Angular + Node.js Express + MySQL - CRUD App

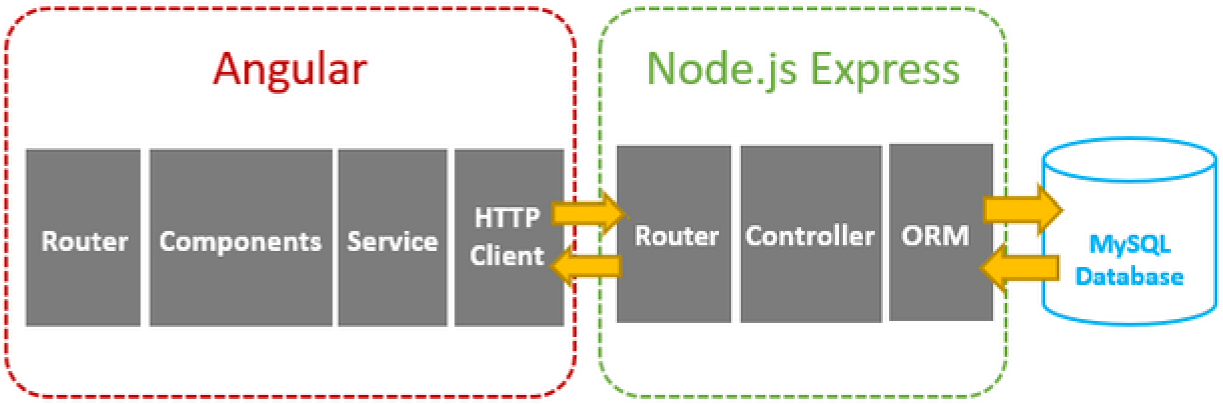
Podczas zajęć spróbujemy stworzyć aplikację CRUD typu full-stack (Angular + Node.js + Express + MySQL). Serwer (back-end) używa Node.js + Express dla REST API, strona klienta (front-end) będzie aplikacją opartą o Angular oraz klienta HttpClient.

W aplikacji full-stack, którą stworzymy będzie:

- Tabela Tutorial z polami id, tytul, opis, opublikowany.
- Użytkownik będzie mógł tworzyć (create), wybrać (retrieve), zaktualizować (update), usunąć (delete) Tutoriale
- Dodatkową możliwośią będzie wyszukanie Tutorialu według tytułu.

Architektura Full-stack Angular & Node Express

Zbudujemy aplikację według następującej struktury:



– Node.js Express exports REST APIs & interacts with MySQL Database using Sequelize ORM.

– Angular Client sends HTTP Requests and retrieves HTTP Responses using *HTTPClient*, consume data on the components. Angular Router is used for navigating to pages.

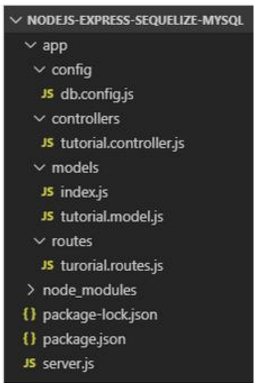
Node.js Express Back-end

Overview

These are APIs that Node.js Express App will export:

Methods	Urls	Actions
GET	api/tutorials	get all Tutorials
GET	api/tutorials/:id	get Tutorial by `id`
POST	api/tutorials	add new Tutorial
PUT	api/tutorials/:id	update Tutorial by `id`
DELETE	api/tutorials/:id	remove Tutorial by `id`
DELETE	api/tutorials	remove all Tutorials
GET	api/tutorials?title=[kw]	find all Tutorials which tytul contains `'kw'`

Project Structure



```
NODEJS-EXPRESS-SEQUELIZE-MYSQL
  app
    config
      db.config.js
    controllers
      tutorial.controller.js
    models
      index.js
      tutorial.model.js
    routes
      turorial.routes.js
  node_modules
  package-lock.json
  package.json
  server.js
```

– *db.config.js* exports configuring parameters for MySQL connection & Sequelize.

– **Express** web server in *server.js* where we configure CORS, initialize & run Express REST APIs.

– Next, we add configuration for MySQL database in **models**/*index.js*, create **Sequelize** data model in **models**/*tutorial.model.js*.

– Tutorial controller in **controllers**.

– Routes for handling all CRUD operations (including custom finder) in *tutorial.routes.js*.

Create Node.js App

First, we create a folder:

```
mkdir app_sequalize_mysql
cd app_sequalize_mysql
```

Next, we initialize the Node.js App with a *package.json* file:

```
npm init

name: (nodejs-express-sequelize-mysql)
version: (1.0.0)
description: Node.js Rest Apis with Express, Sequelize & MySQL.
entry point: (index.js) server.js
test command:
git repository:
keywords: nodejs, express, sequelize, mysql, rest, api
author: bezkoder
license: (ISC)

Is this ok? (yes) yes
```

We need to install necessary modules: `express`, `sequelize` and `mysql2`.

