**School of Computing**

**ST0503 Back End Web Development**

**Practical 6**

**RESTFul APIs with Express and MySQL**

**Objectives:**

After completing this lab, you should be able to:

* Understand when to use Get, Post, Put or Delete
* Use NodeMon
* Test API endpoints with POSTMAN

## Table of content

1. Introduction to REST APIs
2. Automatically restart server upon changes with Nodemon
3. Handling HTTP requests with Express

## 1. Introduction to REST APIs

### 1.1. What is an API?

An API (Application Programming Interface) defines a set of operations that allow applications to communicate with it. Examples of APIs include web APIs and software libraries (like those on NPM).

We will focus on web APIs in this practical. Our web API should allow us to manipulate and retrieve data on our MySQL database through HTTP requests.

### 1.2. What are REST APIs?

REST is an architectural style we will be using to design our API.

Data (resources) in the database is exposed through the endpoints (URLs) of our API. Resources can be created, read, updated, and deleted (CRUD) through the endpoints.

You can read more about REST here:

https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design

### 1.3. HTTP Methods

CRUD (create, read, update, delete) operations correspond to HTTP methods that our browsers can send.

* A GET request is used to retrieve (read) resources.
* A POST request is used to create resources.
* A PUT request is used to update resources.
* A DELETE request is used to delete resources.

Because we can only manually send GET requests to endpoints through our browser, we will have to use a program called Postman that can fire all sorts of HTTP methods to our API.

**Download Postman here**:

https://www.getpostman.com/downloads/

### 1.4. The Express framework

Express.js is a Node.js web framework used to create REST APIs. We can use Express to easily create endpoints for our server to handle requests and serve resources.

Install express in our friendbook project with the following command:

>> npm install --save express

### 1.5. HTTP response status codes

When we send a HTTP request, the response will contain a status code that tells us whether our operation was successful or not.

Here are the important one’s that will be used in the practical:

#### 2xx Success

**200 OK**

Standard response when a request has succeeded. It used when a GET request succeeds.

**201 Created**

Request has succeeded and a new resource has been created. It is used when a POST request succeeds.

**204 No Content**

Same as 200 OK, but used when there is no content to be sent back. Typically used when PUT or DELETE succeeds but there is nothing to be sent back.

#### 4xx Client errors

**400 Bad Request**

The server cannot processs the request due to invalid syntax. The client should not repeat the same request.

**404 Not Found**

Server cannot find the requested resource. It used when we request for a nonexistent resource.

#### 5xx Server errors

**500 Internal Server Error**

The server has encountered a situation it does not know how to handle. Used as a generic error message for handling unexpected errors.

You can find the full list of status codes here:

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

## 2. Automatically restart server upon changes with Nodemon

Remember that we had to manually restart our server when we made changes to the code for them to take effect?

nodemon is a command-line interface (CLI) that automatically detects changes to the codebase and restarts the server for us.

Install nodemon with the following command:

>> npm install -g nodemon

To start our server with nodemon, run the following command:

nodemon server.js

The server should start.

Try making a change to one of the files in the friendbook project. You should see the server automatically restart.

Let’s save it as an NPM script. Make the follow changes to the scripts section of package.json:

{  
 ...  
 "scripts": {  
 "start-dev": "nodemon server.js",  
 "test": "echo \"Error: no test specified\" && exit 1"  
 },  
 ...  
}

Run the following command to start the server:

>> npm run start-dev

We will use this command from now on to start our server instead of node server.js.

## 3. Handling requests with Express

### 3.1. User Endpoints

Here are the endpoints we will create for the user endpoint:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Endpoint | HTTP Method | Description | Succeed Response | Error Response |
| /users/ | GET | Returns all users. | 200 | 500 |
| /users/ | POST | Add a new user. | 201 | 500 |
| /users/:userID/ | GET | Returns user with the given id. | 200 | If no user with the id: 404.Otherwise 500. |
| /users/:userID/ | PUT | Edits a user with the given id. | 204 | 500 |
| /users/:userID/friends | GET | Returns the friends of the user of the given id. | 200 | 500 |
| /users/:userID/friends/:friendID | POST | Create friendship between users with ids id and friendID | 201 | If userID === friendID, send a 400.If friendships already exists (error.code === "ER\_DUP\_ENTRY"), send a 201.If either one or both of the ids are non-existent (error.code === "ER\_NO\_REFERENCED\_ROW\_2"), send a 400.Otherwise, 500 |
| /users/:userID/friends/:friendID | DELETE | Destroys a friendship between user of id id and friendID | 204 | 500 |

### 3.2. Initializing Our Express Server

An Express “app” has methods like app.get(), app.post(), app.put(), app.delete() to handle GET, POST, PUT, DELETE requests respectively.

Refer to the Express documention on the “app” object and it’s methods:

https://expressjs.com/en/4x/api.html#app

Create a file named **app.js** in a new **controller** directory with the following inside:

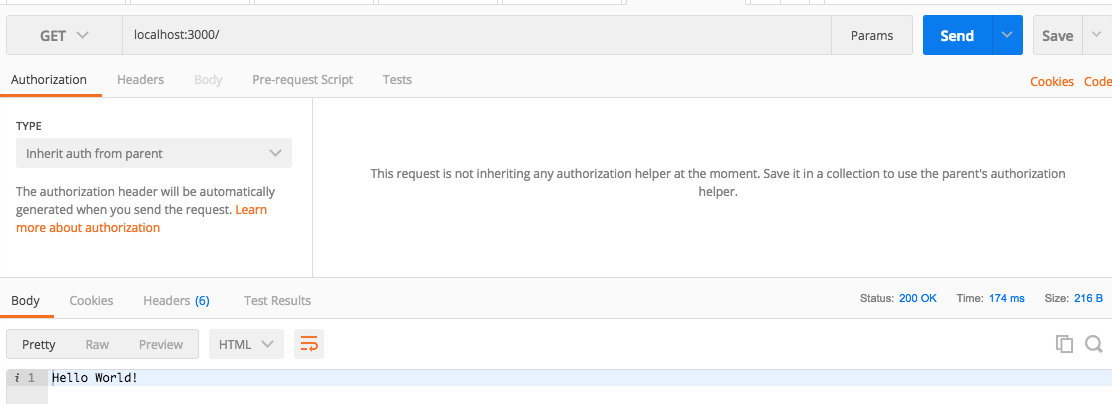
const express = require("express");  
const app = express();  
  
app.get("/", (req, res) => {  
 res.send("Hello World!");  
});  
  
module.exports = app;

Import the app object in the root server.js and use the app.listen() method to listen for requests on port 3000:

const app = require("./controller/app");  
  
const PORT = 3000;  
app.listen(PORT, () => {  
 console.log(`Server started on port ${PORT}`);  
});

Our Express server is now listening for requests made through port 3000 on localhost.

Make a GET request made to localhost:3000/ through Postman. "Hello World!" should be send back.



