

# Deploy your apps to a supercloud in a few clicks

This Engineering Education program is supported by Section. Instantly deploy your GitHub apps, Docker containers or K8s namespaces to a supercloud.

Try It For Free

## Understanding Node.js Sequelize ORM Models

November 15, 2021

Topics: [Languages](#)

**Sequelize** is an Object Relational Mapper for Node.js. Sequelize lets us connect to a database and perform operations without writing raw SQL queries. It abstracts SQL queries and makes it easier to interact with database models as objects.

This article will help you understand Sequelize models, from model definition to model usage. Sequelize works with all the SQL-based databases. In this tutorial, I will use MySQL. However, you can use any SQL-based database of your choice.

## Table of contents

Sequelize set up

Introduction to models in Sequelize

Sequelize data types

Creating database model

Sequelize model constraints and validations

Sequelize model associations

App testing

Conclusion

## Prerequisites

For easier follow up on this article, you may require the following:

Basic understanding of [Node.js](#).

Node.js installed on your computer.

Knowledge in [relational database management systems \(RDBMS\)](#).

An SQL-based DBMS installed on your computer.

Basic knowledge in Sequelize installation and set-up. Luckily, this has already been covered in the article, [Introduction to Sequelize ORM for Node.js](#).

## Objectives

At the end of this tutorial, you should be able to:

Validate models.

Define model associations.

Define models in Sequelize.

Generate database schema from Sequelize models.

## Sequelize set up

This tutorial assumes that you have already installed [Node.js](#), an SQL-based DBMS, and a code editor.

Install Sequelize by running the command below on your command line.

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

Install Sequelize database driver for the database you would like to use by running one of the commands below.

```
npm install --save pg pg-hstore # PostgreSQL  
npm install --save mysql2 # MySQL  
npm install --save tedious # MS SQL
```

Install npm package [Sequelize-CLI](#).

```
npm install - g sequelize-cli
```

```
sequelize init
```

The above command creates the following folders:

`config`: a folder that contains config file

`migrations`: a folder that holds all migration files

`models`: a folder that contains all models for the project

`seeders`: a folder that holds all the seed files

For more Sequelize-CLI commands, run `sequelize` on the command line or refer to the [official documentation](#).

**All Sequelize-CLI commands are supposed to run in the project folder.**

## Database connection

Navigate to the `config.json` file under the `config` folder generated earlier to connect to the database. This file holds the database connections for development, test, and production databases. Edit the development database connection as shown in the code snippet below.

```
"development": {  
  "username": "root",  
  "password": null, // Your password. If the password is blank,
```

```
"port": 3307  
}
```

## Create database

We need to create the database defined in the database connection above. Create a database by running the command below on the command line. You can also create the database directly on your DBMS.

```
sequelize db:create
```

## Introduction to models in Sequelize

Models are the backbone of any Object-Relational Mapping (ORM). Therefore, you must understand how to work with models extensively to realize the full benefits of ORM.

Models are database-independent. A model is an abstraction that reflects an entity or a table in the database. A model in Sequelize defines the entity's name in the database, the entity attributes, and their data types.

In this tutorial, let's consider a database that stores users and blog posts. We need two tables to store users and posts. There will be a one-to-many relationship between the entity `User` and the entity `Post`. For this tutorial, we will keep it simple.

## Sequelize data types

## Common data types in Sequelize:

```
DataTypees.STRING      // VARCHAR(255)
DataTypees.TEXT        // Text
DataTypees.BOOLEAN     // TINYINT(1)
DataTypees.INTEGER     // Integer
DataTypees.FLOAT       // Float
DataTypees.DOUBLE      // Double
DataTypees.DECIMAL     // Decimal
DataTypees.DATE        // Date
DataTypees.DATEONLY    // Date without time
```

For more supported data types, refer to the [Sequelize documentation](#).

## Creating database model

In this tutorial, we will use Sequelize-CLI to create models. Run the commands below on the command line in the project folder to create user and post models. Replace `ModelName`, `attribute1`, `attribute2`, `attribute3`, and `datatype` with your values.

```
sequelize model:generate --name ModelName --attributes attribute1
```

The above command will create the models. New files in the `migrations` folder and `models` folder are created. To modify each model, edit the files as per the model name.

the primary key for each table. The attribute is automatically generated.

