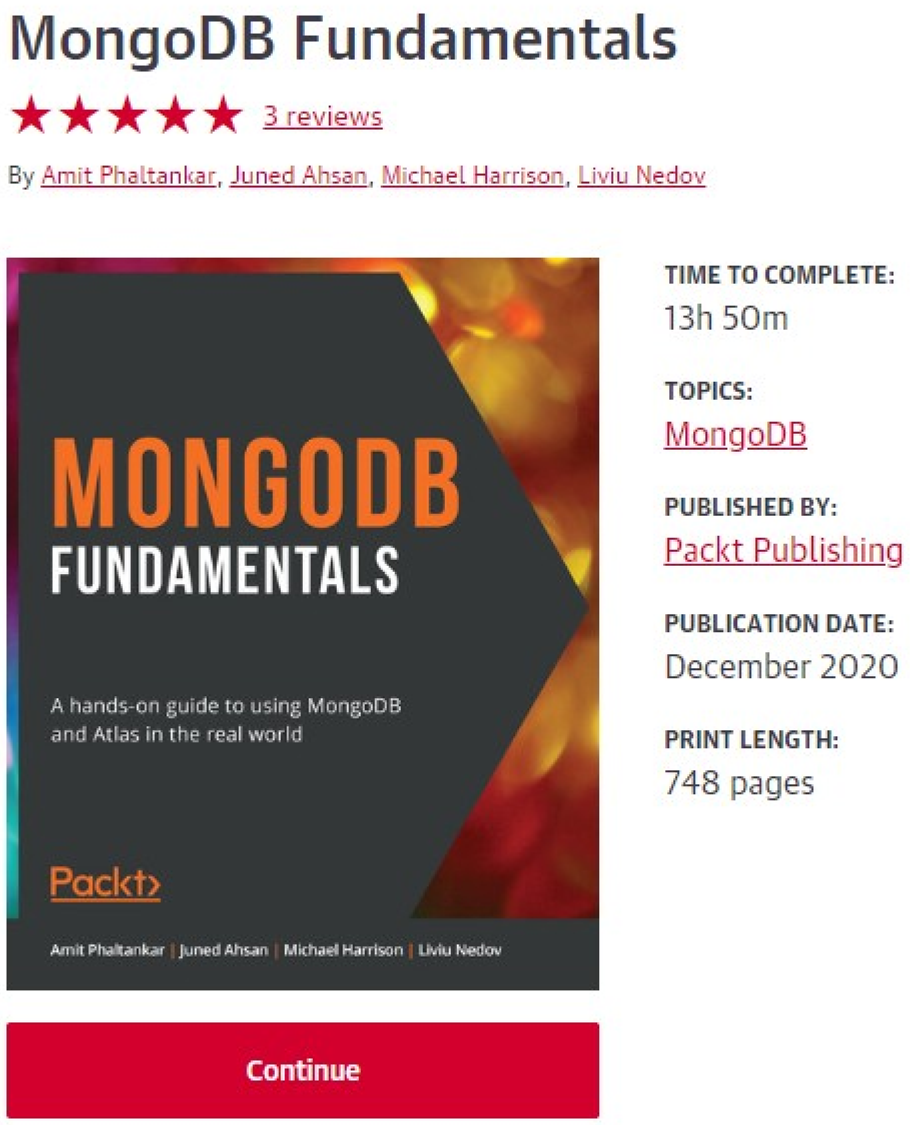
Web Programming
JavaScript – node.js - react

Lecture 7-1:
MongoDB
07.12.

Stefan Noll
Manuel Fehrenbach
Winter Semester 22/23



MongoDB Fundamentals

★★★★★ 3 reviews

By Amit Phaltankar, Juned Ahsan, Michael Harrison, Liviu Nedov

MONGODB
FUNDAMENTALS
A hands-on guide to using MongoDB and Atlas in the real world
Packt>
Amit Phaltankar | Juned Ahsan | Michael Harrison | Liviu Nedov
Continue

TIME TO COMPLETE:
13h 50m

TOPICS:
MongoDB

PUBLISHED BY:
Packt Publishing

PUBLICATION DATE:
December 2020

PRINT LENGTH:
748 pages

- Introduction MongoDB

- Currently databases are divided into two categories.
  - **Non-relational** databases or **NoSQL** database
  - **Relational** databases
- NoSQL databases are used to store large quantities of complex and diverse data like **product catalogs, logs, analytics**, **user interactions, etc.**
- MongoDB is one of the most established NoSQL databases.

  It also follows the **ACID** rules (**Atomicity, Consistency, Isolation, Durability**)
- MongoDB provides essential and extravagant features to store real-world big data

- Essential and extravagant features (MongoDB):

- Flexible and Dynamic Schema:

MongoDB you can use flexible schemas for your data. A flexible schema allows variance in fields in different documents. Simple: A record in the database may or may not have the same number of attributes. You can store evolving data without making any changes to the schema itself.

- Rich Query Language:

You can make simple yet powerfule queries. You can group and filter data as required.

Built-in support for general-purpose text search and specific purpose like geospatial searches

- Multi-Document ACID Transactions:

Features that allow your data to be stored and updated to maintain its accuracy.

Atomicity: means all or nothing (either all operations are part of a tansaction or none)

Consistency: Means keeping the data consistent as per the rules defined for the database.

Isolation: Changes to data appear only after all the operations are executed and are fully comitted

Durability: Ensures that changes are committed by the transaction even in case of a system crash

- High Performance:

MongoDB provides high performance using embedded data models to reduce disk I/O usage.

- High Availability:

MongoDB supports distributed clusters with a minimum of three nodes.

A cluster is a database deployment that uses multiple nodes/machines for data storage and retrieval

- Scalability:

MongoDB provides a way to scale the database horizontally across hundreds of nodes.

- Basic Elements of MongoDB:

Databases are basically aggregations of collections, wich in turn, are made of documents.

- Documents

MongoDB stores data in documents. A document is a **collection** of **field names** and **values** and structured in a JSON like format. Through this it is easy to read and to understand as JSON consist of easy to understand key-value pairs to describe data. Documents in MongoDB are stored as an extension of the JSON type called **BSON (Binary JSON)**

- Document Structures

Documents contain fields and value pairs and follow a basic structure:

```
{
    ”field1”:value1,
    ”field2”:value2,
    ...
    ”fieldN”:valueN
}
```

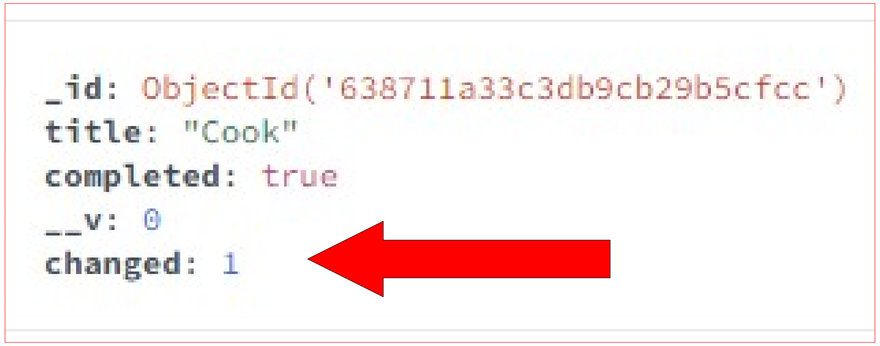
- Collections:

Documents are stored in **collections**. Collections are analogous to tables in relational databases.

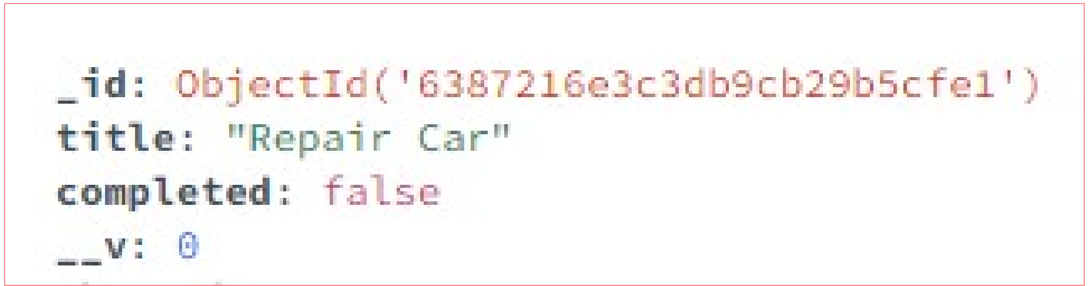
Within queries (insert, retrieve, delete, etc.) you need to use the collection name

- Documents in Detail

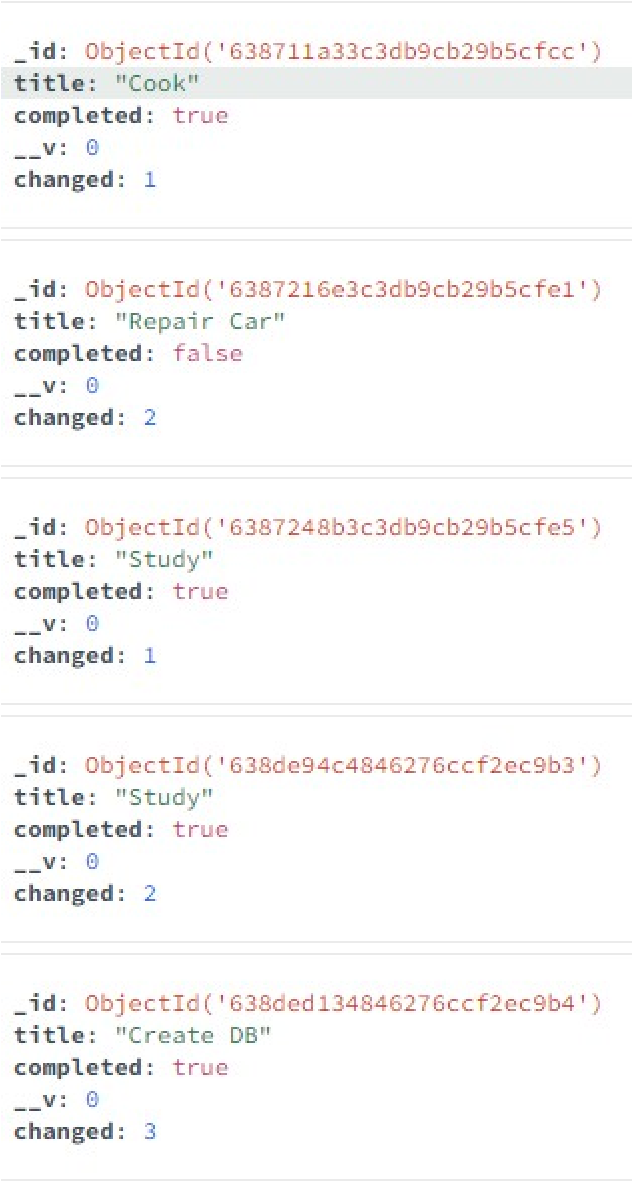
- With tabular data models like in SQL you can’t support complex data structures
    - e.g. nested objects or collection of objects and the data needs to be split in multiple tables
- In MongoDB you can store more complex data structures because of JSON-like format.
- Major Features of MongoDB document-based data model:
    - flexible and natural way of representing data
    - objects, arrays, etc. in a document are relatable to the object structure in programming language
- Through flexible schemas documents are agile. Schema can change or update without major downtime
- Documents are self-contained pieces of data. Alle data within one document instead multiple tables.
- Documents are extensible. Documents can be used to store entire object structures, as a map, key-value pair lookup or flat structure that resembles a relational table.
- Size schould not exceed 16MB
- Nesting limit is 100 levels
- The fields shown after a query is known as a **projection,** in MongoDB you can change the **projection**



```
_id: ObjectId('638711a33c3db9cb29b5cfcc')
title: "Cook"
completed: true
__v: 0
changed: 1
```



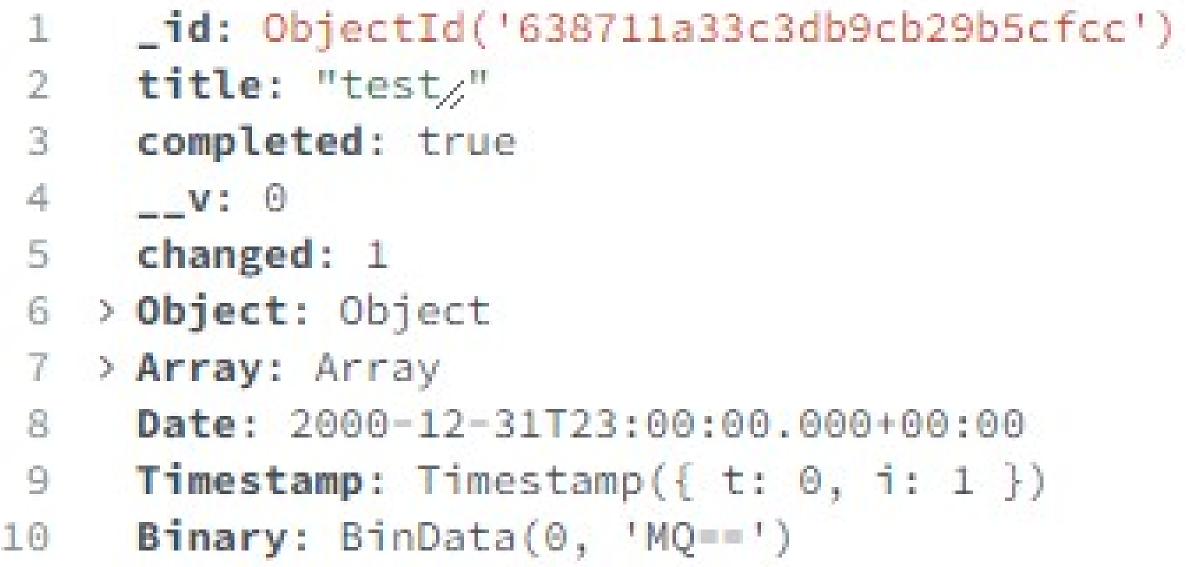
```
_id: ObjectId('6387216e3c3db9cb29b5cfe1')
title: "Repair Car"
completed: false
__v: 0
```



```
_id: ObjectId('638711a33c3db9cb29b5cfcc')
title: "Cook"
completed: true
__v: 0
changed: 1

_id: ObjectId('6387216e3c3db9cb29b5cfe1')
title: "Repair Car"
completed: false
__v: 0
changed: 2

_id: ObjectId('6387248b3c3db9cb29b5cfe5')
title: "Study"
completed: true
__v: 0
changed: 1

_id: ObjectId('638de94c4846276ccf2ec9b3')
title: "Study"
completed: true
__v: 0
changed: 2

_id: ObjectId('638ded134846276ccf2ec9b4')
title: "Create DB"
completed: true
__v: 0
changed: 3
```

