Event Ticket Booking & Entry Management System (T.B.E.M.S.)

1) Story

A regional events company, CityFest Live, runs multiple concerts, workshops, and stand-up shows every week. They currently manage:

- Event catalog and ticket pricing in spreadsheets
- Ticket availability manually (often oversold)
- Customer details in WhatsApp / notes
- Payments and refunds without a unified audit trail
- Daily revenue and top-selling event reports via manual calculations

This causes issues:

- Overselling due to incorrect ticket counts
- No reliable record of who changed ticket availability and why
- Payment mismatches and missing receipts
- No quick way to view low-availability events or daily sales summary
- Poor visibility into event-wise performance

CityFest Live wants a Java-based Ticket Booking & Entry Management System that uses a relational DB, supports role-based actions (Admin vs Sales Agent), logs all operations, and produces business reports. You are hired as the backend team to build the first working version.

2) What You Will Build

A console-based Java application (no UI frameworks required) that:

- Manages Events, Ticket Types, Ticket Inventory, Customers, Bookings, Payments, and Check-ins
- Persists data in MySQL / PostgreSQL using SQL + JDBC
- Uses layered architecture
- Has JUnit tests for core services/validations
- Uses Log4j logging for audit/debug
- Uses Git with proper branching + commit discipline

3) Mandatory Tech / Concept Coverage

Core Java (must use)

- OOP (encapsulation, abstraction, interfaces)
- Collections (List/Map usage in service layer + validations)
- Enums (roles, booking status, payment mode/status, check-in status)
- Java Time API (LocalDateTime / LocalDate for timestamps and event schedule)
- Custom exceptions + proper handling
- Input validation and defensive programming

SQL + JDBC (must use)

- Normalized schema (min 3NF where reasonable)
- CRUD using PreparedStatement
- Transactions (commit/rollback) in booking checkout + payment
- Joins & aggregations for reporting queries
- Constraints: PK, FK, UNIQUE, NOT NULL, CHECK (if supported)

Git (must use)

- Feature branches

- Meaningful commits
- PR-style workflow even if solo (merge feature branch to main)

JUnit (must use)

- Unit tests for service-layer logic
- DAO tests optional (bonus)
- Assertions verifying behavior + edge cases

Log4j (must use)

- Separate loggers for app flow + DB errors (or separate categories)
- Levels: INFO/WARN/ERROR/DEBUG
- Log file output + console output
- No System.out.println() for application flow (only menu prompts / final display outputs)

Architecture + Quality (must use)

- Layered design + clean separation
- Centralized exception strategy
- No SQL inside service classes (DAO only)
- Config via properties file
- Code quality: naming, SRP, no massive classes

4) Functional Requirements

A) User Roles

ADMIN

- Add/update/deactivate events
- Add/update ticket types (e.g., REGULAR/VIP) and base prices
- Adjust ticket inventory (+/- with reason)
- View reports

SALES_AGENT

- Register customer (optional)
- Search events and ticket availability
- Create booking + payment
- Print booking receipt (console output)
- Mark check-in at entry gate (by bookingId)

Authentication can be simple: username + role input (not full security). Role gating: show/hide menu actions based on role.

B) Event Management

Each event must have:

- eventId (auto)
- eventCode (unique)
- title
- category (Concert/Workshop/Comedy/etc.)
- venue
- eventDateTime
- active (true/false)
- createdAt, updatedAt

Operations:

- Create event
- Update event details (venue/time/title/category)
- Deactivate event (soft delete)
- Search event by title/category/venue/date range

C) Ticket Types & Pricing

Ticket Type must have:

- ticketTypeId (auto)
- code (unique) e.g., REG, VIP
- displayName
- basePrice
- active (true/false)

Operations:

- Create/update/deactivate ticket types
- Ticket price used in booking must be stored as unitPriceAtSale to preserve history

D) Ticket Inventory

Track ticket availability per event and ticket type:

- eventId FK
- ticketTypeId FK
- availableQuantity
- reorderLevel (low availability threshold)

Operations:

- Increase/decrease inventory (admin) with reason
- Low availability report: availableQuantity <= reorderLevel

Audit requirement: every admin inventory change must insert a row in ticket_adjustments with deltaQty and reason.