You can add more attributes by editing the model files generated, `user.js`, and `post.js`. By default, the table name is as same as the model name. Open model files and define the table name as shown below.

```
{
  sequelize,
  //define table name
  tableName: 'users',
  modelName: 'User',
});
```

```
{
  sequelize,
  //define table name
  tableName: 'posts',
  modelName: 'Post',
});
```

To implement the models into the database, we need to run the Node.js app. In the project root, create a script, `app.js`. We will use the [Express framework](#). Express is a Node.js web framework for creating servers in a simplified manner. Install Express by running the command below on your terminal.

Add the code below in the script `app.js`.

```
const express = require("express");
const {sequelize, User, Post} = require('./models'); // import models

const app = express(); // create a new express app
app.use(express.json());

app.get('/', function (req, res) {
  res.send('App running')
});

app.listen({port: 5005}, async() =>{
  await User.sync({ force: true });
});
```

On the command line, run the command below to start the app.

```
node app.js
```

The app automatically connects to the database and synchronizes the models with the database.

Sequelize automatically creates two new columns named `createdAt` and `updatedAt`. The two columns store timestamps showing when a record was created and updated, respectively.



## Sequelize model constraints and validations

Constraints are data rules defined at the SQL level. If the SQL query does not meet the constraint rules, the database throws an error. Sequelize forwards the error to JavaScript. Open `user.js` and add constraints to the `email` attribute, as shown below.

```
email: {  
  type: DataTypes.STRING,  
  unique: true,  
  allowNull: false  
}
```

A database error will be thrown if the email is null or not unique in the above example.

Validation is done at the JavaScript level by Sequelize. Sequelize provides built-in validator functions. You can also create your custom validation functions. SQL queries will execute if only the validation passes. Below are some of the commonly used Sequelize built-in validators. For more Sequelize validators, check the [documentation](#).

```
isAlphanumeric: true, // checks for alphanumeric characters only  
isNumeric: true, // allow numbers only  
isLowercase: true, // allow lowercase only  
isUppercase: true, // allow uppercase allow
```

```
email:{
  type:DataTypes.STRING,
  unique: true,
  allowNull: false,
  validate:{
    isEmail: {msg: "It must be a valid Email address"},
  }
}
```

## Sequelize model associations

In this section, we are going to implement associations in Sequelize. You may want to revisit the basics [here](#). Sequelize supports the three model associations:

One-to-One

One-to-Many

Many-to-Many

In the model we created earlier, we have a One-to-Many association. This results in two Sequelize relationships:

```
User.hasMany(Post);
```

```
Post.BelongsTo(User);
```

Define the foreign key in the `User` and `Post` models, as shown below in the code snippets.

```
static associate({Post}) {  
  // define association here  
  this.hasMany(Post, {foreignKey: 'userId', as: 'posts' })  
}  
};
```

```
// Post Model  
class Post extends Model {  
  static associate({User}) {  
    // define association here  
    this.belongsTo(User, {foreignKey: 'userId', as: 'user' })  
  }  
};
```

The source code for the `User` and `Post` models is on [Github](#).

## App testing

We can now test our app with data. We will use [Postman](#) to make requests.

We will create an endpoint for each functionality. To implement the changes to the database, run the applications. We will not make more changes to the models. Edit `app.js`. Replace the line

```
await sequelize.sync({force: true});
```

```
await sequelize.authenticate();
```

This will ensure that our database tables are not recreated every time we run the app.

