

# Hands-on Lab - CRUD operations with Node.js



## Estimated Time Needed: 1 hour

In this lab you will learn how to create a **Friend's list** using Express server. Your application should allow you to add a friend with the following details: First name, Last name, Email and Date of birth. You will also be providing the application the ability to retrieve details, change details and delete the details.

You will be creating an application with API endpoints to perform Create, Retrieve, Update and Delete operations on the above data using an Express server.

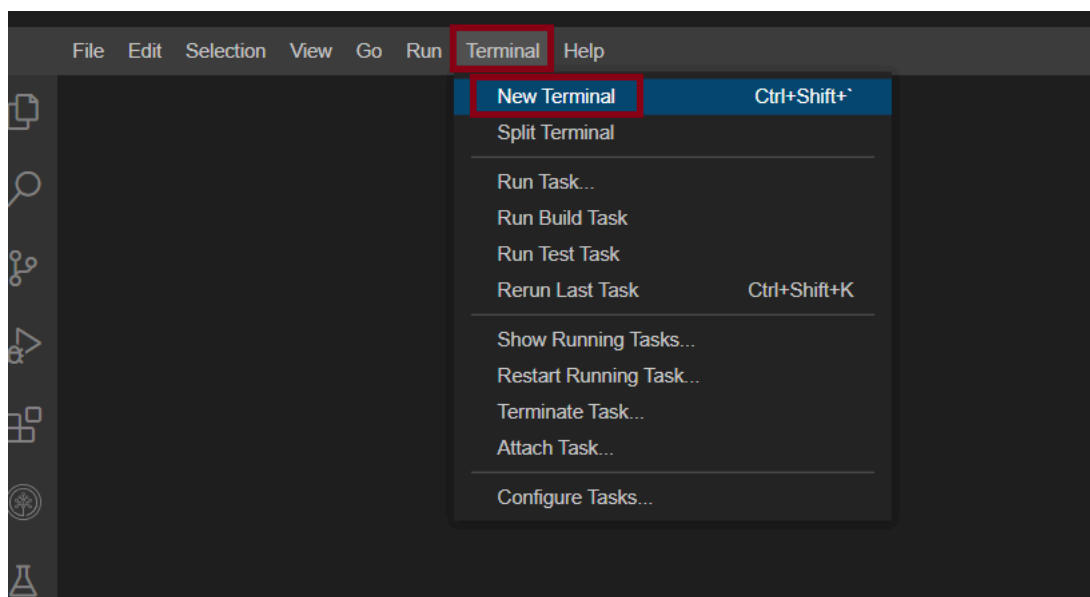
You will also learn to provide authenticated access to the endpoints. You will use cURL and Postman to test the implemented endpoints.

## Objectives:

- Create API endpoints to perform Create, Retrieve, Update and Delete operations on transient data with an Express server.
- Implement authentication at the session level using JSON Web Tokens (JWT) for authorized access.

## Set-up : Create application

1. Open a terminal window by using the menu in the editor: Terminal > New Terminal.



2. Change to your project folder, if you are not in the project folder already.

```
cd /home/project
```

3. Run the following command to clone the git repository that contains the starter code needed for this lab, if it doesn't already exist.

```
[ ! -d 'mxpfu-nodejsLabs' ] && git clone https://github.com/ibm-developer-skills-network/mxpfu-nodejsLabs.git
```

```
theia@theiadocke /home/project$ [ ! -d 'mxpfu-nodejsLabs' ] && git clone https://github.com/ibm-developer-skills-network/mxpfu-nodejsLabs.git
Cloning into 'mxpfu-nodejsLabs'...
remote: Enumerating objects: 100, done.
remote: Counting objects: 100% (47/47), done.
remote: Compressing objects: 100% (30/30), done.
remote: Total 100 (delta 28), reused 21 (delta 13), pack-reused 53
Receiving objects: 100% (100/100), 55.52 KiB | 5.55 MiB/s, done.
Resolving deltas: 100% (36/36), done.
theia@theiadocke /home/project$
```

5. Change to the directory **mxpfu-nodejsLabs** directory to start working on the lab.

```
cd mxpfu-nodejsLabs/
```

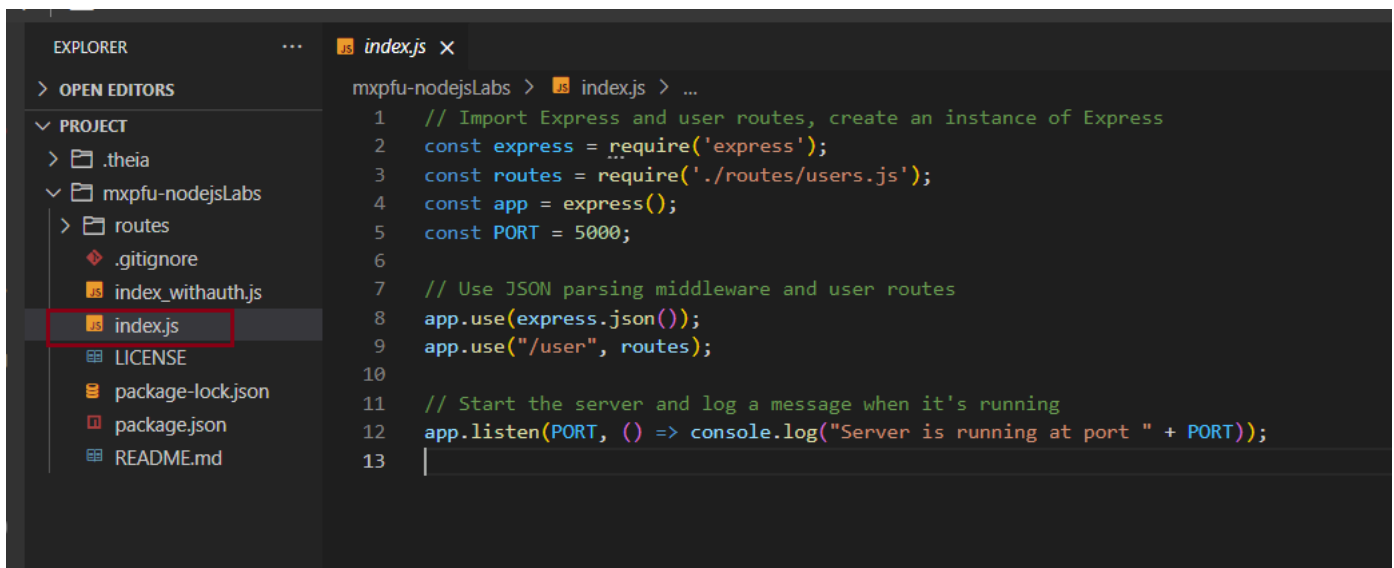
6. List the contents of this directory to see the artifacts for this lab.

```
ls
```

```
theia@theiadocker-lavanyas: /home/project/mxpfu-nodejsLabs$ ls
index.js          LICENSE           package-lock.json  routes
index_withauth.js package.json      README.md
```

## Exercise 1: Understand the server application

1. In the Files Explorer open the **mxpfu-nodejsLabs** folder and view **index.js**.



You have an Express server that has been configured to run at port 5000. When you access the server with **/user** you can access the endpoints defined in **routes/users.js**.

Recall that GET, POST, PUT and DELETE are the commonly used HTTP methods to perform CRUD operations. Those operations retrieve and send data to the server.

- GET is used to request data from a specified resource.
- POST is used to send data to a server for creating a resource.
- PUT is used to send data to a server to update a resource.
- DELETE is used for deleting a specified resource.

POST AND PUT are sometimes used interchangeably.

2. This lab requires some packages to be installed. The **express** and **nodemon** package for starting and running the Express server and **jsonwebtoken** and **express-session** for session based authentication.

- **express** - This is for creating a server to serve the API endpoints.
- **nodemon** - This will help to restart the server when you make any changes to the code.
- **jsonwebtoken** - This package helps in generating a JSON web token which we will use for authentication. A **JSON web token (JWT)** is a JSON object used to communicate information securely over the internet (between two parties). It can be used for information exchange and is typically used for authentication systems.
- **express-session** - This package will help us to maintain the authentication for the session.

