

P. PORTO



POLITÉCNICO
DO PORTO
ESMAD

PROGRAMAÇÃO WEB II
TSIW

SUMMARY

- SQL versus NoSQL
- MongoDB
- ODM: Object Document Mapping
- Mongoose

HOW TO WRITE A CV

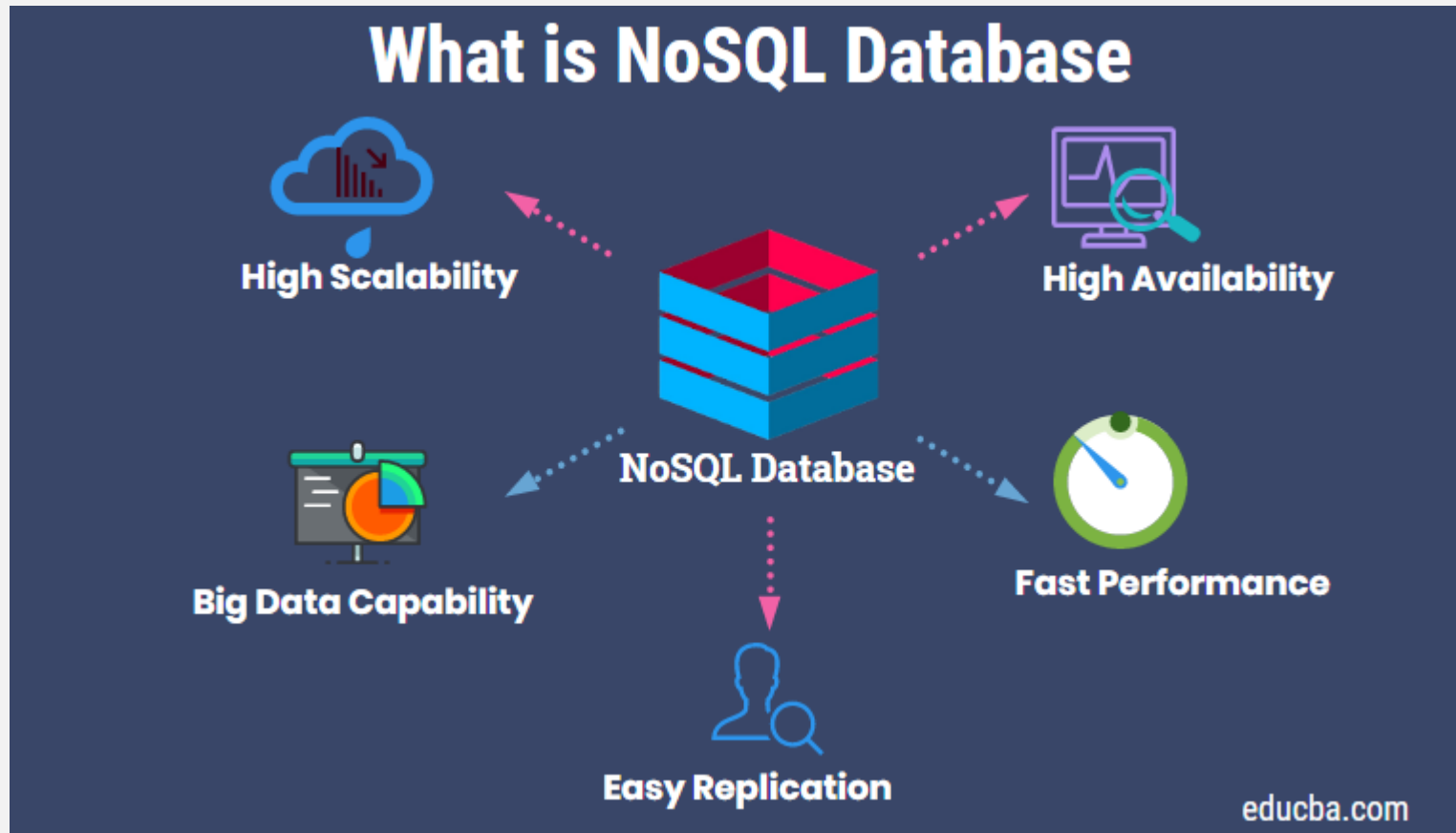


Leverage the NoSQL boom

SQL versus NoSQL

- Document-oriented databases have existed since the late 1960s but have only been recognized as NoSQL with increasing popularity at the beginning of the 21st century, triggered by the needs of Web 2.0 companies as Facebook, Google and Amazon
- NoSQL DBs, also called as **Non-relational** databases, are increasingly used to **store large volumes of data** and process web applications **in real time**
- NoSQL systems are also sometimes referred to as “**Not just SQL**” for emphasize that they can support SQL as query languages

SQL versus NoSQL

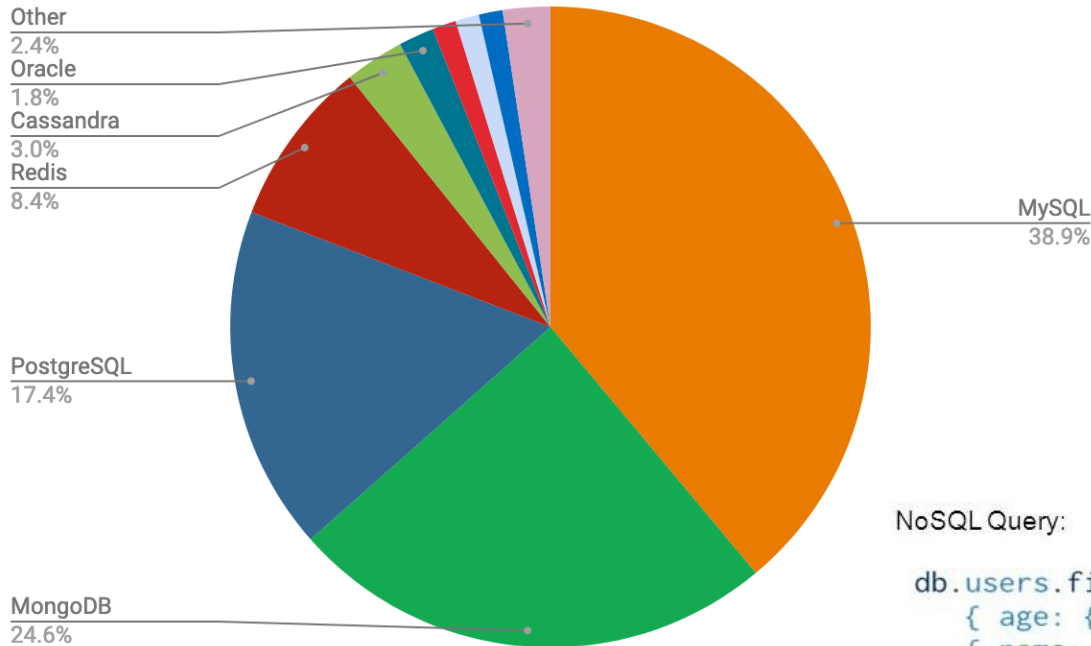


SOURCE: <https://www.educba.com/what-is-nosql-database/>

NoSQL databases

- Are self-describing, **do not require any schema**
- NoSQL DBs **do not enforce a relationship** between relations in all cases
- NoSQL documents are **non-structured documents**, which are complete entities that a user can readily read and understand the document
- NoSQL refers to **high-performance**, non-relational databases that utilize a wide variety of data models in the industry
- These databases are highly recognized for their scalable performance, ease-of-use, strong resilience, and wide availability for the users

