinator"
```

```
_id: ObjectId('66547338344b526f59773570')
image: "https://share.cederdorff.com/images/race.jpg"
mail: "race@eaaa.dk"
name: "Rasmus Cederdorff"
title: "Senior Lecturer"
```

```
_id: ObjectId('666a70525452546a5a487772')
image: "https://www.baaa.dk/media/5buh1xeo/anne-kirketerp.jpg?anchor=center&mo..."
mail: "anki@eaaa.dk"
name: "Anne Kirketerp"
title: "Head of Department"
```

```
_id: ObjectId('486c76524872353843303567')
image: "https://www.eaaa.dk/media/bdojel41/dan-okkels-brendstrup.jpg?anchor=ce..."
mail: "dob@eaaa.dk"
name: "Dan Okkels Brendstrup"
title: "Lecturer"
```

```
const user = {
  _id: "666a70525452546a5a487772",
  image: "https://www.baaa.dk/media/5buh1xeo/anne-kirketerp.jpg?anchor=cen...
  mail: "anki@eaaa.dk",
  name: "Anne Kirketerp",
  title: "Head of Department"
};
```

From document to object

# Object Document Mapping

ODM

```
_id: ObjectId('70717a4759314d6e48596d33')
image: "https://www.eaaa.dk/media/14qpfeq4/line-skjodt.jpg?width=800&height=45..."
mail: "lskj@eaaa.dk"
name: "Line Skjødt"
title: "Senior Lecturer & Internship Coordinator"

_id: ObjectId('66547338344b526f59773570')
image: "https://share.cederdorff.com/images/race.jpg"
mail: "race@eaaa.dk"
name: "Rasmus Cederdorff"
title: "Senior Lecturer"

_id: ObjectId('666a70525452546a5a487772')
image: "https://www.baaa.dk/media/5buh1xeo/anne-kirketerp.jpg?anchor=center&mode=crop&w=800&h=450"
mail: "anki@eaaa.dk"
name: "Anne Kirketerp"
title: "Head of Department"

_id: ObjectId('486c76524872353843303567')
image: "https://www.eaaa.dk/media/bdojel41/dan-okkels-brendstrup.jpg?anchor=center&mode=crop&w=800&h=450"
mail: "dob@eaaa.dk"
name: "Dan Okkels Brendstrup"
title: "Lecturer"
```

```
const users = [
  {
    _id: "5a66505456454d514b663976",
    image: "https://www.baaa.dk/media/b5ahrlra/maria-louise-bendixen.jpg?anchor=center&mode=crop&w=800&h=450",
    mail: "mlbe@eaaa.dk",
    name: "Maria Louise Bendixen",
    title: "Senior Lecturer"
  },
  {
    _id: "66547338344b526f59773570",
    image: "https://share.cederdorff.com/images/race.jpg",
    mail: "race@eaaa.dk",
    name: "Rasmus Cederdorff",
    title: "Senior Lecturer"
  },
  {
    _id: "666a70525452546a5a487772",
    image: "https://www.baaa.dk/media/5buh1xeo/anne-kirketerp.jpg?anchor=center&mode=crop&w=800&h=450",
    mail: "anki@eaaa.dk",
    name: "Anne Kirketerp",
    title: "Head of Department"
  },
  {
    _id: "70717a4759314d6e48596d33",
    image: "https://www.eaaa.dk/media/14qpfeq4/line-skjodt.jpg?width=800&height=450&mode=crop&w=800&h=450",
    mail: "lskj@eaaa.dk",
    name: "Line Skjødt",
    title: "Senior Lecturer & Internship Coordinator"
  },
  {
    _id: "486c76524872353843303567",
    image: "https://www.eaaa.dk/media/bdojel41/dan-okkels-brendstrup.jpg?anchor=center&mode=crop&w=800&h=450",
    mail: "dob@eaaa.dk",
    name: "Dan Okkels Brendstrup",
    title: "Lecturer"
  }
]
```

From collection to array of objects

# Mongoose: Schema, Model and Object

```
// ===== SCHEMAS & MODELS ===== //

const artistSchema = new mongoose.Schema({
  name: { type: String, required: true },
  genre: String,
  image: String,
  birthdate: { type: Date, default: Date.now },
  gender: String
});

const Artist = mongoose.model("Artist", artistSchema);

// ===== CREATE TEST DATA ===== //

const adele = new Artist({
  name: "Adele",
  genre: "Pop",
  image: "https://upload.wikimedia.org/wikipedia/commons/thumb/2/20/Adele_at_the_2011_Brave_Heart_Music_Awards_in_Los_Angeles_%28cropped%29.jpg/220px-Adele_at_the_2011_Brave_Heart_Music_Awards_in_Los_Angeles_%28cropped%29.jpg",
  birthdate: new Date("1988-05-05"),
  gender: "Female"
});
await adele.save();

const beyonce = new Artist({
  name: "Beyoncé",
  genre: "Pop",
  image: "https://upload.wikimedia.org/wikipedia/commons/thumb/0/0d/Beyonc%C3%A9_in_2014_%28cropped%29.jpg/220px-Beyonc%C3%A9_in_2014_%28cropped%29.jpg",
  birthdate: new Date("1981-09-04"),
  gender: "Female"
});
await beyonce.save();
```

# Mongoose: Schema, Model and Object

```
// ===== SCHEMAS & MODELS ===== //
const artistSchema = new mongoose.Schema({
  name: { type: String, required: true },
  genre: String,
  image: String,
  birthdate: { type: Date, default: Date.now },
  gender: String
});
const Artist = mongoose.model("Artist", artistSchema);

// GET Endpoint "/artists" - get all artists
app.get("/artists", async (request, response) => {
  const artists = await Artist.find();
  response.json(artists);
});

// GET Endpoint "/artists/:id" - get one artist
app.get("/artists/:id", async (request, response) => {
  const artist = await Artist.findById(request.params.id);
  response.json(artist);
});

// POST Endpoint "/artists" - create one artist
app.post("/artists", async (request, response) => {
  const artist = new Artist(request.body);
  const result = await artist.save();
  response.json(result);
});