Run the command:

```
npm install express sequelize mysql2 cors --save
```

Setup Express web server

In the root folder, let’s create a new *server.js* file:

```
const express = require("express");
const cors = require("cors");

const app = express();

var corsOptions = {
  origin: "http://localhost:3001"
};

app.use(cors(corsOptions));

// parse requests of content-type - application/json
app.use(express.json());

// parse requests of content-type - application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }));

// simple route
app.get("/", (req, res) => {
  res.json({ message: "To ja SERWER!!!." });
});

// set port, listen for requests
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is running on port ${PORT}.`);
});
```

What we do are:

– import `express`, and `cors` modules:

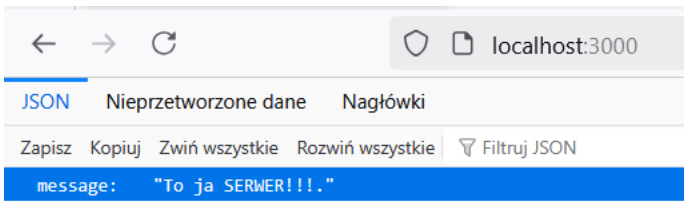
- Express is for building the Rest apis
- cors provides Express middleware to enable CORS (Cross-Origin Resource Sharing) with various options.

– create an Express app, then add body-parser (`json` and `urlencoded`) and `cors` middlewares using `app.use()` method. Notice that we set origin: `http://localhost:3001`.

– define a GET route which is simple for test.

– listen on port 3000 for incoming requests.

Now let’s run the app with command: `node server.js`. Open your browser with url http://localhost:3000/, you will see:



```
message:   "To ja SERWER!!!."
```

Yeah, the first step is done. We’re gonna work with Sequelize in the next section.

Configure MySQL database & Sequelize

In the *app* folder, we create a separate *config* folder for configuration with *db.config.js* file like this:

```
module.exports = {
    HOST: "localhost",
    USER: "root",
    PASSWORD: "",
    DB: "baza_tutorial",
    dialect: "mysql",
    pool: {
      max: 5,
      min: 0,
      acquire: 30000,
      idle: 10000
    }
  };
```

First five parameters are for MySQL connection.

`pool` is optional, it will be used for Sequelize connection pool configuration:

- `max`: maximum number of connection in pool
- `min`: minimum number of connection in pool
- `idle`: maximum time, in milliseconds, that a connection can be idle before being released
- `acquire`: maximum time, in milliseconds, that pool will try to get connection before throwing error

For more details, please visit API Reference for the Sequelize constructor.

Initialize Sequelize

We're gonna initialize Sequelize in **app/models** folder that will contain model in the next step.

Now create **app/models**/*index.js* with the following code:

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

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

const db = {};
db.Sequelize = Sequelize;
db.sequelize = sequelize;
db.tutorials = require("./tutorial.model.js")(sequelize, Sequelize);
module.exports = db;
```

Don't forget to call `sync()` method in *server.js*:

```
...
const app = express();
app.use(...);

const db = require("./app/models");
db.sequelize.sync()
  .then(() => {
    console.log("Baza zsynchronizowana");
  })
  .catch((err) => {
    console.log("Błąd synchronizacji bazy: " + err.message);
  });

...
```

In development, you may need to drop existing tables and re-sync database. Just use `force: true` as following code:

```
db.sequelize.sync({ force: true }).then(() => {
  console.log("Drop and re-sync db.");
});
```

Define the Sequelize Model

In *models* folder, create *tutorial.model.js* file like this:

```
module.exports = (sequelize, Sequelize) => {
    const Tutorial = sequelize.define("tutorial", {
      tytul: {
        type: Sequelize.STRING
      },
      opis: {
        type: Sequelize.STRING
      },
      opublikowany: {
        type: Sequelize.BOOLEAN
      }
    });

    return Tutorial;
  };
```

This Sequelize Model represents **tutorials** table in MySQL database. These columns will be generated automatically: *id*, *tytul*, *opis*, *opublikowany*, *createdAt*, *updatedAt*.

After initializing Sequelize, we don't need to write CRUD functions, Sequelize supports all of them:

- create a new Tutorial: `create(object)`
- find a Tutorial by id: `findByPk(id)`
- get all Tutorials: `findAll()`
- update a Tutorial by id: `update(data, where: { id: id })`
- remove a Tutorial: `destroy(where: { id: id })`
- remove all Tutorials: `destroy(where: {})`
- find all Tutorials by title: `findAll({ where: { tytul: ... } })`

These functions will be used in our Controller.

Create the Controller

Inside **app/controllers** folder, let’s create *tutorial.controller.js* with these CRUD functions:

- create
- findAll
- findOne
- update
- delete
- deleteAll
- findAllPublished

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

// Create and Save a new Tutorial
exports.create = (req, res) => {

};

// Retrieve all Tutorials from the database.
exports.findAll = (req, res) => {

};

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

};

// Update a Tutorial by the id in the request
exports.update = (req, res) => {

};

// Delete a Tutorial with the specified id in the request
exports.delete = (req, res) => {

};

// Delete all Tutorials from the database.
exports.deleteAll = (req, res) => {

};

// Find all published Tutorials
exports.findAllPublished = (req, res) => {

};
```

Let’s implement these functions.

Create a new object

Create and Save a new Tutorial:

```
exports.create = (req, res) => {
    // Validate request
    if (!req.body.tytul) {
      res.status(400).send({
        message: "Zawartość nie może być pusta!"
      });
      return;
    }

    // Create a Tutorial
    const tutorial = {
      tytul: req.body.tytul,
      opis: req.body.opis,
      opublikowany: req.body.opublikowany ? req.body.opublikowany : false
    };

    // Save Tutorial in the database
    Tutorial.create(tutorial)
      .then(data => {
        res.send(data);
      })
      .catch(err => {
        res.status(500).send({
          message:
            err.message || "Podczas zapisywania wystąpił błąd."
        });
      });
  };
```

Retrieve objects (with condition)

Retrieve all Tutorials/ find by title from the database:

```
exports.findAll = (req, res) => {
    const title = req.query.tytul;
    var condition = tytul ? { tytul: { [Op.like]: `%${tytul}%` } } : null;

    Tutorial.findAll({ where: condition })
      .then(data => {
        res.send(data);
      })
      .catch(err => {
        res.status(500).send({
          message:
            err.message || "Podczas odczytywania wystąpił błąd."
        });
      });
  };
```

We use `req.query.tytul` to get query string from the Request and consider it as condition for `findAll()` method.

