Università degli Studi di Salerno
Corso di Ingegneria del Software

BEAT – Booking Events And Tickets

System Design Document

Versione 1.0

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

Revision History

Data	Versione	Descrizione	Autore
18-11-2025	1.0	Creazione SDD	Carnevale Luigi, Di Manso Carmine, Clemente Manuel
20-11-2025	1.0	1 e 2	Carnevale Luigi, Di Manso Carmine, Clemente Manuel
21-11-2025	1.1	Persistent data management	Carnevale Luigi
22-11-2025	1.2	Access control and security	Carnevale Luigi
23-11-2025	1.3	Boundary conditions	Carnevale Luigi
24-11-2025	1.4	3.6,4.5	Di Manso Carmine

Indice

1 Introduction

1.1 Purpose of the system

L'obiettivo del progetto BEAT è sviluppare un sistema con interfaccia web per la gestione e prenotazione di eventi, che consenta agli utenti di consultare, prenotare e gestire biglietti per una varietà di eventi (concerti, seminari, corsi, spettacoli). Il sistema intende fornire una piattaforma intuitiva e dinamica che semplifichi sia la gestione amministrativa degli eventi sia l'esperienza dell'utente finale nella consultazione e prenotazione.

1.2 Design goals

ID	Obiettivo (Design Goal)	Descrizione per BEAT	Derivazione (da RNF)	Priorità
DG.01	Prestazioni	L'acquisto dei biglietti deve essere gestito con bassa latenza e alta concorrenza.	RNF-05 (Tempo di risposta)	ALTA
DG.02	Disponibilità	Il sistema deve essere operativo 24/7 per consentire la prenotazione continua.	RNF-01 (Uptime)	ALTA
DG.03	Sicurezza	Protezione dei dati utente e conformità ai protocolli di pagamento standard.	RNF-07 (Crittografia dati)	ALTA
DG.04	Scalabilità	Deve supportare un aumento del 50% di utenti attivi simultanei rispetto al picco previsto.	RNF-03 (Gestione carico)	MEDIA
DG.05	Manutenibilità	Il codice del Backend (Spring Boot) deve avere un basso accoppiamento per facilitare la modifica dei moduli.	RNF-12 (Modularità)	MEDIA

1.3 Definitions, acronyms, and abbreviations

1. **SC** = Scenarios;
2. **UC** = Use Case;
3. **CRUD** = Create Read Update Delete;

4. **DBA** = DataBase Administrator;
5. **MVC** = Model View Controller;
6. **JSP** = JavaServer Pages;
7. **DAO** = Data Access Object;
8. **JDBC** = Java DataBase Connectivity;
9. **XSS** = Cross-Site Scripting;
10. **CSRF** = Cross-Site request forgery;
11. **HTTP** = HyperText Transfer Protocol;
12. **HTTPS** = HyperText Transfer Protocol Secure;
13. **SDD** = System Design Document
14. **RAD** = Requirement Analysis Document
15. **ACID** = Atomicity, Consistency, Isolation, Durability.
16. **DG** = Design Goals
17. **RNF** = Requisiti Non Funzionali
18. **UI** = User Interface (in italiano: Interfaccia Utente).

1.4 References

- "BEAT Booking Events And Tickets - Problem Statement", Versione 1.1.
- B. Bruegge , A. H. Dutoit , "Object-Oriented Software Engineering: Using UML, Patterns, and Java", Third Edition.
- "BEAT Booking Events And Tickets - RAD", Versione 2.0.

1.5 Overview

- **Sezione 1 (Introduzione):** Stabilisce l'ambito del progetto BEAT, definisce gli obiettivi di progettazione (Design Goals) che guideranno le decisioni architetturali e fornisce un glossario di acronimi e riferimenti.
- **Sezione 2 (Current software architecture):** Si da una definizione dell'architettura adottata per lo sviluppo della web app, in quanto, essendo che la stiamo sviluppando, non abbiamo un architettura di sistema precedente per questo progetto.
- **Sezione 3 (Proposed software architecture):** Presenta il risultato dell'attività di **Subsystem Decomposition**. In questo capitolo, il sistema BEAT viene scomposto in un insieme di sottosistemi debolmente accoppiati, insieme alla definizione delle loro interfacce e delle relazioni di dipendenza.