// PUT Endpoint "/artists/:id" - update one artist
app.put("/artists/:id", async (request, response) => {
  const artist = await Artist.findById(request.params.id);
  artist.set(request.body);
  const result = await artist.save();
  response.json(result);
});

// DELETE Endpoint "/artists/:id" - delete one artist
app.delete("/artists/:id", async (request, response) => {
  const result = await Artist.deleteOne({ _id: request.params.id });
  response.json(result);
});
```

<https://github.com/cederdorff/musicbase-mongodb/tree/mongoose>

# Object Document Mapping

ODM

Object document mapping (ODM) is a software design pattern that maps objects in an object-oriented programming language to documents in a document-oriented database. This pattern allows developers to store and retrieve data in a way that is both flexible and efficient.

## Key features:

- **Flexibility:** ODM allows developers to define their data structures, which can be tailored to the specific needs of their application. This is in contrast to relational databases, which have a fixed schema that can be limiting.
- **Efficiency:** ODM can improve the performance of data access by eliminating the need to join tables. In object-oriented languages, objects are naturally related to each other, and ODM can leverage these relationships to efficiently retrieve and store data.
- **Reusability:** ODM can be used to create reusable components that can be shared across multiple applications. This can save developers time and effort, and it can also improve the overall maintainability of their code.

# Object Document Mapping Pros

- **Flexibility:** ODM allows developers to define their own data structures, which can be tailored to the specific needs of their application. This is in contrast to relational databases, which have a fixed schema that can be limiting.
- **Efficiency:** ODM can improve the performance of data access by eliminating the need to join tables. In object-oriented languages, objects are naturally related to each other, and ODM can leverage these relationships to efficiently retrieve and store data.
- **Reusability:** ODM can be used to create reusable components that can be shared across multiple applications. This can save developers time and effort, and it can also improve the overall maintainability of their code.
- **Ease of use:** ODM can make it easier for developers to learn and use document-oriented databases. This is because ODM provides a high-level abstraction that hides the complexity of the underlying database.
- **No need for SQL:** ODM (and NoSQL) can eliminate the need for developers to write SQL queries. This can be a significant benefit for developers who are not familiar with SQL.

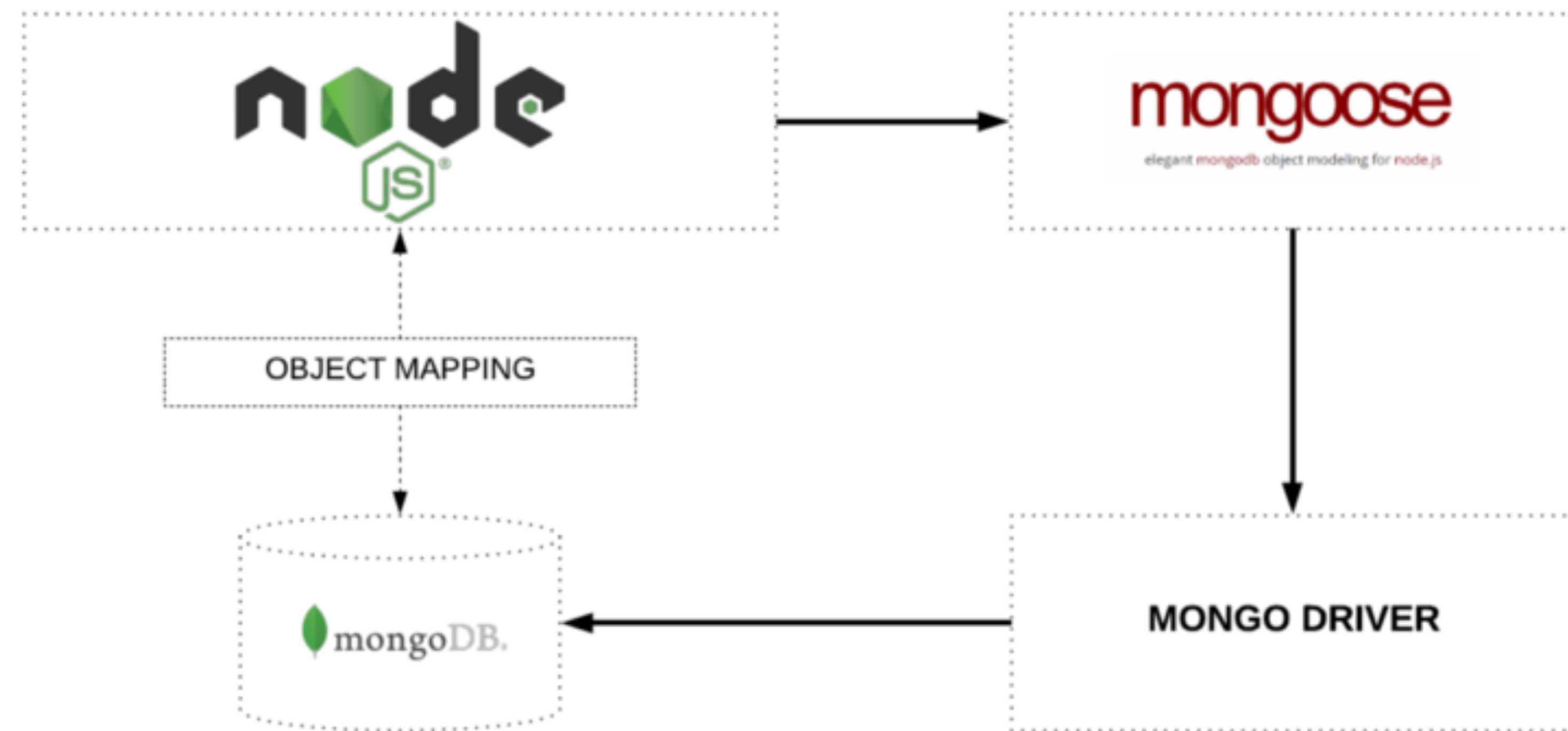
# Object Document Mapping

## Cons

- **Performance overhead:** ODM can introduce some performance overhead compared to directly writing SQL queries. This is because ODM must translate object-oriented data structures into the format that the database understands.
- **Potential for data integrity issues:** ODM can make it more difficult to enforce data integrity rules. This is because ODM is not as strict as SQL about data types and relationships.
- **Complexity:** ODM can add some complexity to the development process. This is because developers must learn how to use the ODM API and how to map object-oriented data structures to the database.
- **Lack of standardization:** There is no single standard for ODM, which can make it difficult to choose an ODM and to move between ODMs.

# mongoose

elegant `mongodb` object modeling for `node.js`



<https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/>

# Mongoose

## ODM for MongoDB

Mongoose is an ODM (object document mapper) for MongoDB. It is a popular and widely used ODM that provides a high-level abstraction layer on top of MongoDB, making it easier for developers to work with document-oriented data. Mongoose provides several features that make it a powerful tool for developing applications with MongoDB, including:

- **Object modelling:** Mongoose provides a schema-based approach to modelling data, allowing developers to define their data structures in an object-oriented way. This makes it easier to work with data in a way that is familiar to developers who are accustomed to working with object-oriented programming languages.
- **Data validation:** Mongoose provides built-in data validation features that can help to ensure that data is always in a consistent and valid state. This can help to prevent errors and improve the overall reliability of applications.
- **Query building:** Mongoose provides a fluent query builder that makes it easy to construct complex queries to retrieve and manipulate data. This can help to reduce the amount of code that developers need to write and improve the overall clarity of their code.
- **Relationship management:** Mongoose provides built-in support for managing relationships between documents, which can help to model complex data structures and improve the efficiency of data access.

# Is MongoDB (client) an ORM?

No, MongoDB is a NoSQL database management system (DBMS) that stores data in documents, which are JSON-like objects. It is not an ODM itself, but it supports ODMs like Mongoose. ODMs provide a higher-level abstraction layer on top of MongoDB, making it easier for developers to work with document-oriented data.

Here's a table summarizing the difference between MongoDB and ODMs:

Feature	MongoDB	ODM
Type	NoSQL database	Object document mapper
Purpose	Stores data in documents	Manages data access and relationships in MongoDB
Functionality	Stores, retrieves, and manages data	Provides abstraction layer for developers
Popularity	Widely used, but requires some SQL knowledge	Less common, but easier for developers to use

[View other drafts](#)  [Export to Sheets](#)

In summary, MongoDB is the underlying database that stores data, while ODMs are libraries that simplify working with that data. ODMs are particularly useful for developers who are not familiar with SQL or who want to focus on application logic instead of database interactions.

Like Dislike Share Google More

