CISC 5550 Final Project Report

Features

What's New

- Docker compose
- User login system
- MangoDB database system
- Security improvement by using JSON Web Token

Environment

- Ubuntu 16.04 on VirtualBox
- Docker
- Flask 1.0.2
- Requests 2.21.0
- pymongo 3.8.0
- flask_pymongo 2.3.0

Deployment

docker_setup.sh

```
sudo apt-get remove -y docker docker-engine docker.io

if [ "$(lsb_release -r -s)" == "16.04" ]; then
    sudo apt-get -y update
    sudo apt-get install -y linux-image-extra-$(uname -r) linux-image-extra-
virtual
fi

sudo apt-get update -y
sudo apt-get install -y apt-transport-https ca-certificates curl software-
properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo apt-key fingerprint 0EBFCD88
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/
ubuntu $(lsb_release -cs) stable"
sudo apt-get update -y
sudo apt-get install -y docker-ce
sudo curl -L https://github.com/docker/compose/releases/download/1.21.0/docker-
compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
```

```
sudo chmod +x /usr/local/bin/docker-compose
sudo groupadd docker
sudo usermod -aG docker $USER
```

Reference: Get Docker CE for Ubuntu | Docker Documentation

Set up the docker environment

```
chmod +x docker_setup.sh
./docker_setup.sh
```

Dockerfile

```
FROM python:3.4
RUN touch /kube-build-image
RUN chmod -R a+rwx /usr/local/go/pkg
RUN mkdir /var/run/kubernetes && chmod a+rwx /var/run/kubernetes
ENV HOME /go/src/k8s.io/kubernetes
WORKDIR /usr/src/app
COPY requirements.txt requirements.txt
RUN pip install --upgrade pip
RUN pip install -r requirements.txt
COPY . .
RUN useradd -ms /bin/bash todo
USER todo
EXPOSE 4000
ENTRYPOINT ["python","index.py"]
```

docker-compose.yml

```
version: '3.5'
services:
  web_dev:
    build: .
    ports:
     - "4000:4000"
    volumes:
     - .:/usr/src/app
    environment:
     - ENV=development
```

```
      - PORT=4000
      - DB=mongodb://mongodb:27017/todoDev
      - SECRET='to-do-app-secret'
  mongodb:
    image: mongo:latest
    container_name: "mongodb"
    environment:
      - MONGO_DATA_DIR=/usr/data/db
      - MONGO_LOG_DIR=/dev/null
    volumes:
      - ./data/db:/usr/data/db
    ports:
        - 27017:27017
    command: mongod --smallfiles --logpath=/dev/null *# --quiet*
networks:
  default:
    name: web_dev
```

docker-compose.prod.yml

```
version: '3.5'
services:
  web_prod:
    build: .
    ports:
     - "3000:4000"
    volumes:
     - .:/app
    environment:
     - ENV=production
     - PORT=4000
     - DB=mongodb://localhost:27017/todoProd
  mongodb:
    image: mongo:latest
    container_name: "mongodb"
    environment:
      - MONGO_DATA_DIR=/usr/data/db
      - MONGO_LOG_DIR=/dev/null
    volumes:
```

```
      - ./data/db:/usr/data/db
    ports:
        - 27017:27017
    command: mongod --smallfiles --logpath=/dev/null *# --quiet*
networks:
  default:
    name: web_prod
```

Implementation

First Step

```
import os
import json
import datetime
from flask import Flask
from flask_pymongo import PyMongo

# create the flask object
app = Flask(__name__)
```

This part of code would create an app object of Flask, which would run the server by the code from index.py.

```
import os
import sys
import requests
from flask import jsonify, request, make_response, send_from_directory

ROOT_PATH = os.path.dirname(os.path.realpath(__file__))
os.environ.update({'ROOT_PATH': ROOT_PATH})
sys.path.append(os.path.join(ROOT_PATH, 'modules'))

import logger
from app import app

# Create a logger object to log the info and debug
LOG = logger.get_root_logger(os.environ.get(
    'ROOT_LOGGER', 'root'), filename=os.path.join(ROOT_PATH, 'output.log'))
```

```
# Port variable to run the server on.
PORT = os.environ.get('PORT')

@app.errorhandler(404)
def not_found(error):
    """ error handler """
    LOG.error(error)
    return make_response(jsonify({'error': 'Not found'}), 404)

@app.route('/')
def index():
    """ static files serve """
    return send_from_directory('dist', 'index.html')

@app.route('/<path:path>')
def static_proxy(path):
    """ static folder serve """
    file_name = path.split('/')[-1]
    dir_name = os.path.join('dist', '/'.join(path.split('/')[:-1]))
    return send_from_directory(dir_name, file_name)

if __name__ == '__main__':
    LOG.info('running environment: %s', os.environ.get('ENV'))
    app.config['DEBUG'] = os.environ.get('ENV') == 'development' # Debug mode
if development env
    app.run(host='0.0.0.0', port=int(PORT)) # Run the app
```

