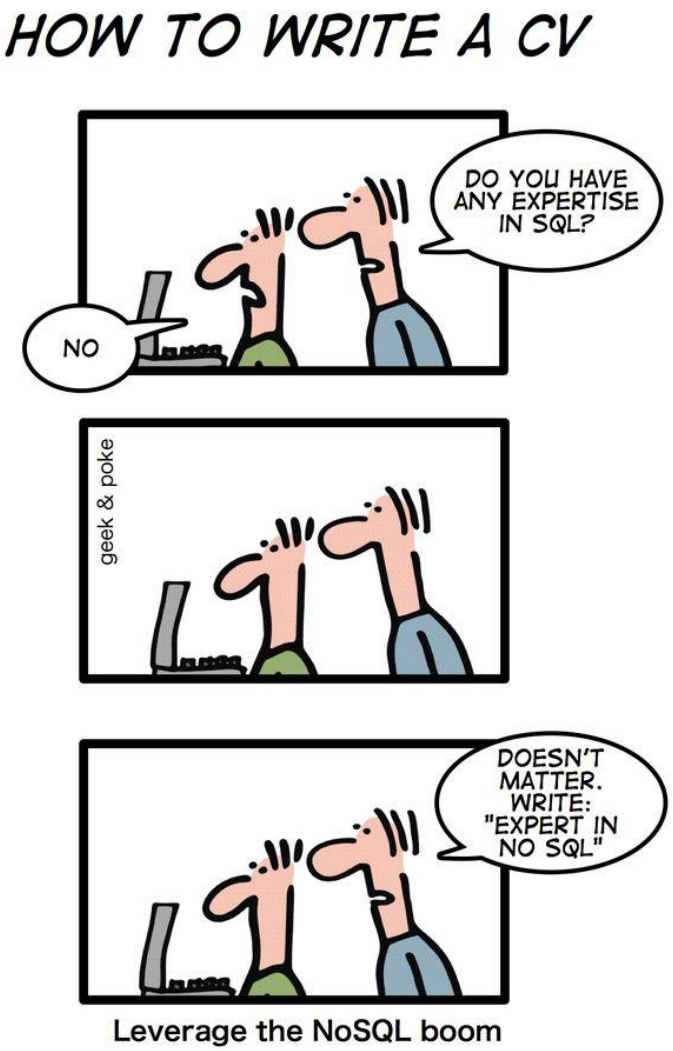
P.PORTO

POLITÉCNICO
DO PORTO
ESMAD

PROGRAMAÇÃO WEB II
TSIW

SUMMARY

