LAB 5: Router, database connection, object-relational mapping, and working with APIs in NodeJS

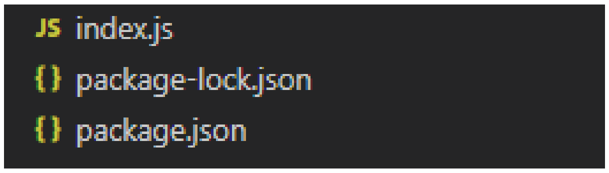
I. Create a Node.js project

Initialize the project: **npm init**

This command creates a **package.json** file.

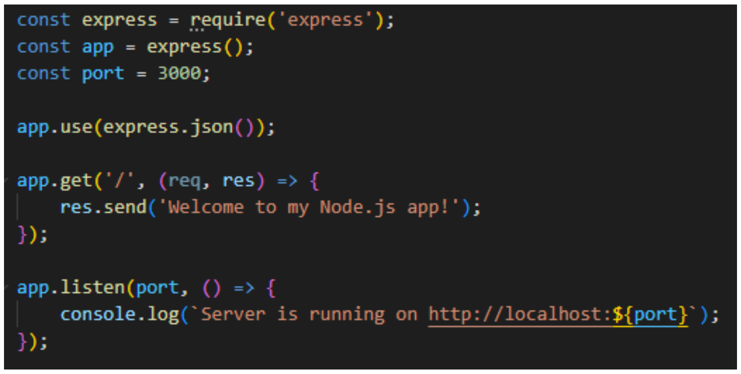
Install Express.js: **npm install express**

Create the main file: Create **index.js**



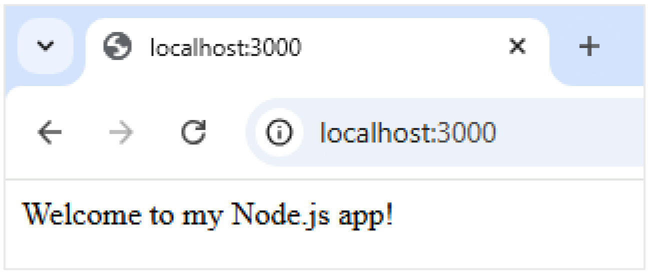
```
index.js
package-lock.json
package.json
```

Set up a basic server: Add the following code to **index.js**:



```
const express = require('express');
const app = express();
const port = 3000;

app.use(express.json());

app.get('/', (req, res) => {
    res.send('Welcome to my Node.js app!');
});

app.listen(port, () => {
    console.log(`Server is running on http://localhost:${port}`);
});
```

Run the server: **node index.js**

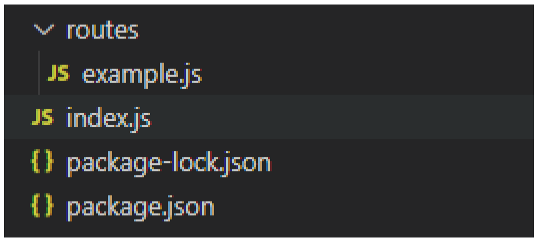


Welcome to my Node.js app!

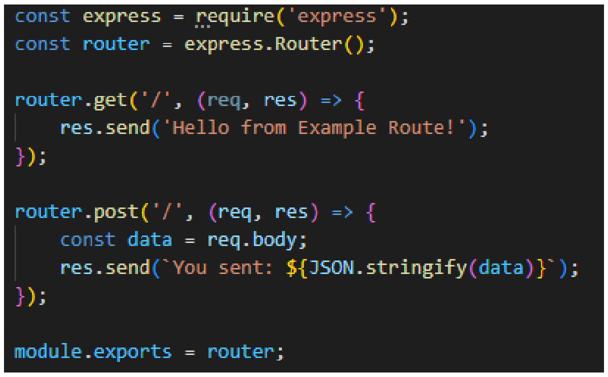
II. Router

Example 1: Basic Router

Create a file **routes/example.js**:

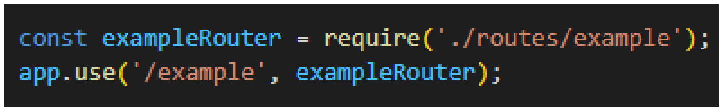


```
routes
  example.js
index.js
package-lock.json
package.json
```



```javascript
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
    res.send('Hello from Example Route!');
});

router.post('/', (req, res) => {
    const data = req.body;
    res.send(`You sent: ${JSON.stringify(data)}`);
});

module.exports = router;
```

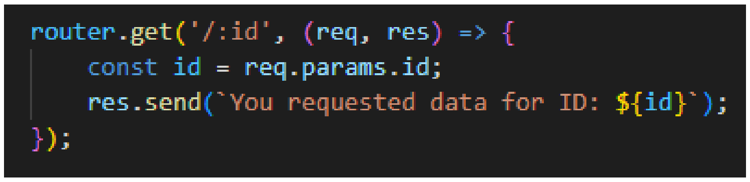
Update **index.js** to use the router:



```javascript
const exampleRouter = require('./routes/example');
app.use('/example', exampleRouter);
```

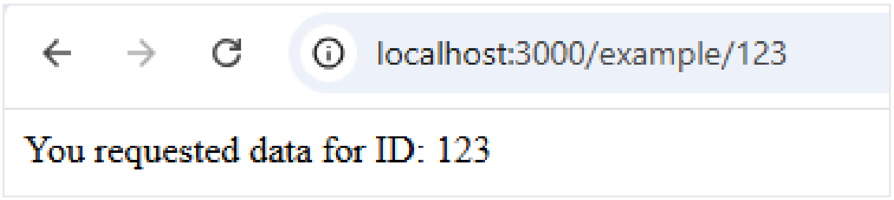
Example 2: Router with URL Parameters

Add a route to handle URL parameters:



```javascript
router.get('/:id', (req, res) => {
    const id = req.params.id;
    res.send(`You requested data for ID: ${id}`);
});
```

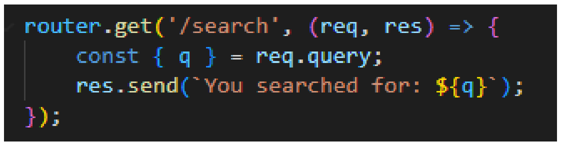
Accessing *http://localhost:3000/example/123* will return:



```
localhost:3000/example/123

You requested data for ID: 123
```

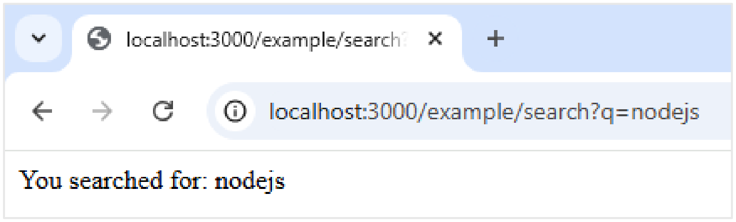
Example 3: Router with Query Parameters

Add a route to handle query strings:



```
router.get('/search', (req, res) => {
    const { q } = req.query;
    res.send(`You searched for: ${q}`);
});
```

Accessing *http://localhost:3000/example/search?q=nodejs* will return:

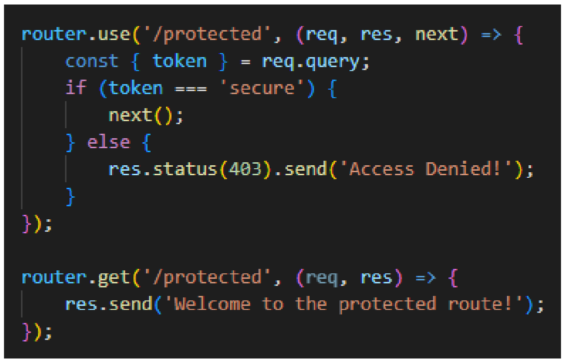


```
localhost:3000/example/search?q=nodejs
```

```
You searched for: nodejs
```