E) Booking + Payment (Core Transaction)

A booking includes:

- bookingId
- customer info (name + phone OR anonymous)
- eventId
- items (ticketTypeId, quantity, unitPriceAtSale)
- totalAmount
- status (CREATED, PAID, CANCELLED)
- createdAt

Rules:

- Cannot book inactive events
- Cannot book inactive ticket types
- Cannot book quantity > available tickets for that event+type
- Payment marks booking as PAID
- Must be transactional: if any step fails -> rollback everything

Design rule (to avoid inconsistent implementations):

✅ Ticket inventory is decremented only after payment is SUCCESS.

If payment fails -> booking remains CREATED and inventory remains unchanged.

F) Check-in (Entry Gate)

Check-in captures entry validation:

- checkInId
- bookingId FK (unique in v1: one check-in per booking)
- status (CHECKED_IN, REJECTED)
- checkedInAt
- notes (optional)

Rules:

- Only PAID bookings can be checked in
- Cannot check-in CANCELLED bookings
- Cannot check-in the same booking twice

G) Reporting Requirements (SQL-heavy)

Implement as menu options:

- Daily Sales Summary: date -> total bookings paid, total revenue, top 3 events by revenue
- Event Ticket Sales Report: event-wise tickets sold + revenue (date range)
- Low Availability Report: events/ticket types where availableQuantity <= reorderLevel
- Inactive Events List
- Booking Lookup: by bookingId OR by customer phone (sorted by created_at DESC)
- Check-in Summary: event -> checked-in count vs paid bookings (date)

Reports must use SQL joins/aggregations.

5) Exception Handling Rules (Non-negotiable)

Create custom exceptions such as:

- ValidationException
- EntityNotFoundException
- InsufficientStockException
- InactiveEntityException
- AlreadyCheckedInException
- DatabaseOperationException

Guidelines:

- Validate early in service layer
- Catch SQLExceptions in DAO and wrap into DatabaseOperationException
- Log errors with stack traces at ERROR level
- Show user-friendly message in console (no stack trace in UI)

6) JUnit Testing Requirements (Must Have)

You must include at least 12 meaningful tests, including these minimum required tests:

1. shouldRejectInactiveEventBooking()
2. shouldRejectInactiveTicketTypeBooking()
3. shouldRejectBookingWithZeroItems()
4. shouldRejectBookingItemWithNonPositiveQty()
5. shouldThrowInsufficientStockWhenQtyExceedsAvailable()

6. shouldComputeBookingTotalUsingUnitPriceAtSale()
7. shouldFailPaymentWhenAmountMismatch()
8. shouldNotDecrementInventoryWhenPaymentFails()
9. shouldRollbackWhenAnyStepFailsDuringCheckout() (bonus with fake DAO / controlled failure)
10. shouldRejectCheckInForUnpaidBooking()
11. shouldPreventDuplicateCheckIn()
12. shouldReturnDailySalesSummaryAggregatedCorrectly()

Clear naming example: shouldThrowInsufficientStockWhenQtyExceedsAvailable()

7) Evaluation Rubric

- Architecture & separation of concerns: 25%
- SQL schema + query quality + JDBC correctness: 20%
- Checkout transaction correctness + rollback safety: 15%
- Testing quality (JUnit): 15%
- Logging & exception handling: 10%
- Git hygiene + README quality: 10%
- Code cleanliness (naming, SRP, duplication): 5%