SQL versus NoSQL



NoSQL Query:

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
) .limit(5)
```

- ← collection
- ← query criteria
- ← projection
- ← cursor modifier

SQL Query:

```
SELECT _id, name, address  
FROM users  
WHERE age > 18  
LIMIT 5
```

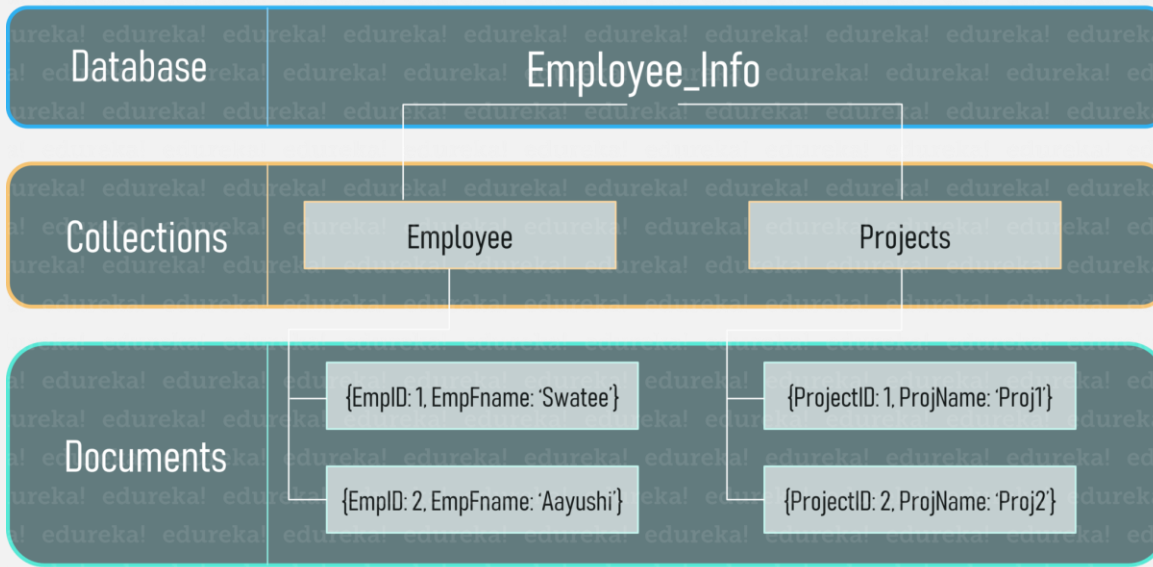
- ← projection
- ← table
- ← select criteria
- ← cursor modifier

MongoDB

- Open-source database and with a **data model oriented to documents**
- Fits in the NoSQL database category which means it does not follow the fixed structure of SQL DBs
- The way of working is different from SQL because it uses the OO **(Object-Oriented) paradigm**
- It does not support joins but can represent hierarchical data structures
- There are **no schemas** which means that each collection can contain different types of objects
- Easily scalable and high performance



MongoDB



```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
    phone: "123-456-7890",
    email: "xyz@example.com"
  },
  access: {
    level: 5,
    group: "dev"
  }
}
```

Embedded sub-document

Embedded sub-document

user document

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

contact document

```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone: "123-456-7890",
  email: "xyz@example.com"
}
```

access document

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```


MySQL vs MongoDB

MySQL	MongoDB
Database	Database
Tables	Collections
Tuple/Rows	Documents (JSON)
Columns	Fields
SELECT	FIND
INSERT	INSERT
UPDATE	UPDATE
DELETE	REMOVE

```
db.users.insert (
  {
    name: "sue",
    age: 26,
    status: "A"
  }
)
```

← collection

← field: value

← field: value

← field: value

} document

To insert data into an employee table

```
INSERT INTO employees (employee_id,
empage)
VALUES ('abc001', '23')
```

To insert data into an employee document

```
db.employees.insert({
  employee_id: 'abc001',
  age: 23,
})
```

edureka!

LINKS:

<https://www.mongodb.com/> (official website)

<https://docs.mongodb.com/manual/crud/> (docs)

<https://account.mongodb.com/account/login> (Atlas Online Archive)

<https://www.tutorialspoint.com/mongodb/index.htm> (tutorial)

ODM - Object Document Mapping

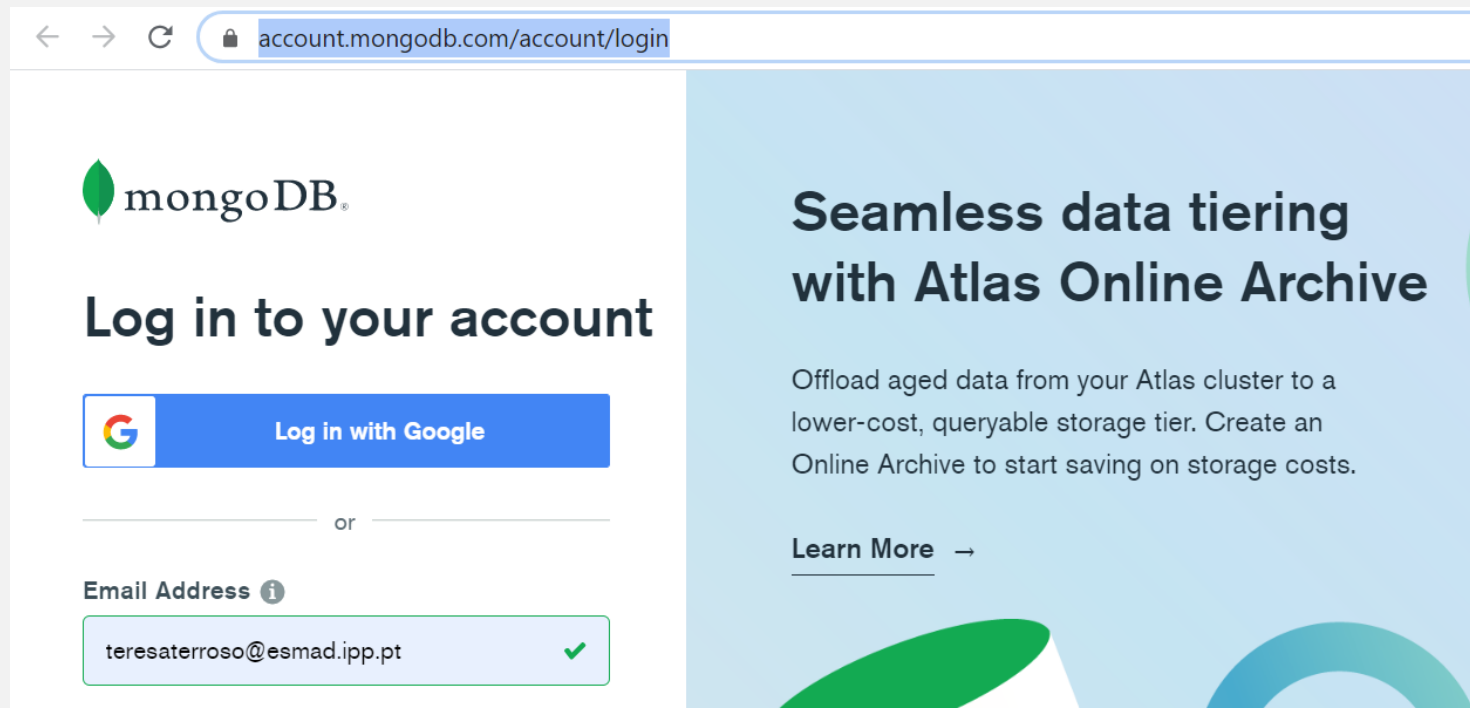
- **ODM** is the ORM for databases oriented towards non-relational documents, such as MongoDB, CouchDB, ...
- ODM provides **persistence services** for these data stores
- ODM allows to:
 - store new documents, updating and removing existing ones
 - track what has been modified, execute and save changes
 - have collections and 1:M and N:M associations
 - have removal of objects and cascade of persistence
 - ...

Mongo ODMs

- Examples of ODMs that can be used in Node.js:
 - [Mongoose](#)
 - [Mongorito](#)
 - Doctrine
 - Mongolian
 - MongoJS
 - ...

Get started with Atlas

- MongoDB Atlas provides an easy way to host and manage your data in the cloud
- Create a free MongoDB Atlas cluster: follow this [tutorial](#)
- Atlas login page: <https://account.mongodb.com/account/login>



Get started with Atlas

- Main steps:
 1. Deploy a Free Tier Cluster
 2. Add Your Connection IP Address to IP Access List
 3. Create a Database User for your Cluster
 4. Connect your Application to your Cluster: copy the provided connection string

The screenshot shows the MongoDB Atlas 'Connect' wizard. It has three steps: 'Setup connection security' (completed), 'Choose a connection method' (current step), and 'Connect'. Step 1, 'Select your driver and version', shows 'Node.js' selected for the driver and '3.6 or later' for the version. Step 2, 'Add your connection string into your application code', includes a checkbox for 'Include full driver code example' and a text area containing the connection string: `mongodb+srv://dbTeresa:<password>@cluster0.mun28.mongodb.net/myFirstDatabase?retryWrites=true&w=majority`. A copy icon is to the right of the text area. Below the text area, instructions state: 'Replace <password> with the password for the dbTeresa user. Replace myFirstDatabase with the name of the database that connections will use by default. Ensure any option params are URL encoded.'