## Insert User

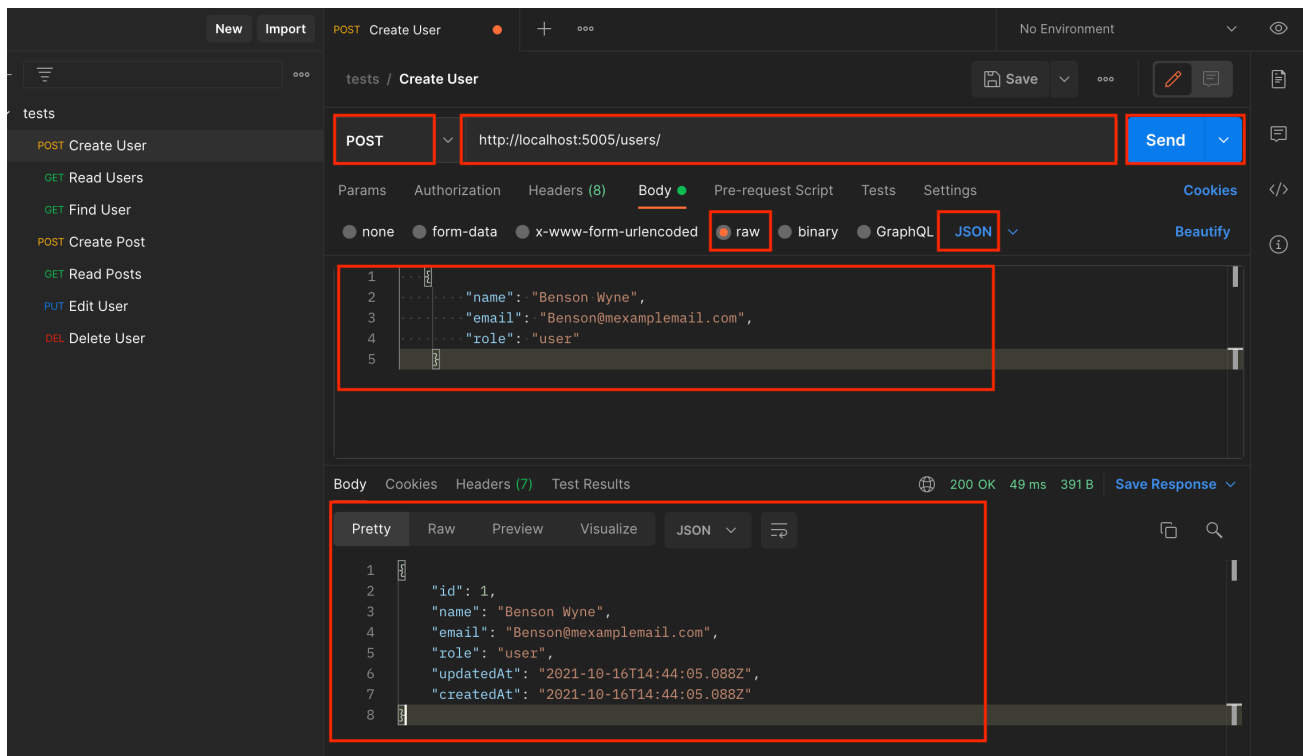
We will create an endpoint that will insert new users into the database. The endpoint takes in data in JSON format. Add the code below in `app.js` and run the app.

```
// Create new user
app.post("/users", async(req,res) =>{
  const { name, email, role} = req.body
  try{
    const user = await User.create({name, email, role});
    return res.json(user);
  }catch(err){
    return res.status(500).json(err);
  }
});
```

In Postman, create a `POST` request with the endpoint location as `http://localhost:5005/users/`. In the Body section, select `raw JSON` and insert the JSON data below.

```
{
  "name": "Benson Wyne",
```

Click send to run the request. Check the response. The expected response is as shown in the screenshot below.

