

Università degli Studi di Salerno
Corso di Ingegneria del Software

BEAT – Booking Events And Tickets

Object Design Document (ODD)
Versione 1.3b



Data: 25/11/2025

Coordinatore del progetto: Prof. Andrea De Lucia

Partecipanti: Carnevale Luigi[0512119029], Di Manso Carmine[0512119521], Clemente Manuel[0512119395]

Scritto da: Carnevale Luigi, Di Manso Carmine, Clemente Manuel

Repository GitHub: [github.com/Luigi-Carnevale/BEAT-Booking Events And Tickets](https://github.com/Luigi-Carnevale/BEAT-Booking-Events-And-Tickets)

Revision History

Data	Versione	Descrizione	Autore
25-11-2025	1.0	Creazione ODD	Carnevale Luigi, Di Manso Carmine, Clemente Manuel
21-12-2025	1.1	fondamenta e accesso	Carnevale Luigi
21-12-2025	1.2	completata parte relativa al business	Di Manso Carmine
22-12-2025	1.3b	3.2...3.2.2	Clemente Manuel

Indice

Revision History

1	Introduction	1
1.1	Object design trade-offs	1
1.2	Assunzioni di progettazione	2
1.3	Interface documentation guidelines	2
1.4	Definitions, acronyms, and abbreviations	3
1.5	References	4
2	Packages	5
2.1	Overview dei package (architettura complessiva)	5
2.2	Dipendenze tra package (architettura complessiva)	5
2.3	Sottopackage (struttura pronta per il team)	5
2.3.1	User Management (COMPLETATO – Task 1)	6
2.3.2	Event Management (TODO)	6
2.3.3	Booking & Ticketing (completato—task 2)	6
2.3.4	Payment Management (completato—task 2)	6
2.3.5	Notification (TODO)	6
2.3.6	Persistence & Data Management (TODO)	6
2.4	Uso atteso e organizzazione dei file (linee guida comuni)	7
3	Class interfaces	8
3.1	User Management Subsystem (COMPLETATO – Task 1)	8
3.1.1	Classi di dominio principali	8
3.1.1.1	User	8
3.1.1.2	Role	9
3.1.1.3	Session	9
3.1.2	Componenti di sicurezza	9
3.1.2.1	PasswordHasher	9
3.1.2.2	SessionManager	10
3.1.3	Service layer	10
3.1.3.1	AuthService	10
3.1.3.2	UserService	10
3.1.3.3	RoleAdminService	10
3.1.4	DAO e persistenza	11
3.1.4.1	UserDAO	11
3.1.5	Controller	11
3.1.5.1	UserController	11
3.2	Event Management Subsystem	12
3.2.1	Classi di dominio principali	12
3.2.2	Service layer ()	13
3.2.3	DAO/Repository (TODO)	13
3.2.4	Controller/Boundary (TODO)	13
3.3	Booking & Ticketing Subsystem	14
3.3.1	Classi di dominio principali	14
3.3.2	Service layer	14

3.3.3	"DAO/Repository"	15
3.3.4	Controller/Boundary	15
3.4	Payment Management Subsystem	16
3.4.1	Service layer	16
3.4.2	DAO/Repository	16
3.5	Notification Subsystem (TODO)	17
3.5.1	Service + adapter (TODO)	17
3.6	Persistence & Data Management Subsystem (TODO)	18
3.6.1	DataSource/Transaction management (TODO)	18
3.6.2	DAO comuni (TODO)	18

1 Introduction

1.1 Object design trade-offs

Questa sezione descrive i principali compromessi effettuati durante la progettazione a oggetti del sistema *BEAT – Booking Events And Tickets*. I trade-off riportati sono stati selezionati in modo coerente con i requisiti del progetto (RAD) e con gli obiettivi di progettazione (SDD), con particolare attenzione a sicurezza, manutenibilità e prestazioni. In particolare, le scelte progettuali qui motivate sono valutate rispetto ai *Design Goals* definiti nello SDD (es. prestazioni/tempi di risposta, sicurezza e manutenibilità), e mirano a preservare la coerenza architetturale complessiva del sistema.