Starter Kit

```
None

cityfest-tbems/
├─ README.md
│  └─ HINT: Setup, DB steps, how to run, sample menu flows, and design decisions (esp. checkout timing +
inventory update).
│
├─ schema.sql
│  └─ HINT: Tables + constraints + seed data (5+ events, 3+ ticket types). Normalize and enforce FK/UNIQUE/NOT
NULL.
│
├─ pom.xml (or build.gradle)
│  └─ HINT: Dependencies: JDBC driver (MySQL/Postgres), Log4j2, JUnit5 + Surefire plugin.
│
├─ src/
│  ├─ main/
│  │  ├─ java/com/cityfest/tbems/
│  │  │  ├─ App.java
│  │  │  │  └─ HINT: Entry point. Show main menu, route to controllers, handle global errors.
│  │  │  │
│  │  │  ├─ config/
│  │  │  │  ├─ AppConfig.java
│  │  │  │  │  └─ HINT: Manual wiring of services + DAOs (simple DI). Keep object creation here.
│  │  │  │  └─ DbConfig.java
│  │  │  │     └─ HINT: Read db.properties and expose DB connection settings. No hardcoded credentials.
│  │  │  │
│  │  │  ├─ controller/
│  │  │  │  ├─ EventController.java
```

```
|   |   |   |   |   └─ HINT: Admin menu for events (create/update/deactivate/search). Calls EventService.
|   |   |   |   ├─ TicketTypeController.java
|   |   |   |   |   └─ HINT: Admin menu for ticket types (CRUD). Calls TicketTypeService.
|   |   |   |   ├─ InventoryController.java
|   |   |   |   |   └─ HINT: Admin menu to adjust inventory (+/- with reason) and view low availability.
|   |   |   |   ├─ BookingController.java
|   |   |   |   |   └─ HINT: Sales flow: customer/anonymous -> pick event -> add ticket items -> checkout ->
receipt.
|   |   |   |   ├─ CheckInController.java
|   |   |   |   |   └─ HINT: Entry gate: check-in by bookingId; validate PAID + not already checked in.
|   |   |   |   └─ ReportController.java
|   |   |   |       └─ HINT: Report menus: daily sales, event sales, low availability, check-in summary.
|   |   |   |
|   |   |   ├─ dao/
|   |   |   |   ├─ EventDao.java
|   |   |   |   |   └─ HINT: CRUD/search only. No validation.
|   |   |   |   ├─ TicketTypeDao.java
|   |   |   |   |   └─ HINT: CRUD for ticket types; list active.
|   |   |   |   ├─ TicketInventoryDao.java
|   |   |   |   |   └─ HINT: Stock reads/updates for event+type. Ensure no negative quantity.
|   |   |   |   ├─ CustomerDao.java
|   |   |   |   |   └─ HINT: Create/find/search. Phone is UNIQUE.
|   |   |   |   ├─ BookingDao.java
|   |   |   |   |   └─ HINT: Insert booking header + items + update status/total + fetch for receipt.
|   |   |   |   ├─ PaymentDao.java
|   |   |   |   |   └─ HINT: Insert payment + fetch by bookingId. One payment per booking in v1.
|   |   |   |   ├─ TicketAdjustmentDao.java
|   |   |   |   |   └─ HINT: Insert adjustment records (delta+reason). Optional query history (bonus).
|   |   |   |   ├─ CheckInDao.java
|   |   |   |   |   └─ HINT: Insert check-in (unique per booking) and fetch check-in status.
|   |   |   |   ├─ ReportDao.java
|   |   |   |   |   └─ HINT: SQL-heavy report queries (joins/aggregates). Return DTO rows.
|   |   |   |   └─ impl/
|   |   |   |       ├─ JdbcEventDao.java
|   |   |   |       ├─ JdbcTicketTypeDao.java
|   |   |   |       ├─ JdbcTicketInventoryDao.java
|   |   |   |       ├─ JdbcCustomerDao.java
|   |   |   |       ├─ JdbcBookingDao.java
|   |   |   |       ├─ JdbcPaymentDao.java
|   |   |   |       ├─ JdbcTicketAdjustmentDao.java
|   |   |   |       ├─ JdbcCheckInDao.java
|   |   |   |       └─ JdbcReportDao.java
|   |   |   |           └─ HINT: JDBC only. Catch SQLException and throw DatabaseOperationException.
|   |   |   |
|   |   |   ├─ service/
|   |   |   |   ├─ EventService.java
|   |   |   |   |   └─ HINT: Validations + uniqueness + orchestration. No SQL strings.