### 3.3. GET /users/ endpoint

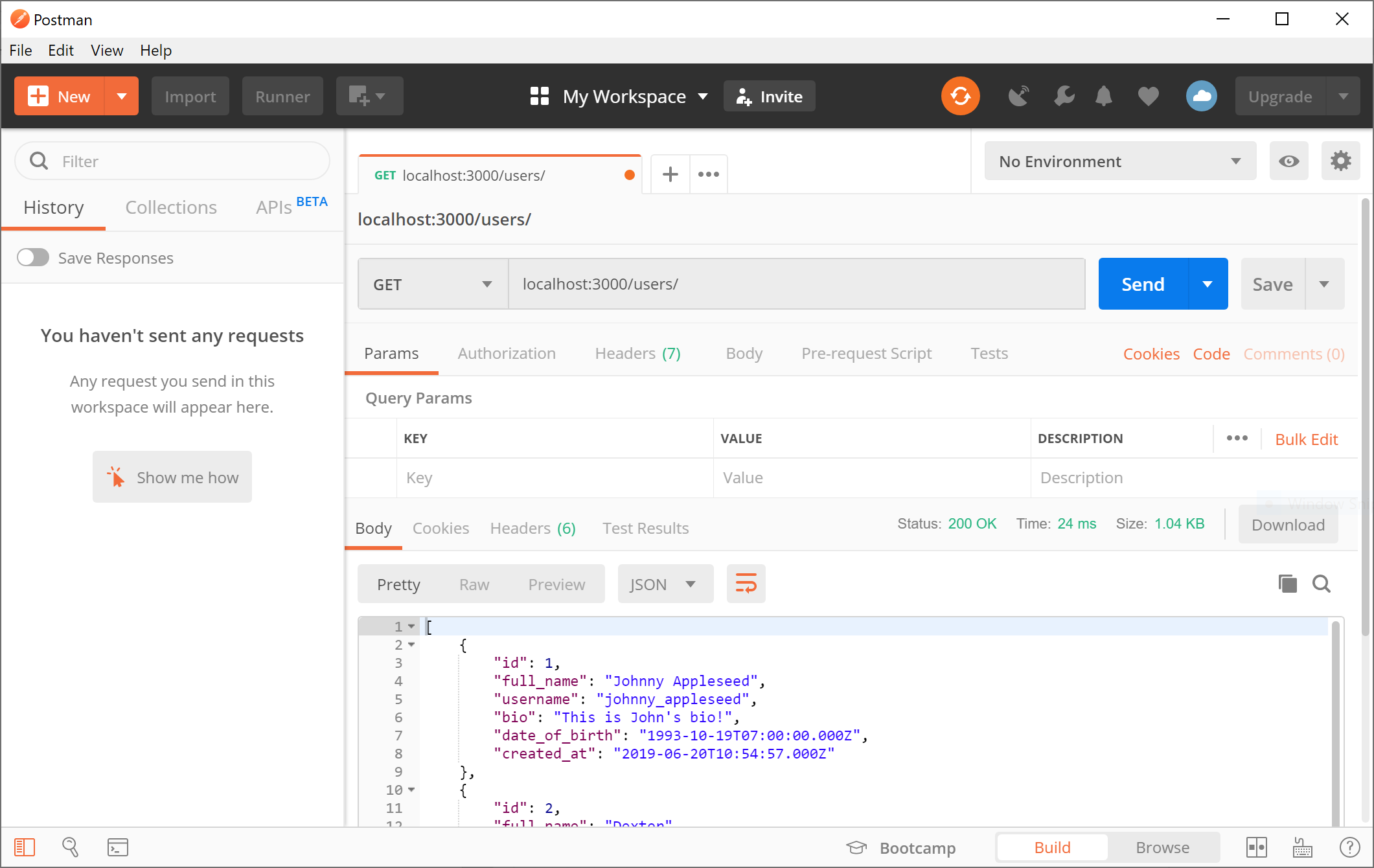
Create the route handler and use the User.findAll() method to retrieve all the users:

const express = require("express");  
const app = express();  
  
// import the User model  
const User = require("../models/User");  
  
app.get("/", (req, res) => {  
 res.send("Hello World!");  
});  
  
app.get("/users/", (req, res, next) => {  
 User.findAll((error, users) => {  
 if (error) {  
 console.log(error);  
 res.status(500).send();  
 };  
 res.status(200).send(users);  
 });  
});  
  
module.exports = app;

When a client sends a GET request to localhost:3000/users/, the server should send back all the users in the database using the User model with a status code of 200 OK.

In the event an error occurs and we cannot identify and handle it, we send a status code of 500 Internal Server Error.

Send a GET request to localhost:3000/users/ through Postman. You should see all the users in the database in JSON format:



Notice the similarity between JSON and a object literal in Javascript?

JSON (JavaScript Object Notation) is a data-interchange format used for transmitting information between the frontend and the backend of a web application. While it looks similar to an object literal in Javascript, they are different things. JSON is received in a string format and has to be parsed to be read from in our application as an object literal.

### 3.4. GET /users/:userID endpoint

Similar to how functions can have parameters, routes can be dynamic and have parameters.

We will now implement an endpoint to retrieve information of a single user given his id.

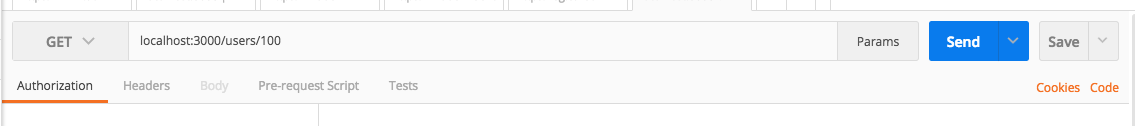
A GET request to localhost:3000/users/5 should retrieve a user with id 5. If the user does not exist, send back a status code of 404 Not Found.

#### 3.4.1. Add the request handler for GET requests to /users/:userID to controller.js:

app.get("/users/:userID/", (req, res, next) => {  
 console.log(req.params);  
  
 // this is in string format instead of an integer like it is in the database.  
 console.log(typeof(req.params.userID));  
  
 // parse it to an integer  
 const userID = parseInt(req.params.userID)  
 console.log(typeof(userID));  
});

We can extract the userID through req.params. req.params is an object that maps the parameter name to the supplied value for that request. We have to manually parse the userID to an integer.

#### 3.4.2. Make a GET request to localhost:3000/users/100/ through Postman:



You should see the following output in the terminal:

{ userID: '100' }  
string  
number

You can read more about req.params here:

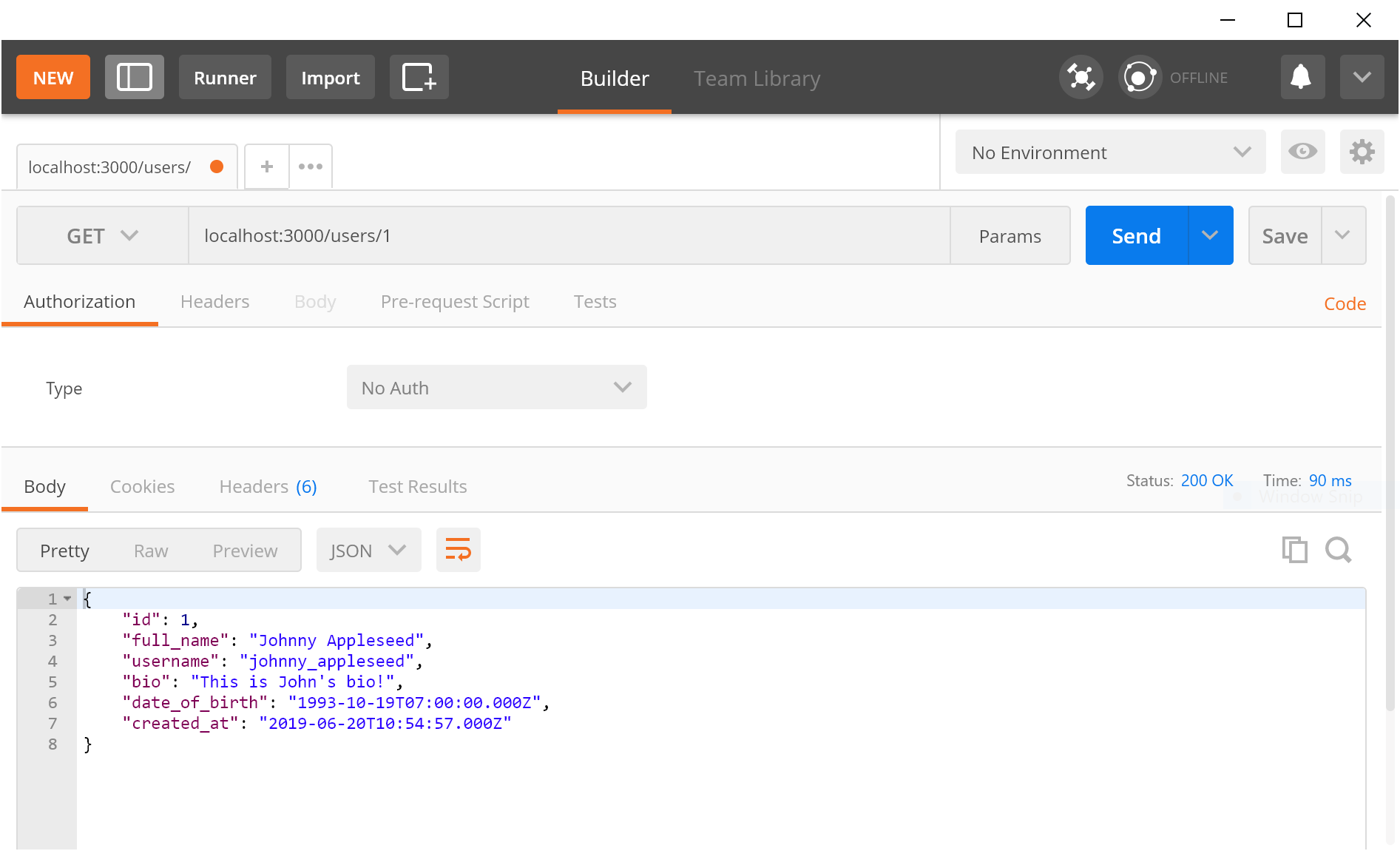
https://expressjs.com/en/4x/api.html#req.params