- **Sezione 4 (Subsystem services):** Descrive nel dettaglio i servizi offerti dai sottosistemi, specificando le funzionalità che offrono, i parametri, di ingresso/uscita e una descrizione dell'operazione.

2 Current software architecture

L'architettura software di riferimento per il sistema BEAT è basata sul pattern architetturale a **tre livelli (Three-Tier Architecture)**, scelto per garantire una separazione delle responsabilità tra l'interfaccia utente, la logica di business e la gestione dei dati. Questa struttura modulare è fondamentale per soddisfare gli obiettivi di progettazione definiti nella sezione 1.2, in particolare la **Manutenibilità** e la **Scalabilità**, permettendo lo sviluppo e l'aggiornamento indipendente di ciascun livello. Il sistema è suddiviso in; **Presentation Tier**, responsabile dell'interazione con l'utente finale (Client e Organizzatore), implementato utilizzando tecnologie web standard quali **HTML, CSS e JavaScript**. Per migliorare l'esperienza utente e ridurre la latenza percepita, vengono impiegate chiamate asincrone tramite **AJAX**, che comunicano con il server senza richiedere il ricaricamento completo della pagina. Il cuore del sistema risiede nell'**Application Tier**, sviluppato in linguaggio Java mediante il framework **Spring Boot**. Questo livello gestisce l'autenticazione, il controllo degli accessi e l'intera logica applicativa. Infine, la persistenza delle informazioni è affidata al **Data Tier**, costituito da un database relazionale **MySQL**, che assicura l'integrità e l'affidabilità delle transazioni critiche relative a prenotazioni e pagamenti. L'architettura prevede che il Frontend non acceda mai direttamente al Database, ma interagisca esclusivamente con il Backend tramite protocollo HTTP/HTTPS, garantendo così un maggiore livello di sicurezza e disaccoppiamento tra i componenti.

3 Proposed software architecture

3.1 Overview

3.2 Subsystem decomposition

La **subsystem decomposition** del sistema BEAT mira a ottenere sottosistemi fortemente coesi, in linea con i **Design Goals** precedentemente presentati.

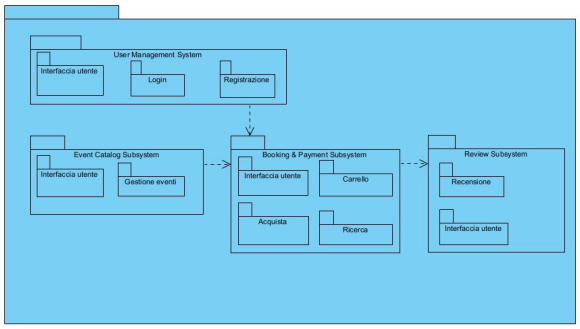

Figura 1: Package diagram

Di seguito vengono descritti i principali sottosistemi individuati per questo progetto.

3.2.1 Subsystem User Management

Questo sottosistema è responsabile della gestione degli utenti registrati alla piattaforma. Espone servizi verso il *Presentation Layer* (form di login/registrazione) e verso altri sottosistemi che necessitano di informazioni sul ruolo o sull'identità dell'utente che sta usando il sistema. Le principali responsabilità di questo sottosistema sono:

- registrazione di nuovi utenti e la gestione del loro profilo;
- autenticazione e autorizzazione (*login*, *logout*, gestione ruoli utente/amministratori);
- gestione sicura delle credenziali (hashing a 60 caratteri delle password tramite algoritmi quali `bcrypt`);
- gestione delle sessioni utente e integrazione con i servizi di sicurezza (protezione da accessi non autorizzati).

3.2.2 Subsystem Event Management

Interagisce con il *Subsystem Booking & Ticketing* per verificare la disponibilità di posti e aggiornare le capienze. Questo sottosistema si occupa di gestire l'intero ciclo di vita degli eventi:

- creazione, modifica e cancellazione di eventi da parte degli organizzatori;
- definizione di capienza, categorie, date/luoghi e tipologie di biglietti;
- pubblicazione degli eventi verso il *Presentation Layer* per la consultazione da parte degli utenti.

