Why the Singleton Pattern is the Best Choice

ENSURES A SINGLE POINT OF CONTROL

In a real-time application—such as a community or event-driven app—many components may need to access the same resource, like a notification manager, database connection, or event handler. The Singleton Pattern ensures that only one instance of such a resource exists across the entire system. This eliminates confusion and prevents errors that could occur if multiple managers or services tried to perform the same task at once. **Example**: If multiple notification managers existed, each could send duplicate notifications or miss some users. With a Singleton, all notifications are controlled from one central instance, ensuring accuracy and consistency.

PREVENTS CONFLICTS AND INCONSISTENCIES (CONCURRENCY CONTROL)

In real-time systems, multiple users may act simultaneously — for instance, registering for an event at the same time or triggering notifications concurrently. Without a Singleton, different parts of the system might try to modify the same data simultaneously, leading to duplication, conflicting updates, or missed notifications. The Singleton Pattern prevents such conflicts because only one object instance is responsible for managing shared resources, ensuring operations are handled in a controlled and synchronized way.

IMPROVES EFFICIENCY AND RESOURCE MANAGEMENT

Creating new instances of complex classes (like a notification service connected to a server) can be resource-intensive. The Singleton Pattern avoids unnecessary duplication by reusing the same instance for all requests. This reduces memory use and processing time, which is especially valuable in real-time apps that need fast responses.

PROMOTES CENTRALIZED MANAGEMENT

With one instance managing notifications, logs, or updates, the system's behaviour is easy to manage and monitor. Any change to the Singleton instance—such as changing notification settings or update formats—applies globally across the application, ensuring consistency for every user.

SUITABLE FOR REAL-TIME FEATURES

In event-driven or real-time systems, communication happens continuously between the server and users. The Singleton Pattern supports this by maintaining a persistent, global instance that listens for events, handles updates, and dispatches notifications immediately. This design works seamlessly with the Observer Pattern if needed, where the Singleton acts as the "Subject" that manages all observers (users).

EASY INTEGRATION WITH OTHER PATTERNS

The Singleton Pattern can combine effectively with other patterns such as: **Observer Pattern:** The Singleton manages the list of observers and triggers updates. **Factory Pattern:** The Singleton may include or call a factory to create notification objects. This combination makes it ideal for apps that handle multiple users and need consistent, real-time coordination.

