1st Part:-------------------------------------------------------------------------

Spring Cloud LoadBalancer for dynamic load balancing.

Resilience4j for circuit breaker.

Spring Cloud Consul for service discovery and registration.

Java Future for asynchronous communication within the Customer service.

2.

**Customer Service** calling **Order Service** and **Product Service** asynchronously.

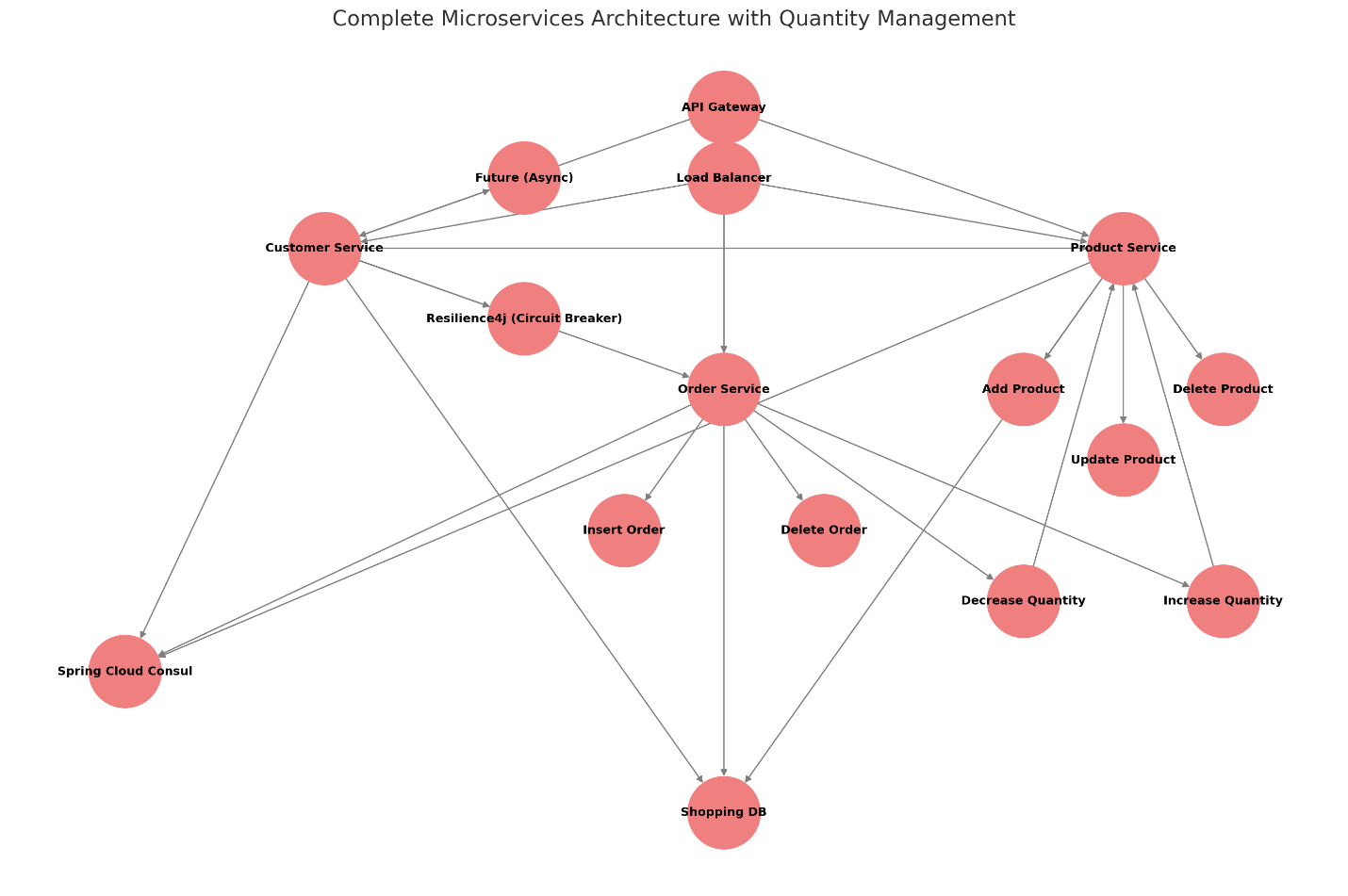
Using **Future (Async)** for parallel calls.

Data aggregation happening in **Customer Service**.

3.