```
const mongoConnectionString =  
`mongodb+srv://${USER}:${PASSWORD}@${CLUSTER}.mongodb.net/${DB}?  
retryWrites=true&w=majority`
```

Mongoose in Node.js

- [Mongoose](#): “elegant mongoDB object modeling for node.js”
- Provides a straight-forward, schema-based solution to model the application data
 - Facilitates query building
 - Built-in type casting and validation
 - Allows to pre-define events, for example to perform an operation before saving a document
 - Allows to represent the logical rules of the business
- How to install:
`npm install mongoose --save`

Mongoose in Node.js

- Everything in Mongoose starts with a [Schema](#)
- Each schema maps to a MongoDB **collection** and defines the shape of the documents within that collection

```
import mongoose from 'mongoose';
const { Schema } = mongoose;

const blogSchema = new Schema({
  title: { type: String, required: true }, // mandatory String key
  author: String, // shorthand for {type: String}
  comments: [{ body: String, date: Date }], // array of documents
  date: { type: Date, default: Date.now }, // define a default value
  id: { type: Number, unique: true }, // define an unique field
  meta: {
    votes: Number,
    favs: Number
  }
});
```

- [Schema types](#): handle definition of path defaults, validation, getters, setters, field selection defaults for queries, and other general characteristics for Mongoose document properties

Mongoose in Node.js

- Creating a model: to start creating documents based on a schema, it is required to compile the model, based on the defined schema, using `mongoose.model(modelName, schema)`

```
import mongoose from 'mongoose';
const { Schema } = mongoose;

const blogSchema = new Schema({
  ...
});
// creates a new model Blog using the defined schema above
const Blog = mongoose.model('Blog', blogSchema);

const newBlog = new Blog(...); // creates a Document (instance of a Model)
newBlog._id instanceof mongoose.Types.ObjectId; // true: by default,
Mongoose creates a new _id field of type ObjectId to the document
```

- Schemas have a few configurable [options](#) which can be passed to the constructor or to the set method: e.g., tell mongoose to ignore `createdAt` and `updatedAt` fields to the schema

```
const blogSchema = new Schema({ ... }, { timestamps: false });
```


Mongoose in Node.js

- Mongoose will **cast documents** to match the given schema types; this means you can safely pass untrusted data to Mongoose and trust that the data will match your schema

```
const userSchema = new mongoose.Schema({
  name: String,
  age: Number
});

const UserModel = mongoose.model('User', userSchema);

const doc = new UserModel({
  name: 'Jean-Luc Picard',
  age: '59', // Mongoose will convert this to a number
  rank: 'Captain'
});
doc.age; // 59
doc.rank; // undefined, Mongoose strips out `rank` because it isn't in the schema

await doc.save(); // saves document into Database

// Mongoose will convert '60' from a string to a number, even in on an update
await UserModel.updateOne({}, { $set: { age: '60' } });
```

Mongoose in Node.js

- In addition to casting values, Mongoose also lets you define validation in your schemas
 - Validations are defined in the SchemaType
 - Validation is middleware: Mongoose registers validation as a `pre('save')` hook on every schema by default
 - Validators are not run on **undefined** values (the only exception is the **required** validator)
 - Sub-documents of a document are also validated
 - Mongoose has several built-in validators but it is also customizable
 - Validation on update is **off** by default – one need to specify the `runValidators` option

Read more [here](#)

Mongoose in Node.js

- Examples using built-in validators

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  age: Number
});
const UserModel = mongoose.model('User', userSchema);


const doc = new UserModel({ age: 30 }); // create a user without the required field 'name'

const error = await doc.save().catch(err => err); // saves document into Database
err.message; // "Path `name` is required"
```

Mongoose in Node.js

- Example using custom validator

```
const userSchema = new Schema({
  phone: {
    type: String,
    validate: {
      validator: function(v) {
        return /\d{2}-\d{7}/.test(v);
      },
      message: props => `${props.value} is not a valid phone number!`
    },
    required: [true, 'User phone number required']
  }
});
```



Custom validation is declared by passing
a **validation function**
and its **error message**

```
const User = mongoose.model('user', userSchema);
```

```
const user = new User();
let error = user.validateSync(); // manually run validation
assert.equal(error.errors['phone'].message, 'User phone number required');
```

```
user.phone = '22.012.3456'; // bad phone number: should be 22-0123456
error = user.validateSync(); // run validation again
assert.equal(error.errors['phone'].message, '555.0123 is not a valid phone number!');
```