|   |   |   |   ├─ TicketTypeService.java
|   |   |   |   |   └─ HINT: Validate basePrice > 0; manage active/inactive ticket types.
|   |   |   |   ├─ InventoryService.java
|   |   |   |   |   └─ HINT: Admin inventory adjust updates ticket_inventory + inserts ticket_adjustments with
reason.
|   |   |   |   ├─ CustomerService.java
|   |   |   |   |   └─ HINT: Normalize phone, create/find customer, support getOrCreate by phone.
|   |   |   |   ├─ BookingService.java
|   |   |   |   |   └─ HINT: Transaction boundary: validate event/type active + availability, insert booking+items,
|   |   |   |   |          insert payment, decrement inventory on SUCCESS, update booking status, commit.
```

```
│   │   │   │   ├─ CheckInService.java
│   │   │   │   │   └─ HINT: Validate booking is PAID + not checked-in. Insert check-in record with timestamp.
│   │   │   │   └─ ReportService.java
│   │   │   │       └─ HINT: Validate date/range, call ReportDao, return DTOs/rows for printing.
│   │   │   │
│   │   │   ├─ model/
│   │   │   │   ├─ Event.java
│   │   │   │   ├─ TicketType.java
│   │   │   │   ├─ TicketInventory.java
│   │   │   │   ├─ Customer.java
│   │   │   │   ├─ Booking.java
│   │   │   │   ├─ BookingItem.java
│   │   │   │   ├─ Payment.java
│   │   │   │   ├─ CheckIn.java
│   │   │   │   └─ report/ (DTOs)
│   │   │   │       ├─ DailySalesRow.java
│   │   │   │       ├─ EventSalesRow.java
│   │   │   │       ├─ LowAvailabilityRow.java
│   │   │   │       └─ CheckInSummaryRow.java
│   │   │   │       └─ HINT: DTOs keep reporting output clean and easy to print.
│   │   │   │
│   │   │   ├─ exception/
│   │   │   │   ├─ ValidationException.java
│   │   │   │   ├─ EntityNotFoundException.java
│   │   │   │   ├─ InsufficientStockException.java
│   │   │   │   ├─ InactiveEntityException.java
│   │   │   │   ├─ AlreadyCheckedInException.java
│   │   │   │   └─ DatabaseOperationException.java
│   │   │   │
│   │   │   ├─ util/
│   │   │   │   ├─ DbConnectionFactory.java
│   │   │   │   │   └─ HINT: Single place to create Connections from db.properties.
│   │   │   │   ├─ InputUtil.java
│   │   │   │   │   └─ HINT: Safe input parsing for int/string/date/datetime.
│   │   │   │   ├─ ValidationUtil.java
│   │   │   │   │   └─ HINT: Validators for phone normalization, eventCode, qty>0, etc.
│   │   │   │   ├─ MoneyUtil.java
│   │   │   │   │   └─ HINT: BigDecimal parsing/formatting (scale=2). No doubles.
│   │   │   │   └─ DateUtil.java
│   │   │   │       └─ HINT: Parse date/time consistently (ISO format recommended).
│   │   │   │
│   │   │   └─ constants/
│   │   │       ├─ Role.java
│   │   │       ├─ BookingStatus.java
│   │   │       ├─ PaymentMode.java
│   │   │       ├─ PaymentStatus.java
│   │   │       └─ CheckInStatus.java
│   │   │       └─ HINT: Store enum values as text in DB for readability.
│   │   └─ resources/
│   │       ├─ db.properties
│   │       │   └─ HINT: Commit template only (no password). Mention in README.
│   │       └─ log4j2.xml
│   │           └─ HINT: Console + file appenders. INFO for flow, ERROR for stack traces.
│   └─ test/java/com/cityfest/tbems/service/
│       ├─ BookingServiceTest.java
│       │   └─ HINT: Test stock checks, totals, payment mismatch, and rollback behavior (fake DAO ok).
│       ├─ CheckInServiceTest.java
```

```
|     |  └─ HINT: Test paid-only check-in and duplicate check-in prevention.
|     ├─ EventServiceTest.java
|     |  └─ HINT: Test event validations and uniqueness.
|     └─ ValidationUtilTest.java
|        └─ HINT: Test phone normalization, quantity validation, code formats.
```