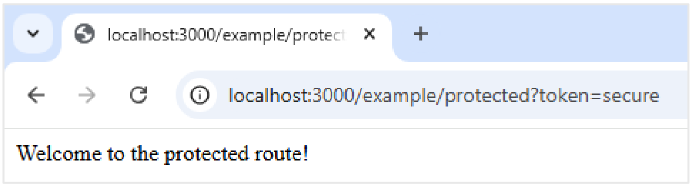
Example 4: Router with Middleware

Add middleware for a specific route:



```
router.use('/protected', (req, res, next) => {
    const { token } = req.query;
    if (token === 'secure') {
        next();
    } else {
        res.status(403).send('Access Denied!');
    }
});

router.get('/protected', (req, res) => {
    res.send('Welcome to the protected route!');
});
```

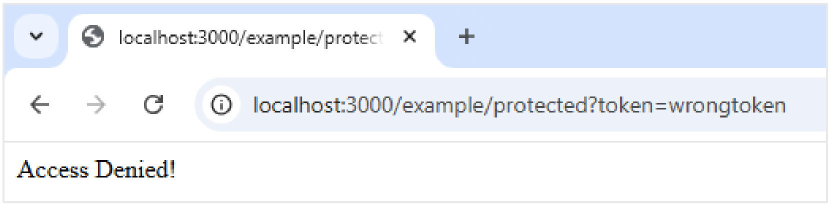
Accessing *http://localhost:3000/example/protected?token=secure* will return:



```
localhost:3000/example/protected?token=secure
```

```
Welcome to the protected route!
```

Accessing *http://localhost:3000/example/protected?token=wrongtoken* will return:



```
localhost:3000/example/protected?token=wrongtoken
```

```
Access Denied!
```

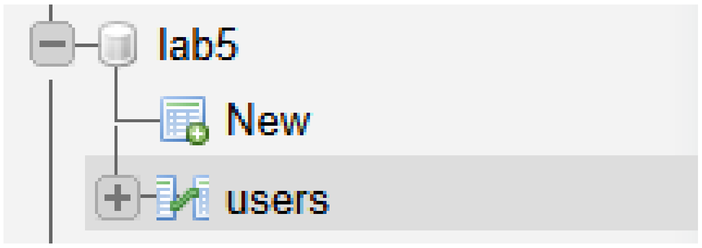
III. Connect to Database

Step 1: Install Required Libraries: **npm install mysql2**

Step 2: Create a Database For MySQL, create a database and a users table:

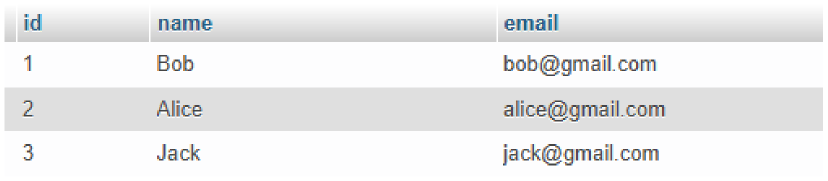
SQL: "CREATE DATABASE lab5;

```
USE lab5;
CREATE TABLE
 users ( id INT IDENTITY(1,1) PRIMARY KEY,
 name NVARCHAR(50) NOT NULL,
 email NVARCHAR(50) NOT NULL );"
```



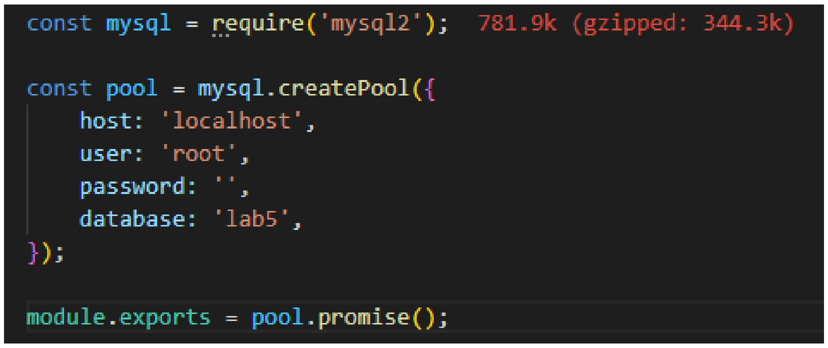
SQL: "INSERT INTO

```
 `users` (`id`, `name`, `email`)
 VALUES
 (NULL, 'Bob', 'bob@gmail.com'),
 (NULL, 'Alice', 'alice@gmail.com');"
```



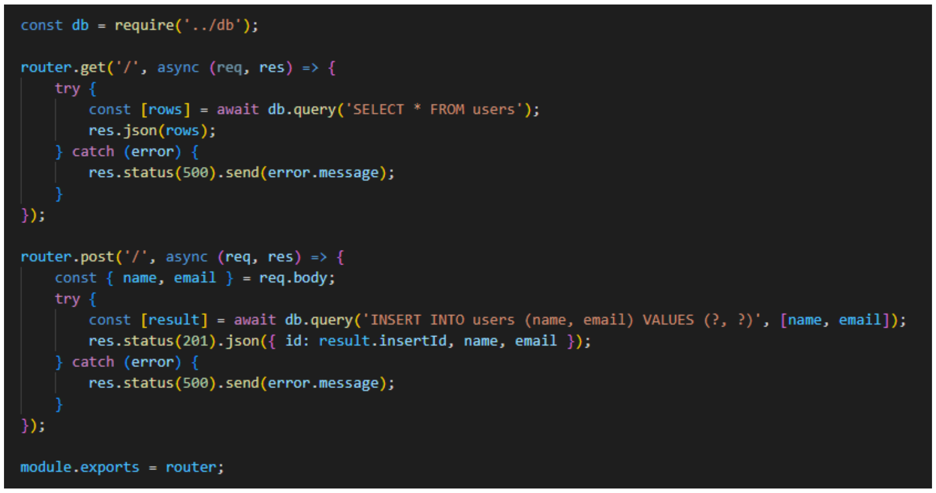
id	name	email
1	Bob	bob@gmail.com
2	Alice	alice@gmail.com
3	Jack	jack@gmail.com

Step 3: Connect and Query the Database File: db.js



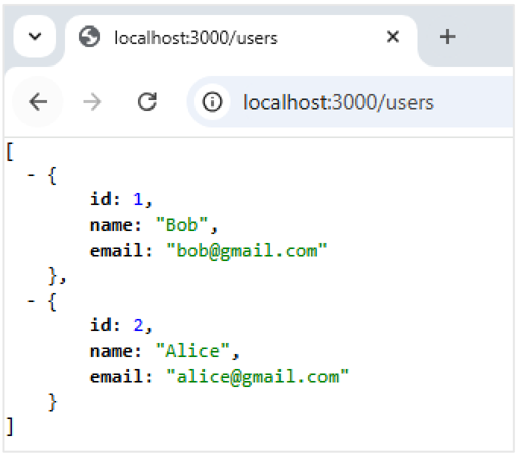
```
const mysql = require('mysql2');  781.9k (gzipped: 344.3k)

const pool = mysql.createPool({
    host: 'localhost',
    user: 'root',
    password: '',
    database: 'lab5',
});

module.exports = pool.promise();
```