```
const collection = database.collection("myCollection");

const document = { name: "John Doe", age: 30 };

await collection.insertOne(document);
```

Well, what's the difference?

View other drafts ▾

🔊

In a sense, interacting with MongoDB directly using the MongoDB driver does involve some ODM-like features, such as creating collections, inserting documents, and querying documents. However, it's important to distinguish between using the MongoDB driver directly and using a dedicated ODM like Mongoose.

An ODM provides a higher-level abstraction layer on top of the MongoDB driver, simplifying the process of working with document-oriented data. It offers several advantages over direct driver usage:

- 1. Schema Management:** ODMs typically provide schema management capabilities, allowing developers to define data models that represent the structure of their data. This helps maintain data consistency and prevent invalid or incompatible data from being stored in the database.
- 2. Fluent Query Builder:** ODMs often provide a fluent query builder, a tool that simplifies the process of constructing complex queries to retrieve and manipulate data. This eliminates the need to write raw SQL queries, making the code more readable and maintainable.
- 3. Automatic Type Conversion:** ODMs handle automatic type conversion between the application's objects and the database's documents. This ensures that data is correctly interpreted and stored without manual type casting.
- 4. Relationship Management:** ODMs manage relationships between documents, making it easier to model complex data structures and efficiently connect related data. This is particularly useful for applications that handle hierarchical or interconnected data.

While direct driver usage provides more control over database interactions, ODMs offer significant advantages in terms of ease of use, development efficiency, and code maintainability. They are particularly well-suited for developers who are not familiar with raw SQL or who want a more streamlined approach to working with document-oriented data in MongoDB.

<https://g.co/bard/share/27290e7ec30d>

# Well, what's the difference?

Feature	Directly Using MongoDB Driver	Using an ODM
Schema management	Manual definition using JSON documents	Built-in schema management using data models
Query building	Constructing SQL queries manually	Using a fluent query builder for more concise and readable queries
Type conversion	Manual type casting required	Automatic type conversion between application objects and database documents
Relationship management	Managing relationships manually	Automating relationship management for complex data structures
Ease of use	More complex and requires manual coding	Easier to use and more developer-friendly
Development efficiency	Less efficient for complex data handling	More efficient for rapid development and code maintainability

 Export to Sheets

```
// Create a Mongoose schema for the "artists" collection
const artistSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Name of the artist is required and must be a string"],
    minlength: [1, "Name must be at least 1 character long"],
    maxlength: [255, "Name cannot exceed 255 characters"]
  },
  genre: {
    type: String,
    required: [true, "Genre of the artist is required and must be a string"],
    enum: {
      values: ["Pop", "Rock", "Hip-Hop", "Country", "Other"],
      message: "Invalid genre. Must be one of: Pop, Rock, Hip-Hop, Country, Other"
    }
  },
  image: {
    type: String,
    required: [true, "Image of the artist is required and must be a string"],
    minlength: [4, "Image must be at least 4 character long"]
  },
  birthdate: {
    type: Date,
    required: [true, "Birthdate of the artist is required and must be a date"]
  },
  gender: {
    type: String,
    required: [true, "Gender of the artist is required and must be a string"],
    enum: {
      values: ["Male", "Female", "Non-Binary", "Other"],
      message: "Invalid gender. Must be one of: Male, Female, Non-Binary, Other"
    }
  }
});

const Artist = mongoose.model("Artist", artistSchema);
```

# MongoDB versus Mongoose

```
// GET Endpoint "/artists" - get all artists
app.get("/artists", async (request, response) => {
  const artistsCollection = db.collection("artists");
  const artists = await artistsCollection.find().toArray(); // Use toArray() to retrieve documents as an array
  response.json(artists);
});