- **Buy vs. build (riuso vs. sviluppo ad hoc):** per ridurre complessità e rischi, il progetto adotta componenti consolidati dove possibile. In particolare, per la gestione sicura delle credenziali si preferiscono algoritmi standard e robusti (es. *bcrypt* con output a lunghezza fissa), evitando implementazioni personalizzate. Al contrario, la logica di dominio (registrazione, controllo unicità email, gestione ruoli e sessioni applicative) viene implementata specificamente per BEAT, poiché dipende direttamente dai requisiti funzionali.
- **Sicurezza vs. usabilità:** BEAT deve garantire autenticazione e autorizzazione robuste (controllo degli accessi per ruoli). Questo introduce controlli aggiuntivi (validazioni server-side, gestione sessioni, protezioni contro attacchi comuni), che aumentano il numero di verifiche rispetto a un flusso minimalista, ma sono necessari per soddisfare requisiti di sicurezza e prevenire accessi non autorizzati.
- **Robustezza vs. semplicità:** si privilegia una gestione esplicita di errori e casi limite (input non validi, credenziali errate, account non abilitato, tentativi ripetuti), anche a costo di aumentare leggermente la quantità di logica nei servizi. Questa scelta migliora affidabilità e testabilità, evitando comportamenti impliciti o non deterministici. È inoltre coerente con i flussi alternativi e le eccezioni esplicitate nel RAD (es. registrazione fallita per email già in uso, autenticazione fallita, operazioni non consentite per mancanza di autorizzazione).
- **Accoppiamento vs. coesione (separazione a strati):** per mantenere basso l'accoppiamento e alta la coesione, la logica è separata in: *Controller* (gestione richieste), *Service* (regole di dominio e casi d'uso) e *DAO* (persistenza). Le dipendenze verso il DB sono incapsulate nel layer di persistenza (DAO), evitando accessi diretti dalle classi di dominio.
- **Sessione server-side vs. token stateless:** BEAT è una web application con necessità di controllo ruolo e gestione flussi (login, cambio profilo/ruolo, logout). Per mantenere la soluzione coerente con un'impostazione MVC tradizionale e con i vincoli di progetto, si adotta una gestione di sessione applicativa (es. *SessionManager*) che associa un identificativo di sessione ad un utente autenticato. Laddove necessario, la sessione può esporre un *token* applicativo, ma la fonte di verità resta lo stato lato server. Tale token, se presente, non è utilizzato come meccanismo di autenticazione *stateless*, ma esclusivamente come identificatore di sessione o correlatore applicativo.

- **Prestazioni vs. consistenza (in particolare su autenticazione/autorizzazione):** le operazioni di login e controllo ruolo devono essere frequenti e veloci. Si evitano chiamate ridondanti al database memorizzando in sessione le informazioni minime (es. `userId` e `role`), senza duplicare dati sensibili. Questo migliora tempi di risposta mantenendo consistenza dei permessi.

1.2 Assunzioni di progettazione

La progettazione a oggetti del sistema BEAT si basa sulle seguenti assunzioni, derivate dai vincoli e dalle scelte architetturali riportate in RAD e SDD:

- Il sistema è una web application conforme al paradigma MVC e a una decomposizione a strati (Presentation → Application → Data).
- Il controllo degli accessi è basato su ruoli applicativi distinti (RBAC), con verifiche di autorizzazione eseguite lato server.
- La persistenza è implementata su DB relazionale, con accesso mediato da DAO e garanzie di consistenza transazionale (ACID) quando richieste.
- Le operazioni critiche (autenticazione, autorizzazione, modifica ruoli, scritture sul DB) sono eseguite lato server e non demandate al client.
- I sottosistemi collaborano tramite interfacce ben definite, minimizzando l'accoppiamento e preservando la coesione interna.

