P.PORTO

POLITÉCNICO
DO PORTO
ESMAD

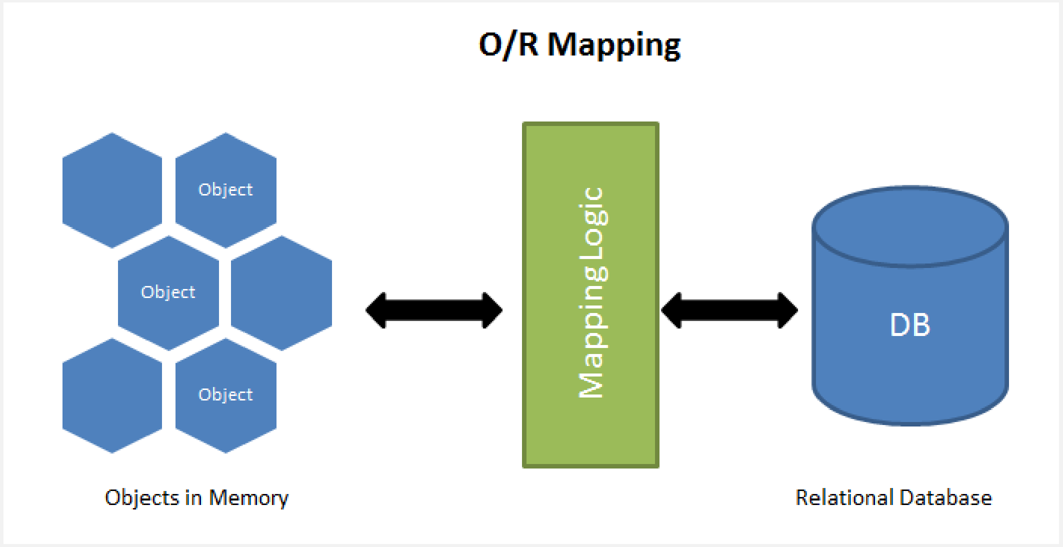
PROGRAMAÇÃO WEB II
TSIW

SUMMARY

- ORM - Object-Relational Mapper
- Sequelize: promise-based Node.js ORM with support to MySQL DBs

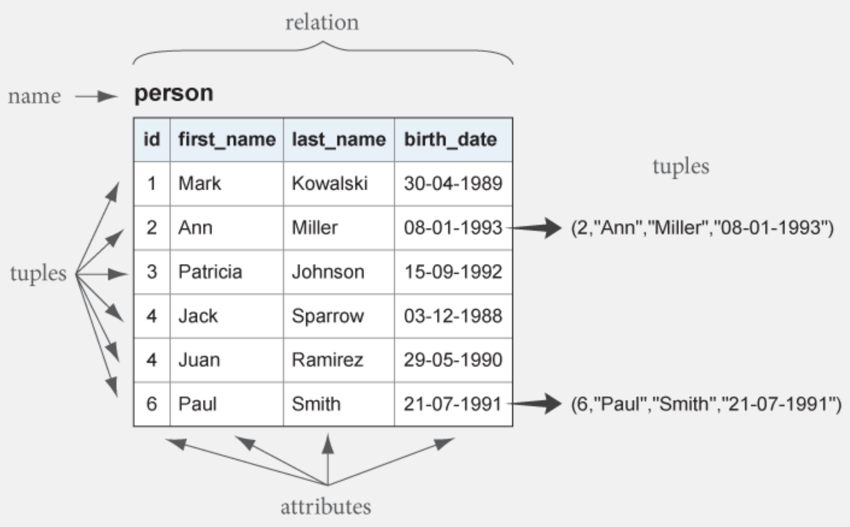
ORM

- Object-Relational Mapper: library that allows to write queries using the object-oriented paradigm of your preferred programming language
  - programming technique for converting data between incompatible type systems using object-oriented programming languages
  - ORM sets the mapping between the set of objects which are written in a programming language like JavaScript and a relational database like MySQL
  - It hides and encapsulates the SQL queries into "virtual database objects" and, instead of SQL queries, one can directly use the objects' methods to implement the SQL query



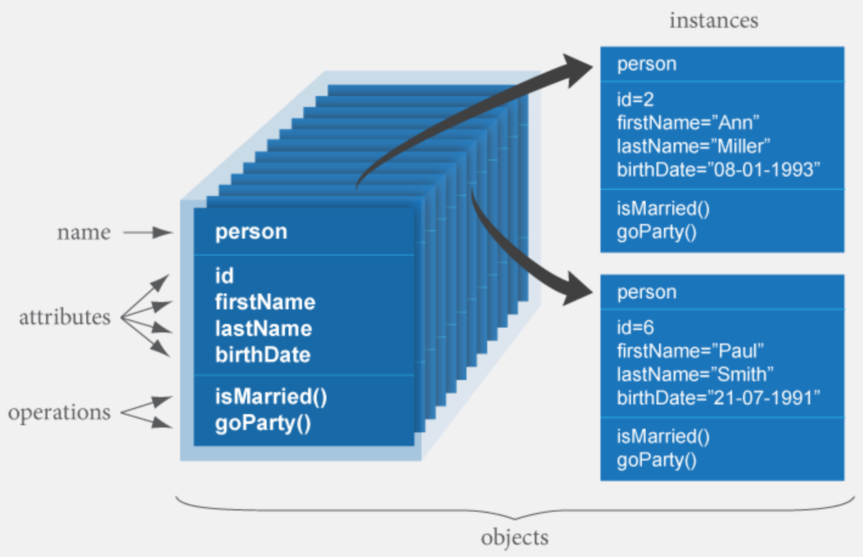
ORM

Data representation in a relational database


relation
name
person
tuples
tuples
(2,"Ann","Miller","08-01-1993")
(6,"Paul","Smith","21-07-1991")
attributes

id	first_name	last_name	birth_date
1	Mark	Kowalski	30-04-1989
2	Ann	Miller	08-01-1993
3	Patricia	Johnson	15-09-1992
4	Jack	Sparrow	03-12-1988
4	Juan	Ramirez	29-05-1990
6	Paul	Smith	21-07-1991

Data representation in object-oriented programming



ORM: pros and cons

- Advantages of using ORMs:
  - let the developer think in terms of objects rather than tables
  - no need to write SQL code
  - advanced features like eager or lazy loading, soft deletion, ...
  - database independent: no need to write code specific to a particular database
  - reduces code and allows developers to focus on their business logic, rather than complex database queries
  - most ORMs provide a rich query interface

ORM: pros and cons

- Disadvantages of using ORMs:
  - complex (because they handle a bidirectional mapping); their complexity implies a grueling learning curve – they have a special query language which developers must learn
  - provides only a leaky abstraction over a relational data store
  - systems using an ORM can perform badly if, due to naive interactions with the underlying database, one uses ORM without knowing SQL
  - ORM, by adding a layer of abstraction, speeds up the development but adds overhead to the application

ORM - Sequelize

- Sequelize: promise-based Node.js ORM for relational databases like Postgres, MySQL, MariaDB, SQLite, Microsoft SQL Server, among others
  - Has been around for a long time – 2011, has thousands of GitHub stars and is used by tons of applications
  - It is stable and has plenty of documentation available online
  - Sequelize has a large feature set that covers: queries, scopes, relations, transactions, raw queries, migrations, read replication, etc.