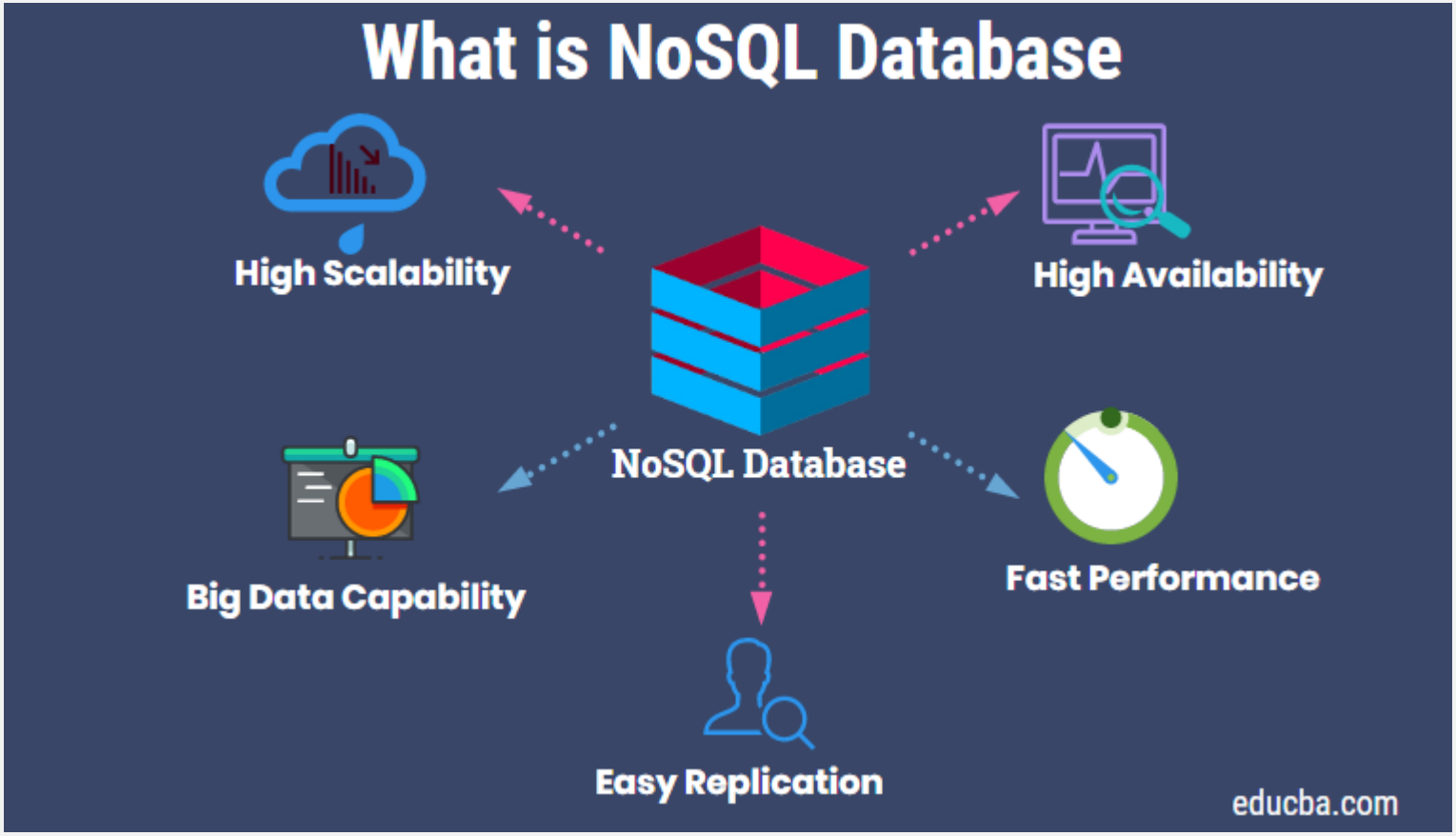
- SQL versus NoSQL
- MongoDB
- ODM: Object Document Mapping
- Mongoose