1.3 Interface documentation guidelines

Questa sezione definisce le regole di documentazione e le convenzioni adottate per le interfacce pubbliche (classi e metodi). L'obiettivo è garantire uniformità nello stile, ridurre ambiguità e facilitare sia lo sviluppo che il testing.

Convenzioni di naming e responsabilità

- Le **classi** sono nominate con **sostantivi singolari** (es. `User`, `Session`, `Role`).
- I **metodi** sono nominati con **verbi** (es. `login()`, `registerClient()`, `changeRole()`).
- Le classi **Service** contengono la logica di dominio e coordinano DAO e componenti di sicurezza.
- Le classi **DAO** incapsulano completamente l'accesso ai dati e non contengono logica di business.
- I **Controller** gestiscono richieste/risposte e delegano ai Service, senza implementare regole di dominio.

Regole minime per documentare le operazioni pubbliche

Per ogni operazione pubblica esposta:

- **Scopo** e descrizione del comportamento.
- **Parametri** con vincoli (es. non null, range, formato).
- **Ritorno** e significato (incluso cosa rappresenta un valore “vuoto”).
- **Precondizioni** (stato richiesto, vincoli sugli input).
- **Postcondizioni** (garanzie dopo l’esecuzione).
- **Effetti collaterali** (scritture su DB, creazione sessione, log, ecc.).
- **Eccezioni** (tipi e condizioni) con messaggi non informativi in modo pericoloso (evitare *user enumeration*).
- **Casi limite** (stringhe vuote, formati errati, tentativi ripetuti, concorrenza).

Linee guida di sicurezza (obbligatorie)

- Le password **non sono mai memorizzate in chiaro**: viene memorizzato solo l’hash.
- L’autorizzazione è verificata **lato server** prima di eseguire azioni protette (RBAC).
- Le query verso il DB usano statement parametrizzati (o equivalenti) per prevenire SQL Injection.
- Gli input testuali destinati alla vista devono essere sanificati/validati per mitigare XSS.

1.4 Definitions, acronyms, and abbreviations

1. **ODD** = Object Design Document;
2. **OOSE** = Object-Oriented Software Engineering;
3. **API** = Application Programming Interface;
4. **MVC** = Model View Controller;
5. **DAO** = Data Access Object;
6. **DTO** = Data Transfer Object;
7. **JDBC** = Java DataBase Connectivity;
8. **HTTP/HTTPS** = HyperText Transfer Protocol / Secure;
9. **ACID** = Atomicity, Consistency, Isolation, Durability;
10. **RBAC** = Role-Based Access Control;
11. **XSS** = Cross-Site Scripting;
12. **CSRF** = Cross-Site Request Forgery.

1.5 References

- “BEAT – Booking Events And Tickets - Problem Statement”, versione 1.1.
- “BEAT – Booking Events And Tickets - Requirement Analysis Document (RAD)”, versione 2.0.
- “BEAT – Booking Events And Tickets - System Design Document (SDD)”, versione 1.0.
- B. Bruegge, A. H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns, and Java*, Third Edition.

2 Packages

Questa sezione descrive la decomposizione del sistema in package e l'organizzazione dei file. Il documento è condiviso: alcune sottosezioni sono completate (Task 1), mentre altre sono lasciate come struttura pronta (**TODO**) per i membri del team responsabili.

2.1 Overview dei package (architettura complessiva)

Package	Responsabilità	Layer
<code>beat.user</code>	Gestione utenti, autenticazione/autorizzazione, profilo e ruoli.	Application
<code>beat.event</code>	Gestione eventi (creazione, modifica, pubblicazione, capienza).	Application
<code>beat.booking</code>	Prenotazioni, carrello, emissione biglietti, prevenzione overbooking.	Application
<code>beat.payment</code>	Gestione transazioni e integrazione con gateway di pagamento.	Application
<code>beat.notification</code>	Notifiche (email conferme, annulli, modifiche eventi).	Application
<code>beat.persistence</code>	DAO/Repository, accesso DB, transazioni, mapping.	Data

2.2 Dipendenze tra package (architettura complessiva)

Le dipendenze devono essere orientate “verso il basso” (Presentation → Application → Data). I package applicativi non devono dipendere dall'implementazione concreta della persistenza oltre le interfacce previste.