Minimal bootstrap classes (copy/paste)

Java

App.java

```java
package com.cityfest.tbems;

import com.cityfest.tbems.controller.BookingController;
import com.cityfest.tbems.controller.CheckInController;
import com.cityfest.tbems.controller.EventController;
import com.cityfest.tbems.controller.InventoryController;
import com.cityfest.tbems.controller.ReportController;
import com.cityfest.tbems.controller.TicketTypeController;
import com.cityfest.tbems.util.InputUtil;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class App {
    private static final Logger log = LogManager.getLogger(App.class);

    public static void main(String[] args) {
        log.info("CityFest T.B.E.M.S. started");

        EventController eventController = new EventController();
        TicketTypeController ticketTypeController = new TicketTypeController();
        InventoryController inventoryController = new InventoryController();
        BookingController bookingController = new BookingController();
        CheckInController checkInController = new CheckInController();
        ReportController reportController = new ReportController();

        while (true) {
            System.out.println("\n=== CityFest T.B.E.M.S. ===");
            System.out.println("1. Events (Admin)");
            System.out.println("2. Ticket Types (Admin)");
            System.out.println("3. Inventory (Admin)");
            System.out.println("4. Bookings");
            System.out.println("5. Check-in");
            System.out.println("6. Reports");
            System.out.println("0. Exit");

            int choice = InputUtil.readInt("Choose: ");
            switch (choice) {
                case 1 -> eventController.menu();
                case 2 -> ticketTypeController.menu();
                case 3 -> inventoryController.menu();
                case 4 -> bookingController.menu();
                case 5 -> checkInController.menu();
                case 6 -> reportController.menu();
                case 0 -> {
                    log.info("CityFest T.B.E.M.S. stopped");
                    System.out.println("Bye!");
                    return;
                }
                default -> System.out.println("Invalid option.");
            }
        }
    }
}
```

`DbConnectionFactory.java` (single place for connections)

```java
package com.cityfest.tbems.util;

import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

public final class DbConnectionFactory {
    private static final Properties props = new Properties();

    static {
        try (InputStream in =
DbConnectionFactory.class.getClassLoader().getResourceAsStream("db.properties")) {
            if (in == null) throw new IllegalStateException("db.properties not found in
resources/");
            props.load(in);
        } catch (Exception e) {
            throw new ExceptionInInitializerError("Failed to load db.properties: " +
e.getMessage());
        }
    }

    private DbConnectionFactory() {}

    public static Connection getConnection() {
        try {
            return DriverManager.getConnection(
                    props.getProperty("db.url"),
                    props.getProperty("db.username"),
                    props.getProperty("db.password")
            );
        } catch (Exception e) {
            throw new RuntimeException("DB connection failed: " + e.getMessage(), e);
        }
    }
}
```

