**System Design Document – Notification System for Insyd**

**1. Introduction**

The purpose of this document is to outline the system design for a proof-of-concept (POC) notification system for **Insyd**, a social web platform for the Architecture Industry. The system is expected to handle ~100 Daily Active Users (DAU) initially but must be designed with scalability in mind to support future growth to **1M DAU**.

Notifications are central to increasing user engagement, and the system must deliver them reliably across multiple channels (in-app, email, push).

**2. Requirements**

**Functional Requirements**

1. Users should be able to receive notifications when events (e.g., like, comment, follow, post) occur.
2. Supported channels:
   * In-app notifications
   * Email notifications (future)
   * Push notifications (future)
3. Users can set **preferences** for notification channels.
4. System should store all notifications for later retrieval.

**Non-Functional Requirements**

* **Reliability**: Notifications must not be lost.
* **Latency**: Notifications should appear in near real-time (<2 seconds for in-app).
* **Scalability**: From 100 DAU to 1M DAU.
* **Extensibility**: Easy to add new event types and channels.
* **Security**: Access control, user privacy.

**3. High-Level Architecture**

Frontend App

API Layer

Event Producer

Notification DB

Worker Service

Delivery Channels  
(In-app, Email, Push)

* **Frontend App**: ReactJS client to display notifications.
* **API Layer**: Node.js/Express for event submission and notification retrieval.
* **Event Producer**: Writes events to MongoDB.
* **Worker Service**: Background processor that converts events → notifications.
* **Notification DB**: Stores users, events, and notifications.
* **Delivery Channels**: Actual delivery mechanisms.

**4. Data Model**

**User**

{

"\_id": ObjectId,

"username": "John Doe",

"email": "john@example.com",

"preferences": {

"inApp": true,

"email": false,

"push": false

}

}

**Event**

{

"\_id": ObjectId,

"type": "like | comment | follow | post",

"sourceUserId": ObjectId,

"targetUserId": ObjectId,

"data": { "text": "optional" },

"createdAt": ISODate

}

**Notification**

{

"\_id": ObjectId,

"userId": ObjectId,

"eventId": ObjectId,

"channel": "inApp | email | push",

"status": "pending | sent | failed",

"createdAt": ISODate

}

**5. API Design**

**Event API**

* POST /api/events
  + Input: { type, sourceUserId, targetUserId, data }
  + Output: { ok: true, event }

**User API**

* POST /api/users
  + Register new user
* GET /api/users/:id
  + Get user info with preferences

**Notifications API**

* GET /api/notifications/:userId
  + Returns list of notifications for user

**6. Event Flow**

1. **Trigger**: User performs an action (like/comment).
2. **Event Created**: API stores event in DB.
3. **Worker**: Consumes new events, checks target user’s preferences.
4. **Notification**: Worker creates a notification entry and dispatches it.
5. **Delivery**: Notification sent via in-app/email/push.
6. **Frontend Poll/WebSocket**: Frontend retrieves or listens for notifications.

**7. Scaling Considerations**

**Current (100 DAU)**

* Simple: MongoDB, Express, Worker running as cron/queue processor.
* In-app delivery via polling.

**Future (1M DAU)**

* **Queue System**: Use Kafka or RabbitMQ for event stream.
* **Workers**: Horizontally scale worker pods.
* **Delivery Services**: Dedicated microservices for each channel.
* **Database**: Sharding for MongoDB, or use DynamoDB.
* **Frontend**: Switch from polling → WebSocket/GraphQL subscriptions.

**8. Performance & Reliability**

* Index DB fields (userId, eventId).
* Use Redis cache for recent notifications.
* Retry mechanism for failed deliveries.
* Dead-letter queue for undeliverable notifications.

**9. Observability**

* Logging: Request logs (morgan, Winston).
* Metrics: API latency, event throughput, delivery success.
* Alerts: On error rate spikes.

**10. Failure Modes**

* **DB down**: Queue holds events until recovery.
* **Email server down**: Retry with exponential backoff.
* **Worker crash**: Restart via container orchestrator.

**11. Deployment Plan**

* **Backend**: Deploy Node.js/Express on Render.
* **DB**: MongoDB Atlas cluster.
* **Frontend**: React app deployed on Netlify.
* **Worker**: Run as background process/container.

**12. Example .http File for Testing**

### Create User

POST http://localhost:5000/api/users

Content-Type: application/json

{

"username": "John Doe",

"email": "john@example.com",

"preferences": { "inApp": true }

}

### Create Event

POST http://localhost:5000/api/events

Content-Type: application/json

{

"type": "like",

"sourceUserId": "111",

"targetUserId": "222",

"data": { "text": "Nice post!" }

}

### Get Notifications

GET http://localhost:5000/api/notifications/222

**13. Conclusion**

This system design ensures a **scalable, reliable, and extensible notification system** for Insyd. The initial implementation with ~100 DAU can run on a simple stack (Express + MongoDB + background worker). As the platform grows, the design naturally evolves into a distributed event-driven architecture with message queues, sharded databases, and horizontally scalable workers. With these strategies, the notification system will deliver **low-latency, high-throughput, and fault-tolerant** notifications to up to **1M DAU** while remaining maintainable and future-proof.