Retrieve a single object

Find a single Tutorial with an `id`:

```
exports.findOne = (req, res) => {
    const id = req.params.id;

    Tutorial.findByPk(id)
      .then(data => {
        if (data) {
          res.send(data);
        } else {
          res.status(404).send({
            message: `Nie ma Tutorialu o id=${id}.`
          });
        }
      })
      .catch(err => {
        res.status(500).send({
          message: "Błąd szukania tutorialu o id=" + id
        });
      });
  };
```

Update an object

Update a Tutorial identified by the `id` in the request:

```
exports.update = (req, res) => {
    const id = req.params.id;

    Tutorial.update(req.body, {
      where: { id: id }
    })
      .then(num => {
        if (num == 1) {
          res.send({
            message: "Tutorial został zmieniony."
          });
        } else {
          res.send({
            message: `Nie mogę zmienić Tutorialu o id=${id}. Być może nie ma takiego
Tutorialu lub req.body jest puste!`
          });
        }
      })
      .catch(err => {
        res.status(500).send({
          message: "Błąd zmiany Tutorialu o id=" + id
        });
      });
  };
```

Delete an object

Delete a Tutorial with the specified `id`:

```
exports.delete = (req, res) => {
    const id = req.params.id;

    Tutorial.destroy({
      where: { id: id }
    })
      .then(num => {
        if (num == 1) {
          res.send({
            message: " Tutorial został usunięty!"
          });
        } else {
          res.send({
            message: `Nie mogę usunąć Tutorial o id=${id}. Może nie ma takiego Tutorialu!`
          });
        }
      })
      .catch(err => {
        res.status(500).send({
          message: "Nie mogę usunąć Tutorialu o id=" + id
        });
      });
  };
```

Delete all objects

Delete all Tutorials from the database:

```
exports.deleteAll = (req, res) => {
    Tutorial.destroy({
      where: {},
      truncate: false
    })
      .then(nums => {
        res.send({ message: `Tutoriale ${nums} zostały usunięte!` });
      })
      .catch(err => {
        res.status(500).send({
          message:
            err.message || "Podczas usuwania wystąpiły błędy."
        });
      });
  };
```

Find all objects by condition

Find all Tutorials with `published = true`:

```
exports.findAllPublished = (req, res) => {
    Tutorial.findAll({ where: { opublikowany: true } })
      .then(data => {
        res.send(data);
      })
      .catch(err => {
        res.status(500).send({
          message:
            err.message || "Podczas znajdowania Tutoriali wystąpiły błędy."
        });
      });
  };
```

Define Routes

When a client sends request for an endpoint using HTTP request (GET, POST, PUT, DELETE), we need to determine how the server will response by setting up the routes.

These are our routes:

- `/api/tutorials`: GET, POST, DELETE
- `/api/tutorials/:id`: GET, PUT, DELETE
- `/api/tutorials/opublikowany`: GET

Create a *turorial.routes.js* inside *app/routes* folder with content like this:

```
module.exports = app => {
    const tutorials = require("../controllers/tutorial.controller.js");

    var router = require("express").Router();

    // Create a new Tutorial
    router.post("/", tutorials.create);

    // Retrieve all Tutorials
    router.get("/", tutorials.findAll);

    // Retrieve all published Tutorials
    router.get("/opublikowany", tutorials.findAllPublished);

    // Retrieve a single Tutorial with id
    router.get("/:id", tutorials.findOne);

    // Update a Tutorial with id
    router.put("/:id", tutorials.update);

    // Delete a Tutorial with id
    router.delete("/:id", tutorials.delete);

    // Delete all Tutorials
    router.delete("/", tutorials.deleteAll);

    app.use('/api/tutorials', router);
  };
```

You can see that we use a controller from `/controllers/tutorial.controller.js`.

We also need to include routes in *server.js* (right before `app.listen()`):

```
...

require("./app/routes/turorial.routes")(app);


// set port, listen for requests
const PORT = ...;
app.listen(...);
```

