# Flask Application for CRUD operations on MongoDB

## Docker:

To use Docker to run a Flask app, you will need to create a Dockerfile that specifies how to build your container. Here's an example Dockerfile:

```
FROM python:3.9-alpine

WORKDIR /Corider-PyMongo

COPY requirements.txt requirements.txt

RUN pip install -r requirements.txt

COPY . .

EXPOSE 3000

CMD [ "python", "app.py" ]
```

This Dockerfile specifies the environment for building and running a Python Flask app with PyMongo. Here are the steps it follows:

1. It starts from a base image of Python 3.9 running on Alpine Linux, which is a lightweight distribution of Linux.
2. It sets the working directory inside the container to `/Corider-PyMongo`.
3. It copies the `requirements.txt` file to the container.
4. It installs the required Python packages from the `requirements.txt` file using pip.
5. It copies the rest of the files in the current directory to the container.
6. It exposes port 3000 to allow connections to the Flask app.
7. It sets the default command to run the Flask app using the command `python app.py`.

So, this Dockerfile creates a container that installs the required Python packages and runs a Flask app with PyMongo on port 3000.

### Building the container:

```
docker build -t myflaskapp .
```

### Running the dockerized Flask app:

```
docker run -p 3000:3000 myflaskapp
```

since we exposed port 3000 we need to map our port to docker's 3000 port.


## app.py

This is a Flask application that provides an API to perform CRUD operations on a MongoDB database.

### Import Necessary Libraries:

```
from flask import Flask, request,jsonify
from flask_restful import Api, Resource, reqparse
from flask_pymongo import PyMongo
from bson.objectid import ObjectId
```

### Initialize Flask app and Create new instance of MongoDB Client:

```
app = Flask(__name__)
app.config['MONGO_URI'] =" "# Update this with your own MongoDB URI
mongo = PyMongo(app)
db = mongo.db
api = Api(app)
```

### REST API

```
class home(Resource):
    def get(self):
        return {'message':'hello world'}

class Create(Resource):
    def post(self):
        data = request.get_json()
```

```
        result = db.users.insert_one(data)
        return {'id': str(result.inserted_id)}

import logging
logging.basicConfig(level=logging.DEBUG)

class Read(Resource):
    def get(self):
        users = db.users.find()
        output = []
        for user in users:
            output.append({'id': str(user['_id']), 'name': user['name'], 'email': user['email'], 'password': user['password']})
        return jsonify({'result': output})

class Read_id(Resource):

    def get(self, id):
        print(id)
        result = db.users.find_one({'_id': ObjectId(id)})
        print(result)
        if result:
            result['_id'] = str(result['_id'])
            return result
        else:
            return {'error': 'Not found'}, 404


class Update(Resource):
    def put(self, id):
        data = request.get_json()
        data = request.get_json()
        result = db.users.update_one({'_id': ObjectId(id)}, {'$set': data})
        if result.modified_count == 1:
            return {'message': 'Updated successfully'}
        else:
            return {'error': 'Not found'}, 404

class Delete(Resource):
    def delete(self, id):
        result = db.users.delete_one({'_id': ObjectId(id)})
        if result.deleted_count == 1:
            return {'message': 'Deleted successfully'}
        else:
            return {'error': 'Not found'}, 404
```

- The `home` class defines a simple `/` endpoint that returns a hello world message.
- The `Create` class defines a `/create` endpoint that accepts a JSON payload via a POST request and inserts the data into the MongoDB `users` collection. The newly inserted document's ID is returned in the response.
- The `Read` class defines a `/read` endpoint that retrieves all the documents in the `users` collection and returns them in a JSON format.
- The `Read_id` class defines a `/read-id/<string:id>` endpoint that accepts a document ID as a parameter and retrieves the corresponding document from the `users` collection. If the document is found, it is returned in a JSON format; otherwise, an error message is returned.
- The `Update` class defines an `/update/<string:id>` endpoint that accepts a document ID as a parameter and a JSON payload via a PUT request. The payload contains the updated fields for the corresponding document, which are updated in the `users` collection. If the update is successful, a success message is returned; otherwise, an error message is returned.
- The `Delete` class defines a `/delete/<string:id>` endpoint that accepts a document ID as a parameter and deletes the corresponding document from the `users` collection. If the delete is successful, a success message is returned; otherwise, an error message is returned.