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

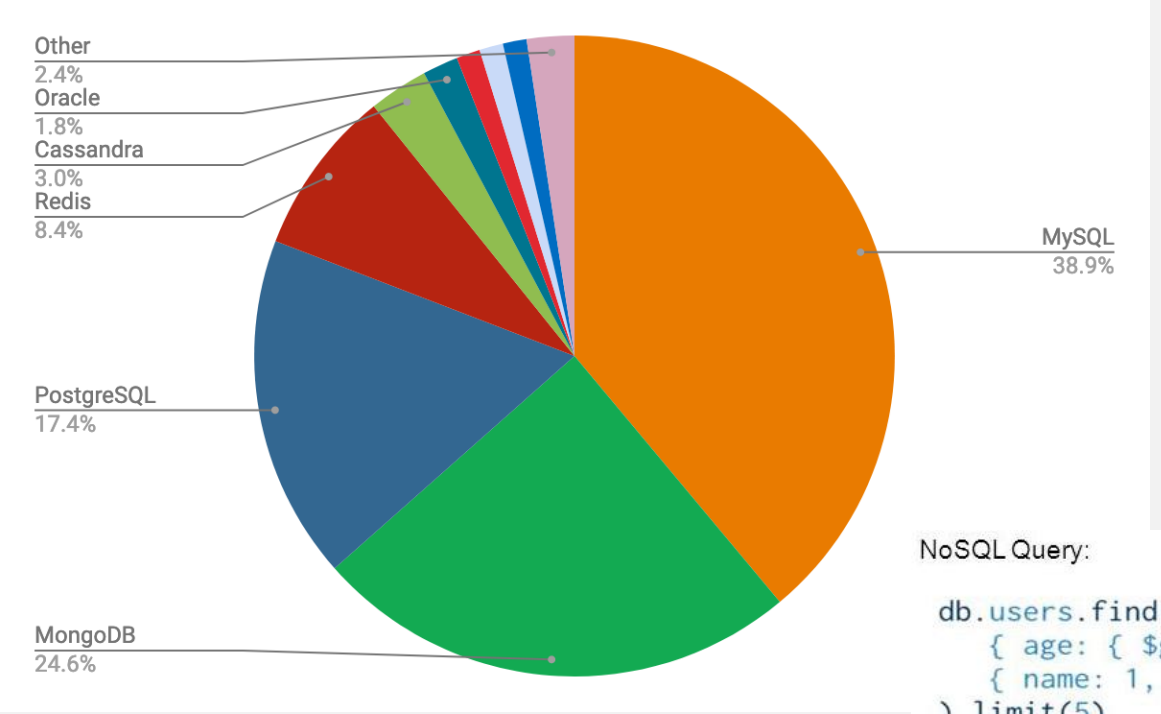


SOURCE: https://www.educba.com/what-is-nosql-database/

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(                          ← collection
    { age: { $gt: 18 } },               ← query criteria
    { name: 1, address: 1 }             ← projection
).limit(5)                              ← cursor modifier
```

SQL Query:

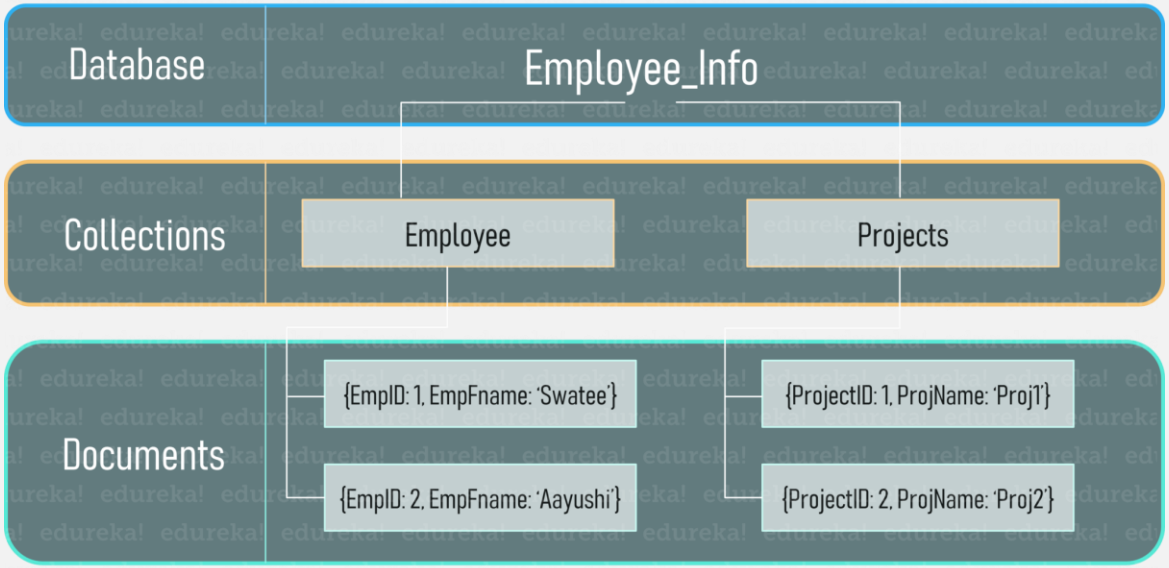
```
SELECT _id, name, address   ← projection
FROM   users                ← table
WHERE  age > 18             ← select criteria
LIMIT  5                    ← cursor modifier
```

SOURCE: https://medium.com/nerd-for-tech/sql-vs-nosql-faef10e3852d

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	DELETE