Mongoose in Node.js

- Examples using built-in validators

```
const breakfastSchema = new Schema({
  eggs: {
    type: Number, min: [6, 'Must be at least 6, got {VALUE}'], max: 12
  },
  drink: {
    type: String,
    enum: {
      values: ['Coffee', 'Tea'], message: '{VALUE} is not supported'
    }
  },
  bacon: {
    type: Number,
    required: [true, 'Why no bacon?']
  }
});
const Breakfast = db.model('Breakfast', breakfastSchema);

const badBreakfast = new Breakfast({ eggs: 2, bacon: 0, drink: 'Milk' });
let error = badBreakfast.validateSync(); // manually run validation

assert.equal(error.errors['eggs'].message, 'Must be at least 6, got 2');
assert.equal(error.errors['drink'].message, '`Milk` is not supported.');
```

Customization of the error message: {VALUE} will be replaced with the value being validated

Mongoose in Node.js

- Models are fancy constructors compiled from Schema definitions
 - When you call `mongoose.model()` on a schema, Mongoose compiles that model
- Models are responsible for creating and reading documents from the underlying MongoDB database
 - Mongoose automatically looks for the **plural, lowercased** version of the model name
- An instance of a Model is a **Document**
 - Creating them and saving to the database is easy
 - Nothing will be created/removed until the connection your model uses is open

Mongoose in Node.js

- Connections: use `mongoose.connect()` method

```
const mongoose = require("mongoose");
(async () => {
  try {
    await mongoose.connect('mongodb://username:password@host:port/database?options...',
      {
        useNewUrlParser: true, // to remove deprecation warnings
        useUnifiedTopology: true
      }
    );
    console.log("Connected to the database!");
  } catch (error) {
    console.log("Cannot connect to the database!", err);
  }
})();
```

Insert here your Atlas connection string

Mongoose in Node.js

- Saving documents

```
// define a schema
const schema = new mongoose.Schema({ name: 'string', size: 'string' });

// define a model Tank for the above schema
const Tank = mongoose.model('tank', schema); // the model Tank is for the tanks collection in the DB

// creates a Document (instance of a Model)
const small = new Tank({ size: 'small' });
let smallTank = small.save(); // saves document it into the DB
// on success, the document as an _id unique field of type ObjectId (acts as primary key)
console.log(smallTank._id);

// or, for inserting large batches of documents
Tank.insertMany([{ size: 'small' }, { size: 'large' }], function(err) {
  // if one document has a validation error, no documents will be saved
  if (err) return handleError(err);
});
```


Mongoose in Node.js

- Mongoose models provide several static helper functions for CRUD operations
 - `Model.deleteMany()`
 - `Model.deleteOne()`
 - `Model.find()`
 - `Model.findById()`
 - `Model.findByIdAndRemove()`
 - `Model.findByIdAndUpdate()`
 - `Model.findOne()`
 - `Model.findOneAndRemove()`
 - `Model.findOneAndReplace()`
 - `Model.findOneAndUpdate()`
 - `Model.updateMany()`
 - `Model.updateOne()`
 - ...

Mongoose in Node.js

- The first parameter is the JSON query as in the [MongoDB shell](#)
 - The query is specified as a JSON document
- Each of those query functions returns a mongoose [Query](#) object
- The successful return of a query execution depends on the operation:
for `findOne()` it is a potentially-null single document, `find()` a list of documents, `update()` the number of documents affected, etc.
 - The API documentation for [models](#) provide more detail on what is passed
- If an error occurs the error parameter of the callback will contain an error document, and result will be null

Mongoose in Node.js

```
const Person = mongoose.model('Person', someSchema);

// find a person with last name matching 'Ghost', selecting the `lastname` and `occupation` fields
const query = Person.findOne({ 'lastname': 'Ghost' }, 'lastname occupation' , function (err, person) {
  if (person != null)
    console.log(person.lastname, person.occupation);
});
```

```
const Person = mongoose.model('Person', someSchema);

// find a person with last name matching 'Ghost'
const query = Person.findOne({
  'lastname': 'Ghost'
});

// selecting the `lastname` and `occupation` fields
query.select('lastname occupation');

// execute the query
query.exec(function (err, person) {
  if (person != null)
    console.log(person.lastname, person.occupation);
});
```

```
const Person = mongoose.model('Person',
someSchema);

// chaining syntax
let person = Person.findOne(
  { 'lastname': 'Ghost' })
  .select('lastname occupation')
  .exec();

if (person != null)
  console.log( person.lastname,
               person.occupation);
```

Mongoose in Node.js

- `find(filter, projection, options)`

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

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

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

// executes, name LIKE john and only selecting the "name" and "friends" fields
let name = 'john';
let docs = await MyModel.find({ name: new RegExp(name, 'i') }, 'name friends').exec();

// executes, name LIKE john and passing options: use skip and limit for pagination purposes
// skip: #items_per_page * #pages
// limit: #items_per_page
let docs = await MyModel.find({ name: /john/i }, null, { skip: 100, limit: 20 }).exec();
```

Mongoose in Node.js

- `findById(id, projection, options)`

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

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

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

Mongoose in Node.js

- `findByIdAndRemove(id, options)`
 - Finds a matching document, removes it, passing the found document (if any) to the callback

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

- `findByIdAndUpdate(id, update, options)`
 - Finds a matching document, updates it according to the update arguments, passing any options, and returns the found document (if any) to the callback

```
// Updates the adventure with the given `id` altering its name
await Adventure.findByIdAndUpdate(id, { name: 'john'},
  {
    returnOriginal: false, // to return the updated document
    runValidators: true,   //runs update validators on update command
    useFindAndModify: false //remove deprecation warning
  }
).exec();
```

Mongoose in Node.js

- For more complex queries, read documentation on [queries](#) for more details on how to use the [Query api](#)

```
// search for Tanks, of size small, created more than 1 year ago
Tank.find({ size: 'small' })
  .where('createdAt').gt(oneYearAgo)
  .exec();

// get all Tanks name data (excluding the field 'size')
Tank.find()
  .select('name -size')
  .exec();

// Deletes the first document that matches conditions from the collection
Tank.deleteOne({ size: 'large' }, function (err) {
  if (err) return handleError(err);
  // deleted at most one tank document
});

// update only the first document that matches filter
Tank.updateOne({ size: 'large' }, { name: 'T-90' }, function(err, res) {
  // Updated at most one doc, `res.nModified` contains the number of docs that MongoDB updated
});
```

Mongoose - relationships

- Compared to a traditional relational database (SQL), a document-oriented (NoSQL) database has poor or non-existent support for relations between objects (data schema)
- A NoSQL datastore persists and retrieves documents (often in JSON format) and any relationship between the documents is something the programmer must implement by itself
- Depending on the types of relationships, on data access patterns, or on data cohesion, the programmer must decide how to implement the data model, in other words, decide if it should be **denormalized** or **normalized** data

Mongoose - relationships

- **Reference Data Models** (Normalization): all the documents are kept 'separated'
 - For example, we have documents for **Tutorials** and **Comments**, and because they are all completely different documents, the Tutorial need a way to know which Comments it contains, by using IDs to make references on documents

```
// Tutorial
{
  _id: "609851dccda6e15c941eb27f",
  title: "Vue Tut #1",
  description: "Tut#1 Description",
  published: true,
  comments: [ "5db57a03faf1f8434098f7f8", "5db57a04faf1f8434098f7f9" ]
}
```

Child Referencing: the parent
references its children
Be aware, that it can grow a lot!