3.2.3 Subsystem Booking & Ticketing

Si appoggia al *Subsystem Event Management* per i dati realtivi agli eventi e al *Subsystem Payment Management* per la finalizzazione dei pagamenti. Questo sottosistema realizza il cuore funzionale del sistema BEAT:

- gestione del processo di prenotazione e acquisto dei biglietti;
- controllo della disponibilità residua e prevenzione dell'overbooking;
- generazione e associazione dei biglietti all'utente (es. codice univoco QR/ID del biglietto);
- gestione delle operazioni CRUD sulle prenotazioni.

3.2.4 Subsystem Payment Management

Il sottosistema Payment Management è quello dedicato alla gestione dei pagamenti, le operazioni che deve svolgere sono:

- integrazione con un gateway di pagamento esterno tramite protocolli sicuri (**HTTPS**);
- gestione dello stato delle transazioni (in corso, completata, fallita);
- applicazione delle politiche di sicurezza e conformità ai protocolli standard;
- supporto ai meccanismi di *rollback* in caso di errore (coerenza con le proprietà **ACID** del database).

3.2.5 Subsystem Notification

Responsabile dell'invio di notifiche all'utente, si occupa di:

- invio email di conferma prenotazione/acquisto;
- eventuali notifiche di annullamento o modifica eventi;
- integrazione con altri sottosistemi (Booking & Ticketing, Payment Management) per la generazione dei contenuti delle notifiche.

3.2.6 Subsystem Persistence & Data Management

Gli altri sottosistemi accedono ai dati esclusivamente tramite questo livello, riducendo l'accoppiamento con la tecnologia di storage. Questo sottosistema dunque incapsula tutte le operazioni di accesso ai dati:

- definizione delle classi **DAO** per le principali entità di dominio (Utente, Evento, Biglietto, Prenotazione, Pagamento);
- gestione delle connessioni al database tramite **JDBC** e DataSource;
- gestione delle transazioni con garanzia delle proprietà **ACID**;
- implementazione di meccanismi di gestione degli errori e logging a livello di persistenza.

3.2.7 Subsystem Administration & Reporting (opzionale)

Sottosistema dedicato alle funzionalità avanzate per i vari amministratori della piattaforma e permette in base a quale ruolo amministrativo si copre:

- consultazione di statistiche sulle vendite di biglietti e sulla partecipazione agli eventi;
- gestione massiva di eventi e utenti;
- esportazione di report sui dati gestiti dal sistema.

3.3 Hardware and software mapping

L'architettura proposta per BEAT segue un modello **client–server** a più livelli. Di seguito si descrive la mappatura principale tra componenti software e infrastruttura hardware.

3.3.1 Client side

Il lato client è costituito dai **browser web** degli utenti (desktop, tablet, smartphone), che accedono al sistema tramite protocollo **HTTPS**. Il browser:

- renderizza le pagine generate dal server (JSP/HTML);
- esegue codice **JavaScript** per la validazione lato client e per eventuali chiamate **AJAX**;
- interagisce con il *Presentation Layer* inviando richieste HTTP/HTTPS ai controller del Backend.

3.3.2 Application Server

Il Backend del sistema BEAT è eseguito su un **application server Apache Tomcat**, configurato per eseguire applicazioni Java basate su *Servlet*, *JSP* e, in più in generale, sul modello **MVC**. Tomcat ospita:

- i **controller**che si occupano di gestire le richieste HTTP provenienti dai vari client;
- i componenti della logica applicativa (*Service Layer*), che coordinano i casi d'uso descritti nel *Requirement Analysis Document*;
- i moduli di sicurezza (*gestione sessioni*, *filtri*, *autenticazione*, *autorizzazione*, *protezioni XSS/CSRF*);
- il livello di accesso ai dati contenuti nel database, composto dalle classi *DAO* e i connettori *JDBC*;
- Il motore per la generazione delle viste, quindi le *view*, basate su *JSP*.

L'application server può essere eseguito su una macchina fisica o virtuale (es. server Linux) con risorse adeguate (CPU multi-core, memoria RAM sufficiente al carico previsto). In ambiente di produzione è possibile prevedere una configurazione scalabile con più istanze dietro un bilanciatore di carico per soddisfare i requisiti di disponibilità e prestazioni descritti nei precedenti *Design Goals*.

