M7024E Laboratory 4: Programming Cloud Services - RESTful APIs

Ameer Hamza, Wania Khan

December 24, 2021

Lab Report

Questions

1. What are microservices? Describe in detail the pros and cons of microservices architecture by giving examples.

Microservices are an architectural approach that structures an application as collection of small independent services that communicates over APIs. These microservices are highly maintainable, independently deployable and testable which allow easy scaling and fast development of applications. [1] [2]

There are several pros and cons of microservices architecture [3] [4] and few of them are discussed below:

1. **Independence**: as we know that in microservices architecture, each service is developed and deployed independently which makes it easier to update any one component and not the entire application.
    - Let's take an example of e-commerce web application, we have different microservices such as payment section, shopping cart, advertisements etc. So adding a new feature or updating any existing payment method does not require the entire e-commerce application to come down in order to deploy that update.
2. **Scalability**: since all the services are separate, it is easy to scale any specific service at appropriate times instead of the whole application.
    - Considering the same e-commerce example as above, during specific times of a year these websites have to scale their shopping cart services due to high demand which doesn't require the scaling of entire application.
3. **Flexibility**: since all the communication happens over the network, there is more flexibility in how each endpoint is composed.
    - For example, people with different skill sets can contribute to a project without operating outside of their areas of expertise.
4. **Robust**: applications developed using microservices architecture are more robust as a whole. Large applications mostly remains unaffected by the failure of a single service as there is a loose coupling between the services.
    - Assume an example of food restaurant application where an online service of ordering is down but their contact service is open which helps customers to order food by contacting customer service representatives.
5. **Complex communication**: as above mention that everything is now an independent service, it is needed to handle requests carefully between modules. The developers may have to deal with complications and need to write extra code to avoid disruption.
    - For example; when remote calls experience latency.

6. **Global Testing**: spinning up test environments is more involved with microservices-based applications due to increased number of nodes. As each dependent service has to be confirmed before testing can occur.
7. **Performance**: as it is known that communication happens over a network which is considerably slower than in memory. For many applications, it might not be a problem as network speeds improve, but this is something that should be taken into consideration.
8. **Refactoring**: refactoring an application across multiple services is harder than a monolith. For example, using different languages for different modules and components might cause issues when functionality from one service has to be ported over to another.

Exercise

1. Develop a simple RESTful API for any application.

2. Use docker container for deployment of the service.

3. Use microservices as an architectural paradigm to design a simple service(s). Describe and document that service.

4. Create a RESTful API to implement your service.

Overview

For this task we developed a simple user management application which allows a user to list, add, modify and delete the details of the users. Application is built using bootstrap for front-end, flask and RESTful APIs for back-end and MySQL for the database.

