IMPLEMENTATION DETAILS

For interprocess communication between Order service and Reporting service, I used RabbitMQ. So that these services became loosely coupled.

Gateway:

Gateway takes Order Service and Reporting Service urls' at startup and routes based on the path

/order** -> Order Service
/report** -> Reporting Service

Order Management Service:

- Db Migration:
  I used a Postgresql Database, and flyway handles the migrations. Migration file can be found under src/main/resources/db/migration

- OrderRestController, OrderRestControllerAdvisor, OrderService:
  After request validation, restController sends dto to OrderService, OrderRestControllerAdvisor handles exceptions and generates error responses.

- Transactional Outbox Pattern
  I used this pattern to make sure events are saved to "outbox" when an "order" is created, updated or deleted within a transaction. This ensures in OrderManagement Service, each event (order change in database) is saved to outbox table.

  This table is consumed by a scheduledTask in DomainEventPublisherService class which is polling the outbox table, sending to RabbitMQ, and deletes from the outbox table.

  That is not a very desired solution to that, a log trailing CDC (like Debezium) could be used but for this case, I preferred this simpler approach.
  Also if this DB was an Aurora RDS Postgres instance, then lambda_invoke trigger would be a great solution that provides event driven capability like DynamoDb Streams.

  DomainEventPublisherService uses DomainEventPublisher interface to send messages to queues. RabbitMQDomainEventPublisher is an implementation of this class, and this class is injected into the context with configuration. So in future if Kafka is preferred, then a class that implements the interface needs to be injected into the context, and there is no need to change any other implementation.

- Todo
  There is one TODO in OrderRepositoryIntegrationTest which is commented.

Reporting Service

- Db Migration
  Same as Order Service, but additionally I used triggers to update cumulative order counts (that was not a requirement though)

- RabbitListener
  This bean is injected into context and after mapping messages to DomainEvents, it passes them to OrderEventConsumer. So in future if Kafka listener is needed, then Kafka Listener bean must be injected into the context, and no other class is affected from that change.

- Idempotency
  To achieve idempotency, instead of using the transactional “ProcessedMessages” approach, I used the “aggregateVersion” (provided by OrderManagement Service) inside the table “geometry_orders” as orderVersion.
  Since the events are not incremental but cumulative, for instance if I processed V6 of a specific order, I must not process the V5 of that order when it comes after.

  So the update statement checks version like an optimistic locking mechanism.

  One little improvement here, when the OrderDeleted event comes, instead of deleting, we can mark it as “deleted” to eliminate one very unlikely case that OrderCreated comes after OrderDeleted.

- TODO
  There is one TODO in RabbitMqIntegrationTests which is commented
  Also I did not write all tests for this service, because actually the deadline was tough for this case. If requested, and more time is provided, I can provide as well.