// GET Endpoint "/artists/:id" - get one artist
app.get("/artists/:id", async (request, response) => {
  const id = request.params.id;
  const artistsCollection = db.collection("artists");
  const artists = await artistsCollection.findOne({
    _id: new ObjectId(id)
  });
  response.json(artists);
});

// POST Endpoint "/artists" - create one artist
app.post("/artists", async (request, response) => {
  const artist = request.body;
  const result = await db.collection("artists").insertOne(artist);
  response.json(result);
});

// PUT Endpoint "/artists/:id" - update one artist
app.put("/artists/:id", async (request, response) => {
  const id = request.params.id;
  const artist = request.body;
  const result = await db.collection("artists").updateOne({ _id: new ObjectId(id) }, { $set: artist });
  response.json(result);
});

// DELETE Endpoint "/artists/:id" - delete one artist
app.delete("/artists/:id", async (request, response) => {
  const id = request.params.id;
  const result = await db.collection("artists").deleteOne({ _id: new ObjectId(id) });
  response.json(result);
});

https://github.com/cederdorff/musicbase-mongodb/tree/mongodb-schema-validation
```

```
// GET Endpoint "/artists" - get all artists
app.get("/artists", async (request, response) => {
  const artists = await Artist.find();
  response.json(artists);
});

// GET Endpoint "/artists/:id" - get one artist
app.get("/artists/:id", async (request, response) => {
  const artist = await Artist.findById(request.params.id);
  response.json(artist);
});

// POST Endpoint "/artists" - create one artist
app.post("/artists", async (request, response) => {
  const artist = new Artist(request.body);
  const result = await artist.save();
  response.json(result);
});

// PUT Endpoint "/artists/:id" - update one artist
app.put("/artists/:id", async (request, response) => {
  const artist = await Artist.findById(request.params.id);
  artist.set(request.body);
  const result = await artist.save();
  response.json(result);
});

app.delete("/artists/:id", async (request, response) => {
  const result = await Artist.deleteOne({ _id: request.params.id });
  response.json(result);
});

https://github.com/cederdorff/musicbase-mongodb/tree/mongoose-schema-validation
```

# MongoDB: JSON Schema Validation

- Schema validation in MongoDB refers to the process of ensuring that documents stored in collections adhere to a predefined schema.
- This ensures data consistency and prevents invalid or incompatible data from being inserted into the database.
- JSON Schema is a standard for defining the structure of JSON documents. It defines properties, their data types, and validation rules for each property.
- By defining a JSON Schema for a collection, developers can enforce data consistency and prevent invalid or incompatible documents from being inserted.

```
await db.createCollection("artists", {  
  validator: {  
    $jsonSchema: {  
      bsonType: "object",  
      required: ["name", "genre", "image", "birthdate", "gender"],  
      properties: {  
        name: {  
          bsonType: "string",  
          description: "Name of the artist is required and must be a string",  
          minLength: 1,  
          maxLength: 255  
        },  
        genre: {  
          bsonType: "string",  
          description: "Genre of the artist is required and must be a string",  
          enum: ["Pop", "Rock", "Hip-Hop", "Country", "Other"]  
        },  
        image: {  
          bsonType: "string",  
          description: "Image of the artist is required and must be a string",  
          pattern: "^https?://[a-zA-Z0-9-.~:/?#[\\]@!$&'()*)+,;=]+$"  
        },  
        birthdate: {  
          bsonType: "date",  
          description: "Birthdate of the artist is required and must be a date"  
        },  
        gender: {  
          bsonType: "string",  
          description: "Gender of the artist is required and must be a string",  
          enum: ["Male", "Female", "Non-Binary", "Other"]  
        }  
      }  
    }  
  }  
});
```

# MongoDB: JSON Schema Validation

## Required Properties:

- `name`: A string representing the artist's name, with a minimum length of 1 character and a maximum length of 255 characters.
- `genre`: A string representing the artist's genre, from the specified list of options: "Pop", "Rock", "Hip-Hop", "Country", or "Other".
- `image`: A URL string representing the artist's image. The URL must follow the pattern of a valid URL, starting with "https://" or "http://".
- `birthdate`: A date string representing the artist's birthdate.
- `gender`: A string representing the artist's gender, from the specified list of options: "Male", "Female", "Non-Binary", or "Other".

```
await db.createCollection("artists", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: ["name", "genre", "image", "birthdate", "gender"],
      properties: {
        name: {
          bsonType: "string",
          description: "Name of the artist is required and must be a string",
          minLength: 1,
          maxLength: 255
        },
        genre: {
          bsonType: "string",
          description: "Genre of the artist is required and must be a string",
          enum: ["Pop", "Rock", "Hip-Hop", "Country", "Other"]
        },
        image: {
          bsonType: "string",
          description: "Image of the artist is required and must be a string",
          pattern: "^(https?://[a-zA-Z0-9-.~:/?#%[\\\]@!$&'()*)+,*+$"
        },
        birthdate: {
          bsonType: "date",
          description: "Birthdate of the artist is required and must be a date"
        },
        gender: {
          bsonType: "string",
          description: "Gender of the artist is required and must be a string",
          enum: ["Male", "Female", "Non-Binary", "Other"]
        }
      }
    }
  }
});
```

# Mongoose Schema Validation

- Using ODMs: Object document mappers (ODMs) like Mongoose provide built-in schema validation capabilities.
- ODMs allow developers to define data models that represent the structure of their data, and they automatically validate documents against these models before inserting them into the database.