#### 3.4.3. Use the User.findByID() method we wrote in the previous practical to retrieve a user given a user id:

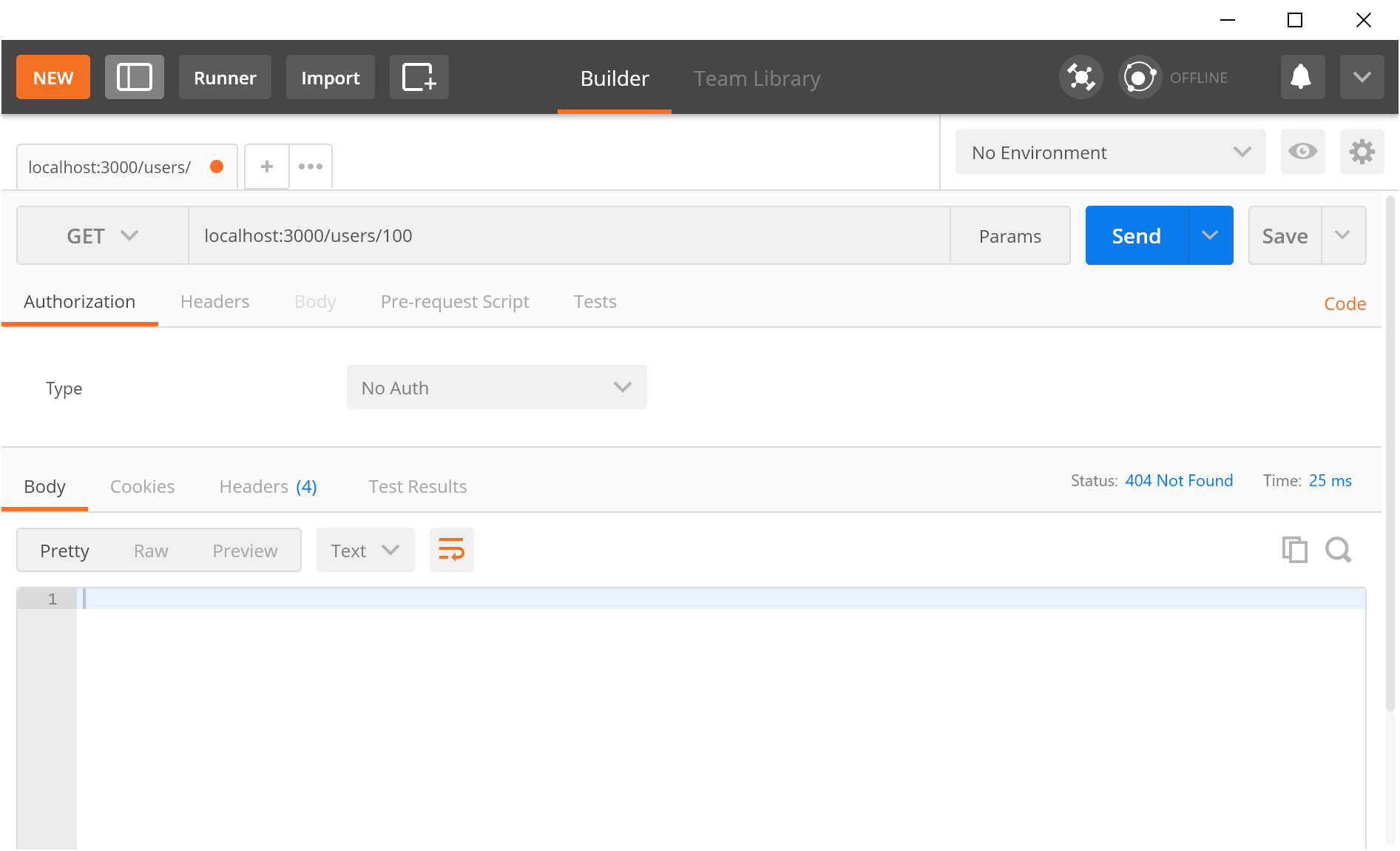
app.get("/users/:userID/", (req, res, next) => {  
 const userID = parseInt(req.params.userID);  
 // if userID is not a number, send a 400.  
 if (isNaN(userID)) {  
 res.status(400).send();  
 return;  
 }  
  
 User.findByID(userID, (error, user) => {  
 if (error) {  
 res.status(500).send();  
 return;  
 };  
  
 // send a 404 if user is not found.  
 if (user === null) {  
 res.status(404).send();  
 return;  
 };  
 res.status(200).send(user);  
 });  
});

Test out the endpoint by finding a user of id 1 (which exists):

You should see the user “Johnny Appleseed”. The response status code should be 200 OK:

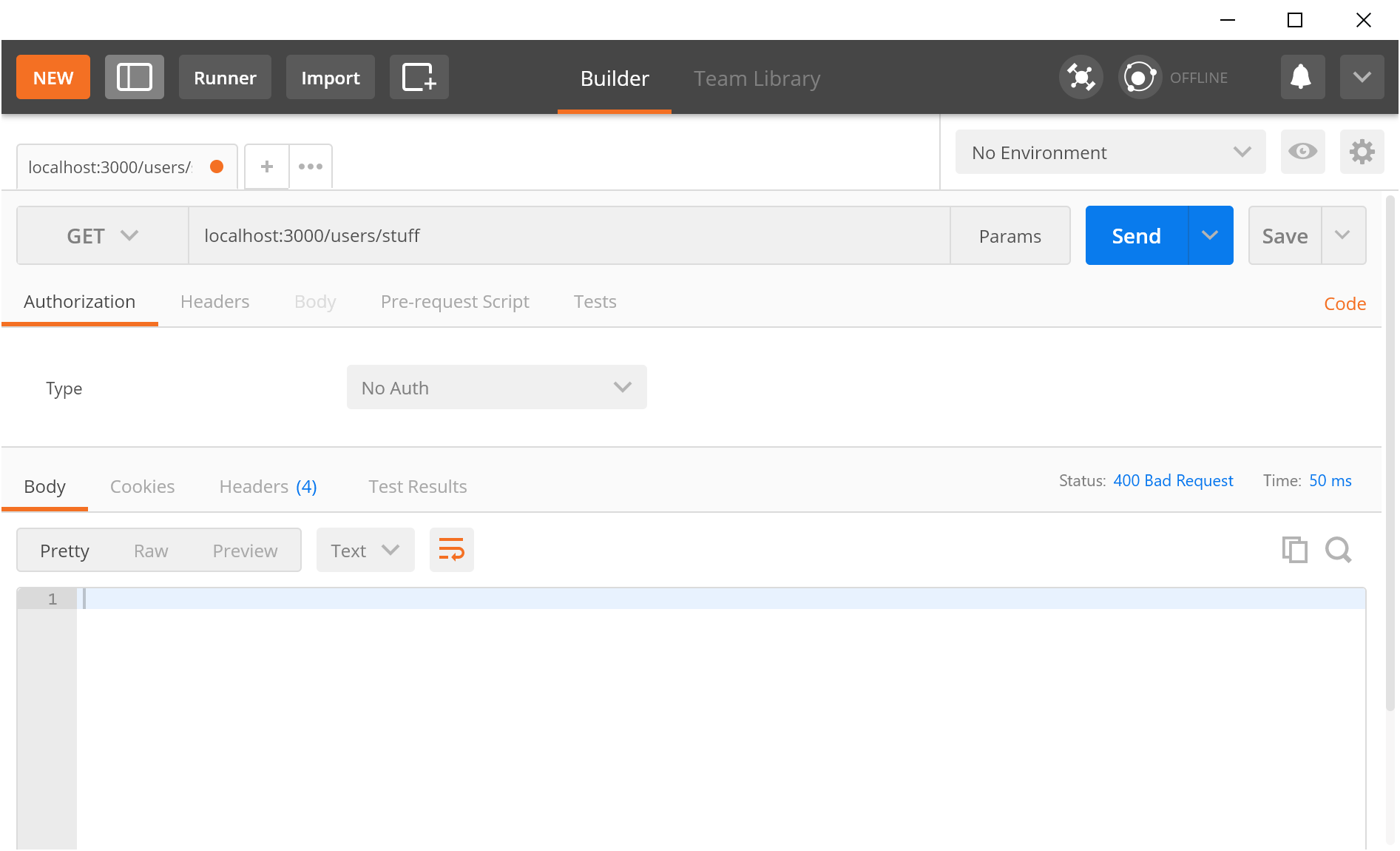


Let’s test if a 404 Not Found is returned when a non-existent user id is provided. Provide an id that no user in the database has:



You should see the status of 404 Not Found.

Provide a string that cannot be converted to an integer for the user id:



The response should have a status code of 400 Bad Request.

### 3.5. POST /users/ endpoint

We will create the endpoint POST /users/ for creating users using app.post():

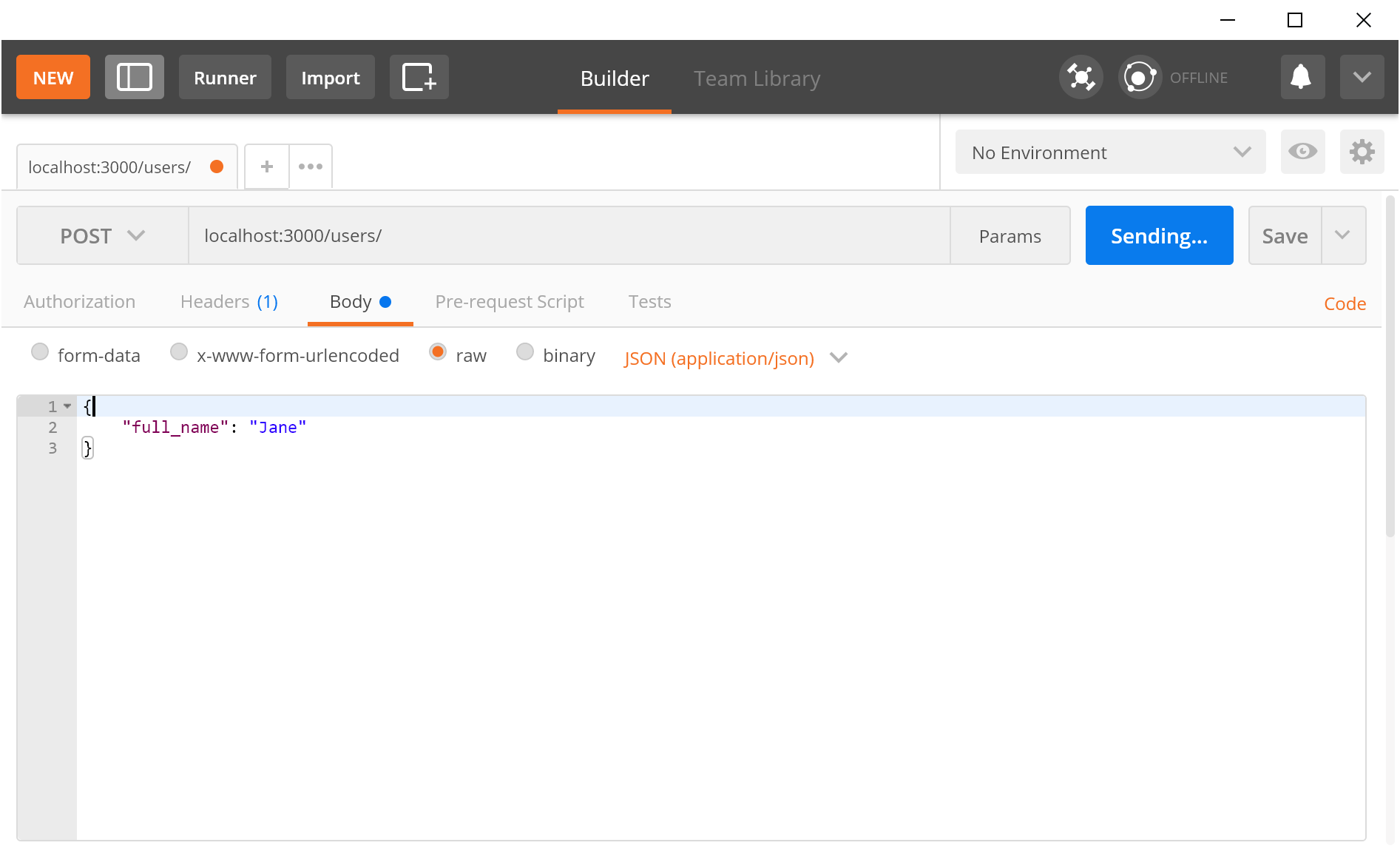
Three things to note about POST requests:

1. POST requests are used to create a new resource.
2. The created resource should be sent back with a status code of 201 Created if it’s a success.
3. While we could put all the new user information in the parameters, it is a better practice to put it in the request body in the form of JSON.

#### 3.5.1. Add the route handler to controller.js:

app.post("/users/", (req, res, next) => {  
 // we can access the request body through req.body  
 console.log(req.body);  
});

Make a POST request to localhost:3000/users/ from Postman with the following body:



Take a look at the printed output in the terminal and you should see undefined.

Strange. Read the Express documentation on req.body:

https://expressjs.com/en/api.html#req.body

According to the docs, We need a body-parsing “middleware”. We need to parse the body because it is in JSON format and our Node.js application works with normal Javascript objects instead.

An Express middleware is a function that performs actions before our route handlers are called. The body-parser middleware parses the request body from JSON to Javascript object and stores it in req.body.

#### 3.5.2. Kill the Node.js server with CTRL-C. Then install body-parser and start the server again:

npm install --save body-parser  
...  
...  
...  
npm run start-dev

#### 3.5.3. Mounting the body-parser middleware.

In app.js, import body-parser and use it with app.use():

const express = require("express");  
const app = express();  
  
const User = require("../models/User");  
  
// import body-parser middleware  
const bodyParser = require("body-parser");  
  
// use the middleware  
app.use(bodyParser.json());  
  
...

Make sure that the middleware is mounted before all the route handlers.

