Build Crud API with Node.js, Express, and MongoDB

We'll be building a CRUD App with Node.js, Express, and MongoDB. We'll use Mongoose for interacting with the MongoDB instance.

Currently, most of the websites operate on an API-based backend structure, where we just send a request from the front end of the website by calling an API and obtaining the required results. In this blog, we are going to build a simple CRUD (Create, Read, Update and Delete) app with the application of Node JS, Express JS, and MongoDB from the basics. Before we jump into the application, let's look into the tools we are going to use.

Express is one of the most popular web frameworks for node.js. It is built on top of the node.js HTTP module and adds support for routing, middleware, view system, etc. It is very simple and minimal, unlike other frameworks that try to do way too much, thereby reducing the flexibility for developers to have their own design choices.

Mongoose is an ODM (Object Document Mapping) tool for Node.js and MongoDB. It helps you convert the objects in your code to documents in the database and vice versa. Mongoose provides a straightforward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.

CRUD is an acronym for Create, Read, Update and Delete. It is a set of operations we get servers to execute (`POST`, `GET`, `PUT` and `DELETE` requests respectively). This is what each operation does:

- **Create (POST)** - Make something
- **Read (GET)**- Get something
- **Update (PUT)** - Change something
- **Delete (DELETE)**- Remove something

we'll heavily use ES6 features like `let`, `const`, `arrow functions`, `promises` etc. It's good to familiarize yourself with these features.

we'll be building a CRUD App with Node.js, Express, and MongoDB. We'll use Mongoose for interacting with the MongoDB instance.

Step 1: Creating the Application

Fire up your terminal and create a new folder for the application.

```
mkdir crud-node-express
```

Initialize the application with a package.json file

Go to the root folder of your application and type `npm init` to initialize your app with a `package.json` file.

```
cd crud-node-express
npm init
```

Note that I've specified a file named `server.js` as the entry point of our application. We'll create `server.js` file in the next section.

Step 2: Install dependencies

We will need express, mongoose, and body-parser modules in our application. Let's install them by typing the following command:

```
npm install express body-parser mongoose --save
```

Setting up the Web Server

Let's now create the main entry point of our application. Create a new file named `server.js` in the root folder of the application with the following contents:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();

app.use(bodyParser.urlencoded({ extended: true }))

app.use(bodyParser.json())

app.get('/', (req, res) => {
    res.json({"message": "Hello Crud Node Express"});
});

app.listen(3000, () => {
    console.log("Server is listening on port 3000");
});
```

First, We import express and body-parser modules. Express, as you know, is a web framework that we'll be using for building the REST APIs, and body-parser is a module that parses the request (of various content types) and creates a `req.body` object that we can access in our routes.

Then, We create an express app and add two `body-parser` middlewares using express's `app.use()` method. A middleware is a function that has access to `request` and `response` objects. It can execute any code, transform the request object, or return a response.

Then, We define a simple route that returns a welcome message to the clients.

Finally, We listen on port 3000 for incoming connections.

Step 3: Configuring and Connecting to the database

Let’s create a new folder `config` in the root folder of our application for keeping all the configurations

```
mkdir config
cd  config
```

Now, Create a new file `database.config.js` inside `config` folder with the following contents:

```
module.exports = {
    url: 'mongodb://localhost:27017/crud-node-express'
}
```

We’ll now import the above database configuration in `server.js` and connect to the database using mongoose.

Add the following code to the `server.js` file after `app.use(bodyParser.json())` line

```
const dbConfig =
require('./config/database.config.js');
const mongoose = require('mongoose');

mongoose.Promise = global.Promise;

mongoose.connect(dbConfig.url, {
    useNewUrlParser: true
}).then(() => {
    console.log("Databse Connected Successfully!!");
}).catch(err => {
    console.log('Could not connect to the database',
err);
    process.exit();
});
```

Please run the server and make sure that you’re able to connect to the database

```
node server.js
```

Step 4: Create Mongoose Model

Models are fancy constructors compiled from Schema definitions. An instance of a model is called a document. Models are responsible for creating and reading documents from the underlying MongoDB database.

create a folder called model inside the app folder. Create a user.js file and paste the below code.

