

Building & Running & Testing Flask Microservices with Docker and Jenkins CI/CD : Music Classifier

Realized by

Ben Salem Oussama

Introduction

In the ever-evolving landscape of software development, the demand for scalable and modular applications has led to the adoption of microservices architecture. This report documents the development and deployment process of a sophisticated application composed of three Flask microservices. Leveraging the power of Docker for containerization and Jenkins for Continuous Integration and Continuous Deployment (CI/CD), this project aims to streamline the development, testing, and deployment workflow.

Application Overview

The application at the heart of this report is designed as a collection of independent Flask microservices, each catering to specific functionalities. These microservices interact seamlessly to deliver a cohesive and responsive user experience. The use of Flask, a lightweight and extensible web framework for Python, ensures flexibility and ease of development.

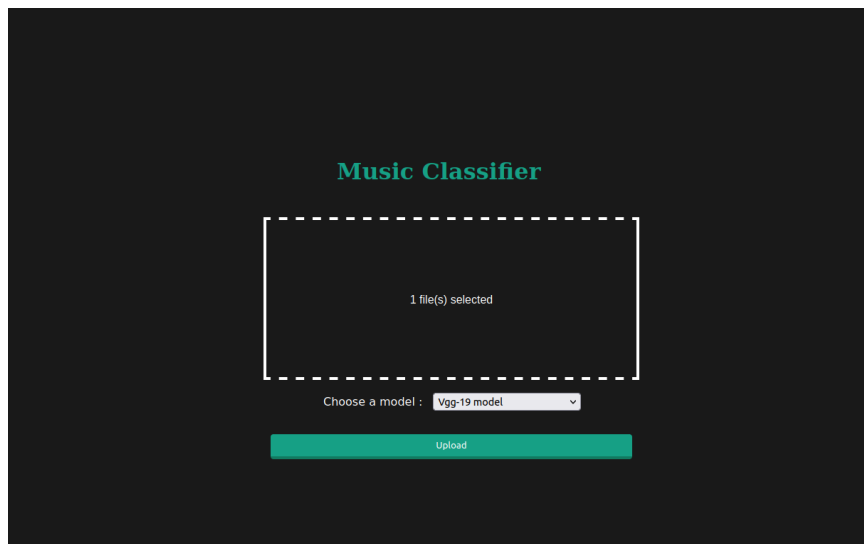
Flask Microservices Description

1. Frontend Microservice

The frontend microservice serves as the interface between the user and the application, handling the reception and processing of audio files in Base64 format. Its primary responsibility is to accept audio files submitted by the user, convert them to a suitable format, and then dispatch the processed audio data to one of the backend microservices for class prediction.

```
front_service > main.py
1 import requests
2 import os
3 import pickle
4 import joblib
5
6 import numpy as np
7 import librosa
8 from sklearn.preprocessing import StandardScaler
9
10 import io
11 import base64
12
13 app = Flask(__name__, static_url_path='/static')
14
15 @app.route('/')
16 def home():
17     return render_template('form.html')
18
19 @app.route('/classify', methods=['POST'])
20 def classify():
21     if request.method == 'POST':
22         audio_file=request.files['file']
23         audio_data = audio_file.read()
24         encoded_audio = base64.b64encode(audio_data).decode('utf-8')
25         payload = {'audio_data': encoded_audio}
26
27         selected_value = request.form['model_selected']
28
29         if selected_value == "none" :
30             return render_template('form.html')
31
32         if selected_value == "svm" :
33             svm_response = requests.post("http://svm_service:80/svm-base64", json=payload)
34             return jsonify(svm_response.text)
35
36         if selected_value == "vgg" :
37             svm_response = requests.post("http://vgg19_service:80/vgg19-base64", json=payload)
38             return jsonify(svm_response.text)
39
40
41
42
```

“Code Front-Service”



“FrontEnd Interface”

2. SVM Backend Microservice

The Support Vector Machine (SVM) backend microservice specializes in applying machine learning techniques to classify audio data. Upon receiving the processed audio data from the frontend, this microservice employs a trained SVM model to predict the class or category of the audio. SVMs are well-suited for classification tasks, making them an ideal choice for discerning patterns in audio data and providing accurate predictions.

