

Project Documentation: Nginx Docker Microservices

CI/CD Pipeline

This project demonstrates a microservices-based architecture using Docker, NGINX, and GitLab CI to implement a scalable, robust system with a focus on API gateway functionality, monitoring, and state management.

1. **Service1:** A Java-based microservice for handling API requests.
2. **Service2:** A Python-based microservice managing state and metrics.
3. **NGINX and API Gateway:** Handles all incoming HTTP traffic. Acts as a reverse proxy to route requests to Service1 and Service2. Exposes REST-like APIs for testing and state management.

The CI/CD pipeline automates build, test, and deployment, ensuring consistent and reliable operations. Key features include monitoring, API testing, and state management for the system. The instructions below explain how to test and evaluate the system.

Instructions for the Teaching Assistant

Clone:

```
git clone -b project https://github.com/AnisulMahmud/Nginx-Docker-Microservices.git
```

```
cd Nginx-Docker-Microservices
```

Build and Run:

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

this will eventually build and run the system

Run tests:

```
docker-compose run test
```

Implemented Features

This project successfully implements the following:

1. **Core Microservices Functionality:** Service1 routes API requests, and Service2 manages states and provides metrics.
2. **API Gateway with NGINX:** Routes traffic securely and supports state management and metrics retrieval.
3. **CI/CD Pipeline:** Automated pipeline with build, test, and deploy stages using GitLab CI.
4. **API Testing Framework:** Python-based tests validate API endpoints, metrics, and state transitions.
5. **Monitoring and Logging:** Displays real-time metrics like service start time and request counts.
6. **User-Triggered Docker Shutdown:** A feature to stop containers directly from the browser interface.
7. **Log in:** Initially it will ask user to log in , but if the user set the state to init, then again it will ask for log in to run the system again and it won't affect the log information at all.
8. **CSS Enhancements:** Added custom CSS to improve the visual appearance of the browser interface, making the system more user-friendly and visually appealing.

Implemented Optional Features

- a. **Testing of Individual Services:**
 - Comprehensive testing of the APIs to validate both **service1** and **service2** functionality.
 - API testing ensured the functionality of endpoints like /state, /metrics, /run-log, and /request, which indirectly tested the individual services' behavior and integration through the API gateway.
- b. **Monitoring:**
 - Implemented a basic monitoring system accessible through the browser interface (port 8198).
 - Displayed critical metrics such as:
 - a. Start time of the service.
 - b. Total number of requests received since startup.
 - c. Current state of the system (INIT, RUNNING, PAUSED, SHUTDOWN).

Instructions for Examiner to Test the System

1. Eventually this command can build and start the system->

`docker-compose up --build`

But one can run this individually

- `docker-compose build --no-cache`
- `docker-compose up -d`

2. Access the services

Browser traffic: <http://localhost:8198>.

3. Validate tests: I use docker for running the test. There are two test files. So eventually with this command both test files will run perfectly.

`docker-compose run test`

4. Testing Using Curl:

`curl -X PUT localhost:8197/state \`

`-d "PAUSED" \`

`-H "Content-Type: text/plain" \`

`-H "Accept: text/plain"`

5. One can Access the metric in browser and also using curl

- <http://localhost:8197/metrics> (for browser)

- `curl http://localhost:8197/metrics` (Using curl)

Example Output:

Included screenshots showcasing the test results and the frontend interface for better visualization and understanding of the system's functionality.

```
anislmaahmud@Ubuntu: ~/Desktop/Nginx-Docker-Microservices$ docker ps
```

CONTAINER ID	IMAGE	NAMES	COMMAND	CREATED	STATUS	PORTS
131f833711d1	nginx-docker-microservices-nginx	nginx-docker-microservices-nginx-1	"/docker-entrypoint..."	26 seconds ago	Up 21 seconds	80/tcp, 0.0.0.0:8197-8198->8197-8198/tcp, ::
:8197-8198->8197-8198/tcp						
66d708bce65d	nginx-docker-microservices-service1-2	nginx-docker-microservices-service1-2-1	"java -cp src Main"	26 seconds ago	Up 24 seconds	8199/tcp
e85dc004402b	nginx-docker-microservices-service1-1	nginx-docker-microservices-service1-1-1	"java -cp src Main"	26 seconds ago	Up 24 seconds	8199/tcp
d958b58a5ac0	nginx-docker-microservices-service1-3	nginx-docker-microservices-service1-3-1	"java -cp src Main"	26 seconds ago	Up 24 seconds	8199/tcp
ce91f29d8796	nginx-docker-microservices-service2	nginx-docker-microservices-service2-1	"python app.py"	26 seconds ago	Up 24 seconds	0.0.0.0:5000->5000/tcp, ::5000->5000/tcp