By putting the index.html and 404.html into our working directory, the rough version of the app is basically ready. Run the following commands, we can start the server:

```
docker-compose up --build
```

The output may look like this;

```
Building web_dev
Step 1/6 : FROM python:3.4
 ---> 2863c80c418c
Step 2/6 : ADD . /app
```

```
 ---> Using cache
 ---> 33352b6d855f
Step 3/6 : WORKDIR /app
 ---> Using cache
 ---> 24b5bc8d0b64
Step 4/6 : EXPOSE 4000
 ---> Using cache
 ---> d880187326eb
Step 5/6 : RUN pip install -r requirements.txt
 ---> Running in 77329798090f
Collecting Flask==0.12.2 (from -r requirements.txt (line 1))
  Downloading Flask-0.12.2-py2.py3-none-any.whl (83kB)
.
.
.
.
Step 6/6 : ENTRYPOINT ["python","index.py"]
 ---> Running in c5ce8be1c83d
Successfully built 6b3552b6c7fc
Successfully tagged pythonflask_web_dev:latest
Creating pythonflask_web_dev_1 ... done
Attaching to pythonflask_web_dev_1
web_dev_1  | 2019-05-05 15:58:12,066 - root - INFO - running environment:
development
web_dev_1  |  * Running on  [http://0.0.0.0:4000/](http://0.0.0.0:4000/)
(Press CTRL+C to quit)
web_dev_1  |  * Restarting with stat
web_dev_1  | 2019-05-05 15:58:12,425 - root - INFO - running environment:
development
web_dev_1  |  * Debugger is active!
web_dev_1  |  * Debugger PIN: 170-374-745
```

After that, access the http://localhost:4000/ in the browser. And we can see the index.html loaded.

Connect MongoDB

Update the **modules/app/**init**.py** file:

```
''' flask app with mongo '''
import os
```

```
import json
import datetime
from bson.objectid import ObjectId
from flask import Flask
from flask_pymongo import PyMongo

class JSONEncoder(json.JSONEncoder):
    ''' extend json-encoder class'''
    def default(self, o):
        if isinstance(o, ObjectId):
            return str(o)
        if isinstance(o, datetime.datetime):
            return str(o)
        return json.JSONEncoder.default(self, o)

# create the flask object
app = Flask(__name__)

# add mongo url to flask config, so that flask_pymongo can use it to make
connection
app.config['MONGO_URI'] = os.environ.get('DB')
mongo = PyMongo(app)

# use the modified encoder class to handle ObjectId & datetime object while
jsonifying the response.
app.json_encoder = JSONEncoder

from app.controllers import *
```

Write Controller for Users

Inside controllers directory, we will add*init.py* file like this.

This will import all the routes in all the files inside controllers directory. When controllers module will be imported, it will automatically define all the routes.

Here we import app & mongo object from app module. mongo object can be used to query into the database. General query goes like this:

> mongo.db.<collection>.<query>

Where <collection> is collection name. For us its 'users' at the moment. <query> will be anything like 'find', 'update', 'delete' etc.

We get the **ROOT_PATH** from environment variable, and define a logger with 'output.log'

file in root path.

```
ROOT_PATH = os.environ.get('ROOT_PATH')
LOG = logger.get_root_logger(
    __name__, filename=os.path.join(ROOT_PATH, 'output.log'))
```

We register a route with GET, POST, DELETE & PATCH methods allowed. Inside the routine, we check if the method is GET, we find the user from the database with query sent in request arguments, and return the user data with status code 200.

```
@app.route('/user', methods=['GET', 'POST', 'DELETE', 'PATCH'])
def user():
    if request.method == 'GET':
        query = request.args
        data = mongo.db.users.find_one(query)
        return jsonify(data), 200

    data = request.get_json()
    if request.method == 'POST':
        if data.get('name', None) is not None and data.get('email', None) is
not None:
            mongo.db.users.insert_one(data)
            return jsonify({'ok': True, 'message': 'User created
successfully!'}), 200
        else:
            return jsonify({'ok': False, 'message': 'Bad request
parameters!'}), 400
```