- **MongoDB Data Types**
    - Strings
        - textbased fields
        - UTF-8 encoded
        - Value wrapped in double quotes is considered a string
    - Numbers
        - double: 64-bit floating point
        - int: 32-bit signed integer
        - long: 64-bit unsigned integer
        - decimal: 128-bit floating point – which is IEE 754-compliant
    - Booleans
    - Objects
        - nested objects can be accessed through **.** notation
    - Arrays
        - unlimited number of entries, but document size should not exceed 16MB
        - nested objects can be accessed through **.** notation

- **MongoDB Data Types**
  - ObjectId
    - Id of the document
    - Created from MongoDB with each new document
  - Dates
    - MongoDB dates are stored in the form of milliseconds since the Unix epoch,
  - Timestamp
    - The timestamp is a 64-bit representation of date and time
  - Binary Data
    - Binary data, also called BinData, is a BSON data type for storing data that exists in a binary format

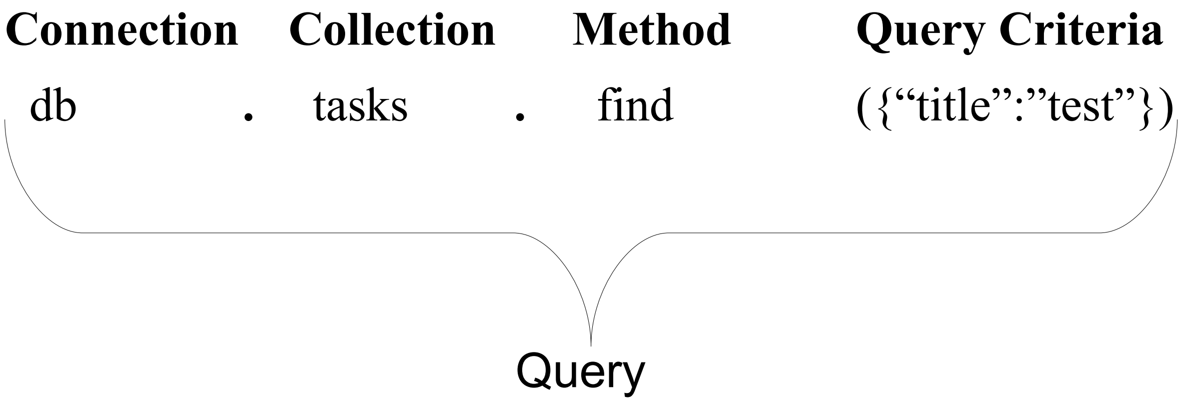


```
  _id: ObjectId('638711a33c3db9cb29b5cfcc')
  title: "test"
  completed: true
  __v: 0
  changed: 1
> Object: Object
> Array: Array
  Date: 2000-12-31T23:00:00.000+00:00
  Timestamp: Timestamp({ t: 0, i: 1 })
  Binary: BinData(0, 'MQ==')
```

- Queries in MongoDB

- Queries in MongoDB are based on JSON documents in which you write your criteria in the form of valid JSON-like documents.

- Structure of a query in MongoDB:



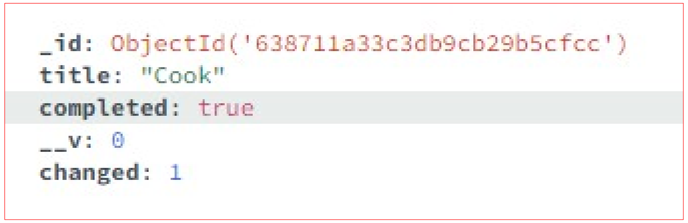
Connection		Collection		Method	Query Criteria
db	.	tasks	.	find	({“title”:”test”})

Query

- Basic Queries:
  - find(), findOne(), count(), countDocuments()
- Conditional Operators
  - equals ($eq), not equal to ($ne), greater than ($gt) or equal ($gte), less than ($lt) or equal ($lte), in ($in), not in ($nin)
- Logical Operators
  - and ($and), or ($or), nor ($nor), not ($not)

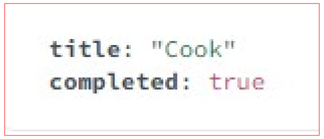
- **find()**
  - If this function is executed without any arguments, it returns all documents in a collection
  - **e.g. db.tasks.find()**

  - If only specific documents should be returned, you can add a condition to the **find()** method.
    The condition is a JSON-like Object
  - **e.g. db.tasks.find({„title“ : “Cook“})**
    - The query above returns all tasks with the title equals test

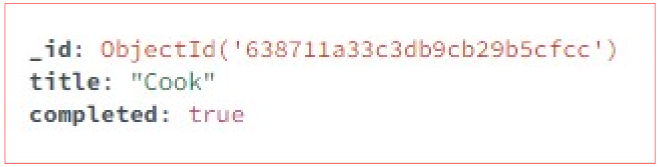


```
_id: ObjectId('638711a33c3db9cb29b5cfcc')
title: "Cook"
completed: true
__v: 0
changed: 1
```

  - You can also choose the fields for the Output also known as **projection**.
    This will work for the **find()** and **findOne()** method
  - **e.g. db.tasks.find({„title“:“test“}, {„title“:1,“completed“:1,“_id“:0})**
    - In the query above we still get the same task, but now we will only get the selected fields.
    - If we specify the fields we want, all other fields not mentioned are excluded except **_id**
    - If we don’t want the id field, we need to exclude it explicit like: **„_id“:0**



```
title: "Cook"
completed: true
```



```
_id: ObjectId('638711a33c3db9cb29b5cfcc')
title: "Cook"
completed: true
```

- findOne()

- This method returns only one matching record.
- Useful when looking to isolate a specific record.
- Syntax is similar to **find()**
- Can also have conditions and filters for fields
- **e.g. db.tasks.findOne()**

- Valid for find() and findeOne()

- If you use mongo shell you get also the functions **next()** and **hasNext()**
- next() returns the next document and hasNext() checks if there is still a document left
- This is possible as you have access to the current cursor, which points to the current document
- **e.g. var tasks = db.tasks. find({„completed“:true})**

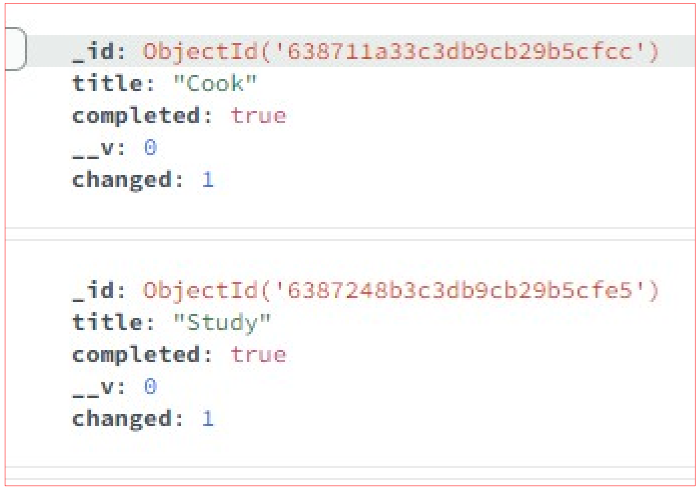
  tasks.next()

  tasks.hasNext()

- **count()**
    - Returns the count of documents in a collection
    - Will not count all documents, but read through the collection’s metadata and return count
        - No guarantee that the metadata are correct
    - **e.g. db.tasks.count()**

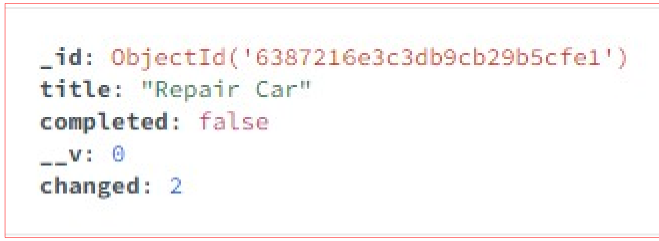
- **countDocuments()**
    - Returns the count of documents that are matched by the given condition.
    - Will never use collection metadata to find the count
    - A condition is mandatory unlike in
    - **e.g. db.tasks.countDocuments({})**

- **Equals ($eq)**
  - You can also use dedicated operators as $eq to find documents with fields that match a given value
  - **e.g. db.tasks.find({"completed":{$eq:true}})**



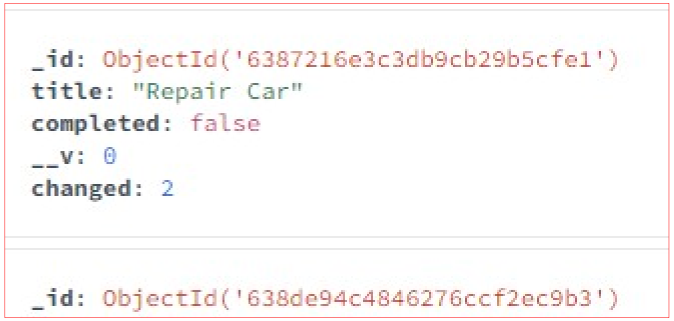
```
_id: ObjectId('638711a33c3db9cb29b5cfcc')
title: "Cook"
completed: true
__v: 0
changed: 1

_id: ObjectId('6387248b3c3db9cb29b5cfe5')
title: "Study"
completed: true
__v: 0
changed: 1
```

- **Not Equal To ($ne)**
  - reverse effect of using equals ($eq)
  - selectes all the documents where the value of the field doesn’t match
  - **e.g. db.tasks.find({"completed":{$ne:true}})**



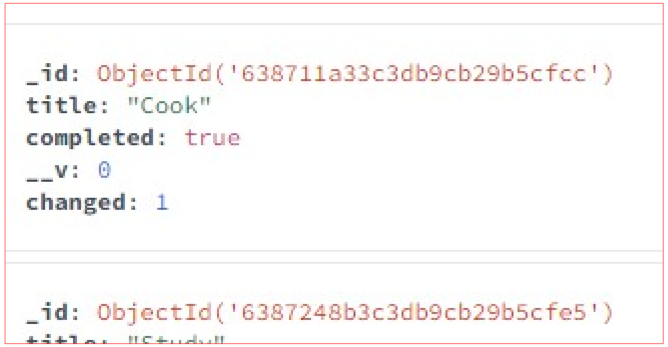
```
_id: ObjectId('6387216e3c3db9cb29b5cfe1')
title: "Repair Car"
completed: false
__v: 0
changed: 2
```

- **Greater Than ($gt) and Greater Than or Equal To ($gte)**
  - Find documents where the value of the field is greater than / greater than or equal to the value in the field
  - **e.g. db.tasks.find({"changed":{$gt:1}})**



```
_id: ObjectId('6387216e3c3db9cb29b5cfe1')
title: "Repair Car"
completed: false
__v: 0
changed: 2

_id: ObjectId('638de94c4846276ccf2ec9b3')
```

- **Less Than ($lt) and Less Than or Equal To ($lte)**
  - Find documents where the value of the field is less / less or equal
  - **e.g. db.tasks.find({"changed":{$lt:2}})**

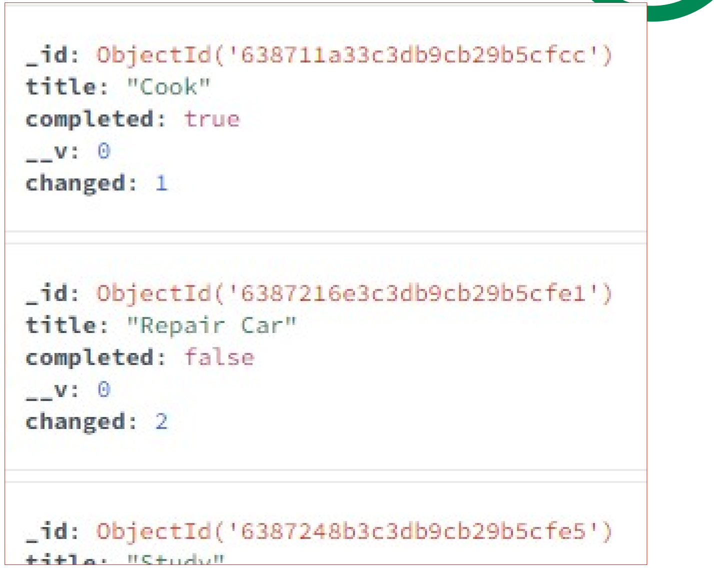


```
_id: ObjectId('638711a33c3db9cb29b5cfcc')
title: "Cook"
completed: true
__v: 0
changed: 1

_id: ObjectId('6387248b3c3db9cb29b5cfe5')
title: "Study"
```