- **Installation**: sequelize is available via npm

```
npm install --save sequelize
```

You'll also have to manually install the driver for your database of choice:

```
npm install --save mysql2    #for MySQL databases
```

Comparison between mysql and mysql2 node modules: https://npmcompare.com/compare/mysql,mysql2

Sequelize

- Simple example: connection to a MySQL database and create a DB entry

```
const { Sequelize, DataTypes } = require('sequelize');
const sequelize = new Sequelize('database', 'username', 'password',
{
  host: 'host',
  dialect: 'mysql'
});
```

Database connection

```
const User = sequelize.define("user",
{
  username: DataTypes.STRING,
  birthday: DataTypes.DATE
});
```

Model definition

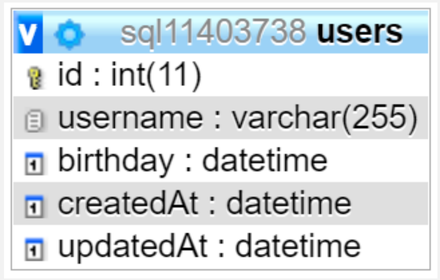
```
(async () => {
  await sequelize.sync();
```

creates the table if it doesn't exist (and does nothing if it already exists)

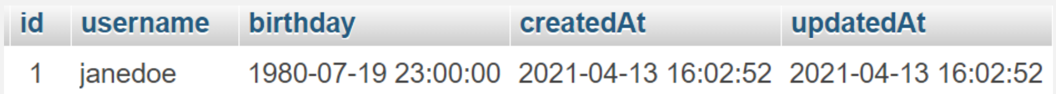
```
  const jane = await User.create({
    username: 'janedoe',
    birthday: new Date(1980, 6, 20)
  });
```

Instantiate object and save it in database

```
  console.log(jane.toJSON());
})();
```



sql11403738 users
- id : int(11)
- username : varchar(255)
- birthday : datetime
- createdAt : datetime
- updatedAt : datetime



id	username	birthday	createdAt	updatedAt
1	janedoe	1980-07-19 23:00:00	2021-04-13 16:02:52	2021-04-13 16:02:52

Sequelize

- **Connecting to a database**: to connect to the database, you must create a Sequelize instance
  - Example for MySQL databases

```
const sequelize = new Sequelize('database', 'username', 'password', {
     host: 'hostname',
     dialect: 'mysql'
});
```

Replace with your MySQL database credentials

  - **Terminology convention**: observe that Sequelize refers to the library itself while sequelize refers to an instance of Sequelize, which represents a connection to one database
  - This is the recommended convention, and is followed throughout the ORM documentation

Sequelize

- You can test the connection using `authenticate()`, which creates an instance to check whether the connection is working:

```
try {
    await sequelize.authenticate();
    console.log('Connection has been established successfully.');
 } catch (error) {
    console.error('Unable to connect to the database:', error);
 }
```

- **Closing the connection:** Sequelize will keep the connection open by default and use the same connection for all queries. If you need to close the connection, call `sequelize.close()`
  - Once called, it's impossible to open a new connection
  - For that, one will need to create a new Sequelize instance to access the database again

Sequelize: promises

- **Promises:** most of the methods provided by Sequelize are asynchronous and therefore return Promises
  - ➢ you can use the Promise API (using `then`, `catch`, `finally`)
  - ➢ or you can use `async` and `await` as well

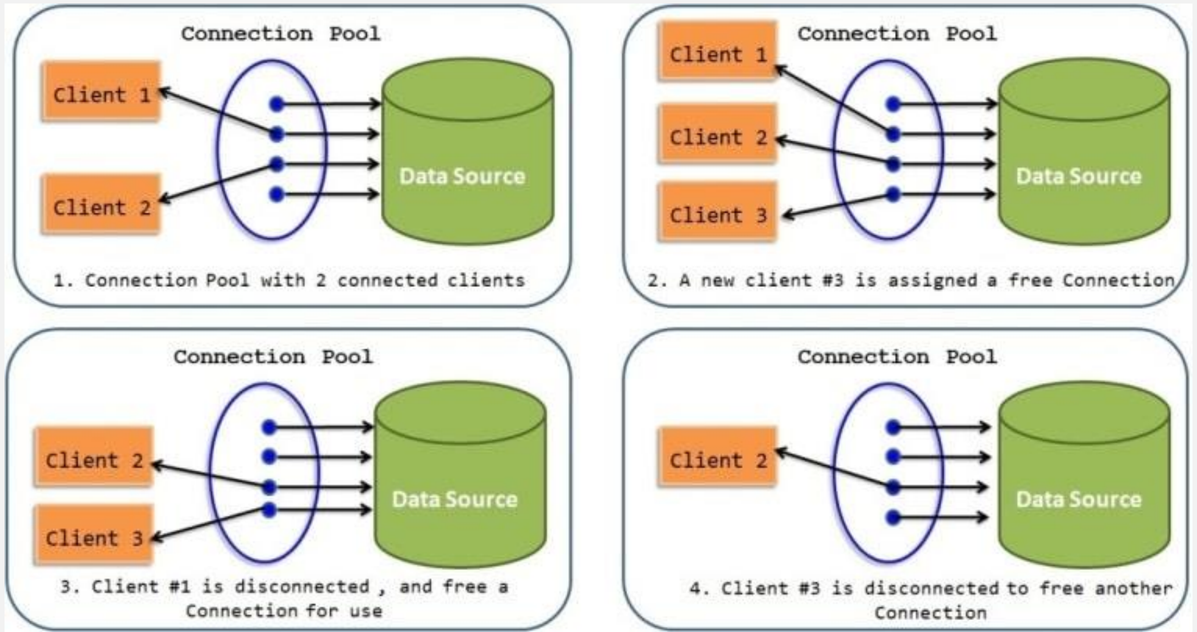
```
sequelize.authenticate()
    .then(() => {
        console.log('Connection has been established successfully.');
    })
    .catch(err => {
        console.error('Unable to connect to the database:', err);
    });
```

Two ways of using promises

```
(async () => {
    try {
        await sequelize.authenticate();
        console.log('Connection has been established successfully.');
    } catch (error) {
        console.error('Unable to connect to the database:', error);
    }
})();
```

Connection pooling

- Database connection pooling is a method used to keep database connections open so they can be reused by others
  - ➢ When an application starts, several connections are created and added to the pool
  - ➢ Then, the pool starts to assign connections for different requests. After it reached its maximum number of connections at the same time (something configured in the app), it waits until one of the connections is free again


Connection Pool
Client 1
Client 2
Data Source
1. Connection Pool with 2 connected clients
Connection Pool
Client 1
Client 2
Client 3
Data Source
2. A new client #3 is assigned a free Connection
Connection Pool
Client 2
Client 3
Data Source
3. Client #1 is disconnected , and free a Connection for use
Connection Pool
Client 2
Data Source
4. Client #3 is disconnected to free another Connection

