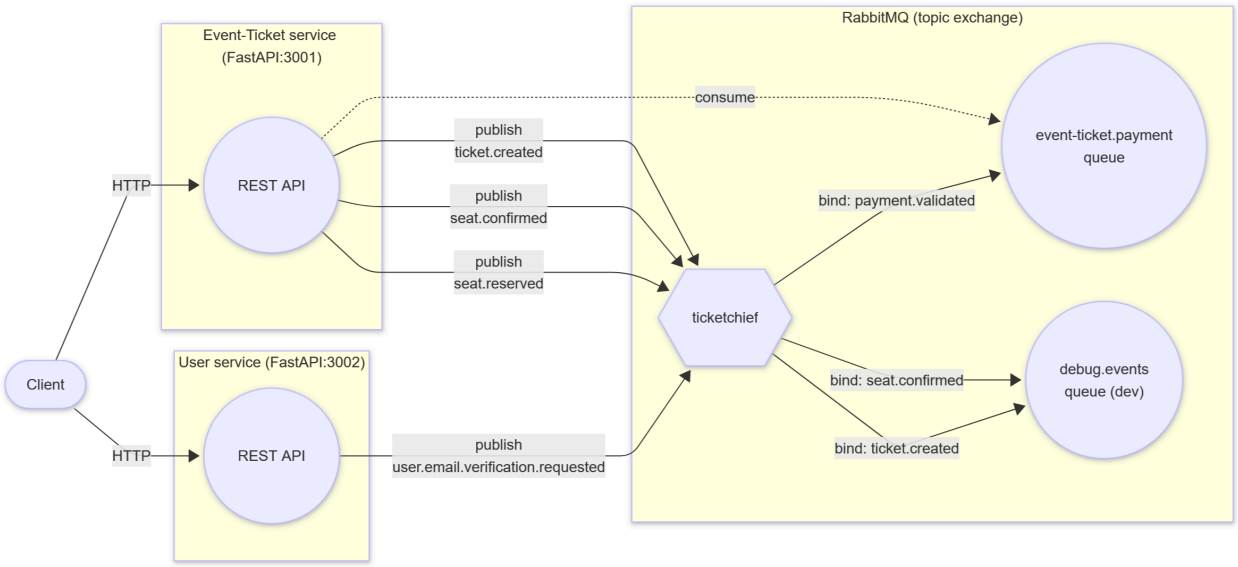
Sahib Khokhar

TicketChief: Event Ticket and User Services

Description

TicketChief is a simple ticketing platform for events. This delivery implements two hexagonal services: `user` and `event-ticket`. Both services expose REST APIs and integrate message-driven communication with RabbitMQ. There is no database in this as all state is in memory. The scenario shown in `SCENARIO.md` allows a user to create an account, create an event, reserve seats, and a payment confirmation (message) results in an issued per-seat ticket with a generated unique QR code. This QR code can then be used to verify the seat is theirs.

High-level Architecture



- Two hexagonal services: `user` and `event-ticket`.
- A single topic exchange `ticketchief` is used to communicate.
- REST is used for synchronous operations; messages are used for asynchronous workflows and fan-out.
- No database is used in this phase; all state is in memory.
- Flow: user registers → creates event → reserves seats → payment is validated (message) → seats are confirmed → per-seat tickets are created.
- `payment.validated` is consumed by `event-ticket` to confirm reserved seats.
- On reservation, `seat.reserved` is published; after confirmation, `event-ticket` publishes `seat.confirmed` and one `ticket.created` per seat (with QR).

Service: User

Interfaces

- REST

  - POST `/users` (register with email)
  - GET `/users/{id}` (profile)
  - POST `/login` (mock for this phase)
  - DELETE `/users/{id}` (remove the user)
- Messages (publish)
  - `user.email.verification.requested` after registration

Justification

- These actions are interactive and they need to be immediate and deterministic. Each REST endpoint here should return a response immediately as it is the only way to ensure the user gets a response on the frontend later. The message gets published to the message bus to be consumed later by the `notifications` service so that an email can be sent to the user. This is a message and not a REST endpoint because the user does not need to wait for the email to be sent to them and the email service can be slow or fail at any point.

DTOs and relevance

- `UserCreate { email, name }`: this is the minimal input to establish an identity for a user.
- `UserView { id, email, name }`: this is the view of a user that is returned to the user.
- `LoginRequest { email }`: this is a test login request to ensure the user can login to the system.
- Message `user.email.verification.requested { userId, email }`: minimal data to trigger the email delivery later.

Hexagonal architecture adherence

- Inbound adapter: FastAPI routes (HTTP) will translate the request to the application use case.
- Domain: user registry with uniqueness defined on the email.
- Outbound adapter: we use RabbitMQ to publish a message that is consumed later separately.

Here we isolate application logic from external dependencies by using adapters to translate the information through HTTP and RabbitMQ into domain inputs.

Service: Event & Ticket

Interfaces

- REST
  - POST `/events` (create, requires `userId`)
  - PATCH `/events/{id}` (rename, creator-only)
  - GET `/events/{id}` (read seat map for the event)
  - POST `/events/{id}/reserve` (reserve seats; requires `userId`)

  - ▸ POST `/tickets/verify` (check `ticketId` ↔ `eventId` and confirmed seat)
- Messages
  - ▸ Consume: `payment.validated { orderId, eventId, seats[], userId }`
  - ▸ Publish: `seat.reserved { eventId, seats[] }`, `seat.confirmed { eventId, seats[] }`, `ticket.created { ticketId, eventId, seat, qr }`

Justification

- Event creation, reservation, and verification need strict and immediate validation and clear synchronous outcomes for the user. If these came in asynchronously, the user would not get a response on the frontend when they would need it. We consume the payment validated message to confirm the seats instead of waiting since that can happen at anytime without blocking the user. We publish the seat reserved and confirmed messages to notify other services that are interested in these events. These would be like notifications or UI updates at any point.

DTOs and relevance

- `EventCreate { name, rows, cols, userId }`: minimal inputs to creating an event.
- `EventView { id, name, rows, cols, seats: map }`: minimal view of an event that is returned to the user.
- `ReserveRequest { userId, seats[] }`: capture the identity of the user trying to reserve seats so that we can validate the user is the one trying to buy the tickets later.
- `TicketVerifyRequest { ticketId, eventId }` → `TicketVerifyResponse { valid, reason?, ticketId?, eventId?, seat? }`: we need to verify the ticket is valid and if it is, we return the seat that is reserved for the user.

Hexagonal architecture adherence

- Inbound adapter: FastAPI routes map HTTP to use cases.
- Domain: we have an in memory seat grid and rules for the event. This is a core domain that is not dependent on the external dependencies at all.
- Outbound adapters: RabbitMQ publisher to publish the messages that will be consume by other services like `notifications`.
- Isolation: these domain decisions are independent of the other dependencies.

Coupling and Messaging Rationale

- We use REST when a human needs a fast and deterministic answer.
- We use messages for long-running orchestration, retries, and multi-consumer fan-out messages like payments, confirmations, ticket delivery.
- We use a single topic exchange (`ticketchief`) to reduce assumptions between services and avoid synchronous coupling.