File: **routes/users.js**



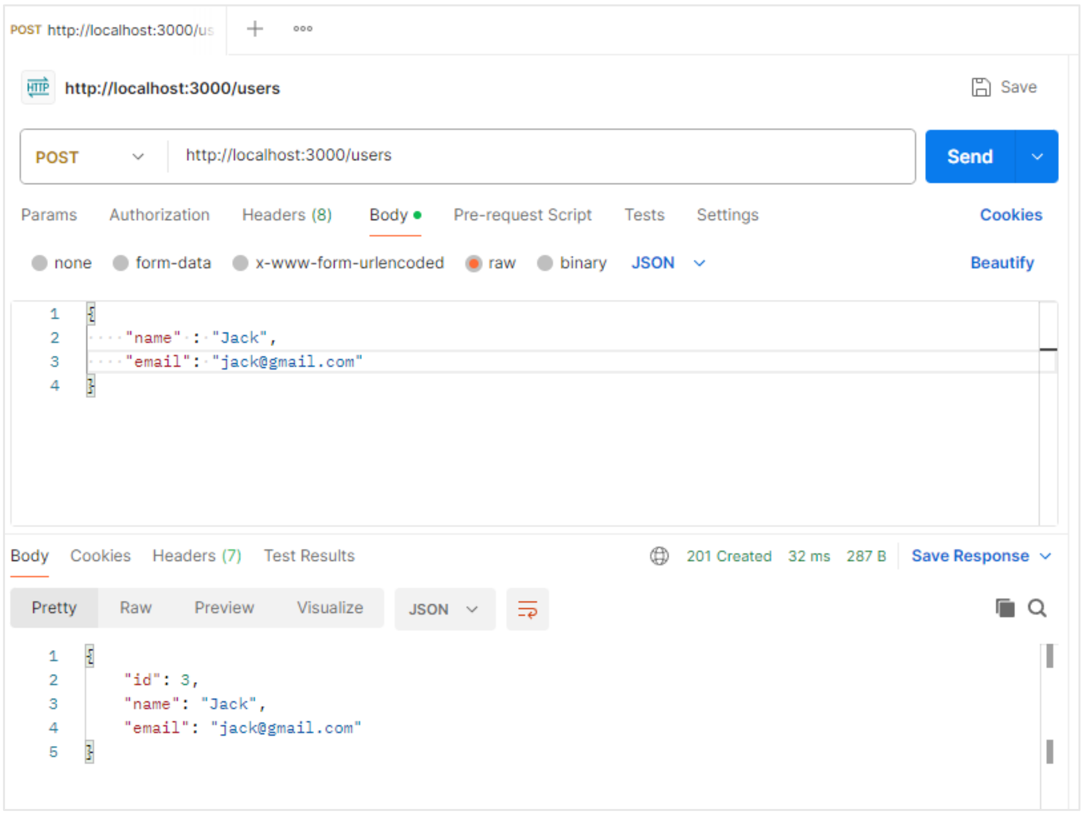
```
const db = require('../db');

router.get('/', async (req, res) => {
    try {
        const [rows] = await db.query('SELECT * FROM users');
        res.json(rows);
    } catch (error) {
        res.status(500).send(error.message);
    }
});

router.post('/', async (req, res) => {
    const { name, email } = req.body;
    try {
        const [result] = await db.query('INSERT INTO users (name, email) VALUES (?, ?)', [name, email]);
        res.status(201).json({ id: result.insertId, name, email });
    } catch (error) {
        res.status(500).send(error.message);
    }
});

module.exports = router;
```

Result:



```
[
  - {
        id: 1,
        name: "Bob",
        email: "bob@gmail.com"
    },
  - {
        id: 2,
        name: "Alice",
        email: "alice@gmail.com"
    }
]
```

Accessing *http://localhost:3000/users*:



POST http://localhost:3000/users

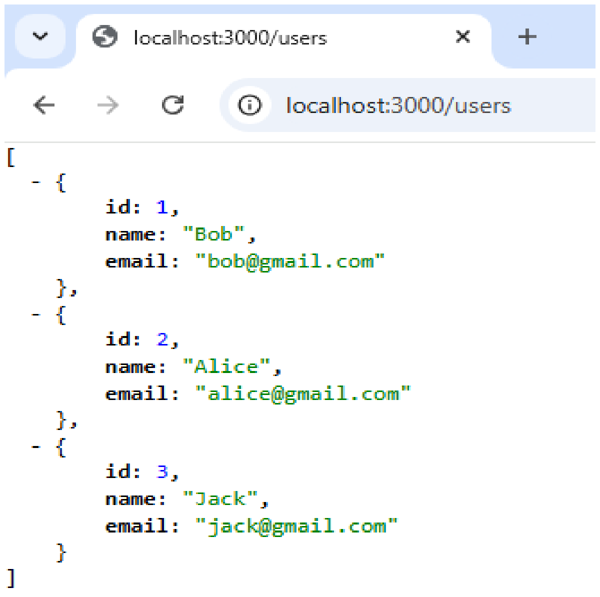
```
{
    "name" : "Jack",
    "email": "jack@gmail.com"
}
```

201 Created 32 ms 287 B

JSON

```
{
    "id": 3,
    "name": "Jack",
    "email": "jack@gmail.com"
}
```

Result:



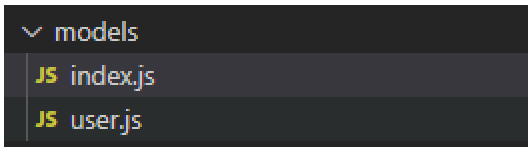
```
[
  - {
        id: 1,
        name: "Bob",
        email: "bob@gmail.com"
    },
  - {
        id: 2,
        name: "Alice",
        email: "alice@gmail.com"
    },
  - {
        id: 3,
        name: "Jack",
        email: "jack@gmail.com"
    }
]
```

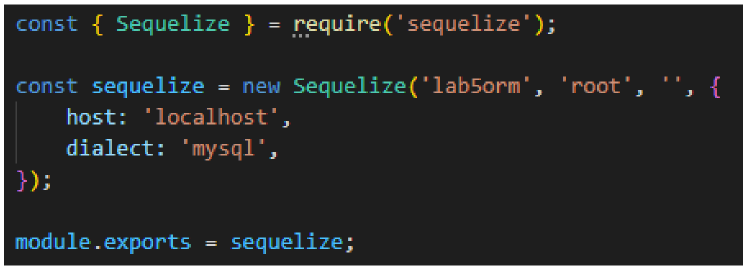
IV. Object Relational Mapping (ORM)

Step 1: Install Sequelize: **npm install sequelize mysql2**

Step 2: Configure Sequelize

Create **models/index.js**:

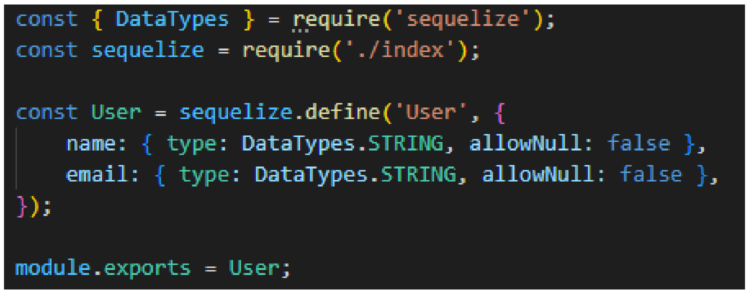




```
const { Sequelize } = require('sequelize');

const sequelize = new Sequelize('lab5orm', 'root', '', {
    host: 'localhost',
    dialect: 'mysql',
});

module.exports = sequelize;
```

Step 3: Define a Model

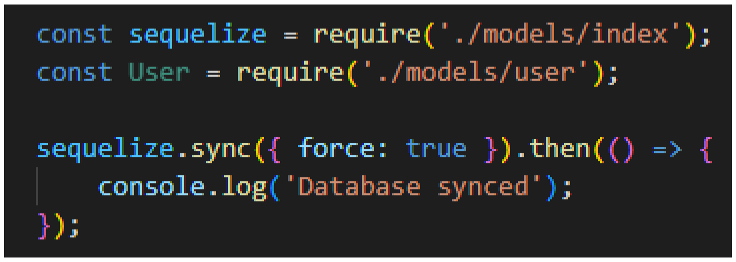
Create **models/user.js**:



```
const { DataTypes } = require('sequelize');
const sequelize = require('./index');

const User = sequelize.define('User', {
    name: { type: DataTypes.STRING, allowNull: false },
    email: { type: DataTypes.STRING, allowNull: false },
});

module.exports = User;
```

Step 4: Sync and Use ORM

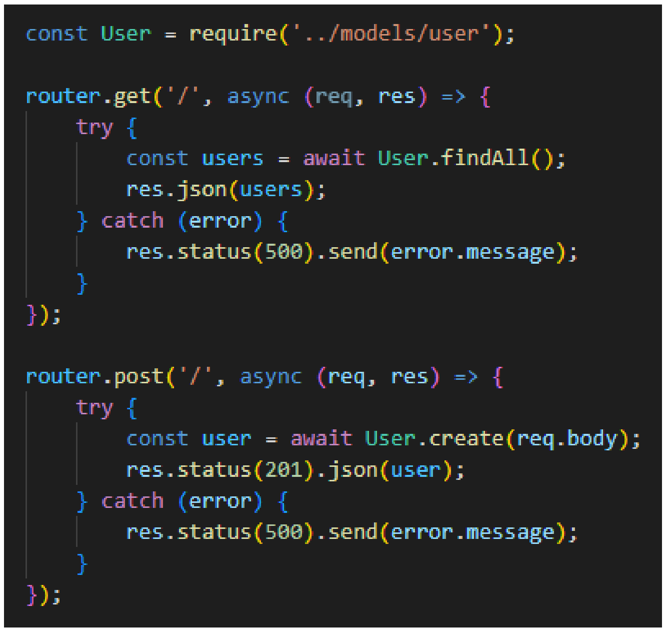
Modify **index.js** to sync the database:



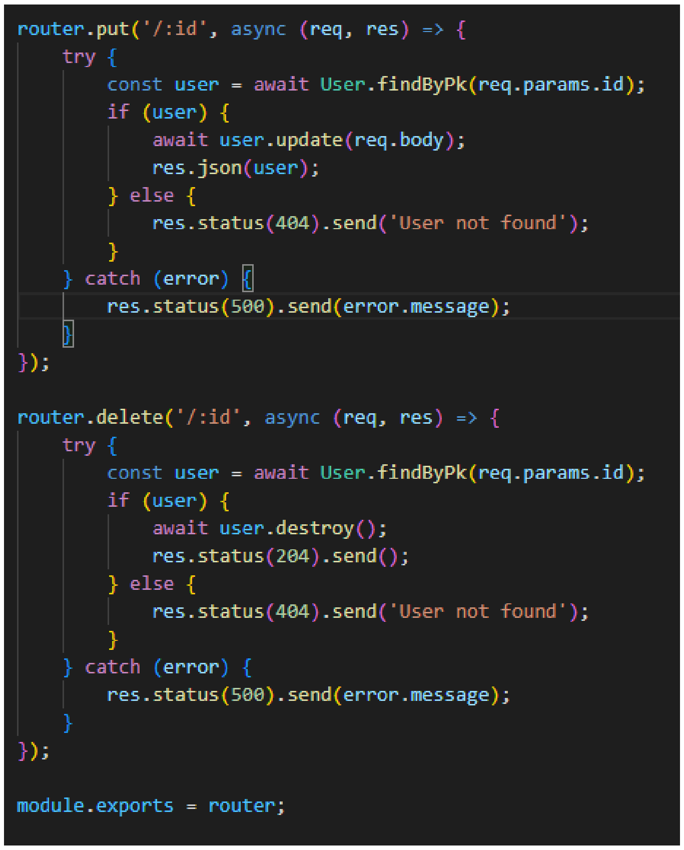
```
const sequelize = require('./models/index');
const User = require('./models/user');

sequelize.sync({ force: true }).then(() => {
    console.log('Database synced');
});
```

File: **routes/users.js**

Using ORM for CRUD operations:



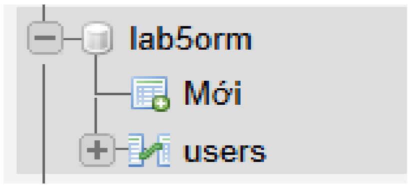
```javascript
const User = require('../models/user');

router.get('/', async (req, res) => {
    try {
        const users = await User.findAll();
        res.json(users);
    } catch (error) {
        res.status(500).send(error.message);
    }
});

router.post('/', async (req, res) => {
    try {
        const user = await User.create(req.body);
        res.status(201).json(user);
    } catch (error) {
        res.status(500).send(error.message);
    }
});
```

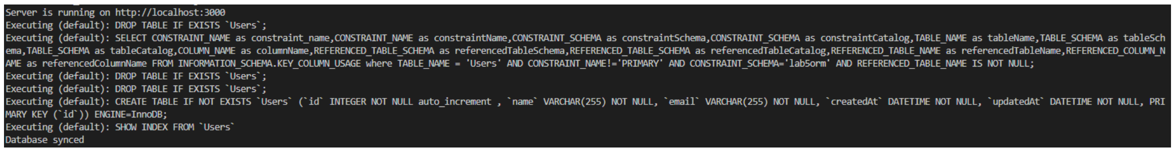


```javascript
router.put('/:id', async (req, res) => {
    try {
        const user = await User.findByPk(req.params.id);
        if (user) {
            await user.update(req.body);
            res.json(user);
        } else {
            res.status(404).send('User not found');
        }
    } catch (error) {
        res.status(500).send(error.message);
    }
});

router.delete('/:id', async (req, res) => {
    try {
        const user = await User.findByPk(req.params.id);
        if (user) {
            await user.destroy();
            res.status(204).send();
        } else {
            res.status(404).send('User not found');
        }
    } catch (error) {
        res.status(500).send(error.message);
    }
});

module.exports = router;
```

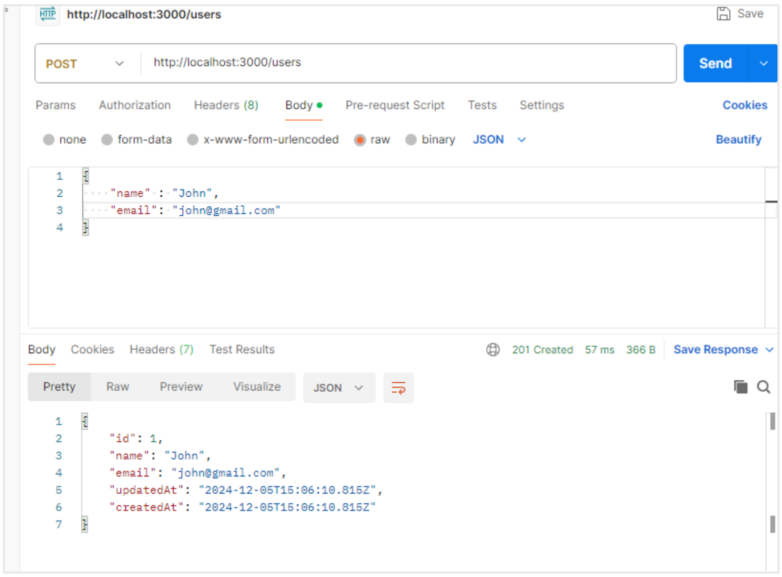
Result:

MySQL database:





```
Server is running on http://localhost:3000
Executing (default): DROP TABLE IF EXISTS `Users`;
Executing (default): SELECT CONSTRAINT_NAME as constraint_name,CONSTRAINT_NAME as constraintName,CONSTRAINT_SCHEMA as constraintSchema,CONSTRAINT_SCHEMA as constraintCatalog,TABLE_NAME as tableName,TABLE_SCHEMA as tableSch
ema,TABLE_SCHEMA as tableCatalog,COLUMN_NAME as columnName,REFERENCED_TABLE_SCHEMA as referencedTableSchema,REFERENCED_TABLE_SCHEMA as referencedTableCatalog,REFERENCED_TABLE_NAME as referencedTableName,REFERENCED_COLUMN_N
AME as referencedColumnName FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE where TABLE_NAME = 'Users' AND CONSTRAINT_NAME!='PRIMARY' AND CONSTRAINT_SCHEMA='lab5orm' AND REFERENCED_TABLE_NAME IS NOT NULL;
Executing (default): DROP TABLE IF EXISTS `Users`;
Executing (default): DROP TABLE IF EXISTS `Users`;
Executing (default): CREATE TABLE IF NOT EXISTS `Users` (`id` INTEGER NOT NULL auto_increment , `name` VARCHAR(255) NOT NULL, `email` VARCHAR(255) NOT NULL, `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, PRI
MARY KEY (`id`)) ENGINE=InnoDB;
Executing (default): SHOW INDEX FROM `Users`
Database synced
```



http://localhost:3000/users

POST http://localhost:3000/users

```
{
    "name" : "John",
    "email": "john@gmail.com"
}
```

201 Created 57 ms 366 B

JSON

```
{
    "id": 1,
    "name": "John",
    "email": "john@gmail.com",
    "updatedAt": "2024-12-05T15:06:10.815Z",
    "createdAt": "2024-12-05T15:06:10.815Z"
}
```

Query string:

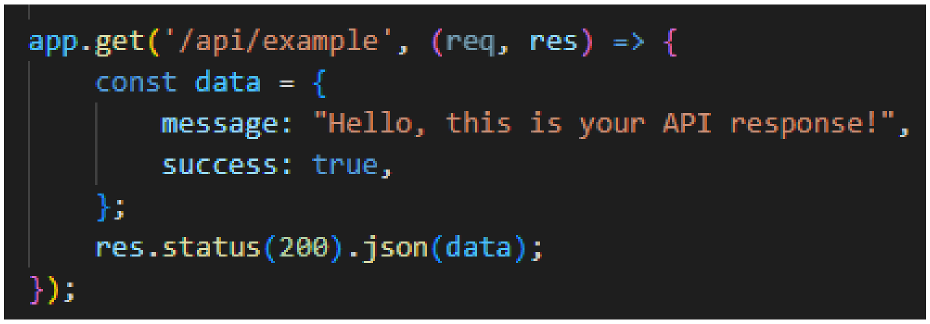
```
Executing (default): INSERT INTO `Users` (`id`,`name`,`email`,`createdAt`,`updatedAt`) VALUES (DEFAULT,?,?,?,?);
```

The behavior of **DROP TABLE IF EXISTS** typically occurs when using an ORM like Sequelize with the **sync({ force: true })** option. This option drops and recreates all tables, which is useful during development but should be avoided in production or when you don't want to lose data. Here’s how to ensure DROP TABLE is not executed:

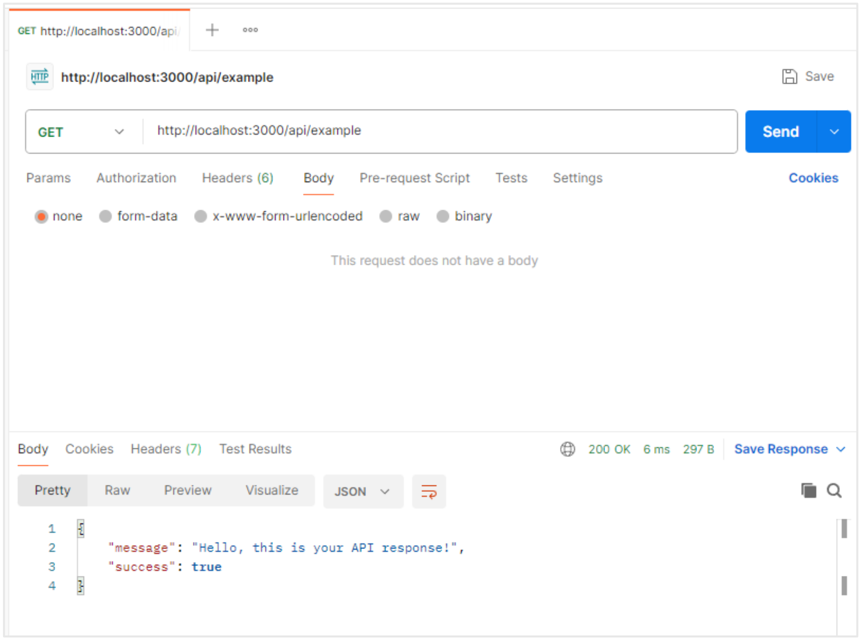
```
sequelize.sync()
    .then(() => {
        console.log('Database synced without dropping tables');
    })
    .catch((err) => {
        console.error('Failed to sync database:', err.message);
    });
```

V. Working with API

1. Basic JSON Response



```
app.get('/api/example', (req, res) => {
    const data = {
        message: "Hello, this is your API response!",
        success: true,
    };
    res.status(200).json(data);
});
```



GET http://localhost:3000/api/example

● none ○ form-data ○ x-www-form-urlencoded ○ raw ○ binary

This request does not have a body

200 OK 6 ms 297 B

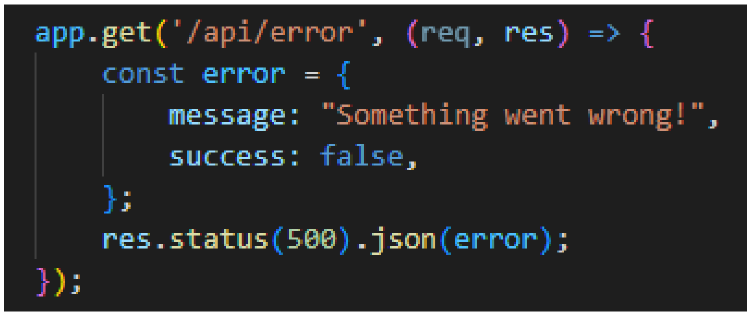
```
{
    "message": "Hello, this is your API response!",
    "success": true
}
```

res.json(): Automatically sets the Content-Type header to application/json.

res.status(): Allows setting an HTTP status code.

2. Sending Error Responses

You can standardize error responses with an appropriate status code and message:

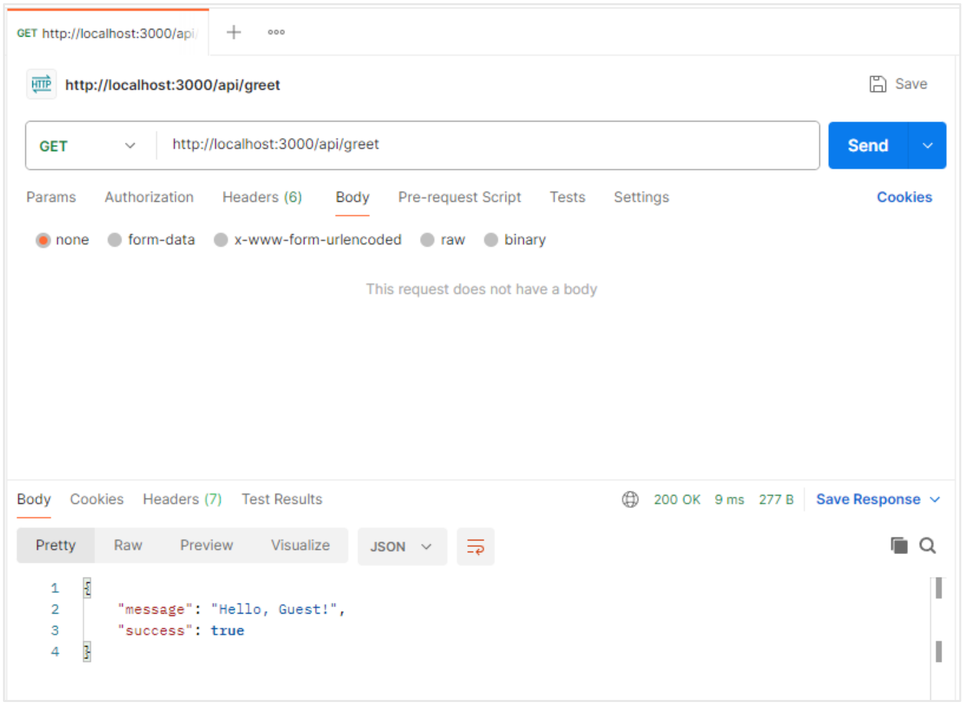


```
app.get('/api/error', (req, res) => {
    const error = {
        message: "Something went wrong!",
        success: false,
    };
    res.status(500).json(error);
});
```

3. Sending Data with Query Parameters

You can retrieve query parameters from the URL and return data dynamically:

```
app.get('/api/greet', (req, res) => {
    const name = req.query.name || 'Guest';
    res.status(200).json({
        message: `Hello, ${name}!`,
        success: true,
    });
});
```



GET http://localhost:3000/api/

http://localhost:3000/api/greet

Save

GET http://localhost:3000/api/greet Send

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

[✓ none] form-data x-www-form-urlencoded raw binary

This request does not have a body

Body Cookies Headers (7) Test Results 200 OK 9 ms 277 B Save Response

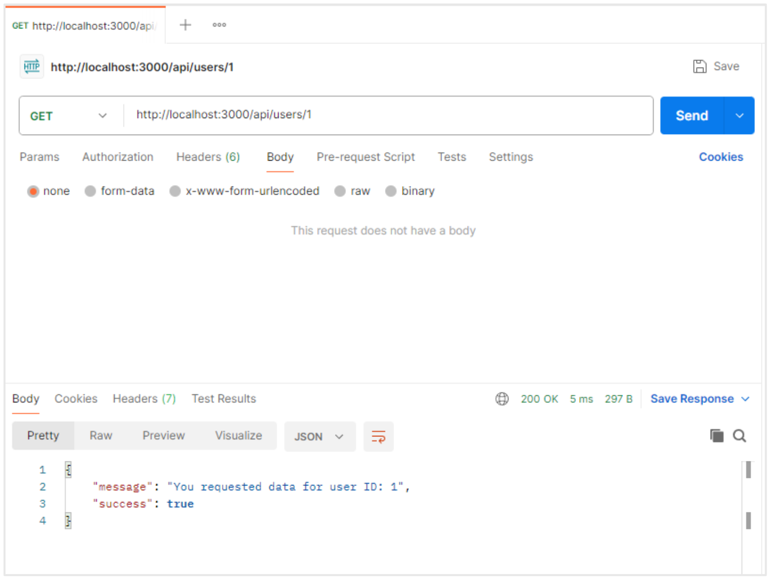
Pretty Raw Preview Visualize JSON

```
{
    "message": "Hello, Guest!",
    "success": true
}
```

