A Real-Time Notification System for a Community Event App

A simple mobile/web app for a local community center that hosts events like workshops, sports games, or meetings. Users (community members) can sign up for events, and the app needs to handle real-time updates so everyone stays informed instantly.

REQUIREMENTS

- Allow multiple users to browse and register for events at the same time (e.g., 50-100 users online).
- Collect registrations and send instant notifications to users' devices without delays.
- Use a basic AI feature to suggest similar events based on what users like (e.g., if someone signs up for a yoga class, suggest nearby events).
- Log all activities for admins to review later, handling things like if two users try to register for the last spot at the exact same moment.

Design Patterns

- Observer /Publish–Subscribe Pattern – for Real-time notifications. It Lets the system instantly notify users when a notification on an event is sent. Subscribers receive updates automatically. When an event state changes about an event, a notification is sent to all observers via a message broker.

Technologies used

- RabbitMQ - used to send, store, and receive messages between systems or components, ensuring reliable

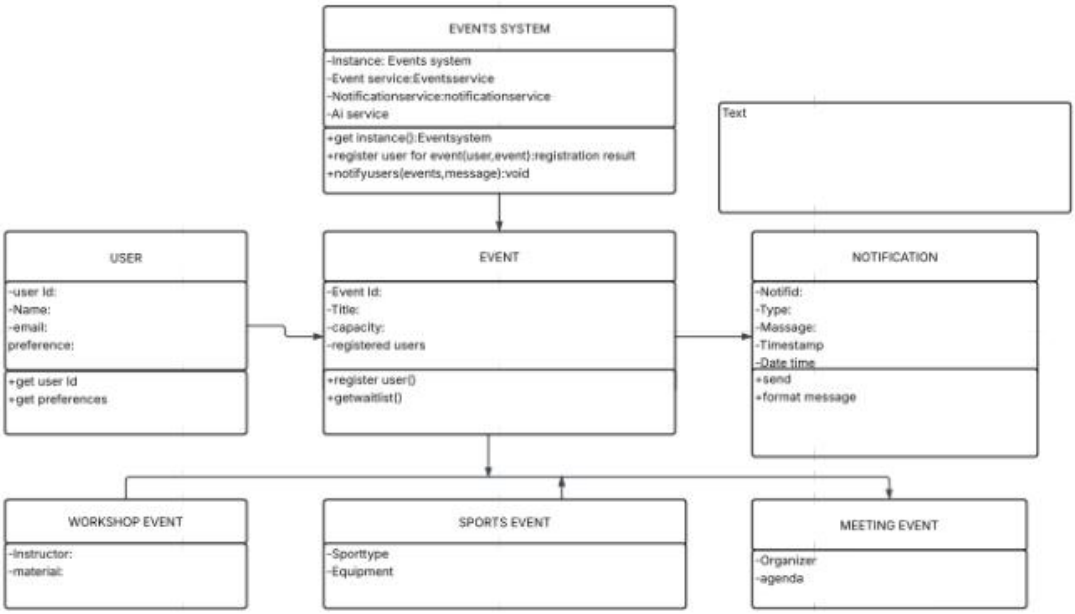
- Mongo DB – For the Database

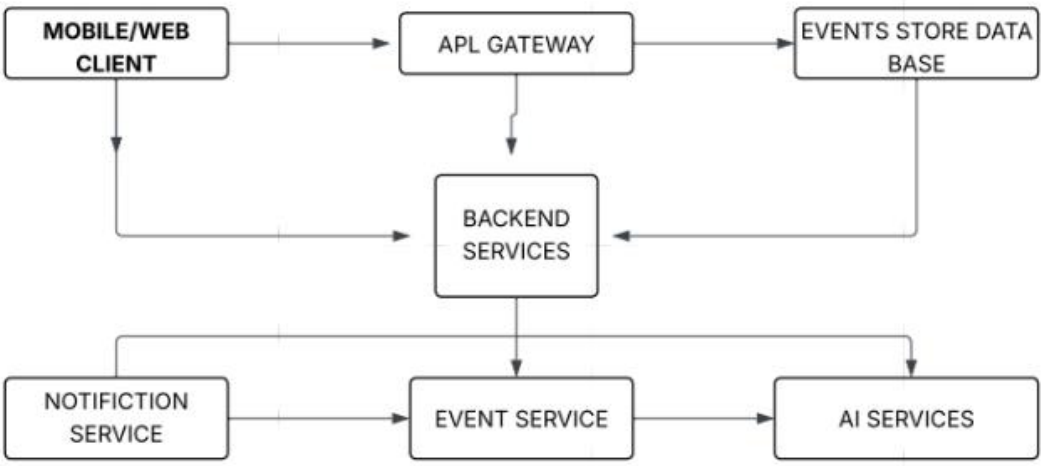
Factory Pattern – For Notification creation. It Dynamically creates different types of notifications (push, email, in-app) without hardcoding logic, which makes notification creation flexible.