3.3.3 Database Server

Il **database relazionale** che memorizza le informazioni su utenti, eventi, biglietti, prenotazioni e pagamenti è ospitato su un **database server**. Il Backend vi accede tramite **JDBC**:

- tutte le operazioni CRUD sono effettuate dal sottosistema Persistence & Data Management;
- le transazioni rispettano le proprietà **ACID**, garantendo consistenza dei dati anche in presenza di errori o concorrenza;
- sono previsti meccanismi di backup periodico e, in scenario reale, la possibilità di replica per aumentare affidabilità e disponibilità (DG.02).

3.3.4 Librerie e framework software

Dal punto di vista software, il sistema utilizza i seguenti framework e librerie:

- **Java** (versione compatibile con Spring Boot) per lo sviluppo del Backend;
- **Apache tomcat** come server applicativo per gestire le Servlet, le JSP e tutto il ciclo di vite della web app;
- **Spring Security** con algoritmo di hashing `bcrypt` per la gestione sicura delle credenziali utente;
- **JDBC** per l'accesso al database relazionale tramite pattern DAO;
- **JSP**, **HTML**, **CSS** e **JavaScript** per la realizzazione dell'interfaccia utente;
- eventuali librerie per l'invio di email (per il sottosistema Notification) e per l'integrazione con gateway di pagamento esterni.

3.4 Persistent data management

3.4.1 Mappatura Classi → Tabelle MySQL

La progettazione del database deriva direttamente dall'*Object Model* presente nel *Requirement Analysis Document*, più precisamente dal *diagramma delle classi* e dal relativo *Data Dictionary*.
L'obiettivo della mappatura è trasformare il modello concettuale orientato agli oggetti in uno schema relazionale robusto, coerente e facilmente estendibile.
Nel processo sono stati adottati i seguenti principi:

- **Ogni classe diventa una tabella relazionale**, con nome corrispondente ove possibile;
- **Ogni attributo diventa una "colonna"**, con tipo SQL appropriato sulla base delle specifiche del *Data Dictionary*;
- **Ogni entità possiede una chiave primaria autonumerata** *INT AUTO_INCREMENT*, che ne garantisce l'identificazione univoca;

- **Le relazioni 1-N sono realizzate tramite Foreign Key**, garantendo integrità referenziale;
- **Le relazioni N-M sono realizzate mediante "tabelle ponte"**, salvo nei casi in cui l'associazione stessa rappresenti una vera entità (es. *Prenotazione*);
- **Tutti i vincoli** (*NOT NULL*, *UNIQUE*, *CHECK*, *integrità referenziale*) sono stati definiti per rifletter i vincoli di dominio espressi nel *Requirement Analysis Document*.

Ecco qualche esempio di trasformazione:

- **Classe Utente → Tabella Utente**:
  id_utente INT PK AUTO_INCREMENT;
  username VARCHAR(30);
  email VARCHAR(100) UNIQUE;
  password_hash CHAR(60);
  ruolo ENUM('Cliente', 'Organizzatore', 'AdminCatalogo', 'AdminOrdini', 'AdminRuoli');

  La scelta di CHAR(60) per *password_hash* dipende dall'utilizzo di algoritmi come bcrypt, i quali producono hash di lunghezza fissa.

- **Classe Evento → Tabella Evento**:
  id_evento INT PK AUTO_INCREMENT;
  titolo VARCHAR(100);
  data_evento DATE;
  ora_evento TIME;
  luogo VARCHAR(100);
  posti_totali INT;
  posti_disponibili INT;
  prezzo DECIMAL(10,2);
  is_gratuito BOOLEAN;
  id_organizzatore INT FK → Utente(id_utente);

  posti_disponibili è mantenuto come campo separato per ottimizzare le operazioni di lettura, evitando query di conteggio che sarebbero costose.

- **Classe prenotazione → Tabella prenotazione**:
  id_prenotazione INT PK;
  id_utente INT FK;
  id_evento INT FK;
  data_acquisto DATETIME;
  codice_univoco VARCHAR(50);

- **Classe Recensione → Tabella recensione**:
  id_recensione INT PK AUTO_INCREMENT;
  id_utente INT FK;
  id_evento INT FK;
  voto TINYINT;
  contenuto TEXT;
  data_recensone DATETIME;

  Il sistema permette una sola recensione per coppia (utente, evento), quindi a livello progettuale è possibile imporre anche un vincolo UNIQUE composto.

