1. The 5 Microservices

product-service (Port 8081)

- Job: Manages the product catalog (name, description, price).
- Code: A Spring Boot project with a Product entity, like your Student sample.

inventory-service (Port 8082)

- Job: Tracks stock levels for each product.
- Code: Has its own Inventory entity (tracking productId and quantity).

customer-service (Port 8083)

- Job: Manages customer information, registration, login, and addresses.
- Code: Will have a Customer entity.

order-service (Port 8084)

- Job: Coordinates checkout. When a user places an order, this service gets the request.
- Code: It talks to other services ("inter-service communication"). It asks the product-service for prices and the inventory-service for stock, then creates an Order in its own database.

payment-service (Port 8085)

- Job: Handles payments. It takes the final order total, processes the payment, and tells the order-service if it was successful.
- Code: Will have a Payment entity.

2. The 5 Databases (Separate Clipboards)

No problem, bro. Here's a final list of all 5 databases, their tables, and some example rows to show exactly what data they will hold.

supermarket_products_db

- Service: product-service
- Table: product

id (PK)	name	description	price
1	"Milk"	"1L Full Cream Milk"	1.50
2	"Bread"	"Fresh White Loaf"	1.00

supermarket_inventory_db

- Service: inventory-service
- Table: inventory

id (PK)	product_id	quantity
1	1	100
2	2	50

supermarket_customers_db

- Service: customer-service
- Table: customer

id (PK)	name	email	password	address
1	"Saman"	"saman@gmail.com"	"[hashed_password]"	"123 Galle Rd, Colombo"
2	"Nimala"	"nimala@gmail.com"	"[hashed_password]"	"45 Kandy Rd, Kandy"

supermarket_orders_db

- Service: order-service
- Table 1: orders

id (PK)	customer_id	order_date	total_amount
101	1	"2025-11-09 10:30:00"	4.00

- Table 2: order_items

id (PK)	order_id	product_id	quantity	price
1	101	1	2	1.50
2	101	2	1	1.00

supermarket_payments_db

- Service: payment-service
- Table: payment

id (PK)	order_id	amount	payment_status	transaction_id
1	101	4.00	"COMPLETED"	"txn_abc123"

Here's the simple flow:

1. A customer places an order.
    - (orders.customer_id -> customer.id)
2. That order is made up of multiple order_items.
    - (order_items.order_id -> orders.id)
3. Each order_item points to a product.
    - (order_items.product_id -> product.id)
4. The order has a single payment.
    - (payment.order_id -> orders.id)
5. *Separately*, the inventory table tracks the stock for each product.
    - (inventory.product_id -> product.id)

3. The Architecture "Glue" (The Advanced Part)
Here's the shorter version of the architecture "glue":

API Gateway (Port 8080)
This is the **single entry point** for our React frontend . The frontend only sends requests to the gateway on port 8080. The gateway then **routes** those requests to the correct internal microservice (e.g., /products goes to the product-service on port 8081).

Service Registry (Port 8761)
This solves the problem of "how to know the location of a service" . Each microservice **registers** itself with the registry when it starts. When another service needs to talk to it, it **discovers** its location (like http://localhost:8081) by asking the registry for its name (like product-service). This means we don't have to hard-code port numbers.

Why not MVC (Model-View-Controller)? We **are** using MVC, just not for the *overall system*. MVC is the pattern *inside* each of our Spring Boot microservices.
Controller: ProductController.java
Model: ProductService.java and ProductRepository.java
View: The JSON data we send to the frontend. The **Microservice** pattern is our choice for the *high-level system architecture* (how the big pieces connect). MVC is the low-level pattern we use to build each piece.

Why not Pipe-and-Filter? The Pipe-and-Filter pattern is for systems that process data in a single, one-way stream (like an assembly line). Our supermarket is not a one-way stream. It's a complex system where a user needs to browse, log in, and check out, all at different times. The **Microservice** pattern is a much better fit because it's designed to break a complex application into separate, independent business capabilities (Products, Customers, Orders).

4. The React Frontend (The "Dining Room")
This is the client. It's a separate project in its own folder, as your coursework allows.

- What it is: The "Dining Room" and "Menu" that the user sees. It's just JavaScript, HTML, and CSS.
- How it communicates: It will run on its own port (e.g., 3000). It will *only* make API calls to our "Waiter" (the api-gateway on port 8080).
- Example Flow:
    1. User opens http://localhost:3000 (React App).
    2. User clicks "View Products."
    3. React (the "customer") calls the "Waiter" (api-gateway at http://localhost:8080/api/products).
    4. The "Waiter" (api-gateway) walks to the "Pantry Chef" (product-service at http://localhost:8081/products) and gets the list of products.
    5. The "Waiter" hands the list back to the React app, which displays it to the user.