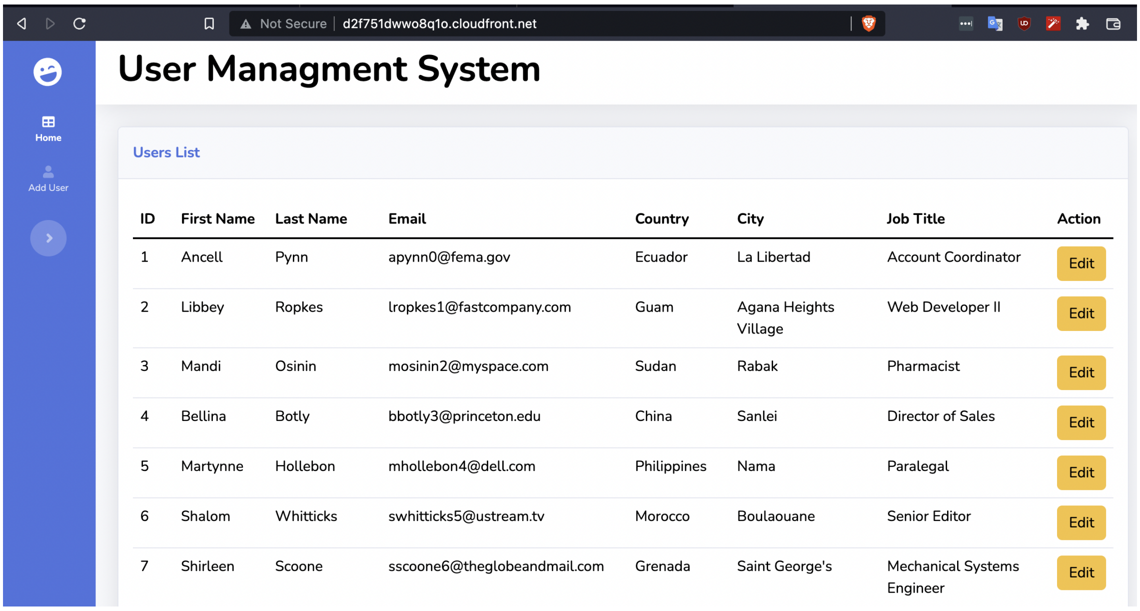

Figure 1: User Interface - Front Page

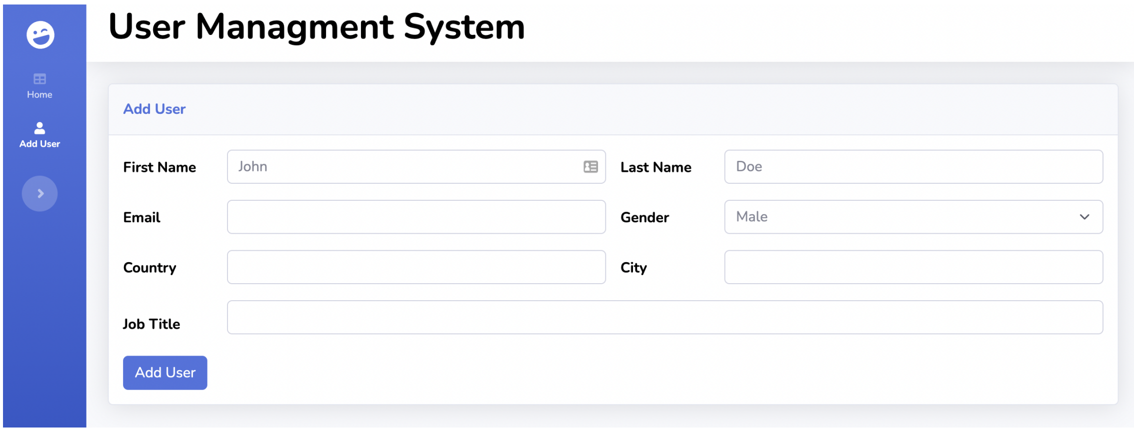

Figure 2: User Interface - Add a User

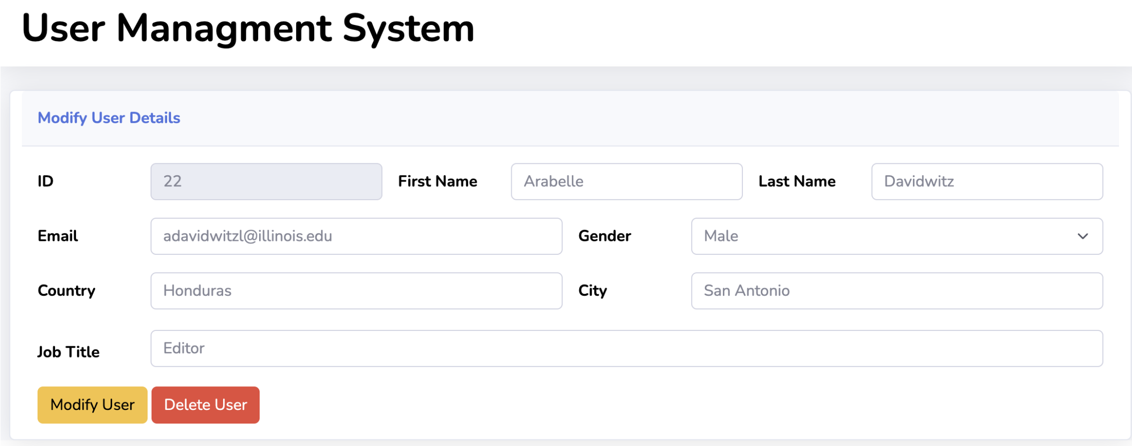

Figure 3: User Interface - Modify and Delete User

RESTful API

Our application has the following endpoints:

1. /api/v1/users – GET →List all users
2. /api/v1/users/<id>– GET →List details of a particular user
3. /api/v1/users – POST →Create a new user
4. /api/v1/users/<id>– PUT →Modify details of a particular user
5. /api/v1/users/<id>– DELETE →Remove a user

```python
# API Routes

# List all users
@app.route("/api/v1/users", methods=["GET"])
def index():⋯

# List a single user
@app.route("/api/v1/users/<id>", methods=["GET"])
def get_user_by_id(id):⋯

# Add a new user
@app.route("/api/v1/users", methods=["POST"])
def create_user():⋯

# Modify a user
@app.route("/api/v1/users/<id>", methods=["PUT"])
def update_user_by_id(id):⋯

# Delete a user
@app.route("/api/v1/users/<id>", methods=["DELETE"])
def delete_user_by_id(id):⋯
```