Questa mappatura garantisce integrità referenziale, riduzione della ridondanza e piena coerenza con la struttura logica definita precedentemente nel Requirement Analysis Document.

3.4.2 Gestione delle transazioni

L'acquisto di un biglietto rappresente una delle operazioni più critiche dell'intero sistema, in quanto coinvolge la sincronizzazione di più utenti in concorrenza potenzialmente interessati allo stesso evento. Per evitare situazioni di *race conditions* (es. due utenti che acquistano l'ultimo biglietto disponibile) il sistema adotta transazioni ACID fornite da MySQL, gestite tramite JDBC.
Segue una breve illustrazione del workflow transazionale utilizzato durante l'acquisto:

- Begin transition;
- **SELECT posti_disponibili FROM Evendo WHERE id_evento = X FOR UPDATE**, FOR UPDATE blocca la riga sul database, nessun altro utente può leggerla o modificarla finchè non avviene il *COMMIT/ROLLBACK*;
- **Verifica disponibilità dei posti**: Se posti_diponibili > 0 allora continua, altrimenti ROLLBACK e notifica all'utente;
- **Inserimento della prenotazione nella tabella Prenotazione**;
- **Update posti disponibili**: UPDATE Evento; SET posti_disponibili = posti_disponibili - 1 WHERE id_evento = X;
- **COMMIT**;

Questo processo assicura un corretto funzionamento perchè il blocco della riga impedisce ad altri utenti di sottrarre posti contemporaneamente. Le proprietà ACID garantiscono coerenza del sistema anche in caso di crash. Se uno dei passi fallisce, tutto viene annullato tramite il ROLLBACK.

3.4.3 Database Design Decisions

Questa sezione risponde a tutte le domande architetturali sulla gestione dei dati.

1. **Should the data be distributed?** No, i dati non vengono distribuiti.

   Il sistema utilizza un *database relazionale centralizzato* (MySQL) ospitato sul server applicativo. La distribuzione non è necessaria per il carico previsto, aumenterebbe la complessità della sincronizzazione e introdurrebbe problematiche di consistenza (replicazione, partizionamento, etc...).

   Un singolo database centralizzato garantisce:

   - Semplicità di gestione;
   - Transazioni ACID affidabili;
   - Assenza di sincronizzazioni distribuite;
   - Performance più che adeguate per lo scenario d'uso.

2. **Should the database be extensible?** Si.

   L'intero sistema è progettato in modo normalizzato e modulare, con entità ben separate (Utente, Evento, etc...).

   Questo permette di aggiungere facilmente nuovi elementi, come:

   - Categorie evento;
   - Coupon o promozioni;
   - Pagamenti avanzati;

   L'estensibilità è una chiave della soluzione, che può evolvere senza modifiche invasive.

3. **How often is the database accessed?** Il database viene interrogato molto frequentemente, in particolare per:

   - Lettura del catalogo eventi;
   - Autenticazione degli utenti;
   - Lettura delle prenotazioni personali;
   - Aggiornamento dei posti disponibili durante l'acquisto.

   Il sistema è *read-intensive*, cioè, le query di lettura rappresentano circa l'80% del traffico, mentre le scritture sono meno frequenti.

4. **What is the expected request (query) rate? In the worst case?** Carico medio: Decine di richieste al minuto.

   Worst case: picco simultaneo su un evento molto popolare. Questo caso viene gestito grazie a:

   - Transazioni con *SELECT ... FOR UPDATE*;
   - Connection pooling;
   - Lock a livello di riga;
   - Isolamento delle transazioni;

   Il sistema non mira a supportare migliaia di accessi al secondo, è progettato per scalare facilmente se necessario.

5. **What is the size of typical and worst case requests?** Richisete tipiche:

   - Query su 10-50 righe;
   - Payload leggeri.

   Worst case:

   - Query su tutti gli eventi presenti nel catalogo;
   - Visualizzazione di centinaia di prenotazioni per un signolo evento.