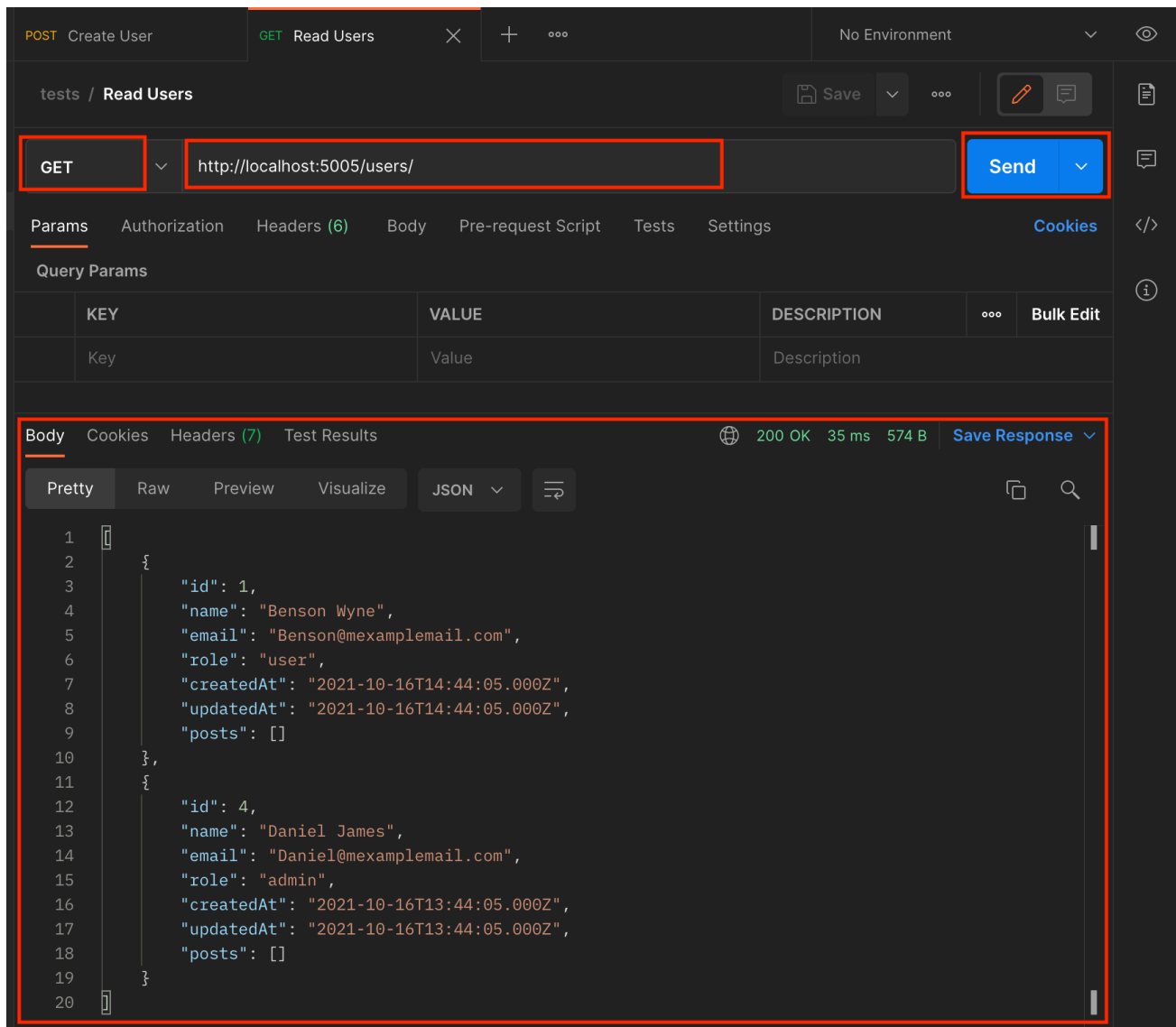


## Get users

We will create an endpoint that will return users from the `users` table. Add the code below in `app.js` and run the app.

```
//Fetch users
app.get("/users", async(req,res) =>{
  try{
    const users = await User.findAll({include:'posts'});
    return res.json(users);
  }catch(err){
    return res.status(500).json({err: "An error occurred"});
  }
});
```

In Postman, create a `GET` request with the endpoint location as `http://localhost:5005/users/`. You will get a JSON response with all the users in the database.