Sequelize

- **Connection pooling:** Sequelize helps in the configuration of a connection pool

```
const sequelize = new Sequelize('database', 'username', 'password', {
     host: 'hostname',
     dialect: 'mysql',
     pool: {
          max: 5,   //maximum number of connections in pool
          min: 1,   //minimum number of connections in pool
          acquire: 30000, // maximum time (ms) that pool will try to get
connection before throwing error
          idle: 10000 // maximum time (ms) that a connection can be idle before
being released
          }
});
```

Sequelize - models basics

- Models are the essence of Sequelize, since a model is an abstraction that represents a table in the database
  - ➢ The model tells Sequelize several things about the entity it represents, such as the table name in the database and which columns it has (and their data types)
- Models can be defined in two equivalent ways in Sequelize:
  1. Calling `sequelize.define(modelName, attributes, options)`
  2. Extending class Model and calling `init(attributes, options)`
- After a model is defined, it is available within `sequelize.models` by its model name

Sequelize - models basics

- Model definition examples

Model name

```
const User = sequelize.define("User", {
      // ... (attributes)
}, {
      // other model options here
});
// `sequelize.define` also returns the model
console.log(User === sequelize.models.User); // true
```

```
class User extends Model {}
User.init({
  // ... (attributes)
}, {
  // other model options go here
  sequelize, // pass the connection instance
  modelName: 'User' // choose the model name
});

// the defined model is the class itself
console.log(User === sequelize.models.User); // true
```

Sequelize - models basics

- In the previous examples, the table name was not explicitly defined
- The model name in Sequelize does not have to be the same name of the table it represents in the database
- Usually, models have singular names (such as *User*) while tables have pluralized names (such as *Users*)
  - **Sequelize automatically pluralizes the model name** and uses that as the table name
  - And it is smart since models named *Person* will correspond to tables named *People*
- This behaviour however this is fully configurable:

```
sequelize.define("User", {
     // ... (attributes)
}, {
     // table name is equal to the model name
     freezeTableName: true
});
```

```
sequelize.define("User", {
     // ... (attributes)
}, {
     // tell Sequelize the table name directly
     tableName: 'Employees'
});
```

Sequelize - models basics

- What if the table actually doesn't even exist in the database, when a model is defined with Sequelize? What if it exists, but it has different columns, less columns, or any other difference?
- **Model synchronization**: a model can be synchronized with the database by calling `model.sync(options)`, which will automatically perform an SQL query to the database

```
// creates the table if it doesn't exist (and does nothing if it already exists)
User.sync();

// creates the table, dropping it first if it already existed
User.sync({ force: true });

// checks the table in the database (which columns it has, what are their data types, etc.),
and then performs the necessary changes to make it match the model
User.sync({ alter: true });

// automatically synchronize ALL models
sequelize.sync({ force: true });
```

Sequelize - models basics

```
// example showing synchronization of all models in a database
(async () => {
    try {
        await db.sequelize.sync();
        console.log('DB is successfully synchronized')
    } catch (error) {
        console.log(e)
    }
})();
```

- Model synchronization can perform destructive operations
  - they are not recommended for production-level software
- Synchronization should be done with the advanced concept of Migrations (keep track of changes to the database), with the help of the Sequelize CLI

Sequelize - models basics

- **Timestamps**: by default, Sequelize automatically adds the fields `createdAt` and `updatedAt` to every model, using the data type `DataTypes.DATE`
  - `createdAt`: timestamp representing the moment of creation
  - `updatedAt`: will contain the timestamp of the latest update
- This behaviour can be disabled when defining a model:

```
sequelize.define("User", {
     // ... (attributes)
}, {
     // table will NOT contain createdAt or updatedAt fields
     timestamps: false
});
```

Sequelize - models basics

- **Column declaration**: if the only thing being specified about a column is its data type, the syntax can be shortened

```
sequelize.define("tutorial", {
        title: {
            type: DataTypes.STRING
        }
});
```

→

```
sequelize.define("tutorial", {title: DataTypes.STRING});
```

column name   column data type

- Sequelize assumes that the default value of a column is NULL
  - This behavior can be changed by passing a specific `defaultValue` to the column definition

```
sequelize.define("User", {
        name: {
            type: DataTypes.STRING,
            defaultValue: "John Doe"
        }
});
```

```
sequelize.define("Meeting", {
        date: {
            type: DataTypes.DATETIME,
            defaultValue: Sequelize.NOW
        }
});
```

Sequelize datatypes

- Every column defined in the model must have a data type
- Import DataTypes to access a Sequelize built-in data type

```
const { DataTypes } = require("sequelize"); // Import the built-in data types
```

```
DataTypes.STRING                // VARCHAR(255)
DataTypes.STRING(1234)          // VARCHAR(1234)
DataTypes.TEXT                  // TEXT
DataTypes.TEXT('tiny')          // TINYTEXT
DataTypes.BOOLEAN               // TINYINT(1)
DataTypes.INTEGER               // INTEGER
DataTypes.BIGINT                // BIGINT
DataTypes.FLOAT                 // FLOAT
DataTypes.DOUBLE                // DOUBLE
DataTypes.DECIMAL               // DECIMAL
DataTypes.INTEGER.UNSIGNED      // UNSIGNED INT
DataTypes.DATE                  // DATETIME
```

There are other data types, covered here

Sequelize datatypes

- When defining a column, apart from specifying its type, there are a lot more options that can be used

```
const Foo = sequelize.define("Foo", {
      // automatically set the flag to true if not set (allowNull: adds NOT NULL to the column)
     flag: { type: DataTypes.BOOLEAN, allowNull: false, defaultValue: true },

     // default values for dates => current time
     myDate: { type: DataTypes.DATE, defaultValue: DataTypes.NOW },

     // set primary key as autoincrementing integer column
     identifier: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },

     // an attempt to insert a username that already exists will throw an error
     username: { type: DataTypes.TEXT, allowNull: false, unique: true }

     // create foreign key
     bar_id: {
          type: DataTypes.INTEGER,
          references: {
               model: Bar, // This is a reference to another model
               key: 'id' // This is the column name of the referenced model
    }
});
```

Sequelize validators

- Validations are checks performed in the Sequelize level, in pure JavaScript
  - ➢ They can be arbitrarily complex if you provide a **custom validator function** or can be one of the built-in validators offered by Sequelize
  - ➢ Validations are automatically run on **create**, **update** and **save**. You can also call **validate()** to manually validate an instance
  - ➢ If a validation fails, no SQL query will be sent to the database at all

```
const Foo = sequelize.define("Foo", {
      // validates username length to be between 5 and 10 characters
     username: { type: DataTypes.STRING, validate: { len: [5, 10] } },

     // checks for email format (foo@bar.com)
     email: { type: DataTypes.STRING, validate: { isEmail: true } },

     // age must be >= 18 and set up a custom error message
     age: { type: DataTypes.INT, validate: { min: { args: 18, msg: "Must be of legal age" } } },

     language: {type: DataTypes.STRING, validate: { isIn: [['en', 'fr']] } }
});
```

Sequelize: model querying

- **INSERT queries**: method create of Sequelize class `Model`

```
// Create a new user
const jane = await User.create({ firstName: "Jane", lastName: "Doe" });
console.log("Jane's auto-generated ID:", jane.id);
```