```
// Comment
{
  _id: "5db57a03faf1f8434098f7f8",
  author: "Teresa Azevedo",
  text: "Thank you, it helps me a lot."
}
```

Mongoose - relationships

- **Reference Data Models (Normalization):** all the documents are kept 'separated'
 - For example, we have documents for **Tutorials** and **Comments**, and because they are all completely different documents, the Tutorial need a way to know which Comments it contains, by using IDs to make references on documents

```
// Comment
{
  _id: "5db57a03faf1f8434098f7f8",
  author: "Teresa Azevedo",
  text: "Thank you, it helps me a lot.",
  tutorial_id: "609851dccda6e15c941eb27f"
}
```

```
// Tutorial
{
  _id: "609851dccda6e15c941eb27f",
  title: "Vue Tut #1",
  description: "Tut#1 Description",
  published: true
}
```

Parent Referencing: the child
references its parent

Mongoose - relationships

- **Embedded Data Models** (Denormalization): one can also have data in a denormalized form simply by embedding the related documents right into the main document
 - Using this form, all the relevant data of a sub-document is right inside the parent document without the need to separate documents, collections, and IDs

Sub-documents

```
// Tutorial document with Comments sub-documents
{
  _id: "609851dccda6e15c941eb27f",
  title: "Vue Tut #1",
  description: "Tut#1 Description",
  published: true
  comments: [
    {
      author: "Teresa Azevedo",
      text: "Thank you, it helps me a lot."
    }
    {
      author: "Teresa Terroso",
      text: "This is a great tutorial."
    }
  ]
}
```

Mongoose - relationships

- Types of Relationships
 - Usually when we have **one-to-few** relationship, **embed** the related documents into the parent documents
 - For a **one-to-many** relationship, you can either embed or reference
 - With **one-to-aLot** relationship, always use data **references**; that's because if you embed a lot of documents inside one document, they could quickly become too large
 - With **many-to-many** relationship, always use data **references** (you can use it on both sides of the relationship - **Two-way Referencing**)

Mongoose - relationships

- Data access patterns: consider how often data is read and written
 - If documents are mostly read and the data is not updated a lot, then you should probably embed the data (with embedding it requires only one trip to the database per query)
 - if data is updated a lot, then you should consider referencing (normalizing) it; that's because the database engine does more work to update an embed document than a standalone document

Mongoose - relationships

- Data cohesion: the last criterion is just a measure for how much the data is related
 - if two collections intrinsically belong together then they should probably be embedded into one another (example, one-to-one relationships)
 - If we frequently need to query both of collections on their own, we should normalize the data into two separate collections, even if they are closely related
 - Another way is still embedding documents (with appropriate fields) in parent document, but also create child collection

```
// Comments
{
  _id: "5db57a03faf1f8434098f7f8",
  author: "Teresa Azevedo",
  text: "Thank you, it helps me a lot."
}
{
  _id: "609850d6d2b8e35c801fcc34",
  author: "Teresa Terroso",
  text: "This is a great tutorial."
}
```

```
// Tutorial document with Comments sub-documents
{
  _id: "609851dccda6e15c941eb27f",
  title: "Vue Tut #1",
  description: "Tut#1 Description",
  published: true
  comments: [
    {
      _id: "5db57a03faf1f8434098f7f8",
      author: "Teresa Azevedo",
      text: "Thank you, it helps me a lot.",
    }
    {
      _id: "609850d6d2b8e35c801fcc34",
      author: "Teresa Terroso",
      text: "This is a great tutorial.",
    }
  ]
}
```

Mongoose - relationships

- How to define Mongoose models for **embedded documents** (without defining a second model for sub-documents)
 - Considerer that a tutorial has some images (15 or less), each one related to just that tutorial, and once these images are saved to the database they are not really updated anymore
 - No need to define a new model for images; just embed them into the tutorials

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

const Tutorial = mongoose.model(
  "Tutorial",
  new mongoose.Schema({
    title: String,
    author: String,
    images: []
  })
);

module.exports = Tutorial;
```

Tutorial model

Mongoose - relationships

By default, `findByIdAndUpdate()` returns the document as it was before update was applied.
With option `new: true`, it will return the object after its update

Operator `$push`: appends a value to an array
<https://docs.mongodb.com/manual/reference/operator/update/push/>

```
// Tutorial document with 1 Image
{
  _id: "609851dccda6e15c941eb27f",
  title: "Tutorial #1",
  author: "Teresa",
  images: [
    {
      url: "/images/mongodb.png",
      caption: "MongoDB Database"
    }
  ]
}
```

Created document in database

```
const mongoose = require("mongoose");
const db = require("./models");

const run = async function () {
  let tutorial = new Tutorial({
    title: "Tutorial #1",
    author: "Teresa"
  });
  await tutorial.save(); //save a new tutorial into DB
  let updatedTutorial = await Tutorial.findByIdAndUpdate(
    tutorial._id,
    {
      $push: {
        images: {
          url: "/images/mongodb.png",
          caption: "MongoDB Database"
        }
      }
    },
    { new: true, useFindAndModify: false });
  console.log(updatedTutorial);
};
run();
```

new model instance
(without ANY image)

Server App

IMPORTANT NOTES:

- 1) it misses DB connection and model definition
- 2) It is a simple server application, NOT a REST API

Mongoose - relationships

The same, but now using `save()`



```
const mongoose = require("mongoose");
const db = require("../models");

const run = async function () {
  let tutorial = new Tutorial({
    title: "Tutorial #1",
    author: "Teresa"
  });
  await tutorial.save();

  tutorial.images.push({
    url: "/images/mongodb.png",
    caption: "MongoDB Database"
  });

  await tutorial.save();
  console.log(tutorial);
};
run();
```

Server App

```
// Tutorial document with 1 Image
{
  _id: "609851dccda6e15c941eb27f",
  title: "Tutorial #1",
  author: "Teresa",
  images: [
    {
      url: "/images/mongodb.png",
      caption: "MongoDB Database"
    }
  ]
}
```

Created document in database

Mongoose - relationships

- How to define Mongoose models for **embedded documents** (but also defining a second model for sub-documents)
 - Considerer that a tutorial has some images, but also **want to query Images** on their own collections without necessarily querying for the Tutorials themselves

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

const Tutorial = mongoose.model(
  "Tutorial",
  new mongoose.Schema({
    title: String,
    author: String,
    images: []
  })
);

module.exports = Tutorial;
```

Tutorial model
(unaltered)

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

const Image = mongoose.model(
  "Image",
  new mongoose.Schema({
    url: String,
    caption: String
  })
);

module.exports = Image;
```

Image model

Mongoose - relationships

```
...
const run = async function () {
  let tutorial = new Tutorial({
    title: "Tutorial #1",
    author: "Teresa"
  });
  await tutorialDoc.save();

  let image = new Image({
    url: "/images/mongodb.png",
    caption: "MongoDB Database"
  });
  await imageDoc.save();

  let updatedTutorial = await Tutorial.findByIdAndUpdate(
    tutorial._id,
    {
      $push: {
        images: {
          _id: image._id,
          url: "/images/mongodb.png",
          caption: "MongoDB Database"
        }
      }
    },
    { new: true, useFindAndModify: false });
  console.log(updatedTutorial)
};
run();
```