Test the APIs

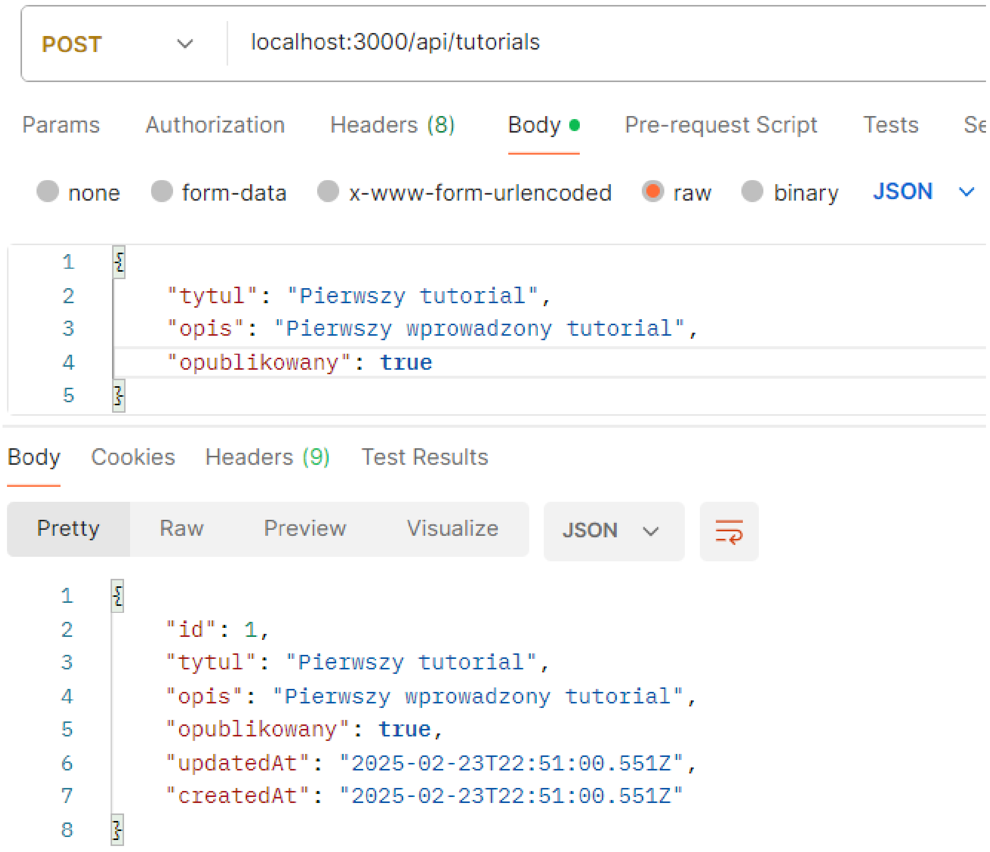
Run our Node.js application with command: `node server.js`.

The console shows:

```
Server is running on port 3000.
Executing (default): DROP TABLE IF EXISTS `tutorials`;
Executing (default): CREATE TABLE IF NOT EXISTS `tutorials` (`id` INTEGER NOT NULL auto_increment
, `title` VARCHAR(255), `description` VARCHAR(255), `published` TINYINT(1), `createdAt` DATETIME
NOT NULL, `updatedAt` DATETIME NOT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB;
Executing (default): SHOW INDEX FROM `tutorials`
Drop and re-sync db.
```

Using Postman, we're gonna test all the Apis above.