```
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

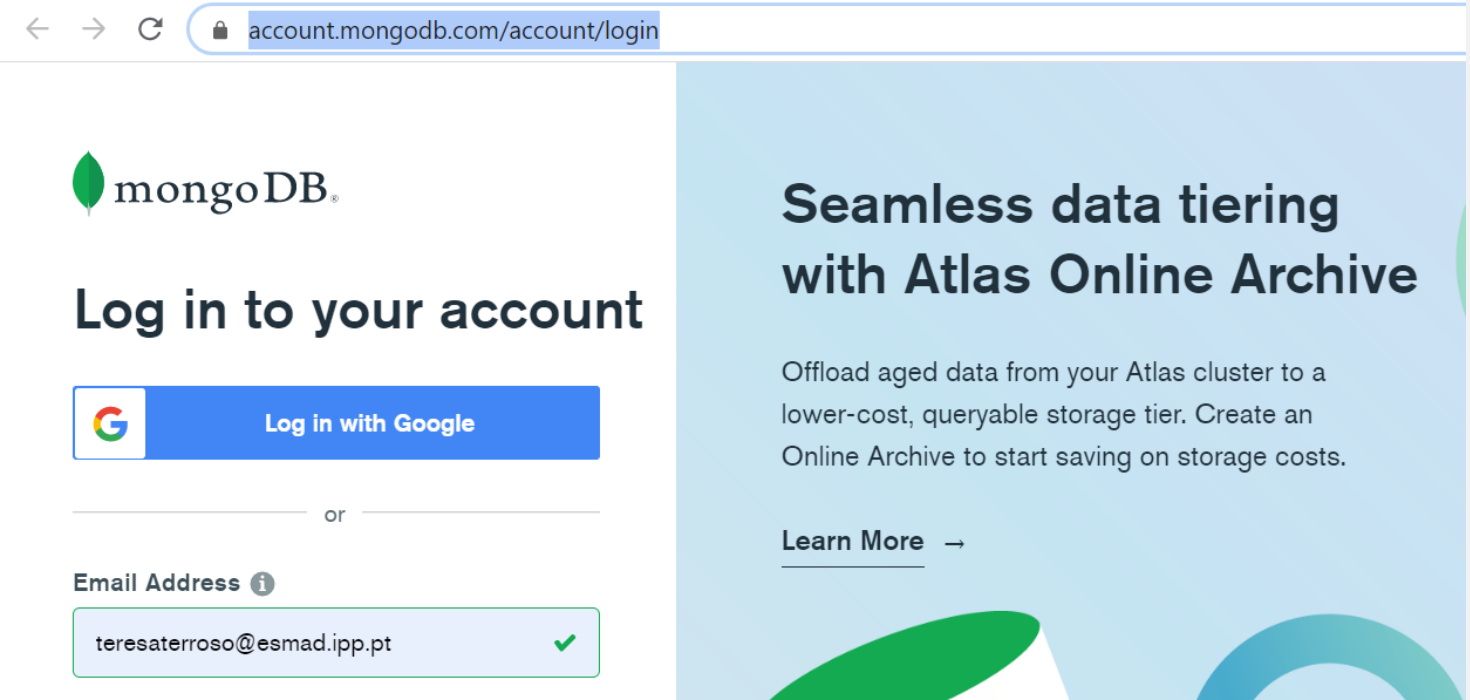
- **ODM** is the ORM for databases oriented towards non-relational documents, such as MongoDB, CouchDB, …
- ODM provides **persistence services** for these data stores
- ODM allows to:
  - **store** new documents, updating and removing existing ones
  - **track** what has been modified, execute and save 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
  - …

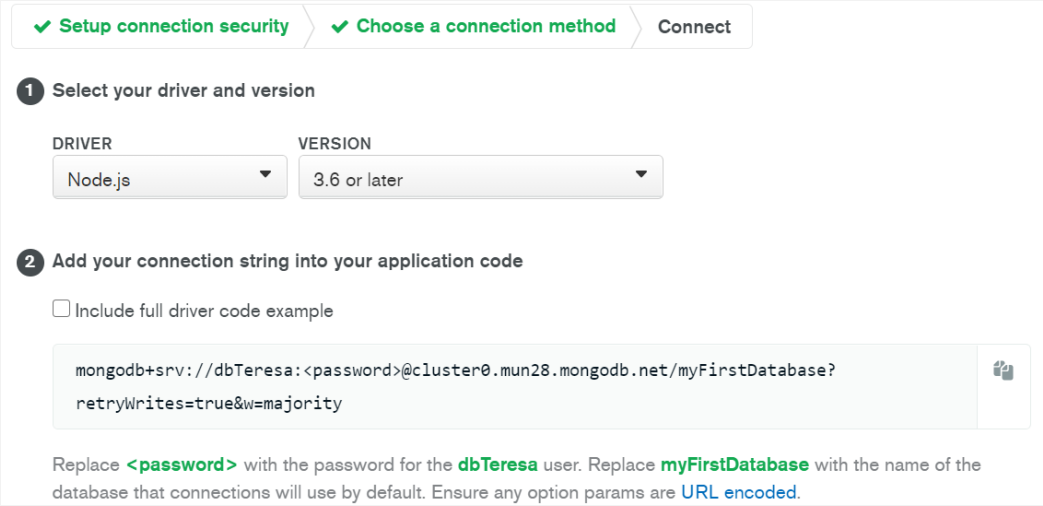
Get started with Atlas

- 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




```
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

