Report – Ticketchief

Maximilian Fuchs

Interfaces

Input ports:

- `OrderServicePort (placeOrder, cancelOrder, finalizeOrder, addItem,removeItem, getInvoice)`
  How it's used: implemented by as REST-API for user-initiated actions, handles fundamental operations
  Relevance: the core use cases are exposed as pure Java methods with domain types.

- `OrderPaymentServicePort(onPaymentProcessed)`
  How it's used: implemented to handle incoming messages from the payment system
  Relevance: isolates message broker details; the core only sees a semantic "payment processed" event.

Output ports:

- `InvoicePort (generateInvoice)`
  How it's used: The invoice system is reached through a port. Whether the adapter calls a REST API or an SDK is irrelevant to the core; the port hides it.
  Relevance: replace invoice provider without touching domain/application code.

- `OrdersRepositoryPort (save, delete, find)`
  How it's used: Abstracts persistence using Spring Data JPA
  Relevance: lets tests use in-memory fakes and enables persistence changes without impacting business code.

- `PublishPaymentRequestedPort (publishPaymentRequested)`
  How it's used: publishes a *PaymentRequested* event to trigger payment flow.
  Relevance: decouples order lifecycle from payment latency/failures and enables reliable, retryable processing.

- `PublishEmailRequestedPort (publishEmailRequest)`
  How it's used: After invoicing, the service publishes an *EmailRequested* message to the mail system.
  Relevance: keeps email concerns out of the core and allows separate scaling/failure domains.

DTOs

- `InvoicePort.InvoiceResult(invoiceId, url)`
  Relevance: the core needs only an identifier and an access URL to proceed (e.g., for emailing). Anything more stays outside the domain.

Domain model (behavior + state):

- `Order (id, userId, items, status` with state transitions
  Relevance: enforces invariants (e.g., prevents edits when frozen) and encapsulates lifecycle, so application services orchestrate through domain behavior rather than mutating raw structures

- `CartItem(id, eventId, seatId, unitPriceCents`)
  Relevance*:* a value object for line items that the domain Order aggregates. Using a record keeps it immutable and serialization-friendly.

Event DTOs (integration):

- `PaymentRequestedEvent(correlationId, orderId, amountCents)`
  Relevance: correlationId to identitfy potential duplicates. Reason used for feedback if payment failed. canonical event shape at the boundary lets the app react to external facts without coupling to a specific broker library.