For all other request methods, we need to get the request data using:

data = request.get_json()

Next we check if the request data contains 'name' & 'email' of a user. If yes, we insert the user document and return success message. Else, we give a bad request response.

```
    if request.method == 'DELETE':
        if data.get('email', None) is not None:
            db_response = mongo.db.users.delete_one({'email': data['email']})
            if db_response.deleted_count == 1:
                response = {'ok': True, 'message': 'record deleted'}
            else:
```

```
                response = {'ok': True, 'message': 'no record found'}
            return jsonify(response), 200
        else:
            return jsonify({'ok': False, 'message': 'Bad request
parameters!'}), 400
```

Finally, the PATCH method is used to modify the user record. If the 'query' exists in request data, we modify the record matching that query with the 'payload' given in request data.

```
    if request.method == 'PATCH':
        if data.get('query', {}) != {}:
            mongo.db.users.update_one(
                data['query'], {'$set': data.get('payload', {})})
            return jsonify({'ok': True, 'message': 'record updated'}), 200
        else:
            return jsonify({'ok': False, 'message': 'Bad request
parameters!'}), 400
```

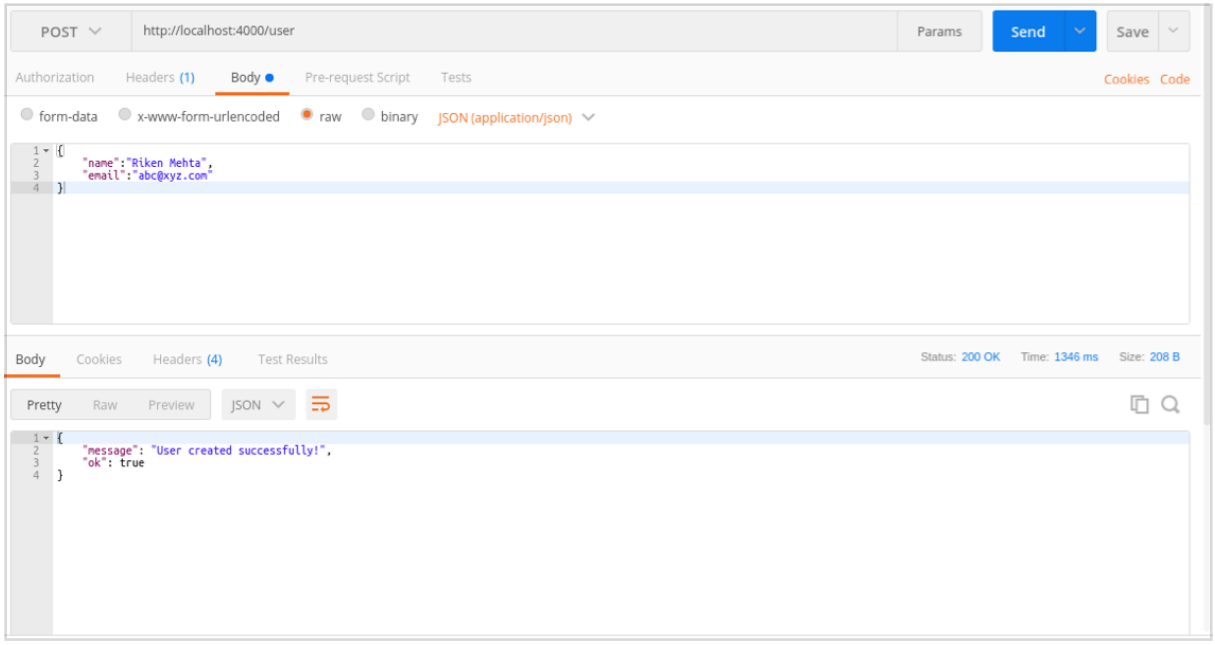
Support mongoDB as a service by using docker-compose files

In this project, Dockerfile is used to launch the docker container. And the commands in every line of this file represents some action to be taken at the time of launch.
Docker takes the python:3.4 image **FROM** docker repository, **ADD** every file of the current directory into*/usr/src/app*directory of the container we are about to launch. Make the same directory the **working directory** and EXPOSE 4000 port to the host os. Now it installs the required python packages. And defines the **ENTRYPOINT** as the index.py file.
Then, in the docker-compose file, we define a new service called mongoDB. And we add the additional flask_pymongo dependency in requirements.txt file.
Start the app using docker-compose command.

