Highlighting Docker/Docker Compose: Local Environment Orchestration

Docker and Docker Compose are the cornerstones of your local enterprise data platform, providing the essential capabilities for containerization and multi-container application orchestration. They ensure that your entire data stack – from FastAPI and Kafka to Spark and Grafana – runs in isolated, reproducible, and portable environments, closely mirroring a production cloud setup without the complexity of managing virtual machines directly.
This guide will demonstrate basic and advanced use cases of Docker and Docker Compose, leveraging your **Advanced Track** local environment setup.
Reference: This guide builds upon the concepts and setup described in **Section 4.2. Core Technology Deep Dive** of the **Core Handbook** and the **Progressive Path Setup Guide Deep-Dive Addendum**, which extensively uses Docker Compose for environment provisioning.

Basic Use Case: Starting, Stopping, and Inspecting Services

Objective: To demonstrate the fundamental commands for managing your multi-service data platform using Docker Compose, including bringing services up, bringing them down, and inspecting their status and logs.
Role in Platform: Simplify the setup and teardown of the entire complex data stack, making it easy for developers to start and stop their local environment.
Setup/Configuration (Local Environment - Advanced Track):

1. **Ensure Docker Desktop is running:** Docker Desktop (or Docker Engine on Linux) must be active for Docker Compose to function.
2. **Navigate to your project root:** All Docker Compose commands are executed from the directory containing your docker-compose.yml file.
   ```
   cd /path/to/your/data-ingestion-platform
   ```

Steps to Exercise:

1. Bring up all services in detached mode:
   This command reads your docker-compose.yml and starts all defined services in the background. The --build flag ensures Docker images are rebuilt if there are changes to your Dockerfiles, and -d runs them in detached mode.
   ```
   docker compose up --build -d
   ```
   Observe: You will see messages indicating each service being created/started.
2. Check the status of running services:
   This command lists all services defined in your docker-compose.yml and shows their

current state (running, exited, unhealthy) and exposed ports.
docker compose ps

Observe: All services should show running or healthy (if healthchecks are configured and passed).

3. View logs for all services:
   This command streams the logs from all running containers, aggregated in a single output. It's invaluable for initial troubleshooting.
   docker compose logs -f

   Observe: You'll see startup logs, application output, and any errors from all containers. Use Ctrl+C to exit. To see logs for a specific service: docker compose logs -f <service_name> (e.g., docker compose logs -f spark).
4. Stop all services:
   This command gracefully stops all running containers defined in your docker-compose.yml.
   docker compose stop

   Observe: Services will transition from running to exited. docker compose ps will confirm they are stopped.
5. Stop and remove all services and their networks/volumes:
   This command stops all services, removes their containers, and by adding the -v flag, also removes any anonymous volumes created by Docker Compose. This is useful for a clean slate, but be cautious as it will remove persistent data if volumes are not explicitly named or host-mounted.
   docker compose down -v

   Observe: Confirmation that containers, networks, and volumes are removed.

Verification:

- docker compose ps shows all services in the desired state (running/healthy or exited).
- docker compose logs -f displays expected startup messages and no critical errors.
- The environment can be reliably started and stopped, demonstrating fundamental control over the platform's lifecycle.

Advanced Use Case 1: Managing Service Dependencies and Health Checks

Objective: To demonstrate how depends_on and healthcheck configurations in docker-compose.yml ensure that services start in the correct order and are truly ready before dependent services try to connect, enhancing local environment stability.
Role in Platform: Prevent common startup failures (e.g., Spark trying to connect to Kafka before Kafka is ready, Airflow trying to connect to PostgreSQL before it's initialized), leading to a more robust and predictable development environment.

Setup/Configuration:

1. **Review docker-compose.yml dependencies:** Look at services like kafka and spark, airflow-webserver and postgres.
    - kafka typically depends_on: zookeeper.
    - spark often depends_on: kafka and minio.
    - airflow-webserver depends_on: postgres and airflow-scheduler.
    - Crucially, these depends_on clauses should use condition: service_healthy.