Image 1: Running Containers

```

curl: (0) Could not resolve host: curl
● anisulmahmud@Ubuntu:~/Desktop/Nginx-Docker-Microservices$ curl localhost:8197/state -X PUT -d "PAUSED" -H "Content-Type: text/plain" -H "Accept: text/plain"
{"message":"No change in state"}
● anisulmahmud@Ubuntu:~/Desktop/Nginx-Docker-Microservices$ curl localhost:8197/state -X GET \
-H "Accept: text/plain"
{"state":"PAUSED"}
● anisulmahmud@Ubuntu:~/Desktop/Nginx-Docker-Microservices$ curl localhost:8197/metrics \
-H "Accept: text/plain"
{"Request Count":1,"Service Start Time":"2024-12-08 22:34:34","State":"PAUSED","Uptime (seconds)":"208.686508"}
● anisulmahmud@Ubuntu:~/Desktop/Nginx-Docker-Microservices$ curl localhost:8197/run-log \
-H "Accept: text/plain"
2024-12-08T22:34:38.588981+00:00: INIT->RUNNING
2024-12-08T22:34:38.763793+00:00: RUNNING->INIT
2024-12-08T22:34:38.770576+00:00: INIT->RUNNING
○ 2024-12-08T22:37:08.778799+00:00: RUNNING->PAUSED
anisulmahmud@Ubuntu:~/Desktop/Nginx-Docker-Microservices$

```

Image 2: Testing using curl

```

anisulmahmud@Ubuntu:~/Desktop/Nginx-Docker-Microservices$ docker-compose run test
[+] Building 0.0s (0/0)                                                                                               docker:default
[+] Creating 5/0
✓ Container nginx-docker-microservices-service1-1-1 Running 0.0s
✓ Container nginx-docker-microservices-service1-2-1 Running 0.0s
✓ Container nginx-docker-microservices-service1-3-1 Running 0.0s
✓ Container nginx-docker-microservices-service2-1 R... 0.0s
✓ Container nginx-docker-microservices-nginx-1 Runn... 0.0s
[+] Building 0.0s (0/0)                                                                                               docker:default
.....
-----
Ran 6 tests in 1.243s
OK