- In ($in) and Not In ($nin)

- If you want all documents that have specific values in a field, $in can be used

- e.g. db.tasks.find({"changed":{$in:[1,2]}})

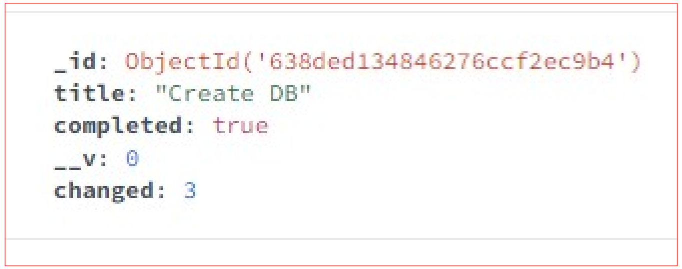


```
_id: ObjectId('638711a33c3db9cb29b5cfcc')
title: "Cook"
completed: true
__v: 0
changed: 1

_id: ObjectId('6387216e3c3db9cb29b5cfe1')
title: "Repair Car"
completed: false
__v: 0
changed: 2

_id: ObjectId('6387248b3c3db9cb29b5cfe5')
title: "Study"
```

or

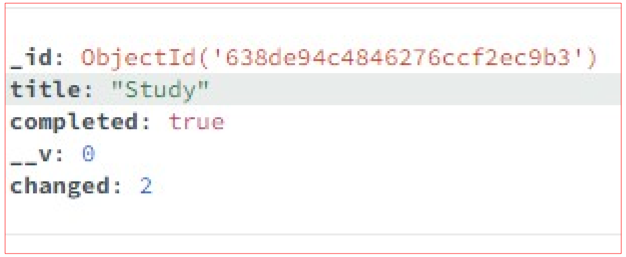
- e.g. db.tasks.find({"changed":{$nin:[1,2]}})

- opposite of $in / exclude all where the value is in the field



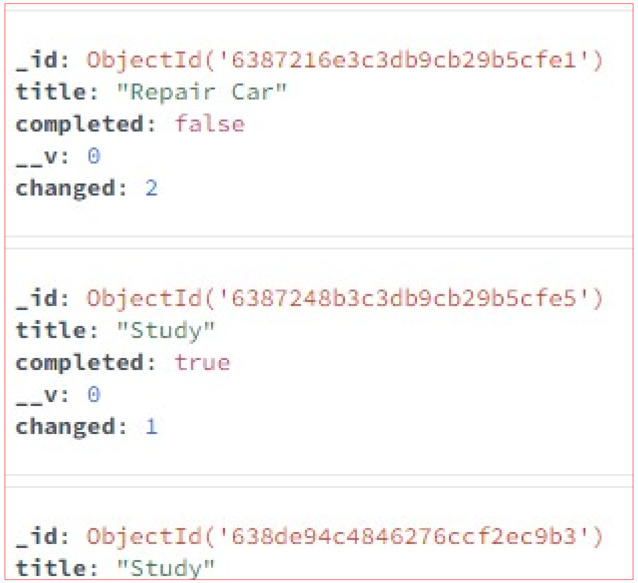
```
_id: ObjectId('638ded134846276ccf2ec9b4')
title: "Create DB"
completed: true
__v: 0
changed: 3
```

- **Logical operators**
    - **$and**
        - Using the $and operator you can have any number of conditions wrapped in an array and the operator will return only the documents that satisfy all the conditions
        - **e.g. db.tasks.find({$and:[{"title":"Study"},{"changed":2}]})**



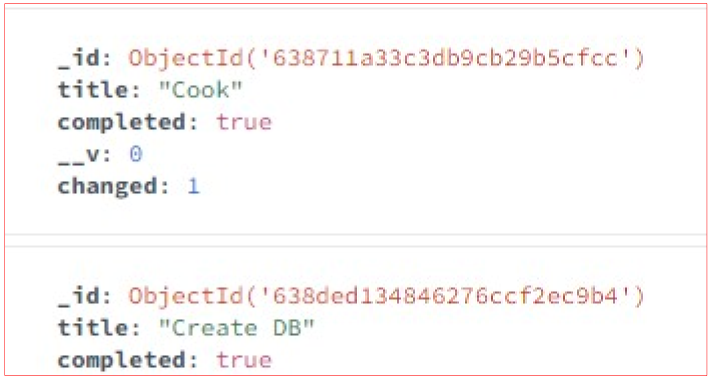
```
_id: ObjectId('638de94c4846276ccf2ec9b3')
title: "Study"
completed: true
__v: 0
changed: 2
```

- **$or**
    - If any codition is satisfied the document will be returned
    - can be mixed up with **$in** but **$or** and **$in** are different and used in different scenarios. **$in** is used to determine whether a given field has at least one of the values provided in an array. **$or** is not bound to any specific fields
    - **e.g. db.tasks.find({$or:[{"title":"Study"},{"changed":2}]})**



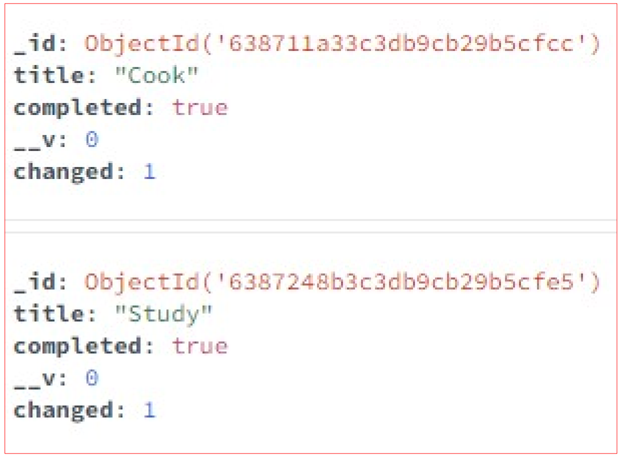
```
_id: ObjectId('6387216e3c3db9cb29b5cfe1')
title: "Repair Car"
completed: false
__v: 0
changed: 2

_id: ObjectId('6387248b3c3db9cb29b5cfe5')
title: "Study"
completed: true
__v: 0
changed: 1

_id: ObjectId('638de94c4846276ccf2ec9b3')
title: "Study"
```

- **Logical operators**
  - **$nor**
    - syntactically like **$or** but behaves in the opposite way.
    - **e.g. db.tasks.find({$nor:[{"title":"Study"},{"changed":2}]})**



```
_id: ObjectId('638711a33c3db9cb29b5cfcc')
title: "Cook"
completed: true
__v: 0
changed: 1

_id: ObjectId('638ded134846276ccf2ec9b4')
title: "Create DB"
completed: true
```

  - **$not**
    - **$not** represents the logical NOT operator that negates the given condition
    - Accepts a conditional expression and matches all the documents that do not satisfy it.
    - **e.g. db.tasks.find({"changed":{$not:{$gte:2}}})**



```
_id: ObjectId('638711a33c3db9cb29b5cfcc')
title: "Cook"
completed: true
__v: 0
changed: 1

_id: ObjectId('6387248b3c3db9cb29b5cfe5')
title: "Study"
completed: true
__v: 0
changed: 1
```

- **Inserting, Updating and Deleting Documents**

  - As with SQL you can also insert, update, and delete documents (table entries in SQL)

Inserting	Deleting	Replacing	Modify/Updating
insert()	deleteOne()	replaceOne()	updateOne()
insertMany()	deleteMany()	findOneAndReplace()	updateMany()
	findOneAndDelete()		findOneAndUpdate()

- **Inserting:**

  - insert()

    - used to create a new document
    - uses the document that should be inserted as an argument
    - If no **_id** field is provided MongoDB creates this field automatically
    - If you use an **id** that does already exists you get an error
    - An **id** needs to be unique
    - If insert() is called and the collection doesn’t exists, a new collection is created
    - Syntax: **db.collection.insert({document to be inserted})**
    - **e.g. db.tasks.insert({"title":"Cook", "completed":true, "changed":1}) or**
    - **e.g. db.tasks.insert({"_id":1,"title":"Cook", "completed":true, "changed":1})**

- **Inserting:**

  - insertMany()

    - used to create a new document
    - uses the document that should be inserted as an argument
    - If no **_id** field is provided MongoDB creates this field automatically
    - If insert() is called and the collection doesn’t exists, a new collection is created
    - Batch insert should not exceed 100k
    - If some documents have the same **id**, the other documents will be inserted anyway except for the documents with duplicate **id**s
    - Syntax: **db.collection.insertMany([{array of documents to be inserted}])**
    - **e.g. db.tasks.insert([**

```
{"title":"Test", "completed":true, "changed":1},
{"title":"Cleaning", "completed":false, "changed":0},
])
```

- **Deleting:**

  - deleteOne()

    - used to delete a single document

    - Accepts a document representing a query condition

    - If execution was successful it returns a document containing the total number of documents deleted (**deletedCount**) and confirmation through acknowledged

    - Syntax: **db.collection.deleteOne({document/query condition})**

    - e.g. db.tasks.deleteOne({"_id":1})

      - returns: { "acknowledged" : true, "deletedCount" : 1 }

- **Deleting:**
  - deleteMany()
    - used to delete multiple documents that match the criteria
    - Accepts a document representing a query condition
    - If execution was successful it returns a document containing the total number of documents deleted (**deletedCount**) and confirmation through acknowledged
    - Syntax: **db.collection.deleteMany({document/query condition})**
    - **e.g. db.tasks.deleteMany({"changed":1})**
      - **returns: { "acknowledged" : true, "deletedCount" : 2 }**

- db.collection.deleteMany({})

- deletes all documents

- db.collection.deleteOne({})

- deletes the first document found

- **Deleting:**
  - findOneAndDelete()
    - find a documents and deletes it.
    - behaves similarly like the deleteOne() function, but provides more options
      - It finds one document and deletes it
      - If more than one document is found, the first one will be deleted
      - Once deleted, the deleted document will be the response
      - In case multiple matches, **sort** option can be used to influence which document gets deleted
      - Projection can be used to include or exclude fields from the response
    - Syntax: **db.collection.findOneAndDelete({document/query condition})**
    - **e.g. db.tasks.findOneAndDelete({"changed":1})**
      - **returns: { "_id" : "6387...", "title" : "Cook", "completed":true, "__v":0,"changed":1 }**

- **Replacing:**

  - Replace incorrectly inserted document in a collection.

  - Data stored in documents is changed over time or change how the document is structured to support new requirements.

  - replaceOne()

    - replace a single document

    - accepts a query filter and a replacement document

    - finds the **first** document that matches the filter and replaces it with the provided document

    - **_id** does not need to be included in the replacement document, as **_id** is immutable

    - Syntax: **db.collection.replaceOne({query condition},{replacement document})**

    - **e.g. db.tasks.replaceOne({"_id":"6387...fcc"},**

      { "title" : "Learn MongoDB",

      "completed":true, "__v":0,"changed":1 })

      - **returns: { "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }**

- **Replacing / Upsert:**

  - Upsert means:

    - Replace if document exists, if not insert a new document

    - update (if found) or insert (if not found)

    - Syntax: **db.collection.replaceOne({query condition},{replacement document},{upsert:true})**

    - **e.g. db.tasks.replaceOne({"_id":"6387...fcc"},**

```
{ "title" : "Learn MongoDB",
  "completed":true, "__v":0,"changed":1 },
  {upsert:true})
```

- **Replacing:**
  - findOneAndReplace()
    - replace a single document
    - same as repalceOne() only with more options:
      - if more than one matches first document found will be replaced
      - A **sort** option can be used to influence which document gets replaced
      - by default returns the original document
      - With **{returnNewDocument:true}** set, the newly added document will be returned.
      - Field projection can be used to include or exclude specific fields.
    - Syntax: **db.collection.findOneAndReplace({query condition},{replacement document})**
    - **e.g. db.tasks.findOneAndReplace({"title":"Cook"},**

```
db.tasks.findOneAndReplace({"title":"Cook"},
                           { "title" : "Learn MongoDB",
                             "completed":true, "__v":0,"changed":1 },
                             {sort:{"_id":-1}, projection:{"_id":0},returnNewDocument:true})
```

- **Modify/Updating:**
  - With replace function a document will be completly replaced with a new document
  - However in most cases updates only affect one or few fields within a document.
  - Replacement is useful if all or most fields of a document are modified
  - With smale documents replacing them is still ok but with larger documents bulky and error prone
  - updateOne()
    - update a single document and the first that matches the query condition
    - accepts query condition and a document that specifies the field-level update expressions, last parameter is for options and optional
    - **_id** field can’t be updated
    - Syntax: **db.collection.updateOne({query condition},{update expression},{options})**
    - **e.g. db.tasks.updateOne({"title":"Cook"},**

      { $set: {"title" : "Learn MongoDB"}})

      - **returns: { "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }**
    - **e.g. db.tasks.updateOne({"title":"Cook"},**

      { $set: {"title" : "Learn MongoDB","changed" : 200 }})

- **Updating / Upsert:**

  - Upsert means:

    - Update if document exists, if not insert a new document

    - update (if found) or insert (if not found)

    - Syntax: **db.collection.updateOne({query condition},{update expression},{options})**

    - **e.g. db.tasks.updateOne({"title":"Cook"},**

      { $set: {"title" : "Learn MongoDB","changed" : 200 }},

      {upsert:true})

