Activity 1: Monolith to Microservices Handheld Demo

Objective

In Activity 0, we saw the limitations of the Monolith. In this activity, we will execute a **"Strangler Fig" migration**. Like a vine that grows around a tree and eventually replaces it, we will spin up new microservices alongside the legacy Monolith, gradually routing traffic to them.

By the end of this activity you will have:

- **Cleaned up** previous environments to avoid port conflicts.
- Understood the **Theory of Decomposition** (Coupling & DDD).
- Implemented the **Strangler Fig Pattern** to run Legacy (v1) and Modern (v2) systems in parallel.
- Deployed an **Aggregate (Anti-Corruption) Service** to orchestrate transactions.

Prerequisites

- Completion of Activity 0.
- Source code available in `/home/labDirectory`.
- Docker and Docker Compose installed.

Theoretical Context: How to Split a Monolith?

Before we run the code, we must understand *how* we decided to split the Monolith. We used **Domain-Driven Design (DDD)** and analyzed **Coupling**.

1. The Strategy: Domain-Driven Design (DDD)

To ensure our microservices are independent, we followed these DDD principles:

- **Ubiquitous Language:** We established a common vocabulary. "Inventory" means *physical stock*, while "Product" means *catalog description*. These are separate concepts.

- **Bounded Context:** We drew a boundary around these concepts. The `Inventory Service` only cares about *counts* (Integers), while the `Product Service` cares about *names and descriptions* (Strings).
- **Aggregates:** We identified that an "Order" is an **Aggregate**—it doesn't exist in isolation; it binds together a User, Products, and a Payment.

2. The Danger: Four Levels of Coupling

We avoided bad coupling to ensure our services don't crash each other:

1. **Content Coupling (Highest/Worst):** One service directly modifies the private data of another (e.g., The Monolith directly updating the Inventory Table). *We removed this by giving Inventory its own Database.*
2. **Common Coupling:** Multiple services sharing the same Global Data or Shared Database. *We broke this by splitting the schema.*
3. **Control/Pass-through Coupling:** One service telling another *how* to do its job, or passing data it doesn't need.
4. **Domain Coupling (Lowest/Best):** Services connect only through explicit, clean APIs (REST).

3. The Patterns

- **Strangler Fig Pattern:** We don't rewrite the whole Monolith at once. We build the new System (v2) around the edges of the Old System (v1). As v2 grows, v1 shrinks, until the Monolith is "strangled" and removed.
- **Anti-Corruption Layer (The Aggregate):** The **Order Service** acts as a buffer. It takes clean requests from the user and handles the messy coordination with Inventory, Payment, and Product services. It prevents the complexity of those subsystems from "corrupting" the client experience.

Step-by-Step Guidance

Task 1: Environment Cleanup (Critical)

Goal: Ensure no ports (especially 30000) are occupied by the previous activity.

Step 1: Stop and Remove Old Containers Run this command to forcefully stop and remove any running containers from Activity 0.

```Shell
docker rm -f $(docker ps -aq)
docker container prune -f
```

Note: If you receive an error saying "requires at least 1 argument", it means you have no containers running. That is fine.

Step 2: Verify Ports are Free Ensure nothing is running on ports 30000-30005.

```Shell
docker ps
```

Output should be empty.

Task 2: The Database-per-Service Pattern (Polygot-Persistency)

Goal: Extract the Inventory module and give it its own dedicated database to break **Common Coupling**.

Step 1: Create the Dedicated Network Microservices need a private network to talk to each other.

```Shell
docker network create shop-network
```

Step 2: Start the Inventory Database (PostgreSQL) Unlike the Monolith's in-memory DB, this data survives restarts.

```Shell
docker run -d --name inventory-db \
  --network shop-network \
  -e POSTGRES_DB=inventorydb \
  -e POSTGRES_USER=postgres \
  -e POSTGRES_PASSWORD=postgres \
  postgres:15
```

Step 3: Build and Run the Inventory Service (Java) Navigate to the inventory directory and build the service.