DB Schema

```
None
-- ==========================================================
-- CityFest T.B.E.M.S. - Database Schema (MySQL 8+)
-- ==========================================================

CREATE DATABASE IF NOT EXISTS cityfest_tbems;
USE cityfest_tbems;

-- ==========================================================
-- Customers (optional: booking can be anonymous)
-- ==========================================================
CREATE TABLE customers (
  customer_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  name        VARCHAR(120) NOT NULL,
  phone       VARCHAR(20) NOT NULL UNIQUE,
  created_at  DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);

-- ==========================================================
-- Events
-- ==========================================================
CREATE TABLE events (
  event_id       BIGINT PRIMARY KEY AUTO_INCREMENT,
  event_code     VARCHAR(40) NOT NULL UNIQUE,
  title          VARCHAR(160) NOT NULL,
  category       VARCHAR(60) NOT NULL,
  venue          VARCHAR(120) NOT NULL,
  event_datetime DATETIME NOT NULL,
  active         BOOLEAN NOT NULL DEFAULT TRUE,
  created_at     DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at     DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

-- ==========================================================
-- Ticket Types (master)
-- ==========================================================
CREATE TABLE ticket_types (
  ticket_type_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  code           VARCHAR(20) NOT NULL UNIQUE, -- REG, VIP, EARLYBIRD
  display_name   VARCHAR(60) NOT NULL,
  base_price     DECIMAL(10,2) NOT NULL,
  active         BOOLEAN NOT NULL DEFAULT TRUE,
  created_at     DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  updated_at     DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

-- ==========================================================
-- Ticket Inventory (per event + type)
-- ========================================================
CREATE TABLE ticket_inventory (
  event_id            BIGINT NOT NULL,
  ticket_type_id      BIGINT NOT NULL,
  available_quantity  INT NOT NULL,
  reorder_level       INT NOT NULL DEFAULT 20,
  updated_at          DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (event_id, ticket_type_id),
  CONSTRAINT fk_inv_event FOREIGN KEY (event_id) REFERENCES events(event_id) ON DELETE RESTRICT,
  CONSTRAINT fk_inv_type  FOREIGN KEY (ticket_type_id) REFERENCES ticket_types(ticket_type_id) ON
DELETE RESTRICT
);

-- ========================================================
-- Bookings (1 booking per checkout)
-- ========================================================
CREATE TABLE bookings (
  booking_id   BIGINT PRIMARY KEY AUTO_INCREMENT,
  customer_id  BIGINT NULL,
  event_id     BIGINT NOT NULL,
  total_amount DECIMAL(12,2) NOT NULL,
  status       VARCHAR(20) NOT NULL, -- CREATED, PAID, CANCELLED
  created_at   DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT fk_booking_customer FOREIGN KEY (customer_id) REFERENCES customers(customer_id) ON
DELETE SET NULL,
  CONSTRAINT fk_booking_event    FOREIGN KEY (event_id) REFERENCES events(event_id) ON DELETE
RESTRICT
);

-- ========================================================
-- Booking Items (ticket lines)
-- ========================================================
CREATE TABLE booking_items (
  booking_item_id     BIGINT PRIMARY KEY AUTO_INCREMENT,
  booking_id          BIGINT NOT NULL,
  ticket_type_id      BIGINT NOT NULL,
  quantity            INT NOT NULL,
  unit_price_at_sale  DECIMAL(10,2) NOT NULL,
  line_total          DECIMAL(12,2) NOT NULL,
  CONSTRAINT fk_bi_booking FOREIGN KEY (booking_id) REFERENCES bookings(booking_id) ON DELETE
CASCADE,
  CONSTRAINT fk_bi_type    FOREIGN KEY (ticket_type_id) REFERENCES ticket_types(ticket_type_id) ON
DELETE RESTRICT
);

-- ========================================================
-- Payments (1 payment per booking in v1)
-- ========================================================
CREATE TABLE payments (
  payment_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  booking_id BIGINT NOT NULL UNIQUE,
  mode       VARCHAR(20) NOT NULL, -- CASH, CARD, UPI
  amount     DECIMAL(12,2) NOT NULL,
  status     VARCHAR(20) NOT NULL, -- SUCCESS, FAILED
  paid_at    DATETIME NULL,
  CONSTRAINT fk_pay_booking FOREIGN KEY (booking_id) REFERENCES bookings(booking_id) ON DELETE
CASCADE
);

-- ========================================================
-- Ticket inventory adjustments (admin audit)
```

```sql
-- =========================================================
CREATE TABLE ticket_adjustments (
  adjustment_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  event_id       BIGINT NOT NULL,
  ticket_type_id BIGINT NOT NULL,
  delta_qty      INT NOT NULL, -- + add, - reduce
  reason         VARCHAR(200) NOT NULL,
  created_at     DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  CONSTRAINT fk_adj_event FOREIGN KEY (event_id) REFERENCES events(event_id) ON DELETE RESTRICT,
  CONSTRAINT fk_adj_type  FOREIGN KEY (ticket_type_id) REFERENCES ticket_types(ticket_type_id) ON
DELETE RESTRICT
);

-- =========================================================
-- Check-ins (one per booking in v1)
-- =========================================================
CREATE TABLE check_ins (
  check_in_id  BIGINT PRIMARY KEY AUTO_INCREMENT,
  booking_id   BIGINT NOT NULL UNIQUE,
  status       VARCHAR(20) NOT NULL, -- CHECKED_IN, REJECTED
  checked_in_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  notes        VARCHAR(200) NULL,
  CONSTRAINT fk_ci_booking FOREIGN KEY (booking_id) REFERENCES bookings(booking_id) ON DELETE
CASCADE
);

-- =========================================================
-- CHECK constraints (MySQL 8+ enforces CHECK)
-- =========================================================
ALTER TABLE ticket_types
  ADD CONSTRAINT chk_tt_price CHECK (base_price > 0);

ALTER TABLE ticket_inventory
  ADD CONSTRAINT chk_inv_qty CHECK (available_quantity >= 0),
  ADD CONSTRAINT chk_inv_reorder CHECK (reorder_level >= 0);

ALTER TABLE booking_items
  ADD CONSTRAINT chk_bi_qty CHECK (quantity > 0),
  ADD CONSTRAINT chk_bi_line CHECK (line_total >= 0);

ALTER TABLE ticket_adjustments
  ADD CONSTRAINT chk_adj_delta CHECK (delta_qty <> 0);

-- =========================================================
-- Seed Data (5 events, 3 ticket types, inventory)
-- =========================================================
INSERT INTO events (event_code, title, category, venue, event_datetime, active) VALUES
('EVT-1001', 'CityFest Rock Night', 'Concert', 'Arena Hall', '2026-03-10 19:00:00', TRUE),
('EVT-1002', 'Stand-up Special: Laugh Riot', 'Comedy', 'Downtown Theatre', '2026-03-12 20:00:00',
TRUE),
('EVT-1003', 'Photography Workshop', 'Workshop', 'Studio 5', '2026-03-15 10:00:00', TRUE),
('EVT-1004', 'Tech Talk: AI for Builders', 'Talk', 'Innovation Hub', '2026-03-18 18:30:00', TRUE),
('EVT-1005', 'Acoustic Evening', 'Concert', 'Riverside Stage', '2026-03-20 19:30:00', TRUE);

INSERT INTO ticket_types (code, display_name, base_price, active) VALUES
('REG', 'Regular', 499.00, TRUE),
('VIP', 'VIP', 1299.00, TRUE),
('EB',  'Early Bird', 399.00, TRUE);

-- Inventory for each event + type
INSERT INTO ticket_inventory (event_id, ticket_type_id, available_quantity, reorder_level)
```

```
SELECT e.event_id, t.ticket_type_id,
       CASE t.code WHEN 'VIP' THEN 50 WHEN 'EB' THEN 100 ELSE 300 END AS available_quantity,
       20
FROM events e
CROSS JOIN ticket_types t;
```