```

Image 3: Running the test files

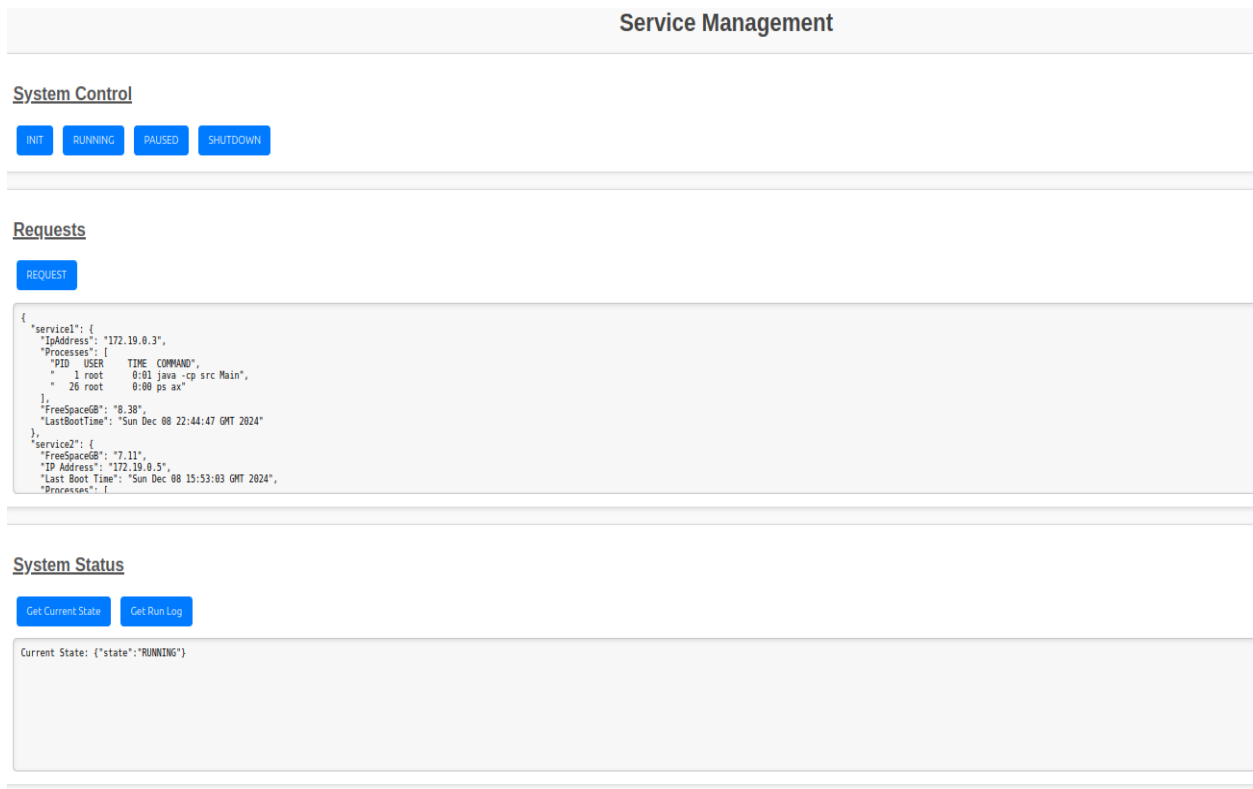


Image 4: Output in RUNNING state

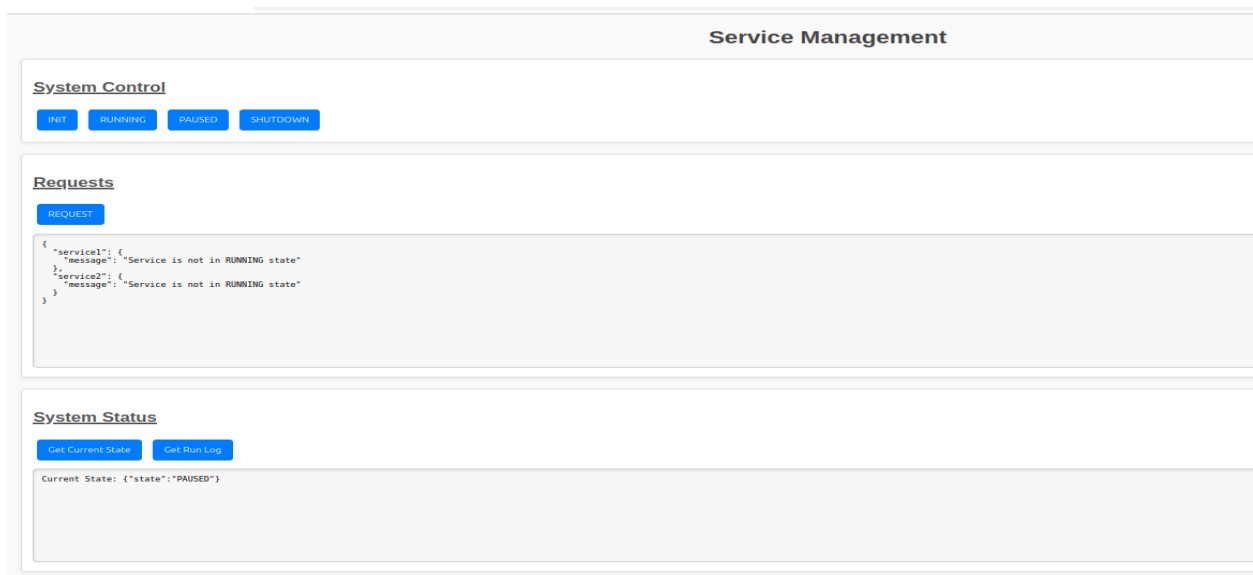


Image 5: Output in PAUSED state

JSON		Raw Data	Headers
Save	Copy	Collapse All	Expand All Filter JSON
Request Count:		2	
Service Start Time:		"2024-12-08 23:08:50"	
State:		"PAUSED"	
Uptime (seconds):		"176.077567"	

Image 6: Output of logging

Platform Details

- Virtualization Environment:
 - a. Virtual Machine: Oracle VirtualBox.
 - b. Operating System: Ubuntu (64-bit).
- Hardware Specifications:
 - a. Base Memory: 4GB allocated for the virtual machine.
 - b. CPU Architecture: x64.
- Tools and Software:
 - a. Docker Version: 27.3.1, build ce12230
 - b. Docker Compose Version: v2.22.0
 - c. IDE : VScode
- System Resources:

Sufficient memory and processing power allocated to ensure the smooth operation of Docker containers for service1, service2, the API gateway (nginx), and the testing framework.