`create()` is a shorthand for building an instance of the model object and saving it into the DB

```
User.create({ username: 'alice123', isAdmin: true },
            { fields: ['username'] })
     .then( data => {
           console.log(data.username); // 'alice123'
           console.log(data.isAdmin); // false
     });
```

It is also possible to define which attributes can be set in the create method: in the example the `isAdmin` attribute is set with the default value (false)

Sequelize: model querying

- **INSERT queries**: method create of Sequelize class Model

```
User.create( req.body ) // use request body data to create a new User instance
      .then( data => {
            res.status(201).json(data.id); // ID of the new instance is retrieved in data.id
      })
      .catch(err => {
             // if model has validations and data from request does not fulfill them
            if (err.name === 'SequelizeValidationError')
                 // loop over err.errors array and get their messages
                 res.status(400).json({ msg: err.errors.map(e => e.message) });
            else
                 res.status(500).json({ msg: "Some error occurred while creating User."});
     });
```

Example of creating an instance using the HTTP request body data

Sequelize: model querying

- **SELECT queries**: method findAll of Sequelize class Model

```
// Find all users
const users = await User.findAll();
console.log(users.every(user => user instanceof User)); // true
console.log("All users:", JSON.stringify(users));
```

Read the whole table from the database:
SELECT * FROM users;

```
User.findAll({ attributes: ['username', 'age'] },
     .then( data => {
          //...
     });
```

Read only the **listed attributes**:
SELECT username, age FROM users;

```
User.findAll({ attributes: [['username','name'], 'age'] });
```

Attributes can be **renamed** using a nested array:
SELECT username AS name, age FROM users;

Sequelize: model querying

- **SELECT queries**: method findAll of Sequelize class Model

```
// Count hats column
const users = await User.findAll( {
     attributes: ['foo',
                 [sequelize.fn('COUNT', sequelize.col('hats')), 'n_hats'],
                 'bar'
                ]
     });
);
```

Use sequelize.fn to do **aggregations** (when using aggregation function, you must give it an alias to be able to access it from the model)
SELECT foo, COUNT(hats) AS n_hats, bar FROM ....

Sequelize: model querying

- SELECT queries with WHERE clauses: there are lots of operators to use for the where clause, available as `Symbols` from variable Op

```
// SELECT * FROM post WHERE authorId = 2
Post.findAll({
      where: {
            authorId: 2
      }
});
```

No operator (from Op) was explicitly passed, so **Sequelize assumed an equality comparison by default**.

The promise is resolved with an array of Model instances if the query succeeds

```
// SELECT * FROM post WHERE authorId = 2
AND status = 'active';
Post.findAll({
      where: {
            authorId: 12
            status: 'active'
      }
});
```

same as

```
const { Op } = require("sequelize");
Post.findAll({
      where: {
            [Op.and]: [
                  { authorId: {[Op.eq]: 12} },
                  { status: {[Op.eq]: 'active'}
            ]
      }
});
```

In multiple checks, if no operator (from Op) is explicitly passed, **Sequelize infers that the caller wanted an AND**

Check here for more examples!

Sequelize: model querying

- SELECT queries provided PRIMARY KEY: method findByPk

```
// SELECT * FROM project WHERE id = 123
const project = await Project.findByPk(123);
if (project === null) {
     console.log('Not found!');
} else {
     console.log(project instanceof Project); // true
}
```

The promise is resolved with one model instance if the query succeeds; otherwise returns null

- Method findOne: obtains the first entry it finds (that fulfills the optional query options, if provided)

```
const project = await Project.findOne({ where: { title: 'My Title' } });
if (project === null) {
     console.log('Not found!');
} else {
     console.log(project instanceof Project); // true
     console.log(project.title); // 'My Title'
}
```

Returns the first instance found, or null if none can be found

Sequelize: model querying

- Method findOrCreate: create an entry in the table unless it can find one fulfilling the query options
  - In both cases, it will return an instance (either the found instance or the created instance) and a boolean indicating whether that instance was created or already existed

```
const [user, created] = await User.findOrCreate({
  where: { username: 'sdepold' },
  defaults: { job: 'Technical Lead' }
});

console.log(user.username); // 'sdepold'

// The boolean indicating whether this instance was just created
if (created) {
  console.log(user.job); // This will certainly be 'Technical Lead'
}
```

The **where** option is considered for finding the entry, and the **defaults** option is used to define what must be created in case nothing was found.

If the defaults do not contain values for every column, Sequelize will take the values given in where

Sequelize: model querying

- Method findAndCountAll: combines findAll and count
  - useful when dealing with queries related to pagination where you want to retrieve data with a **limit** and **offset** but also need to know the total number of records that match the query

```
const { count, rows } = await Project.findAndCountAll({
     where: {
          title: { [Op.like]: 'foo%' }
  },
  offset: 10,
  limit: 2
});
// the total number records matching the query
console.log(count);
// the obtained records (array of objects): rows.length = 2
console.log(rows);
```

In the example, `result.rows` will contain rows 11 and 12, while `result.count` will return the total number of rows that matched the query

Sequelize: model querying

- UPDATE queries: method update

```
// Change everyone without a last name to "Doe"
await User.update({ lastName: "Doe" }, {
     where: {
          lastName: null
     }
});
```

The promise returns an array with the number of actual **affected rows** (if no entry is found on DB or no changes were made, the return value is [0])

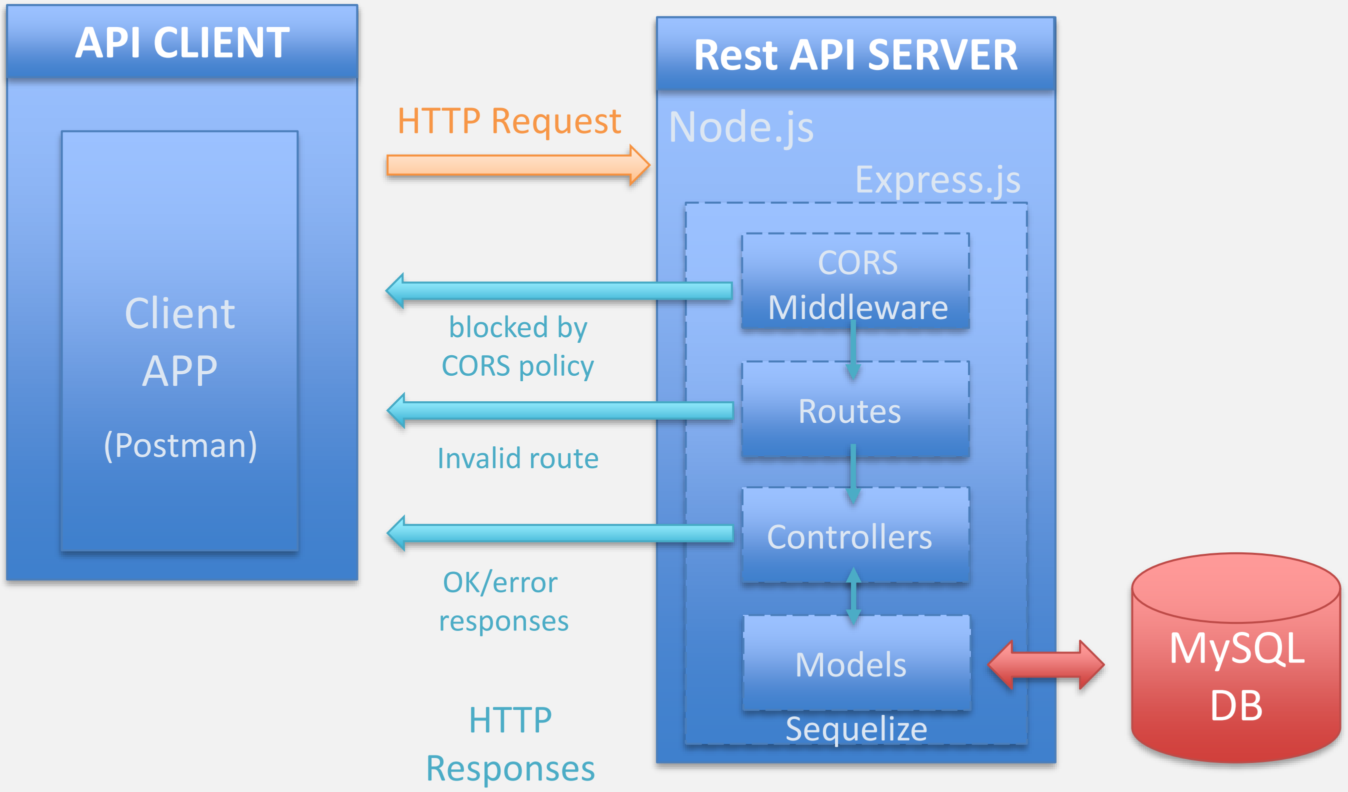
- DELETE queries: method destroy

```
// Delete everyone named "Jane"
User.destroy({ where: {firstName: "Jane" } })
     .then (num => {
          if (num == 0)
               console.log("No Janes in DB");
     });
```

