# Activity 0: Understanding the Monolith

## Objective

The goal of this activity is to establish a baseline by exploring a legacy **Monolithic E-Commerce Application**. You will build the application from the source code provided in your lab environment, run it, and inspect its internal architecture.

By the end of this activity you will have:

- A running instance of the "E-Commerce Monolith" on port 30000.
- Understood the **Scaling Bottleneck** of monolithic architectures.
- Identified why "Independent Scaling" is a primary driver for Microservices.
- Prepared the environment for the **Strangler Fig** migration pattern in future activities.

## Prerequisites

- Docker installed and running.
- Source code available in `~/labdirectory/eCommerce-Monolith`.
- Access to the `ans.json` file in your lab directory for recording results.

## Theoretical Context: The Scaling Problem

Before we migrate, we must understand the critical flaw we are solving: **The Scaling Bottleneck**.

In a **Monolith**, the User, Product, Cart, and Order modules all share the same resources (CPU, Memory, Database Connection Pool).

- **The Scenario:** Imagine it is "Black Friday." Millions of users are browsing (high traffic on `Product Service`), but only a few are buying.
- **The Monolith Problem:** To handle the browsing traffic, you must scale (replicate) the **entire application**. You end up wasting memory scaling the `Order` and `Payment` modules, even though they aren't under load.

- **The Microservices Solution:** You can scale the `Product Service` to 50 instances while keeping the `Order Service` at 2 instances. This is called **Independent Scaling**.

---

## Step-by-Step Guidance

### Task 1: Explore and Build the Monolith

**Goal:** Locate the source code, understand the structure, and package it into a Docker image.

**Step 1: Navigate to the Codebase** The source code is pre-loaded in your lab environment. Navigate to the project folder.

Shell
```
cd /home/labDirectory/eCommerce-Monolith
ls -F
```

*You should see files like `pom.xml`, `Dockerfile`, and a `README.md` file. We encourage you to read the `README.md` to understand the project structure.*

**Step 2: Build the Docker Image** We will package the Java Spring Boot application into a Docker image named `monolith-shop`.

Shell
```
docker build --no-cache -t monolith-shop .
```

**Step 3: Run the Application** We will run the container, mapping the internal port 30000 to your host's port 30000.

Shell
```
docker run -d -p 30000:30000 --name monolith-app monolith-shop
```

**What this does & why:**

- `cd` : Moves you into the context of the application code.
- `docker build`: Compiles the code and packages the Java Runtime Environment (JRE) required to run it.
- `-p 30000:30000`: Explicitly maps the port so you can access the Web UI.

---

## Task 2: Interact with the Monolith (Web UI & API)

**Goal:** Validate the application is working and experience the application from a user's perspective.

**Step 1: 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

**Step 2: Check Product Data via API** Use `curl` to fetch the product list. Note that the price is returned in "Rs."

Shell
```
curl -s http://localhost:30000/api/products | grep "Smartphone"
```

**Step 3: Simulate a User Transaction** We will act as the user "Little Buddha" (ID 1). In this monolith, adding to a cart and checking out happens in a single memory space.

Shell
```
# Add Product ID 1 (Smartphone) to Cart
curl -X POST http://localhost:30000/api/cart/1/add \
  -H "Content-Type: application/json" \
  -d '{"productId": 1, "quantity": 1}'

# Checkout (Triggers Payment & Stock Reduction)
curl -X POST http://localhost:30000/api/checkout/1
```

**What this does & why:**

- The **Web UI** is a Single Page Application (SPA) served directly by the Monolith backend. In a microservices architecture, this frontend might be hosted separately (e.g., on S3 or Nginx).
- The API calls demonstrate the "Happy Path" where all modules are up and running.

---

## Task 3: Inspecting the "Shared Database" Trap

**Goal:** Understand the **Shared Database** anti-pattern. This is the hardest part to decouple.

**Step 1: Access the Database Console** The app uses an H2 in-memory database.

1. Open your browser to: `http://localhost:30000/h2-console`
2. **JDBC URL:** `jdbc:h2:mem:shopdb`
3. **User:** `sa`
4. **Password:** *(leave empty)*
5. Click **Connect**.

**Step 2: Run a Cross-Domain Query** In the SQL window, run the following query. This query joins `USERS` (User Domain) with `ORDER_TABLE` (Order Domain).

**SQL Command (in Browser)**

```sql
SELECT u.NAME, o.TOTAL_AMOUNT
FROM USERS u
JOIN ORDERS o ON u.ID = o.USER_ID;
```

**What this does & why:**

- This works because both tables exist in `shopdb`.

- **The Bottleneck:** If `ORDER_TABLE` locks up due to high writes (many people buying), it can slow down `USERS` queries (people trying to log in). In Microservices, we split these databases to prevent this "Blast Radius."

---

## Task 4: Recording Validation Data

**Goal:** Verify your understanding and environment setup for the grading script.

**Step 1: Retrieve Product Price** Fetch the price of the "Mechanical Keyboard" (ID 5) for your answer file.

```
Shell
curl -s http://localhost:30000/api/products
```

**Step 2: Retrieve User Details**

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

Step 3: Update `ans.json` Open the `ans.json` file in your lab directory and fill in the following fields based on the outputs above:

- `monolith_port`: The port the app is running on (e.g., 30000).
- `keyboard_price`: The numeric price of the Mechanical Keyboard.
- `default_user_name`: The name of User ID 1.

---

## Useful commands for this task

| Command | Purpose | Example |
|---|---|---|
| `cd` | Change directory | `cd ~/labdirectory/eCommerce-Monolith` |
| `docker build` | Creates the image from Dockerfile | `docker build -t shop .` |
| `docker run -p` | Maps host port to container port | `docker run -p 30000:30000 ...` |
| `curl -X POST` | Sends data to the API | `curl -X POST .../api/cart/add` |

---

## Cleanup (Do not skip)

If you are proceeding immediately to Activity 1, you may keep the monolith running (we will need it). If you are taking a break, stop the container to save resources.

```shell
docker stop monolith-app
docker rm monolith-app
```

**Congratulations!** You have analyzed the Monolith. You now understand the baseline architecture and why scaling it is difficult. In **Activity 1**, we will begin the **Strangler Fig** migration.