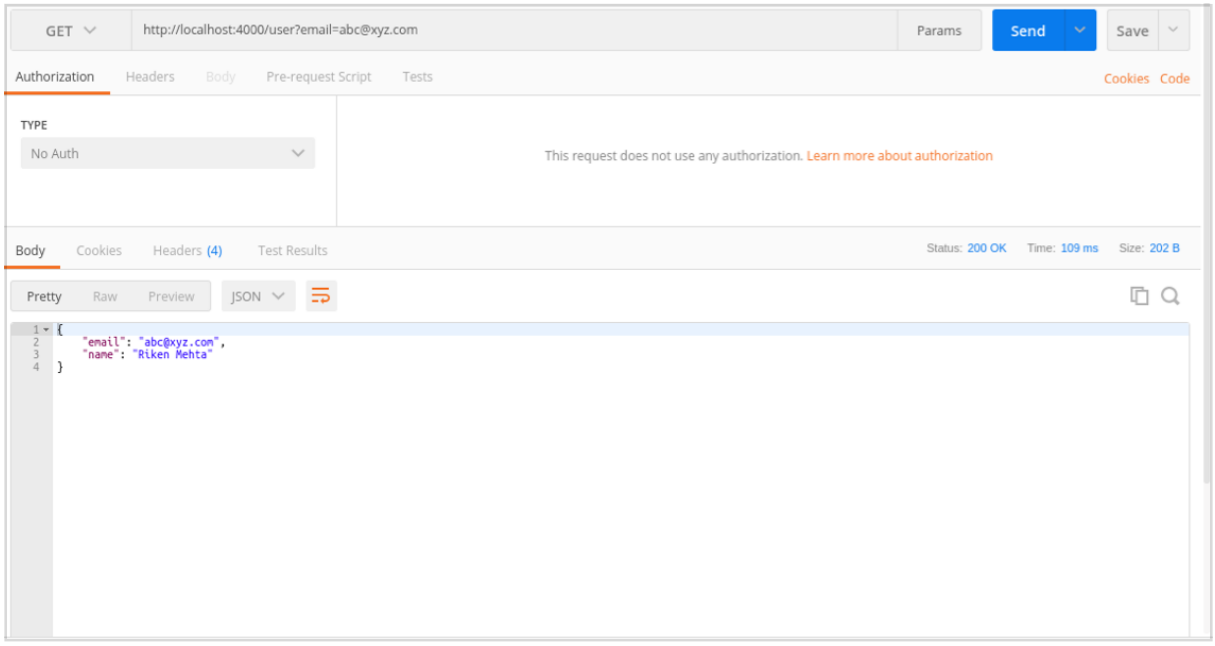
```
docker-compose up –build
```

API Request & Response Result

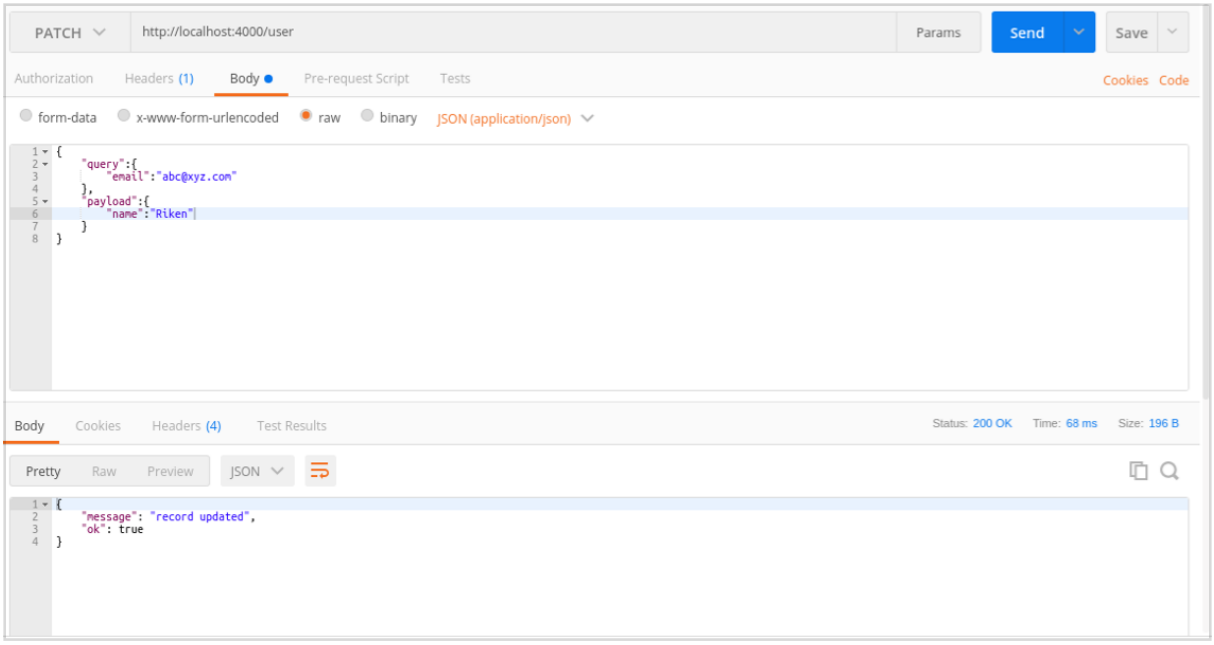
POST


POST
http://localhost:4000/user
Params
Send
Save
Authorization
Headers (1)
Body
Pre-request Script
Tests
Cookies
Code
form-data
x-www-form-urlencoded
raw
binary
JSON (application/json)
{
"name":"Riken Mehta",
"email":"abc@xyz.com"
}
Body
Cookies
Headers (4)
Test Results
Status: 200 OK
Time: 1346 ms
Size: 208 B
Pretty
Raw
Preview