Server app

```
// Image document
{
  _id: 5db6af68c90cdd3a2c3038ab,
  url: "/images/mongodb.png",
  caption: "MongoDB Database"
}

// Tutorial document with 1 Image
{
  _id: 609851dccda6e15c941eb27f,
  title: "Tutorial #1",
  author: "Teresa",
  images: [
    {
      _id: 5db6af68c90cdd3a2c3038ab,
      url: "/images/mongodb.png",
      caption: "MongoDB Database"
    }
  ]
}
```

Created documents in database

Changes: create an Image document before adding it into Tutorial, and then include its _id

Mongoose - relationships

- How to define Mongoose models for **referenced documents**
 - Considerer that a tutorial can have **many** comments (let's use Child Referencing, therefore the parent – Tutorial - references its children – Comments)

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

const Comment = mongoose.model("comment",
  new mongoose.Schema({
    author: String,
    text: String
  })
);

module.exports = Image;
```

Comment model

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const Tutorial = mongoose.model("tutorial",
  new mongoose.Schema({
    title: String,
    author: String,
    comments: [
      {
        type: Schema.Types.ObjectId,
        ref: 'comment'
      }
    ]
  })
);

module.exports = Tutorial;
```

Tutorial model

ref helps get full fields of Category with `populate()` method
<https://mongoosejs.com/docs/populate.html>

Mongoose - relationships

```
const run = async function () {
  let tutorial = new Tutorial({
    title: "Tutorial #1",
    author: "Teresa"
  });
  await tutorial.save();

  let comment = new Comment({
    author: "Teresa",
    text: "Great tutorial!"
  });
  await comment.save();

  await Tutorial.findByIdAndUpdate(
    tutorial._id,
    {
      $push: {
        comments: { _id: newComment._id }
      }
    },
    { new: true, useFindAndModify: false });

  let updatedTutorial = await Tutorial.findById(tutorial._id)
    .populate("comments", "-_id -__v");
  console.log(updatedTutorial);
};
run();
```

Server app

Created documents in database

```
// Comment document
{
  _id: 5db6af68c90cdd3a2c3038ab,
  author: "Teresa",
  text: "Great tutorial!"
}

// Tutorial document with 1 Comment
{
  _id: 609851dccda6e15c941eb27f,
  title: "Tutorial #1",
  author: "Teresa",
  comments: [
    _id: 5db6af68c90cdd3a2c3038ab
  ]
}
```

```
{
  _id: "609851dccda6e15c941eb27f",
  title: "Tutorial #1",
  author: "Teresa",
  comments: [
    {
      author: "Teresa",
      text: "Great tutorial!"
    }
  ]
}
```

Console output

Mongoose - relationships

Server app

```
const run = async function () {
  let tutorial = new Tutorial({
    title: "Tutorial #1",
    author: "Teresa"
  });
  await tutorial.save();

  let comment = new Comment({
    author: "Teresa",
    text: "Great tutorial!"
  });
  await comment.save();

  tutorial.comments.push(comment._id);
  await tutorial.save();

  let updatedTutorial = await Tutorial.findById(tutorial._id)
    .populate("comments", "-_id -__v");
  console.log(updatedTutorial);
};
run();
```

The same, but now using **save()**

Created documents in database

```
// Comment document
{
  _id: 5db6af68c90cdd3a2c3038ab,
  author: "Teresa",
  text: "Great tutorial!"
}

// Tutorial document with 1 Comment
{
  _id: 609851dccda6e15c941eb27f,
  title: "Tutorial #1",
  author: "Teresa",
  comments: [
    _id: 5db6af68c90cdd3a2c3038ab
  ]
}
```

```
{
  _id: "609851dccda6e15c941eb27f",
  title: "Tutorial #1",
  author: "Teresa",
  comments: [
    {
      author: "Teresa",
      text: "Great tutorial!"
    }
  ]
}
```

Console output

Mongoose - relationships

- How to define Mongoose models for **referenced documents**
 - Considerer that a category has a LOT of tutorials (let's use Parent Referencing, therefore the child – Tutorial - references its parent – Cathegory)

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

const Category = mongoose.model("category",
  new mongoose.Schema({
    name: String
  })
);

module.exports = Category;
```

Category model

```
const mongoose = require("mongoose");
const Schema = mongoose.Schema;

const Tutorial = mongoose.model("tutorial",
  new mongoose.Schema({
    title: String,
    author: String,
    category: {
      type: Schema.Types.ObjectId,
      ref: 'category'
    }
  })
);

module.exports = Tutorial;
```

Tutorial model

ref helps get full fields of Category with `populate()` method
<https://mongoosejs.com/docs/populate.html>

Mongoose - relationships

```
const run = async function () {
  let tutorial = new Tutorial({
    title: "Tutorial #1",
    author: "Teresa"
  });
  await tutorial.save();

  let category = new Category({
    name: "Node.js"
  });
  await category.save();

  tutorial.category = category._id;
  await tutorial.save();

  let updatedTutorial = await Tutorial.findById(tutorial._id)
    .populate("category", "-_id -__v");
  console.log(updatedTutorial);
};
run();
```

```
// Category document
{
  _id: 5db6af68c90cdd3a2c3038ab,
  name: "Node.js"
}

// Tutorial document with category
{
  _id: 609851dccda6e15c941eb27f,
  title: "Tutorial #1",
  author: "Teresa",
  category: 5db6af68c90cdd3a2c3038ab
}
```

Created documents in database
(the category field of Tutorial document contains reference ID to a category document)

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
 - Create a MongoDB using Atlas to store tutorials data

The screenshot displays the MongoDB Atlas web interface. The top navigation bar includes the P.PORTO logo, a settings gear, and links for Access Manager, Support, and Billing. The main interface is divided into three sections: Project Library, Atlas (selected), and Realm. The Atlas section shows a tree view of the database structure with 'tutorialsDB' expanded, revealing the 'tutorials' collection. The right-hand panel displays the details for the 'tutorialsDB.tutorials' collection, including its size (305B) and the number of documents (3). Below this, there are tabs for Find, Indexes, Schema, and Anti-Patterns. A filter bar is present with a placeholder query: `{"filter": "example"}`. The bottom section shows the query results, displaying a single document with the following fields: `_id` (ObjectId), `title` (JS: Node Tut #1), `description` (Tut#1 Description), `published` (false), and `__v` (0).

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database

2. Install required dependencies into your project

```
npm init -y  
npm install express mongoose cors dotenv --save  
npm install nodemon --save-dev
```

3. Configure MongoDB database & Mongoose

```
const config = {  
  /* don't expose password or any sensitive info, done only for demo */  
  // if environment variables are not defined, use default values  
  USER: process.env.DB_USER || 'dbTeresa',  
  PASSWORD: process.env.DB_PASSWORD || 'db123456789',  
  DB: process.env.DB_NAME || 'tutorialsDB'  
};  
  
config.URL =  
`mongodb+srv://${config.USER}:${config.PASSWORD}@cluster0.mun28.mongodb.net/${config.DB}?  
retryWrites=true&w=majority`;  
  
module.exports = config;
```

[config/db.config.js](#)

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
 4. Connect to database & define Mongoose models (tutorials and comments)