These packages are defined in as **dependencies** in packages.json.

```
{
  "dependencies": {
    "express": "^4.18.1",
    "express-session": "^1.17.3",
    "jsonwebtoken": "^8.5.1",
    "nodemon": "^2.0.19"
  }
}
```

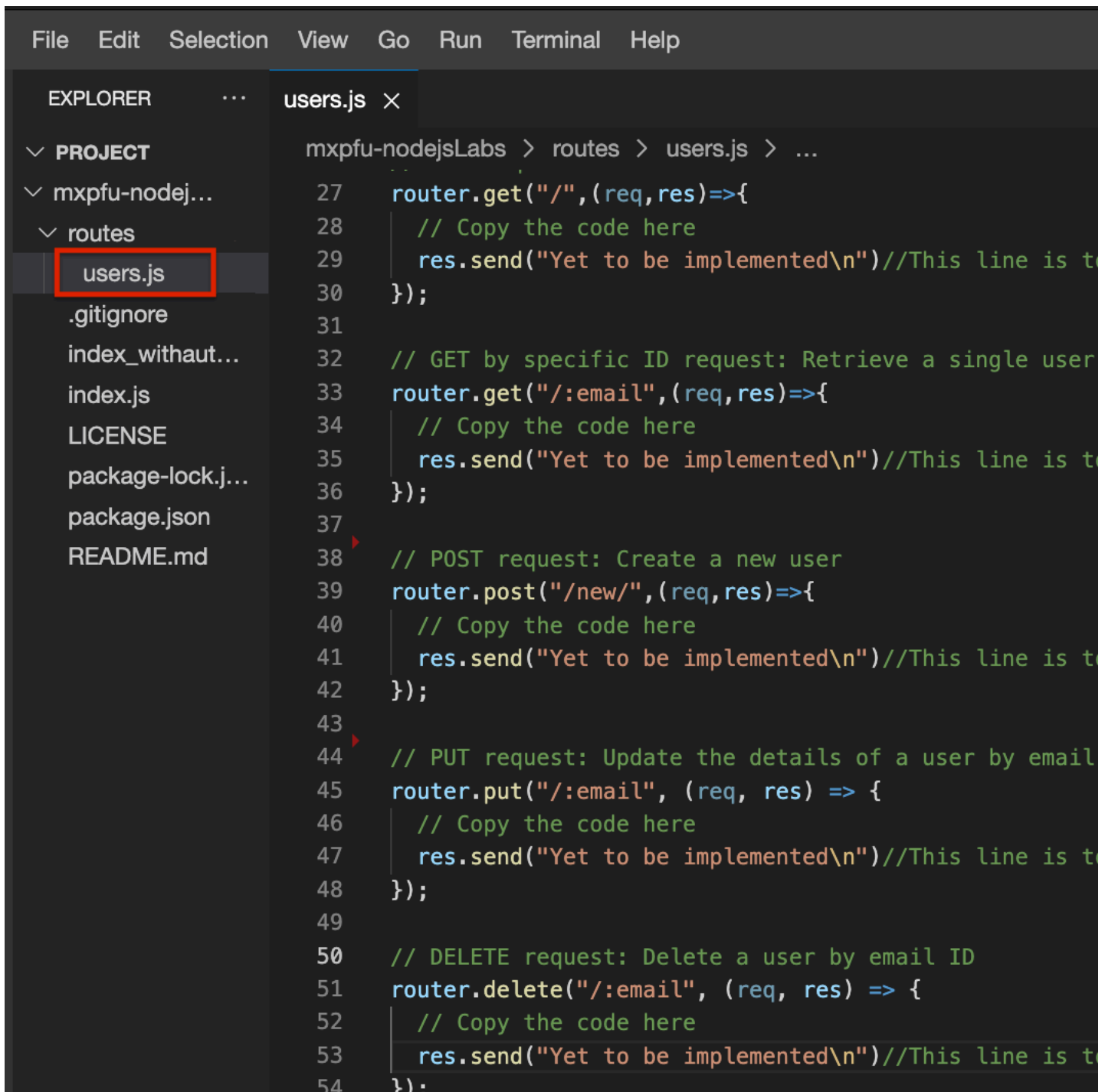
3. Observe that the express app uses the middleware **express.json()** to handle the request as a json object.

```
app.use(express.json());
```

4. Observe that the express app uses routes to handle the endpoints which start with **/user**. This means that for all the endpoints starting with **/user**, the server will go and look for an endpoint handler in **users.js**.

```
app.use("/user", routes);
```

5. All the endpoints have skeletal, but working implementation in **users.js**. Navigate to users.js under the directory routes and observe the endpoints defined in it.



```
File Edit Selection View Go Run Terminal Help

EXPLORER  ...  users.js x
  PROJECT
  mxpfu-nodej...
    routes
      users.js
    .gitignore
    index_without...
    index.js
    LICENSE
    package-lock.j...
    package.json
    README.md

mxpfu-nodejsLabs > routes > users.js > ...
27 router.get("/", (req, res) => {
28   // Copy the code here
29   res.send("Yet to be implemented\n");//This line is t
30 });
31
32 // GET by specific ID request: Retrieve a single user
33 router.get("/:email", (req, res) => {
34   // Copy the code here
35   res.send("Yet to be implemented\n");//This line is t
36 });
37
38 // POST request: Create a new user
39 router.post("/new/", (req, res) => {
40   // Copy the code here
41   res.send("Yet to be implemented\n");//This line is t
42 });
43
44 // PUT request: Update the details of a user by email
45 router.put("/:email", (req, res) => {
46   // Copy the code here
47   res.send("Yet to be implemented\n");//This line is t
48 });
49
50 // DELETE request: Delete a user by email ID
51 router.delete("/:email", (req, res) => {
52   // Copy the code here
53   res.send("Yet to be implemented\n");//This line is t
54 });
```

## Exercise 2: Run the server

The starter code given is a functioning server with dummy return values. Before starting to implement the actual endpoints, run the server.

1. In the terminal, print the working directory to ensure you are in **/home/projects/mxpfu-nodejsLabs**.

```
pwd
```

2. Install all the packages that are required for running the server. Copy, paste, and run the following command.

```
npm install
```

This will install all the required packages as defined in packages.json.

3. Start the express server.

npm start

4. Open a **New Terminal** from the top menu. Test an endpoint to retrieve these users. This has not yet been implemented to return the users.

curl localhost:5000/user

```
theia@theiadocker-lavanyas:/home/project$ curl localhost:5000/user
Yet to be implemented
```

5. If you see the output as displayed above, it means the server is running as expected.

## Exercise 2: Implement your endpoints

1. Navigate to the file named **users.js** in the **routes** folder. The endpoints have been defined and space has been provided for you to implement the endpoints.

```

1  const express = require('express');
2  const router = express.Router();
3
4
5  let users = [
6    {
7      firstName: "John",
8      lastName: "wick",
9      email: "johnwick@gamil.com",
10     DOB: "22-01-1990",
11   },
12   {
13     firstName: "John",
14     lastName: "smith",
15     email: "johnsmith@gamil.com",
16     DOB: "21-07-1983",
17   },
18   {
19     firstName: "Joyal",
20     lastName: "white",
21     email: "joyalwhite@gamil.com",
22     DOB: "21-03-1989",
23   },
24 ];
25

```

2. **R** in CRUD stands for retrieve. You will first add an API endpoint, using the **get** method for getting the details of all users. A few users have been added in the starter code.