Shell
```
cd /home/labDirectory/microservices/eCommerce-InventoryService

docker build -t inventory-service:v2 .

docker run -d --name inventory-app \
  --network shop-network \
  -p 30001:30001 \
  -e SPRING_DATASOURCE_URL=jdbc:postgresql://inventory-db:5432/inventorydb \
  inventory-service:v2
```

Task 3: The Polyglot Programming Pattern (Product Catalog)

Goal: Deploy a service in a different language (Python) to demonstrate **Bounded Contexts**. The Product domain uses different tools than the Inventory domain.

Step 1: Build the Product Service This service uses FastAPI and a lightweight SQLite database.

Shell
```
cd /home/labDirectory/microservices/eCommerce-ProductService
docker build -t product-service:v2 .
```

Step 2: Run the Product Service

```
Shell
docker run -d --name product-app \
  --network shop-network \
  -p 30002:30002 \
  -e DATABASE_URL="sqlite:///./products.db" \
  product-service:v2
```

Task 4: The Aggregate/Anti-Corruption Pattern (Order Service)

Goal: Deploy the **Order Service**. This is our **Anti-Corruption Layer**. It aggregates data from Product, Inventory, and Payment so the client doesn't have to deal with them individually.

Step 1: Automated Deployment via Compose We will now spin up the rest of the ecosystem, including the **Monolith (Legacy)**, **Payment Service**, and **Order Service**.

```
Shell
docker ps
# Removes the existing containers
docker rm -f $(docker ps -aq)

cd /home/labDirectory
# Reruns all the containers step by step
docker compose up --build -d

docker ps
```

Step 2: Verify the Ecosystem Run `docker ps`. You should see 6 containers running:

1. `monolith-app` (Legacy) on :30000
2. `inventory-app` on :30001
3. `product-app` on :30002
4. `order-app` on :30003
5. `payment-app` on :30004
6. `inventory-db` (Postgres)

Task 5: Validating the "Strangler Fig"

Goal: Prove that we are using the new system for sales, but the old system for user management.

Step 1: The "New" World (Buy an Item) We will use the **Order Service (v2)** on Port 30003. This triggers the distributed transaction across our aggregates.

```shell
# Add to Cart (New Service)
curl -X POST http://localhost:30003/api/v2/orders/cart/1/add \
  -H "Content-Type: application/json" \
  -d '{"productId": 1, "quantity": 1}'

# Checkout (Orchestration)
curl -X POST http://localhost:30003/api/v2/orders/checkout/1
```

Step 2: The "Old" World (Check User Profile) We haven't migrated User Profiles yet. We must still ask the Monolith on Port 30000.

```shell
curl -s http://localhost:30000/api/users
```

What this does & why:

- This is the **Strangler Fig** in action.
- **Route A (New):** `/api/v2/orders` Order Microservice.
- **Route B (Old):** `/api/users` Monolith.
- Over time, we will move "Users" to a microservice and shut down Route B forever.

Access the Web UI Open your browser and navigate to the following URL. You should see the E-Commerce store dashboard.

- **URL:** http://localhost:30000

Task 6: Recording Validation Data

Goal: Verify the data integrity across the distributed system.

Step 1: Check Inventory Stock After buying 1 Smartphone, check the **Inventory Service** (Port 30001). It should be 49.

```Shell
curl -s http://localhost:30001/api/v2/inventory/1
```

Useful commands for this task

Command	**Purpose**	**Example**
`docker rm -f $(docker ps -aq)`	**Nuke**: Remove ALL containers	Use with caution!
`docker network create`	Make a private network	`docker network create shop-net`
`docker compose up -d`	Start full stack in background	`docker-compose up -d`

Cleanup (Do not skip)

To prepare for the next activity (Exercise), we need to clear this environment.

```Shell
docker compose down
docker rm -f inventory-app inventory-db product-app
docker system prune
docker network rm shop-network
```

Congratulations! You have successfully migrated the core functionality using **DDD** and **Strangler Fig** patterns. You now understand how to decouple systems. In **Activity 2**, you will have to extract a service on your own.