Da (From)	Dipende da (Depends on)
<code>beat.user</code>	<code>beat.persistence</code>
<code>beat.event</code>	<code>beat.persistence</code> , <code>beat.user</code> (autorizzazione organizzatore/admin)
<code>beat.business.booking</code>	<code>beat.persistence</code> , <code>beat.event</code> , <code>beat.business.payment</code> , <code>beat.notification</code> , <code>beat.user</code>
<code>beat.business.payment</code>	<code>beat.persistence</code>
<code>beat.notification</code>	(provider esterni; accesso mediato da adapter)

2.3 Sottopackage (struttura pronta per il team)

Di seguito si riporta una struttura suggerita per ogni sottosistema. Le sottosezioni marcate **TODO** sono da completare dai responsabili dei task corrispondenti.

2.3.1 User Management (COMPLETATO – Task 1)

- `beat.user.model`
- `beat.user.service`
- `beat.user.controller`
- `beat.user.security`
- `beat.user.dto`

2.3.2 Event Management (TODO)

- `beat.event.model` *TODO*
- `beat.event.service` *TODO*
- `beat.event.controller` *TODO*
- `beat.event.dto` *TODO*

2.3.3 Booking & Ticketing (completato—task 2)

- `beat.booking.model`
- `beat.booking.service`
- `beat.booking.controller`
- `beat.booking.dto`

2.3.4 Payment Management (completato—task 2)

- `beat.payment.model`
- `beat.payment.service`
- `beat.payment.controller`
- `beat.payment.dto`

2.3.5 Notification (TODO)

- `beat.notification.service` *TODO*
- `beat.notification.adapter` *TODO*

2.3.6 Persistence & Data Management (TODO)

- `beat.persistence.dao` *TODO*
- `beat.persistence.datasource` *TODO*
- `beat.persistence.tx` *TODO*

2.4 Uso atteso e organizzazione dei file (linee guida comuni)

- **Regola di accesso ai dati:** l'accesso al DB avviene esclusivamente tramite `beat.persistence`.
- **Separazione delle responsabilità:** la logica di business risiede nei servizi applicativi, non nei controller.
- **Naming:** nomi coerenti con dominio (es. `EventoService`, `PrenotazioneDAO`, `PagamentoDTO`).
- **Gestione eccezioni:** eccezioni di dominio/servizio convertite in risposte coerenti (lato controller) o gestite da handler globali.

3 Class interfaces

Le interfacce di classe descritte in questa sezione rappresentano la traduzione, a livello di progettazione a oggetti, dei requisiti e dei modelli definiti nel RAD e delle scelte architetturelle riportate nel SDD. L'organizzazione delle responsabilità e delle dipendenze tra classi è coerente con la separazione a strati e con la decomposizione in sottosistemi. Questa sezione descrive le classi e le loro interfacce pubbliche. Il documento è condiviso: il sottosistema **User Management** è completato (Task 1), mentre gli altri sottosistemi sono lasciati come struttura pronta (**TODO**) per i rispettivi responsabili.

3.1 User Management Subsystem (COMPLETATO – Task 1)

Il sottosistema di *User Management* è responsabile della gestione dell'identità degli utenti, dell'autenticazione, dell'autorizzazione e dell'amministrazione dei ruoli applicativi. Esso costituisce un sottosistema trasversale, utilizzato implicitamente o esplicitamente da tutti gli altri moduli del sistema BEAT.

Le responsabilità di questo sottosistema derivano direttamente dai casi d'uso e dai requisiti funzionali definiti nel RAD e includono:

- registrazione e gestione del profilo utente;
- autenticazione sicura degli utenti;
- gestione dei ruoli applicativi e controllo degli accessi (RBAC);
- gestione delle sessioni applicative;
- supporto agli amministratori per la gestione dei ruoli.