- **Updating:**
  - findOneAndUpdate()
    - same as updateOne() only with more options:
      - first two arguments are mendatory
      - returns by default the old document
      - with the options expression we can modify the fields we want to be returned
    - Syntax: **db.collection.findOneAndUpdate({query condition},{update document},{options})**
    - **e.g. db.tasks.findOneAndUpdate({"title":"Cook"},**

      { $set: {"title" : "Learn MongoDB","changed" : 200 })
    - **e.g. db.tasks.findOneAndUpdate({"title":"Cook"},**

      { $set: {"title" : "Learn MongoDB","changed" : 200 },

      {"returnNewDocument" : true})
    - **e.g. db.tasks.findOneAndUpdate({"title":"Cook"},**

      { $set: {"title" : "Learn MongoDB","changed" : 200 },

      {"projection" : {"_id" : 0},"returnNewDocument" : true})

- **Updating:**

  - updateMany()

    - perform same update process on many documents
    - first two arguments of the function are mandatory
    - like in updateOne(), findOneAndUpdate() if a field does not exist it will be added
    - Syntax: **db.collection.updateMany({query condition},{update document},{options})**
    - **e.g. db.tasks.updateMany({"changed":2},**

      { $set: {"outdated" : true}})

      - **returns: { "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }**

- **Install MongoDB and Server setup:**
  - Download MongoDB Community Server and install the software:

    https://www.mongodb.com/try/download/community
  - Download MongoDB Compass and install the software:

    https://www.mongodb.com/try/download/compass
  - After installing MongoDB and MongoDB Compass, start MongoDB Compass

    In the URI textarea insert: **mongodb://localhost:27017**

    If MongoDB was installed correctly and the mongoDB service is running,

    you should be able to connect to the MondoDB server after clicking **Connect**
  - Create a new Database **todo** and **tasks** as a collection name
  - Open the node.js project Server from exercise 4

    and install the node.js package mongoose for this project

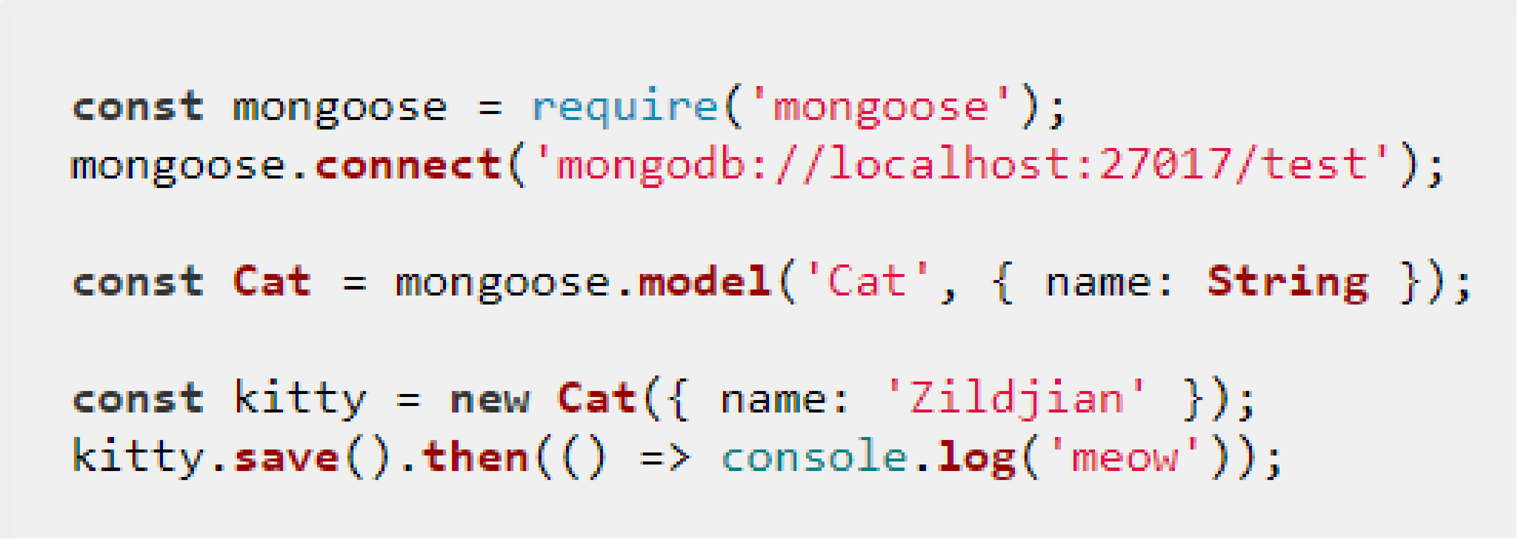
    npm i mongoose

- Possible solution if you can’t connect to the MongoDB:
  - Windows within terminal (if you close the terminal MongoDB will stop):
    - Start terminal as Administrator
    - cd C:\Program Files\MongoDB\Server\6.0\bin
    - mongod -f "C:\Program Files\MongoDB\Server\6.0\bin\mongod.cfg"
  - Linux/MacOs:

```
service mongod status
service mongod start
service mongod stop
```

- Why mongoose?

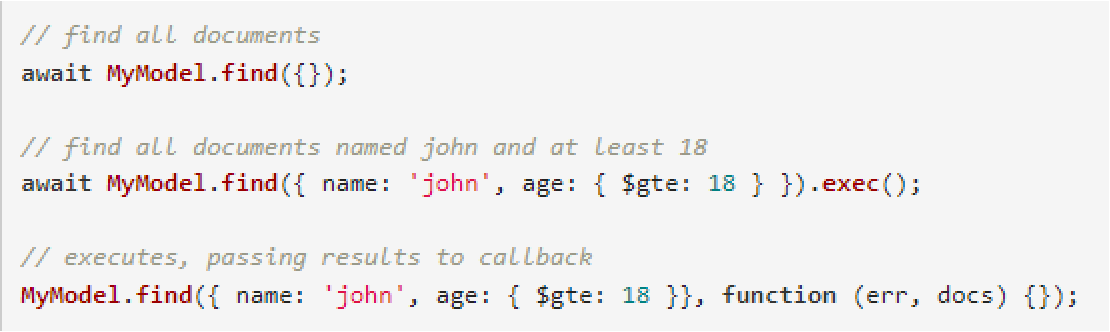
Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.



```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/test');

const Cat = mongoose.model('Cat', { name: String });

const kitty = new Cat({ name: 'Zildjian' });
kitty.save().then(() => console.log('meow'));
```

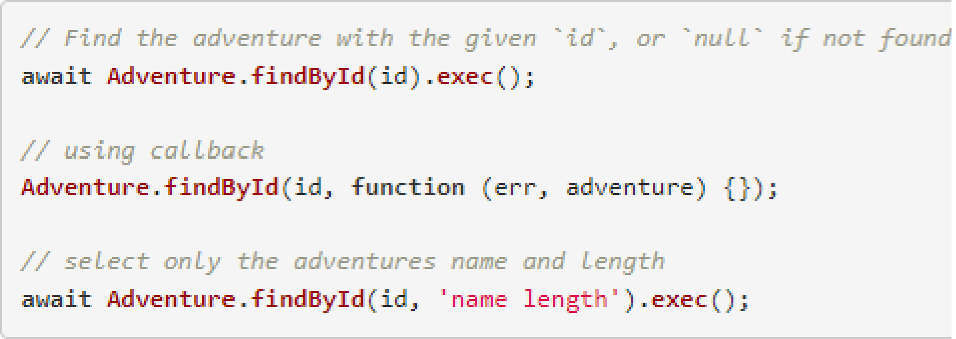
- Queries in mongoose:

- Find:



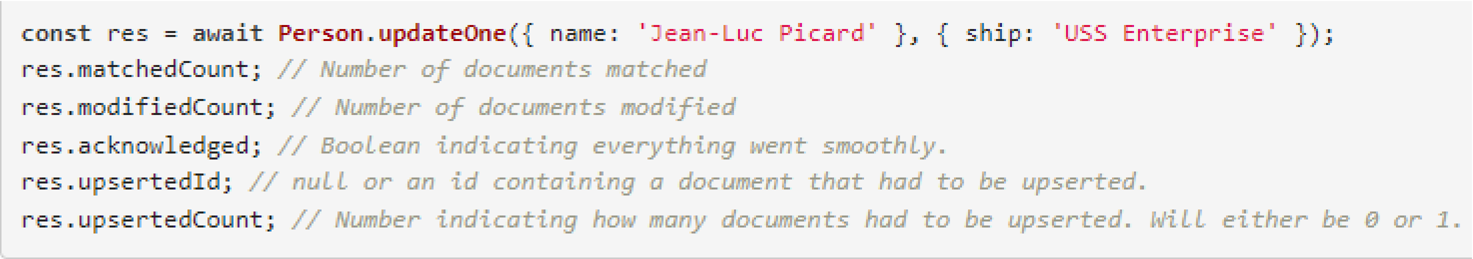
```
// find all documents
await MyModel.find({});

// find all documents named john and at least 18
await MyModel.find({ name: 'john', age: { $gte: 18 } }).exec();

// executes, passing results to callback
MyModel.find({ name: 'john', age: { $gte: 18 }}, function (err, docs) {});
```

- FindById:



```
// Find the adventure with the given `id`, or `null` if not found
await Adventure.findById(id).exec();

// using callback
Adventure.findById(id, function (err, adventure) {});

// select only the adventures name and length
await Adventure.findById(id, 'name length').exec();
```

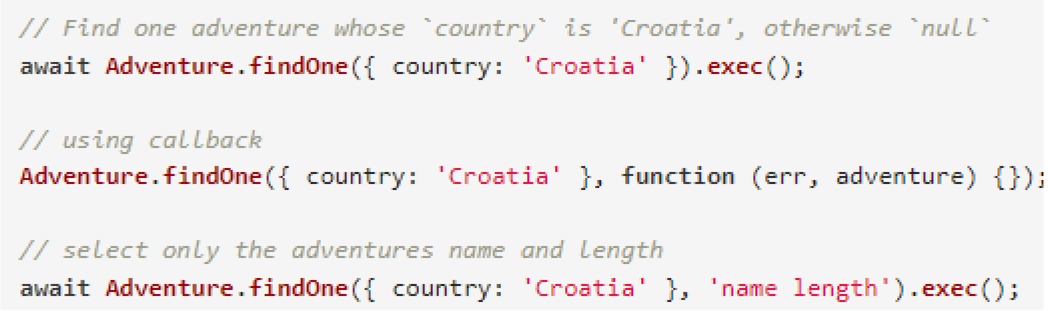
- UpdateOne:



```
const res = await Person.updateOne({ name: 'Jean-Luc Picard' }, { ship: 'USS Enterprise' });
res.matchedCount; // Number of documents matched
res.modifiedCount; // Number of documents modified
res.acknowledged; // Boolean indicating everything went smoothly.
res.upsertedId; // null or an id containing a document that had to be upserted.
res.upsertedCount; // Number indicating how many documents had to be upserted. Will either be 0 or 1.
```

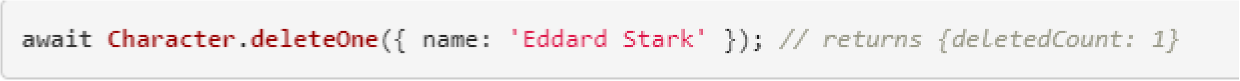
- Queries in mongoose:

- FindOne:

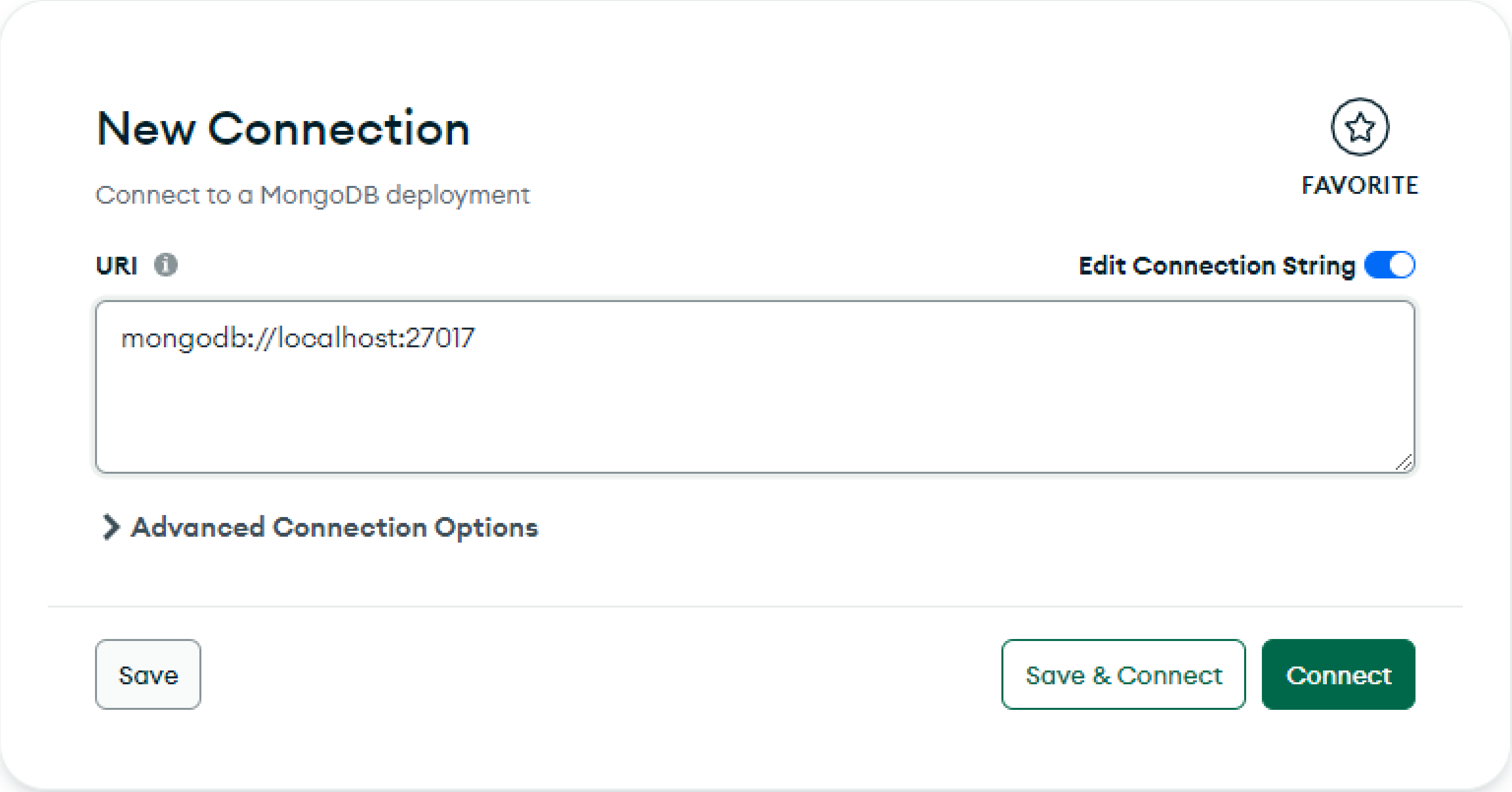


```
// Find one adventure whose `country` is 'Croatia', otherwise `null`
await Adventure.findOne({ country: 'Croatia' }).exec();

// using callback
Adventure.findOne({ country: 'Croatia' }, function (err, adventure) {});

// select only the adventures name and length
await Adventure.findOne({ country: 'Croatia' }, 'name length').exec();
```

