Contents

[**Just Messenger Documentation** 1](#_Toc193822393)

[**Overview** 1](#_Toc193822394)

[**Key Features** 1](#_Toc193822395)

[**Microservices Architecture Overview** 2](#_Toc193822396)

[**Core Services** 2](#_Toc193822397)

[**Simplified Data Flow Diagram (TO DO make better one)** 2](#_Toc193822398)

[**Development Phases for the Microservices MVP** 3](#_Toc193822399)

[**Phase 1: Requirements & Planning** 3](#_Toc193822400)

[**Phase 2: Architecture & Tech Stack Selection** 3](#_Toc193822401)

[**Phase 3: Service Development & Integration** 4](#_Toc193822402)

[**Phase 4: Testing, CI/CD, and Monitoring** 4](#_Toc193822403)

[**Phase 5: Iteration & Scaling** 4](#_Toc193822404)

**Just Messenger Documentation**

**Overview**

Just Messenger (working title) is a one-to-one messaging app that emphasizes simplicity, low-latency delivery, and features for real-time communication. The application is designed to support secure, efficient, and scalable messaging while offering additional functionality such as chat search, notifications, and multi-device support.

**Key Features**

* **One-to-One Chat:** Real-time messaging using persistent connections (e.g., SignalR or WebSockets).
* **Delivered & Read Indicators:** Basic status updates for each message.
* **Online Presence:** Display user online/offline status.
* **Basic Notifications:** Simple push notifications for new messages (email notifications can be added later).
* **Multiple Device Support (Can be added later)** Allow a single account to log in on multiple devices, with basic message synchronization.

**Microservices Architecture Overview**

**Core Services**

1. **Authentication Service**
   * **Responsibilities:**  
     • User registration and login using JWT-based authentication.  
     • Manage user credentials in a dedicated database (e.g., PostgreSQL or MySQL).
2. **Messaging Service**
   * **Responsibilities:**  
     • Handle real-time messaging via SignalR or WebSockets.  
     • Manage delivered/read indicators and message synchronization (maybe across multiple devices).  
     • Store chat history in a NoSQL or key-value store for low-latency access.
3. **Notification Service**
   * **Responsibilities:**  
     • Send push notifications (e.g., via FCM or Web Push) for new messages.  
     • Trigger email notifications for unread messages (can be added later).
4. **API Gateway** **(Optional for MVP)**
   * **Responsibilities:**  
     • Route incoming HTTP requests to the appropriate microservice.  
     • Simplify client interaction by providing a unified API endpoint.

## **Simplified Data Flow Diagram (TO DO make better one)**

┌─────────────────────┐

│ User Device │

│ (Web/Mobile Client) │

└─────────┬───────────┘

│ 1) Login/Message Request (HTTP/WebSocket)

▼

┌─────────────────────┐

│ API Gateway │

└─────────┬───────────┘

│

┌──────────┼─────────────┐

▼ ▼

┌─────────────┐ ┌───────────────── ┐

│ Auth Service│ │ Messaging Service│

└─────────────┘ └─────── ┬─────────┘

│ │

│ 2) Authenticate User │ 3) Process and route messages

▼ │

┌─────────────┐ │

│ User DB │ │

└─────────────┘ │

│

▼

┌─────────────────────┐

│ Message Store/Cache│

└─────────┬───────────┘

│

│

┌─────────▼───────────┐

│ Notification Service│

└─────────┬───────────┘

│

│ 4) Push Notifications

▼

┌─────────────────────┐

│ User Device │

│ (Real-Time Updates) │

└─────────────────────┘

**Flow Explanation:**

1. The **User Device** sends a login or message request via HTTP or WebSocket.
2. The **API Gateway** routes the request to the appropriate microservice.
3. The **Authentication Service** validates credentials against the **User DB**.
4. The **Messaging Service** processes chat messages, updates delivery/read indicators, and stores/retrieves messages from the **Message Store**.
5. The **Notification Service** sends push notifications for new messages or alerts.
6. The **User Device** receives real-time updates through persistent connections.

## **Development Phases for the Microservices MVP**

### **Phase 1: Requirements & Planning**

* **Define Core Features:** Focus on one-to-one chat, basic status indicators, online presence, and notifications.

### **Phase 2: Architecture & Tech Stack Selection**

* **Microservices Approach:**
  + Separate services for authentication, messaging, and notifications.
  + Maybe use an API Gateway to provide a single point of entry.
* **Possible Tech Stack:**
  + **Backend:** ASP.NET Core (or Java-based framework).
  + **Real-Time Communication:** SignalR or WebSockets.
  + **Database:** Relational DB for user data and NoSQL/key-value store for messages.
  + **Containerization & Orchestration:** Docker and Kubernetes for managing services.
  + **CI/CD:** GitHub Actions or Azure DevOps.
  + **Frontend:** **TO DO**

### **Phase 3: Service Development & Integration**

* **Develop Each Microservice Independently:**
  + **Authentication Service:** Implement registration, login, and JWT generation.
  + **Messaging Service:** Set up a real-time messaging API with delivered/read indicators and basic encryption.
  + **Notification Service:** Implement push notification logic (start with mobile/web push).
* **Inter-Service Communication:**
  + Use REST APIs for synchronous calls between services.
  + Use a message broker (e.g., RabbitMQ) if asynchronous processing is needed.
* **Develop a client?**
  + Mobile or Web client

### **Phase 4: Testing, CI/CD, and Monitoring**

* **Testing:**
  + Unit tests for each service using xUnit or NUnit.
  + **Maybe** Integration tests for service interactions (e.g., using Postman).
* **CI/CD:**
  + Deploy services using Docker and Kubernetes manifests.
  + Maybe Automate builds and tests with GitHub Actions or Azure DevOps.
* **Monitoring:**
  + Use Prometheus and Grafana for performance monitoring and logging.

### **Phase 5: Iteration & Scaling**

* **Iterative Improvements:**
  + Gather user feedback and refine performance and security.
  + Optimize real-time messaging and status updates.
* **Scaling (Optional):**
  + Implement Kubernetes auto-scaling policies and improve load balancing with API gateways (e.g., NGINX).

For reference:

https://bytebytego.com/courses/system-design-interview/design-a-chat-system