```
var mongoose = require('mongoose');

var schema = new mongoose.Schema({
    email: {
        type: String,
        required: true,
        unique: true
    },
    firstName: {
        type: String,
        default: ''
    },
    lastName: {
        type: String,
        default: ''
    },
    phone: String,
});

var user = new mongoose.model('User', schema);

module.exports = user;
```

Next, we go for the two most important parts: Routes and the Controllers. Routing is the part where the APIs are actually created and hosted. Normally we do not need to create the controllers but it’s a good practice as sometimes if there are too many control functions, it becomes very hard to manage if we stack all the routing files. So, we define the functions in the Controllers part and import them in the routing section to keep the operation handling smoother.

So, let’s see what routing actually is. When say a user wants to query data from the database or wants to push data into the database, similarly delete or update, the frontend issues requests in the form of API calls. Now, there are different requests for each issue. For querying, we have to GET requests, for sending data we have POST requests. These are called HTTP requests. They enable interactions between the client and the server and work as a request-response protocol. The HTTP requests are:

GET is used to request data from a specified resource.

POST is used to send data to a server to create/update a resource.

HEAD: Same as GET, but it transfers the status line and the header section only.

PUT: Replaces all the current representations of the target resource with the uploaded content.

DELETE: Removes all the current representations of the target resource given by URI.

CONNECT: Establishes a tunnel to the server identified by a given URI.

PATCH: The PATCH method applies partial modifications to a resource

Systems like Node JS are currently operated on an MVC (Model View Controller) architecture. It's a design pattern. The idea is that it helps to focus on a specific part of the application and build it on a modular basis. The components are:

Model: It represents the structure and constraints of the data stored in the database.

View: It is the way the required data is presented to the user as per the need of the user.

Controller: This section controls the requests of the user and generates the appropriate response which is fed to the user.

Step 5: Create the Controller

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

- create
- findAll
- findOne
- update
- destroy

```
const UserModel = require('../model/user')

// Create and Save a new user
exports.create = async (req, res) => {
    if (!req.body.email && !req.body.firstName &&
 !req.body.lastName && !req.body.phone) {
        res.status( 400 ).send({ message : "Content can
not be empty!" });
    }

    const user = new UserModel({
        email : req.body.email,
        firstName : req.body.firstName,
        lastName : req.body.lastName,
        phone : req.body.phone
    });

    await user.save().then( data => {
        res.send({
            message : "User created successfully!!" ,
            user :data
        });
    }).catch( err => {
        res.status( 500 ).send({
            message : err.message || "Some error occurred
while creating user"
        });
    });
};

// Retrieve all users from the database.
exports .findAll = async (req, res) => {
    try {
        const user = await UserModel.find();
        res.status( 200 ).json(user);
    } catch (error) {
        res.status( 404 ).json({ message : error.message});
    }
};

// Find a single User with an id
exports.findOne = async (req, res) => {
    try {
        const user = await
UserModel.findById(req.params.id);
        res.status(200).json(user);
    } catch (error) {
        res.status(404).json({ message:
error.message});
    }
};

// Update a user by the id in the request
exports.update = async (req, res) => {
    if (!req.body) {
        res.status(400).send({
            message: "Data to update can not be empty!"
        });
    }

    const id = req.params.id;

    await UserModel.findByIdAndUpdate(id, req.body, {
useFindAndModify: false }).then(data => {
        if (!data) {
            res.status(404).send({
                message: `User not found.`
            });
        } else {
            res.send({ message : "User updated
successfully." })
        }
    }).catch( err => {
        res.status( 500 ).send({
            message : err.message
        });
    });
};

// Delete a user with the specified id in the request
exports.destroy = async (req, res) => {
    await
UserModel.findByIdAndRemove(req.params.id).then( data => {
        if (!data) {
            res.status( 404 ).send({
            message : `User not found.`
            });
        } else {
            res.send({
            message : "User deleted successfully!"
            });
        }
    }).catch( err => {
        res.status( 500 ).send({
          message : err.message
        });
    });
};
```

We have used **async** and **await** keywords as the database query takes time and so the asynchronous property of node js comes in.

Let’s now look at the implementation of the above controller functions one by one -

Creating a new User