```
module.exports = (mongoose) => {  
  const schema = mongoose.Schema(  
    {  
      title: { type: String, required: [true, 'Title field is mandatory'] },  
      description: String,  
      published: { type: Boolean, default: false},  
      comments: [  
        { type: mongoose.Schema.Types.ObjectId, ref: 'comment' }  
      ]  
    },  
    { timestamps: false }  
  );  
  const Tutorial = mongoose.model("tutorial", schema);  
  return Tutorial;  
};
```

[models/tutorial.model.js](#)

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
 4. Connect to database & define Mongoose models (tutorials and comments)

```
module.exports = (mongoose) => {  
  const schema = mongoose.Schema(  
    {  
      author: String,  
      text: { type: String, required: [true, 'Text field is mandatory'] },  
    },  
    { timestamps: false }  
  );  
  const Comment = mongoose.model("comment", schema);  
  return Comment;  
};
```

[models/comment.model.js](#)

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
4. Connect to database & define Mongoose models

```
const dbConfig = require('../config/db.config.js');
const mongoose = require("mongoose");

const db = {};
db.mongoose = mongoose;
db.url = dbConfig.URL;

db.mongoose
  .connect(db.url, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(() => { console.log("Connected to the database!"); })
  .catch(err => {
    console.log("Cannot connect to the database!", err);
    process.exit();
  });

db.tutorials = require("../tutorial.model.js")(mongoose);
db.comments = require("../comment.model.js")(mongoose);

module.exports = db;
```

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
 5. Create the CRUD functions in the controllers

```
const db = require("../models");
const Tutorial = db.tutorials;

// Create and Save a new Tutorial: use object.save()
exports.create = async (req, res) => {...};

// Retrieve all Tutorials from the database: use Model.find()
exports.findAll = async (req, res) => {...};

// Find a single Tutorial with an id: use Model.findById(), and including its comments
exports.findOne = async (req, res) => {...};

// Update a Tutorial by the id in the request: use Model.findByIdAndUpdate()
exports.update = async (req, res) => {...};

// Delete a Tutorial with the specified id in the request: use Model.findByIdAndRemove() and
delete the comments related to it
exports.delete = async (req, res) => {...};
```

controllers/tutorials.controller.js

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
5. Create the CRUD functions in the controllers

```
const db = require("../models");
const Tutorial = db.tutorials;
const Comment = db.comments;

// Create and Save a new Comment into a certain tutorial:
// use comment.save\(\) to create a new comment document
// use Tutorial.findByIdAndUpdate\(\) (with the $push operator) to include the new comment
// reference in the tutorial document
exports.create = async (req, res) => {...};

// Delete a Comment (from a certain tutorial):
// use Comment.findByIdAndRemove\(\) to delete the comment document
// use Tutorial.findByIdAndUpdate\(\) (with the $pull operator) to remove the comment reference
// in the tutorial document
exports.delete = async (req, res) => {...};
```

`controllers/comments.controller.js`

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
5. Create the CRUD functions in the controllers

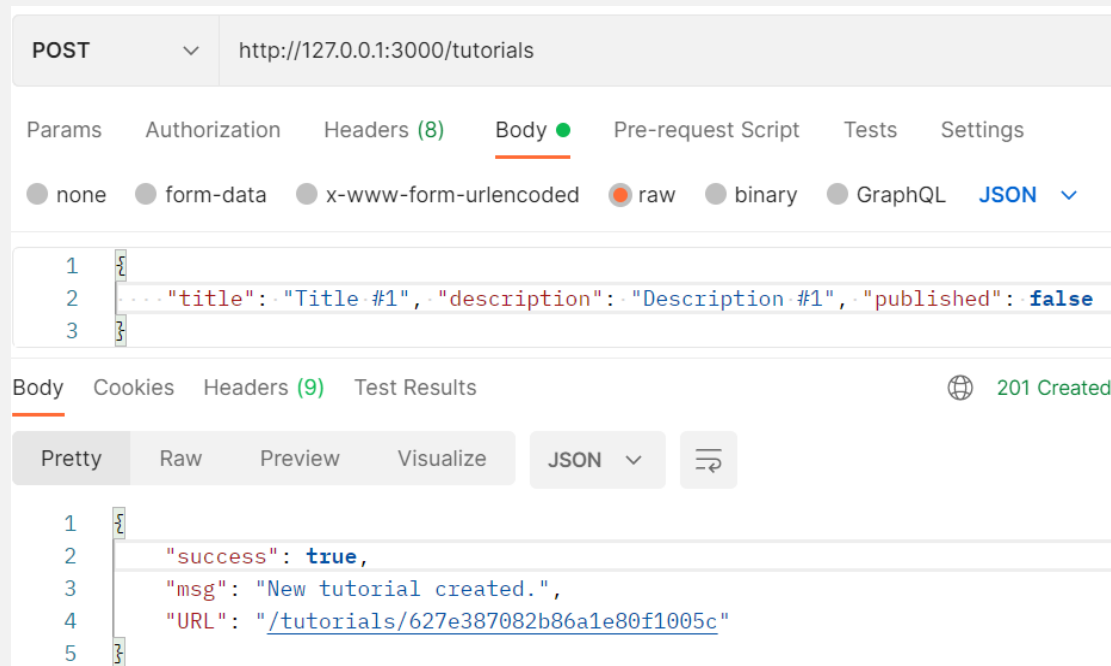
```
exports.create = async (req, res) => {
  const tutorial = new Tutorial({ // create an instance of a Tutorial model
    title: req.body.title,
    description: req.body.description,
    published: req.body.published
  });
  try {
    await tutorial.save(); // save document in the tutorials DB collection
    res.status(201).json({ // if save is successful, the returned promise will fulfill with the document saved
      success: true, msg: "New tutorial created.", URL: `/tutorials/${tutorial._id}` });
  }
  catch (err) {
    // capture mongoose validation errors
    if (err.name === "ValidationError") {
      let errors = [];
      Object.keys(err.errors).forEach((key) => {
        errors.push(err.errors[key].message);
      });
      return res.status(400).json({ success: false, msgs: errors });
    }
    res.status(500).json({ success: false,
      msg: err.message || "Some error occurred while creating the tutorial. " });
  }
};
```

controllers/tutorials.controller.js

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
- 5. Create the CRUD functions in the controllers

```
// create a new Tutorial
exports.create = async (req, res) => {
  ...
};
```



Successful
creation

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
- 5. Create the CRUD functions in the controllers

```
// create a new Tutorial
exports.create = async (req, res) => {
  ...
};
```

POST http://127.0.0.1:3000/tutorials

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   ... "description": "Description #1", "published": false
3 }
```

Body Cookies Headers (9) Test Results 400 Bad Request

Pretty Raw Preview Visualize JSON

```
1 {
2   "success": false,
3   "msgs": [
4     "Why no title?"
5   ]
6 }
```