JSON
{
"message": "User created successfully!",
"ok": true
}

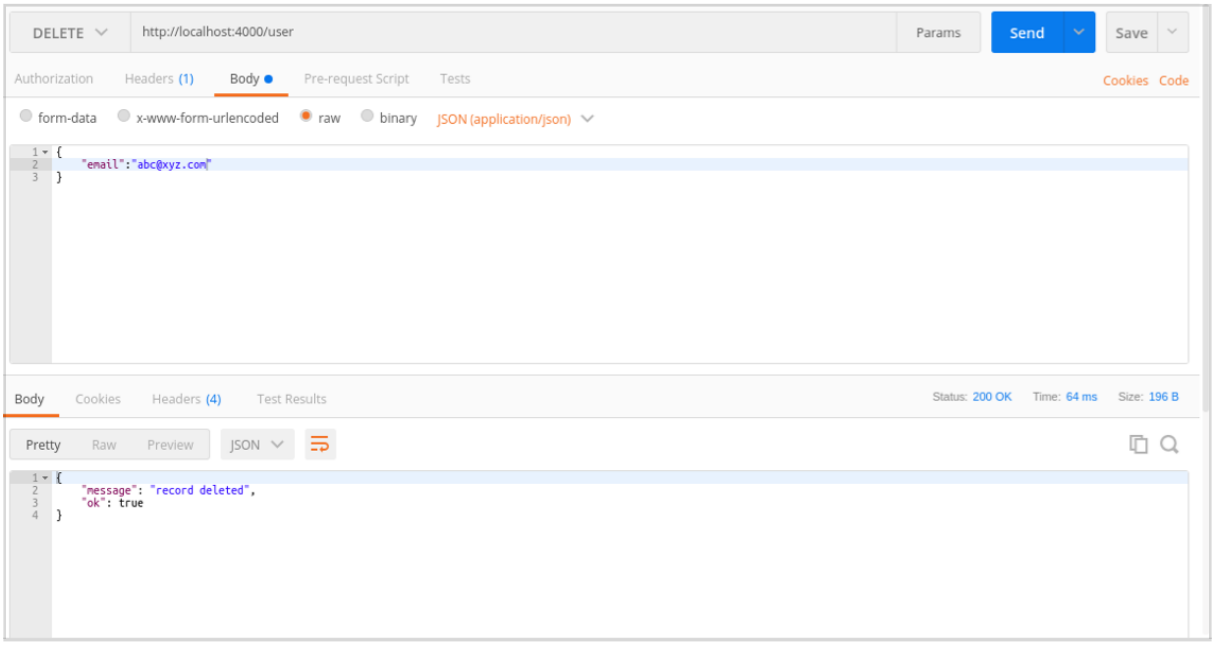
GET


GET
http://localhost:4000/user?email=abc@xyz.com
Params
Send
Save
Authorization
Headers
Body
Pre-request Script
Tests
Cookies
Code
TYPE
No Auth
This request does not use any authorization. Learn more about authorization
Body
Cookies
Headers (4)
Test Results
Status: 200 OK
Time: 109 ms
Size: 202 B
Pretty
Raw
Preview
JSON
{
"email": "abc@xyz.com",
"name": "Riken Mehta"
}