```
// ===== SCHEMAS & MODELS ===== //

// Create a Mongoose schema for the "artists" collection
const artistSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Name of the artist is required and must be a string"],
    minlength: [1, "Name must be at least 1 character long"],
    maxlength: [255, "Name cannot exceed 255 characters"]
  },
  genre: {
    type: String,
    required: [true, "Genre of the artist is required and must be a string"],
    enum: {
      values: ["Pop", "Rock", "Hip-Hop", "Country", "Other"],
      message: "Invalid genre. Must be one of: Pop, Rock, Hip-Hop, Country, Other"
    }
  },
  image: {
    type: String,
    required: [true, "Image of the artist is required and must be a string"],
    minlength: [4, "Image must be at least 4 character long"]
  },
  birthdate: {
    type: Date,
    required: [true, "Birthdate of the artist is required and must be a date"]
  },
  gender: {
    type: String,
    required: [true, "Gender of the artist is required and must be a string"],
    enum: {
      values: ["Male", "Female", "Non-Binary", "Other"],
      message: "Invalid gender. Must be one of: Male, Female, Non-Binary, Other"
    }
  }
});

const Artist = mongoose.model("Artist", artistSchema);
```

# Mongoose Schema Validation

Similar to the JSON Schema definition, the Mongoose code defines the required and optional properties for an artist document, along with their data types and validation rules.

## Required Properties:

- `name`: A string representing the artist's name, ensuring it is not null or an empty string.
- `genre`: A string representing the artist's genre, ensuring it is not null or an empty string and belongs to the specified list of options: "Pop", "Rock", "Hip-Hop", "Country", or "Other".
- `image`: A string representing the artist's image URL, ensuring it is not null or an empty string and has a minimum length of 4 characters.
- `birthdate`: A date representing the artist's birthdate, ensuring it is not null or an empty string and follows the date format expected by MongoDB.
- `gender`: A string representing the artist's gender, ensuring it is not null or an empty string and belongs to the specified list of options: "Male", "Female", "Non-Binary", or "Other".

By defining the schema using Mongoose, developers can create, update, and retrieve artist documents with confidence, knowing that they adhere to the specified validation rules. This helps maintain data integrity and ensures that only valid and consistent artist data is stored in the database.

```
// ===== SCHEMAS & MODELS ===== //

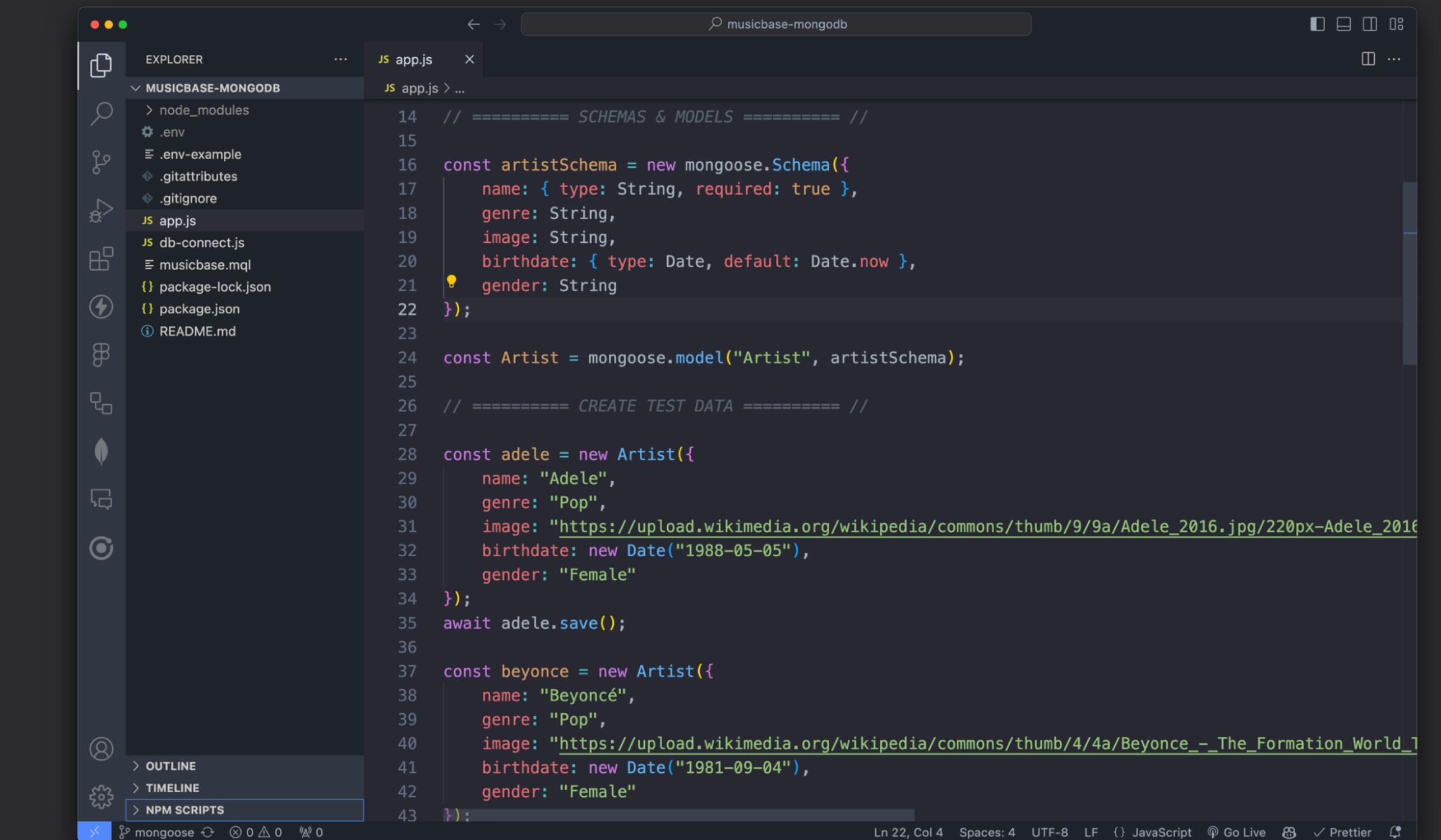
// Create a Mongoose schema for the "artists" collection
const artistSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, "Name of the artist is required and must be a string"],
    minlength: [1, "Name must be at least 1 character long"],
    maxlength: [255, "Name cannot exceed 255 characters"]
  },
  genre: {
    type: String,
    required: [true, "Genre of the artist is required and must be a string"],
    enum: {
      values: ["Pop", "Rock", "Hip-Hop", "Country", "Other"],
      message: "Invalid genre. Must be one of: Pop, Rock, Hip-Hop, Country, Other"
    }
  },
  image: {
    type: String,
    required: [true, "Image of the artist is required and must be a string"],
    minlength: [4, "Image must be at least 4 character long"]
  },
  birthdate: {
    type: Date,
    required: [true, "Birthdate of the artist is required and must be a date"]
  },
  gender: {
    type: String,
    required: [true, "Gender of the artist is required and must be a string"],
    enum: {
      values: ["Male", "Female", "Non-Binary", "Other"],
      message: "Invalid gender. Must be one of: Male, Female, Non-Binary, Other"
    }
  }
});