3. VGG19 Backend Microservice

The VGG19 backend microservice is designed to handle the classification of audio files using a Convolutional Neural Network (CNN) architecture, specifically the VGG19 model. Leveraging deep learning, VGG19 excels in extracting intricate features from audio data, allowing for highly accurate classification. This microservice receives the preprocessed audio data from the frontend and employs the VGG19 model to predict the class or category of the audio content.

4. Conclusion

This architecture allows for a diverse and robust approach to audio classification, combining the strengths of traditional machine learning (SVM) and deep learning (VGG19) models. Each microservice plays a distinct role in the end-to-end workflow, contributing to the accurate and efficient prediction of audio class categories.

Containerisation

To enhance the portability and consistency of our application across different environments, Docker is employed for containerization. The Docker Compose tool orchestrates the deployment of our microservices, simplifying the configuration and management of the application stack. Each microservice is encapsulated within a Docker container, providing isolation and ease of deployment.

```
test_service:
  build:
    context: .
    dockerfile: ./Dockerfile
  ports:
    - "56742:80"
  volumes:
    - ./app
  command: pytest svm_service/tests/ vgg19_service/tests/ front_service/tests/
```


“Sequence of Docker-compose.yml : Test-service”

```
svm_service > Dockerfile > ...
1 FROM tiangolo/uwsgi-nginx-flask:python3.9
2
3 # Install requirements
4 COPY requirements.txt /tmp/requirements.txt
5 RUN pip install --upgrade pip
6 RUN pip install --no-cache-dir -r /tmp/requirements.txt
7 RUN pip install --upgrade numpy
8 RUN pip install pytest
9
10 ENV STATIC_URL /static
11 ENV STATIC_PATH /app/static
12
```

“SVM-Service Dockerfile”

Setting up Github

This GitHub repository setup establishes a solid foundation for collaborative development, version control, and project documentation. The repository serves as a central hub for our Flask microservices application, enabling seamless collaboration among team members and providing a transparent record of the project's evolution

 oussama938	last Commit !	b78c656 · 10 minutes ago	🕒 73 Commits
📁 front_service	last Commit !		24 minutes ago
📁 svm_service	last Commit !		24 minutes ago
📁 vgg19_service	last Commit !		24 minutes ago
📄 .gitignore	Initial commit		2 months ago
📄 Dockerfile	last_commit		3 days ago
📄 Jenkinsfile	last Commit !		10 minutes ago
📄 docker-compose.yml	last Commit !		24 minutes ago
📄 requirements.txt	last_commit		3 days ago

“Respository creation”

Setting up Jenkins for CI/CD

Jenkins, a widely adopted automation server, plays a pivotal role in automating the build, test, and deployment processes of our Flask microservices application. The setup involves the following key steps:

1. Jenkins Installation

The initial step is to install Jenkins on a server or a machine within the development environment. This can be achieved by downloading and installing Jenkins from the official website or using package managers for specific operating systems. Once installed, Jenkins can be accessed through a web browser, and the setup process continues through the Jenkins web interface.

2. Initial Configuration

Upon accessing the Jenkins web interface, an initial setup wizard guides users through the configuration process. This includes setting up an initial administrative user, defining the URL for Jenkins, and installing essential plugins. Jenkins plugins are critical for extending functionality, and in our setup, we installed plugins relevant to Flask, Docker, and Git integration.