The promise returns the number of destroyed rows

EXERCISE: using Sequelize in a REST API

- System architecture



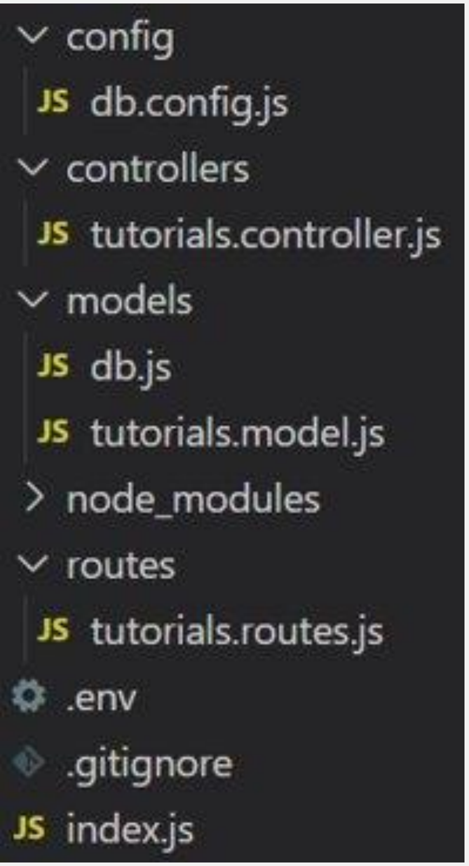
EXERCISE

- API routes

Verb	URI	Description
GET	tutorials[?title={txt}]	Gets the list of all tutorials and their details. **Optionally**, the tutorials can be filtered by title (obtains the tutorials whose title contains 'txt')
GET	tutorials/{id}	Gets details on a particular tutorial. Return 404 error if tutorial does not exist.
POST	tutorials	Creates a new tutorial with the details provided in the request body. Response contains the URI for the newly created resource. Return 400 error if insuficiente body data.
PUT	tutorials/{id}	Modifies a particular tutorial. Response contains the URI for the updated resource. Return 404 error if tutorial does not exist. Return 400 error if insufficient body data.
DELETE	tutorials/{id}	Delete a particular tutorial. Return 404 error if tutorial does not exist.

EXERCISE

- Directory structure: let's structure the project in the following manner, so that the files are laid out logically in folders



```
config
  db.config.js
controllers
  tutorials.controller.js
models
  db.js
  tutorials.model.js
node_modules
routes
  tutorials.routes.js
.env
.gitignore
index.js
```

config folder: project configurations, like database credentials in db.config.js

controllers folder: responsible for request data validation and for sending the API responses to clients

models folder: house the definition of the tutorials table in DB; file db.js is used to talk with the MySQL database

routes folder: define routes handlers

index.js file: sets up an Express web server

.env file: stores all the environment variables (API hostname, API port, DB credentials,…)

.gitignore file: list of files or folders to be ignored when saving your project into your local GIT repository (like node_modules folder or .env file)

EXERCISE: using Sequelize in a REST API

- Create a directory for the API Rest, and build the package.json file with the necessary dependencies

  `cors` `dotenv` `express` `sequelize` `mysql2`

  loads **environment variables** from a `.env` file into `process.env`

- Set up environment variables on .env file
  - Like that, configuration variables and DB credentials are not hardcoded and would not be saved into your code repository

File *.env* under the project root folder

```
NODE_ENV=development
# Server configuration
PORT=3000
HOST='127.0.0.1'
# Database connection information
DB_HOST: 'pw2.joaoferreira.eu'
DB_USER: 'teresaterroso_pw2_user'
DB_PASSWORD: 'AO6?n+JhTKhY'
DB_NAME: 'teresaterroso_pw2'
```

EXERCISE

- Configure the DB connection parameters

File *db.config.js* on *config* folder

```
const config = {
     // read DB credencials from environment variables
     HOST: process.env.DB_HOST ,
     USER: process.env.DB_USER ,
     PASSWORD: process.env.DB_PASSWORD ,
     DB: process.env.DB_NAME ,
     dialect: "mysql",
     // pool is optional, it will be used for Sequelize connection pool configuration
     pool: {
          max: 5,    //maximum number of connections in pool
          min: 0,    //minimum number of connections in pool
          acquire: 30000,//maximum time (ms), that pool will try to get connection before throwing error
          idle: 10000 //maximum time (ms) that a connection can be idle before being released
     }
};

module.exports = config;
```

EXERCISE

- Define the **tutorial model**

File *tutorials.model.js* on *models* folder

```
module.exports = (sequelize, DataTypes) => {
    const Tutorial = sequelize.define("Tutorial", {
        title: {
            type: DataTypes.STRING,
            allowNull: false,
            validate: { notNull: { msg: "Title can not be empty!" } }
        },
        description: {
            type: DataTypes.STRING
        },
        published: {
            type: DataTypes.BOOLEAN,
            defaultValue: 0,
            validate: {
                isBoolean: function (val) { // custom validation function
                    if (typeof (val) != 'boolean')
                        throw new Error('Published must contain a boolean value!');
                }
            }
        }
    }, {
         timestamps: false
    });
    return Tutorial;
```

Defines the model name: **tutorial**

Defines the model attributes (and their data types and validations)

Disables the Sequelize default behavior of automatically adding fields createdAt and updatedAt to every model