### Binding Resource URL:

```
api.add_resource(home, '/')
api.add_resource(Create, '/create')
api.add_resource(Read, '/read')
api.add_resource(Read_id, '/read-id/<string:id>')
api.add_resource(Update, '/update/<string:id>')
api.add_resource(Delete, '/delete/<string:id>')
```

The `api.add_resource()` statements register each endpoint with the Flask app.

### Main method:

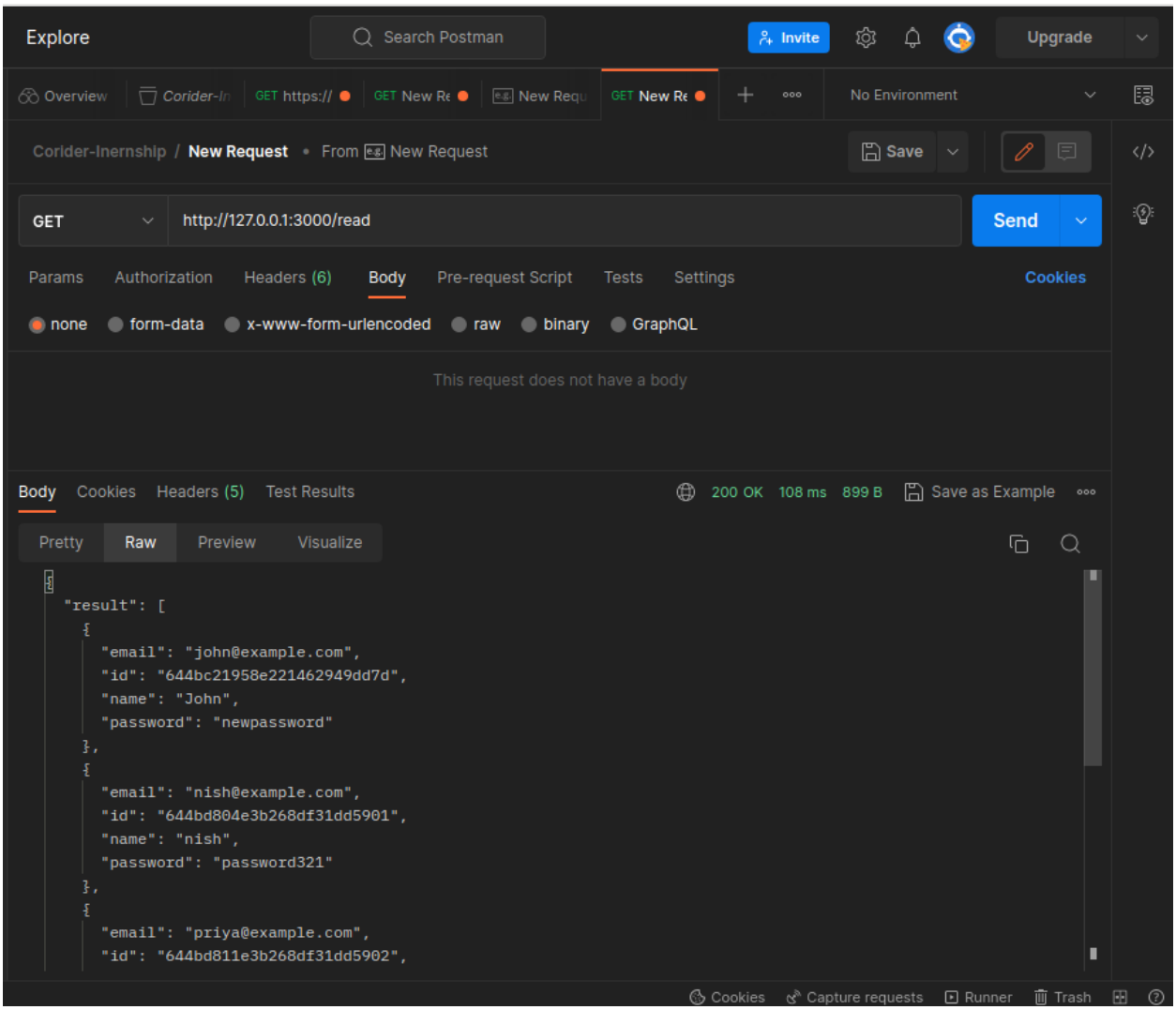
```
if __name__ == '__main__':
    app.run(debug=True,host='0.0.0.0', port=3000)
```