```
// Create and Save a new user
exports.create = async (req, res) => {
    if (!req.body.email && !req.body.firstName &&
    !req.body.lastName && !req.body.phone) {
        res.status(400).send({ message: "Content can
not be empty!" });
    }

    const user = new UserModel({
        email: req.body.email,
        firstName: req.body.firstName,
        lastName: req.body.lastName,
        phone: req.body.phone
    });

    await user.save().then(data => {
        res.send({
            message:"User created successfully!!",
            user:data
        });
    }).catch(err => {
        res.status(500).send({
            message: err.message || "Some error occurred
while creating user"
        });
    });
};
```

Retrieving all Users

```
// Retrieve all users from the database.
exports.findAll = async (req, res) => {
    try {
        const user = await UserModel.find();
        res.status(200).json(user);
    } catch(error) {
        res.status(404).json({message: error.message});
    }
};
```

Retrieving a single User

```
// Find a single User with an id
exports.findOne = async (req, res) => {
    try {
        const user = await
UserModel.findById(req.params.id);
        res.status(200).json(user);
    } catch(error) {
        res.status(404).json({ message:
error.message});
    }
};
```

Updating a User

```
// Update a user by the id in the request
exports.update = async (req, res) => {
    if(!req.body) {
        res.status(400).send({
            message: "Data to update can not be empty!"
            });
        }

    const id = req.params.id;

    await UserModel.findByIdAndUpdate(id, req.body, {
useFindAndModify: false }).then(data => {
            if (!data) {
                res.status(404).send({
                    message: `User not found.`
                });
            } else {
                res.send({ message: "User updated
successfully." })
            }
        }).catch(err => {
            res.status(500).send({
                message: err.message
            });
        });
};
```

The `{new: true}` option in the `findByIdAndUpdate()` a method is used to return the modified document to the `then()` function instead of the original.

Deleting a User

```
// Delete a user with the specified id in the request
exports.destroy = async (req, res) => {
    await
 UserModel.findByIdAndRemove(req.params.id).then(data => {
            if (!data) {
                res.status( 404 ).send({
                    message : `User not found.`
                });
            } else {
                res.send({
                    message : "User deleted successfully!"
                });
            }
        }).catch( err => {
            res.status( 500 ).send({
                message : err.message
            });
        });
};
```

Step 6: Define Routes

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

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

```
const express = require ( 'express' )
const UserController = require ( '../controllers/User' )
const router = express.Router();

router.get( '/' , UserController.findAll);
router.get( '/:id' , UserController.findOne);
router.post( '/' , UserController.create);
router.patch( '/:id' , UserController.update);
router.delete( '/:id' , UserController.destroy);
```