#### 3.5.5. Make the same request from Postman as before. You should see the request body being printed.

#### 3.5.6 Insert user into database using the User.insert() method

Use the User.insert() method. We assume that the request body will contain the full\_name, username, bio, and date\_of\_birth attributes:

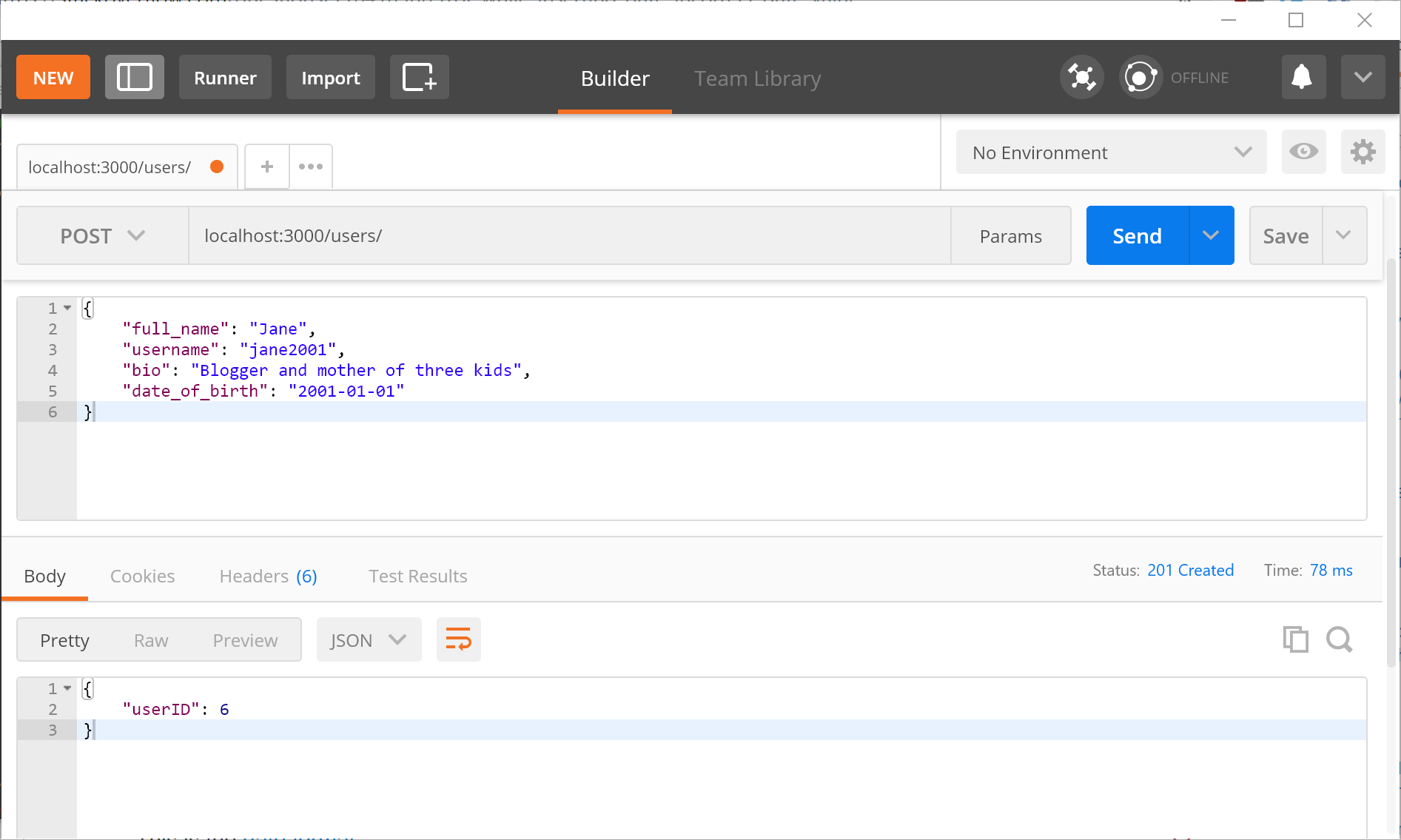
app.post("/users/", (req, res, next) => {  
 User.insert(req.body, (error, userID) => {  
 if (error) {  
 console.log(error);  
 res.status(500).send();  
 return;  
 };  
 res.status(201).send({  
 userID  
 });  
 });  
});

We send a 201 Created instead of a 200 OK when a resource is created.

The userID is also sent back as a response so that we can query for the newly inserted user by sending a GET request to localhost:3000/users/:userID.

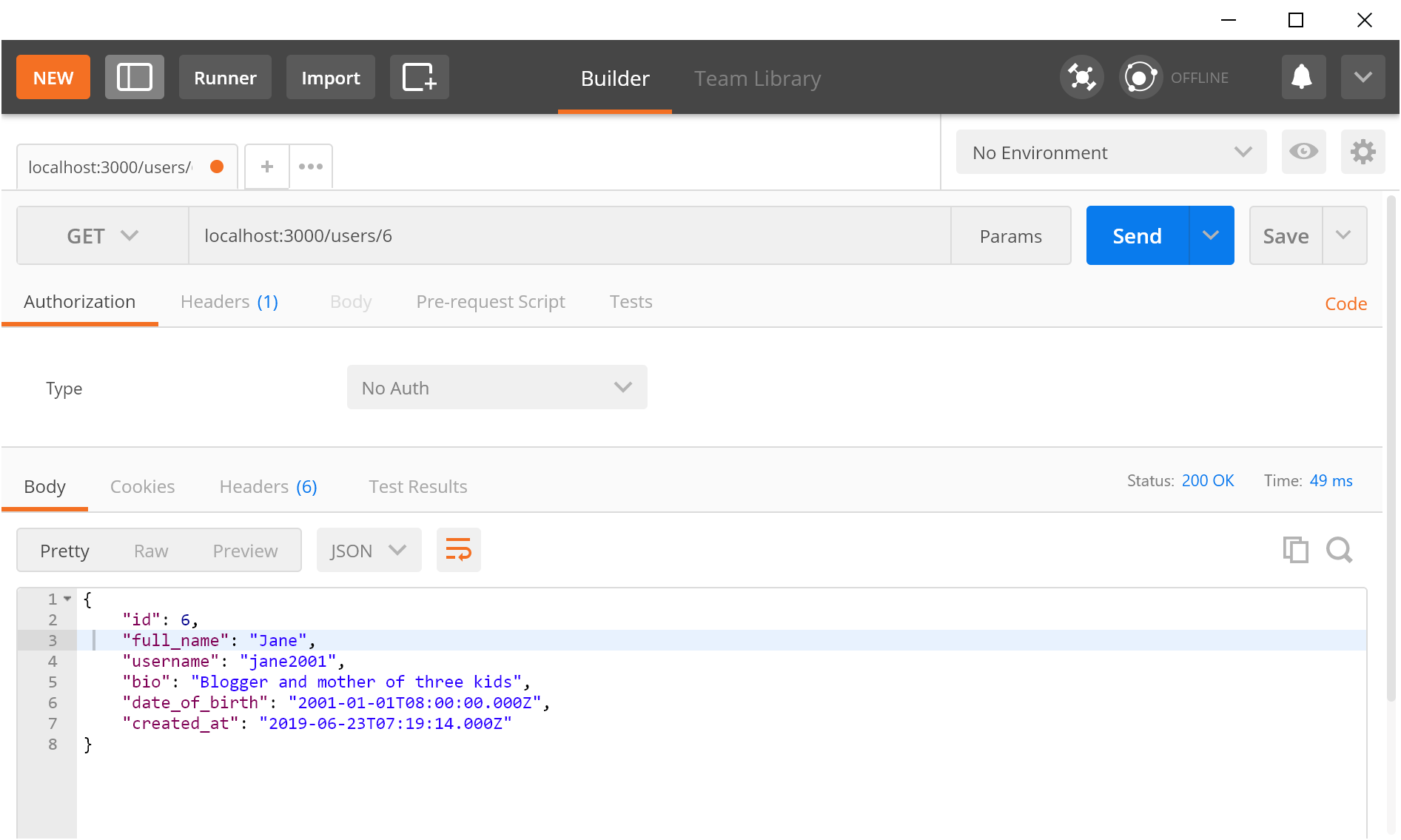
#### 3.5.7. Testing the new endpoint

Test out the POST /users/ endpoint on Postman:



You should see the inserted user’s id as a response.