PATCH



DELETE



Security Improvement

> JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the**HMAC**algorithm) or a public/private key pair using**RSA**or**ECDSA**.
>
> Reference: JSON Web Token Introduction - jwt.io
>
> ### JWT structure
>
> In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

- Header
- Payload

- Signature

  Therefore, a JWT typically looks like the following.

> xxxxx.yyyyy.zzzzz

Integrate Flask JWT Extended Module with our Server

We will update our **modules/app/**init**.py** to initialize the flask-jwt-extended object. Import JWTManager from flask*jwt*extended module. Also, we will need to encrypt the password set by the user before saving it to database. It is considered as bad practice & a major security threat to save the password as it is. So we will be using flask_bcrypt module for that.

```
import os
import json
import datetime
from bson.objectid import ObjectId
from flask import Flask
from flask_pymongo import PyMongo
from flask_jwt_extended import JWTManager
from flask_bcrypt import Bcrypt
```

There are couple of config variables we have to add in app object and then initialize the instance of JWTManager. Same way we also initialize the Bcrypt object. Therefore, we have just integrated both the module with our flask server.

```
*# create the flask object*
app = Flask(__name__)
app.config['MONGO_URI'] = os.environ.get('DB')
app.config['JWT_SECRET_KEY'] = os.environ.get('SECRET')
app.config['JWT_ACCESS_TOKEN_EXPIRES'] = datetime.timedelta(days=1)
mongo = PyMongo(app)
flask_bcrypt = Bcrypt(app)
jwt = JWTManager(app)
app.json_encoder = JSONEncoder
```

Use Jsonschema to Validate the API Request

As the parameters in request object becomes more complex, it's hard to check each entity to see if there's no problem. Besides, we also need to verify if the format and type of the data passed is valid.

That's why we are using jsonschema to improve the progress. Under the directory, create a new folder called schemas.

We will be defining the schema and also it's validation functions for every controller in different files. Let's start by creating a file named user.py inside schemas directory.

```
from jsonschema import validate
from jsonschema.exceptions import ValidationError
from jsonschema.exceptions import SchemaError

user_schema = {
    "type": "object",
    "properties": {
        "name": {
            "type": "string",
        },
        "email": {
            "type": "string",
            "format": "email"
        },
        "password": {
            "type": "string",
            "minLength": 5
        }
    },
    "required": ["email", "password"],
    "additionalProperties": False
}


def validate_user(data):
    try:
        validate(data, user_schema)
    except ValidationError as e:
        return {'ok': False, 'message': e}
    except SchemaError as e:
        return {'ok': False, 'message': e}
    return {'ok': True, 'data': data}
```

The code above basically declare the schema & write a function to validate it. If there are some error, we return the error message along with error flag.