1. Create a new Tutorial using `POST /tutorials` Api

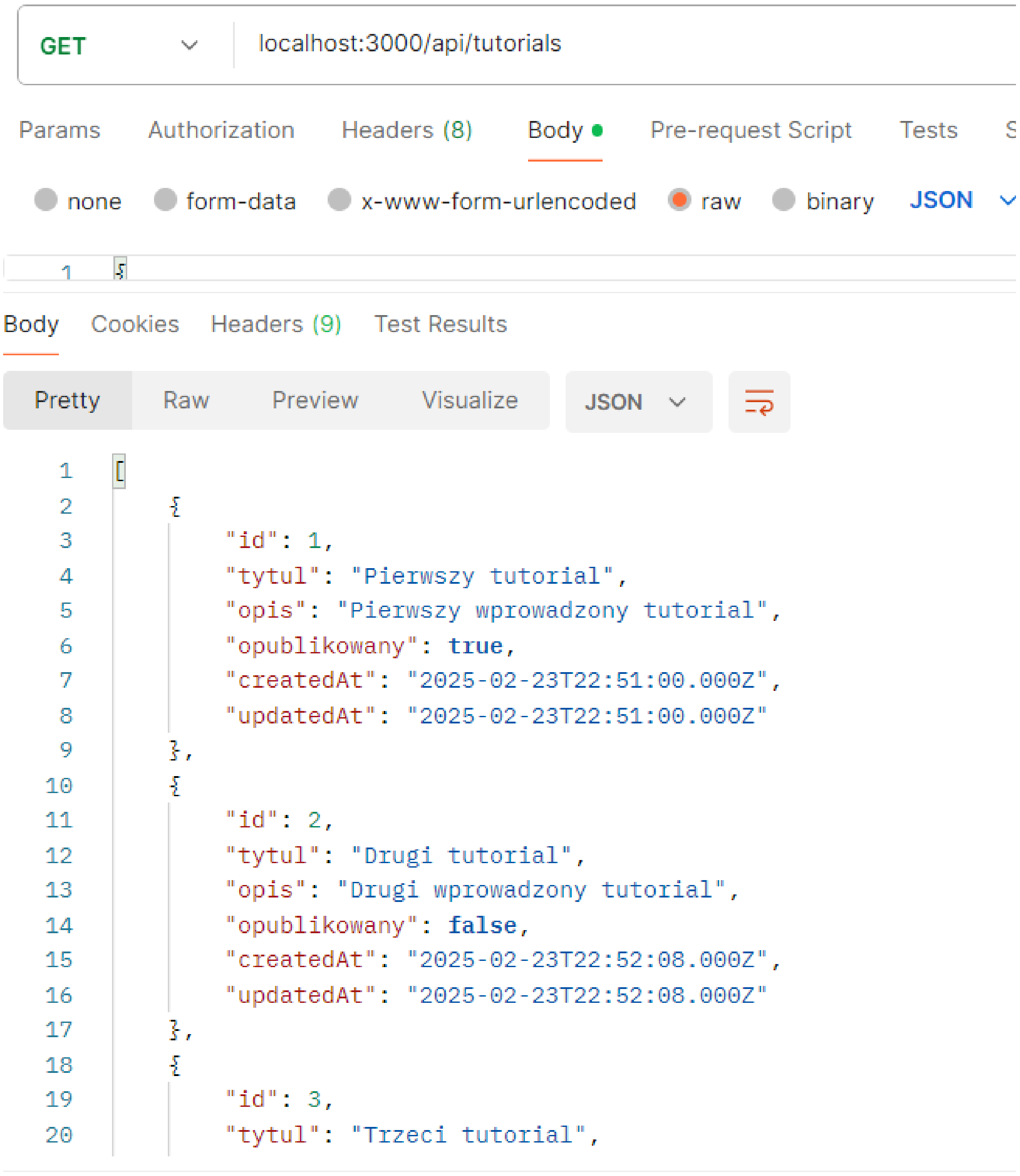


localhost:3000/api/tutorials

```
{
    "tytul": "Pierwszy tutorial",
    "opis": "Pierwszy wprowadzony tutorial",
    "opublikowany": true
}
```

```
{
    "id": 1,
    "tytul": "Pierwszy tutorial",
    "opis": "Pierwszy wprowadzony tutorial",
    "opublikowany": true,
    "updatedAt": "2025-02-23T22:51:00.551Z",
    "createdAt": "2025-02-23T22:51:00.551Z"
}
```

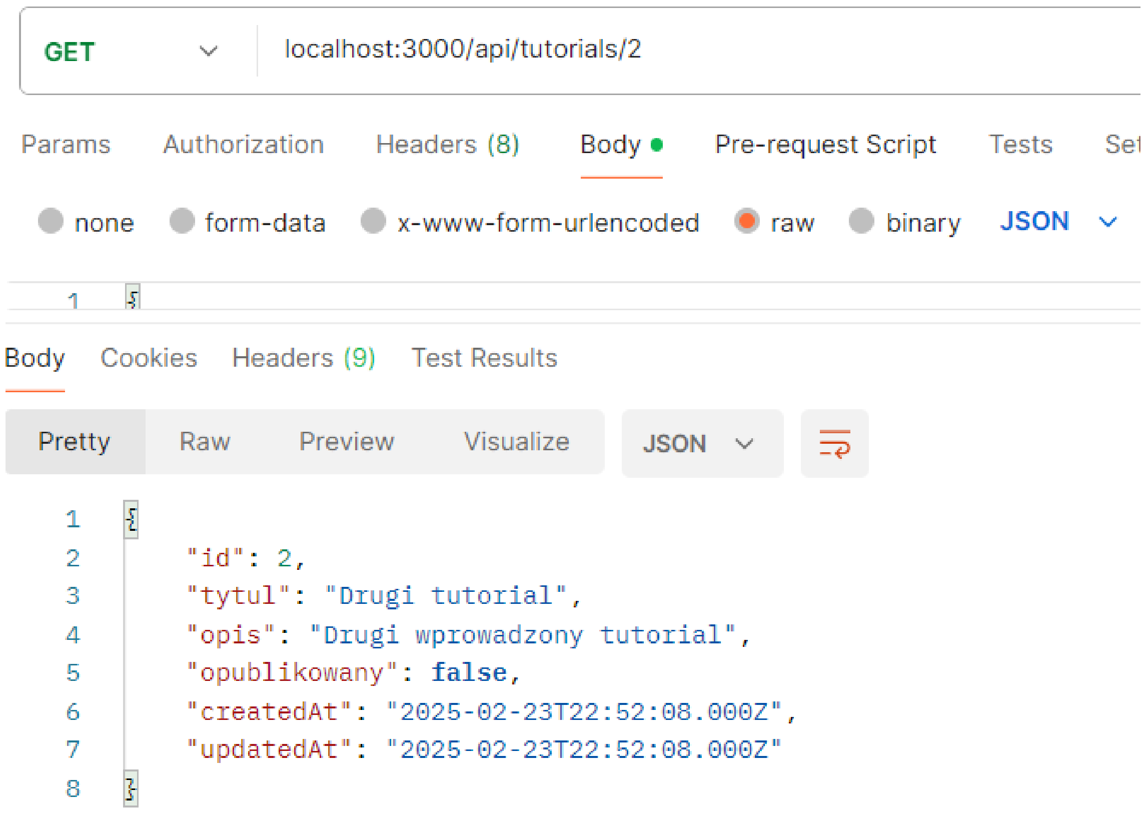
2. Retrieve all Tutorials using GET /tutorials Api



```
{
```

```
[
    {
        "id": 1,
        "tytul": "Pierwszy tutorial",
        "opis": "Pierwszy wprowadzony tutorial",
        "opublikowany": true,
        "createdAt": "2025-02-23T22:51:00.000Z",
        "updatedAt": "2025-02-23T22:51:00.000Z"
    },
    {
        "id": 2,
        "tytul": "Drugi tutorial",
        "opis": "Drugi wprowadzony tutorial",
        "opublikowany": false,
        "createdAt": "2025-02-23T22:52:08.000Z",
        "updatedAt": "2025-02-23T22:52:08.000Z"
    },
    {
        "id": 3,
        "tytul": "Trzeci tutorial",
```