3. Creating a Jenkins Pipeline

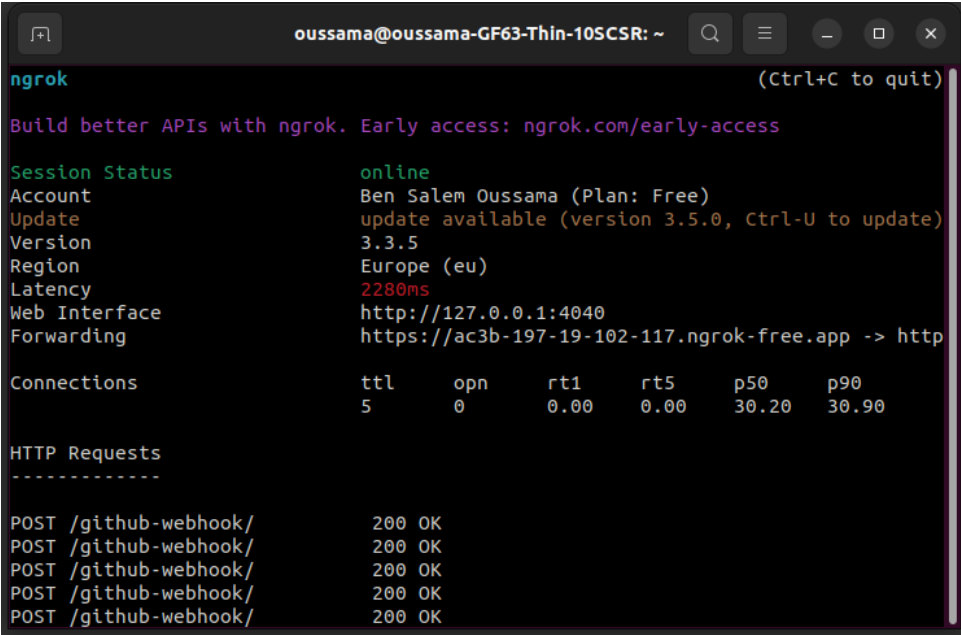
The heart of our CI/CD setup is the Jenkins pipeline. A pipeline is a series of automated steps that define the entire build, test, and deployment process. It is typically defined using a Jenkinsfile, which can be included in the project repository. Our Jenkins pipeline includes stages such as code checkout, building Docker images, running tests, and deploying to various environments.

```
Jenkinsfile
1 pipeline{
2   agent any
3   stages{
4     stage('Checkout'){
5       steps{
6         checkout scmGit(branches: [[name: '*/master']], extensions: [], userRemoteConfigs: [[url: 'h
7       }
8     }
9     stage('Building Containers'){
10      steps{
11        sh 'docker-compose build'
12      }
13    }
14    stage('Testing and Running Containers'){
15      steps{
16        script{
17          sh 'docker-compose up -d'
18
19          def serviceName = 'test service'
20          def exitCode = sh(script: "docker-compose ps -q ${serviceName} | xargs docker inspect --t
21          echo "Exit code of ${serviceName}: ${exitCode}"
22
23          if(exitCode == 0){
24            echo 'Containers are Working !'
25          }
26          else{
27            echo 'Containers DOWN !!!'
28            sh 'docker-compose down'
29          }
30        }
31      }
32    }
33  }
34 }
35
36
37
```

“Pipeline CI”

4. Exposing Jenkins with Nginx

For external collaboration and seamless integration with third-party services, it's essential to expose our local Jenkins server to the internet. Ngrok, a powerful tunneling service, simplifies this process by creating a secure connection to our Jenkins instance and assigning it a publicly accessible URL.