Il sottosistema è progettato secondo una separazione a strati, distinguendo chiaramente tra classi di dominio, servizi applicativi, componenti di sicurezza e accesso ai dati.

3.1.1 Classi di dominio principali

3.1.1.1 User

Rappresenta un utente registrato al sistema. È un'entità persistente e costituisce il fulcro del modello di dominio del sottosistema.

Responsabilità principali:

- rappresentare l'identità dell'utente;
- mantenere le informazioni anagrafiche e di contatto;
- mantenere il riferimento al ruolo applicativo assegnato.

Attributi principali (coerenti con RAD e SDD):

- **id**: identificativo univoco;
- **email**: identificativo logico dell'utente (vincolo di unicità);
- **passwordHash**: hash della password (bcrypt, lunghezza fissa);

- **role**: ruolo applicativo associato;
- **enabled**: stato dell'account.

La classe **User** non contiene logica di sicurezza o di persistenza.

3.1.1.2 Role

Rappresenta un ruolo applicativo del sistema BEAT. I ruoli sono definiti in modo coerente con la matrice degli accessi del SDD.

Ruoli previsti:

- Cliente;
- Organizzatore;
- AdminCatalogo;
- AdminOrdini;
- AdminRuoli.

La classe **Role** può essere implementata come enumerazione o entità persistente, a seconda delle scelte di implementazione, ma mantiene una semantica forte nel dominio.

3.1.1.3 Session

Rappresenta una sessione applicativa attiva associata a un utente autenticato.

Responsabilità:

- mantenere il riferimento all'utente autenticato;
- tracciare lo stato di autenticazione;
- memorizzare informazioni minime di contesto (es. **userId**, **role**).

La sessione è gestita lato server e non contiene dati sensibili non necessari.

3.1.2 Componenti di sicurezza

3.1.2.1 PasswordHasher

Componente responsabile della gestione sicura delle password.

Responsabilità:

- generare hash sicuri delle password in fase di registrazione;
- verificare la validità delle credenziali in fase di login.

L'algoritmo utilizzato è bcrypt, coerentemente con quanto indicato nello SDD, garantendo:

- uso automatico del salt;
- resistenza a brute force e rainbow table;
- output a lunghezza fissa.

3.1.2.2 SessionManager

Responsabile della creazione, invalidazione e gestione delle sessioni applicative.

Responsabilità:

- creare una nuova sessione a seguito di login riuscito;
- invalidare la sessione in fase di logout;
- recuperare informazioni di sessione per il controllo degli accessi.

Il **SessionManager** rappresenta la fonte di verità per lo stato di autenticazione lato server.

3.1.3 Service layer

3.1.3.1 AuthService

Fornisce i servizi di autenticazione del sistema.

Operazioni principali:

- `login(email, password);`
- `logout(sessionId).`

Responsabilità:

- validare le credenziali fornite;
- coordinare `UserDAO`, `PasswordHasher` e `SessionManager`;
- prevenire comportamenti di *user enumeration*.

3.1.3.2 UserService

Gestisce i casi d'uso relativi alla registrazione e alla gestione del profilo utente.

Responsabilità:

- registrazione di nuovi utenti;
- verifica dell'unicità dell'email;
- aggiornamento delle informazioni di profilo.

3.1.3.3 RoleAdminService

Servizio dedicato all'amministrazione dei ruoli applicativi.

Responsabilità:

- assegnazione e modifica dei ruoli utente;
- verifica delle autorizzazioni dell'amministratore;
- coerenza con la matrice degli accessi definita nello SDD.

3.1.4 DAO e persistenza

3.1.4.1 UserDAO

Incapsula completamente l'accesso ai dati relativi agli utenti.

Responsabilità:

- operazioni CRUD sugli utenti;
- recupero utenti per email o identificativo;
- supporto alle operazioni di autenticazione.

Il **UserDAO** utilizza query parametrizzate e non contiene logica di business.