EXERCISE

- Create a database using Sequelize

File *index.js* on *models* folder

```
const dbConfig = require('../config/db.config.js');
//export classes Sequelize and Datatypes
const { Sequelize, DataTypes } = require('sequelize');

const sequelize = new Sequelize(dbConfig.DB, dbConfig.USER, dbConfig.PASSWORD, {
    host: dbConfig.HOST, dialect: dbConfig.dialect,
    pool: {
        max: dbConfig.pool.max, min: dbConfig.pool.min,
        acquire: dbConfig.pool.acquire, idle: dbConfig.pool.idle
    }
});

// OPTIONAL: test the connection
(async () => {
    try {
        await sequelize.authenticate;
        console.log('Connection has been established successfully.');
    } catch (err) {
        console.error('Unable to connect to the database:', err);
    }
})();

...
```

EXERCISE

- Create a database using Sequelize

File *index.js* on *models* folder

```
...

const db = {}; //object to be exported
db.sequelize = sequelize; //save the Sequelize instance (actual connection pool)

//save the TUTORIAL model (and add here any other models defined within the API)
db.tutorial = require("./tutorials.model.js")(sequelize, DataTypes);


// OPTIONAL: synchronize the DB with the sequelize model
(async () => {
    try {
        await db.sequelize.sync();
        console.log('DB is successfully synchronized')
    } catch (error) {
        console.log(e)
    }
})();

module.exports = db; //export the db object with the sequelize instance and tutorial model
```

EXERCISE

- Define the routes

File *tutorials.routes.js* on *routes* folder

```
const express = require('express');
let router = express.Router();
const tutorialController = require('../controllers/tutorials.controller');

// middleware for all routes related with tutorials
router.use((req, res, next) => {
    const start = Date.now();
    res.on("finish", () => { // finish event is emitted once the response is sent to the client
        const diffSeconds = (Date.now() - start) / 1000; // figure out how many seconds elapsed
        console.log(`${req.method} ${req.originalUrl} completed in ${diffSeconds} seconds`);
    });
    next()
})

router.route('/')
    // .get(tutorialController.findAll);
    .post(tutorialController.create);
//…  TO BE COMPLETED

//send a predefined error message for invalid routes on TUTORIALS
router.all('*', function (req, res) {
    res.status(404).json({ message: 'TUTORIALS: what???' });
})

// EXPORT ROUTES (required by APP)
module.exports = router;
```

EXERCISE

- Implement the controller functions
  - ➢ Example of the `create` function, for creating a new tutorial

File *tutorials.controller.js* on *controllers* folder

```
const db = require("../models/index.js");
const Tutorial = db.tutorial;

const { ValidationError } = require('sequelize'); //necessary for model validations using sequelize

exports.create = (req, res) => {
    Tutorial.create(req.body) // Save Tutorial in the DB (IF request body data is validated by Sequelize)
        .then(data => {
            res.status(201).json({ success:true, msg:"New tutorial created", URL:`/tutorials/${data.id}`});
        })
        .catch(err => {
            if (err instanceof ValidationError) // Tutorial model as validations for title and published
                res.status(400).json({ success:false, msg: err.errors.map(e => e.message) });
            else
                res.status(500).json({
                    message: err.message || "Some error occurred while creating the Tutorial."
                });
        });
};
```

Using *then… catch*

EXERCISE

- Implement the controller functions
  - Example of the `create` function, for creating a new tutorial

File *tutorials.controller.js* on *controllers* folder

```
const db = require("../models/index.js");
const Tutorial = db.tutorial;

const { ValidationError } = require('sequelize'); //necessary for model validations using sequelize

exports.create = async (req, res) => {
    // Save Tutorial in the database (IF request body data is validated by Sequelize)
    try {
        const newTut = await Tutorial.create(req.body);
        res.status(201).json({success:true, msg:"New tutorial created", URL:`/tutorials/${newTut.id}`});
    } catch (err) {
        if (err instanceof ValidationError)
            res.status(400).json({ success:false, msg: err.errors.map(e => e.message) });
        else
            res.status(500).json({
                message: err.message || "Some error occurred while creating the Tutorial."
            });
    }};
};
```

Using *async ... await*

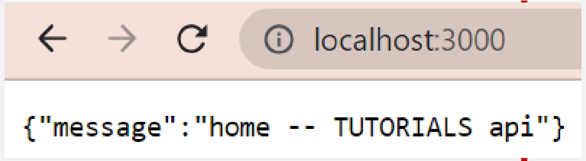
EXERCISE

- Main file

File *index.js* in the root project folder

```
require('dotenv').config();          // read environment variables from .env file
const express = require('express');
const cors = require('cors');        // middleware to enable CORS (Cross-Origin Resource Sharing)

const app = express();
const port = process.env.PORT ;  // use environment variables
const host = process.env.HOST ;

app.use(cors()); //enable ALL CORS requests (client requests from other domain)
app.use(express.json()); //enable parsing JSON body data

// root route -- /api/
app.get('/', function (req, res) {
    res.status(200).json({ message: 'home -- TUTORIALS api' });
});

// routing middleware for resource TUTORIALS
app.use('/tutorials', require('./routes/tutorials.routes.js'))

// handle invalid routes
app.get('*', function (req, res) {
     res.status(404).json({ message: 'WHAT???' });
})
app.listen(port, host, () => console.log(`App listening at http://${host}:${port}/`));
```

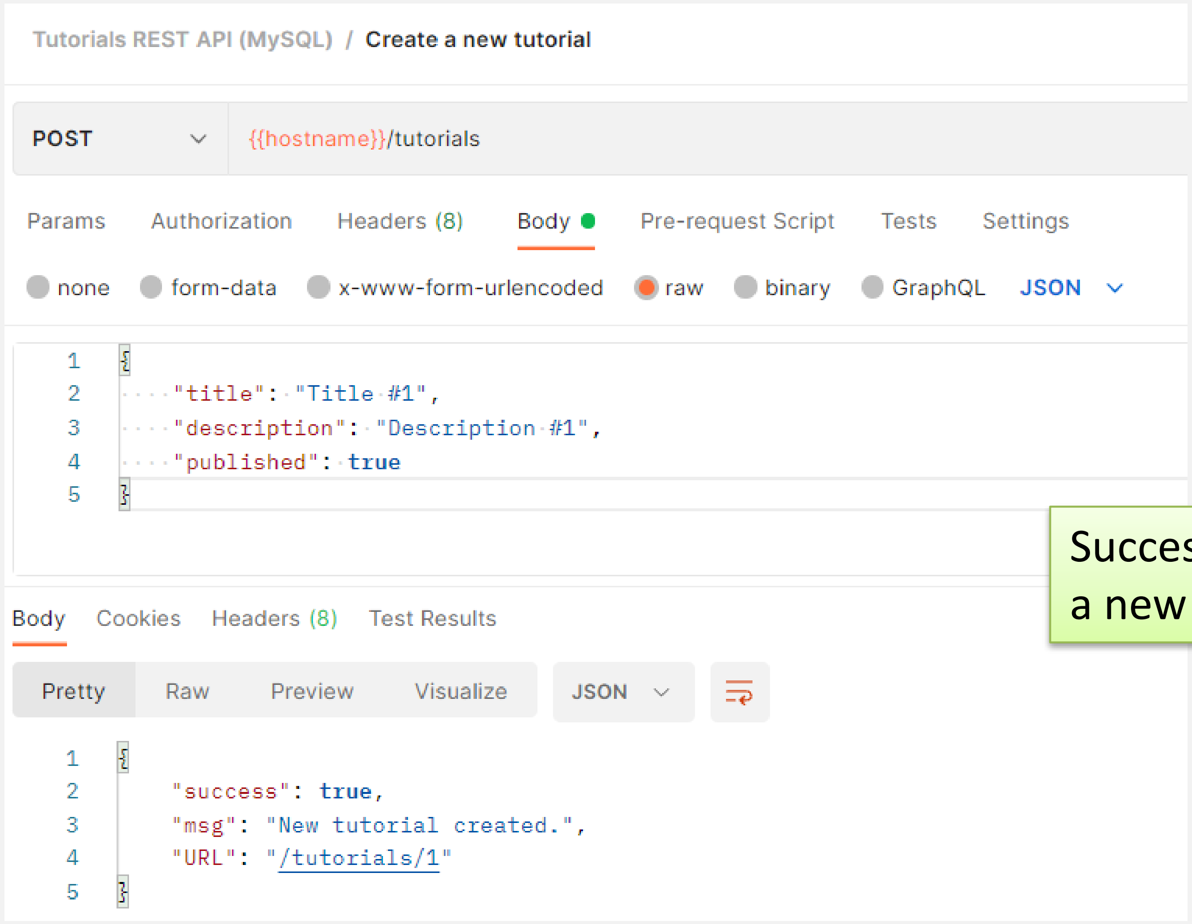
Run the app with command: **node index.js** or **nodemon index.js**
Open your browser, enter the url http://localhost:3000/, you will see



```
{"message":"home -- TUTORIALS api"}
```