```
module.exports = router
```

The last step before trying out our routes is to add the route class to the **server.js**

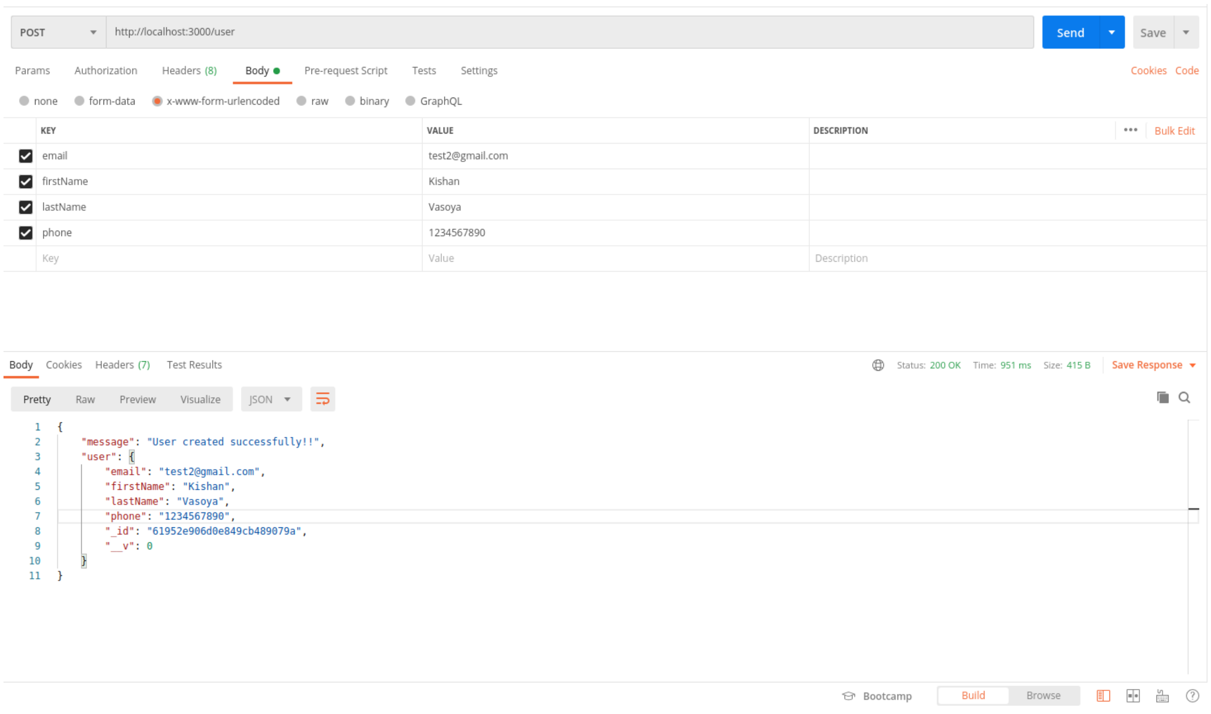
```
const UserRoute = require('./app/routes/User')
```

```
app.use('/user',UserRoute)
```

restart your node.js server and now we have our API ready.

Testing Endpoints

Create a User

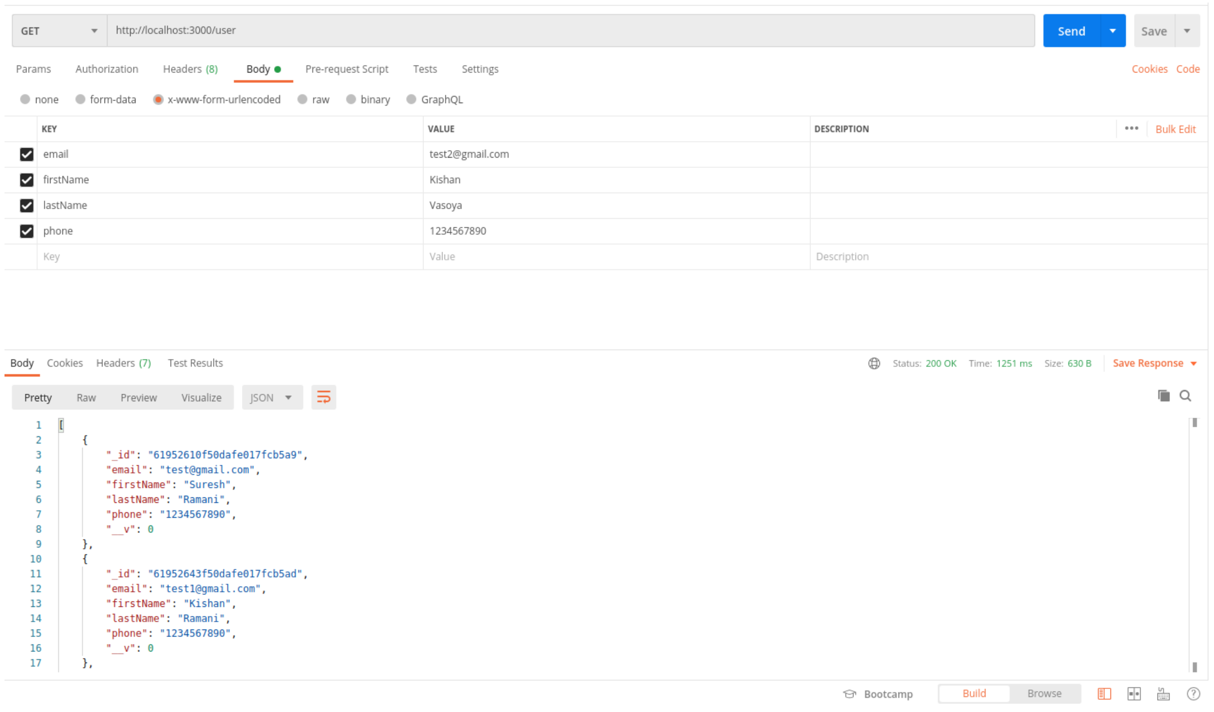

POST http://localhost:3000/user

KEY	VALUE	DESCRIPTION
✓ email	test2@gmail.com	
✓ firstName	Kishan	
✓ lastName	Vasoya	
✓ phone	1234567890	
Key	Value	Description

Status: 200 OK Time: 951 ms Size: 415 B