Testing:

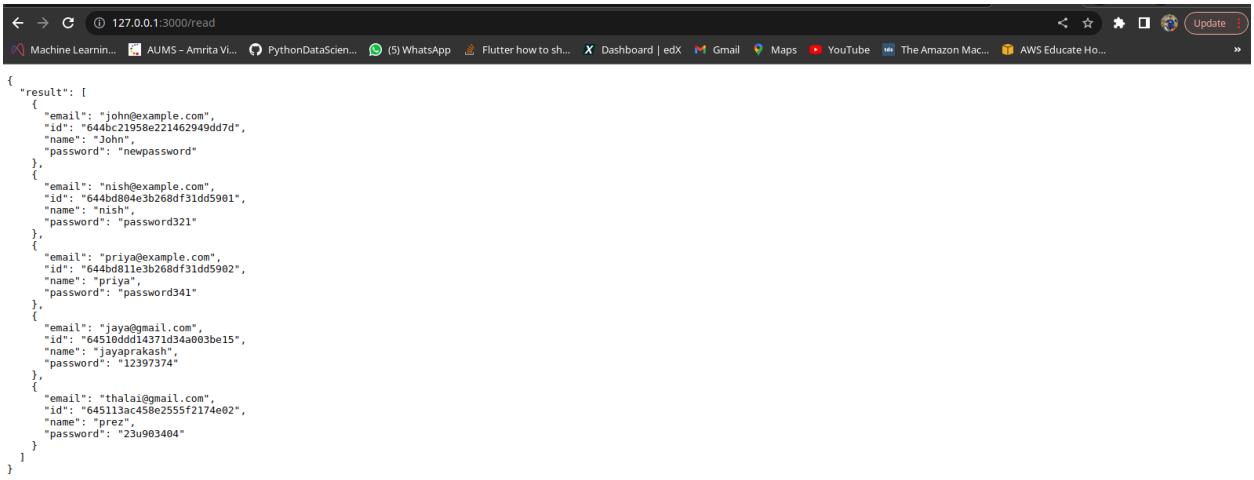
I have made use of postman to test out all the API endpoints and was sucessfully able to Create,Read,Update and Delete from the MongoDB database I created using MongoDB Atlas.

### Read User:

1. Open Postman and create a new request of type GET.
2. Enter the URL for the endpoint you want to test, e.g. `http:// 127.0.0.1:3000/read`.
3. Send the request and check the response in the "Body" tab. It should return a JSON object containing all the users present in the database if the request was successful.