Figure 4: API Routes

Architecture

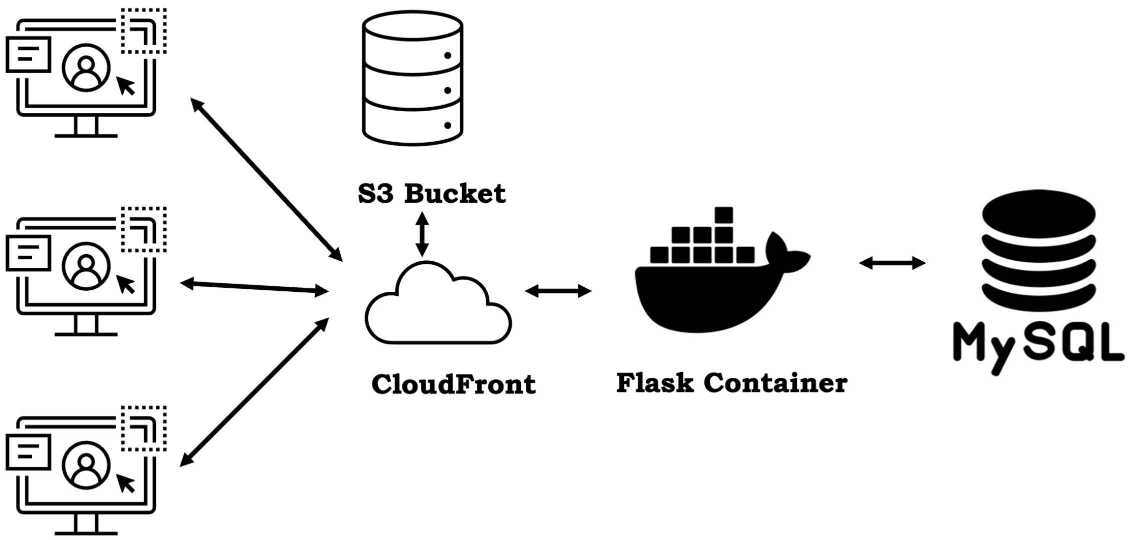

Figure 5: Architecture

There are three main components of our applications as show in figure 5:

Front End

The front-end of the application consists of HTML, CSS and JavaScript file. Since these are static files, they are deployed on S3 bucket followed by CloudFront to handle use traffic dynamically. Static files communicate with back-end using Restful APIs.