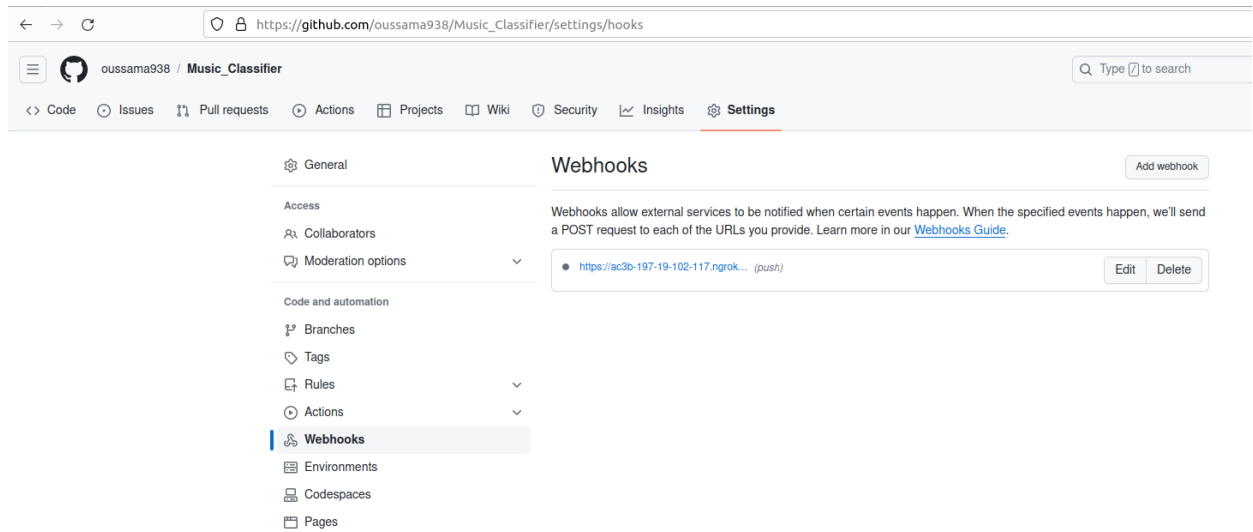


```
oussama@oussama-GF63-Thin-10SCSR: ~  
ngrok (Ctrl+C to quit)  
Build better APIs with ngrok. Early access: ngrok.com/early-access  
Session Status      online  
Account             Ben Salem Oussama (Plan: Free)  
Update              update available (version 3.5.0, Ctrl-U to update)  
Version             3.3.5  
Region              Europe (eu)  
Latency              2280ms  
Web Interface        http://127.0.0.1:4040  
Forwarding           https://ac3b-197-19-102-117.ngrok-free.app -> http  
  
Connections          ttl    opn    rt1    rt5    p50    p90  
                    5      0      0.00   0.00   30.20   30.90  
  
HTTP Requests  
-----  
POST /github-webhook/ 200 OK  
POST /github-webhook/ 200 OK  
POST /github-webhook/ 200 OK  
POST /github-webhook/ 200 OK  
POST /github-webhook/ 200 OK
```

“Exposing Jenkins Internet”

5. Integrating with GitHub Webhooks

To enable automated builds triggered by code changes, Jenkins is integrated with our GitHub repository. Webhooks are set up on the GitHub side to notify Jenkins whenever a new commit is pushed. This integration ensures that our CI/CD pipeline is automatically initiated upon code changes, promoting a continuous and automated development workflow.



“Adding Webhook”

Scenario : Automated CI with Jenkins

Developer Commits Changes:

A developer working on the Flask microservices project completes a set of changes or introduces new features in their local development environment. After testing the changes locally, the developer commits the changes to a feature branch in the GitHub repository.

GitHub Webhook Triggers Jenkins:

The GitHub repository is configured with a webhook that notifies Jenkins whenever a new commit is pushed to the repository. The webhook payload includes information about the commit, such as the commit ID, branch, and repository details.

Jenkins Receives Webhook and Triggers a Build:

Jenkins, configured to listen for GitHub webhooks, receives the notification about the new commit. The configured Jenkins pipeline is triggered automatically in response to the webhook.