```
const mongoose = require("mongoose");
  const { Schema } = mongoose;

  const blogSchema = new Schema({
    title:  { type: Date, 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)`

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

const blogSchema = new mongoose.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
```

schema definition

  - 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: convert '59' from a string to a number
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

- Example 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); // error.message: "Path `name` is required"
```

Mongoose in Node.js

- Example 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.');
assert.ok(!error.errors['bacon']); // no errors with the 'bacon' field
```

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

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']
  }
});
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!');
```

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

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' );
        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

- Model definition and 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' });
// saves document it into the DB
let smallTank = small.save();
// 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.findByIdAndDelete()`
  - `Model.findByIdAndUpdate()`
  - `Model.findOne()`
  - `Model.findOneAndDelete()`
  - `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 person = await Person.findOne({ 'lastname': 'Ghost' }, 'lastname occupation’);
if (person != null)  // if document exists in DB
     console.log(person.lastname, person.occupation);
```

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

// build query to 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
const person = await query.exec()
if (person != null)
     console.log(person.lastname, person.occupation);
```

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

// chaining syntax
let person = await 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 name and length fields
let adventure = await Adventure.findById(id, 'name length').exec();
```

Mongoose in Node.js

- `findByIdAndDelete(`*`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.findByIdAndDelete(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('createdDate').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

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

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

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/

```
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 });
};

run();
```

new model instance (without ANY image)

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

IMPORTANT NOTES:

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

Mongoose - relationships

```
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();
};
run();
```

Server App

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

```
// 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 tutorial.save();


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

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

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 <u>before</u> 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 = Comment;
```

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: comment._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");

};
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: Post REST API using a MongoDB database
  1. Create a MongoDB using Atlas to store Posts and Users data
  2. Install required dependencies into your project

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

  3. Use the same folder/files configuration and configure Mongoose connection

```
NODE_ENV=development
# Server configuration
PORT=3000
HOST='127.0.0.1'
# Database connection information
DB_HOST: 'yourDBhost'
DB_USER: 'yourDBuser'
DB_PASSWORD: 'yourDBpasswd'
DB_NAME: 'yourDBname'
```

File .env under the project root folder

Mongoose - exercise

- Exercise: Post REST API using a MongoDB database
  4. Define Mongoose models (only Posts and Users)

models/users.model.js

```
module.exports = (mongoose) => {
    const schema = new mongoose.Schema(
        {
            username: {
                type: String, unique:  [true, 'Username {VALUE} already in use!'],
                required: [true, 'Username can not be empty or null!']
            },
            password: { type: String, required: [true, 'Password can not be empty or null!'] },
            role: {
                type: String,
                enum: {
                    values: ['admin', 'editor'],
                    message: '{VALUE} is not supported! Role must be admin or editor'
                },
                required: [true, 'Role can not be empty or null!']
            },
        },
        { timestamps: false }
    );
    const User = mongoose.model("user", schema);
    return User;
};
```

Mongoose - exercise

models/posts.model.js

```
module.exports = (mongoose) => {
    const schema = new mongoose.Schema(
        {
           title: {
                type: String, required: [true, 'Username can not be empty or null!'],
                minlength: [5, 'Title must be at least 5 characters long!'],
                maxlength: [50, 'Title must be at most 50 characters long!']
            },
            description: { type: String, required: [true, 'Description can not be empty or null!']},
            published: {
                type: Boolean, default: false,
                validate: {
                    validator: function (value) { return typeof value === 'boolean'; },
                    message: 'Published must be a boolean value.'
                }
            },
            views: {…}, publishedAt: {…},
            author: {
                type: mongoose.Schema.Types.ObjectId, ref: 'user',
                required: [true, 'Author can not be empty or null!']
            },
        },
        { timestamps: false }
    );
    const Post = mongoose.model("post", schema);
    return Post;
};
```

Mongoose - exercise

- Exercise: Post REST API using a MongoDB database
  5. Connect to database

models/db.js

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

const db = {};
db.mongoose = mongoose;

(async () => {
    try {
        const config = {
            USER: process.env.DB_USER, PASSWORD: process.env.DB_PASSWORD,
            DB: process.env.DB_NAME, HOST: process.env.DB_HOST
        };
        const mongoDBURL =
`mongodb+srv://${config.USER}:${config.PASSWORD}@${config.HOST}/${config.DB}?retryWrites=true&w=majority`;
        await db.mongoose.connect(mongoDBURL);
        console.log("Connected to the database!");
    } catch (error) {
        console.log("❌ Unable to connect to the database:", error); process.exit(1);
    }
})();

db.Post = require("./posts.model.js")(mongoose); db.User = require("./users.model.js")(mongoose);
module.exports = db;
```