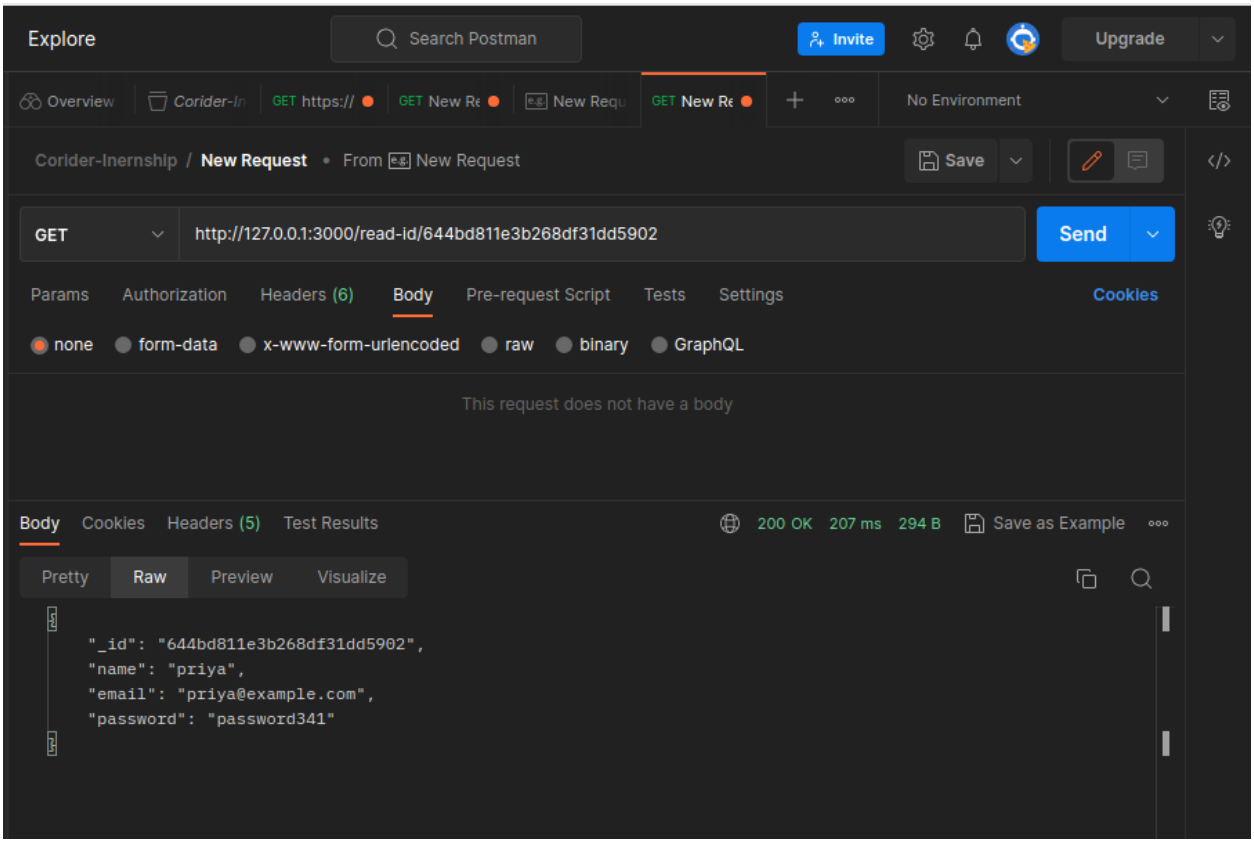



## From the Localhost:



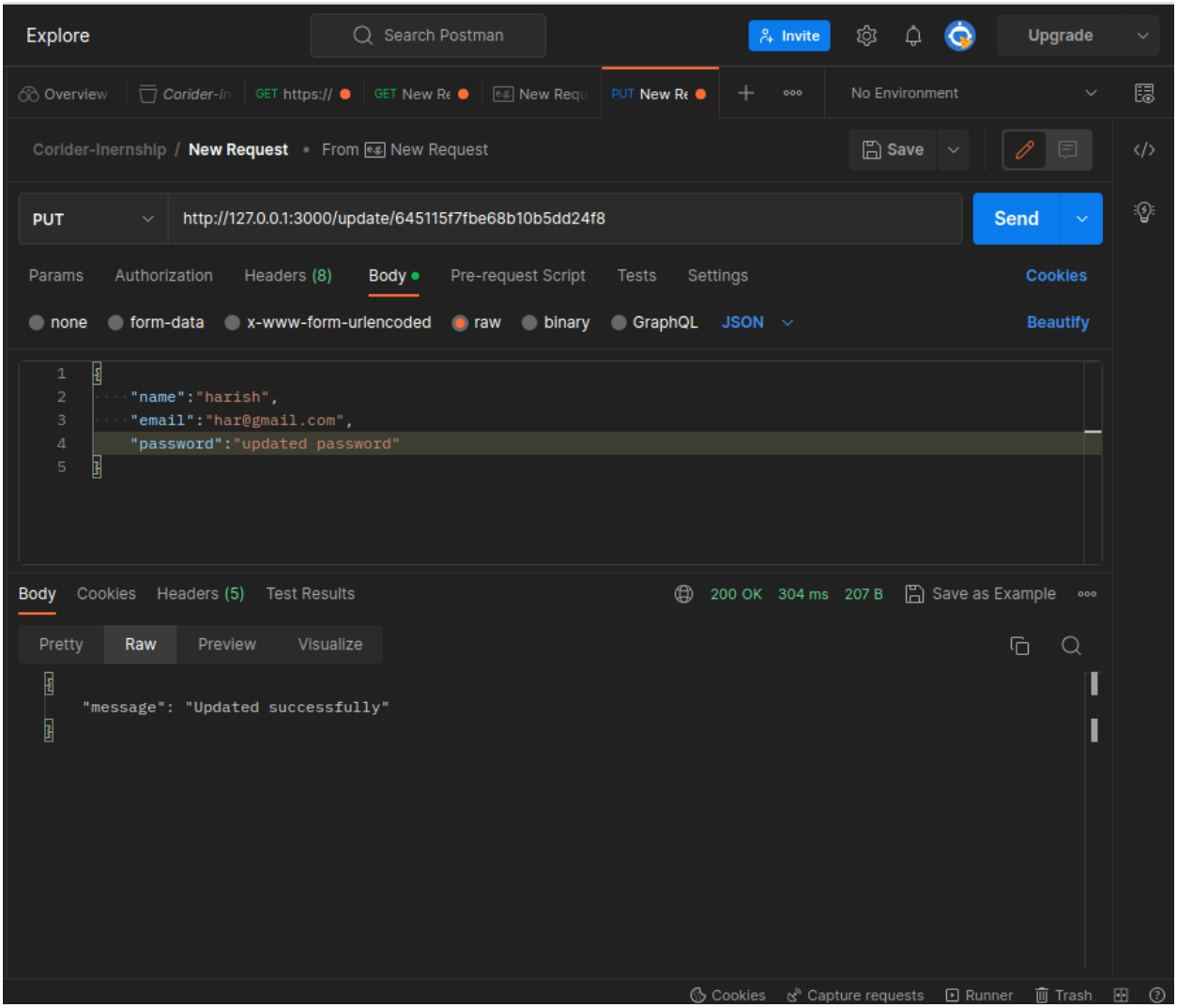
### Read User(by id):

1. Open Postman and create a new request `http:// 127.0.0.1:3000/read-id/644bc21958e221462949dd7d`.
2. Set the HTTP method to GET.
3. Click on the Send button to send the request.
4. Postman will display the response from the server in the Response section, which should contain the user details if the user is found, or a message saying "User not found" if the user is not found.




### Update User (by id):

1. Open Postman and create a new request of type PUT.
2. Enter the URL for the endpoint you want to test, e.g. `http://127.0.0.1:3000/update/<string:id>` where `<id>` is the ID of the user you want to update.
3. In the request body, select the "raw" format and choose "JSON" from the dropdown menu.
4. Enter the JSON object containing the updated user data, e.g. `{"name": "John", "email": "john@example.com", "password": "newpassword"}`.
5. Send the request and check the response in the "Body" tab. It should return a JSON object containing a "result" key with the message "User updated successfully" if the update was successful, or "User not found" if the user with the given ID does not exist in the database.