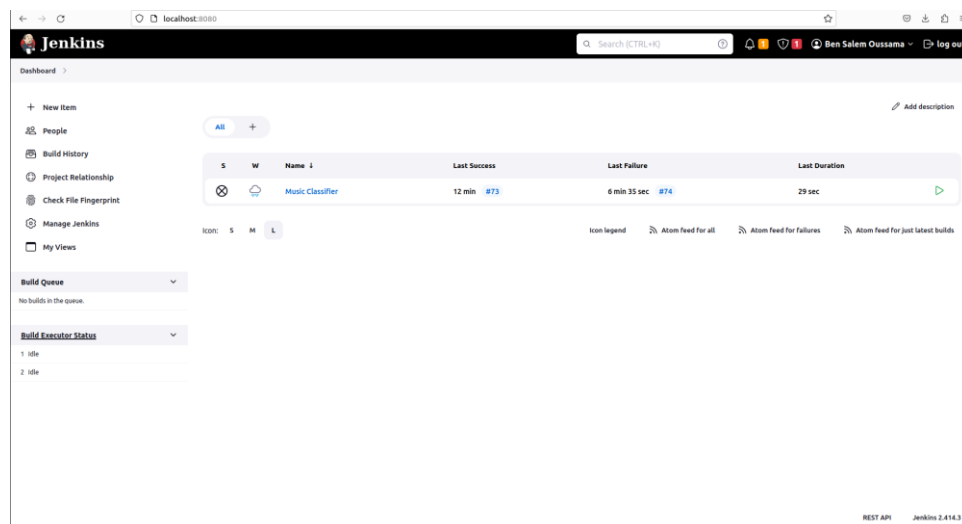
```
root@oussama-GF63-Thin-10SCSR:/var/www/FlaskApplication# git add .
root@oussama-GF63-Thin-10SCSR:/var/www/FlaskApplication# git commit -m 'last Commit !'
[master 9728d5d] last Commit !
1 file changed, 1 insertion(+), 1 deletion(-)
root@oussama-GF63-Thin-10SCSR:/var/www/FlaskApplication# git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 293 bytes | 293.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/oussama938/Music_Classifier.git
9f5b414..9728d5d master -> master
```

“Pushing new version”

Code Checkout :

The first stage of the Jenkins pipeline involves checking out the latest code from the GitHub repository. Jenkins pulls the code to the CI server. The build stage involves executing build scripts, installing dependencies, and preparing the environment for testing and deployment.

Automated Building & Testing :



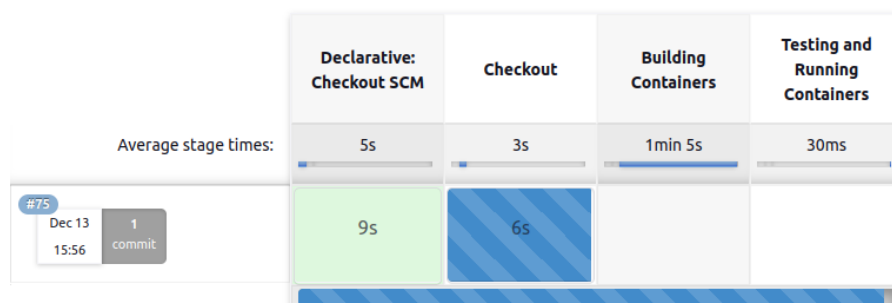

```

svm_service > tests > test_svm_service.py
1  from svm_service.main import app
2  import pytest
3  from flask import Flask, jsonify
4  import warnings
5  import base64
6  from pathlib import Path
7
8  @pytest.fixture
9  def client():
10     app.config['TESTING'] = True
11     with app.test_client() as client:
12         yield client
13
14  def test_svm_audio_base64(client):
15     with warnings.catch_warnings():
16         warnings.simplefilter("ignore", category=DeprecationWarning)
17         warnings.simplefilter("ignore", category=UserWarning)
18         # audio_file_path = 'reggae1.wav'
19         audio_file_path = Path(__file__).resolve().parent / 'reggae1.wav'
20         with open(audio_file_path, 'rb') as audio_file:
21             audio_data = audio_file.read()
22             audio_data_base64 = base64.b64encode(audio_data).decode('utf-8')
23             test_data = {'audio_data': audio_data_base64}
24
25             response = client.post('/svm-base64', json=test_data)
26
27             assert response.status_code == 200
28

```

“Unit test example code”

Stage View



“Building”

With the code successfully built, the next stage involves running automated tests. This ensures that the new changes do not introduce regressions and adhere to the project's quality standards.

Unit tests, integration tests, and any other relevant testing procedures are executed as defined in the Jenkins pipeline.

Conclusion

In the dynamic landscape of modern software development, the adoption of microservices architecture, coupled with robust CI/CD practices, presents an agile and scalable approach to building applications. The fusion of Flask microservices, Jenkins Continuous Integration (CI), and GitHub version control forms a symbiotic ecosystem that empowers development teams to streamline workflows, enhance collaboration, and ensure the consistent delivery of high-quality software.