Mongoose - exercise

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

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

/* Get all posts from a certain user:
 use db.User.findByID(req.params.id) to find the user
(error 404 if not found)
 use db.Post.find({ author: req.params.id }) find all
posts created by the given user */
let getPostsFromUser = async (req, res, next) => {{…};

module.exports = {
    getPostsFromUser
}
```

controllers/users.controller.js

GET http://localhost:3000/users/681deef760b9be11bd1fe3ec/posts

Params Authorization Headers (7) Body Scripts Settings

Body Cookies Headers (7) Test Results

{} JSON Preview Visualize


```
1  {
2      "_id": "681deef760b9be11bd1fe3ec",
3      "username": "ana_dev",
4      "role": "editor",
5      "posts": [
6          {
7              "_id": "681df1dcc16e8344f6a84c1c",
8              "title": "JSON Web Token",
9              "description": "Learn how to implement JSON Web Token (
                   applications.",
10             "published": false,
11             "views": 0,
12             "publishedAt": "2025-05-09T12:15:24.723Z",
13             "links": [
14                 {
15                     "rel": "self",
16                     "href": "/posts/681df1dcc16e8344f6a84c1c",
17                     "method": "GET"
18                 },
```

Mongoose - exercise

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

controllers/posts.controller.js

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

let getAllPosts = async (req, res, next) => {{…};

let getPostById = async (req, res, next) => {{…};

let addPost = async (req, res, next) => {{…};

let updatePost = async (req, res, next) => {{…};

let deletePost = async (req, res, next) => {{…};


module.exports = {
    getAllPosts, getPostById, addPost, updatePost, deletePost
}
```

Mongoose - exercise

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

```
// list all posts, with filter,
sort and pagination options
let getAllPosts = async (req, res, next) => {{…};
```

controllers/posts.controller.js

GET http://localhost:3000/posts?sort=views&&order=asc

Params • Authorization Headers (7) Body Scripts Settings

Body Cookies Headers (7) Test Results 200 OK • 240 ms • 1.34 KB

{} JSON Preview Visualize


```
1  {
2      "totalPages": 1,
3      "currentPage": 1,
4      "total": 2,
5      "data": [
6          {
7              "_id": "681df1dcc16e8344f6a84c1c",
8              "title": "JSON Web Token",
9              "description": "Learn how to implement JSON Web Token (JWT) authentication in your Node.
                   applications.",
10             "published": false,
11             "views": 0,
12             "author": "681deef760b9be11bd1fe3ec",
13             "publishedAt": "2025-05-09T12:15:24.723Z",
14             "links": [
15                 {
16                     "rel": "self",
17                     "href": "/posts/681df1dcc16e8344f6a84c1c",
18                     "method": "GET"
19                 },
```


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

Mongoose - exercise

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

```
// Find a single post (with an id): include details about the author
let getPostById = async (req, res) => {…};
```

controllers/posts.controller.js

GET http://localhost:3000/posts/681df1dcc16e8344f6a84c1c Send

Params Authorization Headers (9) Body Scripts Settings Cookies

Body Cookies Headers (7) Test Results 200 OK 174 ms 723 B Save Response

{} JSON Preview Visualize


```
1  {
2      "_id": "681df1dcc16e8344f6a84c1c",
3      "title": "JSON Web Token",
4      "description": "Learn how to implement JSON Web Token (JWT) authentication in your Node.js and Express applications.",
5      "published": false,
6      "views": 0,
7      "author": {
8          "_id": "681deef760b9be11bd1fe3ec",
9          "username": "ana_dev",
10         "role": "editor"
11     },
12     "publishedAt": "2025-05-09T12:15:24.723Z",
13     "links": [
14         {
15             "rel": "modify",
16             "href": "/posts/681df1dcc16e8344f6a84c1c",
17             "method": "PUT"
18         },
```