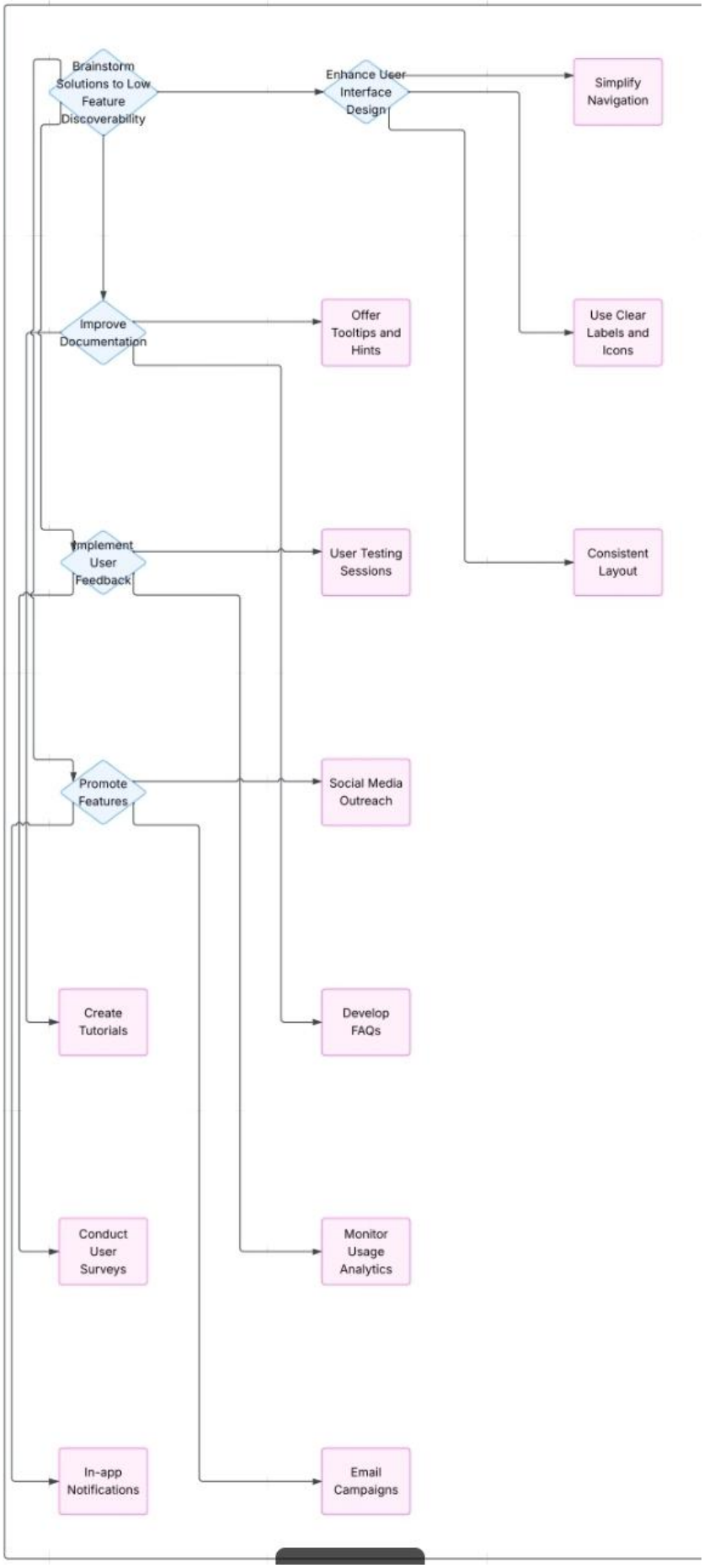
Technologies used:

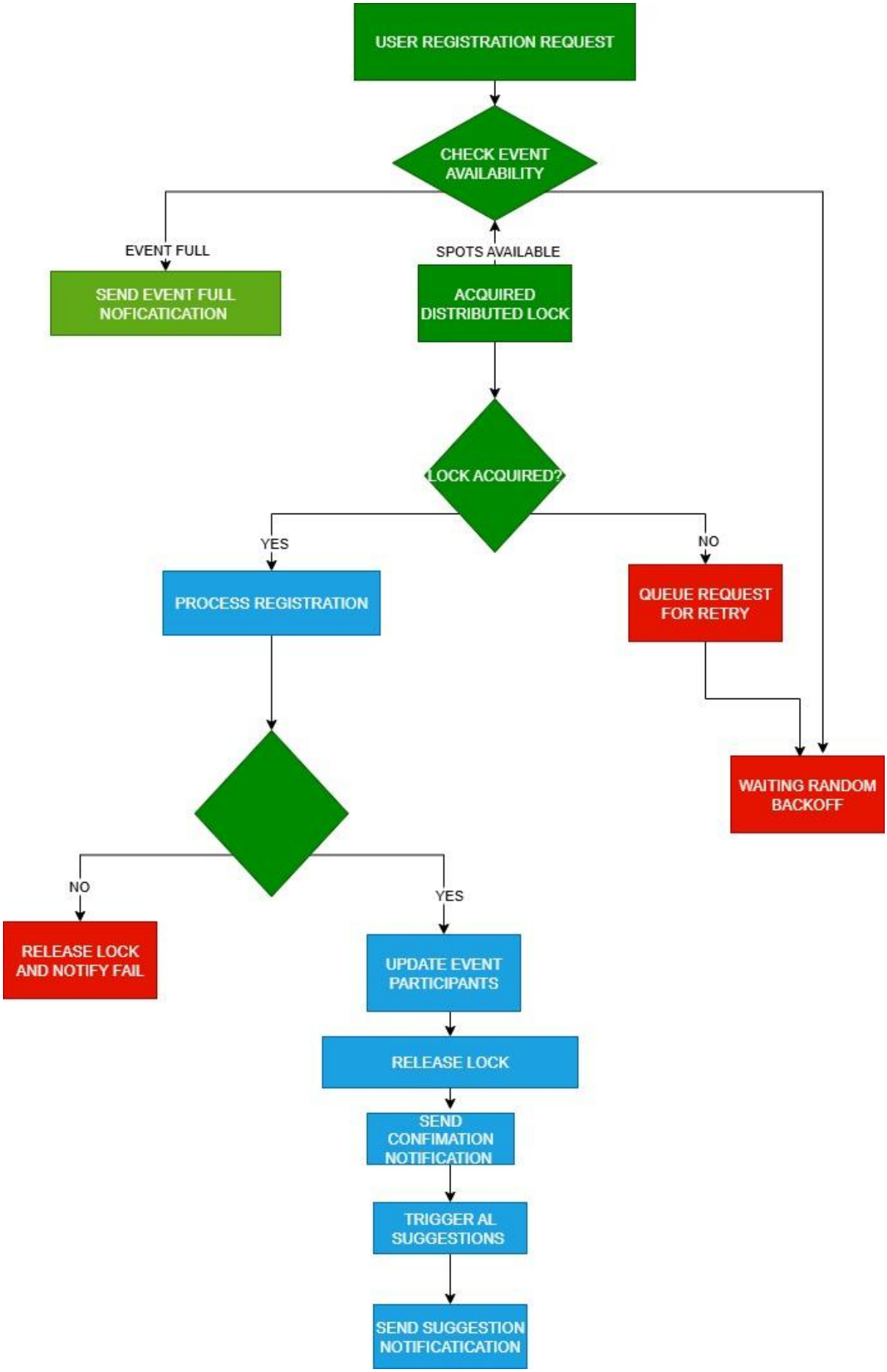
- Java (Spring Boot)

- Python (Django, Flask)