   Anche nel caso peggiore, il volume di dati è pienamente gestibilie in memoria e non compromette i tempi di risposta (<2s).

6. **Do the data neet to be archived?** Si, gli eventi passati e le prenotazioni concluse possono essere gestite in due modi:

   - Essere spostati in delle "tabelle archivio";
   - essere marcati come "archiviati" tramite un flag logico.

   Le motivazioni sono:

   - Migliorare le performance sulle query più comuni;
   - Ridurre lo spazio delle tabelle operative;
   - Obblighi normativi (conservazione dei dati per x anni).

7. **Does the system design try to hide the location of the database (location transparency)?** Si, il livello di accesso ai dati (*DAO*) incapsula completamente:

   - URL del database;
   - Credenziali;
   - Gestione delle connessioni;
   - Dettagli della tecnologia usata (*MySQL/JDBC*).

   Gli strati superiori dell'applicazione (*Servlet*, *JSP*) non conoscono la posizione fisica del database. Questo migliora portabilità e manutenibilità.

8. **Is there a need for a single interface to access the data?** Si, il progetto utilizza un unico layer dedicato alla persistenza (*DAO*):

   - Tutte le operazioni di lettura e scrittura passano da li;
   - Garantisce uniformità nelle gestioni degli errori;
   - Consente l'uso centralizzato delle transazioni;
   - Previene accessi diretti o non autorizzati al database.

   Questo è fondamentale anche per implementare controlli di sicurezza e per mantenere il sistema scalabile.

9. **What is the query format?** Le query sono espresse in:

   - SQL standard;
   - Dialetto MySQL;
   - Tramite PreparedStatement JDBC.

   Il ricorso ai PreparedStatement è essenziale per:

   - Prevenire SQL Injection;
   - Favorire il caching lato database;
   - Gestire correttamente i tipi di dato;
   - Mantenere la portabilità tra motori SQL similari.

10. **Should the database be relational or object-oriented?** Il database è relazionale (*RDBMS*), scelta motivata da:

- Necessità di transazioni ACID solide;
- Modello dati altamente strutturato;
- Facilità di interrogazione tramite SQL;
- Maturità del tooling MySQL;
- Compatibilità con architettura *MVC* e con il layer *DAO*
- Semplicità di manutenzione;
- Casi d'uso oriented-to-structure coerenti con un *RDBMS*.

Un database object-oriented non aggiungerebbe benefici concreti e complicherebbe la gestione delle transazioni e dell'integrità referenziale.

3.5 Access control and security

3.5.1 Matrice degli accessi

La matrice degli accessi deriva direttamente dagli attori e dai casi d'uso del *Requirement Analysis Document* (actor diagram). Essa definisce in modo preciso quali operazioni sono consentite ai diversi ruoli applicativi.

Funzionalità	**Cliente**	**Organizzatore**	**Admin Catalogo**	**Admin Ordini**	**Admin Ruoli**
Visualizzare eventi	✓	✓	✓	✓	✓
Acquistare biglietti	✓	✗	✗	✗	✗
Annullare prenotazioni	✓	✗	✗	✓	✗
Creare eventi	✗	✓	✓	✗	✗
Modificare eventi	✗	✓	✓	✗	✗
Eliminare eventi	✗	✓	✓	✗	✗
Gestire ordini	✗	✗	✗	✓	✗
Gestire ruoli	✗	✗	✗	✗	✓

Tabella 1: Matrice degli accessi per i diversi ruoli

Il controllo degli accessi è realizzato tramite verifiche lato server (*Servlet/Filtri*) e si basa sul ruolo assegnato all'utente autenticato. Ogni endpoint applicativo verifica il ruolo prima dell'esecuzione dell'azione, garantendo il rispetto delle autorizzazioni definite nella matrice.

3.5.2 Cifratura delle password

Le password vengono memorizzate in chiaro ma *hashate* utilizzando un algoritmo sicuro con salt integrato.
Il sistema non memorizza le password in chiaro. Le credenziali vengono trasformate in *hash* sicuri. L'algoritmo scelto è **bcrypt** (output a 60 caratteri), che garantisce:

- Generazione automatica dei salt;
- Output fisso di 60 caratteri;
- Resistente a bruteforce e rainbow tables;
- Work factor configurabile per regolare la complessità.