- Copy the code below and paste in users.js inside the {} brackets within the **router.get("/ ",(req,res)=>{})** method.

```
res.send(users);
```

```

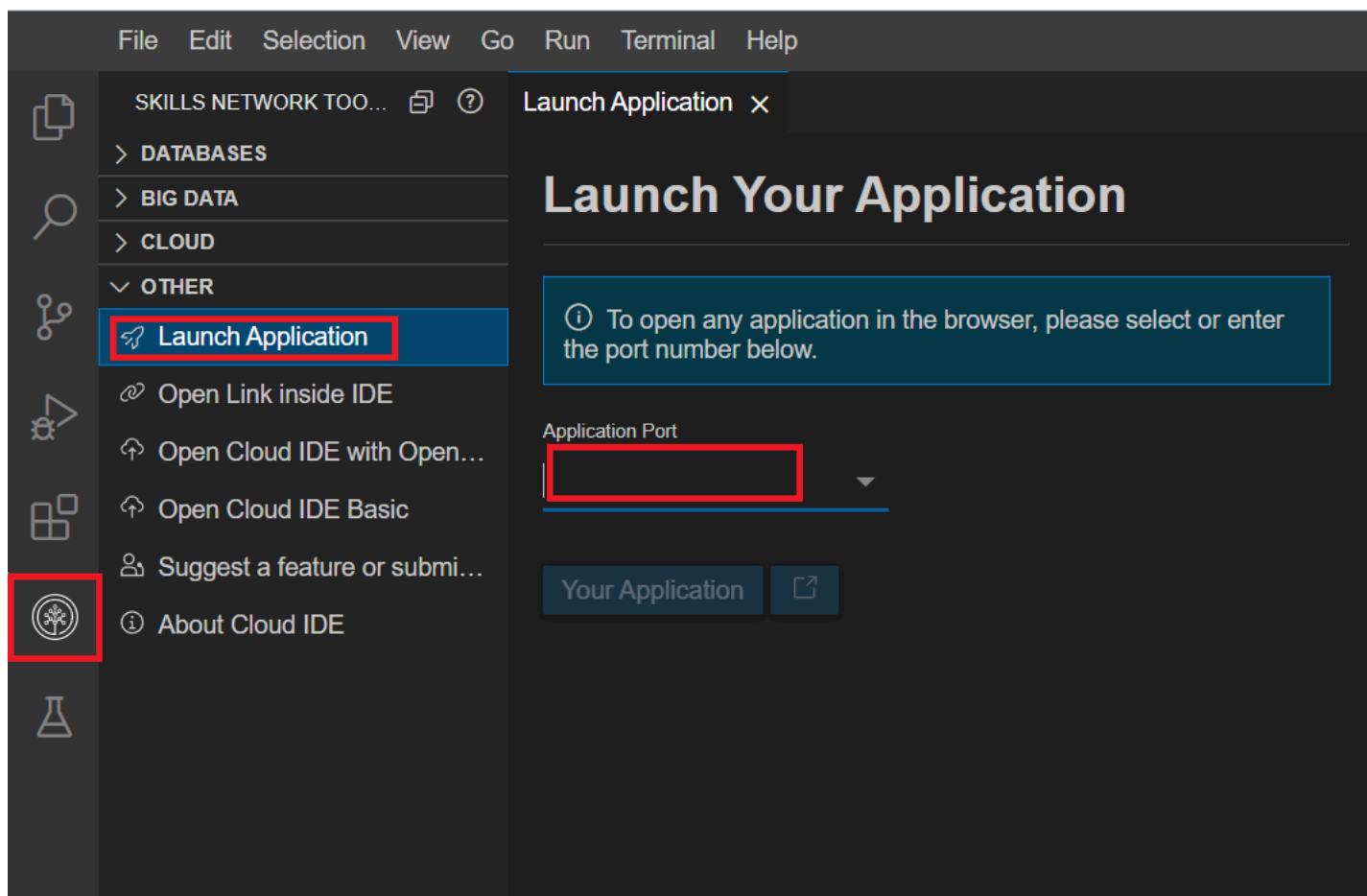
1  const express = require('express');
2  const router = express.Router();
3
4
5  let users = [
6    {
7      firstName: "John",
8      lastName: "wick",
9      email: "johnwick@gamil.com",
10     DOB: "22-01-1990",
11   },
12   {
13     firstName: "John",
14     lastName: "smith",
15     email: "johnsmith@gamil.com",
16     DOB: "21-07-1983",
17   },
18   {
19     firstName: "Joyal",
20     lastName: "white",
21     email: "joyalwhite@gamil.com",
22     DOB: "21-03-1989",
23   },
24 ];
25
26 // GET request: Retrieve all users
27 router.get("/",(req,res)=>{
28   res.send(users);
29 });
30

```

3. Ensure that your server is running. As you make changes to the code, the server that you started in the previous task, should be restart. If the server is not running, start it again.

npm start

3. Click on the **Skills Network** button on the left. It will open the "Skills Network Toolbox". Then click **OTHER** then **Launch Application**. From there you should be able to enter the port as 5000 and launch the development server.



4. When the browser page opens up, suffix **/user** to the end of the URL on the address bar. You will see the below page.



5. Check the output of the GET request using the curl command just the way you did in the previous exercise.

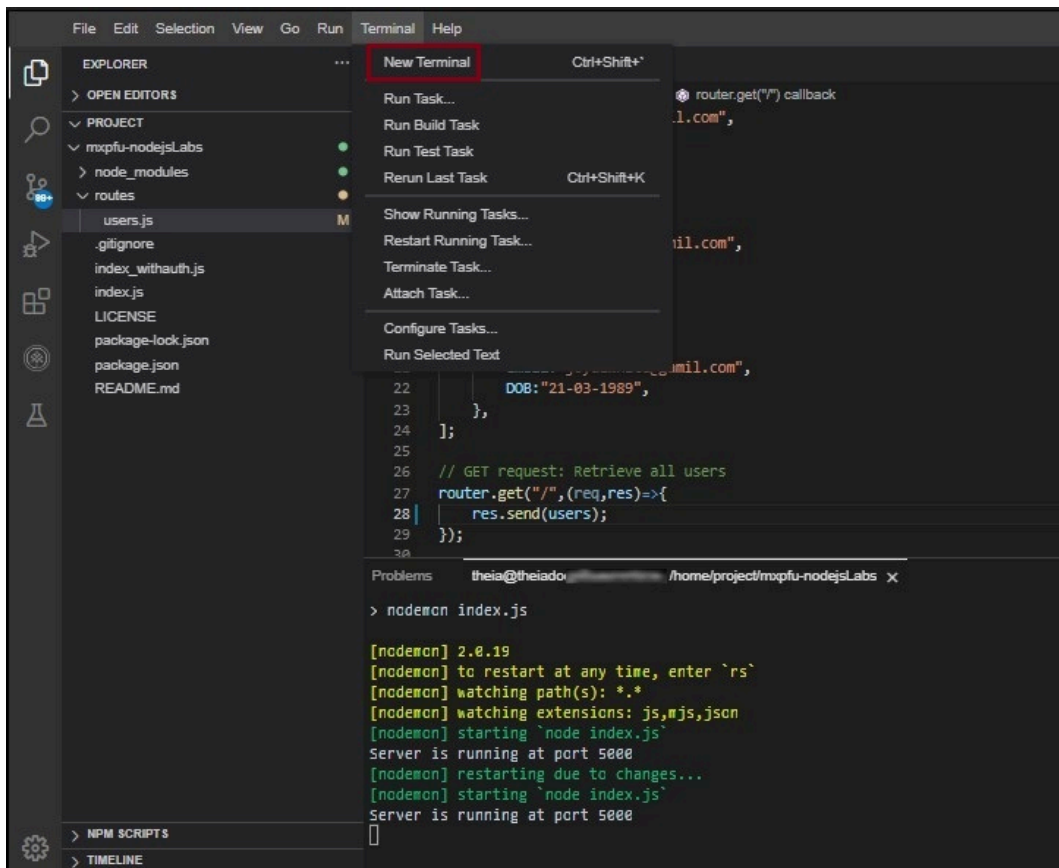
```
curl localhost:5000/user/
```

## Exercise 3: Creating a GET by specific email method:

1. Implement a **get** method for getting the details of a specific user based on their email ID by using the `filter` method on the user collection. Once you write the code and save it, the server will restart.

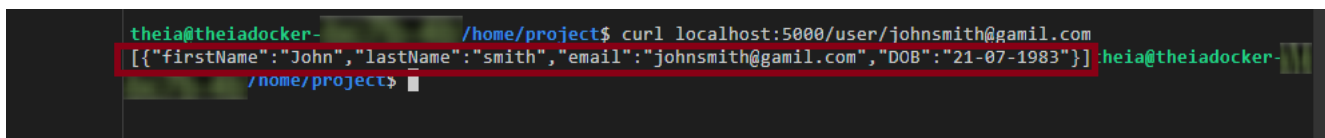
► [Click here to view the code](#)

2. Click on Terminal > New Terminal



3. In the new terminal, use the below command to view the output for the user with mail id [johnsmith@gamil.com](mailto:johnsmith@gamil.com)

```
curl localhost:5000/user/johnsmith@gamil.com
```



## Exercise 4: Creating the POST method:

1. The **C** in CRUD stands for **Create**. Implement the `/user` endpoint with the POST method to create a user and add the user to the list. You can create the user object as a dictionary. You can use the sample user object displayed below.

```
{
  "firstName": "Jon",
  "lastName": "Lovato",
  "email": "jonlovato@theworld.com",
  "DOB": "10/10/1995"
}
```

Use `push` to add the dictionary into the list of users. The user details can be passed as query paramters named `firstName`, `lastName`, `DOB` and `email`.