Example docker-compose.yml snippet illustrating healthchecks and dependencies:

```
version: '3.8'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.4.0
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    healthcheck: # Healthcheck for Zookeeper
      test: ["CMD", "sh", "-c", "nc -z localhost 2181 || exit 1"]
      interval: 5s
      timeout: 3s
      retries: 5
      start_period: 10s # Give it time to start before checking

  kafka:
    image: confluentinc/cp-kafka:7.4.0
    depends_on:
      zookeeper:
        condition: service_healthy # Kafka waits for Zookeeper to be healthy
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    healthcheck: # Healthcheck for Kafka
      test: ["CMD", "sh", "-c", "kafka-topics --bootstrap-server localhost:9092 --list || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 5
      start_period: 20s

  spark:
    image: bitnami/spark:3.5.0
    depends_on:
      kafka:
        condition: service_healthy # Spark waits for Kafka to be healthy
```

```
    minio:
      condition: service_healthy # Spark waits for MinIO to be healthy
  # ... other Spark configurations
```

Steps to Exercise:

1. **Bring down services (if running):** docker compose down -v to ensure a fresh start.
2. **Bring up services again:** docker compose up --build -d
3. **Observe startup order and health:**
   - Use docker compose ps repeatedly. You'll notice services like zookeeper and minio becoming (healthy) first.
   - Then, kafka will become (healthy) *after* zookeeper is healthy.
   - Finally, spark will start and become (healthy) *after* both kafka and minio are healthy.
   - Observe docker compose logs -f for specific messages indicating healthcheck probes passing or services waiting for dependencies.
4. **Simulate a dependency failure (optional, for advanced testing):**
   - While all services are running, manually stop zookeeper: docker compose stop zookeeper.
   - Observe Kafka's state and logs. It will likely become unhealthy or exited because its dependency is gone.
   - Restart zookeeper: docker compose start zookeeper.
   - Observe kafka recovering its healthy state.

Verification:

- docker compose ps consistently reports (healthy) for all services once dependencies are met.
- Logs clearly show services waiting for service_healthy conditions before starting their main processes, demonstrating proper orchestration of startup order.

Advanced Use Case 2: Volume Management & Data Persistence

Objective: To demonstrate how Docker volumes are used to persist data generated by stateful services (databases, Kafka logs, Delta Lake files) across container restarts and even docker compose down operations.
Role in Platform: Ensure that your valuable data (e.g., PostgreSQL data, Kafka messages, Delta Lake snapshots) is not lost when containers are stopped, removed, or updated, providing a production-like persistence layer for local development.
Setup/Configuration:

1. **Review docker-compose.yml for volumes:** Identify named volumes and host-mounted bind mounts.
   - **Named volumes:** (e.g., postgres_data, minio_data) are managed by Docker and typically live in /var/lib/docker/volumes/ on your host. They are automatically created by Docker Compose if they don't exist and persist across docker

compose down unless -v is explicitly used.
    - **Bind mounts:** (e.g., ./data/postgres:/var/lib/postgresql/data) map a host directory directly into the container. Data persists in the host directory.

Example docker-compose.yml snippet illustrating volume types:# ...

```
services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: main_db
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
    volumes:
      - postgres_data:/var/lib/postgresql/data # Named volume for PostgreSQL data
    # ...

  minio:
    image: minio/minio:latest
    environment:
      MINIO_ROOT_USER: minioadmin
      MINIO_ROOT_PASSWORD: minioadmin
    volumes:
      - ./data/minio:/data # Bind mount for MinIO data (S3 bucket content)
    # ...

  kafka:
    image: confluentinc/cp-kafka:7.4.0
    environment:
      KAFKA_LOG_DIRS: /kafka/kafka-logs # Internal path for Kafka logs
    volumes:
      - kafka_data:/kafka/kafka-logs # Named volume for Kafka message logs
    # ...

  spark:
    # ...
    volumes:
      - ./pyspark_jobs:/opt/bitnami/spark/jobs # Bind mount for Spark job scripts
      - ./data/spark-events:/opt/bitnami/spark/events # Bind mount for Spark History
Server logs
    # ...
# Define named volumes at the bottom of the file
volumes:
  postgres_data:
  kafka_data:
```

2. **Ensure data/ subdirectories exist on your host** for bind mounts (./data/minio, ./data/spark-events).

Steps to Exercise:

1. **Start Services with Volumes:** docker compose up --build -d
2. **Generate Data:**
    - Run python3 simulate_data.py for a few minutes to ensure data is ingested into FastAPI, then Kafka, then processed by Spark to Delta Lake in MinIO, and finally persisted in PostgreSQL.
    - Verify data exists in PostgreSQL (e.g., docker exec -it postgres psql -U user -d main_db -c "SELECT COUNT(*) FROM financial_transactions;").
    - Verify Delta Lake files exist in MinIO (check http://localhost:9001).
3. **Stop and Remove Containers (keeping volumes):**
   docker compose stop # Stops containers
   docker compose rm -s -v # Removes stopped containers and their anonymous volumes, but NOT named volumes or bind mounts