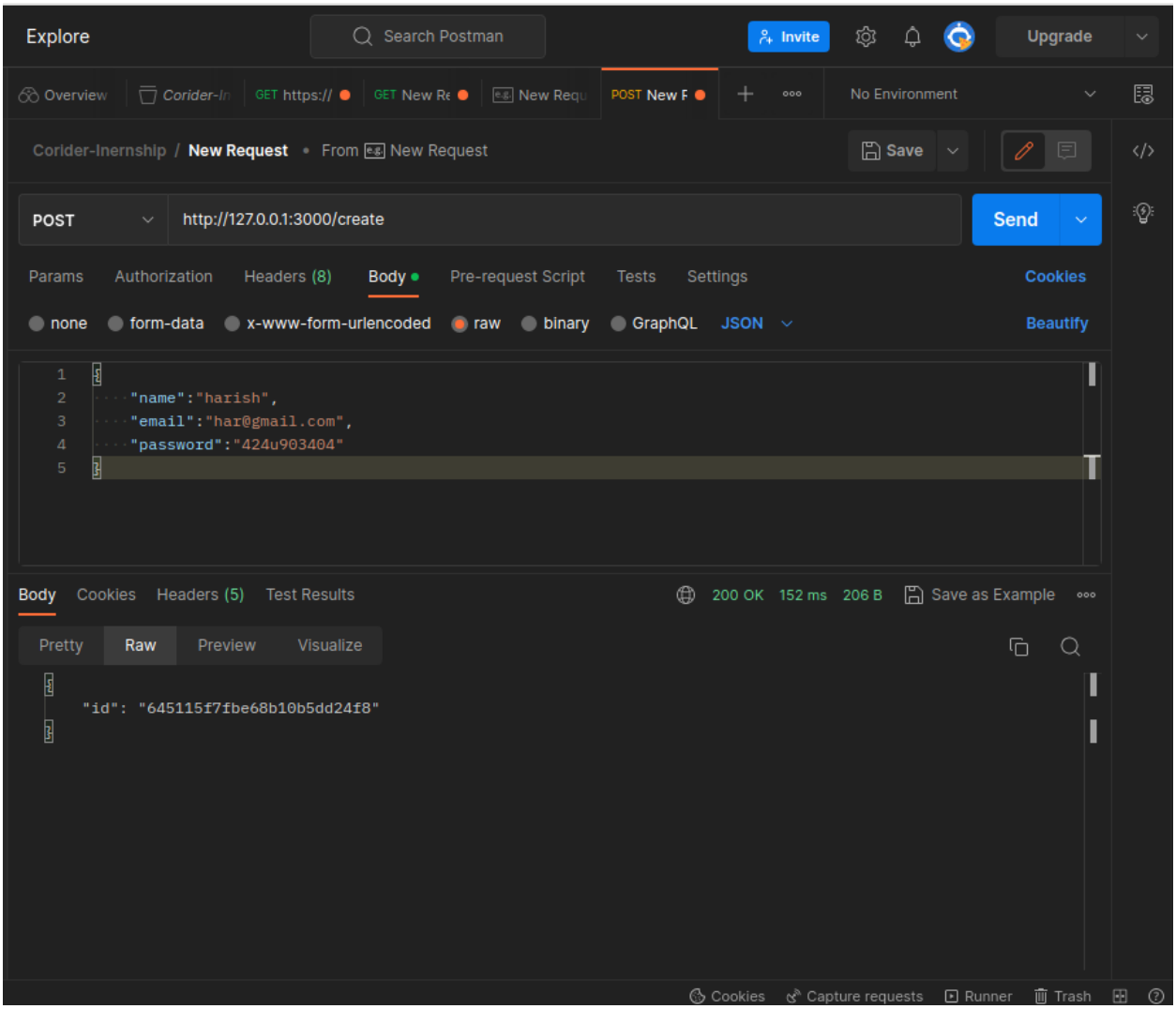

## Create User:

1. Open Postman and create a new request.
2. Set the request method to POST and enter the URL of your API endpoint, which should be `http://127.0.0.1:3000/create`.
3. Select the "Body" tab and choose "raw" as the input format. Set the content type to "JSON (application/json)".
4. In the request body, enter the user data in JSON format. For example:

```
{
  "name": "John Smith",
  "email": "john.smith@example.com",
  "password": "password123"
}
```