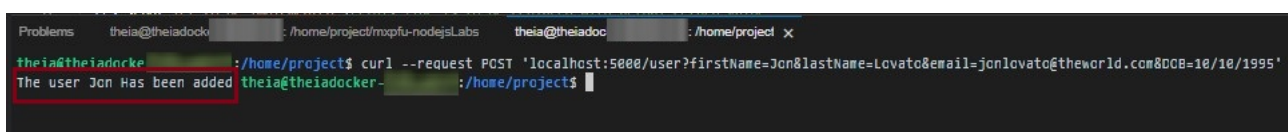
Hint: Query param can be retrieved from the request object using `request.query.paramname`

► [Click here to view the code](#)

2. Use the below command to post a new user with mail id '[jonlovato@theworld.com](mailto:jonlovato@theworld.com)' on the new terminal:

```
curl --request POST 'localhost:5000/user?firstName=Jon&lastName=Lovato&email=jonlovato@theworld.com&DOB=10/10/1995'
```

3. The output will be as below:



4. To verify if the user with email '[jonlovato@theworld.com](mailto:jonlovato@theworld.com)' has been added, you can send a GET request as below:

```
curl localhost:5000/user/jonlovato@theworld.com
```



```

theia@theiadocker-: /home/project$ curl localhost:5000/user/jonlovato@theworld.com
[{"firstName":"Jon","lastName":"Lovato","email":"jonlovato@theworld.com","DOB":"10/10/1995"}]theia@theiadocker-: /home/project$

```

## Exercise 5: Creating the PUT method:

1. The **U** in CRUD stands for update which can be achieved using the PUT method. To make updates in the data, you will use the PUT method. You should first look at the user with the specified email id and then modify it. The code below shows how the date of birth (DOB) of a user can be modified. Make the necessary code changes to allow changes to the other attributes of the user.

```

router.put("/:email", (req, res) => {
  // Extract email parameter and find users with matching email
  const email = req.params.email;
  let filtered_users = users.filter((user) => user.email === email);

  if (filtered_users.length > 0) {
    // Select the first matching user and update attributes if provided
    let filtered_user = filtered_users[0];

    // Extract and update DOB if provided
    let DOB = req.query.DOB;
    if (DOB) {
      filtered_user.DOB = DOB;
    }

    /*
    Include similar code here for updating other attributes as needed
    */

    // Replace old user entry with updated user
    users = users.filter((user) => user.email !== email);
    users.push(filtered_user);

    // Send success message indicating the user has been updated
    res.send(`User with the email ${email} updated.`);
  } else {
    // Send error message if no user found
    res.send("Unable to find user!");
  }
});

```

2. The completed code will look like this.

```

56 // PUT request: Update the details of a user by email ID
57 router.put("/:email", (req, res) => {
58   // Extract email parameter and find users with matching email
59   const email = req.params.email;
60   let filtered_users = users.filter((user) => user.email === email);
61
62   if (filtered_users.length > 0) {
63     // Select the first matching user and update attributes if provided
64     let filtered_user = filtered_users[0];
65
66     // Extract and update DOB if provided
67     let DOB = req.query.DOB;
68     if (DOB) {
69       filtered_user.DOB = DOB;
70     }
71
72     /*
73     Include similar code here for updating other attributes as needed
74     */
75     // Extract and update firstName if provided
76     let firstName = req.query.firstName;
77     if (firstName) {
78       filtered_user.firstName = firstName;
79     }
80
81     // Extract and update lastName if provided
82     let lastName = req.query.lastName;
83     if (lastName) {
84       filtered_user.lastName = lastName;
85     }
86
87     // Replace old user entry with updated user
88     users = users.filter((user) => user.email !== email);
89     users.push(filtered_user);
90
91     // Send success message indicating the user has been updated
92     res.send(`User with the email ${email} updated.`);
93   } else {
94     // Send error message if no user found
95     res.send("Unable to find user!");
96   }
97 });
98

```

3. Use the below command to update the DOB as 1/1/1971 for the user with mail id '[johnsmith@gamil.com](mailto:johnsmith@gamil.com)' in the split terminal:

```
curl --request PUT 'localhost:5000/user/johnsmith@gamil.com?DOB=1/1/1971'
```

4. The output will be as below:

```
Problems theia@theiadocker: /home/project/mxpfu-nodejsLabs theia@theiadocker: /home/project x
theia@theiadocker: /home/project$ curl --request PUT 'localhost:5000/user/johnsmith@gamil.com?DOB=1/1/1971'
User with the email johnsmith@gamil.com updated.theia@theiadocker: /home/project$
```

5. To verify if the DOB of the user with email '[johnsmith@gamil.com](mailto:johnsmith@gamil.com)' has been updated, you can send a GET request as below:

```
curl localhost:5000/user/johnsmith@gamil.com
```

```
[nodemon] starting `node index.js`
Server is running
[{"firstName":"John","lastName":"smith","email":"johnsmith@gamil.com"}]
theia@theiadocker: /home/project$ curl localhost:5000/user/johnsmith@gamil.com
[{"firstName":"John","lastName":"smith","email":"johnsmith@gamil.com"}]
theia@theiadocker: /home/project$
```

## Exercise 6: Creating the DELETE method:

1. The "D" in CRUD stands for **Delete**. Implement the DELETE method for deleting a specific user's email by using the below code:

```
router.delete("/:email", (req, res) => {
  // Extract the email parameter from the request URL
  const email = req.params.email;
  // Filter the users array to exclude the user with the specified email
  users = users.filter((user) => user.email !== email);
  // Send a success message as the response, indicating the user has been deleted
  res.send(`User with the email ${email} deleted.`);
});
```

2. The completed code will look like this.

```
99
100 // DELETE request: Delete a user by email ID
101 router.delete("/:email", (req, res) => {
102   // Extract the email parameter from the request URL
103   const email = req.params.email;
104   // Filter the users array to exclude the user with the specified email
105   users = users.filter((user) => user.email !== email);
106   // Send a success message as the response, indicating the user has been deleted
107   res.send(`User with the email ${email} deleted.`);
108 });
```

3. Use the below command to delete the user with mail id '[johnsmith@gamil.com](mailto:johnsmith@gamil.com)' in the split terminal:

```
curl --request DELETE 'localhost:5000/user/johnsmith@gamil.com'
```

4. The output will be as below:

```
Problems theia@theiadocker: /home/project/mxpfu-nodejsLabs theia@theiadocker: /home/project x
theia@theiadocker: /home/project$ curl --request DELETE 'localhost:5000/user/johnsmith@gamil.com'
User with the email johnsmith@gamil.com deleted.theia@theiadocker: /home/project$
```

5. Send a GET request for the user with email '[johnsmith@gamil.com](mailto:johnsmith@gamil.com)' and ensure that a null object is returned:

```
theia@theiadocker: /home/project$ curl localhost:5000/user/johnsmith@gamil.com
[]
theia@theiadocker: /home/project$
```

## Optional Exercise: Formatting the output

1. To make the output more readable, you can use the JSON stringify method as given below. Please update the code for the GET method to:

```
// Define a route handler for GET requests to the root path "/"
router.get("/", (req, res) => {
  // Send a JSON response containing the users array, formatted with an indentation of 4 spaces for readability
  res.send(JSON.stringify(users, null, 4));
});
```

2. Launch the app on port 5000 & append 'user' to the end of the URL.

3. This will render the output of the GET method as a JSON string per the updated GET method shown below:



← → ↻ 🔒 -5000.theiadocker-2-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user

```
{ "users": [ { "firstName": "John", "lastName": "wick", "email": "johnwick@gamil.com", "DOB": "22-01-1990" }, { "firstName": "Joyal", "lastName": "white", "email": "joyalwhite@gamil.com", "DOB": "21-03-1989" } ] }
```

## Exercise 7: Implementing Authentication

All these endpoints are accessible by anyone. You will now see how to add authentication to the CRUD operations. This code has been implemented in **index\_withauth.js**.