4. Sending Data with URL Parameters

Use dynamic segments in routes to retrieve values:

```
app.get('/api/users/:id', (req, res) => {
    const userId = req.params.id;
    res.status(200).json({
        message: `You requested data for user ID: ${userId}`,
        success: true,
    });
});
```


GET http://localhost:3000/api/users/1

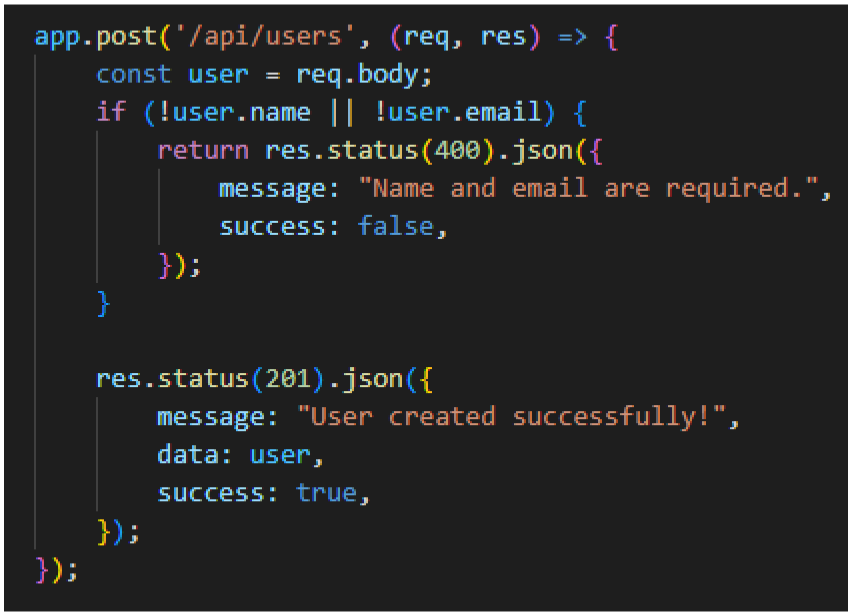
Body: none — This request does not have a body

200 OK 5 ms 297 B

```
{
    "message": "You requested data for user ID: 1",
    "success": true
}
```

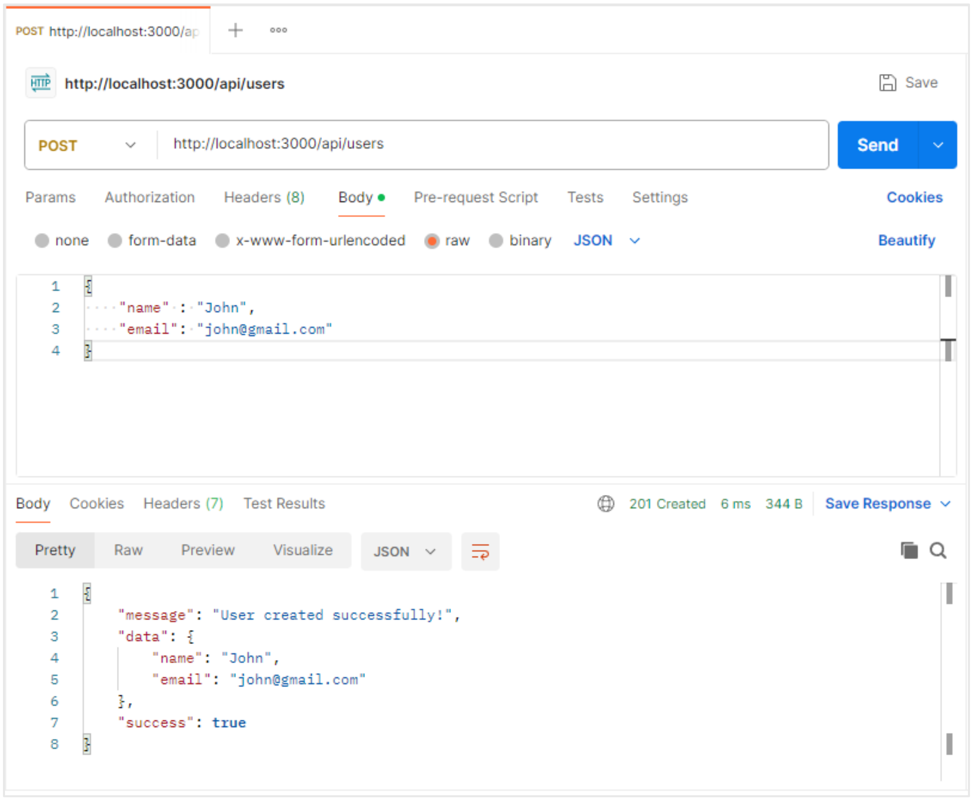
5. Handling POST Requests (Receive Data)

To handle POST requests, use middleware to parse JSON payloads.



```
app.post('/api/users', (req, res) => {
    const user = req.body;
    if (!user.name || !user.email) {
        return res.status(400).json({
            message: "Name and email are required.",
            success: false,
        });
    }

    res.status(201).json({
        message: "User created successfully!",
        data: user,
        success: true,
    });
});
```

Result:



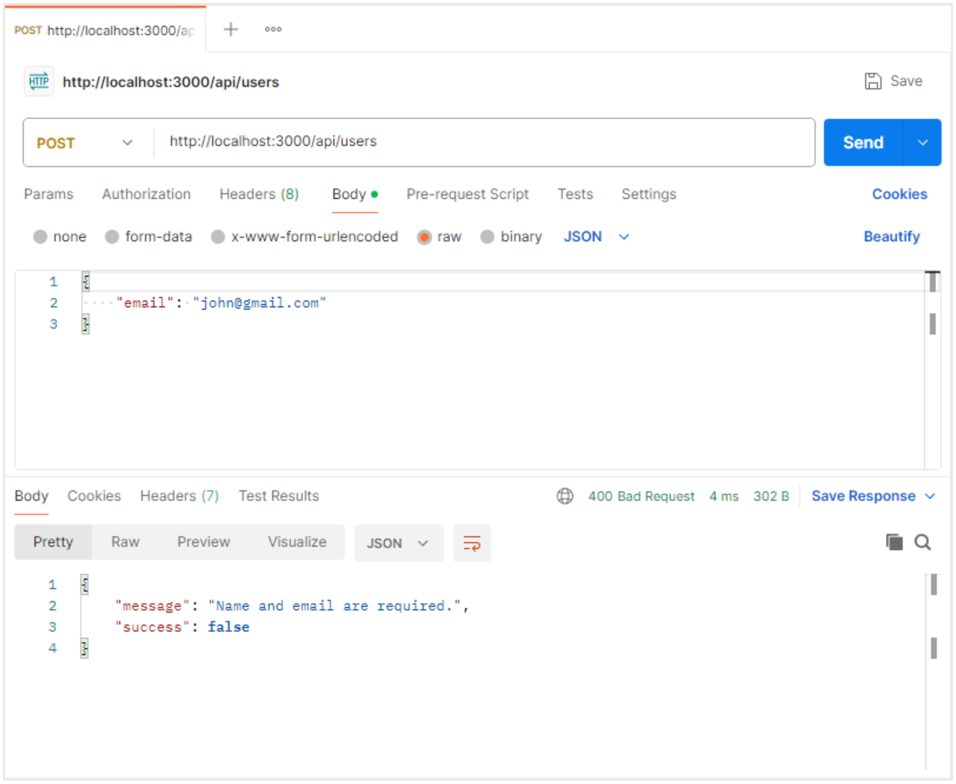
POST http://localhost:3000/api/users

Body: raw JSON

```
{
    "name" : "John",
    "email": "john@gmail.com"
}
```

201 Created 6 ms 344 B

```
{
    "message": "User created successfully!",
    "data": {
        "name": "John",
        "email": "john@gmail.com"
    },
    "success": true
}
```