Description of the CI/CD Pipeline

Version Management

- **Branches Used:** The project is managed using a Git branch structure. Development occurs on a dedicated **project** branch, ensuring isolation from the main production branch (main).
- **Purpose:** Using separate branches allows for iterative development and testing without disrupting stable releases.

Building Tools

- **Tools Used:** Docker and Docker Compose.

Build Process:

- The build stage of the pipeline runs the docker-compose build command to build all Docker services from their respective Dockerfiles. All files has its own Docker file.
- The pipeline also displays the resulting Docker images using the docker images command for verification.

Testing

- a. **Tools Used:**
 - API tests implemented in Python using unittest.
 - Docker Compose for test orchestration
- b. **Test Cases:**
 - Validation of the /api/ endpoint to ensure service1 and service2 interactions.
 - Testing /metrics endpoint for monitoring and system state management.
 - The tests ensure the functional correctness of the system in different states (INIT, RUNNING, PAUSED, SHUTDOWN).
- c. **Execution:**
 - The test stage builds and runs the test services using docker-compose up --build test.
 - Post-test, the system is cleaned up using docker-compose down.

Packing

- **Containerized Services:** All services (service1, service2, and tests) are containerized to ensure portability and consistency across environments.
- **Configuration Management:** Docker Compose orchestrates the services and their dependencies in a reproducible manner.

Deployment

- The deploy stage deploys all services using docker-compose up -d to run them in detached mode.
- Running containers are verified using docker-compose ps to confirm successful deployment. As I can't do future as this a test production.
- The deployment is automated to both the project and main branches, ensuring a continuous deployment process. At first I worked in the project branch. So for the testing in the project branch, I kept the project branch in the deployment stage, but only the main one is the appropriate one.

Operating and Monitoring

- **Monitoring:**
 - a. System metrics such as uptime, request count, and current state are exposed via the /metrics endpoint.
 - b. Logs are captured using docker-compose logs for troubleshooting and stored as artifacts.
- **Operational Control:**

A web interface allows users to trigger actions such as shutting down Docker containers via a browser, providing operational flexibility.

Example Runs of the Pipeline

Passing log given below

In build Stage:

Running with gitlab-runner 17.6.0 (374d34fd)
on MyRunner t3_DwDzf, system ID: s_67c963c10c86

Preparing the "docker" executor

Using Docker executor with image docker:latest ...

Starting service docker:dind...

Pulling docker image docker:dind ...

Using docker image sha256:37123e40ec5295ac68158ace2d5c302a8a5d33f6e40b9a3397be063f941d3705 for docker:dind with digest docker@sha256:6ca9a6811085e2cf769cbac04bc47daf66629102990391caab7cf37426e939da ...


Waiting for services to be up and running (timeout 30 seconds)...

00:46

rs ago	745MB				
registry.gitlab.com/gitlab-org/gitlab-runner/gitlab-runner-helper		x86_64-v17.6.0	ee6fb0f6ea3a	2 days	
ago	87.4MB				
<none>		<none>	bfa96909f313	3 days	
ago	148MB				
<none>		<none>	515ad8ebccbd	3 days	
ago	445MB				
docker		dind	37123e40ec52	2 mont	
hs ago	373MB				
docker		latest	37123e40ec52	2 mont	
hs ago	373MB				
Cleaning up project directory and file based variables					
Job succeeded					

00:01

Related jobs

→  build

In Test Stage:

Running with gitlab-runner 17.6.0 (374d34fd)
on MyRunner t3_DwDzf, system ID: s_67c963c10c86

Preparing the "docker" executor

Using Docker executor with image docker:latest ...

Starting service docker:dind...

Pulling docker image docker:dind ...

Using docker image sha256:37123e40ec5295ac68158ace2d5c302a8a5d33f6e40b9a3397be063f941d3705 for docker:dind with digest docker@sha256:6ca9a6811085e2cf769cbac04bc47daf66629102990391caab7cf37426e939da ...

Waiting for services to be up and running (timeout 30 seconds)...