* ✅ **Customer Service** interacting with **Order Service** and **Product Service** asynchronously.
* ✅ **Order Service** managing order creation, deletion, and quantity sync.
* ✅ **Product Service** managing product addition, update, deletion, and quantity adjustment.
* ✅ **Product quantity** automatically decreases/increases based on order status.
* ✅ **Service Discovery, Load Balancer, and Circuit Breaker** still in place.

I'll generate the diagram now. Please hold on. 👍🎨



Here is the **final complete architecture diagram**:

* ✅ **Customer Service** interacts with **Order Service** and **Product Service** asynchronously.
* ✅ **Order Service** supports inserting/deleting orders and triggers **Product Quantity Decrease/Increase**.
* ✅ **Product Service** handles add, update, delete, and adjusts quantity based on order status.
* ✅ **Spring Cloud Consul, Load Balancer, and Circuit Breaker** ensure high availability.
* ✅ Product quantity decreases when an order is placed and increases when an order is canceled.

2ND PART: -----------------------------------------------------------------------

Q. if customer cancel oreder what will happen new row will be created in oreder ?

No, a **new row** will **NOT** be created in the **Order Table** when a customer cancels an order.

Instead:  
✅ The **same row** will be **updated**, and the **status** will change from 'PLACED' to 'CANCELLED'.  
✅ The trigger trg\_increase\_quantity will increase the **Product quantity** based on the **cancelled order quantity**.

For example:  
If Order ID: 1 (2 Laptops) is cancelled:

* The **status** in the **Order Table** will change to **CANCELLED**.
* The **Product Table** will automatically increase the **Laptop's quantity** by **2**.

Would you like me to create the **Order Service API** to handle cancellation now? 🚀

**You said:**

Q. if again customer order same

**ChatGPT said:**

If the customer **places a new order** for the **same product** after **cancelling**, a **new row** will be created in the **Order Table** with:  
✅ A **new order\_id**.  
✅ **status** will be **PLACED**.  
✅ Product quantity will again **decrease** as per the order.

For example:

* **Order 1:** Customer ordered **2 Laptops** → **quantity reduced by 2**.
* **Order 1 Cancelled:** **quantity increased by 2**.
* **Order 2 (New):** Customer ordered **3 Laptops** → **quantity reduced by 3**.

👉 So **every new order**, whether for the same product or different, will always create a **new row** in the **Order Table**.

Would you like me to create the **Order Service API** now to handle:

1. ✅ **Place Order**.
2. ✅ **Cancel Order**.
3. ✅ **View All Orders**. 🚀

3rd Part: -------------------------------------------------------------------------------------------------

You want to implement **Product Quantity Management** **manually in Java** instead of using **SQL Triggers** — which is a more practical and flexible approach in microservices architecture. 🚀

**💡 What Exactly Will Happen?**

We will modify the **Order Service** and **Product Service** to handle **Product Quantity Management** like this:

**✅ 1. When Customer Places an Order:**

* The **Order Service** will call the **Product Service** (via **RestTemplate** or **FeignClient**) to **decrease the product quantity**.
* If the **Product Quantity** is insufficient, the order will be rejected.

**✅ 2. When Customer Cancels an Order:**

* The **Order Service** will call the **Product Service** to **increase the product quantity**.
* This will only happen if the **Order Status** changes to **CANCELLED**.

**✅ 3. If Product Quantity is Zero:**

* The **Product Service** will reject any order where the **quantity is insufficient**.
* This will prevent negative stock.

**💻 How Will the Flow Work in Java?**

✅ We'll use **RestTemplate** or **FeignClient** in **Order Service** to communicate with **Product Service**.  
✅ We'll have endpoints like:

* /decreaseQuantity/{productId}/{quantity} → To reduce quantity.
* /increaseQuantity/{productId}/{quantity} → To increase quantity.

**💯 Why Is This Better Than SQL Triggers?**

| **Approach** | **SQL Triggers** | **Java-Based (Manual)** |
| --- | --- | --- |
| **Flexibility** | Difficult to modify. | Easy to modify in code. |
| **Microservices** | Tight-coupled. | Loose-coupled. |
| **Database Load** | Higher due to triggers. | Lower as logic is in code. |
| **Scalability** | Hard to scale. | Highly scalable. |
| **Error Handling** | No control in triggers. | Full control in Java. |

**💻 What Should I Do Now?**

✅ **Option 1:** I can now modify the **Order Service** to call **Product Service** using **RestTemplate** or **FeignClient**.  
✅ **Option 2:** I can modify the **Product Service** to handle **increase/decrease quantity** based on order status.

👉 **Which one should I start with?** 🚀

1. Modify **Order Service** (Recommended ✅).
2. Modify **Product Service**.
3. Connect both with **API Gateway**.