Let’s query for the new user with the GET /users/:userID endpoint:



### 3.6. PUT /users/:userID

Here are some things to note about PUT requests:

1. PUT is used to update a resource.
2. PUT only allows full replacement of the resource. You must provide all the attributes to be changed or kept.
3. A 204 No Content is sent back if the request was successful. The body of a 204 should be empty.

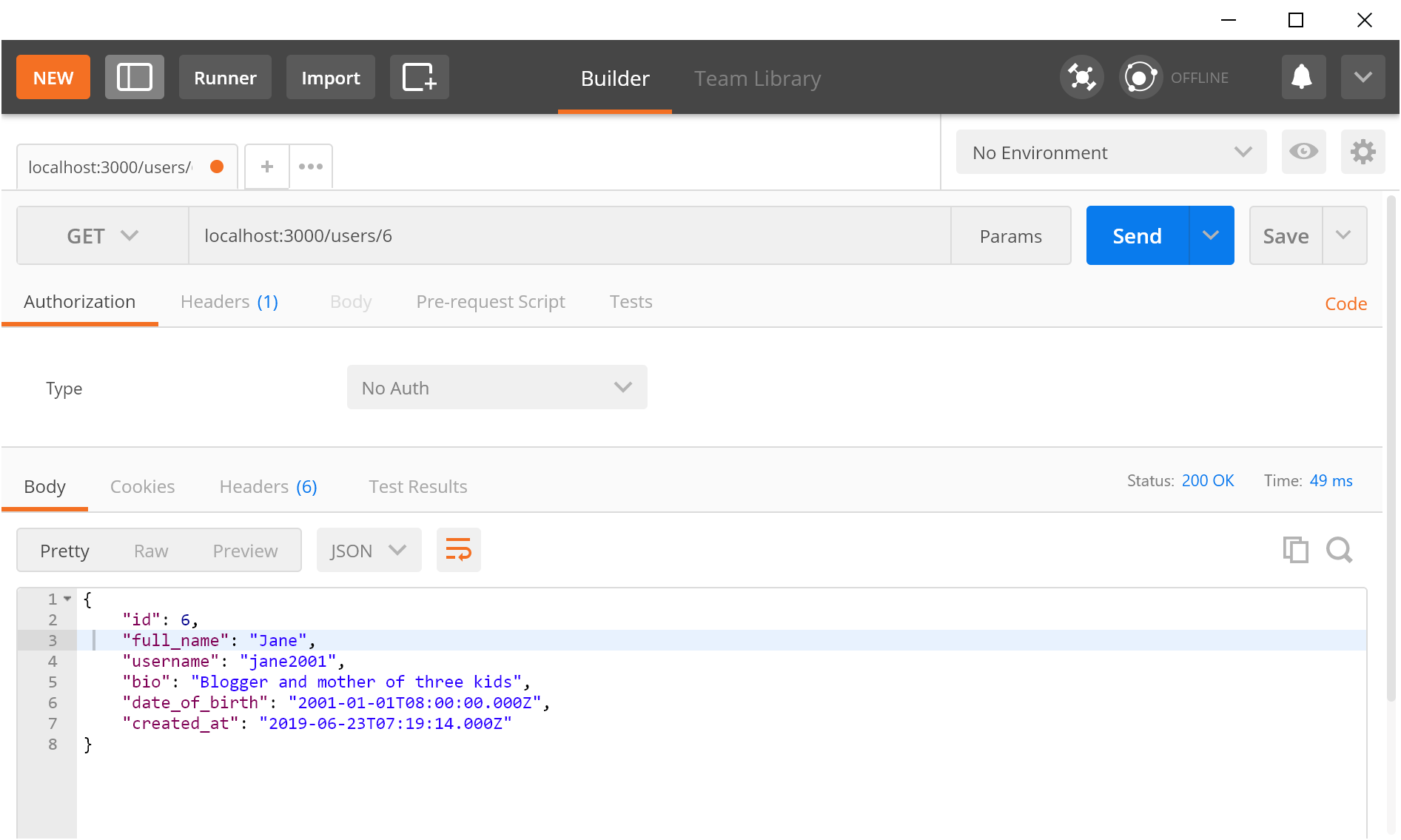
#### 3.6.1. Add the route handler to controller.js:

app.put("/users/:userID/", (req, res, next) => {  
 const userID = parseInt(req.params.userID);  
 if (isNaN(userID)) {  
 res.status(400).send();  
 return;  
 }  
  
 User.edit(userID, req.body, (error) => {  
 if (error) {  
 console.log(error);  
 res.status(500).send();  
 return;  
 };  
 res.status(204).send();  
 });  
});

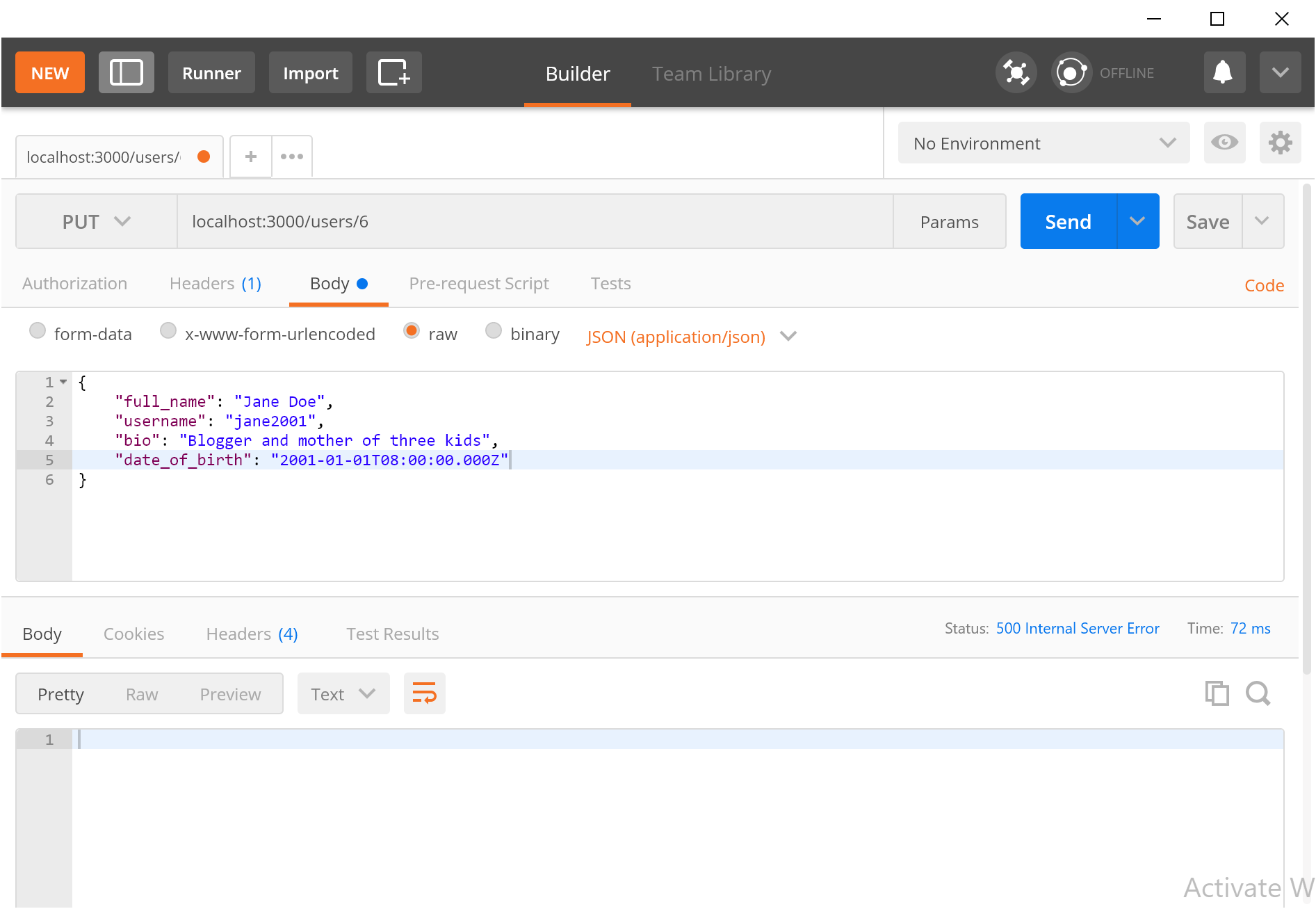
We assume that req.body contains all four attributes. If they are not provided then a 500 Internal Server Error code will be sent. We will address this in a future practical with input validation.