3. **Retrieve a single Tutorial by id using `GET /tutorials/:id` Api**



GET localhost:3000/api/tutorials/2

Params | Authorization | Headers (8) | Body ● | Pre-request Script | Tests | Se

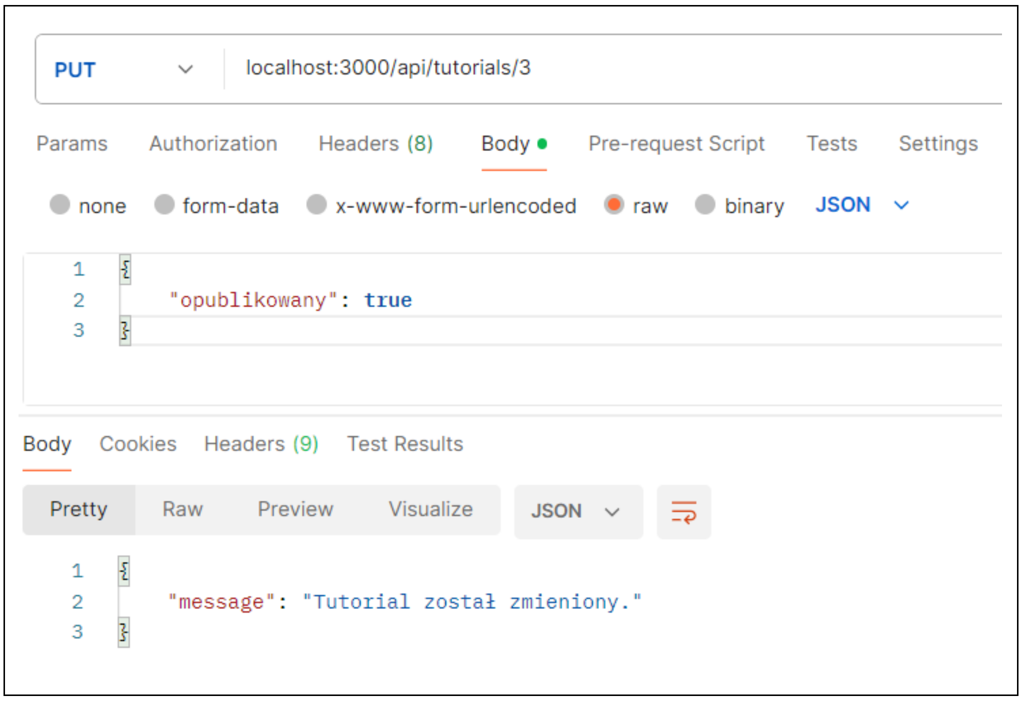
none | form-data | x-www-form-urlencoded | raw | binary | JSON

Body | Cookies | Headers (9) | Test Results

Pretty | Raw | Preview | Visualize | JSON

```
{
    "id": 2,
    "tytul": "Drugi tutorial",
    "opis": "Drugi wprowadzony tutorial",
    "opublikowany": false,
    "createdAt": "2025-02-23T22:52:08.000Z",
    "updatedAt": "2025-02-23T22:52:08.000Z"
}
```

4. **Update a Tutorial using `PUT /tutorials/:id` Api**



PUT localhost:3000/api/tutorials/3

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

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

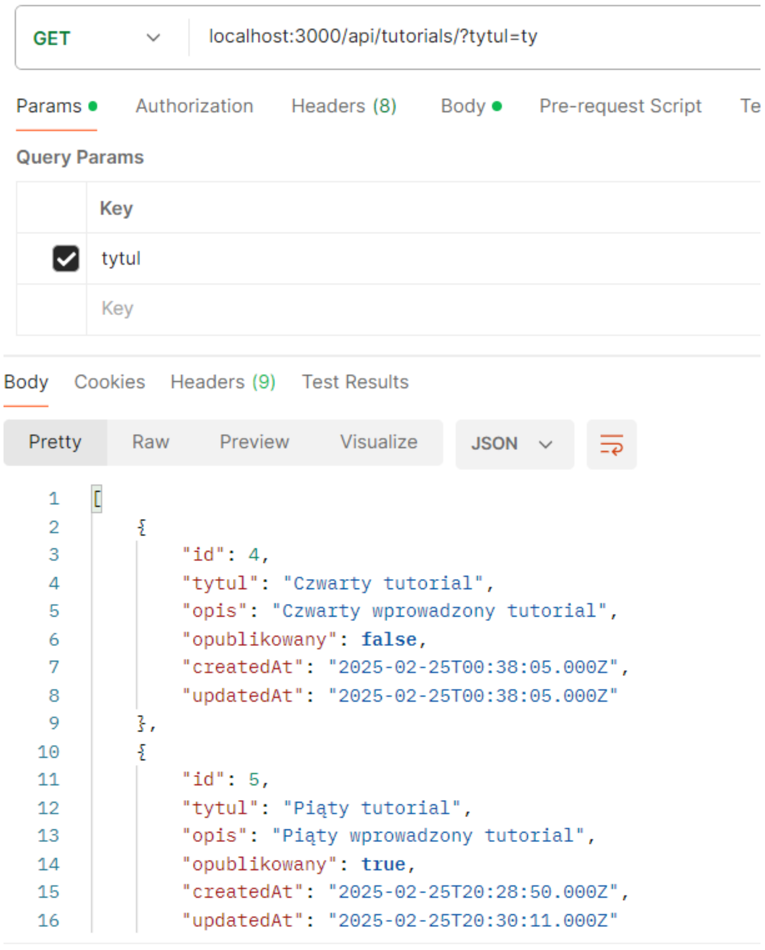
```
{
    "opublikowany": true
}
```

Body | Cookies | Headers (9) | Test Results

Pretty | Raw | Preview | Visualize | JSON

```
{
    "message": "Tutorial został zmieniony."
}
```