3.1.5 Controller

3.1.5.1 UserController

Gestisce le richieste HTTP relative al sottosistema User Management.

Responsabilità:

- ricezione e validazione preliminare degli input;
- delega ai servizi applicativi;
- gestione delle risposte e degli errori.

Il controller non implementa regole di dominio né logica di sicurezza avanzata, che sono demandate ai Service e ai componenti dedicati.

3.2 Event Management Subsystem

Il sottosistema di Event Management è responsabile della gestione del catalogo degli eventi disponibili sulla piattaforma BEAT. Esso supporta le funzionalità di consultazione, ricerca e filtraggio degli eventi, nonché la visualizzazione dei dettagli associati (categoria, luogo, disponibilità).

Le responsabilità di questo sottosistema derivano direttamente dai requisiti funzionali e dai casi d'uso definiti nel RAD e includono:

- visualizzazione del catalogo eventi;
- ricerca eventi per parola chiave;
- filtraggio eventi per categoria e data;
- visualizzazione dei dettagli di un evento;

Il sottosistema è progettato secondo una separazione a strati, distinguendo tra: classi di dominio, servizi applicativi, componenti di accesso ai dati e controller.

3.2.1 Classi di dominio principali

Evento

Rappresenta un evento pubblicato sulla piattaforma BEAT.

Responsabilità principali:

- rappresentare i dati identificativi e descrittivi dell'evento;
- mantenere le informazioni di prezzo e disponibilità;
- mantenere i riferimenti a categoria, location e recensioni.

Attributi principali (coerenti con RAD e SDD):

- id;
 - titolo: titolo dell'evento;
 - descrizione;
 - data: data e ora di svolgimento;
 - prezzo;
 - posti_{disponibili};
 - categoria.
 - luogo: luogo di svolgimento dell'evento;
- La classe Evento non contiene logica di persistenza né di business complessa.

Categoria

Rappresenta una categoria a cui possono appartenere uno o più eventi.

Responsabilità principali:

- classificare gli eventi;
- supportare le funzionalità di filtraggio del catalogo.

Attributi principali:

- id;
- nome.

Luogo

Rappresenta il luogo fisico in cui si svolge un evento.

Responsabilità principali:

- fornire informazioni logistiche associate a un evento;
- supportare la visualizzazione dei dettagli evento.

Attributi principali:

- id;
- nome;

indirizzo;
città.

3.2.2 Service layer ()

EventService

Il servizio applicativo che incapsula la logica di business relativa alla gestione e alla consultazione degli eventi.

Responsabilità:

coordinare l'accesso ai dati tramite i repository;

applicare i criteri di ricerca e filtraggio;

fornire una vista coerente del catalogo eventi ai controller e agli altri sottosistemi.

Operazioni principali:

listaEventi(); restituisce l'elenco completo degli eventi disponibili;

cercaEventi(titolo); ricerca eventi tramite nome;

filtraEventi(categoria, data): filtra eventi per categoria e/o data;

cerca_id(id) : *restituisce i dettagli di un singolo evento.*

3.2.3 DAO/Repository (TODO)

3.2.4 Controller/Boundary (TODO)

3.3 Booking & Ticketing Subsystem

Questo sottosistema BEAT gestisce: la creazione delle prenotazioni, il processamento dei pagamenti e la generazione dei titoli d'ingresso (ticket).

3.3.1 Classi di dominio principali

Queste classi rappresentano gli oggetti persistenti (**Booking**, **Ticket**, **Payment**):

Booking: Rappresenta l'ordine effettuato da un utente.

Attributi:

- id: int,
- totalAmount:
- BigDecimal,
- status: String,
- purchaseDate: LocalDateTime.

Ticket: Rappresenta il singolo titolo d'ingresso associato a una prenotazione.

Attributi:

- id: int,
- qrCode: String (stringa univoca per il QR),
- price: BigDecimal,
- bookingId: int.

Payment: Rappresenta la transazione finanziaria.