- DeleteOne:



```
await Character.deleteOne({ name: 'Eddard Stark' }); // returns {deletedCount: 1}
```



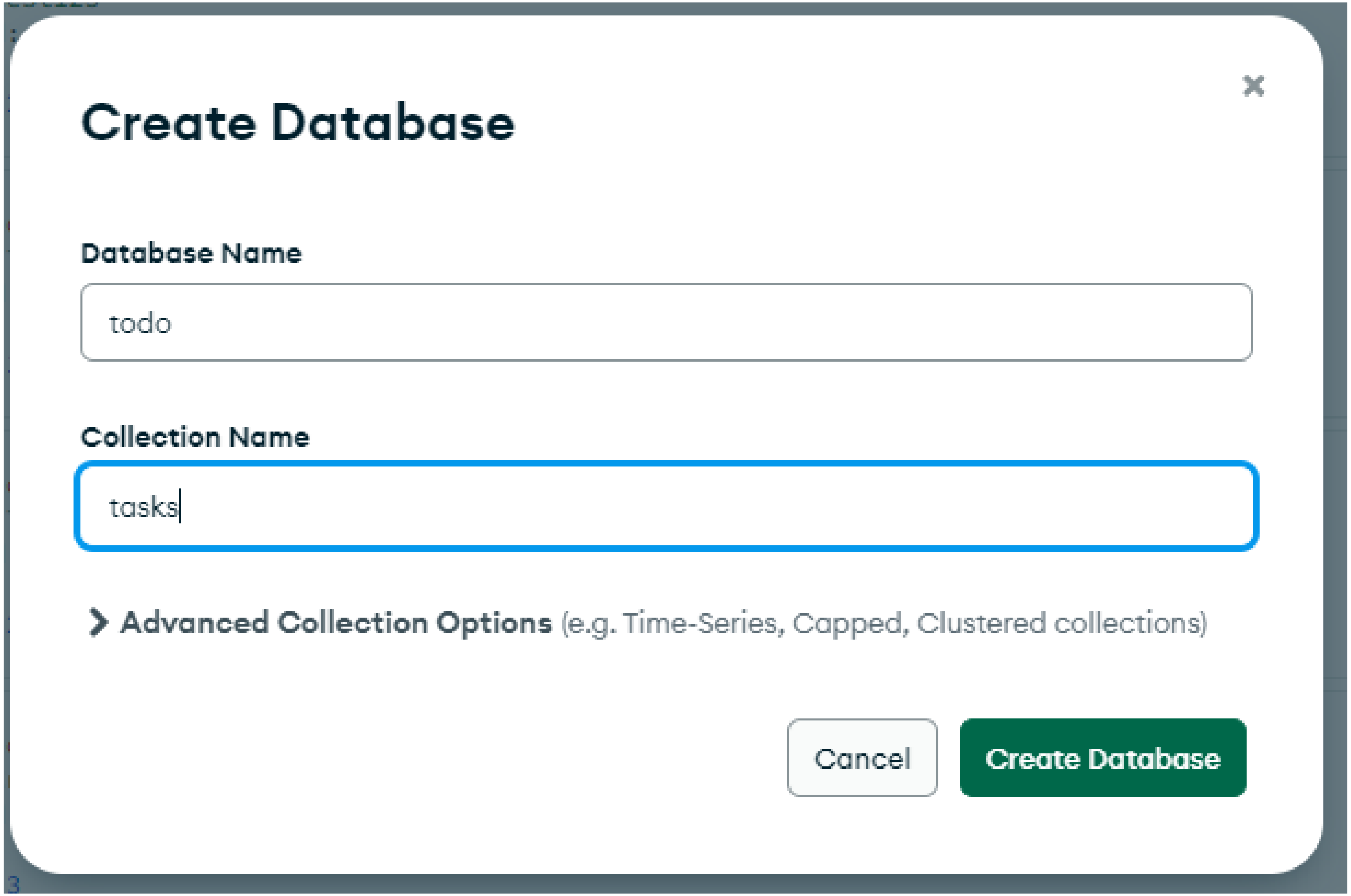
New Connection

Connect to a MongoDB deployment

URI

Edit Connection String

mongodb://localhost:27017



Create Database

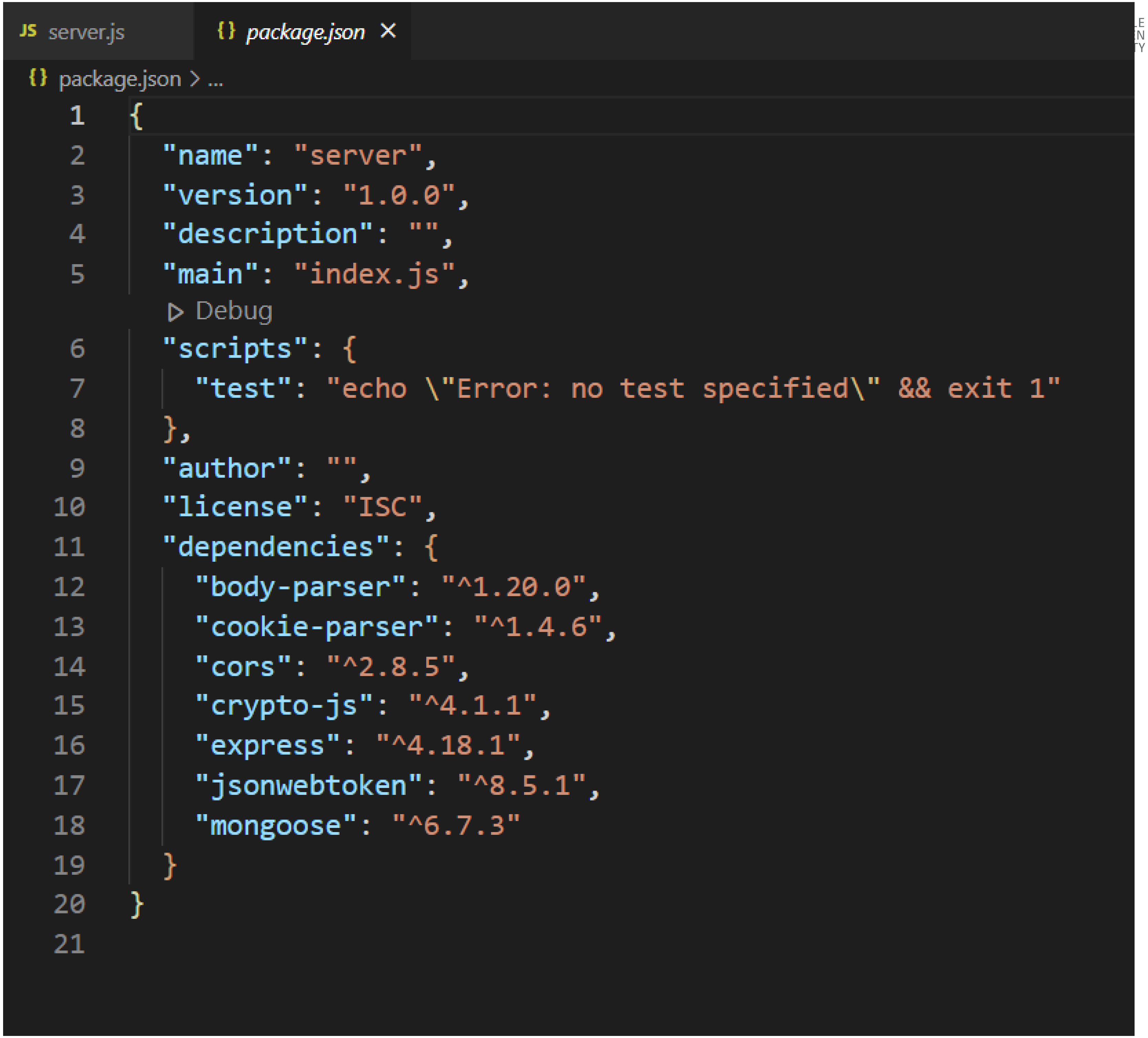
Database Name

todo

Collection Name

tasks

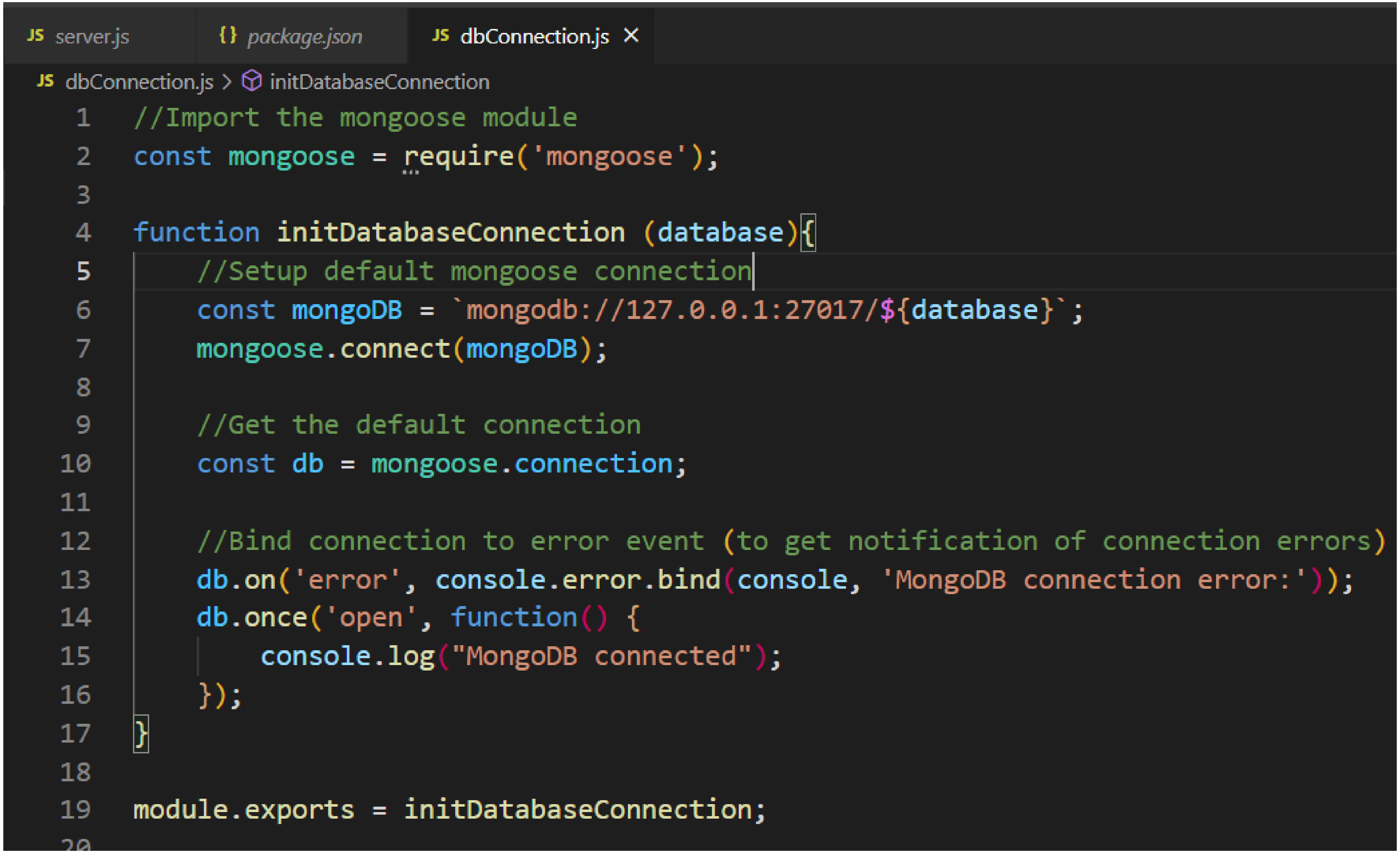
×
❯ Advanced Collection Options (e.g. Time-Series, Capped, Clustered collections)
Cancel
Create Database



```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.20.0",
    "cookie-parser": "^1.4.6",
    "cors": "^2.8.5",
    "crypto-js": "^4.1.1",
    "express": "^4.18.1",
    "jsonwebtoken": "^8.5.1",
    "mongoose": "^6.7.3"
  }
}
```