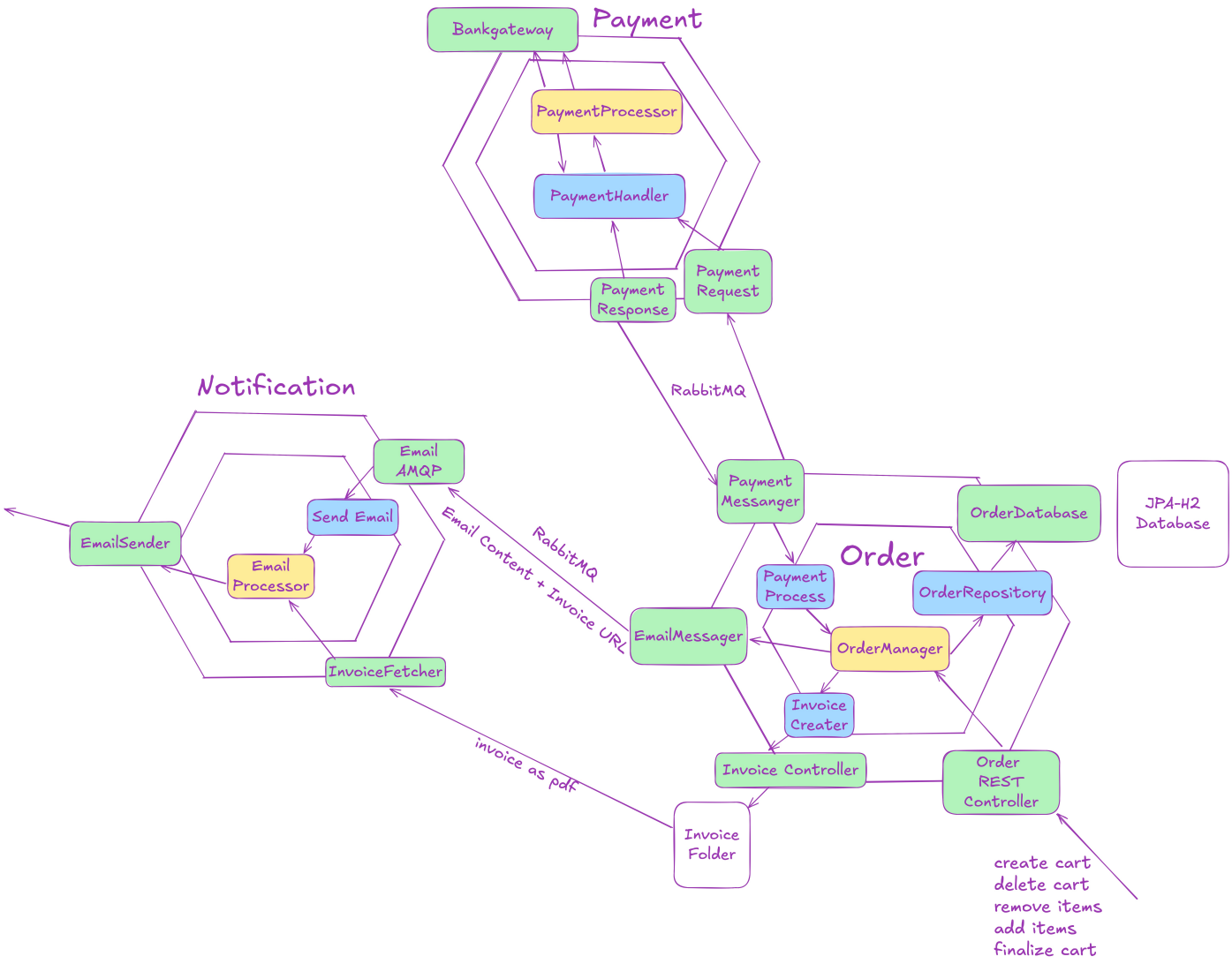
- `PaymentProcessedEvent(correlationId, orderId, status, reason)`
  Relevance: correlationId to identitfy potential duplicates. Reason used for feedback if payment failed. canonical event shape at the boundary lets the app react to external facts without coupling to a specific broker library.

- `EmailSendRequestedEvent(correlationId, orderId, toEmail, subject, bodyText, invoiceUrl)`
  Relevance: encapsulates all the information necessary for downstream email services to operate autonomously.

These DTOs are **small, immutable, serialization-friendly**. They align with ports, so each boundary exchanges only the data the use case needs.

Hexagonal Architecture

This application uses several principles of hexagonal architecture design. The outer layers of the system depend on the core and the business logic. The domain layer, represented by entities like Order and CartItem, remains free of any Spring, JPA, or messaging imports.
Next is the application layer with the services. It manages the use cases and implements the interfaces of the ports. Every use case is encapsulated in a separate port. The adapters connect these abstractions with the actual framework. Because of that, it is quite convenient to change technologies, since only the adapter needs to be modified.
Through the use of ports, every external interaction, such as the invoice service or the email system, is isolated from the core, which makes the codebase easier to maintain.
The design also embraces message orientation without direct coupling to a broker. The events are modeled as domain-driven message contracts, while the adapters determine the actual transport technology. This allows messaging semantics to evolve independently of infrastructure choices.
The architecture also helps make the edges replaceable. Since every external dependency is defined by a port, it is simple to test the application using mocks for the components.


Payment
Bankgateway
PaymentProcessor
PaymentHandler
Payment Request
Payment Response
RabbitMQ
Notification
Email AMQP
Send Email
EmailSender
Email Processor
InvoiceFetcher
RabbitMQ
Email Content + Invoice URL
Payment Messanger
OrderDatabase
JPA-H2 Database
Order
Payment Process
OrderRepository
EmailMessager
OrderManager
Invoice Creater
Invoice Controller
Order REST Controller
invoice as pdf
Invoice Folder
create cart
delete cart
remove items
add items
finalize cart