## Edit user

We will create an endpoint that will update the user details. The endpoint takes in data in JSON format. Add the code below in `app.js` and run the app.

```
const id = req.params.id;
const { name, email, role } = req.body;
try{
  const user = await User.findOne({
    where: {id}
  });
  user.name = name;
  user.email = email;
  user.role = role;

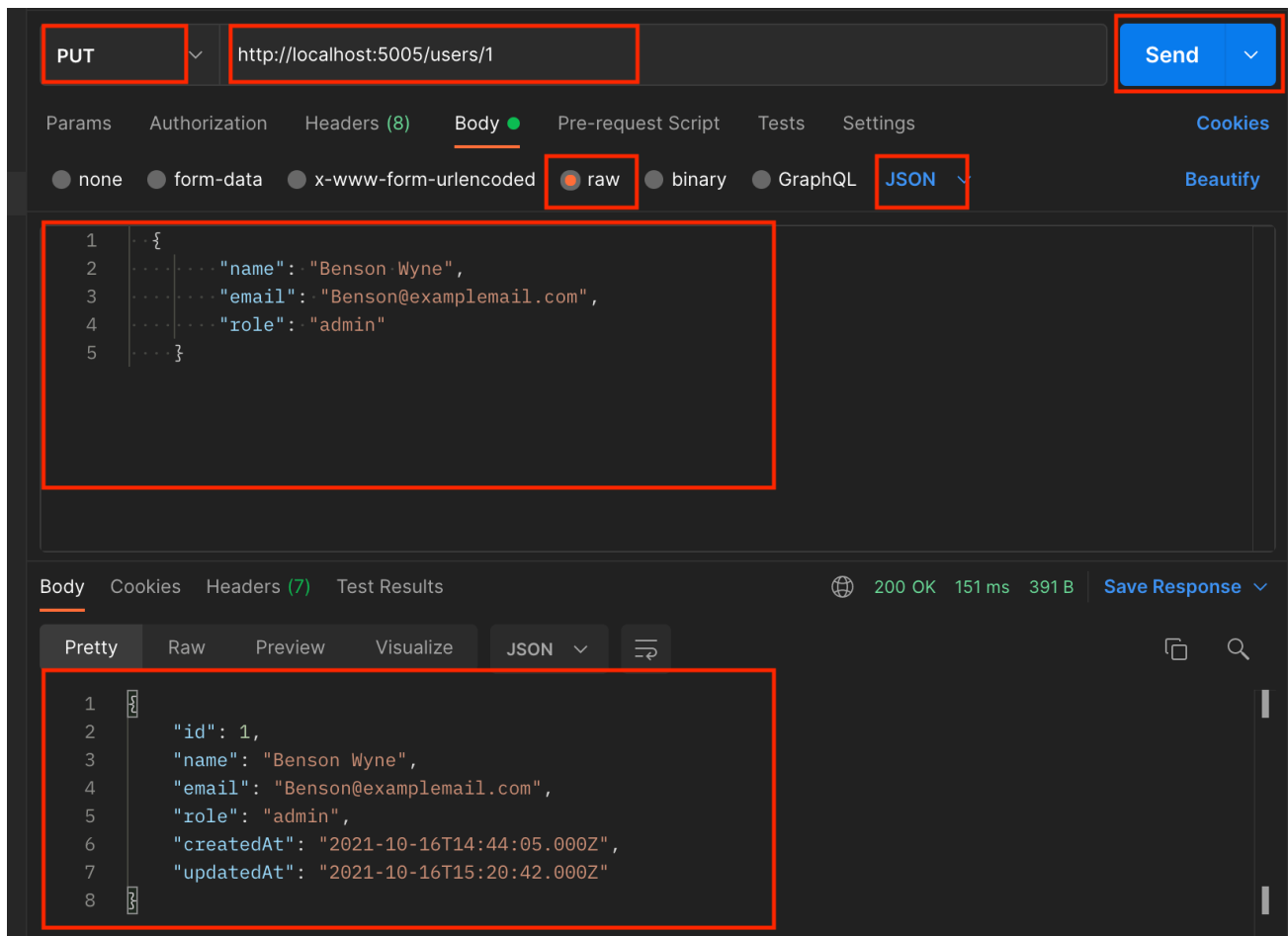
  await user.save();
  return res.json(user);

}catch(err){
  return res.status(500).json({err: "An error occurred"});
}
});
```

In Postman, create a `PUT` request with the endpoint location as `http://localhost:5005/users/1`. The number at the end of the URL represents the primary key to the record to be updated.

In the body section, select `raw JSON` and insert the JSON data with the changes you want to make to the record.

```
{
  "name": "Benson Wyne",
  "email": "Benson@exampleemail.com",
  "role": "admin"
}
```



## Create post

We will create an endpoint that will insert new posts into the database. The endpoint takes in data in JSON format. Add the code below in `app.js` and run the app.

```
// Create new post
app.post("/posts", async(req,res) =>{
  const { content, userId} = req.body
  try{
    const user = await User.findOne({
      where: {id: userId}
```

