COMP.SE.140.Ex1

Exercise 01 for Continuous Development and Deployment - DevOps

Basic Platform Information

Hardware:

- Processor: 13th Gen Intel i7
- RAM: 16GB
- Storage: 500GB

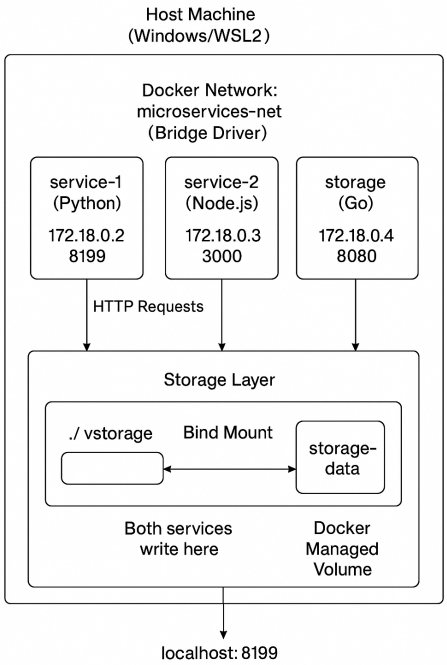
Operating System:

- OS: Windows 11 with WSL2 (Windows Subsystem for Linux)
- WSL Distribution: Ubuntu 22.04 LTS
- Architecture: x86_64

Software Versions:

- Docker: 27.2.0
- Docker Compose: v2.29.2-desktop.2

System Architecture Diagram



Analysis of Status Records

What is Measured

Uptime:

- Measures the time since the container/service started running
- Calculated from application start time, not container creation time
- Unit: hours with 2 decimal precision

Disk Space:

- Measures free space in the root filesystem (`/`) of each container
- Unit: MBytes (Megabytes) or Mbytes depending on service
- Obtained using:
  - Python (service-1): `shutil.disk_usage('/')`
  - Node.js (service-2): `df /` command executed via shell

Relevance of Measurements

Uptime:

- **Limited Relevance**: Measures application uptime, not container or host uptime
- **Container Context**: Resets every time the container restarts
- **Use Case**: Useful for debugging restart loops or understanding service stability
- **Limitation**: Doesn't reflect the actual service availability from user perspective

Disk Space:

- **Very Limited Relevance**: Measures container's filesystem layer, not meaningful storage
- **Isolated View**: Each container has its own filesystem view
- **Misleading**: Large values (e.g., 972GB) come from the host's filesystem, but containers can't actually use all of it
- **Problem**: Doesn't respect Docker resource limits or actual container constraints

What Should Be Done Better

For Uptime:

1. Track service availability from an external monitoring perspective
2. Implement health check history to track when services become unhealthy
3. Add metrics for request count and error rates instead of just time

4. Consider measuring container uptime vs application uptime separately

For Disk Space:

1. Monitor Docker volume usage specifically (where data is actually stored)
2. Track container-specific resource limits if set via `docker-compose`
3. Measure application-specific metrics:
    - Number of log entries
    - Size of stored data
    - Database/cache usage if applicable
4. Monitor host-level resources through a dedicated monitoring service
5. Use container resource constraints and measure against those limits

Better Metrics to Implement:

- **Memory usage**: Current RAM consumption vs limit
- **CPU usage**: Percentage of allocated CPU being used
- **Network I/O**: Bytes sent/received
- **Request metrics**: Request count, latency, error rate
- **Application metrics**: Queue depth, active connections, processing time

Persistent Storage Analysis

Solution 1: Bind Mount (`./vstorage`)

Implementation:

- Direct bind mount from host directory to container
- Path: `./vstorage` on host → `/app/host/vstorage` in container
- Both service-1 and service-2 write to the same file

Advantages:

1. **Direct File Access**: Can inspect/edit file directly on host machine
2. **Easy Debugging**: Use standard tools (`cat` , `tail` , `grep`) on host
3. **Simple Backup**: Just copy the file to backup location
4. **No Docker Commands Needed**: No need for `docker cp` or volume inspection
5. **Transparent**: What you see on host is exactly what's in the container
6. **Version Control Friendly**: Can easily track in git (though shouldn't for logs)
7. **Cross-Platform Development**: Works consistently across different development environments