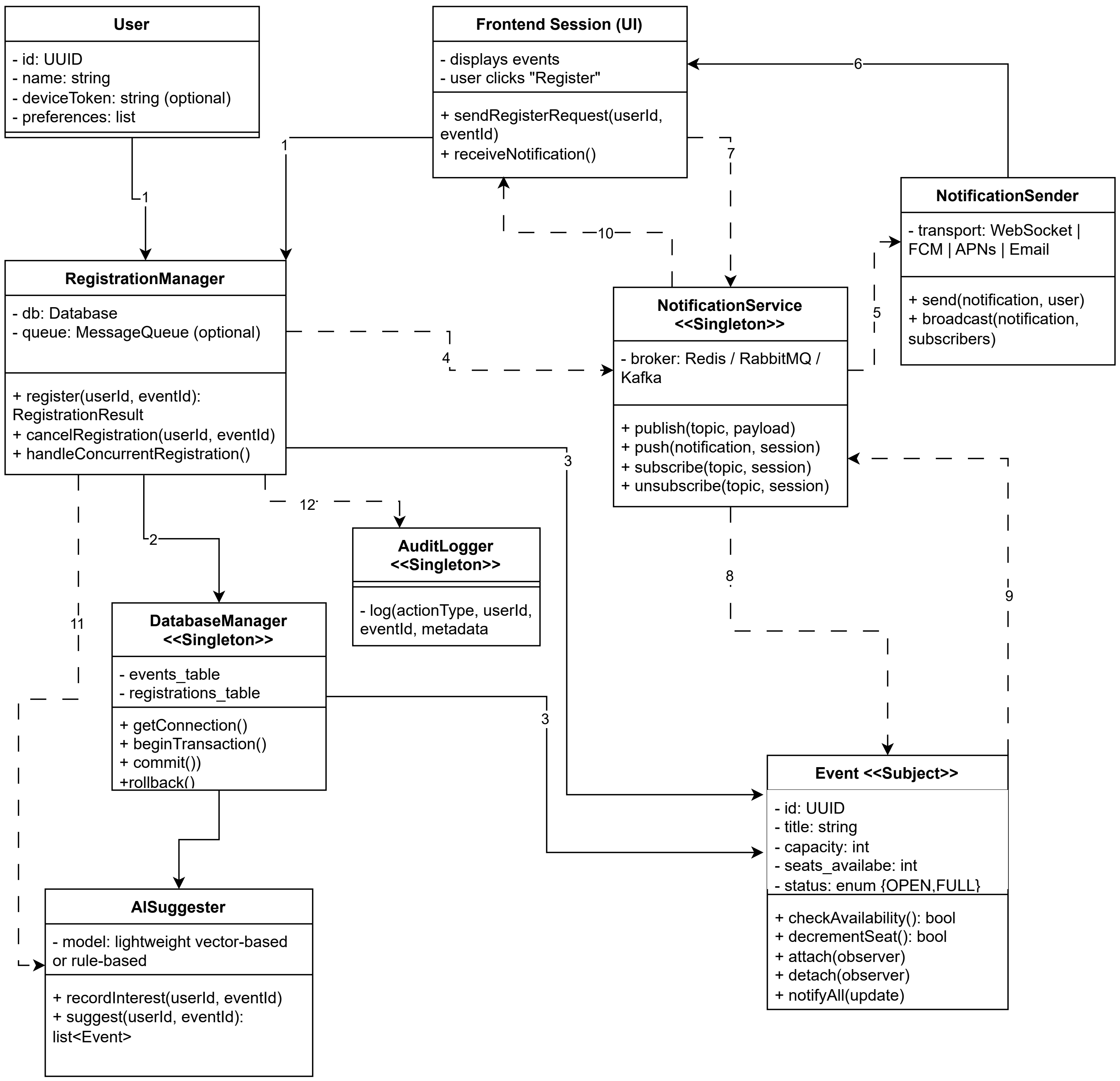
EXAMPLE SCENARIO

Imagine an app where users register for community events. If two users try to register for the last spot at the same time, the Singleton ensures only one registration is processed successfully. The single event manager instance checks the availability and locks it temporarily, preventing others from accessing it simultaneously. This prevents overbooking and maintains system reliability.

■ CONCLUSION

The Singleton Pattern is the best choice for real-time systems and notification-based applications because it: Maintains one consistent source of truth (single instance). Prevents data and notification conflicts. Reduces resource usage and improves performance. Supports real-time operations and concurrency control. Simplifies management and synchronization across multiple users. In short, Singleton ensures stability, accuracy, and efficiency in real-time systems where simultaneous actions and updates are common.