*** WARNING: Service runner-t3dwdzf-project-5-concurrent-0-3a96a685a4001fb9-docker-0 probably didn't start properly.

Health check error:
service "runner-t3dwdzf-project-5-concurrent-0-3a96a685a4001fb9-docker-0-wait-for-service" timeout

Health check container logs:
2024-12-08T21:01:29.612297452Z waiting for TCP connection to 172.17.0.2 on [2375 2376]...
2024-12-08T21:01:29.612440528Z dialing 172.17.0.2:2376...
2024-12-08T21:01:29.612628297Z dialing 172.17.0.2:2375...
2024-12-08T21:01:30.614063595Z dialing 172.17.0.2:2375...
2024-12-08T21:01:30.614091791Z dialing 172.17.0.2:2376...
2024-12-08T21:01:31.615376871Z dialing 172.17.0.2:2376...
2024-12-08T21:01:31.615399889Z dialing 172.17.0.2:2375...

00:42

→ test

→  **deploy**

 test

Reflections

Main Learnings:

- 1. CI/CD Pipeline Implementation:** Building a CI/CD pipeline for a microservices architecture was a valuable learning experience. The pipeline automated building, testing, and deploying the system, demonstrating how to streamline the development process. I build CI/CD previously, but for microservice this is the first time. SO learned a lot.
- 2. API Testing for Both Services:** The API tests validated the functionality of both service1 and service2. For example:
 - Requests to service2 metrics and state endpoints confirmed its responsiveness and adherence to expected states.
 - Interactions routed through Nginx validated the integration between service1, service2, and the gateway itself.
- 3. Stopping Docker Containers via the Browser:** A unique learning was implementing and testing the ability to shut down Docker containers directly from the browser. This functionality was not only a technical achievement but also emphasized the importance of user-triggered operational controls in service management.
- 4. Nginx as an API Gateway:** The configuration of Nginx to route API requests and manage stateful interactions between services was a major milestone. It provided insights into handling load balancing, traffic routing, and access control.

Challenges Faced:

- 1. Debugging CI/CD Pipeline Errors:** YAML validation errors and dependency issues required iterative debugging and fine-tuning. Resolving these pipeline issues consumed a significant amount of time. Configuring this file so that it perfectly work for nginx was the main issue I faced in this project.
- 2. Service State Synchronization:** Ensuring service1 and service2 were in the correct state for testing required careful coordination. For example, service2 had to be in a RUNNING state for API interactions to work as expected.
- 3. Time Constraints for Enhancements:** Limited time prevented adding certain enhancements, such as running the system using cloud, CSC. Tried to use cPouta , which is the IaaS cloud services at CSC . While initial preparations were

made, full integration could not be completed. But this will be done in future for sure.

Key Insights:

- 1. Comprehensive API Testing:** The implemented API testing effectively covered interactions between the services. For example: Testing /api/ endpoints demonstrated the integration of service1 and service2 via the Nginx gateway. Validating /metrics endpoint provided confidence that service2 adhered to its expected behavior.
- 2. User-Triggered Docker Operations:** Implementing the ability to stop all Docker containers via the browser was a practical and user-friendly feature. This emphasized the importance of operational controls in microservices environments.
- 3. Completing all the states:** Successfully implementing and validating all defined states (INIT, RUNNING, PAUSED, SHUTDOWN) of the system was a significant achievement. The ability to seamlessly transition between these states demonstrated the robustness and reliability of the system's state management logic.

What Could Have Been Done Differently:

- 1. Prioritizing Deployment on External Platforms:** Allocating more time early in the project to integrate with an external platform, such as CSC, could have expanded the scope of testing and deployment environments.
- 2. Pipeline Development:** I tried to keep the pipeline good in way that they did not take too much time to build, I use the docker-compose in the pipeline so that it can be built perfectly. Gitlab runner could not give me to add some more things in the pipeline like notifications and some advance deployment staff. So I think pipeline development should be done more.

All things considered, the project offered insightful information on the variety of issues involved in developing, testing, and implementing a microservices architecture. It emphasized how crucial automation, well-organized processes, and efficient testing are to producing software that is dependable and maintainable. The difficulties encountered, whether in maintaining Nginx configurations, synchronizing service states, or debugging the CI/CD pipeline, turned out to be invaluable teaching moments. They emphasized the necessity of proactive problem-solving, progressive development, and careful planning.

Successfully implementing state management, ensuring service interactions through Nginx, and automating deployment pipelines all contributed to a deeper understanding of modern software practices.

Amount of Effort (Hours) Used:

- **Planning and Research:** 7 hours
- **Setting up CI/CD Pipeline:** 11 hours
- **Developing and Testing Services:** 22 hours
- **API Testing:** 5 hours
- **Frontend Enhancements (CSS and Interface):** 2 hours
- **Monitoring and Logging Implementation:** 3 hours
- **Documentation and Finalization:** 6.5 hours

Total Estimate: 56.5 hours