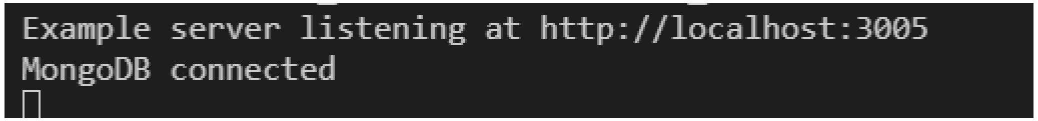
Create connection to the MongoDB Server from our Server:

- After installing mongoose for our project we need to create a connection to our mongoDB server.
- Within our server project, we create a new file: **dbConnection.js** within this file we write a function which will create a connection to our MongoDB Server and export this function
  1) First we import the **mongoose** package
  2) We create a mongoDB URI string
     ```
     const mongoDB = `mongodb://127.0.0.1:27017/${database}`;
     ```
  3) We use the mongoose object from the mongoose package and call connect function with the mongoDB URI string:
     ```
     mongoose.connect(mongoDB);
     ```
  4) In the last step we get the default database connection object and check if everything went right
     ```
     //Get the default connection
     const db = mongoose.connection;

     //Bind connection to error event (to get notification of connection errors)
     db.on('error', console.error.bind(console, 'MongoDB connection error:'));
     db.once('open', function() {
         console.log("MongoDB connected");
     });
     ```

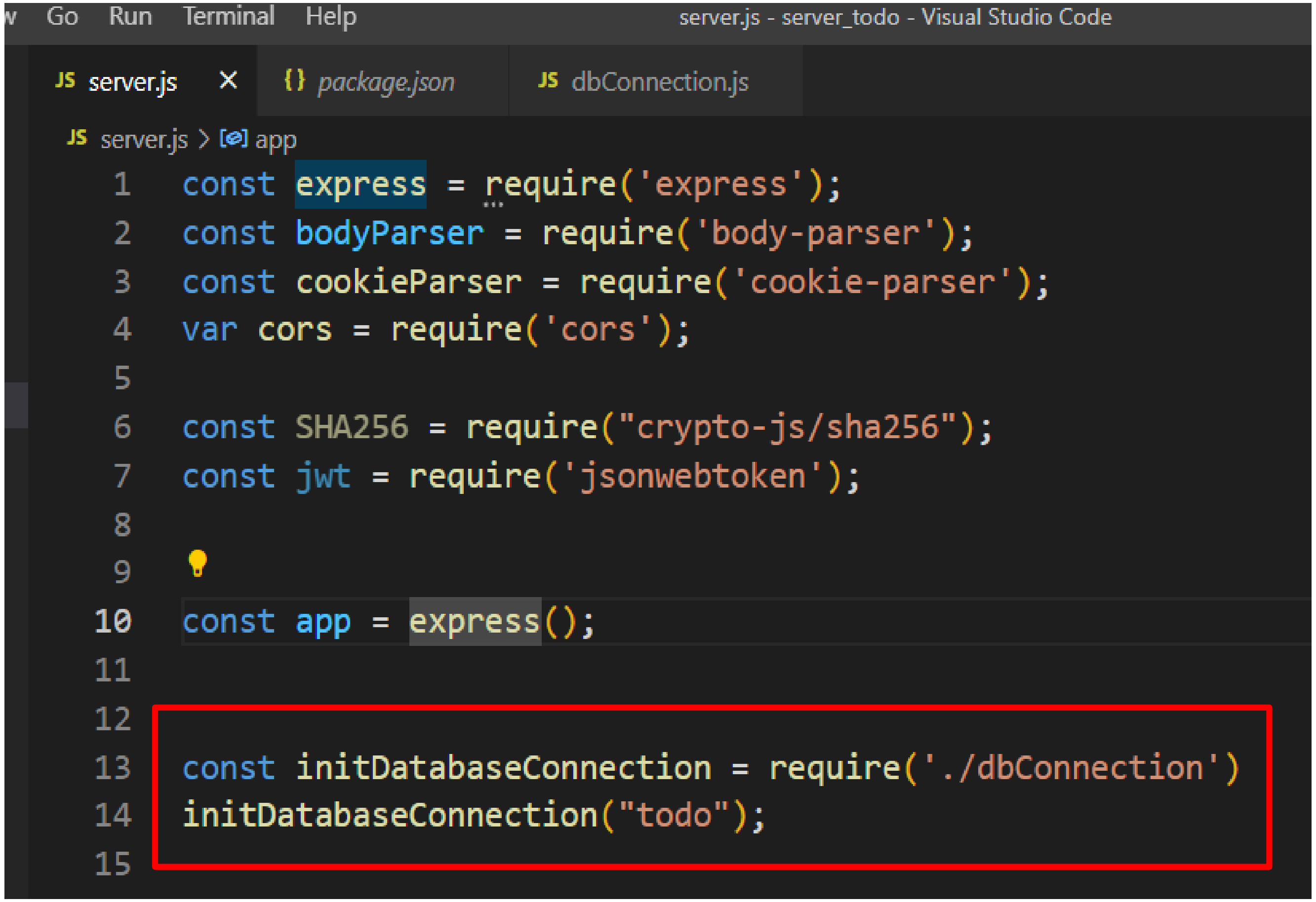


```javascript
//Import the mongoose module
const mongoose = require('mongoose');

function initDatabaseConnection (database){
    //Setup default mongoose connection
    const mongoDB = `mongodb://127.0.0.1:27017/${database}`;
    mongoose.connect(mongoDB);

    //Get the default connection
    const db = mongoose.connection;

    //Bind connection to error event (to get notification of connection errors)
    db.on('error', console.error.bind(console, 'MongoDB connection error:'));
    db.once('open', function() {
        console.log("MongoDB connected");
    });
}

module.exports = initDatabaseConnection;
```

- After we completed the dbConnection.js file we can import the **initDatabaseConnection** function from the dbConnection.js file.
- After we import the function we call it and as the database parameter we use **todo** which we create after installing MongoDB Compass
- If you start the server you should see something like this in the terminal:

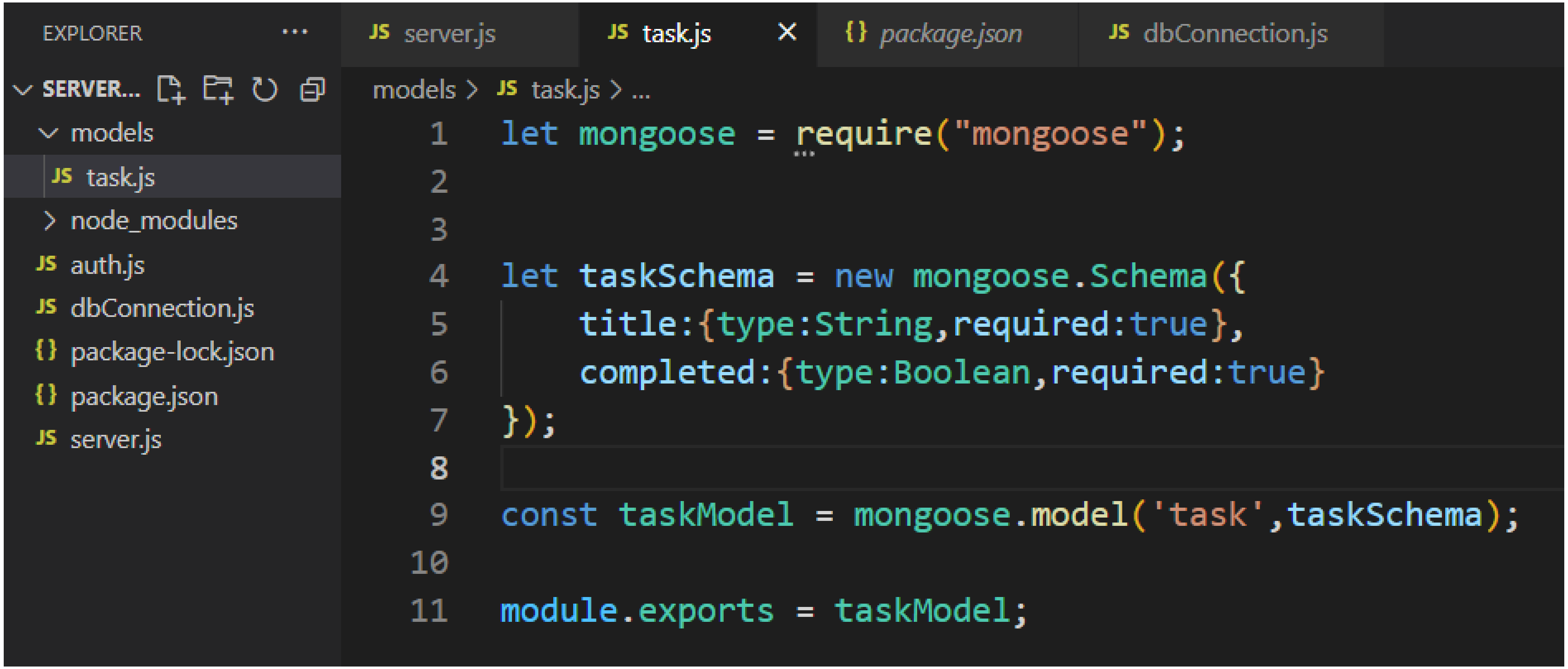


```
Example server listening at http://localhost:3005
MongoDB connected
```



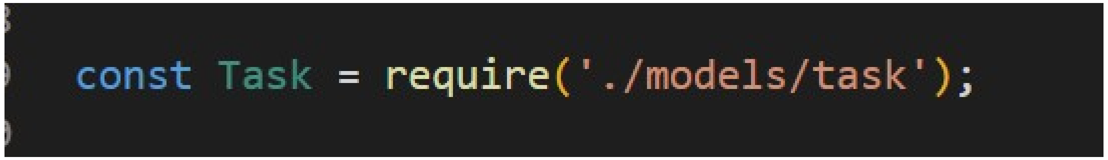
```javascript
const express = require('express');
const bodyParser = require('body-parser');
const cookieParser = require('cookie-parser');
var cors = require('cors');

const SHA256 = require("crypto-js/sha256");
const jwt = require('jsonwebtoken');


const app = express();


const initDatabaseConnection = require('./dbConnection')
initDatabaseConnection("todo");

```

- In our server project we create a new directory called **models**. Within this directory we define our mongoose schemas/models (MongoDB Collection).
- We create a file **task.js** and in this file we define the mongoose schema for our task. First wee need to import the mongoose package.
- The Schema in mongoose define how our document will look like in the MongoDB and what kind of types each property of our data has. We use the function **Schema** from the mongoose object to define our schema. Some of the permitted SchemaTypes are: **String, Number, Date, Boolean, etc.**
- For each property in our Schema we also define what type the values needs to be and if this value is required, if a new entry is made in the MongoDB collection. We do this through a JavaScript object.
- After we have defined our Schema we need to create model out of it. With the Model we are able to use MongoDB functions like searching for an entry, creating a new entry etc.. A model is class with which we construct documents and add them to our database. In the end we export this model.



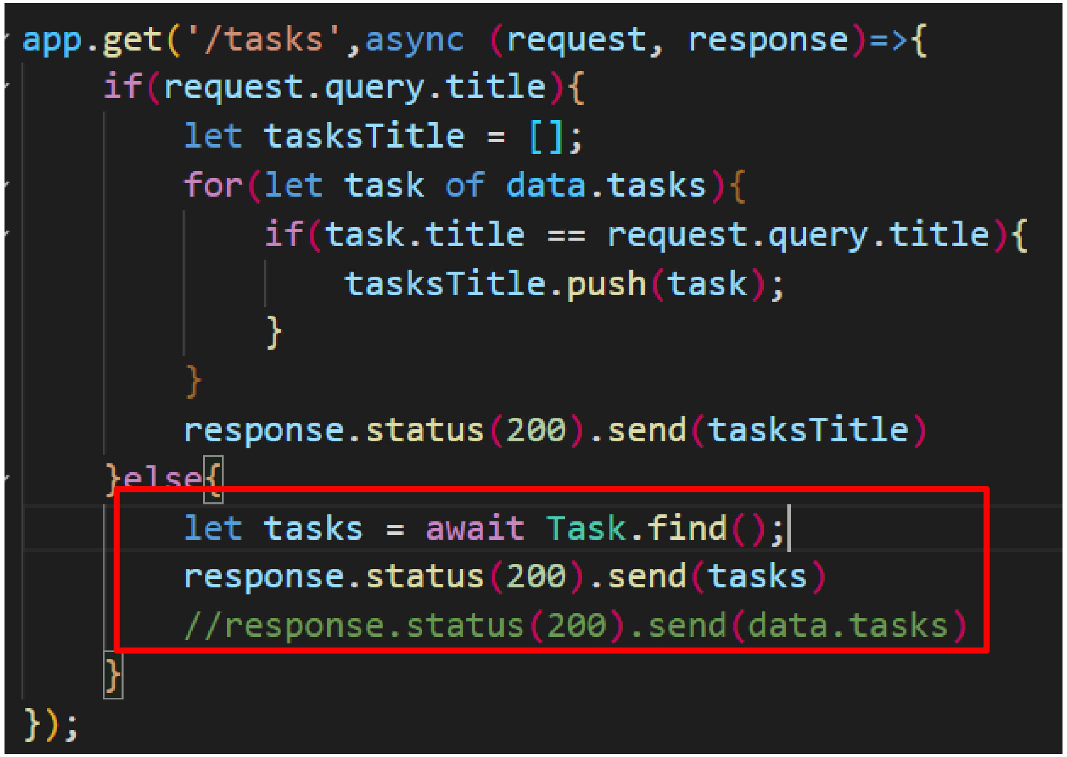
```
let mongoose = require("mongoose");


let taskSchema = new mongoose.Schema({
    title:{type:String,required:true},
    completed:{type:Boolean,required:true}
});

const taskModel = mongoose.model('task',taskSchema);

module.exports = taskModel;
```

- After we defined the Schema and created a model from it we can import the model in out server.js file and use it within our routes.