1. Observe the following code block in **index\_withauth.js**.

```
app.use(session({secret:"fingerprint",resave: true, saveUninitialized: true}))
```

This tells your express app to use the session middleware.

- **secret** - a random unique string key used to authenticate a session.
- **resave** - takes a Boolean value. It enables the session to be stored back to the session store, even if the session was never modified during the request.
- **saveUninitialized** - this allows any uninitialized session to be sent to the store. When a session is created but not modified, it is referred to as **uninitialized**.

The default value of both **resave** and **saveUninitialized** is true, but the default is deprecated. So, set the appropriate value according to the use case.

2. Observe the implementation of the **login** endpoint. A user logs into the system providing a username. An access token that is valid for one hour is generated. You may observe this validity length specified by **60 \* 60**, which signifies the time in seconds. This access token is set into the session object to ensure that only authenticated users can access the endpoints for that length of time.

```
// Login endpoint
app.post("/login", (req, res) => {
  const user = req.body.user;
  if (!user) {
    return res.status(404).json({ message: "Body Empty" });
  }
  // Generate JWT access token
  let accessToken = jwt.sign({
    data: user
  }, 'access', { expiresIn: 60 * 60 });
  // Store access token in session
  req.session.authorization = {
    accessToken
  }
  return res.status(200).send("User successfully logged in");
});
```

3. Observe the implementation of the authentication middleware. All the endpoints starting with **/user** will go through this middleware. It will retrieve the authorization details from the session and verify it. If the token is validated, the user is authenticated and the control is passed on to the next endpoint handler. If the token is invalid, the user is not authenticated and an error message is returned.

```
// Middleware for user authentication
app.use("/user", (req, res, next) => {
  // Check if user is authenticated
  if (req.session.authorization) {
    let token = req.session.authorization['accessToken']; // Access Token

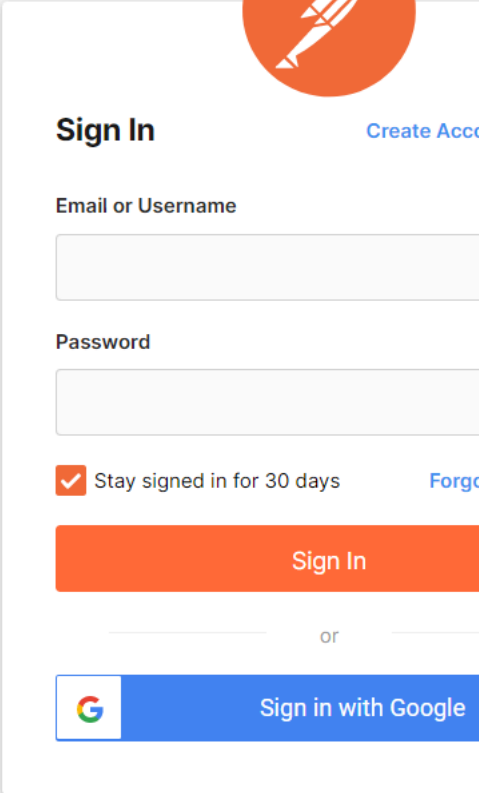
    // Verify JWT token for user authentication
    jwt.verify(token, "access", (err, user) => {
      if (!err) {
        req.user = user; // Set authenticated user data on the request object
        next(); // Proceed to the next middleware
      } else {
        return res.status(403).json({ message: "User not authenticated" }); // Return error if token verification fails
      }
    });
  }

  // Return error if no access token is found in the session
} else {
  return res.status(403).json({ message: "User not logged in" });
}
});
```

## Exercise 8: Testing endpoints with POSTMAN

You have tested the API endpoints with cURL. An easier and more user-friendly way to test these endpoints with the graphical user interface tool (GUI), Postman.

1. Go to [Postman](#). Sign-up for a new Postman account if you don't already have one. Sign-in to your account.



The image shows a screenshot of the Postman application's sign-in modal. The modal is white with rounded corners and is positioned on the right side of a light gray background. At the top right of the modal is the Postman logo, which is an orange circle containing a white silhouette of a person running. Below the logo, the text "Sign In" is displayed in bold black font, and to its right is a blue link that says "Create Account". The form contains two input fields: "Email or Username" and "Password", both with light gray borders. Below the "Password" field is a checkbox with a checkmark and the text "Stay signed in for 30 days", followed by a blue link that says "Forgot Password". A large orange button with the text "Sign In" in white is located below the checkbox. Below the button is a horizontal line with the word "or" in the center. Below the line is a blue button with the Google logo on the left and the text "Sign in with Google" on the right. At the bottom right of the modal, there are two links: "Terms of use" and "Privacy Policy". At the bottom right of the entire screenshot, there is a copyright notice: "©2022 Postman, Inc. All rights reserved".

**Sign In** [Create Account](#)


Email or Username

Password

☒ Stay signed in for 30 days [Forgot Password](#)

Sign In

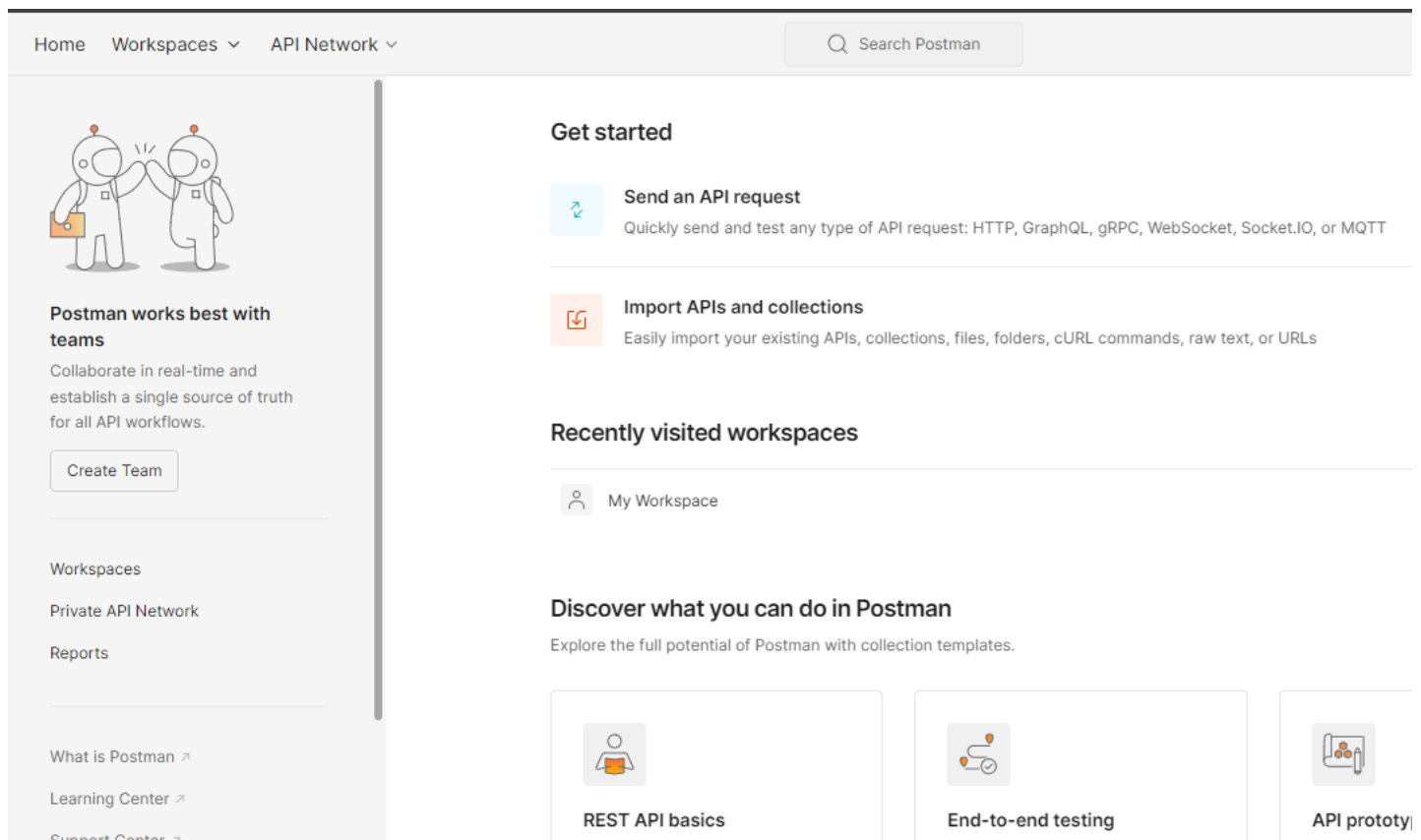
or

 Sign in with Google

[Terms of use](#) [Privacy Policy](#)

©2022 Postman, Inc. All rights reserved

2. After you login to Postman, click on **New Request** as shown below:



The image shows the Postman web interface. On the left is a sidebar with navigation links: Home, Workspaces, API Network, What is Postman, Learning Center, and Support Center. The main area is titled 'Get started' and includes sections for 'Send an API request', 'Import APIs and collections', and 'Recently visited workspaces'. There are also cards for 'REST API basics', 'End-to-end testing', and 'API prototyping'.