const Artist = mongoose.model("Artist", artistSchema);
```

# Øvelse

## Node.js, MongoDB og

## Mongoose



The screenshot shows a dark-themed code editor interface. On the left is the Explorer sidebar with a tree view of files in a folder named "MUSICBASE-MONGODB". The files listed include node\_modules, .env, .env-example, .gitattributes, .gitignore, app.js (which is currently selected), db-connect.js, musicbase.mql, package-lock.json, package.json, and README.md. Below the Explorer is the Outline, Timeline, and NPM Scripts panel. The main editor area displays a JavaScript file named "app.js". The code is written in Mongoose schema syntax to define an "Artist" model. It includes a schema for artists with fields like name, genre, image, birthdate, and gender, and two instances of the Artist model for Adele and Beyoncé.

```
// ===== SCHEMAS & MODELS =====
const artistSchema = new mongoose.Schema({
  name: { type: String, required: true },
  genre: String,
  image: String,
  birthdate: { type: Date, default: Date.now },
  gender: String
});

const Artist = mongoose.model("Artist", artistSchema);

// ===== CREATE TEST DATA =====
const adele = new Artist({
  name: "Adele",
  genre: "Pop",
  image: "https://upload.wikimedia.org/wikipedia/commons/thumb/9/9a/Adele_2016.jpg/220px-Adele_2016.jpg",
  birthdate: new Date("1988-05-05"),
  gender: "Female"
});
await adele.save();

const beyonce = new Artist({
  name: "Beyoncé",
  genre: "Pop",
  image: "https://upload.wikimedia.org/wikipedia/commons/thumb/4/4a/Beyonce_-_The_Formation_World_Tour.jpg/220px-Beyonce_-_The_Formation_World_Tour.jpg",
  birthdate: new Date("1981-09-04"),
  gender: "Female"
});
```

# Øvelse

## MongoDB Musicbase

The screenshot shows a dark-themed code editor interface. On the left, the Explorer sidebar displays a project structure for 'MUSICBASE-MONGODB' containing files like .env, .env-example, .gitattributes, .gitignore, app.js, db-connect.js, musicbase.mql, package-lock.json, package.json, and README.md. The main editor area shows a JavaScript file named 'app.js'. The code defines a mongoose schema for an 'Artist' model with fields for name, genre, image, birthdate, and gender. It then creates two test instances of the Artist model for Adele and Beyoncé, setting their names, genres, images (links to Wikipedia profile pictures), birthdates (new Date("1981-09-04") for Beyoncé and new Date("1988-05-05") for Adele), and genders. The code uses standard ES6 syntax and imports mongoose from the node\_modules.

```
// ===== SCHEMAS & MODELS =====
const artistSchema = new mongoose.Schema({
  name: { type: String, required: true },
  genre: String,
  image: String,
  birthdate: { type: Date, default: Date.now },
  gender: String
});

const Artist = mongoose.model("Artist", artistSchema);

// ===== CREATE TEST DATA =====
const adele = new Artist({
  name: "Adele",
  genre: "Pop",
  image: "https://upload.wikimedia.org/wikipedia/commons/thumb/9/9a/Adele_2016.jpg/220px-Adele_2016.jpg",
  birthdate: new Date("1988-05-05"),
  gender: "Female"
});
await adele.save();