POST http://localhost:3000/api/users

Body: raw JSON

```
{
    "email": "john@gmail.com"
}
```

400 Bad Request 4 ms 302 B

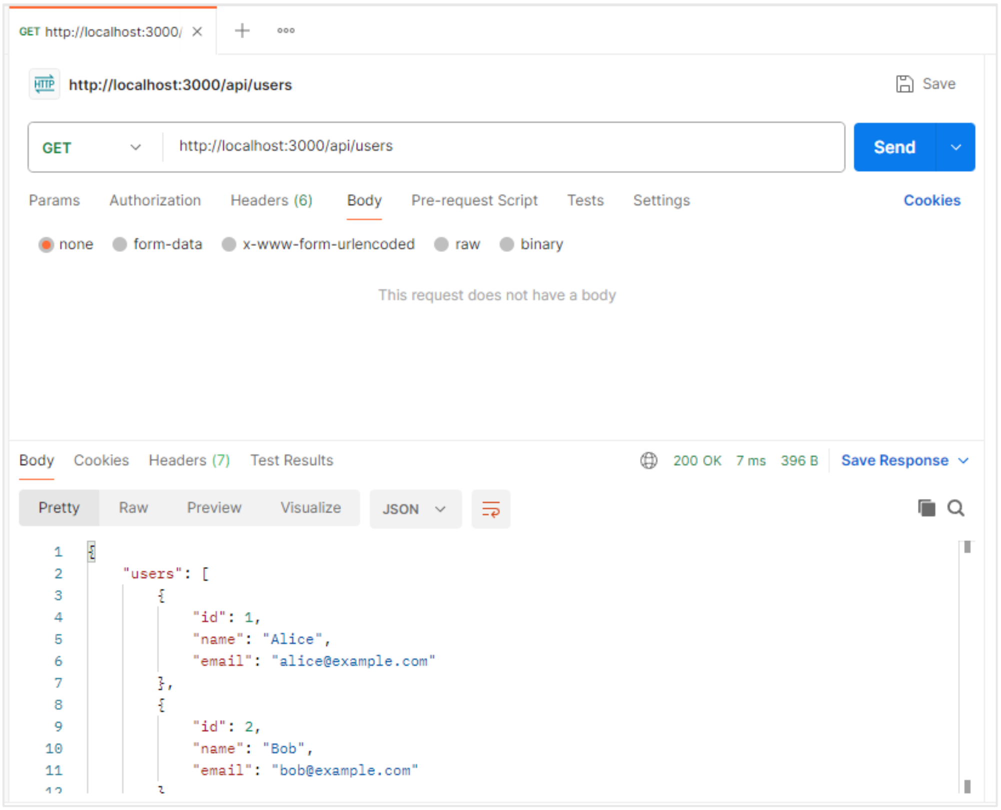
```
{
    "message": "Name and email are required.",
    "success": false
}
```

6. CRUD API Example

```
let users = [
    { id: 1, name: 'Alice', email: 'alice@example.com' },
    { id: 2, name: 'Bob', email: 'bob@example.com' },
];
```

GET /api/users - Get All Users

```
app.get('/api/users', (req, res) => {
    res.status(200).json({ users, success: true });
});

app.get('/api/users/:id', (req, res) => {
    const user = users.find(u => u.id === parseInt(req.params.id));
    if (user) {
        res.status(200).json({ user, success: true });
    } else {
        res.status(404).json({ message: "User not found", success: false });
    }
});
```



http://localhost:3000/api/users

GET http://localhost:3000/api/users

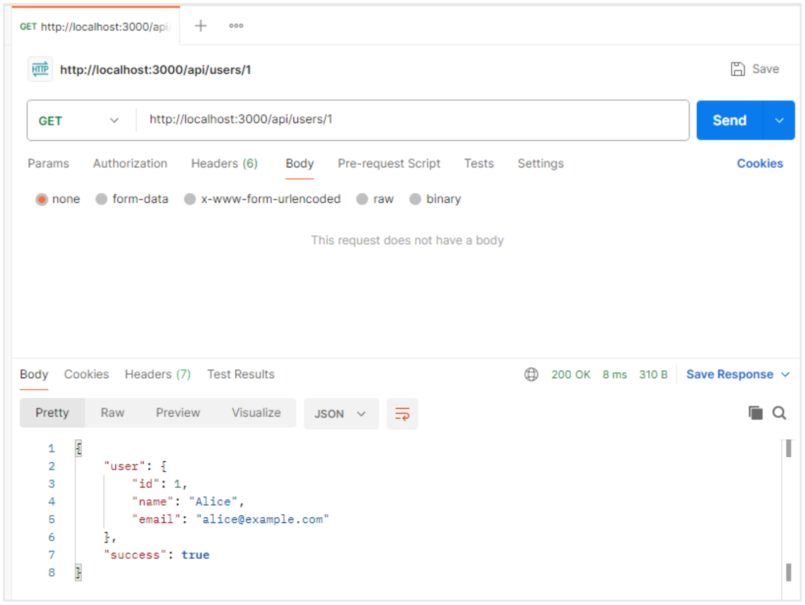
[✓ none] form-data x-www-form-urlencoded raw binary

This request does not have a body

200 OK 7 ms 396 B

```
{
    "users": [
        {
            "id": 1,
            "name": "Alice",
            "email": "alice@example.com"
        },
        {
            "id": 2,
            "name": "Bob",
            "email": "bob@example.com"
        }
```

