Reservation Branch - Introduction

This document introduces the "Reservation" branch of the BookFair Reservation Management System and summarizes the core database schema present in `schema.sql` under `docs/database/`.

Purpose

Provide a concise overview of what the Reservation branch contains, highlight missing pieces, and list concrete things to add next (DB improvements, backend APIs, auth & RBAC, tests, CI, docs, deployment).

High-level contract

- Inputs: requests from frontend or admin tools to create/cancel/view reservations; administrative edits to stalls and genres; authentication tokens.
- Outputs: reservation records (with QR codes), updated stall reservation status, audit timestamps, and events for downstream systems (notifications, invoices).
- Success criteria: reservations are created atomically, stalls can't be double-booked, roles control access, and system recovers cleanly after errors.
- Error modes: concurrent booking conflicts, invalid/missing FK references, expired/duplicated QR codes, unauthenticated requests.

Schema summary (present)

The current `schema.sql` contains the following core tables and notable items:

- `users` - IAM table with: id (BIGSERIAL), email (unique), password (BCrypt hashed sample), name, business_name, phone, role (USER/ADMIN/EMPLOYEE), created_at, updated_at. Indexes: `idx_users_email`, `idx_users_role`.
- `stalls` - inventory of stalls with: id, name (unique), size (SMALL/MEDIUM/LARGE), location, dimensions, price, is_reserved (boolean), created_at, updated_at. Indexes: `idx_stalls_name`, `idx_stalls_is_reserved`, `idx_stalls_size`.
- `literary_genres` - per-user genres: id, user_id, genre_name, created_at. Indexes: `idx_genres_user_id`, `idx_genres_name`.
- `reservations` - reservations table: id, user_id, stall_id, qr_code (unique), status (CONFIRMED/CANCELLED), reservation_date, cancelled_at. Indexes: `idx_reservations_user_id`, `idx_reservations_stall_id`, `idx_reservations_status`, `idx_reservations_qr_code`.
- A trigger+function to update `updated_at` on `users` and `stalls` is present.
- Sample data: some stalls and seeded admin/employee users.

Immediate DB improvements to add

These are the most important, low-risk DB changes to implement next in the Reservation branch:

1. Strong foreign key constraints
    - Add FKs: `reservations.user_id -> users.id`, `reservations.stall_id -> stalls.id`, `literary_genres.user_id -> users.id`.
    - Use `ON DELETE RESTRICT` for users/stalls to avoid accidental cascade deletes, or `ON DELETE SET NULL` where appropriate.
2. Atomic reservation enforcement
    - Remove `is_reserved` boolean from being a sole source of truth OR ensure it is updated transactionally with reservation creation.
    - Add a unique partial index ensuring only one active reservation per stall, e.g.:
        - Unique index on `(stall_id)` for reservations where `status = 'CONFIRMED'`.
3. Referential integrity and cascading rules
    - Decide and document cascade rules for stalls and users.
    - Avoid orphaned reservations by enforcing valid FK constraints.
4. Timestamps and audit
    - Add `created_by`, `updated_by` to key tables where needed.
    - Ensure `updated_at` trigger applies to all relevant tables (reservations, literary_genres, etc.).
5. Indexing and query optimization
    - Add composite indexes for queries the backend will run (e.g., `reservations(stall_id, status)`, `reservations(user_id, reservation_date DESC)`).
6. Data constraints and validation
    - Narrow column types (e.g., `phone` normalization), add CHECK constraints for currency/price >= 0, and limit `qr_code` length if required.
7. Migrations
    - Convert the current `schema.sql` into reversible migration scripts (Flyway / Liquibase / knex / Alembic, depending on stack).

Backend & API requirements

Implement the following endpoints/services with transactional guarantees and RBAC enforcement:

- POST /reservations - create a reservation
    - Input: user_id (from token), stall_id, optional metadata
    - Output: reservation record with `qr_code`
    - Concurrency: ensure atomic check-and-insert (SELECT ... FOR UPDATE or DB constraint + retry)
- GET /reservations/:id - view a reservation (owner/admin/employee)

- GET /users/:id/reservations - list user reservations
- DELETE /reservations/:id or POST /reservations/:id/cancel - cancel a reservation (set status and cancelled_at)

Implementation notes:

- Generate a collision-resistant `qr_code` (e.g., UUIDv4 base64 or signed token), track expiry/version if needed.
- Wrap reservation creation and stall status update in a single DB transaction.
- Emit an event (message queue or webhook) after successful reservation for notifications or downstream processing.

Auth & RBAC

- Enforce roles: USER may create their own reservations; EMPLOYEE may manage reservations for customers; ADMIN may manage stalls, users, and run maintenance tasks.
- Protect all reservation endpoints with authentication middleware and role checks.
- Consider row-level security (Postgres RLS) later for extra safety.

Testing strategy

- Unit tests for DB layer: test FK behavior, unique active-reservation constraint, trigger updates.
- Integration tests: race condition scenario where two requests attempt to reserve the same stall concurrently; assert only one succeeds.
- End-to-end tests for reservation lifecycle (create -> view -> cancel).

Frontend / UX changes (high level)

- Reserve Stall flow: select stall, preview price and dimensions, confirm payment/reservation.
- My Reservations: list with QR codes and cancellation option.
- Admin panel: create/edit stalls, view reservations, bulk operations.

CI / deployments / operational items

- Add DB migration step to CI/CD pipeline and run migrations during deploys.
- Backups & restore plan for Postgres.
- Monitoring: slow-query logs, reservation rate, error rates.

Documentation & diagrams

- Add an ER diagram and a simple sequence diagram for reservation creation.
- Publish OpenAPI / Postman collection for the new endpoints.

Actionable next steps (priority order)

1. Add the foreign keys and a unique partial index to prevent double booking (DB migration).
2. Implement reservation creation API with transactional locking and QR generation.
3. Add integration tests that simulate concurrent reservation attempts.

4. Wire up CI/CD to run migrations and tests.
5. Create ER diagram and update docs.

Edge cases to consider

- Concurrent booking attempts (use DB constraints + retries).
- Reservation cancellation race conditions.
- Stalls deactivated while reservation in-flight.
- Users deleted while they have active reservations.

If you want, I can:

- Convert these DB improvements into a migration script (specify migration tool).
- Implement the reservation creation API stub for the backend stack you use (Node/Express, Spring Boot, etc.).
- Add the concurrency integration test harness.

Tell me which of the above you'd like me to do next and your preferred backend/migration tooling.