#### 3.7.2. Test out the endpoint

Let’s edit user with id of 6’s full\_name from “Jane” to “Jane Doe”. Since you must provide all the attributes of the resource, we must first do a GET to find the user:

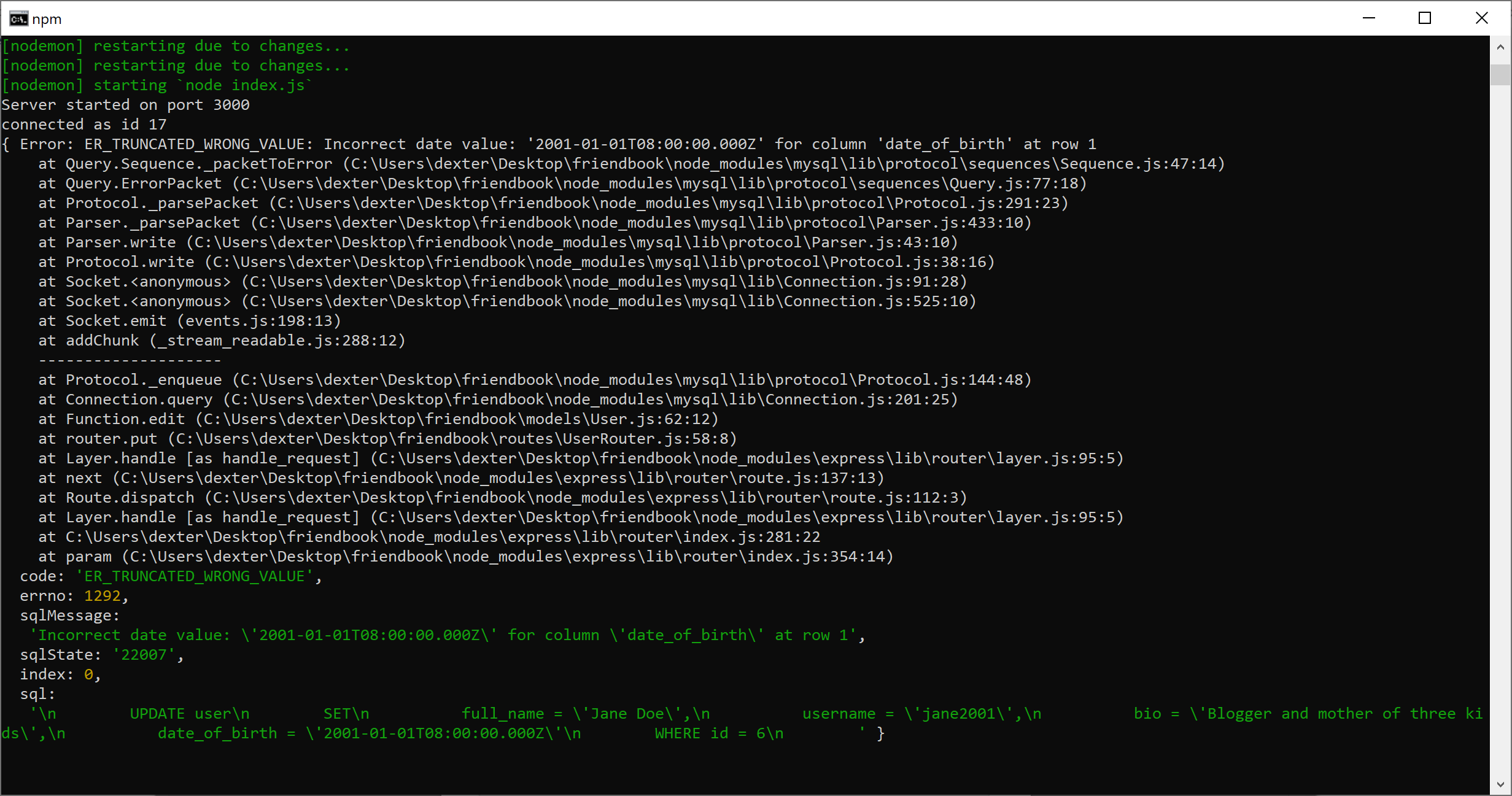


Make a PUT request to /users/:userID/ with the returned JSON in the previous request. Omit the id and created\_at, and make sure to provide full\_name of “Jane Doe”:



Ensure that the “raw” option is selected, and JSON (application/json) is selected from the dropdown menu.

Instead of getting a 200 OK, we receive a 500 Internal Server Error instead. Let’s inspect the error in the terminal:



It looks like something is wrong with the format of date\_of\_birth, which is weird, considering we copied the result of the GET request.

As it turns out, the mysql library automatically casts DATE into a Javascript Date object, which has a timestamp format and it not valid when inserted.

We want to retain the string format of DATE types. According to the mysql documentation, this is can done using by setting the dateStrings option in the connection options to true:

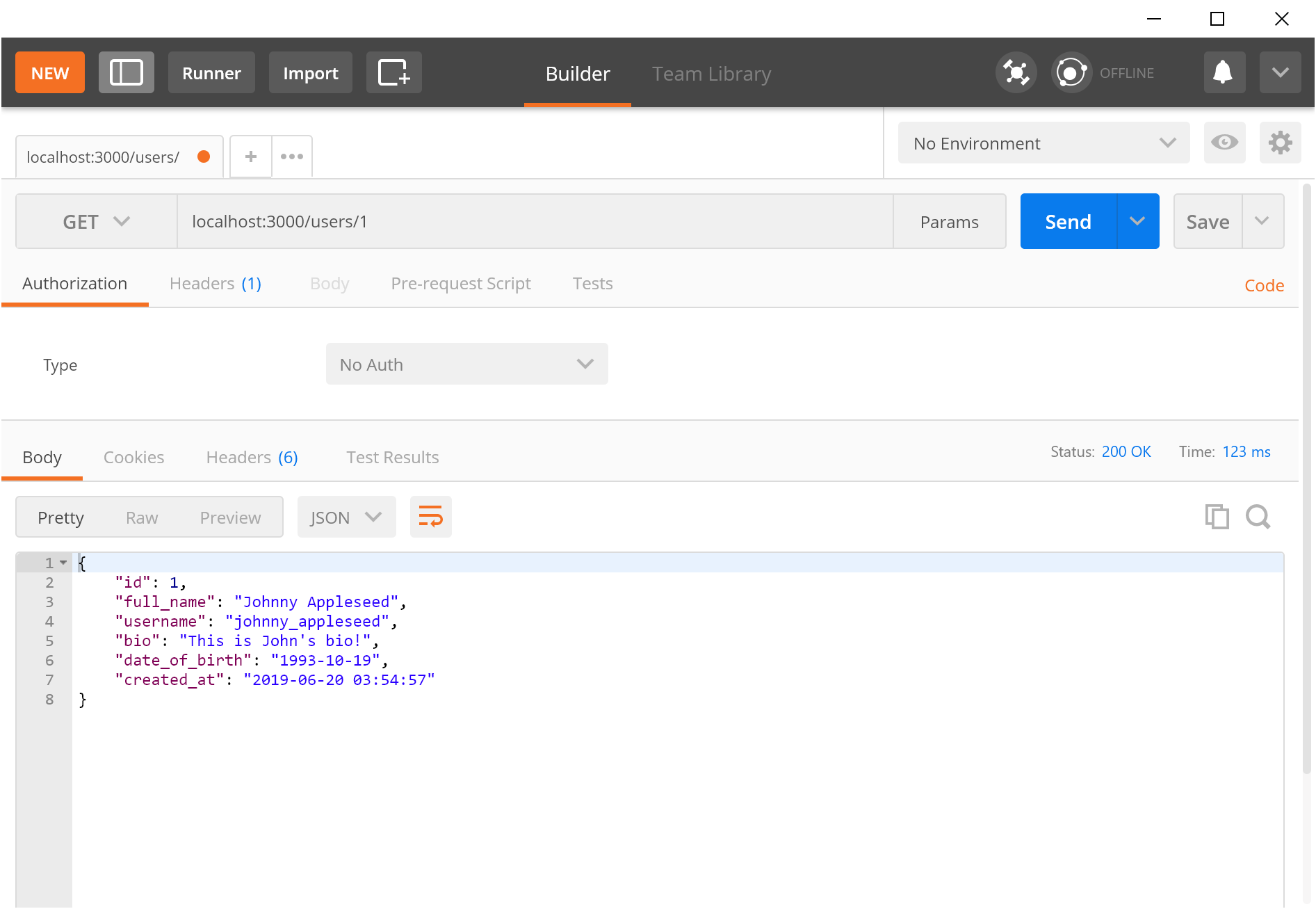
dateStrings: Force date types (TIMESTAMP, DATETIME, DATE) to be returned as strings rather than inflated into JavaScript Date objects. Can be true/false or an array of type names to keep as strings. (Default: false)

From: https://github.com/mysqljs/mysql#connection-options

Add the dateStrings option in the databaseConfig.js file:

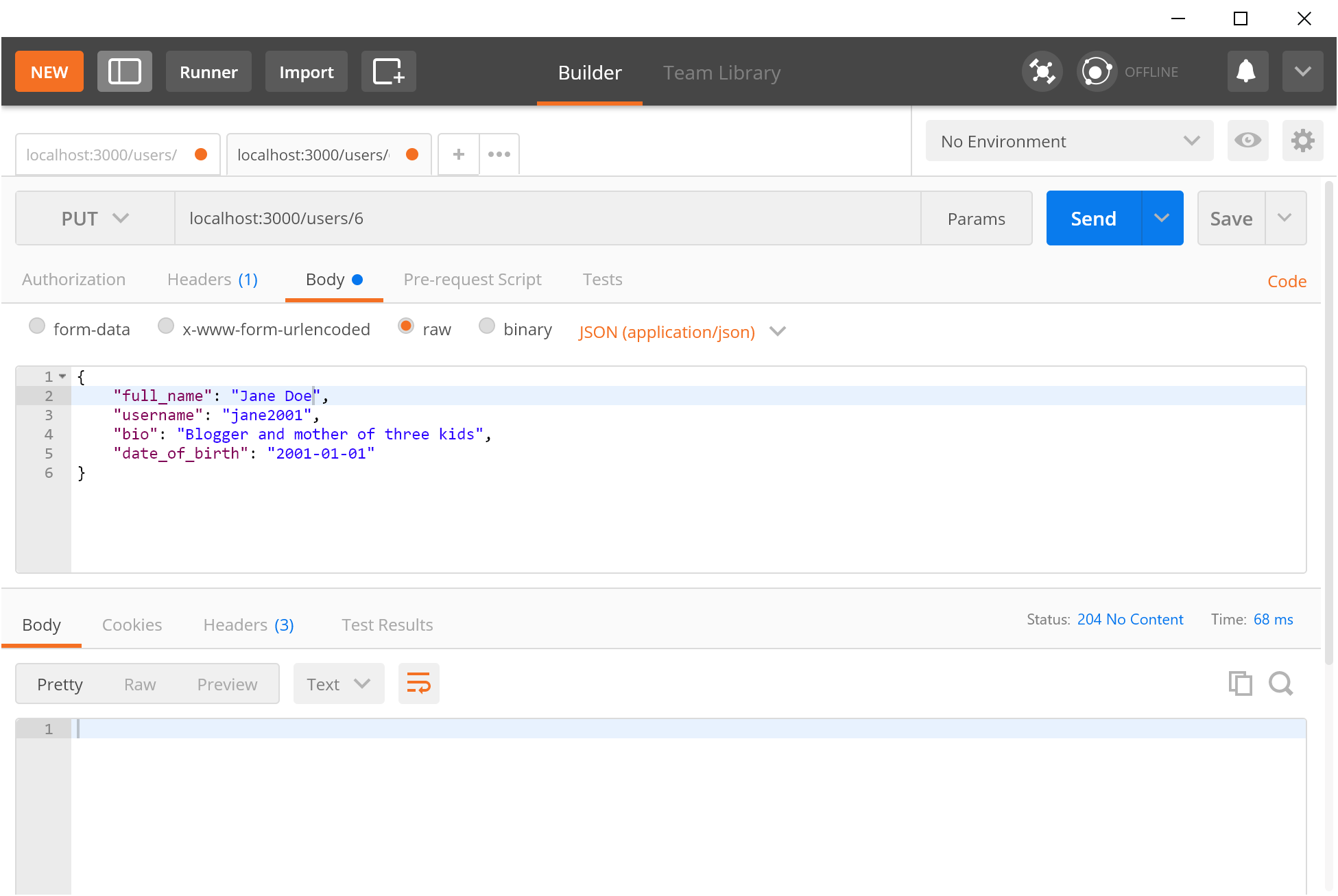
const mysql = require("mysql");  
const connection = mysql.createConnection({  
 host: 'localhost',  
 port: 3306,  
 user: 'root',  
 password: 'password',  
 database: 'friendbook',  
 // retain DATE as a string  
 dateStrings: true  
});  
  
...

Let’s query for Johnny Appleseed again:



As you can see, we no longer get a timestamp format as it is not a Javascript Date object, but a string DATE that the database returns.

Perform the PUT request again with the correct date. You should receive a 204 No Content to indicate a success:



Inspect the database to ensure that the full\_name was changed.

### 3.8. Implementing the rest of the User endpoints

Refer to the table in Section 3.1 and implement the remaining three endpoints that have not been covered.

### 3.9. Implementing the Post endpoints

As an exercise left to the readers, implement the following endpoints:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Endpoint | HTTP Method | Description | Succeed Response | Error Response |
| /posts/ | GET | Returns all posts. | 200 | 500 |
| /posts/ | POST | Add a new user. | 201 | 500 |
| /posts/:postID/ | GET | Returns post with the given id. | 200 | 404 if cannot find post.500 otherwise. |
| /posts/:postID/ | DELETE | Deletes a post with the given id. | 204 | 500 |
| /posts/:postID/ | PUT | Edits a post with the given id. Only allow text\_body to be edited. | 204 | 500 |
| /posts/:postID/likers/:likerID | POST | User with id likerID likes post with postID. | 201 | 500. If user already liked (error.code === "ER\_DUP\_ENTRY"), then send a 201. |
| /posts/:postID/likers/:likerID | DELETE | Unlike the post | 204 | 500 |
| /users/:userID/posts | GET | Returns all posts of user with id userID. | 200 | 500 |
| /posts/:postID/likers | GET | Returns users that liked the post with id postID. | 200 | 500 |

# 4. Conclusion

Now that we have a working REST API with Express, we will build a simple front-end application in the next practical.