GET /api/users/:id - Get a Specific User by ID



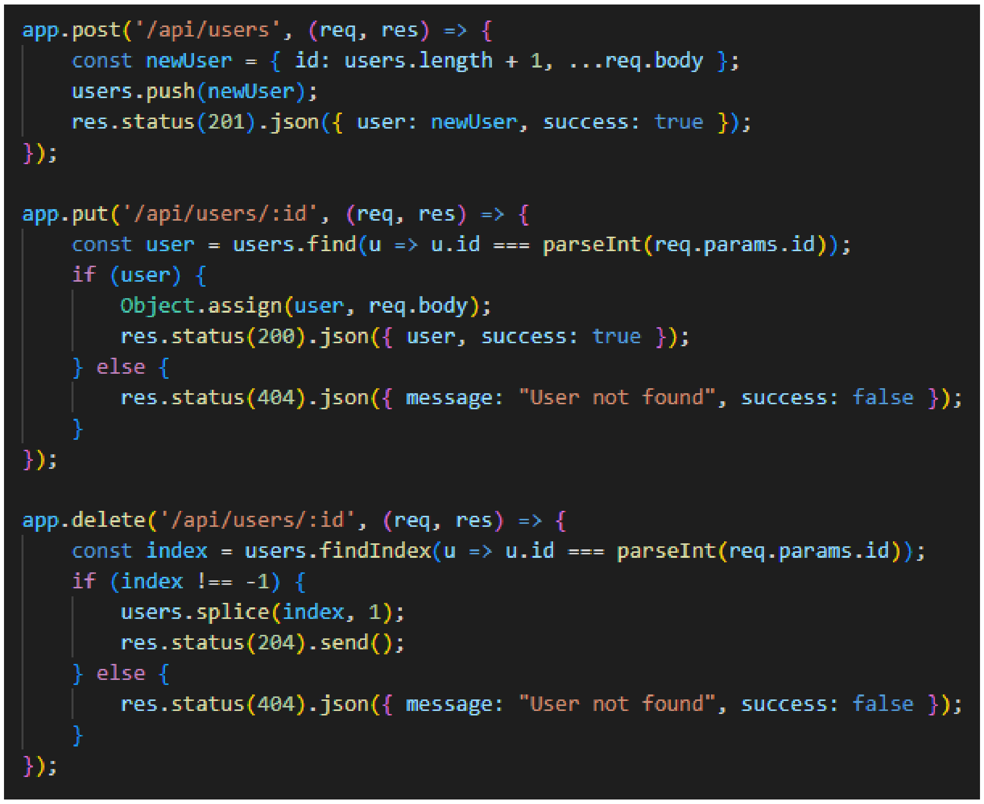
http://localhost:3000/api/users/1

This request does not have a body

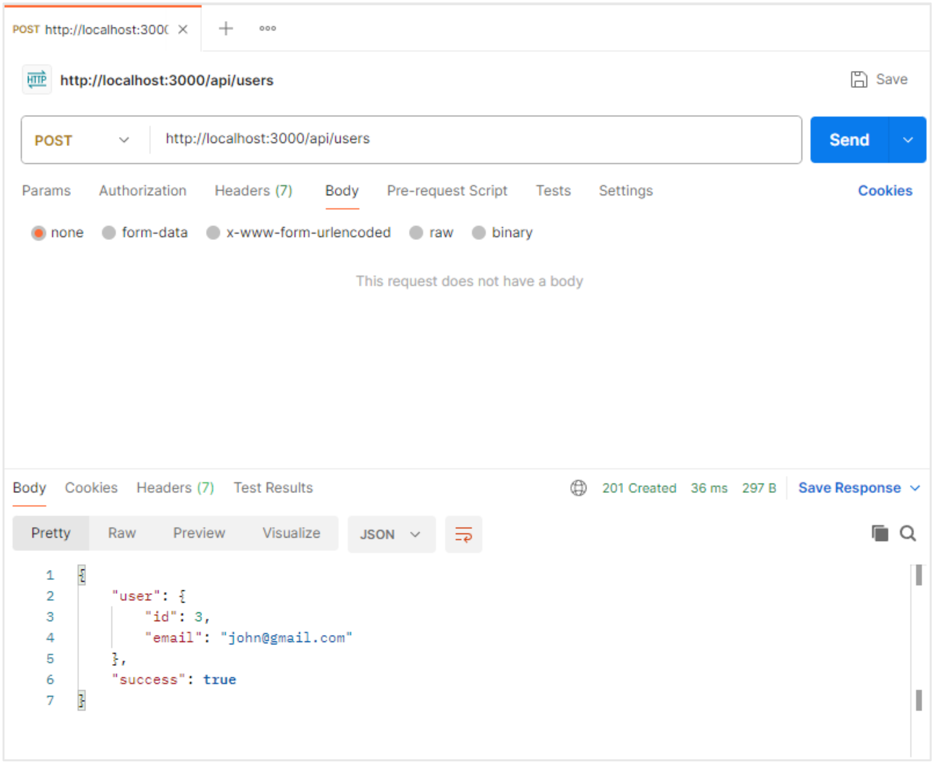
200 OK 8 ms 310 B

```
{
    "user": {
        "id": 1,
        "name": "Alice",
        "email": "alice@example.com"
    },
    "success": true
}
```

POST /api/users - Create a New User



```
app.post('/api/users', (req, res) => {
    const newUser = { id: users.length + 1, ...req.body };
    users.push(newUser);
    res.status(201).json({ user: newUser, success: true });
});

app.put('/api/users/:id', (req, res) => {
    const user = users.find(u => u.id === parseInt(req.params.id));
    if (user) {
        Object.assign(user, req.body);
        res.status(200).json({ user, success: true });
    } else {
        res.status(404).json({ message: "User not found", success: false });
    }
});

app.delete('/api/users/:id', (req, res) => {
    const index = users.findIndex(u => u.id === parseInt(req.params.id));
    if (index !== -1) {
        users.splice(index, 1);
        res.status(204).send();
    } else {
        res.status(404).json({ message: "User not found", success: false });
    }
});
```

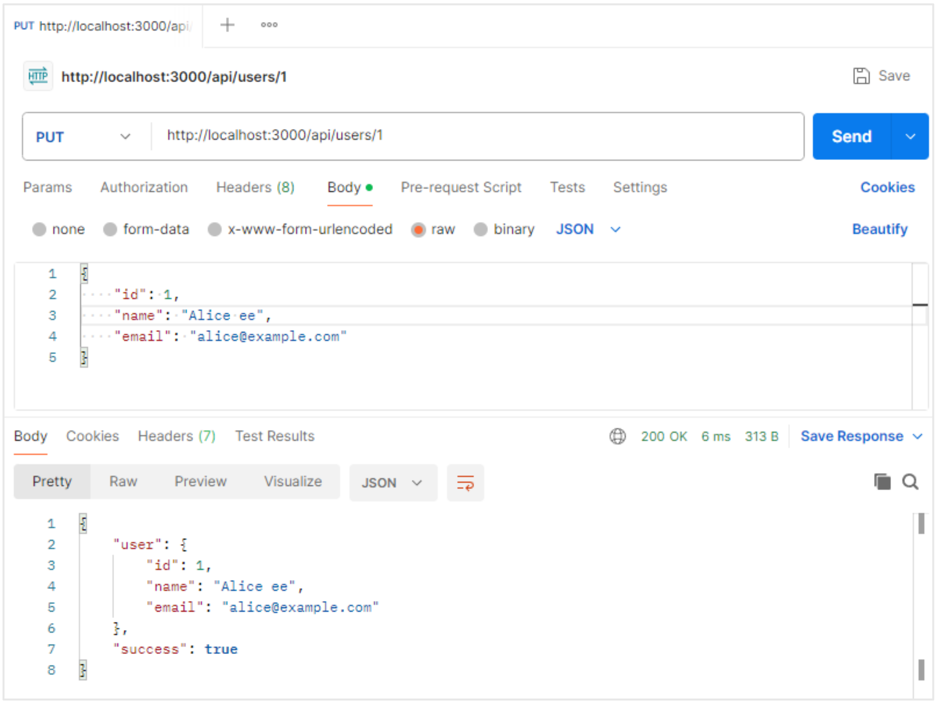

POST http://localhost:3000/api/users

Body: none — This request does not have a body

201 Created 36 ms 297 B

```
{
    "user": {
        "id": 3,
        "email": "john@gmail.com"
    },
    "success": true
}
```

PUT /api/users/:id - Update an Existing User by ID


PUT http://localhost:3000/api/users/1

```
{
    "id": 1,
    "name": "Alice ee",
    "email": "alice@example.com"
}
```

200 OK 6 ms 313 B

```
{
    "user": {
        "id": 1,
        "name": "Alice ee",
        "email": "alice@example.com"
    },
    "success": true
}
```

Exercise:

1. Create a new project named **Lab5_Ex1** with the following requirements:

Create endpoints to view, add, delete, and update User, Product, and ShoppingCart objects in two ways:

1. **Using standard query writing.**

Create a database that includes User, Product, and ShoppingCart with corresponding data types:

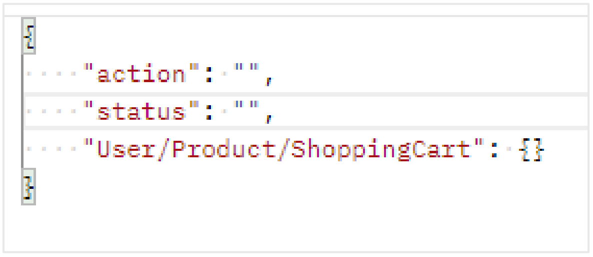
User: Contains the following information: UserId, Full Name, Address, Registration Date.

Product: Contains: ProductId, Product Name, Price, Manufacturing Date.
ShoppingCart: Identify the necessary attributes to store user and shopping cart information.

2. **Using ORM (Object-Relational Mapping).**

Define the required objects and appropriate data types for the models.

Return the results in **API format as follows**:



```
{
    "action": "",
    "status": "",
    "User/Product/ShoppingCart": {}
}
```

- For actions such as adding, deleting, and updating: display information about the object just interacted with.
- For actions such as viewing: display information about all objects.

2. Create an endpoint that accepts a user's email and sends an email with any content to the provided email address.

3. Create an endpoint capable of receiving and storing images, and another endpoint to display the stored images.

4. Create an endpoint that, when called, will fetch all information from the URL *https://jsonplaceholder.typicode.com/users,* create appropriate classes to map the objects from the URL to the classes, and save them to the database.