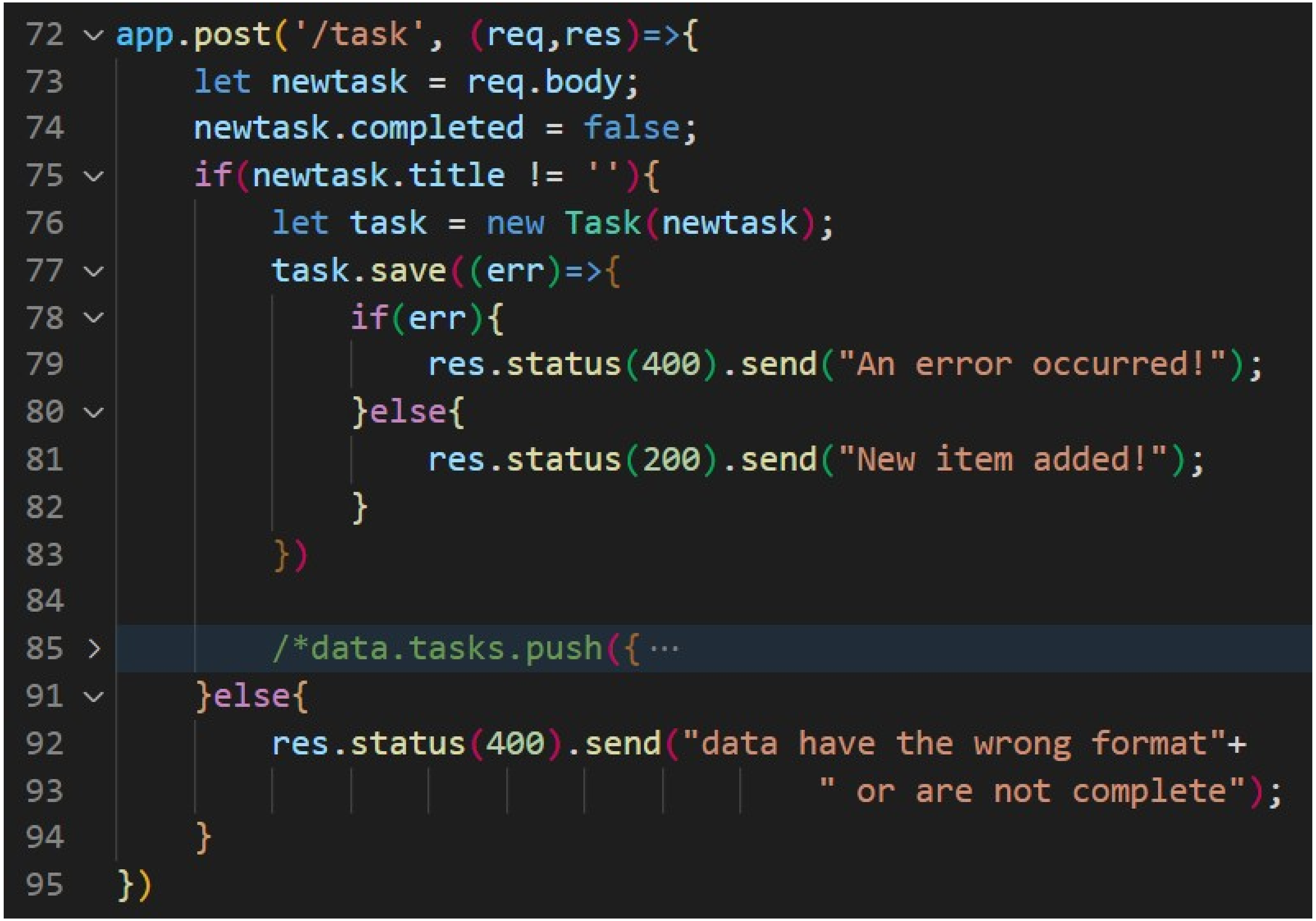
```
const Task = require('./models/task');
```

- The first route we change is the **get** route, which will deliver all tasks which do exist in our database. To get all task within our database we use the **find()** function from our Task object. Through our Task object we get access to all the mongoose functions, which will be translated to MongoDB functions.



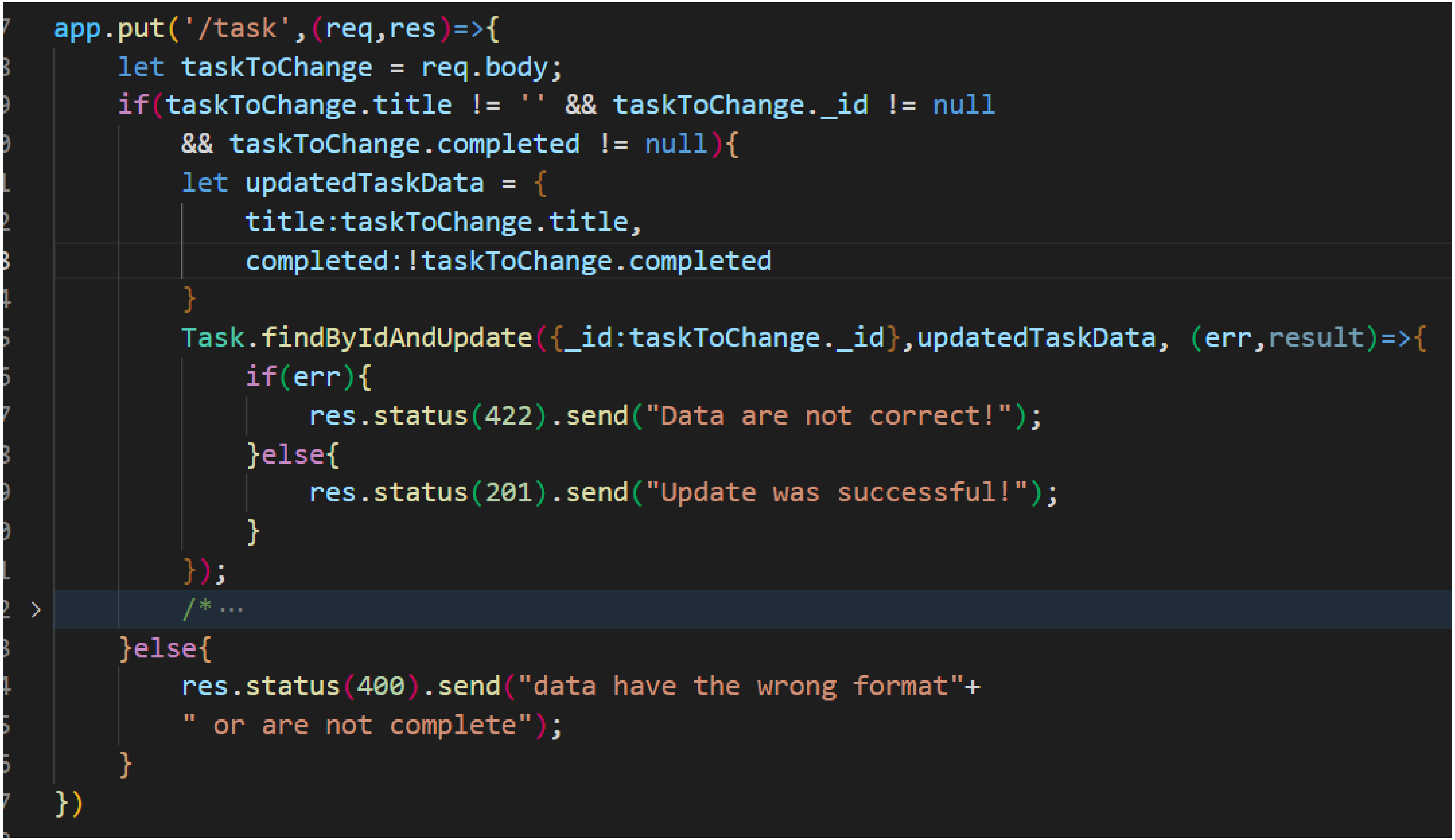
```
app.get('/tasks',async (request, response)=>{
    if(request.query.title){
        let tasksTitle = [];
        for(let task of data.tasks){
            if(task.title == request.query.title){
                tasksTitle.push(task);
            }
        }
        response.status(200).send(tasksTitle)
    }else{
        let tasks = await Task.find();
        response.status(200).send(tasks)
        //response.status(200).send(data.tasks)
    }
});
```

- In the next step we want to change the post route. With this route we want to add a new task to our database. We do this through out Task object. We simply create a new Task object and the parameter is our JSON object we got from our client. After we created a new object we execute the save() function. The save function needs a callback function as parameter. Within this callback function we check for errors. If there was no error we have successfully added a new task to the database.



```
app.post('/task', (req,res)=>{
    let newtask = req.body;
    newtask.completed = false;
    if(newtask.title != ''){
        let task = new Task(newtask);
        task.save((err)=>{
            if(err){
                res.status(400).send("An error occurred!");
            }else{
                res.status(200).send("New item added!");
            }
        })

        /*data.tasks.push({ ···
    }else{
        res.status(400).send("data have the wrong format"+
                                    " or are not complete");
    }
})
```

- To update our task we first create new object with the data we want to update. In our example we create a object and we want to update the title as well as the completed status. We then use the **findByIdAndUpdate** function from the mongoose package. The first parameter is an filter/search object (what are we searching within the database), the second parameter are the updated data that override the current data and last parameter is a callback function.

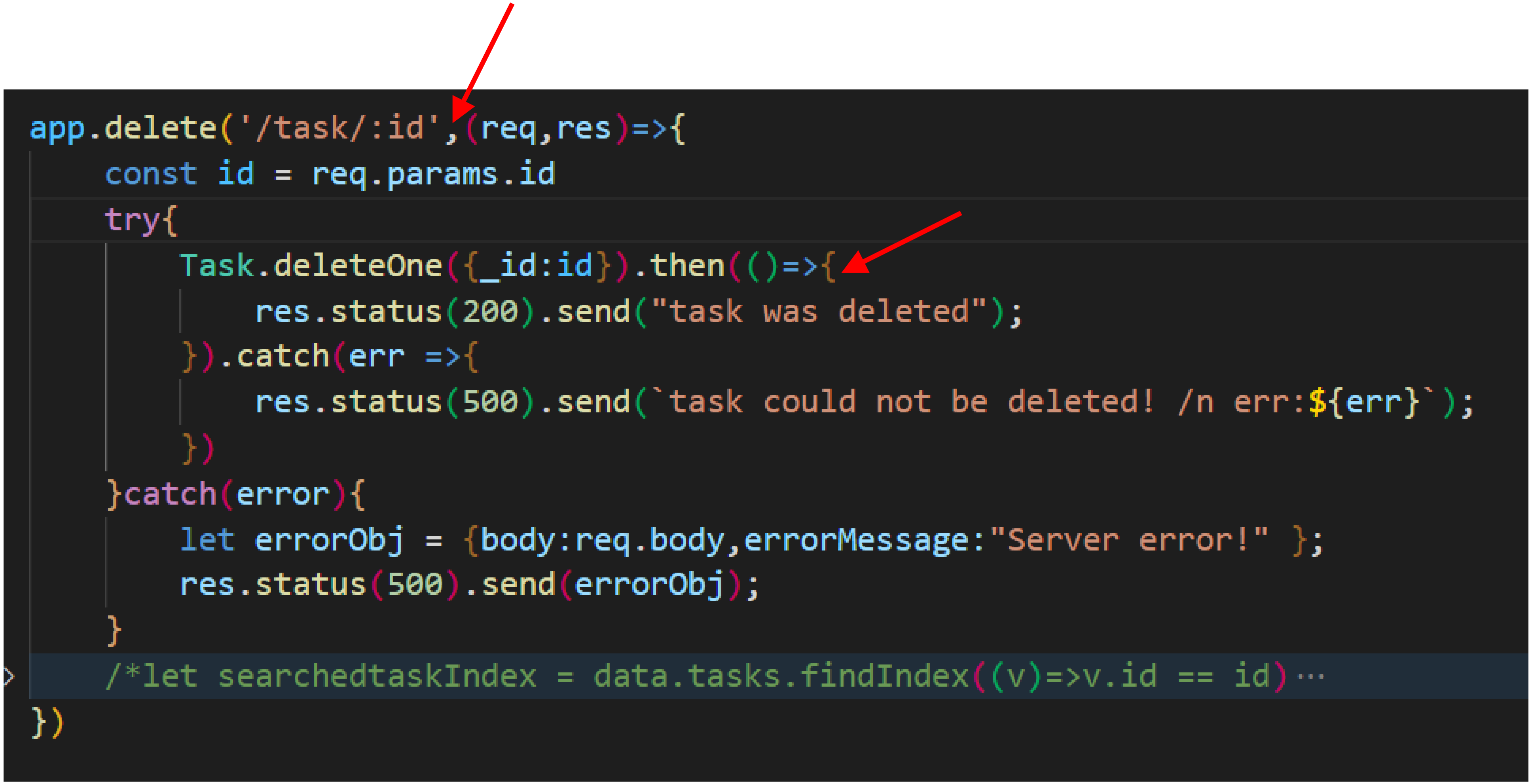


```
app.put('/task',(req,res)=>{
    let taskToChange = req.body;
    if(taskToChange.title != '' && taskToChange._id != null
        && taskToChange.completed != null){
        let updatedTaskData = {
            title:taskToChange.title,
            completed:!taskToChange.completed
        }
        Task.findByIdAndUpdate({_id:taskToChange._id},updatedTaskData, (err,result)=>{
            if(err){
                res.status(422).send("Data are not correct!");
            }else{
                res.status(201).send("Update was successful!");
            }
        });
        /*…
    }else{
        res.status(400).send("data have the wrong format"+
        " or are not complete");
    }
})
```

- In the end we also want to delete tasks from the database. We use the **deleteOne** function from mongoose (from the Task object). The first parameter of the function is a filter/search object and we want filter after ids. Instead of a callback function we use here then and catch. If the task was successful deleted we send an appropriate response to the client.