**Note:** If the server is running in the theia lab please stop the server by press **CTRL + C**. Now start the server by running the below command which will listen to port 5000.

```
npm run start_auth
```

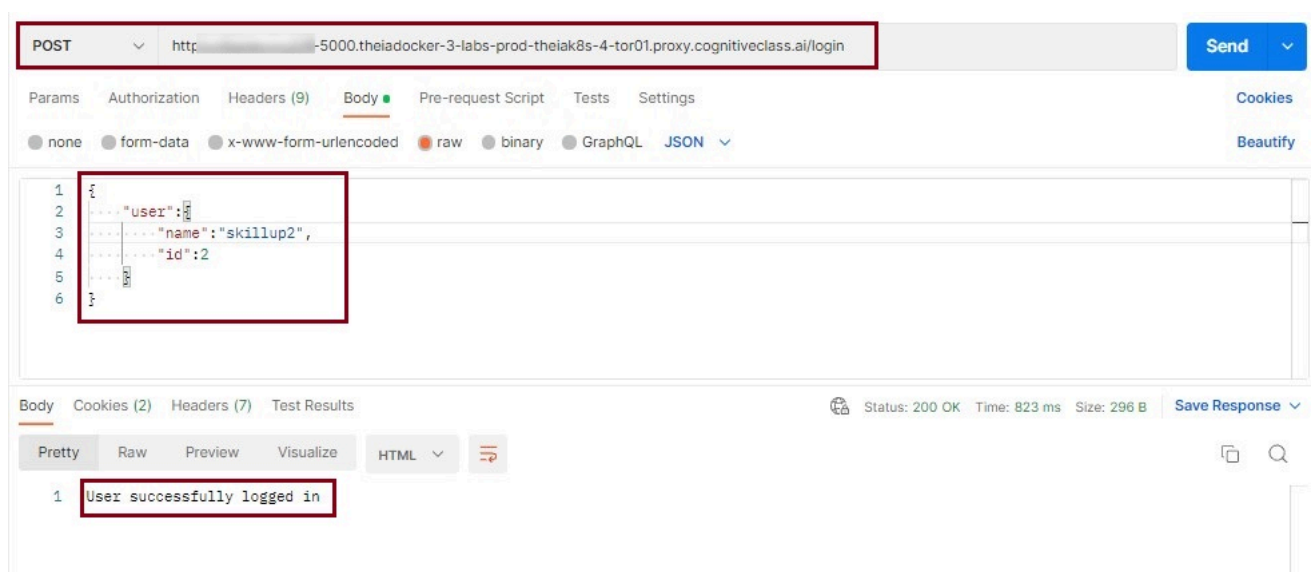
So far we were accessing all the endpoints without authentication but now we will be using authentication to access the endpoints.

- Copy the URL from the Launch application and add the login as an endpoint to add the user details in the **POST REQUEST** which will look like below:

```
https://<sn-lab-username>-5000.theiadocker-2-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/login
```

- User details should be in the below format:

```
{
  "user": {
    "name": "abc",
    "id": 1
  }
}
```



The image shows a Postman POST request configuration. The URL is `http://<sn-lab-username>-5000.theiadocker-3-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/login`. The 'Body' tab is selected, and the content type is set to 'JSON'. The JSON body is:

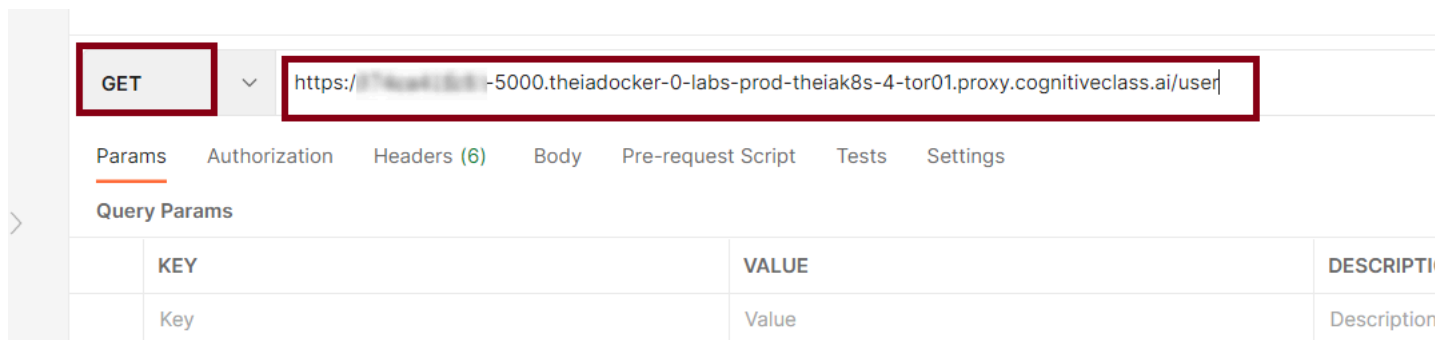
```
{
  "user": {
    "name": "skillup2",
    "id": 2
  }
}
```

The response is shown at the bottom: `1 User successfully logged in`.

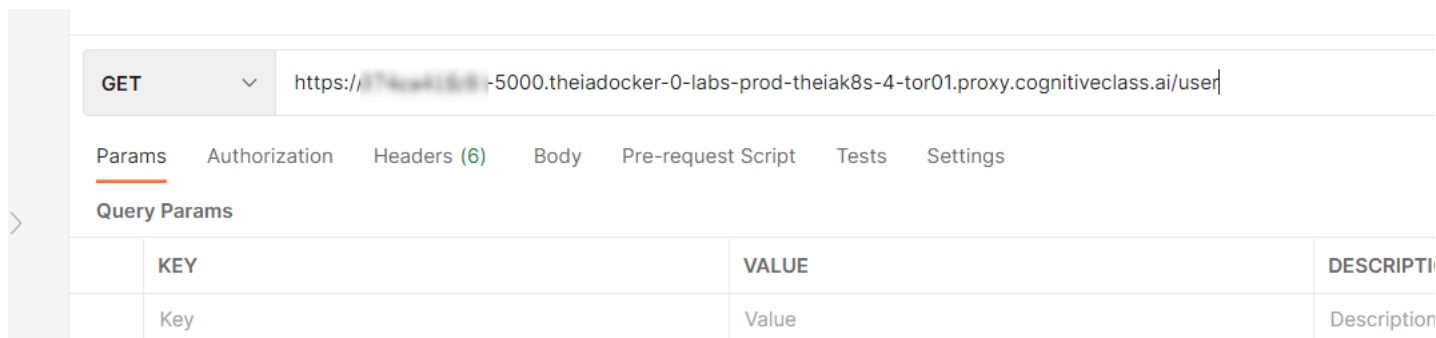
Now let's begin the test by sending an HTTP GET Request.