```
{
    "message": "User created successfully!!",
    "user": {
        "email": "test2@gmail.com",
        "firstName": "Kishan",
        "lastName": "Vasoya",
        "phone": "1234567890",
        "_id": "61952e906d0e849cb489079a",
        "__v": 0
    }
}
```

Get All Users


GET http://localhost:3000/user

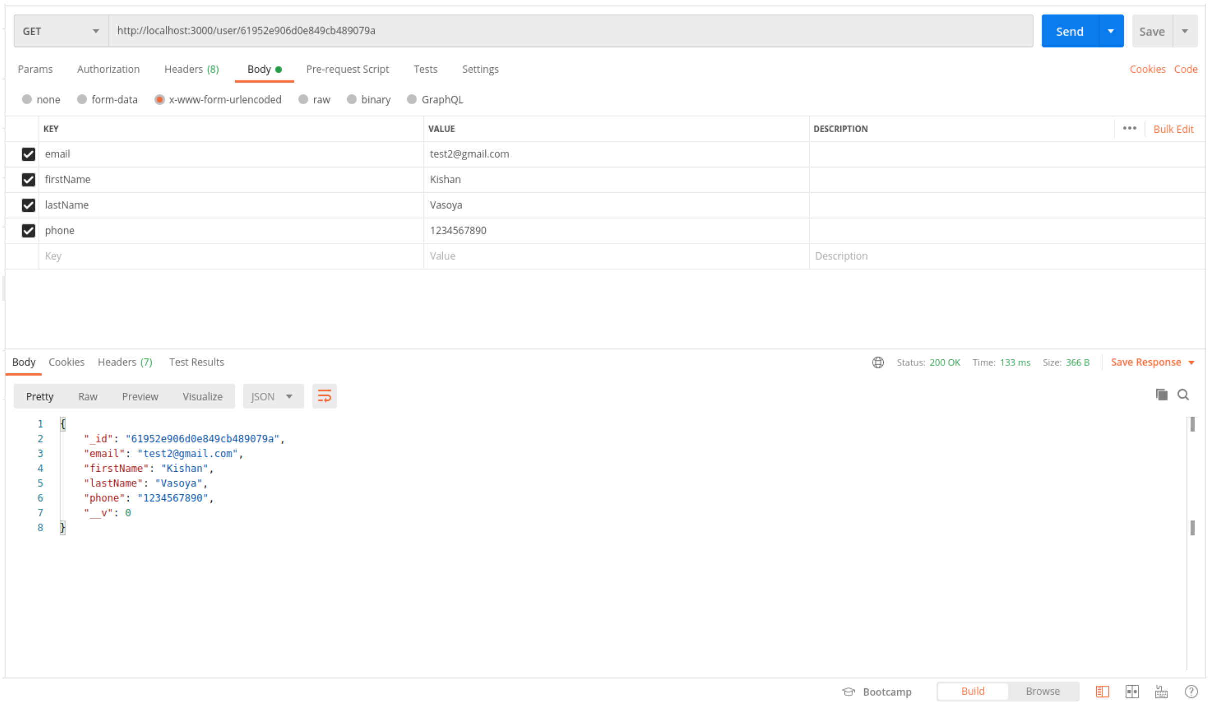
none · form-data · [✓ x-www-form-urlencoded] · raw · binary · GraphQL

	KEY	VALUE	DESCRIPTION
✓	email	test2@gmail.com	
✓	firstName	Kishan	
✓	lastName	Vasoya	
✓	phone	1234567890	
	Key	Value	Description

Status: 200 OK Time: 1251 ms Size: 630 B

