

Final Project: “GatherUp”

A Multi-Model Event & Ticketing Platform: Relational + MongoDB + Redis core, with an elective specialty store

Overview

Teams of 3-4 students design and implement the data layer for **GatherUp**, a platform where people discover local events, buy tickets, and connect with friends who attend the same things. The scenario is deliberately chosen because no single database model handles all of its requirements well.

Every team builds a **required core** (a relational database, MongoDB, and Redis) and then adds **one elective specialty store** from a menu. Students must decide which data belongs in which model, justify those decisions, and solve the hard part: keeping the stores consistent with each other.

The scenario

GatherUp has several kinds of data with very different needs.
The first three rows are required for every team. The final row is the elective.

Need	Why it strains a single model	Model	Status
Users, orders, payments, ticket inventory	Must never oversell or double-charge; requires multi-row transactions and constraints	Relational (MySQL)	Required
Event catalog (concerts, workshops, sports, food festivals)	Each event type has different attributes (lineup, speakers, dietary options, age limits); schema changes constantly	Document (MongoDB)	Required
Sessions, seat holds, rate limits, trending events, sync messaging	Needs sub-millisecond reads, automatic expiry, and lightweight messaging	Key-value and streams (Redis)	Required
One specialty workload (analytics, durable key access at scale, semantic search, or social graph)	Each strains the core stores in a different way; see the elective menu	Time-series, key-value, vector, or graph	Choose one

Ground rule: one owner per fact. The relational store is the source of truth for orders, payments, and ticket inventory. MongoDB, Redis, and the elective store hold derived, cached, or ephemeral data. No store other than the relational database may decide whether a ticket was sold.

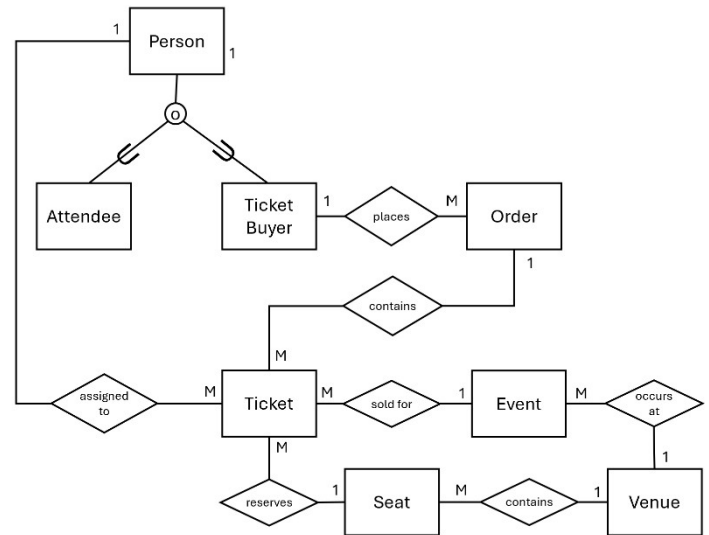
Required functionality

Students must implement the following features. Sections 1-4 are required for every team. Section 5 is the elective.

1. **Structured ticket purchasing data** permanently stored in MySQL (required)
2. **Redis-mediated seat purchasing** to deal with rapidly changing data as the masses scramble to get tickets when the sale begins (required)
3. **Storage of event details** in MongoDB to allow for greater flexibility (required)
4. **Something else in a different database** (required)
5. **Event messaging** in Redis PubSub (extra credit)

1. Structured ticket purchasing (MySQL)

- Schema for users, venues, venue seats, events (only the fields shared by every event type), orders, and tickets (one ticket per seat).
- Purchase flow implemented as a single transaction (ACID) that creates the order and inserts one ticket for each chosen seat. If any chosen seat has already been sold, the whole transaction rolls back and nothing is saved.
- Demonstrate correct behavior under concurrency: two users attempting to buy the same seat at the same moment must not both succeed, and the database itself, not just your application code, must be what refuses the second sale. Include a test that proves it.
A test using two database sessions is enough: start a purchase in the first session without committing, attempt the same seat in the second, and record what happens in each session.



Relational Database-Layer Safeguards

Problems to detect and fix with triggers

Assume the application is not the only thing that writes to the database. For each problem below, demonstrate the failure with direct SQL that bypasses the application, fix it at the database layer, then show the same statement being refused.

#	Problem	Example
1	A ticket reserves a seat that isn't in the event's venue	A ticket for a Pfleeger concert reserves seat 5 of the Robinson Conference Room
2	A sold ticket is quietly rewritten	A ticket's seat or event is edited after the sale, so the attendee is moved with no new purchase and no record
3	An event changes venue after tickets exist	A workshop is switched from Pfleeger to Robinson, and 10 tickets now point at seats in the wrong building
4	A seat changes venue after tickets exist	A Pfleeger seat is reassigned to another venue, and every ticket holding it is now invalid
5	Tickets are sold for an event that has already begun	A script or a stale web page submits a purchase for an event whose start time has passed
6	A ticket disappears without a trace	A ticket row is deleted (a cancellation, a cleanup script, a mistake) and nothing records that it existed or who held it