## 8.1 GET request

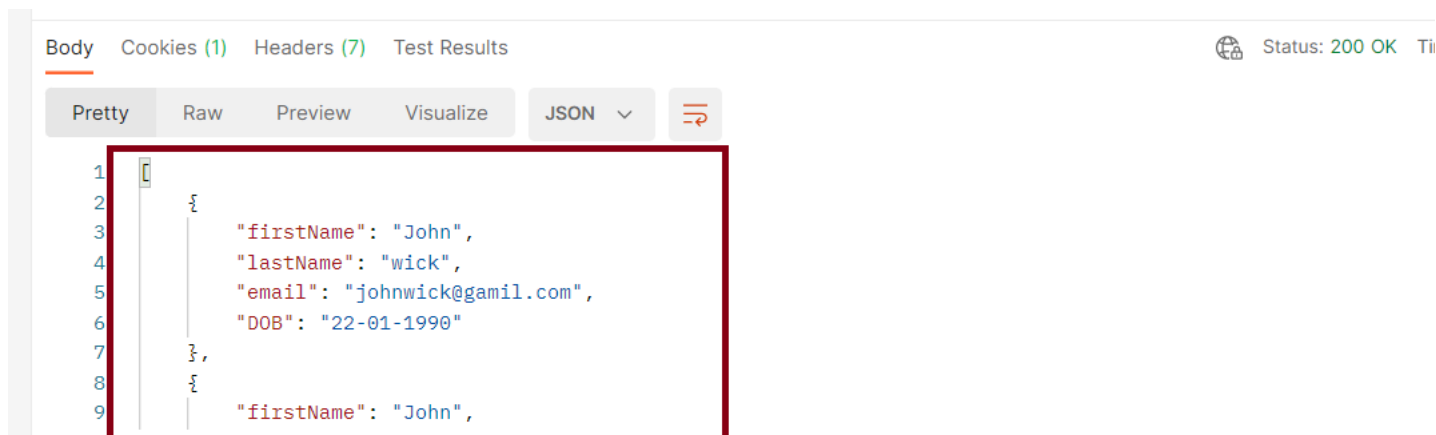
a. Enter the GET request URL: `https://XXXXXXXXXX-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user` in your input box of Postman where you see "Enter Request URL".



b. Click on the Send button after entering the URL.



c. The output will be as below:



## 8.2 GET request by specific ID

a. Enter the request URL by adding the specific email address to the above GET request URL. If the email address is `johnsmith@gamil.com` then enter the following URL in the input box of postman:

`https://XXXXXXXXXX-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user/johnsmith@gamil.com`

The screenshot shows a REST client interface with the following elements:

- Method:** GET
- URL:** https://[redacted]-5000.theiadocker-3-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user/johnsmith@gamil.com
- Tabs:** Params, Authorization, Headers (6), Body, Pre-request Script, Tests, Settings. The 'Params' tab is selected.
- Query Params Table:**

KEY	VALUE	DESCRIPTION
Key	Value	Description

b. Click on the **Send** button after entering the URL to view the output.

The screenshot shows a REST client interface with the following elements:

- Method:** GET
- URL:** https://[redacted]-5000.theiadocker-3-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user/johnsmith@gamil.com
- Tabs:** Params, Authorization, Headers (6), Body, Pre-request Script, Tests, Settings. The 'Params' tab is selected.
- Query Params Table:**

KEY	VALUE	DESCRIPTION
Key	Value	Description

c. The output will be as below:

The screenshot shows the response body of a GET request. The status is 200 OK. The response is displayed in JSON format, showing a list of two user objects. The first object is highlighted with a red box.

```
1 {
2   "firstName": "John",
3   "lastName": "wick",
4   "email": "johnwick@gamil.com",
5   "DOB": "22-01-1990"
6 },
7 {
8   "firstName": "John",
```

## 8.3 POST request :

a. Enter the basic post request URL:

https://XXXXXXXX-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user/

Ensure to select the POST method and select the "Params".

The screenshot shows a REST client interface with the following elements:

- Method:** POST
- URL:** https://[redacted]-5000.theiadocker-3-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user?firstName=Bob&lastName=
- Tabs:** Params, Authorization, Headers (7), Body, Pre-request Script, Tests, Settings. The 'Params' tab is selected.
- Query Params Table:**

KEY	VALUE	DESCRIPTION
Key	Value	Description

b. Enter the firstName as 'Bob', lastName as 'Smith', email as '[bobsmith@gamil.com](mailto:bobsmith@gamil.com)' and DOB as '1/1/1978' for a new user:

POST ▼ https://[redacted]-5000.theiadocker-3-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user?firstName=Bob&lastName=

Params ● Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	firstName	Bob	
<input checked="" type="checkbox"/>	lastName	Smith	
<input checked="" type="checkbox"/>	email	bobsmith@gamil.com	
<input checked="" type="checkbox"/>	DOB	1/1/1978	

c. Click on the **Send** button after entering the URL to view the output.

POST ▼ https://[redacted]-5000.theiadocker-3-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user?firstName=Bob&lastName=

Params ● Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	firstName	Bob	
<input checked="" type="checkbox"/>	lastName	Smith	
<input checked="" type="checkbox"/>	email	bobsmith@gamil.com	
<input checked="" type="checkbox"/>	DOB	1/1/1978	

Verify that the newly added values have been updated by doing the GET request.

Note: Ensure that you delete any parameters that you added for the POST request before sending the GET request.

GET ▼ https://[redacted]-5000.theiadocker-3-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user/bobsmith@gamil.com

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

Query Params

	KEY	VALUE	DESCRIPTION
	Key	Value	Description

Body Cookies (1) Headers (7) Test Results 🌐 Status: 200 OK Tii

Pretty Raw Preview Visualize JSON ▼ ≡

```
{
  "firstName": "Bob",
  "lastName": "Smith",
  "email": "bobsmith@gamil.com",
  "DOB": "1/1/1978"
}
```

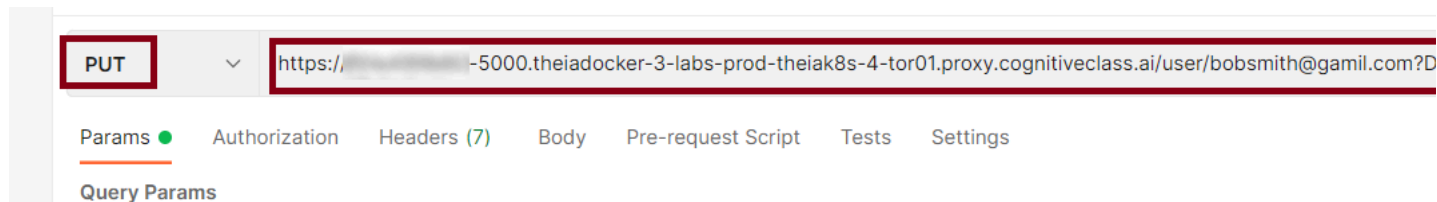


## 8.4 PUT request

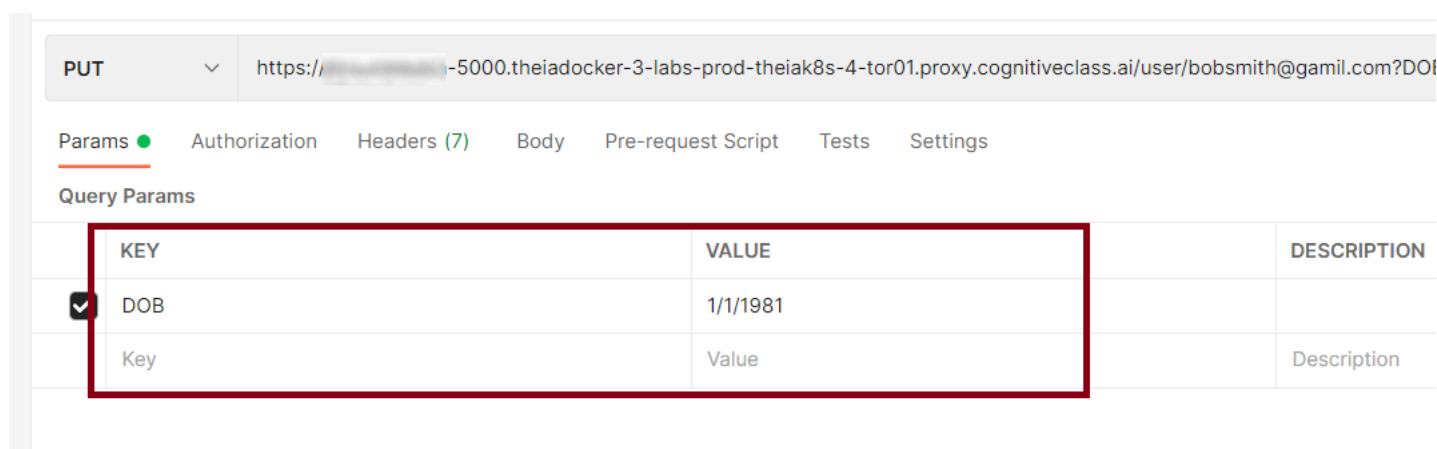
a. Enter the URL by adding the specific email address. If the email address is bobsmith@gamil.com then enter this URL in the input box of the Postman:

```
https://XXXXXXXXXX-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user/bobsmith@gamil.com
```