```
[
    {
        "_id": "61952610f50dafe017fcb5a9",
        "email": "test@gmail.com",
        "firstName": "Suresh",
        "lastName": "Ramani",
        "phone": "1234567890",
        "__v": 0
    },
    {
        "_id": "61952643f50dafe017fcb5ad",
        "email": "test1@gmail.com",
        "firstName": "Kishan",
        "lastName": "Ramani",
        "phone": "1234567890",
        "__v": 0
    },
```

Get User By ID


GET http://localhost:3000/user/61952e906d0e849cb489079a

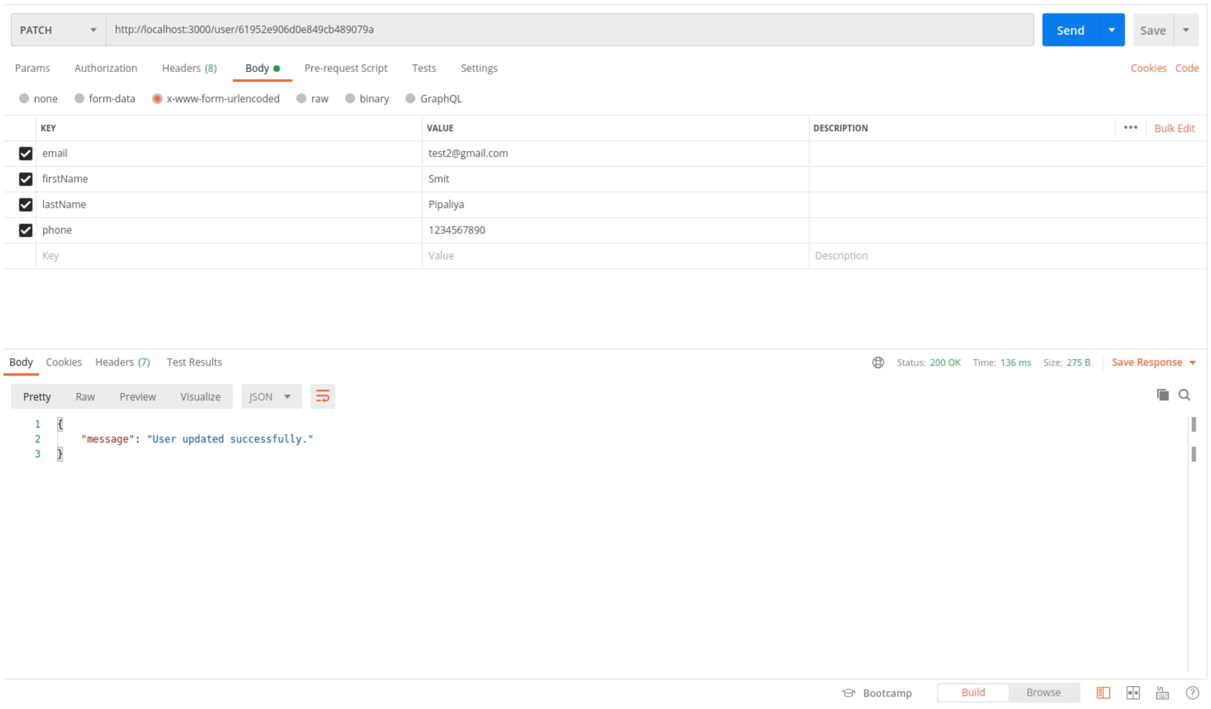
none · form-data · [✓ x-www-form-urlencoded] · raw · binary · GraphQL

	KEY	VALUE	DESCRIPTION
✓	email	test2@gmail.com	
✓	firstName	Kishan	
✓	lastName	Vasoya	
✓	phone	1234567890	
	Key	Value	Description

Status: 200 OK Time: 133 ms Size: 366 B