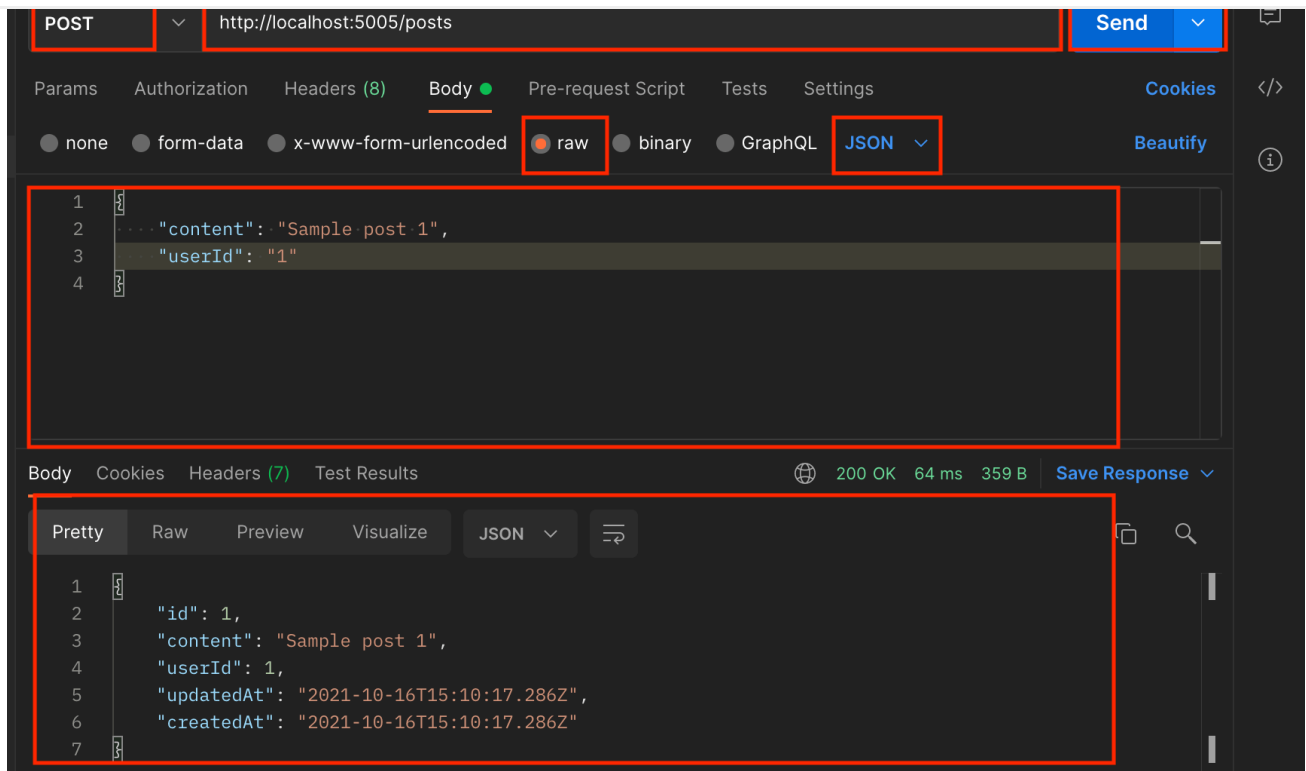


```
    }catch(err){  
        return res.status(500).json(err);  
    }  
});
```

In Postman, create a `POST` request with the endpoint location as `http://localhost:5005/posts/`. In the body section, select `raw JSON` and insert the JSON data below.

```
{  
    "content": "Sample post 1",  
    "userId": "1"  
}
```

Click send to run the request. Check the response. The expected response is as shown in the screenshot below.