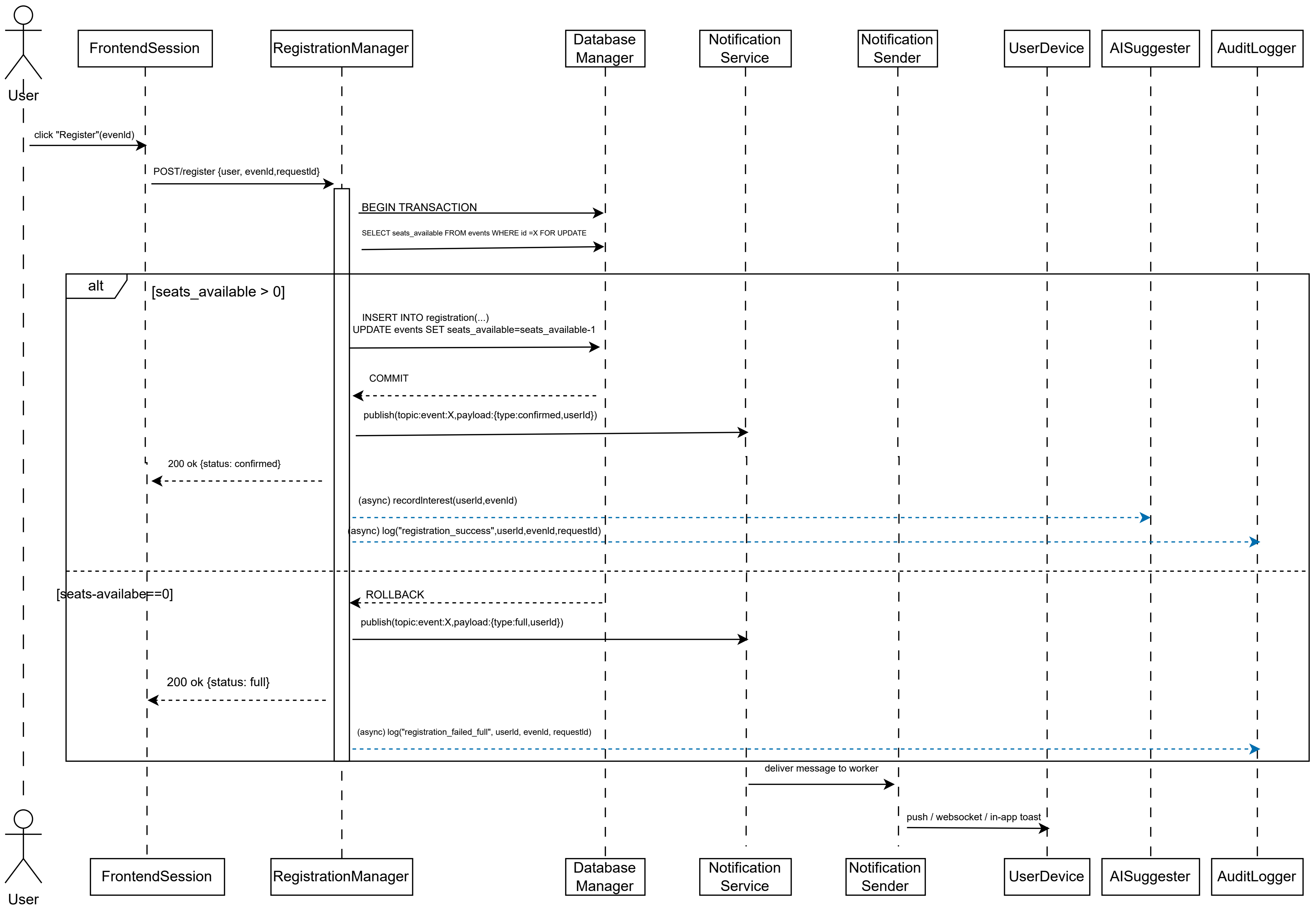
CLASS DIAGRAM


User
- id: UUID
- name: string
- deviceToken: string (optional)
- preferences: list
Frontend Session (UI)
- displays events
- user clicks "Register"
+ sendRegisterRequest(userId, eventId)
+ receiveNotification()
NotificationSender
- transport: WebSocket | FCM | APNs | Email
+ send(notification, user)
+ broadcast(notification, subscribers)
RegistrationManager
- db: Database
- queue: MessageQueue (optional)
+ register(userId, eventId): RegistrationResult
+ cancelRegistration(userId, eventId)
+ handleConcurrentRegistration()
NotificationService <<Singleton>>
- broker: Redis / RabbitMQ / Kafka
+ publish(topic, payload)
+ push(notification, session)
+ subscribe(topic, session)
+ unsubscribe(topic, session)
AuditLogger <<Singleton>>
- log(actionType, userId, eventId, metadata
DatabaseManager <<Singleton>>
- events_table
- registrations_table
+ getConnection()
+ beginTransaction()
+ commit())
+rollback()
Event <<Subject>>
- id: UUID
- title: string
- capacity: int
- seats_availabe: int
- status: enum {OPEN,FULL}
+ checkAvailability(): bool
+ decrementSeat(): bool
+ attach(observer)
+ detach(observer)
+ notifyAll(update)
AISuggester
- model: lightweight vector-based or rule-based
+ recordInterest(userId, eventId)
+ suggest(userId, eventId): list<Event>
1
1
2
3
3
4
5
6
7
8
9
10
11
12