```
{
    "_id": "61952e906d0e849cb489079a",
    "email": "test2@gmail.com",
    "firstName": "Kishan",
    "lastName": "Vasoya",
    "phone": "1234567890",
    "__v": 0
}
```

Update a User


PATCH http://localhost:3000/user/61952e906d0e849cb489079a

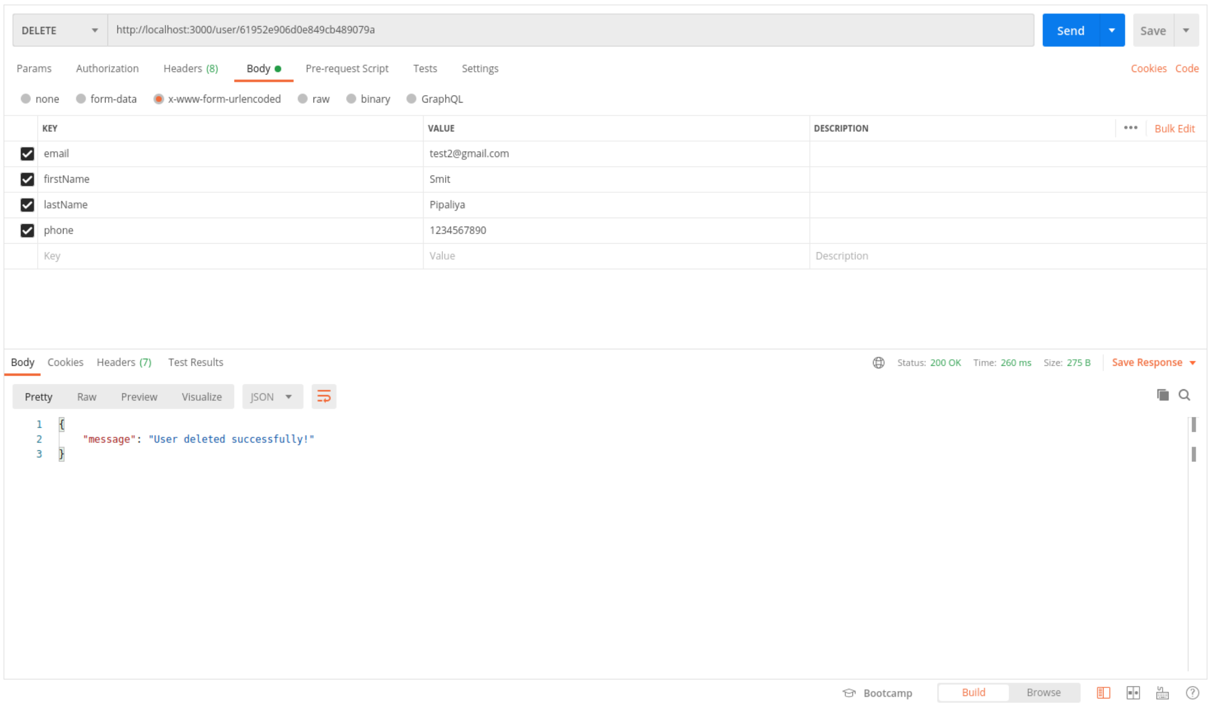
Body: x-www-form-urlencoded

KEY	VALUE	DESCRIPTION
email	test2@gmail.com	
firstName	Smit	
lastName	Pipaliya	
phone	1234567890	

Status: 200 OK Time: 136 ms Size: 275 B

```
{
    "message": "User updated successfully."
}
```

Delete a User


DELETE http://localhost:3000/user/61952e906d0e849cb489079a

Body: x-www-form-urlencoded

KEY	VALUE	DESCRIPTION
email	test2@gmail.com	
firstName	Smit	
lastName	Pipaliya	
phone	1234567890	

Status: 200 OK Time: 260 ms Size: 275 B

```
{
    "message": "User deleted successfully!"
}
```