Creating User Registration & Authentication Route

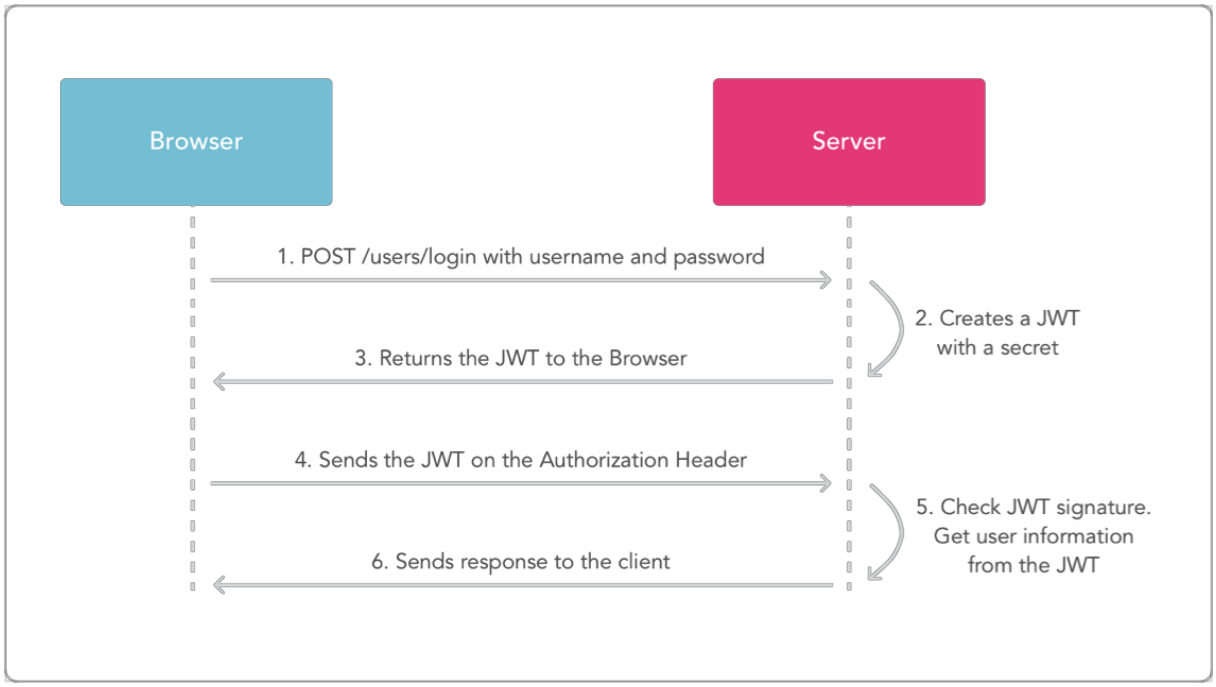
Now, let's see how we can add a register route in the "modules/controllers/user.py" file. At the top, import the validation method we defined in user schema file.

```
from app.schemas import validate_user
```

Then define the route like this. Basically we just moved the POST method of previously written user route in a separate route.

At first, we check if the received data is valid & in required format. If yes, we go on encrypting the password using flask_bcrypt object we created in app.py file. And then we store the data in our database. If there are any error with the request object, we respond the request with bad request parameters message.

Now, let's create a auth route, which will be called the very first time to generate the JSON token. So, the basic pipeline will be something like this.



Reference: JSON Web Token Introduction - jwt.io

In our case, '/auth' will be the login route. We will create a JWT and return that to the browser. Now, on all the subsequent requests, we will check if the same JWT exists in the Authorization header of the request. The '/auth' route will be something like this:

```
@app.route('/register', methods=['POST'])
**def** register():
    ''' register user endpoint '''
    data = validate_user(request.get_json())
    if data['ok']:
        data = data['data']
        data['password'] = flask_bcrypt.generate_password_hash(
            data['password'])
        mongo.db.users.insert_one(data)
```

```
        return jsonify({'ok': True, 'message': 'User created successfully!'}),
200
    else:
        return jsonify({'ok': False, 'message': 'Bad request parameters:
{}'.format(data['message'])}), 400
```