EVENTS SYSTEM
-Instance: Events system
-Event service:Eventsservice
-Notificationservice:notificationservice
-Ai service
+get instance():Eventsystem
+register user for event(user,event):registration result
+notifyusers(events,message):void
Text
USER
-user Id:
-Name:
-email:
preference:
+get user Id
+get preferences
EVENT
-Event Id:
-Title:
-capacity:
-registered users
+register user()
+getwaitlist()
NOTIFICATION
-Notifid:
-Type:
-Massage:
-Timestamp
-Date time
+send
+format message
WORKSHOP EVENT
-Instructor:
-material:
SPORTS EVENT
-Sporttype
-Equipment
MEETING EVENT
-Organizer
-agenda


MOBILE/WEB CLIENT
APL GATEWAY
EVENTS STORE DATA BASE
BACKEND SERVICES
NOTIFICTION SERVICE
EVENT SERVICE
AI SERVICES


Brainstorm Solutions to Low Feature Discoverability
Enhance User Interface Design
Simplify Navigation
Improve Documentation
Offer Tooltips and Hints
Use Clear Labels and Icons
Implement User Feedback
User Testing Sessions
Consistent Layout
Promote Features
Social Media Outreach
Create Tutorials
Develop FAQs
Conduct User Surveys
Monitor Usage Analytics
In-app Notifications
Email Campaigns


USER REGISTRATION REQUEST
CHECK EVENT AVAILABILITY
EVENT FULL
SEND EVENT FULL NOFICATICATION
SPOTS AVAILABLE
ACQUIRED DISTRIBUTED LOCK
LOCK ACQUIRED?
YES
PROCESS REGISTRATION
NO
QUEUE REQUEST FOR RETRY
WAITING RANDOM BACKOFF
NO
RELEASE LOCK AND NOTIFY FAIL
YES
UPDATE EVENT PARTICIPANTS
RELEASE LOCK
SEND CONFIMATION NOTIFICATION
TRIGGER AL SUGGESTIONS
SEND SUGGESTION NOTIFICATICATION

Concurrency Information for the Real-Time Notification System in a Community Event App

Concurrency Overview

Concurrency refers to the ability of the system to handle multiple users (e.g., 50-100) browsing and registering for events simultaneously. Without proper management, this can lead to race conditions, such as two users registering for the last available spot, resulting in overbooking or data corruption. The activity document emphasizes addressing this to meet the requirement of managing simultaneous registrations effectively.

Key Concurrency Techniques

ACID Transactions:

The registration process (checking available spots → reserving a spot → updating the count) must be wrapped in an Atomic, Consistent, Isolated, and Durable (ACID) transaction.

Ensures all steps succeed or fail together, preventing partial updates that could mislead users.

Example: If a spot check succeeds but the update fails, the transaction rolls back, avoiding overbooking.

Pessimistic Locking:

When a user attempts to register, the system places an exclusive lock on the event's spot counter in the database.

No other user can read or modify the spot count until the lock is released after the transaction completes.

Advantage: Guarantees no double-booking by serializing access to the last spot.

Tradeoff: Introduces slight delays, but acceptable for a 50-100 user system where accuracy is critical.

Optimistic Locking:

Instead of locking immediately, the system checks a version number or timestamp before committing the registration.

If the version has changed (e.g., another user registered), the transaction is rejected, and the user is prompted to retry.

Advantage: Better performance under low contention; suits high-traffic scenarios.

Tradeoff: Requires client-side retry logic, which may frustrate users if conflicts are frequent.

Queues:

Uses a First-In-First-Out (FIFO) queue (e.g., via RabbitMQ) to process registrations sequentially.

Excess users are queued and notified if spots fill up, ensuring fair handling.

Advantage: Distributes load and prevents server overload.

Tradeoff: Adds latency for queued users, requiring instant Observer-based rejection notices.

Real-World Concurrency Issues

Race Condition Example: Two users check "1 spot left" simultaneously. Without locking, both may register, exceeding capacity. Pessimistic locking or queues prevent this by ensuring only one succeeds.

Conflict Resolution: If a spot is taken mid-transaction, optimistic locking rejects the second user with a "Try again" message, while pessimistic locking blocks until the first completes.

Integration with Patterns

Singleton: A single Event Manager instance, accessed via getInstance(), centralizes spot management, reducing concurrency risks from multiple instances.

Observer: Notifies all subscribers (users/loggers) instantly after a successful registration or conflict, leveraging real-time event-driven design.