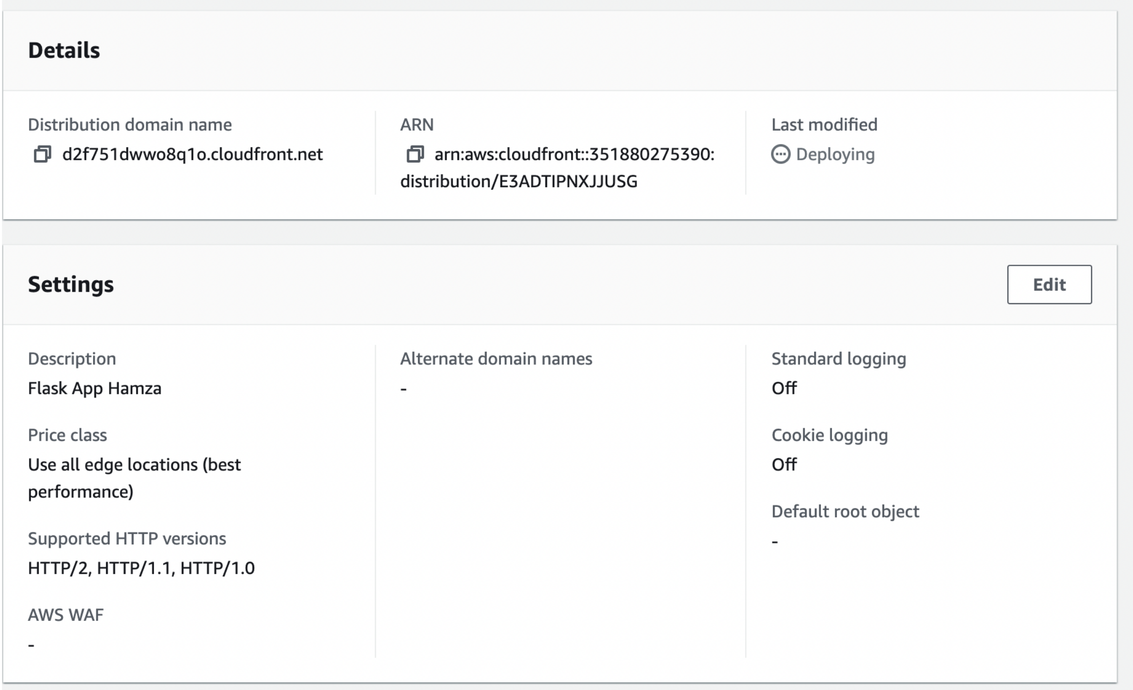

Figure 6: CloudFront

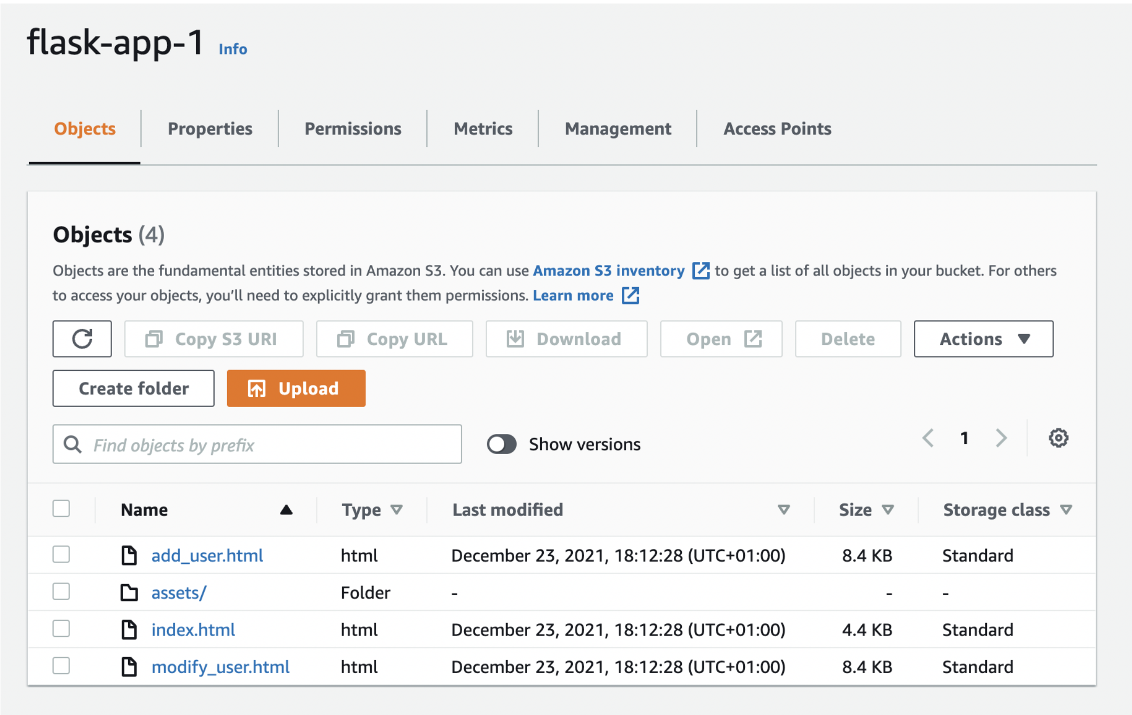

Figure 7: S3

Back End / Docker Container

We have used Python Flask for the back-end and to make Restful API. Once the application is complete it is dockerized using docker file and the image is deployed on EC2. Ideally AWS ECS should be used as it gives a lot of flexibility for scaling the application. Since we didn't have the permission for ECS, docker was installed on EC2 and the image was deployed. Link to docker image: https://hub.docker.com/r/ameerhamza360/flask

```dockerfile
# init a base image (Alpine is small Linux distro)
FROM python:3.9-alpine

# FROM python


RUN apk update && apk add gcc g++ make bash && rm -rf /var/cache/apk/*


# define the present working directory
WORKDIR /flask-img
# copy the contents into the working dir
ADD . /flask-img
# run pip to install the dependencies of the flask app
RUN pip install -r requirements.txt
# define the command to start the container

EXPOSE 5000


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

Figure 8: Docker File

Database

MySQL is used for this application though any relational database can be used without any code change since ORM is used in the Flask App. Relational database is used since our application has a fixed structure though NoSQL database will work equally well. MySQL server is deployed in a different EC2 instance. Database is not deployed using docker container as it introduces unnecessary complexity without relative benefit. Ideally we wanted to use Amazon RDS as it allows scaling of the database very easily. However, we didn't have the permission to use that service.

Hamza-MySQl	i-0f44c1e925145a4df	Running	t3.micro	2/2 checks passed
Hamza-Docker	i-0ad2e0331161512c1	Running	t4g.nano	2/2 checks passed

Figure 9: EC2 Instances for Docker and MySQL

References

[1] "Microservices." [Online]. Available: https://aws.amazon.com/microservices/

[2] "Microservices architecture." [Online]. Available: https://microservices.io/

[3] "Pros and cons microservices architecture." [Online]. Available: https://www.spkaa.com/blog/top-4-pros-cons-microservices-architecture/

[4] "Pros and cons of microservices." [Online]. Available: https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/