EXERCISE

- Test it by creating a new tutorial, using Postman



Tutorials REST API (MySQL) / Create a new tutorial

POST {{hostname}}/tutorials

Body: [✓ raw] JSON

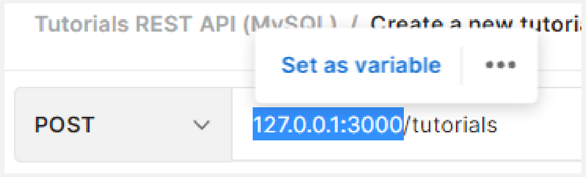
```
{
    "title": "Title #1",
    "description": "Description #1",
    "published": true
}
```

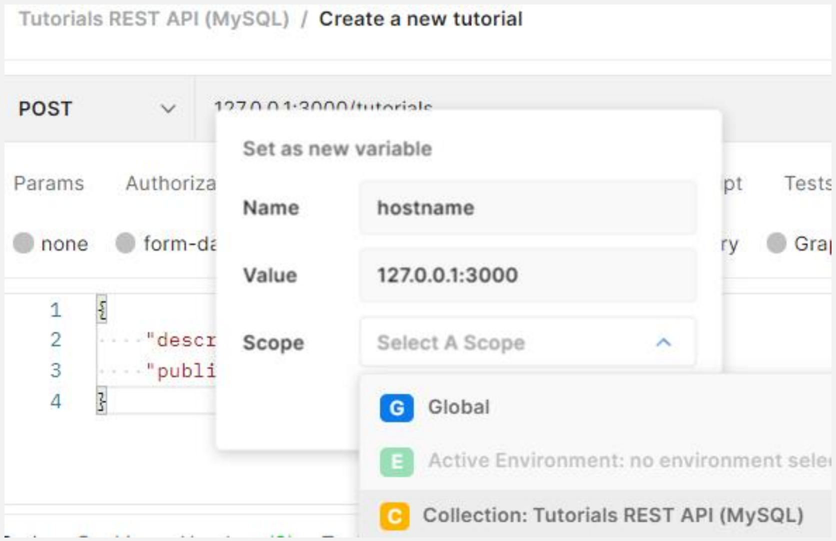
```
{
    "success": true,
    "msg": "New tutorial created.",
    "URL": "/tutorials/1"
}
```

Successfull creation of a new tutorial

EXERCISE

- Postman variables




Tutorials REST API (MySQL) / Create a new tutorial

POST 127.0.0.1:3000/tutorials

Set as new variable

Name	hostname
Value	127.0.0.1:3000
Scope	Select A Scope

Params
Authoriza
Tests
none
form-da
Gra
Global
Active Environment: no environment sele
Collection: Tutorials REST API (MySQL)

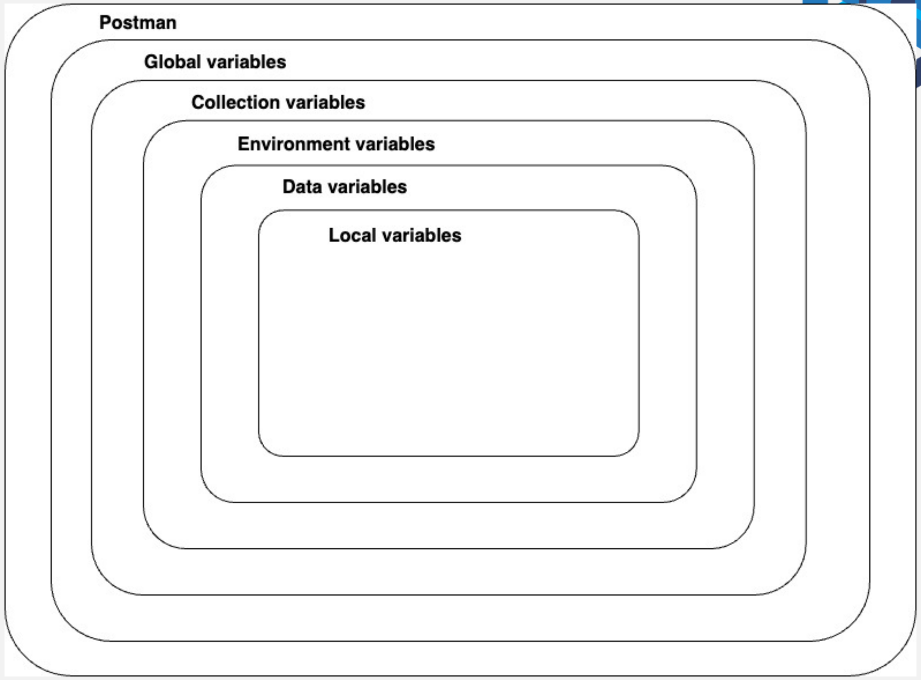
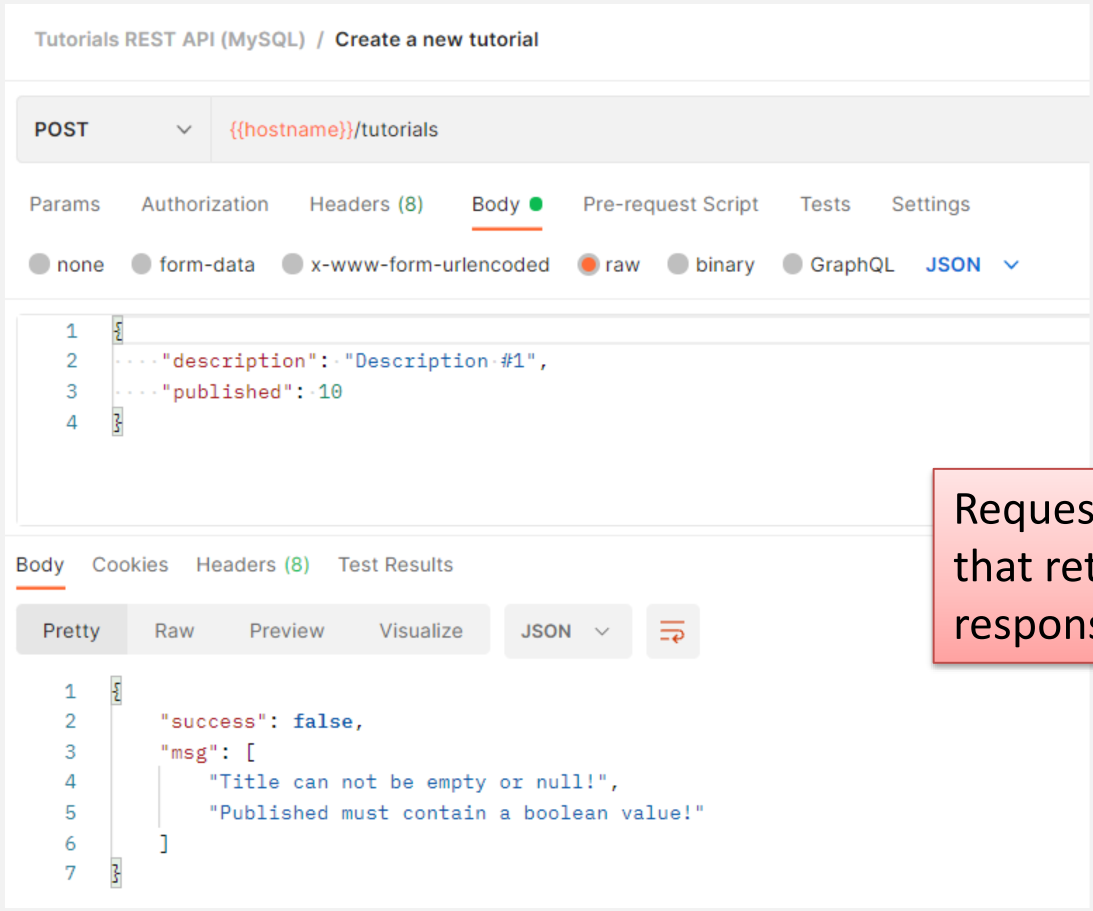


