Simple FastAPI Py with Pytest

This is a simple REST API built with Python and FastAPI and SQLAlchemy for CRUD operations (Create, Read, Update, Delete) on users. I fork the application from BaseMax/SimpleFastPyAPI. I then expand / add these things:

- Pytest with requests library, to test most of the API endpoint
- Integration with allure report for pytest
- pipenv for consistent working environment
- Jenkinsfile, with example of docker compose file to run jenkins locally in docker

I create this to showcasing my knowledge of these tools, and also for my own documentation. Feel free to fork it!

Preparation

- (STRICTLY REQUIRED!!!) python 3.10.x

  Version below / above may not supported / causing some issue, this is because of limitation from some of the dependencies that this app needed.

- Pipenv, you can refer from here
- Allure2, to serve the allure report
- Latest docker engine / desktop if you want to try the jenkins in docker setup

Installation

Clone this repository to your local machine:

```
git clone https://github.com/semutmerah/SimpleFastPyAPI.git
```

Change into the project directory:

```
cd SimpleFastPyAPI
```

Install the project dependencies:

```
pipenv install
```

Make a copy of .env.example, set the copy name to .env inside the same directory as the source

Run the application:

```
pipenv run uvicorn main:app --reload
```

The application will start and should be available at http://localhost:8000.

Run Pytest

Execute pytest with allure report output

You can see the config on pytest.ini file in the root directory

Make sure to run the application first (leave it run), then execute this command:

```
pipenv run pytest tests
```

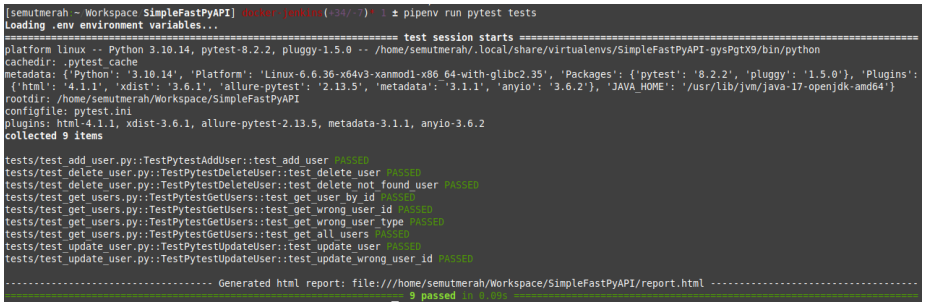

Figure 1: Example of pytest execution result

Serve allure report

```
allure serve allure-results
```

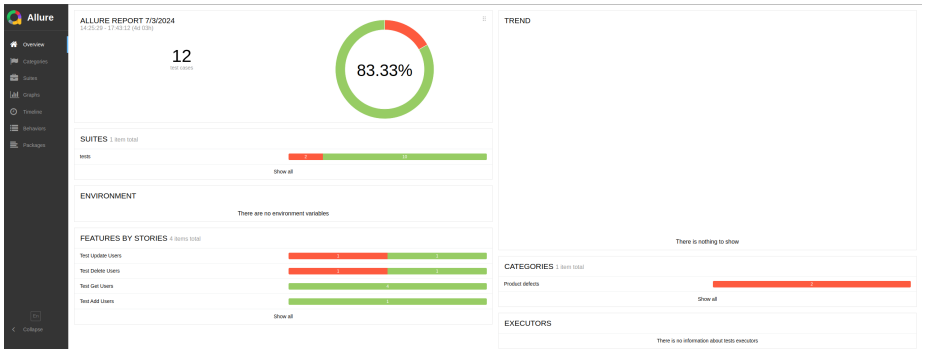

Figure 2: Example of allure report

Jenkins in Docker

Setup initial data for jenkins master

```
mkdir jenkins_master
```

```
docker network create jenkins
```

```
docker pull jenkins/jenkins:2.464-jdk17

docker run --network=jenkins -d --name jenkins-master -v $(pwd)/jenkins_master:/var/jenkins_

docker exec jenkins-master cat /var/jenkins_home/secrets/initialAdminPassword
```

Access jenkins master at http://127.0.0.1:8080, insert the initialadminpassword from the command above, set new username and password, and then install the recommended plugins

If successfull, then add new node until you see the JENKINS_SECRET code. You can follow along these documentation on how to get one.

You can then put the JENKINS_SECRET to file compose.yaml on line 23

You can now stop and remove the container, the data should be saved on jenkins_master directory that we have set. Our docker compose will used this data, so you don't have to redo the setup.

```
docker stop jenkins-master

docker rm jenkins-master
```

Build custom docker for inbound-agent

Since we want to run pytest inside the agent, then we need to have python. The original inbound-agent from jenkins official docker image doesn't have this, so we need to build one. I have prepare Dockerfile to do so. You only need to execute this command

```
docker build -t jenkins-agent .
```

Run jenkins with active agent

Now we only need to start docker compose to launch both jenkins master and jenkins slave

```
docker compose up -d
```

You can then access jenkins master url, set new job, reuse the Jenkinsfile that I have created in this repo

You can stop it with command

```
docker compose stop
```

API Endpoints

Retrieve a list of users:

```
GET /users
```

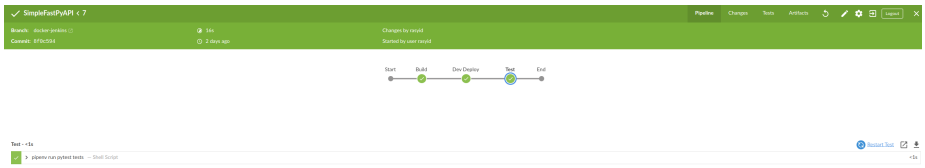

Figure 3: Example of jenkins result

Returns a list of all users in the system:

```
curl http://localhost:8000/users/ -H "Accept: application/json"
```

Response:


```
[
    {
        "email": "alice@example.com",
        "id": 1,
        "password": "password1",
        "name": "Alice"
    },
    {
        "email": "bob@example.com",
        "id": 2,
        "password": "password2",
        "name": "Bob"
    },
    {
        "email": "charlie@example.com",
        "id": 3,
        "password": "password3",
        "name": "Charlie"
    }
]
```


Retrieve details for a specific user:

```
GET /users/{user_id}
```

Returns details for a specific user with the given user_id:

```
curl http://localhost:8000/users/1 -H "Accept: application/json"
```

Response:


```
{
    "email": "alice@example.com",
```

```
    "id": 1,
    "password": "password1",
    "name": "Alice"
}
```

Add a new user

`POST /users`

Adds a new user to the system. The request body should include a JSON object with the following properties:

- `name` (string, required): the name of the user
- `email` (string, required): the email address of the user
- `password` (string, required): the password for the user

```
curl -X POST http://localhost:8000/users/
   -H 'Content-Type: application/json'
   -d '{"name":"Ali","password":"123456", "email": "AliAhmadi@gmail.com"}'
```

Response:

```
{
    "email": "AliAhmadi@gmail.com",
    "password": "123456",
    "id": 4,
    "name": "Ali"
}
```

Update an existing user

`PUT /users/{user_id}`

Updates an existing user with the given user_id. The request body should include a JSON object with the following properties:

- `name` (string): the new name for the user
- `email` (string): the new email address for the user

```
curl -X PUT http://localhost:8000/users/1
     -H "Accept: application/json"
     -d '{"name": "Reza", "email": "reza@yahoo.com"}'
```

Response:

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

Delete a user

`DELETE /users/{user_id}`

Deletes the user with the given user_id:

```
curl -X DELETE http://localhost:8000/2
```

Response:

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

License