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: : Post REST API using a MongoDB database
  6. Create the CRUD functions in the controllers

```
// validate author id and create new post
let addPost = async (req, res) => {…};
```

controllers/posts.controller.js

POST http://localhost:3000/posts

Params Authorization Headers (9) Body Scripts Settings

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

```
1 {
2 "title": "JSON Web Token",
3 "description": "Learn how to implement JSON Web Token (JWT) authentication in your Node.js and E
4 "author": 2
5 }
```

Body Cookies Headers (7) Test Results 201 Created 126 ms 4

{} JSON Preview Visualize

```
1 {
2     "msg": "Post successfully created.",
3     "links": [
4         {
5             "rel": "self",
6             "href": "/posts/4",
7             "method": "GET"
8         },
```

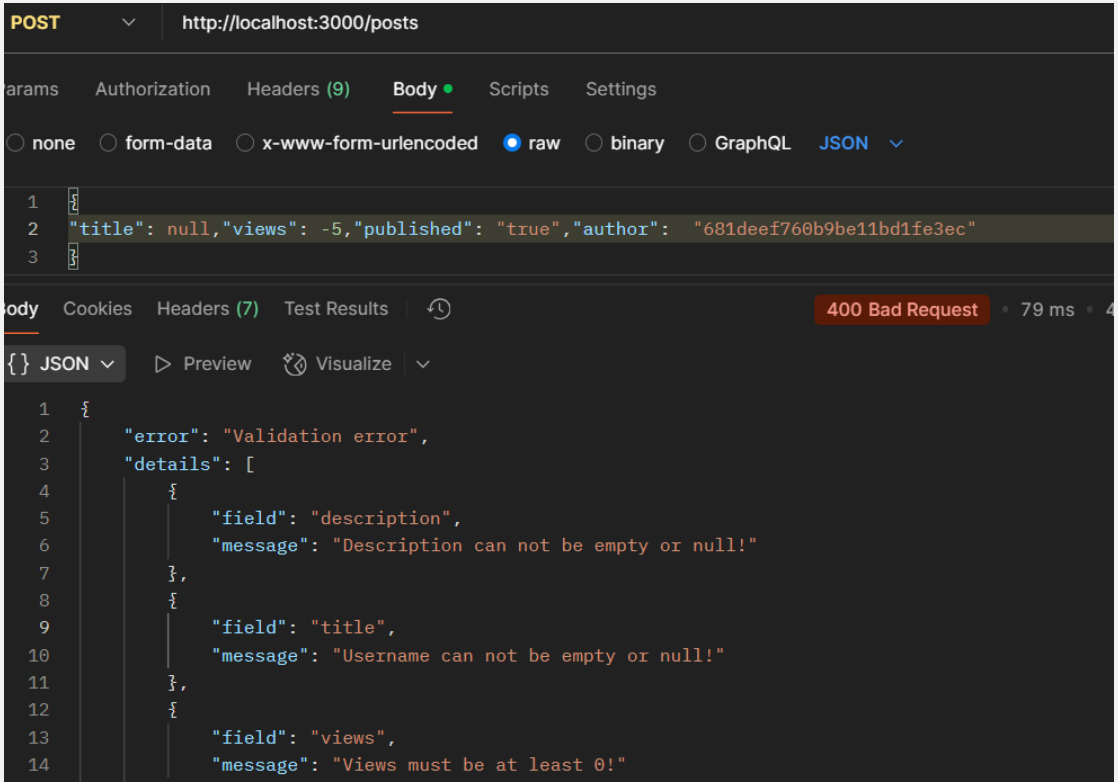
Successfull creation

Mongoose - exercise

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

```
// validate author id and create new post
let addPost = async (req, res) => {…};
```

controllers/posts.controller.js



Bad request!

Mongoose - exercise

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

```
// update a given post
let updatePost = async (req, res) => {…};
```

controllers/posts.controller.js

PUT http://localhost:3000/posts/681df1dcc16e8344f6a84c1c

Params Authorization Headers (9) Body Scripts Settings

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

```
1 {
2     "views": 10
3 }
```

Body Cookies Headers (7) Test Results

{} JSON Preview Visualize

```
1 {
2     "message": "Missing required fields: title, description"
3 }
```

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

Mongoose can validate the given params, using option `runValidators`:

```
A.findByIdAndUpdate(id, updateObj, {runValidators: true})
```