`db.properties` (template)

```
db.url=jdbc:mysql://localhost:3306/cityfest

db.username=root

db.password=YOUR_PASSWORD
```

Sample menu flow (console UX)

Sample menu flow (console UX)

Main Menu

```
=== CityFest T.B.E.M.S. ===
1. Events (Admin)
2. Ticket Types (Admin)
3. Inventory (Admin)
4. Bookings
5. Check-in
6. Reports
0. Exit
```

Events Menu (Admin)

```
--- Events (Admin) ---
1. Add event
2. Update event
3. Deactivate event
4. Search events (title/category/venue/date range)
0. Back
```

Ticket Types Menu (Admin)

```
--- Ticket Types (Admin) ---
1. Add ticket type
2. Update base price
3. Deactivate ticket type
4. List active ticket types
0. Back
```

Inventory Menu (Admin)

```
--- Inventory (Admin) ---
1. Adjust ticket inventory (+/- with reason)
2. View availability for an event
3. Low availability report
0. Back
```

Bookings Menu (Sales Agent)

```
--- Bookings ---
1. Create booking
2. Add ticket item (during creation)
3. Checkout + payment
4. Find booking by ID
5. Find bookings by customer phone
6. Print booking receipt (by booking ID)
0. Back
```

Recommended flow for “Create booking”

- Ask customer phone (or choose 'anonymous') -> fetch/create customer
- Search/select event (only active events)
- Add ticket items in a loop: ticketTypeCode + qty
- Show cart preview + total
- Select payment mode + confirm
- Checkout transactionally (insert booking + items + payment; decrement inventory only on SUCCESS; update booking status; commit)

Check-in Menu (Entry Gate)

```
--- Check-in ---
1. Check-in booking by ID
2. View check-in status by booking ID
0. Back
```

Reports Menu (SQL-heavy)

```
--- Reports ---
1. Daily sales summary (date)
2. Event sales report (date range)
3. Low availability report
4. Inactive events list
5. Check-in summary (date)
0. Back
```