La procedura di funzionamento è la seguente:

- L'utente inserisce la password in fase di registrazione;
- La password viene hashata tramite la funzione bcrypt;
- L'hash risultante viene salvato nel database nel campo *password_hashate*;
- In login, l'hash salvato viene confrontato con quello generato in tempo reale dalla password inserita.

3.6 Global software control

Fase	Componente Responsabile	Tecnologie & Strumenti	Descrizione dell'Azione
1. Event Trigger	**Frontend** (Client)	HTML5, CSS3, JavaScript, AJAX	L'utente interagisce con l'UI . Il browser invia una richiesta asincrona senza ricaricare la pagina.
2. Dispatching	**DispatcherServlet** (Server)	Spring Boot, Apache Tomcat	Il server riceve la richiesta HTTP. Il framework Spring mappa l'URL all'endpoint corretto.
3. Execution	**Controller & Service**	Java, Spring MVC, Spring Data JPA, MySQL	Il Controller elabora la logica, invocando i Service e i Repository (DAO) per leggere/scrivere sul Database.
4. Response	**Web Server**	JSON, HTTP/REST	Il server serializza i dati (DTO) in formato JSON e restituisce una risposta HTTP al client.

Il flusso di controllo globale del sistema BEAT segue il modello **Procedure-Driven** all'interno di un'architettura **Request-Response**. Il sistema non possiede un singolo flusso di controllo centralizzato attivo permanentemente; al contrario, il controllo risiede nel server in attesa di richieste esterne. Il flusso di esecuzione è dettato dalle interazioni dell'utente sul Frontend ed è gestito come segue:

1. **Innesco dell'Evento (Event Trigger):** Il controllo inizia nel Presentation Tier quando un utente compie un'azione (es. click su "Prenota" o "Cerca Evento"). Il browser invia una richiesta HTTP (sincrona o asincrona via AJAX) all'Application Tier.

2. **Dispatching:** Il server web (Apache Tomcat integrato in Spring Boot) intercetta la richiesta. Il componente DispatcherServlet di Spring analizza l'URL e delega il controllo al **Controller** specifico responsabile di quella risorsa.

3. **Esecuzione Procedurale:** Il Controller esegue i metodi necessari, invocando i servizi della Business Logic e, se necessario, i DAO per l'accesso ai dati (Data Tier). Durante questa fase, il flusso è sequenziale e bloccante per il thread assegnato.

4. **Risposta:** Una volta completata l'elaborazione, il controllo torna al framework che costruisce la risposta HTTP (spesso un payload JSON) e la invia al client, terminando il ciclo di controllo per quella specifica richiesta.

Gestione della Concorrenza: Sebbene il flusso logico sia sequenziale per singola richiesta, il sistema supporta la multi-utenza grazie al modello di concorrenza gestito dal container Servlet. *Per ogni richiesta in arrivo, viene allocato un thread separato dal pool del server, permettendo l'esecuzione parallela di flussi di controllo indipendenti senza richiedere una gestione esplicita dei thread nel codice applicativo.*

3.7 Boundary conditions

3.7.1 Avvio del server

Durante lo startup il sistema esegue le seguenti operazioni:

1. Inizializzazione del pool di connesioni (*DataSource*);
2. Test di connettività verso il database;
3. Caricamento configurazioni applicative;
4. Registrazione dei controller e delle servlet;
5. Caricamento dei template JSP e risorse statiche;
6. Logging dello stato di avvio.

Se il database non dovesse essere disponibile:

- Viene mostrata una pagina di manutenzione;
- Il sistema tenta una riconnesione periodicamente.

3.7.2 Spegnimento del server

Nel processi di shutdown:

1. Ogni connessione attiva nel pool viene chiusa ordinatamente;
2. Eventuali transazioin aperte vengono interrotte e rollbackate;
3. Le sessioni utente attive vengono invalidate;
4. I thread del pool vengono terminati;
5. Viene registrato un log di chiusura pulita.