Disadvantages:

1. **Poor Portability**: Hardcoded host paths make it environment-specific
2. **Security Risk**: Exposes host filesystem to containers
3. **Violates Isolation**: Breaks container independence principle
4. **Path Dependencies**: Requires specific host directory structure
5. **Permission Issues**: Can have file ownership conflicts between host and container users
6. **Not Production-Ready**: Considered an anti-pattern for production deployments
7. **Concurrent Write Issues**: Multiple containers writing to same file without proper locking
8. **Platform-Specific**: Path separators and permissions differ between Windows/Linux/Mac
9. **Docker Desktop Limitation**: Performance issues on Windows/Mac due to filesystem layer

When to Use:

- Local development and debugging
- Quick prototyping
- Educational purposes (like this assignment)
- When you need to frequently inspect logs manually

Solution 2: Storage Service with Docker Volume

Implementation:

- Dedicated Go microservice exposing REST API
- Docker managed volume: `storage-data`
- Only storage service has access to the volume
- Other services interact via HTTP requests

Advantages:

1. **Proper Isolation**: Services don't have direct filesystem access
2. **Docker Managed**: Volume lifecycle handled by Docker engine
3. **Portable**: Works identically across different environments
4. **Scalable Architecture**: Can be replaced with distributed storage (S3, databases)
5. **Access Control**: API can implement authentication/authorization
6. **Better Security**: Storage service can validate and sanitize inputs
7. **Microservices Best Practice**: True separation of concerns
8. **Testable**: Can mock the storage service in tests
9. **Monitoring**: Can add metrics, logging, and monitoring to storage operations
10. **Versioning**: Can implement data versioning and audit trails
11. **Future-Proof**: Easy to swap implementation (file → database → cloud storage)
12. **Transaction Support**: Can implement atomic operations and rollbacks

Disadvantages:

1. **Increased Complexity**: Additional service to maintain and monitor

2. **Network Overhead**: HTTP requests add latency vs direct file I/O
3. **Single Point of Failure**: If storage service goes down, logging fails
4. **More Moving Parts**: Need to monitor service health, restarts, etc.
5. **Resource Usage**: Additional container consumes memory and CPU
6. **Debugging Complexity**: Need to check logs of storage service separately
7. **Development Overhead**: More code to write and maintain
8. **Testing Complexity**: Need to test API endpoints and error conditions

When to Use:

- Production deployments
- When data needs to be shared across multiple services
- When you need access control or data validation
- When planning to scale or use distributed storage
- When following microservices architecture patterns

Comparison Summary

Aspect	Bind Mount	Storage Service
Complexity	Low	High
Portability	Poor	Excellent
Security	Weak	Strong
Performance	Fast (direct I/O)	Slower (network overhead)
Scalability	Not scalable	Highly scalable
Development Speed	Fast	Slower
Production Readiness	Not recommended	Recommended
Debugging	Easy	Moderate
Maintenance	Low	Higher

Teacher's Instructions for Cleaning Up

```
docker-compose down --volumes
rm -f ./vstorage
```

What Was Difficult?

- Container Networking: Understanding how Docker DNS resolution works and configuring service-to-service communication.
- Volume Mounting Semantics: Initially created `vstorage` as a directory instead of a file, causing mount issues.
- Cross-Language Implementationnge: Implementing the same functionality differently in different languages.

Main Problems and Solutions

Problem 1: Service Discovery and Communication

Problem: Services couldn't find each other initially.

Root Cause: Not using Docker network and attempting to connect via localhost.

Solution:

- Created dedicated bridge network `microservices-net`
- Used Docker service names for DNS resolution
- Configured all services on the same network

Problem 2: Volume Mount Type Confusion

Problem: `vstorage` being created as directory instead of file.

Root Cause: Mounting a non-existent path directly causes Docker to create a directory.

Solution:

- Mount the parent directory (`./:/app/host`)
- Let application create the file programmatically
- Application code handles file creation with proper path