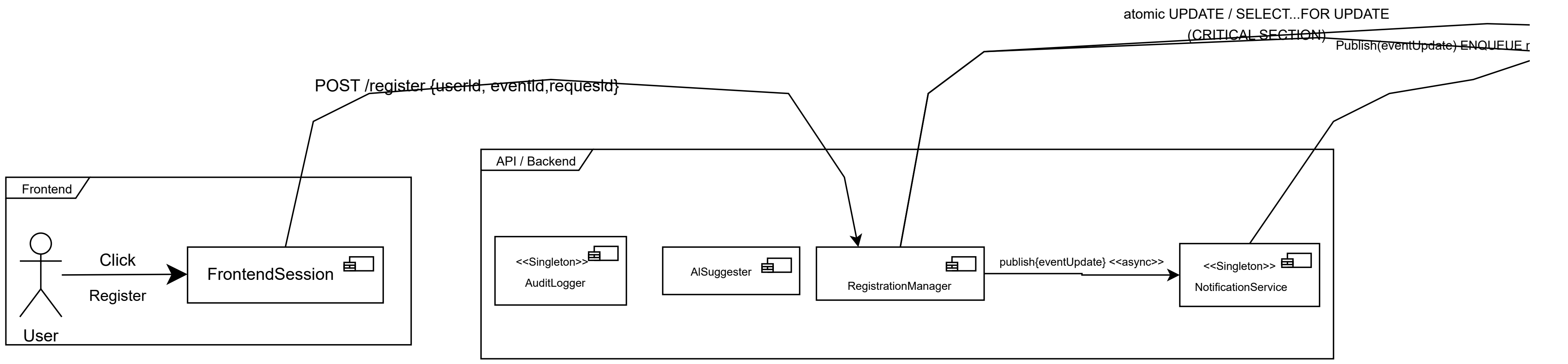
KEY

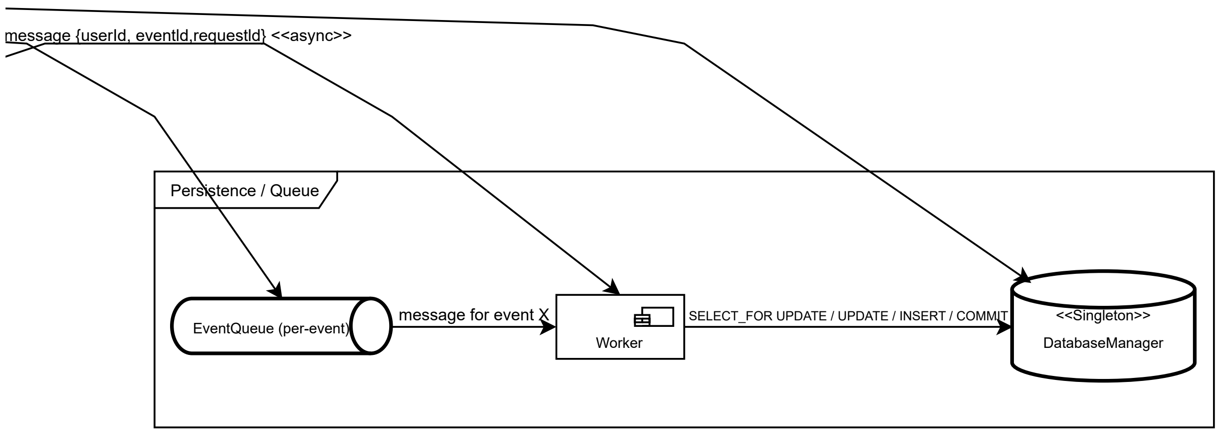
1.User/FrontendSession to RegistraonManager: (POST /register {userId, evenId, requestId})
2.RegistrationManager to DatabaseManager: BEGIN; SELECT seats_available FOR UPDATE
3.RegistrationManager to Event : (checkAvailability(); decrementSeat())
4.RegistrationManager to NotificationService: publish(topic:event:x, payload:{typer,userId})
5.NotificationService to Notification: deliver to worker
6.NotificationSender to User/FrontendSession: push / websocket / in-app toast
7.User/FrontendSession to NotificationService: subscribe(topic:event:X)
8.NotificationService to Event<<subject>>: attach(observer)
9.Event to NotificationService: notifyAll(update)
10.NotificationService to User/FrontendSession: push(notification) / update()
11.RegistrationManager to AISuggester: (async) recordInterest(userId,eventId)
12.RegistrationManager to AuditLogger: (asyn) log("registration_success",userId,eventId,requestId)

SEQUENCE DIAGRAM


User
FrontendSession
RegistrationManager
Database Manager
Notification Service
Notification Sender
UserDevice
AISuggester
AuditLogger
click "Register"(evenId)
POST/register {user, evenId,requestId}
BEGIN TRANSACTION
SELECT seats_available FROM events WHERE id =X FOR UPDATE
alt
[seats_available > 0]
INSERT INTO registration(...)
UPDATE events SET seats_available=seats_available-1
COMMIT
publish(topic:event:X,payload:{type:confirmed,userId})
200 ok {status: confirmed}
(async) recordInterest(userId,evenId)
(async) log("registration_success",userId,evenId,requestId)
[seats-availabe==0]
ROLLBACK
publish(topic:event:X,payload:{type:full,userId})
200 ok {status: full}
(async) log("registration_failed_full", userId, evenId, requestId)
deliver message to worker
push / websocket / in-app toast
User
FrontendSession
RegistrationManager
Database Manager
Notification Service
Notification Sender
UserDevice
AISuggester
AuditLogger

COMPONENT / ARCHITECTURE DIAGRAM - shows DB vs queue options and where concurrency is handled




message {userId, eventId,requestId} <<async>>
Persistence / Queue
EventQueue (per-event)
message for event X
Worker
SELECT_FOR UPDATE / UPDATE / INSERT / COMMIT
<<Singleton>>
DatabaseManager