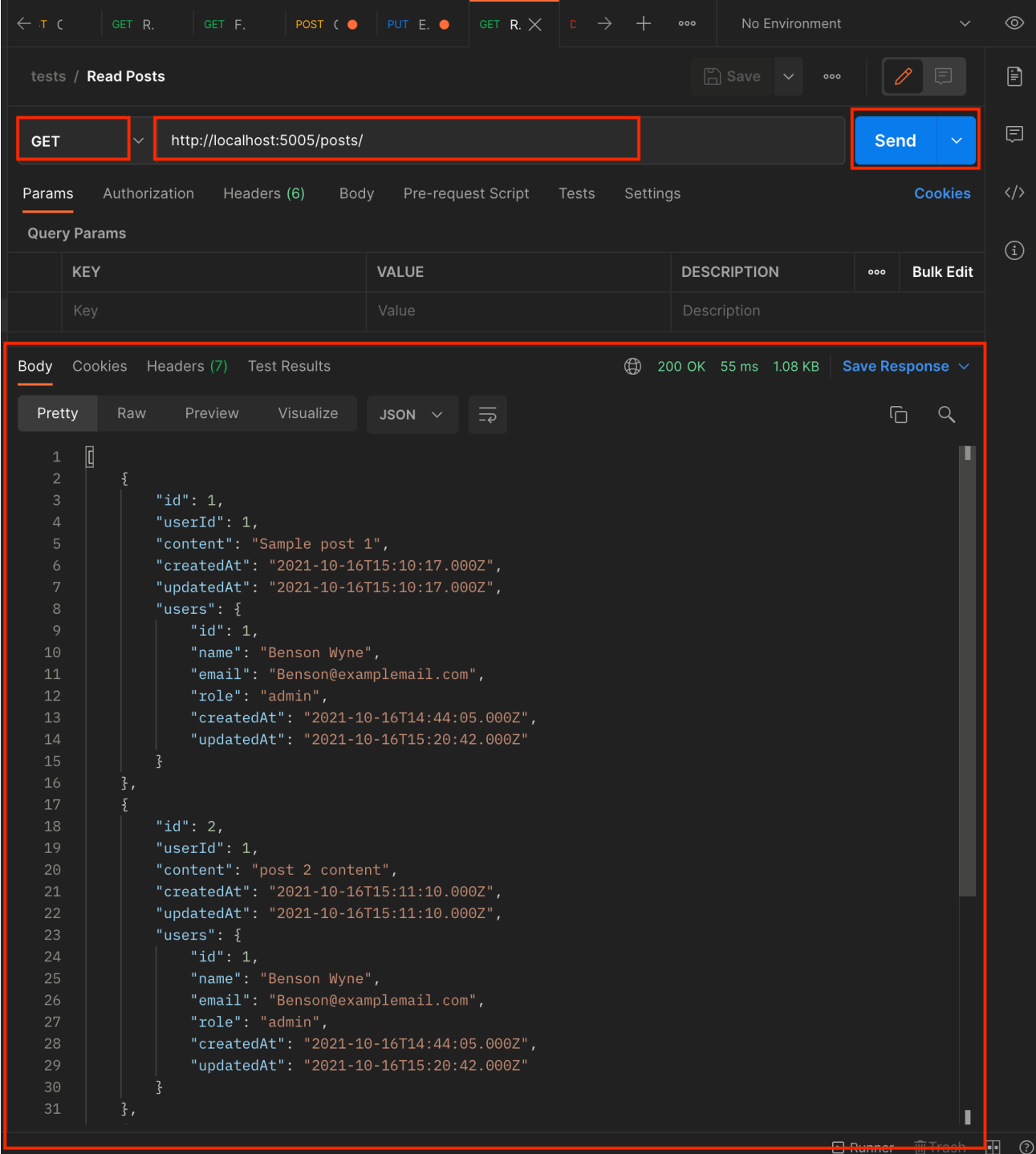


## Get posts

We will create an endpoint that will return posts from the posts table. Add the code below in `app.js` and run the app.

```
// Get posts
app.get("/posts", async(req,res) =>{
  try{
    const posts = await Post.findAll({include:'users'});
    return res.json(posts);
  }catch(err){
    return res.status(500).json(err);
  }
});
```

posts in the database.



The screenshot shows the Postman interface with a GET request to `http://localhost:5005/posts/` successfully executed. The response is a JSON array of two post objects, each containing a user object.

**Request Details:**

- Method: GET
- URL: `http://localhost:5005/posts/`
- Status: 200 OK
- Time: 55 ms
- Size: 1.08 KB