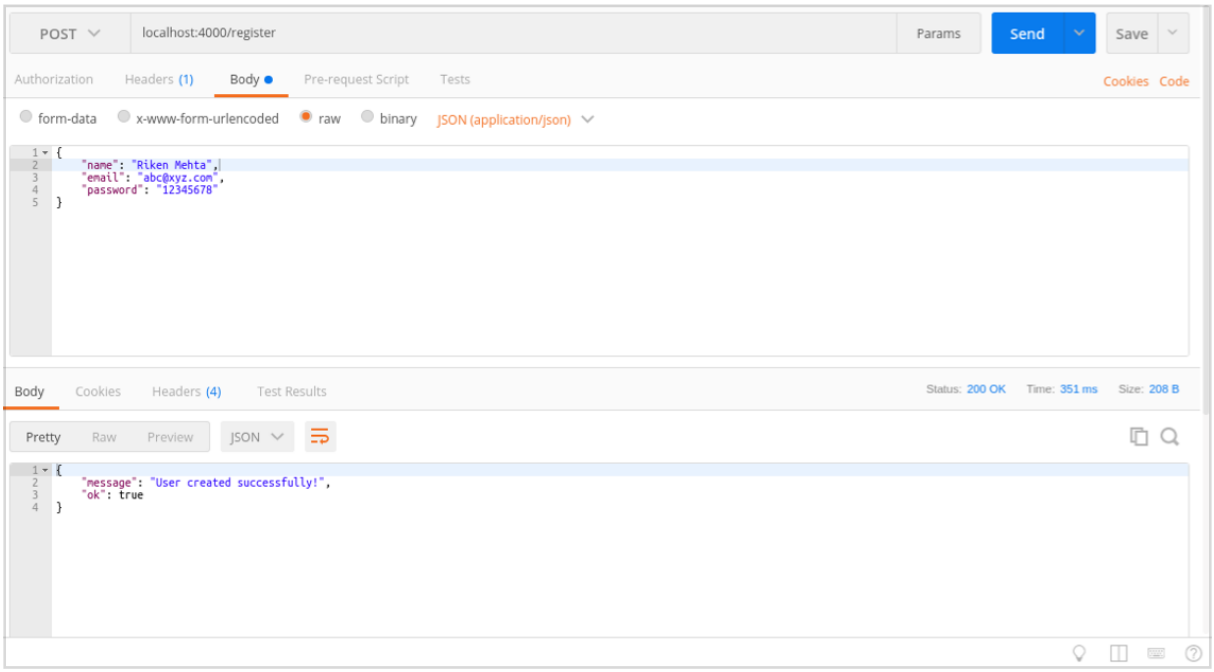
This route will first validate the data. Then see if the user with given username exists in the database, and also check if the password matches. If both conditions passed, it will create access*token & refresh*token. Basically, access_token is the JWT created using the user object.

To start the web server, run the docker compose command with build argument.

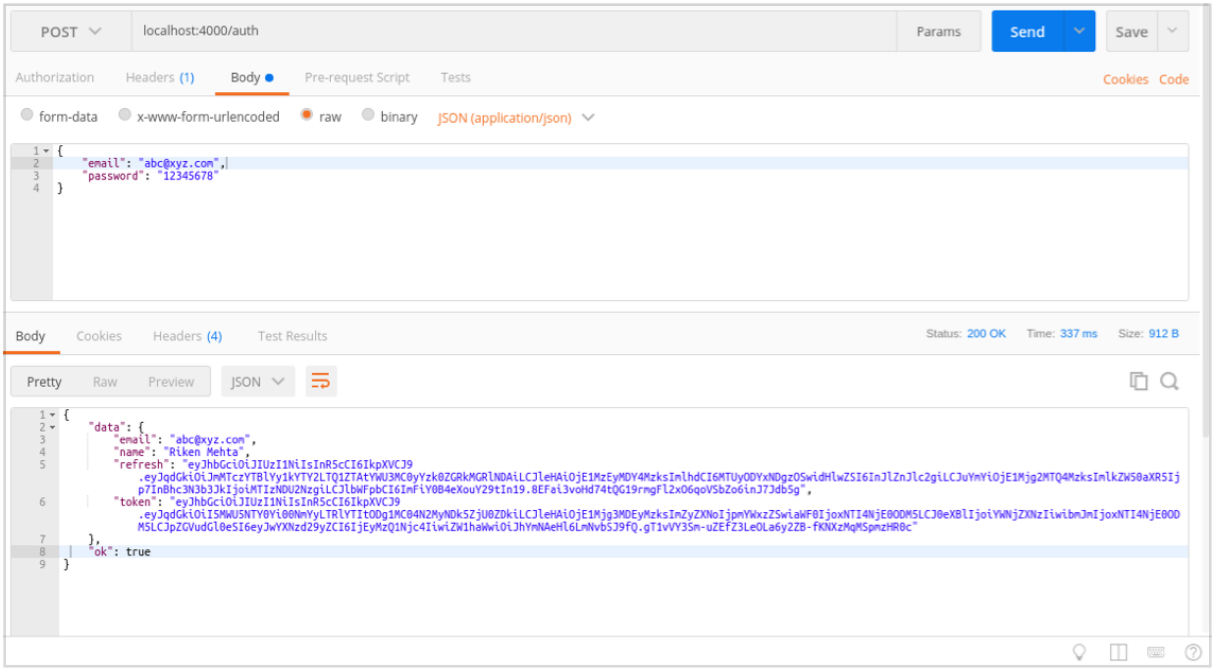
docker-compose up –build

Validation Result

Registering User



Authenticating User



How to Run

Install the npm dependencies

```
yarn install
```

To start and load app from docker container

```
yarn run build

docker-compose up --build

yarn run serve
```