5. **Find all Tutorials which title contains 'node':** `GET /tutorials?tytul=ty`



localhost:3000/api/tutorials/?tytul=ty

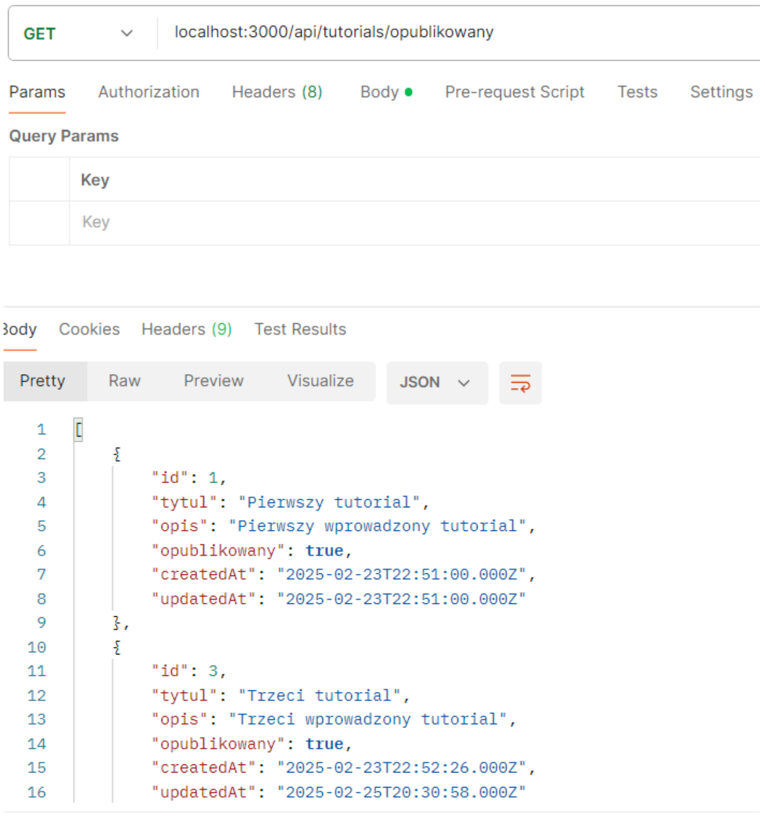
Query Params

Key
✓ tytul
Key

```
[
    {
        "id": 4,
        "tytul": "Czwarty tutorial",
        "opis": "Czwarty wprowadzony tutorial",
        "opublikowany": false,
        "createdAt": "2025-02-25T00:38:05.000Z",
        "updatedAt": "2025-02-25T00:38:05.000Z"
    },
    {
        "id": 5,
        "tytul": "Piąty tutorial",
        "opis": "Piąty wprowadzony tutorial",
        "opublikowany": true,
        "createdAt": "2025-02-25T20:28:50.000Z",
        "updatedAt": "2025-02-25T20:30:11.000Z"
```

5

6. **Find all published Tutorials using** `GET /tutorials/opublikowany` **Api**



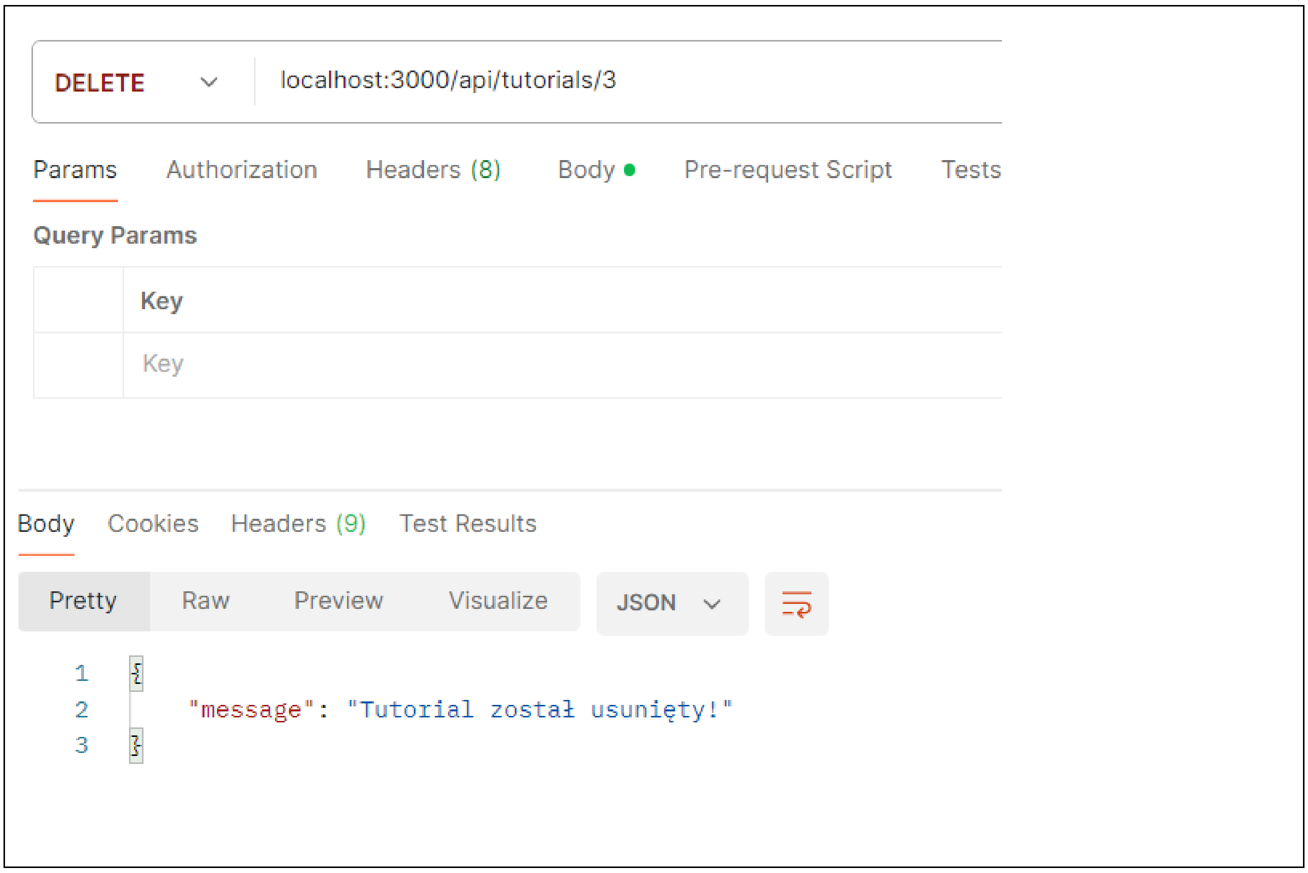
localhost:3000/api/tutorials/opublikowany