   Note: To prove named volumes persist, you must NOT use docker compose down -v. Just docker compose down will preserve named volumes. For bind mounts, the data is directly on your host, so it always persists unless you manually delete the host directory.
4. **Verify Volume Persistence (for named volumes and bind mounts):**
    - **PostgreSQL:**
        - Run docker compose up -d postgres to bring just the PostgreSQL container back up.
        - Connect to PostgreSQL and query: docker exec -it postgres psql -U user -d main_db -c "SELECT COUNT(*) FROM financial_transactions;".
        - **Expected:** The count should be the same as before, demonstrating data persistence.
    - **MinIO:**
        - Access http://localhost:9001. The previously ingested Delta Lake files should still be visible in raw-data-bucket.
    - **Kafka:**
        - Run docker compose up -d kafka zookeeper.
        - Connect a Kafka consumer: docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:29092 --topic raw_financial_transactions --from-beginning.
        - **Expected:** You should see old messages from before the stop, demonstrating Kafka log persistence.
5. **Clean up (optional):** To remove named volumes and start completely fresh, use: docker volume rm <volume_name> (e.g., docker volume rm data-ingestion-platform_postgres_data) or docker compose down -v if you're sure you want to delete all persistent data.

Verification:

- Data stored in PostgreSQL, Kafka, and MinIO remains intact and accessible after stopping and restarting their respective containers, proving the effectiveness of volume management for data persistence.

Advanced Use Case 3: Network Isolation & Inter-Container Communication

Objective: To demonstrate how Docker Compose creates a private, isolated network for all your services, enabling seamless and secure communication between them using service names as hostnames, while also showing how to expose services to your host machine.
Role in Platform: Mimic a cloud-native private network, ensuring services can discover and communicate with each other securely without exposing all ports directly to the host's public network.
Setup/Configuration:

1. **Review docker-compose.yml networking:**
   - By default, Docker Compose creates a single bridge network for all services.
   - Services can communicate with each other using their service names (e.g., fastapi_ingestor can connect to kafka:29092).
   - ports mapping ("HOST_PORT:CONTAINER_PORT") makes a container's port accessible from the host.

Example docker-compose.yml snippet illustrating networking:# ...

```
services:
  fastapi_ingestor:
    # ...
    environment:
      KAFKA_BROKER: kafka:29092 # Refers to 'kafka' service name within the Docker network
    ports:
      - "8000:8000" # Exposes FastAPI to localhost:8000 on the host

  kafka:
    # ...
    environment:
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
      # PLAINTEXT://kafka:29092 is for inter-container communication
      # PLAINTEXT_HOST://localhost:9092 is for host-to-container communication (e.g., console
consumers from host)
    ports:
      - "9092:9092" # Exposes Kafka to localhost:9092 on the host (for external clients)

  spark:
    # ...
    environment:
      KAFKA_BROKER: kafka:29092 # Spark connects to Kafka using its service name
      MINIO_HOST: minio:9000 # Spark connects to MinIO using its service name and port
    # No ports exposed by Spark itself for general use in this setup, only for Spark UI
```

Steps to Exercise:

1. **Start all services:** docker compose up --build -d
2. **Verify Inter-Container Communication (FastAPI to Kafka):**
   - Run python3 simulate_data.py.
   - Open docker compose logs fastapi_ingestor. You should see logs confirming successful message publishing to kafka:29092, demonstrating internal communication.
3. **Verify Host-to-Container Communication (Accessing FastAPI/Kafka from Host):**
   - Access FastAPI health check from your host browser/curl: http://localhost:8000/health.
   - Run a Kafka console consumer directly from your host machine (if Kafka CLI is installed, otherwise use docker exec -it kafka ... which implicitly uses the host network to connect to Kafka's exposed port):
     # If Kafka CLI is installed on host
     kafka-console-consumer --bootstrap-server localhost:9092 --topic raw_financial_transactions --from-beginning
   - These actions confirm that services with exposed ports are accessible from your host machine.
4. **Verify Network Isolation (Conceptual):**
   - Try to directly access an internal-only port of a container from your host that is *not* mapped in ports (e.g., PostgreSQL's internal port 5432 if not mapped). This attempt should fail with a connection refused error, demonstrating isolation.
   - Inside any container (e.g., docker exec -it fastapi_ingestor bash), try ping kafka or curl http://minio:9000. These commands should succeed, confirming that service names resolve within the Docker network.

Verification:

- FastAPI successfully publishes to Kafka using the Kafka service name within the Docker network.
- You can access FastAPI and Kafka (via exposed ports) from your host machine's browser/terminal.
- Attempting to access non-exposed internal ports from the host fails, while internal container-to-container communication by service name succeeds, demonstrating the effective network isolation and routing provided by Docker Compose.

This concludes the guide for Docker and Docker Compose.