const beyonce = new Artist({
  name: "Beyoncé",
  genre: "Pop",
  image: "https://upload.wikimedia.org/wikipedia/commons/thumb/4/4a/Beyonce_-_The_Formation_World_Tour.jpg/220px-Beyonce_-_The_Formation_World_Tour.jpg",
  birthdate: new Date("1981-09-04"),
  gender: "Female"
});
```

Ln 22, Col 4 Spaces: 4 UTF-8 LF {} JavaScript ⚡ Go Live 🎨 Prettier 🔍

# Indexes and search

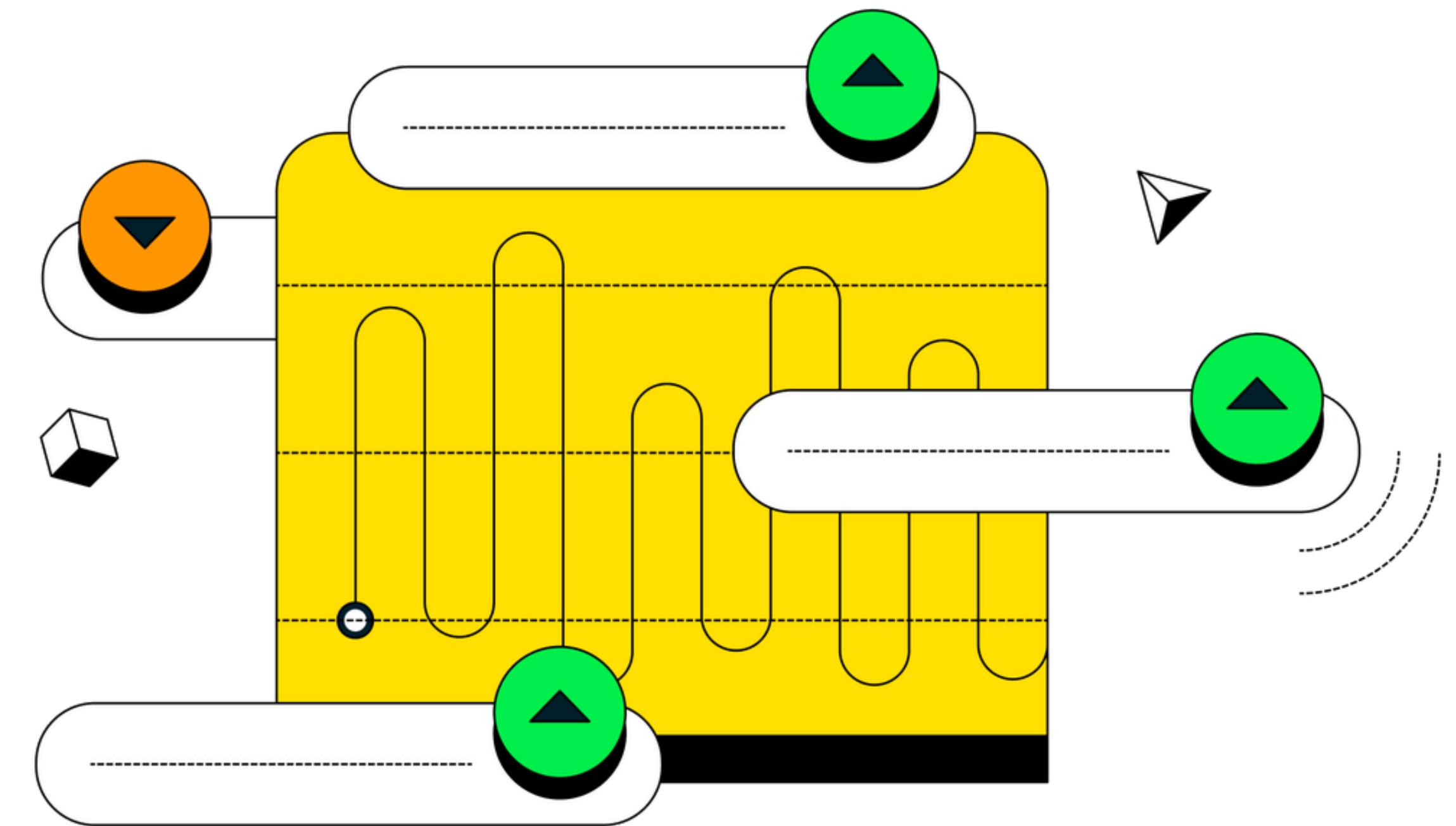




# What is an Index in MongoDB?

Indexes hold a small portion of the collection's data in a form that's easy to traverse. They are used to:

- Speed up queries and updates
- Avoid disk I/O as queries eliminating the need for slow collection scans
- Reduce overall computation



# Search Index

Index for Amounts						
amount: 2600	amount: 1800	amount: 1300	amount: 1200	amount: 1000	amount: 700	amount: 600

Raw Data						
{ date: "Jan 1", amount: 1000 }	{ date: "Feb 29", amount: 600 }	{ date: "Dec 24", amount: 700 }	{ date: "Mar 7", amount: 2600 }	{ date: "Jan 15", amount: 1200 }	{ date: "Apr 2", amount: 1300 }	{ date: "Mar 12", amount: 1800 }

Index for Dates						
date: "Dec 24"	date: "Jan 1"	date: "Jan 15"	date: "Feb 29"	date: "Mar 7"	date: "Mar 12"	date: "Apr 2"

Query for quarterly sales						

Index for Amounts						
amount: 2600	amount: 1800	amount: 1300	amount: 1200	amount: 1000	amount: 700	amount: 600

Raw Data						
{ date: "Jan 1", amount: 1000 }	{ date: "Feb 29", amount: 600 }	{ date: "Dec 24", amount: 700 }	{ date: "Mar 7", amount: 2600 }	{ date: "Jan 15", amount: 1200 }	{ date: "Apr 2", amount: 1300 }	{ date: "Mar 12", amount: 1800 }

Index for Dates						
date: "Dec 24"	date: "Jan 1"	date: "Jan 15"	date: "Feb 29"	date: "Mar 7"	date: "Mar 12"	date: "Apr 2"

Query for top 3 sales						

<https://www.mongodb.com/basics/search-index>

# Index - example

- This code creates an index on the uid field in the posts collection.
- An index is a data structure that improves the performance of queries that filter or sort based on the indexed field. In this case, the index will improve the performance of queries that filter posts based on the user ID.
- The query `db.posts.find({uid: ObjectId("ZfPTVEMQKf9v")})` will be much faster if there is an index on the uid field, as MongoDB can directly identify the documents associated with the specified user ID. This is in contrast to a scenario where there is no index on the uid field, where MongoDB would have to scan the entire collection to locate the matching documents.