Image source: https://learning.postman.com/docs/sending-requests/variables/

EXERCISE

- Test it by creating a new tutorial, using Postman



Tutorials REST API (MySQL) / Create a new tutorial

POST {{hostname}}/tutorials

```
{
    "description": "Description #1",
    "published": 10
}
```

```
{
    "success": false,
    "msg": [
        "Title can not be empty or null!",
        "Published must contain a boolean value!"
    ]
}
```

Request to create a new tutorial that returns 400 Bad Request response

EXERCISE

- Complete the REST API for tutorials, with the routes to:
  - Read all tutorials
  - Read all published tutorials
  - Filter tutorials by title
  - Read just one tutorial, providing its ID
  - Update a tutorial, providing its ID
  - Delete a tutorial, providing its ID

EXERCISE

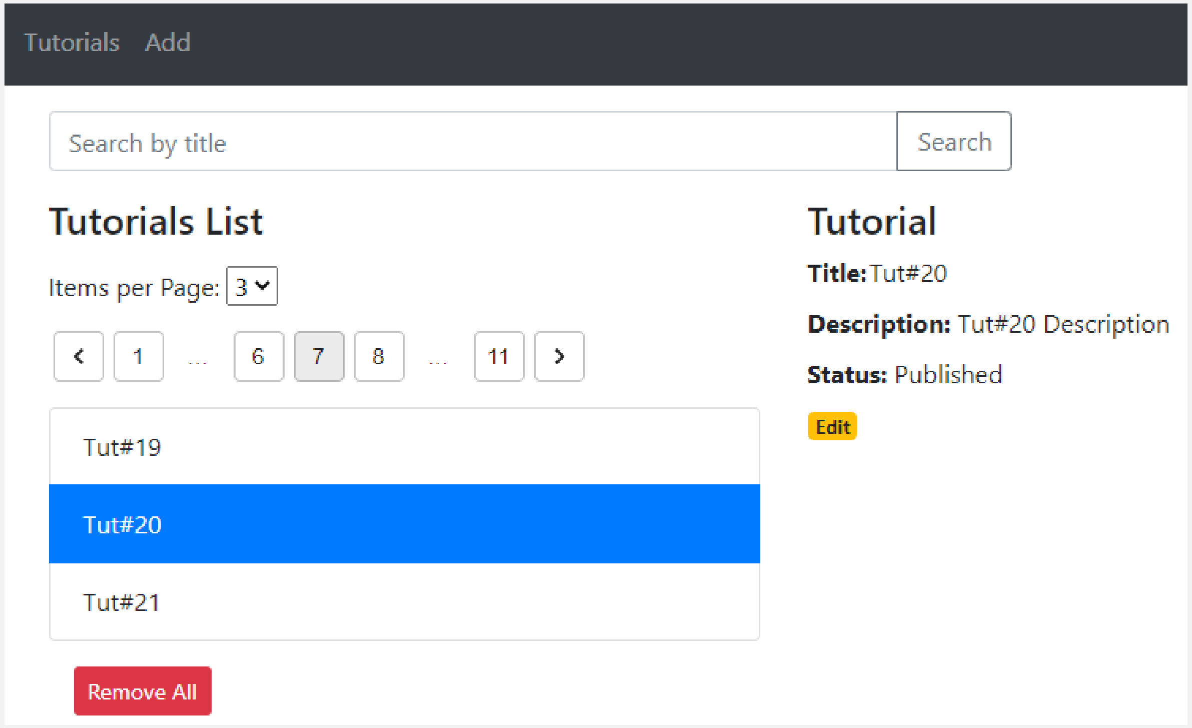
- Add server-side pagination feature to your API, for routes that retrieve a list of tutorials
    - ➢ By default, the page size is 3 (page index starts at 0)
    - ➢ The structure of the result should be as the image on the right

```
{
    "totalItems": 8,
    "tutorials": [...],
    "totalPages": 3,
    "currentPage": 1
}
```

Verb	URI	Description
GET	tutorials	Gets the first page (with the default value of 3 tutorials)
GET	tutorials?page=1	Gets the second page (with the default value of 3 tutorials)
GET	tutorials?page=1&size=5	Gets the second page (with 5 tutorials)
GET	tutorials/published?page=2	Gets the third page of published tutorials
GET	tutorials?title=txt&page=1& size=5	Gets the second page (with 5 tutorials) of tutorials which title contains 'txt'

EXERCISE

- Add server-side pagination feature to your API
  - ➢ With this feature, it is possible to implement a client app looking like this:



Tutorials List

Tut#19

Tut#20

Tut#21

Tutorial

Title: Tut#20

Description: Tut#20 Description

Status: Published