Team Exercise: Mini Order API with Vert.x (No Real Database)

Objective

The goal of this exercise is to help the team get hands-on experience with **Vert.x**, focusing on:

- Reactive request handling
- Routing with `vertx-web`
- Asynchronous flows using `Future`
- Proper error handling
- Validation
- Testing (with mocked dependencies)
- (Optional) Event Bus usage

⚠️ Important: No real database is allowed. The repository layer must be mocked or implemented as in-memory storage.

Functional Requirements

POST /orders

Create a new order.

Request body:

```
{
  "customerId": "c-123",
  "items": [
    { "sku": "A1", "qty": 2 },
    { "sku": "B7", "qty": 1 }
  ]
}
```

Validation Rules

- `customerId` is required
- `items` must contain at least one element
- each `qty` must be greater than 0
- `sku` must not be empty

Response:

```
{
```

"id": "generated-uuid",

"status": "CREATED",

"total": 123.45

}

GET /orders/:id

Returns the order by ID.

- 200 if found
- 404 if not found

POST /orders/:id/confirm

Changes order status to `CONFIRMED`.

Business Rules

- Only allowed transition: `CREATED → CONFIRMED`
- If already confirmed → return 409 Conflict
- If order does not exist → return 404

GET /orders

Returns a list of orders.

Supported Query Parameters

- `status=CREATED|CONFIRMED`
- `customerId=...`
- `limit` (default 20, max 100)

Advance requirements

- Use Event Bus:
  - Publish `order.confirmed` event
- Create a consumer that simulates sending email (log + delay)

- Implement simple API key authentication (`X-Api-Key`)
- Implement simple rate limiting

Order Processing Simulation

Add a new endpoint:

POST /orders/:id/process

This endpoint simulates a multi-step asynchronous processing pipeline.

When `/process` is called:

1. Fetch the order
2. Validate it is in `CONFIRMED` state
3. In parallel:
    - Simulate inventory reservation (async delay)
    - Simulate payment authorization (async delay)
4. When both succeed:
    - Change status to `PROCESSED`
    - Publish event `order.processed`
    - `Order.processed` is consumed and simulates sending email
5. Return response

Rules

- If order not found → 404
- If order not `CONFIRMED` → 409
- If any async step fails → return 500
- All operations must be fully non-blocking

Architecture Requirements

Repository Layer (Mocked):

```
public interface OrderRepository {

    Future<Order> create(OrderDraft draft);

    Future<Optional<Order>> findById(String id);

    Future<List<Order>> find(OrderQuery query);

    Future<Order> updateStatus(String id, Status newStatus);

}
```

Implementation Rules

- Runtime: Use an **InMemory implementation** (Map-based)
- Tests: Use mocks (Mockito or similar)

Repository methods must return `Future` to simulate async behavior.

Business Logic Layer

PriceService

```
Future<BigDecimal> getPrice(String sku);
```

Implementation rules:

- Use hardcoded price map
- Add artificial async delay (e.g., `vertx.setTimer`)
- No database usage

Order Total Calculation

total = sum(price * qty)

This must be done asynchronously using Futures composition.

Technical Requirements

Mandatory

- Use `Vertx`, `Verticle`, `Router`
- Use `vertx-web`
- JSON serialization/deserialization
- Centralized error handling (failureHandler)
- Proper HTTP status codes:
    - 201
    - 400
    - 404
    - 409
- Log a request ID:
    - Either from `X-Request-Id` header
    - Or generate one if missing

Testing

- At least 2 unit tests for service layer (with mocked dependencies)
- At least 2 integration tests for HTTP layer (Vertx JUnit 5)

Definition of Done

- Project builds and runs
- All endpoints functional
- Validation works correctly
- Proper error codes returned
- Tests are passing
- README file includes:
    - How to run
    - Example curl commands
    - How to execute tests