```
db.posts.insertMany([
  {
    caption: "A beautiful morning in Aarhus",
    createdAt: new Date("2023-04-06T09:10:54Z"),
    image: "https://images.unsplash.com/photo-1573997953524-ef",
    uid: ObjectId("HlvRHz58C05g")
  },
  {
    caption: "Rainbow reflections of the city of Aarhus",
    createdAt: new Date("2023-04-02T20:25:34Z"),
    image: "https://images.unsplash.com/photo-1558443336-dbb3c",
    uid: ObjectId("fjpRTjZHwr")
  }
])
// create index
db.posts.createIndex({ uid: 1 });
// posts with specific user
db.posts.find({uid: ObjectId("ZfPTVEMQKf9v")})
```

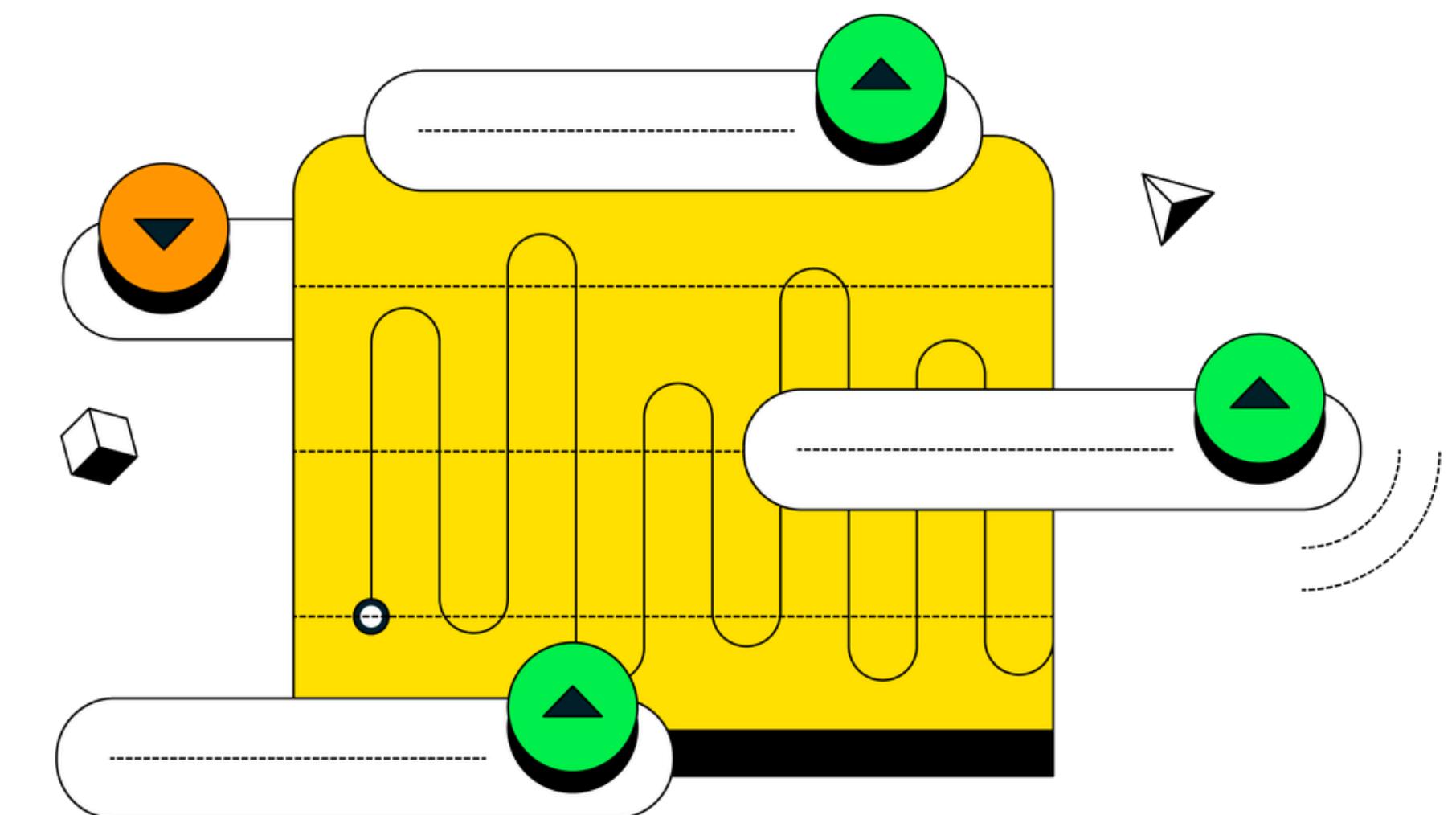


# When to Use Indexes

Developers should use an index when querying data in a collection, especially for frequently run queries.

When determining if an index should be used, consider:

- Four indexes is a good rule of thumb for the ideal number for a given collection.
- Sixty-four indexes are the maximum per collection, however, above 20 and performances renders the system almost unusable for workloads.





# Considerations When Using Indexes

Indexes require RAM.

Avoid unnecessary indexes at all cost, otherwise the write performance will suffer. Each index adds 10% overhead.

When does an index entry get modified?

- Data is inserted (applies to all indexes).
- Data is deleted (applies to all indexes).
- Data is updated in such a way that its indexed field changes.



# Types of Indexes Available

Most common indexes:

- Single Field
- Compound Index

Other types of specialized indexes including:

- Multikey Index
- Geospatial Index
- Text Index
- Hashed Index
- Time-To-Live (TTL) Index
- Hidden Index
- Partial Index
- Wildcard Index

A screenshot of a web browser window titled "Eksamensbeskrivelse: AU-VD". The URL is "eaaa.instructure.com/courses/189...". The page content includes a large heading "Eksamensbeskrivelse" and a section "Introduktion". It describes the task of developing a database application (server application) that can handle large amounts of data and perform CRUD operations (Create, Read, Update, Delete) on data and allow for searching based on various criteria. It also specifies that the solution should consist of a database (MySQL or MongoDB) and a server application implemented in Node.js or Python that interacts with the database. A horizontal line separates this from the next section.

# Eksamensbeskrivelse

## Introduktion

I denne eksamsopgave skal du udvikle en databaseapplikation (serverapplikation), der kan håndtere en større mængde data. Applikationen skal kunne udføre CRUD-operationer (Create, Read, Update, Delete) på dataene og muliggøre søgning baseret på forskellige kriterier.

Løsning skal bestå af

1. en database (MySQL eller MongoDB) efter eget valg, og
2. en serverapplikation implementeret med Node.js eller Python, der interagerer med databasen.

---

A screenshot of a web browser window titled "Eksamensbeskrivelse: AU-VD". The URL is "eaaa.instructure.com/courses/189...". The page content includes a section "Mundtlig eksamen" which states that the oral examination has a duration of 30 minutes and consists of three parts: an oral presentation (max 10 min), examination dialogue (15 min), and voting and feedback (5 min). It also notes that the oral presentation should include a demonstration of the solution. A horizontal line separates this from the next section.

## Mundtlig eksamen

Den mundtlige eksamen har i alt en varighed på 30 minutter og består af :

- et mundtligt oplæg (maks 10 min)
- eksaminationssamtale (15 min)
- votering og tilbagemelding af karakter (5 min)

Dit mundtlige oplæg skal inkludere en demonstration af din løsning. Eksaminationssamtalen vil tage udgangspunkt i dispositionen og det praktiske produkt. Der er intern censur ved den mundtlige prøve.

Den mundtlige eksamen afholdes d. 19. december 2023. Eksamensplan følger.

---

## Bedømmelse

Bedømmelsen vil tage udgangspunkt i en samlet helhedsvurdering af det praktiske produkt, dispositionen og den mundtlige præstation. Der gives en samlet karakter efter 7-trinsskalaen.

---

[◀ Previous](#) [Next ▶](#)

# Code every Day