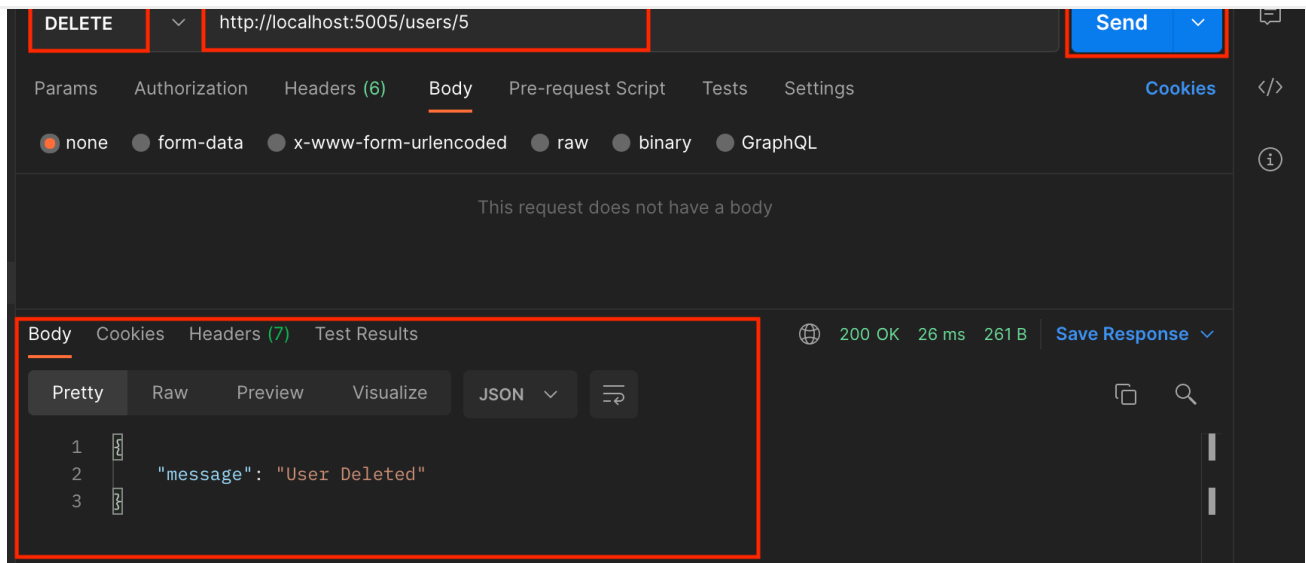
**Response Body (JSON):**

```
{
  "id": 1,
  "userId": 1,
  "content": "Sample post 1",
  "createdAt": "2021-10-16T15:10:17.000Z",
  "updatedAt": "2021-10-16T15:10:17.000Z",
  "users": {
    "id": 1,
    "name": "Benson Wyne",
    "email": "Benson@exampleemail.com",
    "role": "admin",
    "createdAt": "2021-10-16T14:44:05.000Z",
    "updatedAt": "2021-10-16T15:20:42.000Z"
  }
},
{
  "id": 2,
  "userId": 1,
  "content": "post 2 content",
  "createdAt": "2021-10-16T15:11:10.000Z",
  "updatedAt": "2021-10-16T15:11:10.000Z",
  "users": {
    "id": 1,
    "name": "Benson Wyne",
    "email": "Benson@exampleemail.com",
    "role": "admin",
    "createdAt": "2021-10-16T14:44:05.000Z",
    "updatedAt": "2021-10-16T15:20:42.000Z"
  }
}
```

## Delete user

```
//Delete user
app.delete("/users/:id", async(req,res) =>{
  const id = req.params.id;
  try{
    const user = await User.findOne({
      where: {id}
    });
    await user.destroy();
    return res.json({message: "User Deleted"});
  }catch(err){
    return res.status(500).json({err: "An error occurred"});
  }
});
```

In Postman, create a `DELETE` request with the endpoint location as `http://localhost:5005/users/5`. The number at the end of the URL represents the primary key to the record to be deleted.



The final source code for `app.js` is available on [Github](#).

## Conclusion

We have learned how Sequelize models work. Learn more on how to move your project to production using [Sequelize migrations](#) and [seeders](#). Seeders Allow the creation of dummy data on the database. The source code for the project is available on [Github](#).

Happy coding.

Peer Review Contributions by: [Miller Juma](#)

### 0 Comments

The EngEd community is subject to Section's [moderation policy](#).

Be the first to comment...

## Similar Articles



Languages

### How to Create a Basic Browser GraphQL Tool using React.js, Node.js and MongoDB

[Read More](#)

## STIR FRAMEWORK

Languages

### Stir Framework in Action in a Spring Web App

[Read More](#)



## React Hook Form

Languages

### How to Create a Reusable React Form component

[Read More](#)

[EngEd Author Bio](#)



## Benson Kariuki

Benson Kariuki is a graduate computer science student. He is a passionate and solution-oriented computer scientist. His interests are Web Development with WordPress, Big Data, and Machine Learning.

[View author's full profile](#) →

Deploy your apps to a supercloud in a few clicks

Try It For Free

### Company

About

Careers

Legals

### Resources

Blog

Content Library

Engineering Education

### Support

Docs







---

[Release Notes](#)[Platform Status](#)[Contact Us](#)

Section supports many open source projects including:

 [varnish cache  
logo](#) [cloud native computing  
foundation logo](#) [the linux  
foundation logo](#) [Istio  
logo](#)[Privacy Policy](#)   [Terms of Service](#)