Query Params

Key
Key

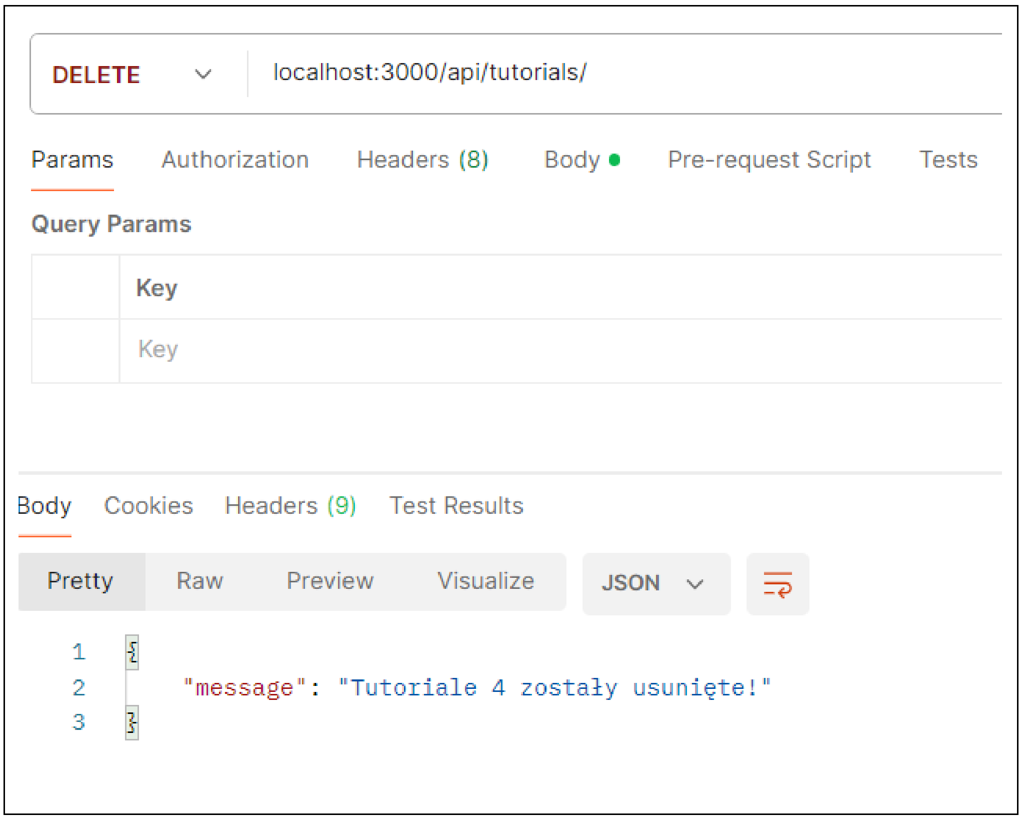
```
[
    {
        "id": 1,
        "tytul": "Pierwszy tutorial",
        "opis": "Pierwszy wprowadzony tutorial",
        "opublikowany": true,
        "createdAt": "2025-02-23T22:51:00.000Z",
        "updatedAt": "2025-02-23T22:51:00.000Z"
    },
    {
        "id": 3,
        "tytul": "Trzeci tutorial",
        "opis": "Trzeci wprowadzony tutorial",
        "opublikowany": true,
        "createdAt": "2025-02-23T22:52:26.000Z",
        "updatedAt": "2025-02-25T20:30:58.000Z"
```

7. Delete a Tutorial using `DELETE /tutorials/:id` Api



```
{
    "message": "Tutorial został usunięty!"
}
```

8. Delete all Tutorials using `DELETE /tutorials` Api



```
{
    "message": "Tutoriale 4 zostały usunięte!"
}
```

Conclusion

Today, we’ve learned how to create Node.js Rest Apis with an Express web server. We also know way to add configuration for MySQL database & Sequelize, create a Sequelize Model, write a controller and define routes for handling all CRUD operations.