3.7.3 Guasto del database

In caso di malfunzionamento del database:

1. Vengono catturate eccezioni *JDBC/SQLState*;
2. Le operazioni critiche vengono interrotte;
3. Le transazioni incompiute vengono automaticamente rollbackate da MySQL;
4. L'utente riceve una pagina di errore personalizzata;
5. Il sistema mantiene operative le sezioni statiche;
6. Viene attivato un meccanismo di riconnessione con backoff progressivo.

Queste misure garantiscono:

- Nessuna perdita di integrità dei dati;
- Nessuna prenotazione duplicata;
- Comportamento prevedibile e controllato anche in condizioni di errore.

4 Subsystem services

4.1 User Management Subsystem

Servizio (API)	**Input**	**Output**	**Descrizione**
login	*email*, password	*AuthToken*, *Role*	Verifica le credenziali e restituisce un token di sessione (JWT) con il ruolo dell'utente.
registerClient	*userData* (nome, cognome, *email*, pass)	*boolean*	Registra un nuovo utente con ruolo "Cliente".
registerOrganizer	*orgData*	*boolean*	Registra un nuovo "Organizzatore".
updateProfile	*userId*, *newData*	*UserProfile*	Aggiorna le informazioni del profilo dell'utente loggato.

4.2 Event Catalog Subsystem

Servizio (API)	**Input**	**Output**	**Descrizione**
searchEvents	keywords, dateRange, category	List<EventDTO>	Restituisce una lista di eventi filtrata in base ai criteri di ricerca.
getEventDetails (???)	eventId	EventDetailDTO	Fornisce i dettagli completi di un evento (descrizione, locandina, prezzi, posti).
createEvent	eventData, organizerId	eventId	(Solo Organizzatore) Crea e pubblica un nuovo evento nel sistema.
modifyEvent	eventId, updates	boolean	(Organizzatore, a.proditi) Modifica i dati di un evento esistente.
deleteEvent	eventId	boolean	(Organizzatore, a.proditi) Elimina evento dal sistema.

4.3 Booking & Payment Subsystem

Servizio (API)	Input	Output	Descrizione
addToCart	*userId, eventId,*	*CartState*	Aggiunge biglietti al carrello dell'utente.
processPayment	*cartId, paymentDetails*	*TransactionReceipt*	Elabora il pagamento e finalizza l'ordine.
createTicket	*bookingId*	*TicketPDF*	Genera il biglietto digitale con QR Code dopo il pagamento.
getOrderHistory	*userId*	*List<Order>*	Recupera lo storico delle prenotazioni passate dell'utente.

4.4 Review Subsystem

Servizio (API)	Input	Output	Descrizione
addToCart	*userId, eventId, qty*	*CartState*	Aggiunge biglietti al carrello dell'utente.
processPayment	*cartId, paymentDetails*	*TransactionReceipt*	Elabora il pagamento e finalizza l'ordine.
createTicket	*bookingId*	*TicketPDF*	Genera il biglietto digitale con QR Code dopo il pagamento.
getOrderHistory	*userId*	*List<Order>*	Recupera lo storico delle prenotazioni passate dell'utente.

5 Glossary

- **API (Application Programming Interface):** Interfacce REST esposte dal Backend per consentire al Frontend di comunicare con il sistema.
- **DAO (Data Access Object):** Pattern architetturale per astrarre l'accesso al database.
- **DTO (Data Transfer Object):** Oggetti utilizzati per trasportare dati tra i processi (Frontend-Backend) per ridurre il numero di chiamate e disaccoppiare il dominio dalla vista.
- **JPA (Java Persistence API):** Specifica Java per la gestione dei dati relazionali nelle applicazioni enterprise (implementata tramite Hibernate).
- **Singleton:** Pattern che garantisce l'esistenza di una sola istanza di una classe; utilizzato in Spring per i Service e i Controller.
- **Three-Tier Architecture:** Architettura a tre livelli (Presentation, Application, Data) adottata per garantire modularità e scalabilità.
- **Spring Boot**: Framework Java utilizzato per lo sviluppo del livello Application Tier, che semplifica la configurazione e il deployment.
- **Mysql:** è un sistema di gestione di database relazionali (RDBMS) open source molto diffuso, che utilizza il linguaggio SQL (Structured Query Language) per gestire, inserire e recuperare dati.