5. Click the "Send" button to send the request to your API endpoint.

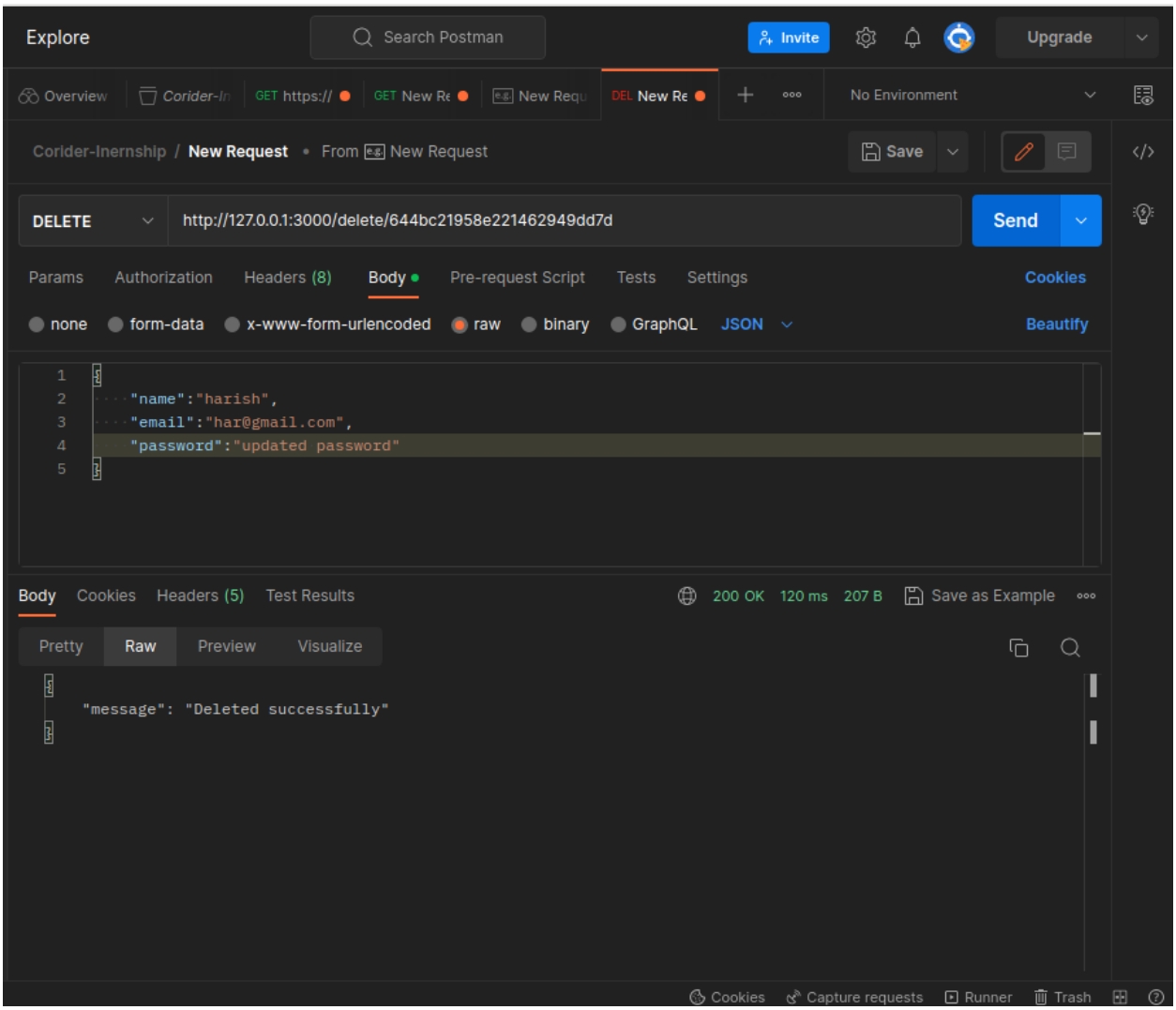
6. The response will be displayed in the "Body" tab of the response panel. It should contain a JSON object with a "result" field that says "User added successfully".