Attributi:

- id: int,
- transactionId: String,
- amount: BigDecimal,
- :

3.3.2 Service layer

Il layer di business coordina la logica di acquisto e garantisce l'atomicità delle operazioni (ACID).

Classe: BookingService

- **Responsabilità:** Gestisce il workflow di acquisto; previene l'overbooking, per gestire le operazioni concorrenti useremo lock ottimistici/pessimistic sugli eventi.
- **Operazioni:**
 - createBooking(userId, eventId, qty): BookingId: Crea una prenotazione in stato "PENDING" (**in attesa**) e riserva temporaneamente i posti. *Eccezioni:* SoldOutException, InvalidQuantityException.
 - finalizePurchase(bookingId, paymentData): Ticket: Coordina con il PaymentService e, in caso di successo, genera i ticket e sposta lo stato in "CONFIRMED" (**confermato**). *Postcondizioni:* Posti decrementati permanentemente, ticket persistiti.

Classe: PaymentService

- **Responsabilità:** Interfaccia con i gateway esterni ("esempi di metodi di pagamento da inserire") per la validazione del pagamento.

- **Operazioni:**

- `processPayment(amount, details): TransactionId`: Esegue l'addebito. *Eccezioni*: `PaymentFailedException`, `InsufficientFundsException`.

3.3.3 "DAO/Repository"

Interfacce per l'accesso ai dati tramite Spring Data JPA.

- **BookingRepository**: Fornisce metodi come `save(Booking)`, `findByUserId(int)`, e `query custom` per recuperare prenotazioni scadute (non pagate) da annullare.
- **TicketRepository**: Gestisce il salvataggio dei ticket e il recupero tramite QR code per la fase di validazione all'ingresso.
- **PaymentRepository**: Logga tutte le transazioni per scopi di audit e rimborsi.

3.3.4 Controller/Boundary

Classe: **BookingController**

- **Endpoint**: `POST /api/bookings/create`
 - *Payload*: { `eventId`, `quantity` }
 - *Descrizione*: Riceve la richiesta di prenotazione dall'utente autenticato.
- **Endpoint**: `POST /api/bookings/{id}/confirm`
 - *Payload*: { `paymentToken` }
 - *Descrizione*: Riceve i dati del pagamento e attiva il processo di finalizzazione nel Service.

3.4 Payment Management Subsystem

Questo sottosistema si occupa di gestire l'interazione con i provider finanziari esterni e della registrazione delle transazioni, garantendo la tracciabilità e la sicurezza dei pagamenti effettuati su BEAT.

3.4.1 Service layer

Classe: PaymentService

- **Responsabilità:** Validare i dati di pagamento forniti dall'utente, comunicare con i gateway esterni () e gestire l'esito della transazione.
- **Operazioni:**
 - processPayment(amount, details): TransactionId: Invia la richiesta di addebito al provider esterno per l'importo specificato. **Eccezioni:** PaymentFailedException (es. carta rifiutata), GatewayTimeoutException (problemi di connessione), InsufficientFundsException.
 - ?refundPayment(transactionId): boolean: Gestisce la procedura di rimborso in caso di annullamento di un evento o di una prenotazione valida. **Postcondizioni:** Stato della transazione aggiornato a "REFUNDED" nel database.

3.4.2 DAO/Repository

- **PaymentRepository:** Responsabile della gestione dei dati persistenti relativi ai pagamenti.
 - **Metodi principali:**
 - * save(Payment): Persiste i dettagli della transazione (ID esterno, timestamp, importo, stato).
 - * findByTransactionId(String): Recupera i dettagli di un pagamento a partire dal codice univoco restituito dal gateway.
 - * findAllByUserId(int): Consente di recuperare lo storico dei pagamenti per un determinato utente (utile per la visualizzazione dei ticket acquistati).

3.5 Notification Subsystem (TODO)

3.5.1 Service + adapter (TODO)

3.6 Persistence & Data Management Subsystem (TODO)

3.6.1 DataSource/Transaction management (TODO)

3.6.2 DAO comuni (TODO)