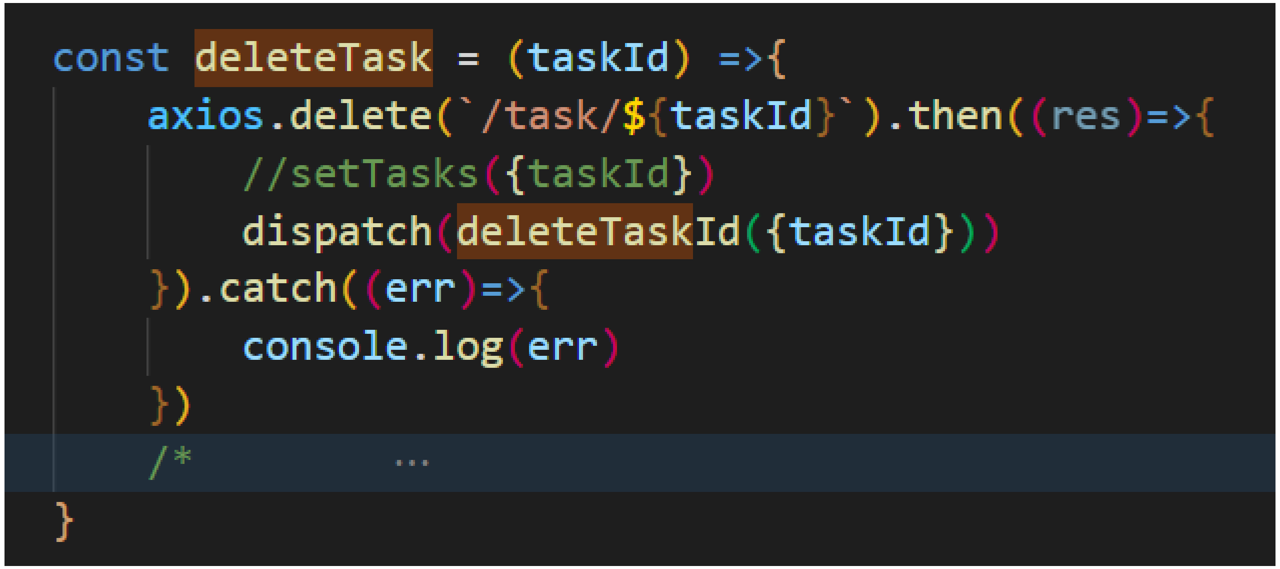
- We also need to change the route as we use query parameter with DELETE routes.

- In our React app we also need to do some changes as with mongoDB the **id** properties changes to **_id**



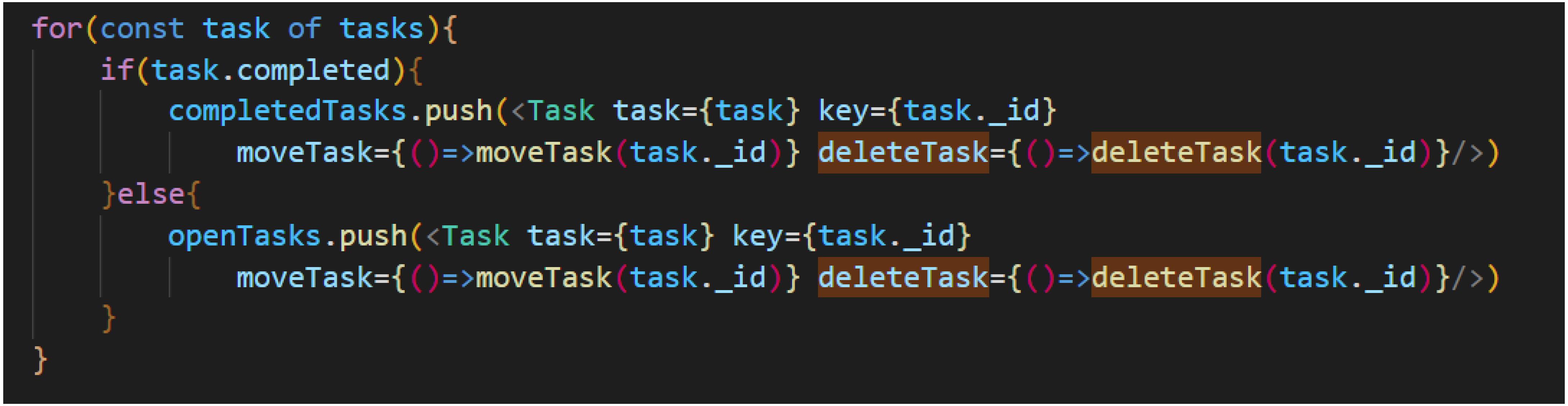
```
app.delete('/task/:id',(req,res)=>{
    const id = req.params.id
    try{
        Task.deleteOne({_id:id}).then(()=>{
            res.status(200).send("task was deleted");
        }).catch(err =>{
            res.status(500).send(`task could not be deleted! /n err:${err}`);
        })
    }catch(error){
        let errorObj = {body:req.body,errorMessage:"Server error!" };
        res.status(500).send(errorObj);
    }
    /*let searchedtaskIndex = data.tasks.findIndex((v)=>v.id == id)···
})
```

The new delete function in ShowToDo.js



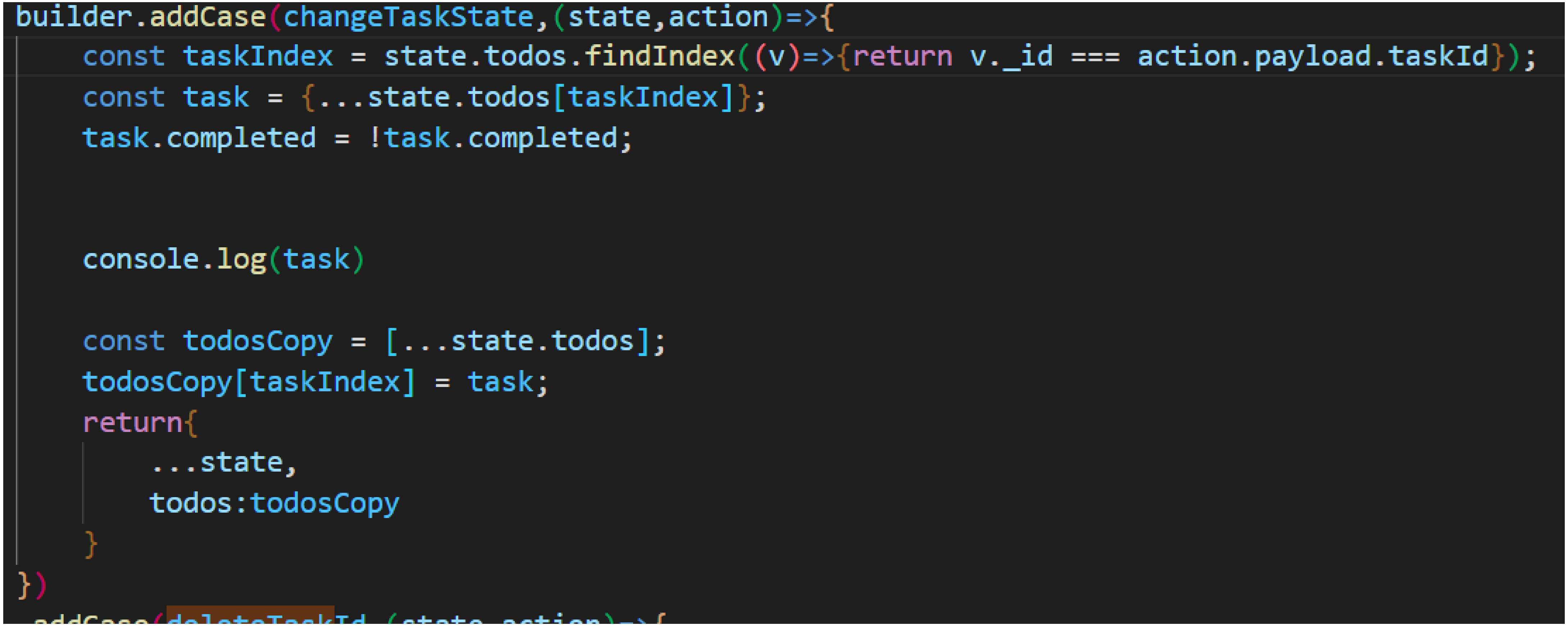
```
const deleteTask = (taskId) =>{
    axios.delete(`/task/${taskId}`).then((res)=>{
        //setTasks({taskId})
        dispatch(deleteTaskId({taskId}))
    }).catch((err)=>{
        console.log(err)
    })
    /*          ...
}
```

The new for loop in function in ShowToDo.js with **_id** instead **id**



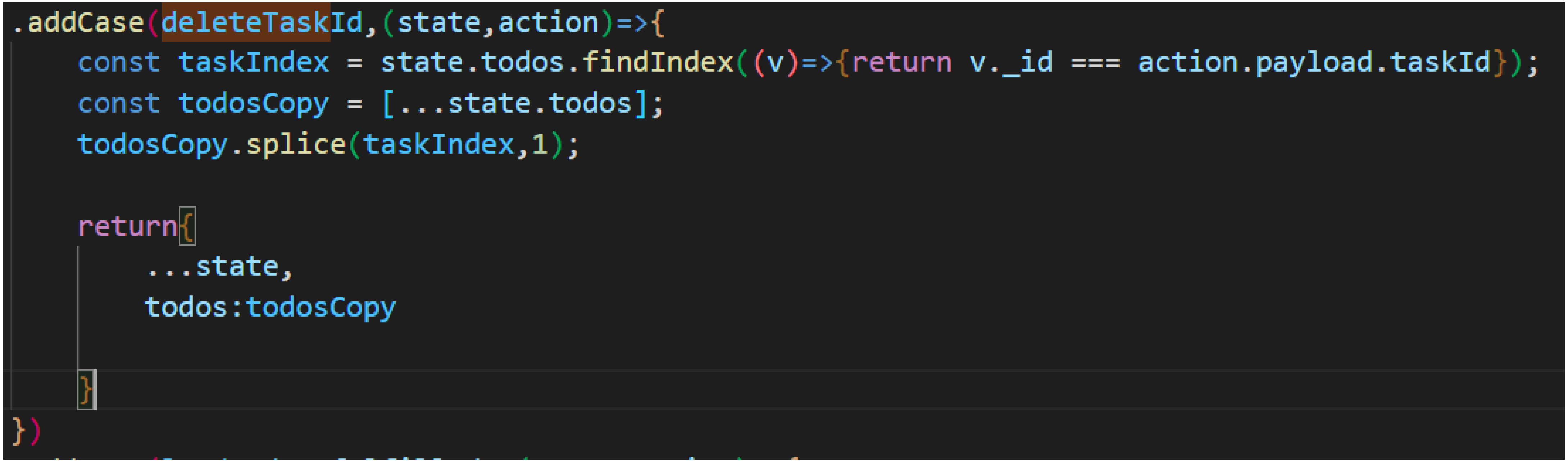
```
for(const task of tasks){
    if(task.completed){
        completedTasks.push(<Task task={task} key={task._id}
            moveTask={()=>moveTask(task._id)} deleteTask={()=>deleteTask(task._id)}/>)
    }else{
        openTasks.push(<Task task={task} key={task._id}
            moveTask={()=>moveTask(task._id)} deleteTask={()=>deleteTask(task._id)}/>)
    }
}
```

New changeTaskState in reducer.js here we changed **id** to **_id**



```
builder.addCase(changeTaskState,(state,action)=>{
    const taskIndex = state.todos.findIndex((v)=>{return v._id === action.payload.taskId});
    const task = {...state.todos[taskIndex]};
    task.completed = !task.completed;


    console.log(task)

    const todosCopy = [...state.todos];
    todosCopy[taskIndex] = task;
    return{
        ...state,
        todos:todosCopy
    }
})
.addCase(deleteTaskId,(state,action)=>{
```

New deleteTaskId in reducer.js here we changed **id** to **_id**



```
.addCase(deleteTaskId,(state,action)=>{
    const taskIndex = state.todos.findIndex((v)=>{return v._id === action.payload.taskId});
    const todosCopy = [...state.todos];
    todosCopy.splice(taskIndex,1);

    return{
        ...state,
        todos:todosCopy

    }
})
```

- Exercise 8

  1) Change the different routes of server in a way, that they use the MongoDB server.

     You should follow the REST guide:

     1) GET: get all the tasks from the database
     2) POST: add a new task to the database
     3) PUT: update a task in the database
     4) DELETE: delete a task from the database

  2) IDs can be created from the mongoDB