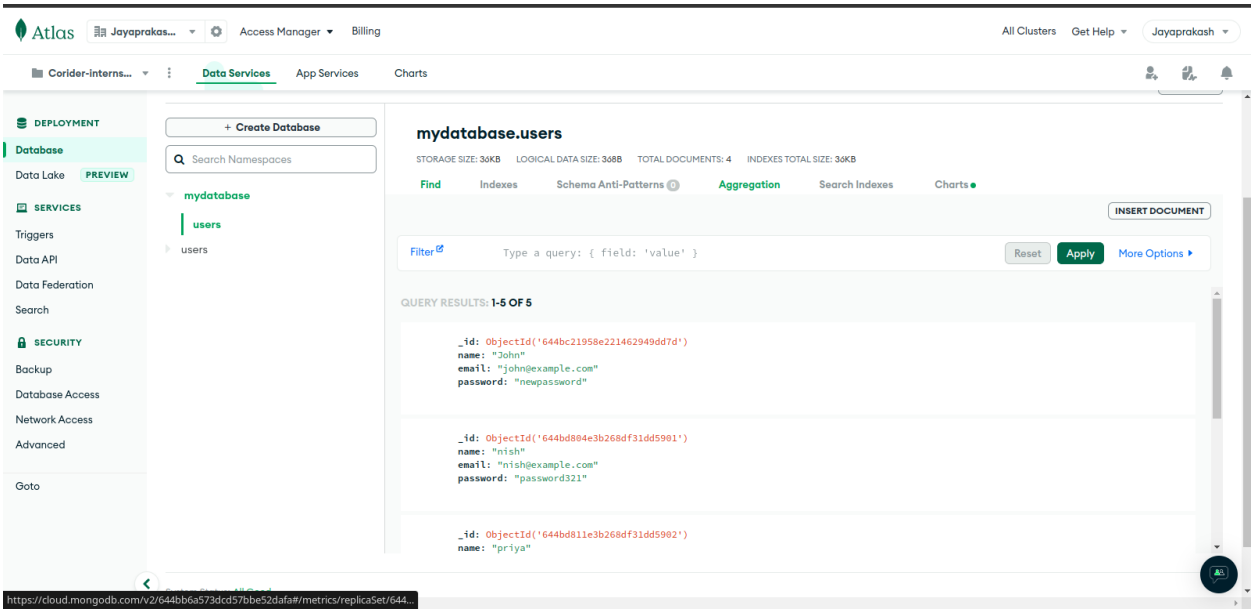

## Delete User( by id):

To evaluate the `delete_user` method in Postman, follow these steps:

1. Open Postman and create a new request.
2. Set the request method to "DELETE".
3. Enter the URL `http://127.0.0.1:3000/delete/<string:id>`, where `id` is the ID of the user you want to delete.
4. Click the "Send" button to send the request.
5. Check the response to see if the user was deleted successfully. The response will be in JSON format and will contain a message indicating whether the user was deleted or not.




## MonogDB database:


Atlas
Jayaprakas...
Access Manager
Billing
All Clusters
Get Help
Jayaprakash
Corider-interns...
Data Services
App Services
Charts
DEPLOYMENT
Database
Data Lake
PREVIEW
SERVICES
Triggers
Data API
Data Federation
Search
SECURITY
Backup
Database Access
Network Access
Advanced
Goto
+ Create Database
Search Namespaces
mydatabase
users
users
mydatabase.users
STORAGE SIZE: 36KB
LOGICAL DATA SIZE: 368B
TOTAL DOCUMENTS: 4
INDEXES TOTAL SIZE: 36KB
Find
Indexes
Schema Anti-Patterns
Aggregation
Search Indexes
Charts
INSERT DOCUMENT
Filter
Type a query: { field: 'value' }
Reset
Apply
More Options
QUERY RESULTS: 1-5 OF 5
_id: ObjectId('644bc21958e221462949dd7d')
name: "John"
email: "john@example.com"
password: "newpassword"
_id: ObjectId('644bd804e3b268df31dd5901')
name: "nish"
email: "nish@example.com"
password: "password321"
_id: ObjectId('644bd811e3b268df31dd5902')
name: "priya"
https://cloud.mongodb.com/v2/644bb6a573dcd57bbe52dafa#/metrics/replicaSet/644...