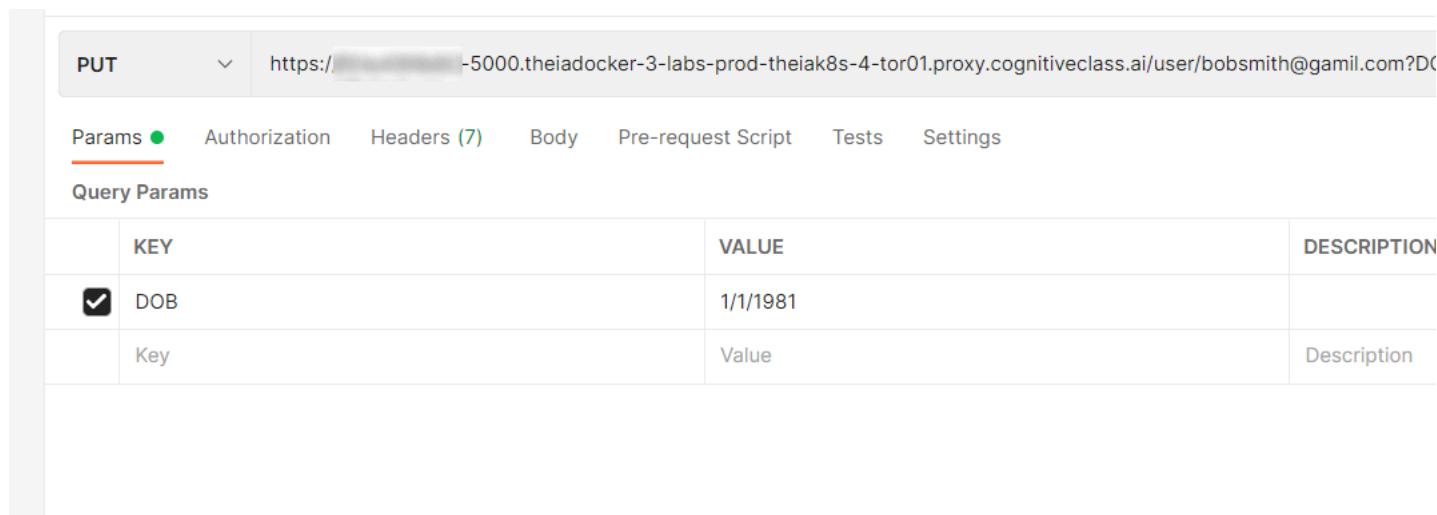
Ensure to select the PUT method and select the "Params".



b. Enter the key and values to be changed. For example, if you want to change the "DOB" key and replace it with the new value 1/1/1981 it will be as below.



c. Click on the **Send** button after entering the URL to view the output.



Verify that the newly added values are been updated by doing a GET request.

Note: Ensure that you delete any parameters that you added for the PUT request before sending the GET request.

The screenshot shows the Postman interface with a GET request. The URL bar contains the URL: `https://[redacted]-5000.theiadocker-3-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user/bobsmith@gamil.com`. The response body is displayed in JSON format, showing a user profile with the following details:

```
{  "firstName": "Bob",  "lastName": "Smith",  "email": "bobsmith@gamil.com",  "DOB": "1/1/1981"}
```

## 8.5 DELETE Request:

a. Enter the URL by adding the specific email address. If the email address is bobsmith@gamil.com then enter this URL in the input box of the Postman:

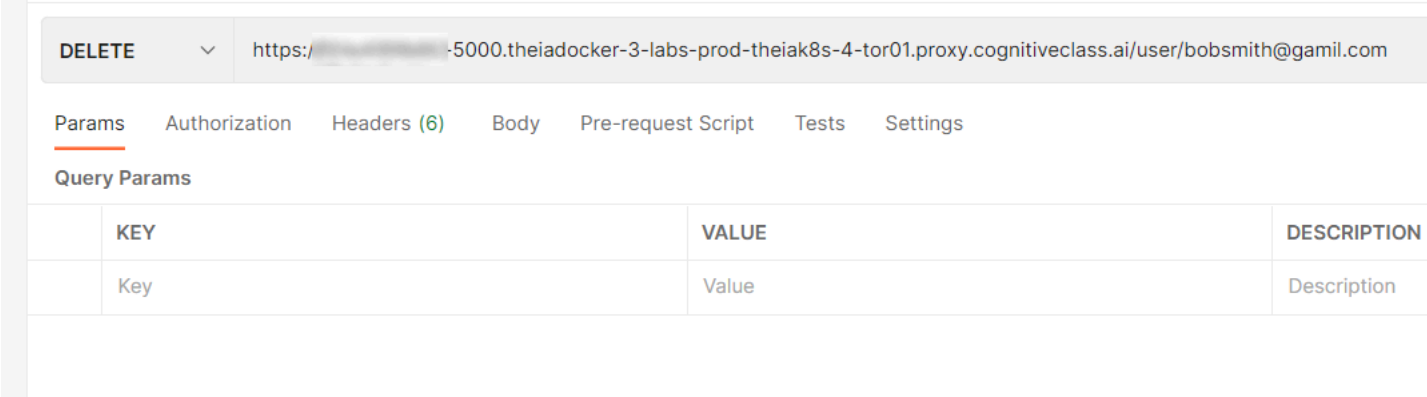
`https://XXXXXXXXXX-5000.theiadocker-0-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user/bobsmith@gamil.com`

Be sure to select the DELETE method.

The screenshot shows the Postman interface with a DELETE request. The URL bar contains the URL: `https://[redacted]-5000.theiadocker-3-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/user/bobsmith@gamil.com`. The interface shows the 'DELETE' method selected, and the 'Query Params' table is empty.

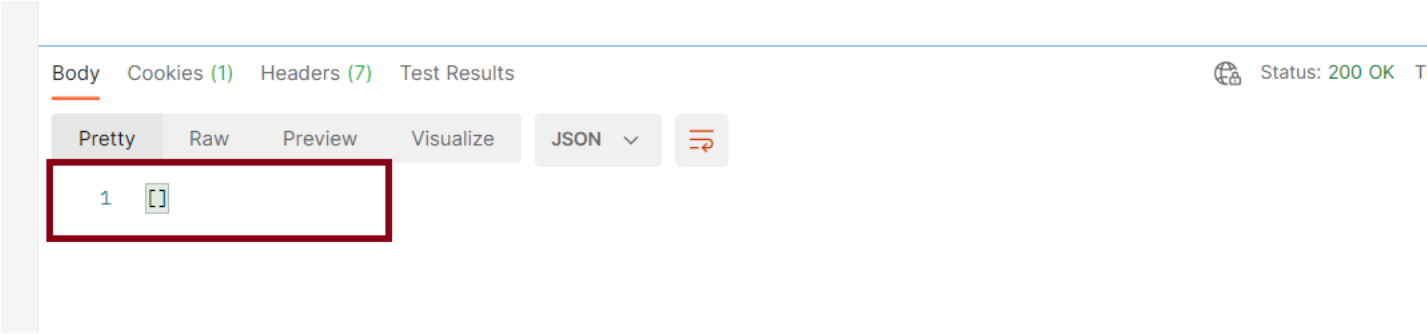
KEY	VALUE	DESCRIPTION
Key	Value	Description

b. Click on the "Send" button after entering the URL to view the output.



c. Verify that the GET user by ID bobsmith@gamil.com returns a null object by sending a GET request.

Note: Ensure that you delete any parameters (if any are there) before sending the GET request.



# Practice labs

- 1. Create an endpoint in the same code for getting all users with a particular Last Name.
  - ▶ Click here for a hint!
  - ▶ Click here for the Solution!
- 2. Create an endpoint in the same code for sorting users by date of birth.
  - ▶ Click here for a hint!
  - ▶ Click here for the Solution!

**Congratulations! You have completed the lab for CRUD operations with Node.js and Express.js using Postman.**

## Summary:

In this lab, we have performed CRUD Operations like GET, POST, PUT and DELETE on an Express App and tested the above methods using Postman.

## Author(s)

Sapthashree K S

K Sundararajan

© IBM Corporation. All rights reserved.