Bad request on creation:
validation error -> no title data

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
 5. Create the CRUD functions in the controllers

```
// retrieve all Tutorials / or find by title
exports.findAll = async (req, res) => {
  const title = req.query.title;
  // build REGEX to filter tutorials titles with a sub-string - i will do a case insensitive match
  (https://docs.mongodb.com/manual/reference/operator/query/regex/)
  const condition = title ? { title: new RegExp(title, 'i') } : {};
  try {
    // find function parameters: filter, projection (select) / returns a list of documents
    let data = await Tutorial.find(condition)
      .select('title description published') // select the fields: do not show versionKey field
      .exec(); // execute the query
    res.status(200).json({success: true, tutorials: data});
  }
  catch (err) {
    res.status(500).json({
      success: false, msg: err.message || "Some error occurred while retrieving the tutorials."
    });
  }
};
```

`controllers/tutorials.controller.js`

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
- 5. Create the CRUD functions in the controllers

```
// retrieve all Tutorials / or find by title
exports.findAll = async (req, res) => {
  ...
};
```

controllers/tutorials.controller.js

GET http://127.0.0.1:3000/tutorials

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
-----	-------	-------------

Body Cookies Headers (9) Test Results 200 OK

Pretty Raw Preview Visualize JSON

```
1 {
2   "success": true,
3   "tutorials": [
4     {
5       "published": false,
6       "_id": "627e387082b86a1e80f1005c",
7       "title": "Title #1",
8       "description": "Description #1"
9     },
10    {
11      "published": true,
12      "_id": "627e3a79b401f57f805e6e40",
```

tutorialsDB.tutorials

STORAGE SIZE: 36KB TOTAL DOCUMENTS: 2 INDEXES TOTAL: 2

Find Indexes Schema Anti-Patterns (0)

FILTER { field: 'value' }

QUERY RESULTS 1-2 OF 2

```
_id: ObjectId("627e387082b86a1e80f1005c")
published: false
title: "Title #1"
description: "Description #1"
__v: 0
```

```
_id: ObjectId("627e3a79b401f57f805e6e40")
published: true
title: "Title #2"
description: "Description #2"
__v: 0
```

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
- 5. Create the CRUD functions in the controller

```
// retrieve all Tutorials / or find by title
exports.findAll = async (req, res) => {
  ...
};
```

controllers/tutorials.controller.js

GET http://127.0.0.1:3000/tutorials?title=2

Params Authorization Headers (6) Body Pre-request S

Key	Value
title	2

Body Cookies Headers (9) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "success": true,
3   "tutorials": [
4     {
5       "published": true,
6       "_id": "627e3a79b401f57f805e6e40",
7       "title": "Title #2",
8       "description": "Description #2"
9     }
10  ]
11 }
```

Read more about queries and query filtering:
<https://mongoosejs.com/docs/queries.html>
https://mongoosejs.com/docs/tutorials/query_casting.html

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
5. Create the CRUD functions in the controller

```
// Find a single Tutorial (with an id)
exports.findOne = async (req, res) => {
  try {
    // findById(id) === findOne( {_id : id })
    const tutorial = await Tutorial.findById(req.params.tutorialID)
      .populate("comments") // include the tutorial comments
      .exec();
    // no data returned means there is no tutorial in DB with that given ID
    if (tutorial === null)
      return res.status(404).json({
        success: false, msg: `Cannot find any tutorial with ID ${req.params.tutorialID}.`
      });
    res.status(200).json(success: false, tutorial: tutorial);
  }
  catch (err) {
    res.status(500).json({
      success: false, msg: `Error retrieving tutorial with ID ${req.params.tutorialID}.`
    });
  }
};
```

[controllers/tutorials.controller.js](#)

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
- 5. Create the CRUD functions in the controllers

```
// Find a single Tutorial (with an id)
exports.findOne = async (req, res) => {
  ...
};
```

[controllers/tutorials.controller.js](#)

The screenshot shows a REST client interface with a GET request to `{{hostname}}/tutorials/644a52c323788b572880c578`. The response is displayed in JSON format, showing a successful result with a tutorial object containing a comment.

```
1 {
2   "success": true,
3   "tutorial": {
4     "published": false,
5     "comments": [
6       {
7         "_id": "644a573dacbb7f390c5c87d6",
8         "author": "João",
9         "text": "Comment #1 from João"
10      },
11      {
12        "_id": "644a57ffec03a84fd099ce53",
```

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
- 5. Create the CRUD functions in the controllers

```
// Find a single Tutorial (with an id)
exports.findOne = async (req, res) => {
  ...
};
```

controllers/tutorials.controller.js



The id is cast based on the Schema before sending the command:

https://mongoosejs.com/docs/api.html#model_Model.findById

API could be improved to read errors of type “CastError” and send a Bad Request error

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
- 5. Create the CRUD functions in the controllers

```
// Find a single Tutorial (with an id)
exports.findOne = async (req, res) => {
  ...
};
```

[controllers/tutorials.controller.js](#)

TUTORIALS API / Get a single tutorial

GET ⌵ http://localhost:8080/tutorials/7099628843a1a33f400d5234

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Body Cookies Headers (9) Test Results 🌐 Status: 404 Not Found

Pretty Raw Preview Visualize JSON ⌵ ↺

```
1 {
2   "message": "Not found Tutorial with id 7099628843a1a33f400d5234."
3 }
```

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
- 5. Create the CRUD functions in the controllers

```
// Update a Tutorial (given its id)
exports.update = async (req, res) => {
  ...
};
```

controllers/tutorials.controller.js

TUTORIALS API / Update one tutorial

PUT

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON**

```
1 {
2   ... "title": "JS: Node Tut #1",
3   ... "description": "Tut#1-Description",
4   ... "published": false
5 }
```

Body Cookies Headers (9) Test Results Status: 200 OK

Pretty Raw Preview Visualize **JSON**

```
1 {
2   "message": "Tutorial id=6099628843a1a33f400d5234 was updated successfully."
3 }
```

Mongoose accepts partial updates, so if POST must perform a full update is, one must check if request body has all resource params;

Mongoose can validate the given params, using the options `runValidators`:
`A.findByIdAndUpdate(id, updateObj, {runValidators: true})`

Mongoose - exercise

- Exercise: Tutorials REST API using a MongoDB database
5. Create the CRUD functions in the controllers

```
// Create a new comment into a certain tutorial
exports.create = async (req, res) => {
  try {
    const comment = new Comment({
      author: req.body.author ? req.body.author : "", text: req.body.text
    });
    let data = await comment.save();
    const tutorial = await Tutorial.findByIdAndUpdate(
      req.params.idT,
      { $push: { comments: [data._id] } },
      { runValidators: true, useFindAndModify: false }
    ).exec();
    // no data returned means there is no tutorial in DB with that given ID
    if (tutorial === null)
      return res.status(404).json({
        success: false, msg: `Cannot find any tutorial with ID ${req.params.idT}.`
      });
    res.status(201).json({ success: true,
      msg: "New Comment created.", URL: "/tutorials/" + req.params.idT + "/comments/" + data._id });
  }
  catch (err) {
    res.status(500).json({ message: err.message || "Some error occurred while creating a new
comment." });
  }
};
```

[controllers/comments.controller.js](#)