Moneybase Chat Assignment and Agent Handling System

Overview

The system manages customer chat requests by assigning them automatically to available agents and delivering them in real time to the agent's interface.
It follows an **message-driven microservices architecture – each microservice designed as clean architecture** , where each core function runs as an independent service and uses **RabbitMQ** for communication between microservices and **SignalR** for real time updates
The system ensures scalability, resilience, and fast response times.

- For demo purposes, I used a single shared database as the source of truth, with the Chat API connecting to it via EF Core.
- In a real microservices environment, each service would own its own database; this setup was chosen to simplify the environment and focus on the core event-driven logic, including queues, background workers, and the assignment flow.
- Other services retrieve the necessary data from the Chat API through internal HTTP calls to its controllers.
- The Chat API microservice is responsible for managing the database and migrations.

Objectives

- Automate chat assignment based on agent availability and capacity.
- Use asynchronous processing through message queues for efficiency.
- Support independent scaling and deployment of each service.
- Enable real-time updates for agents via SignalR or WebSocket.

System Flow

1. Chat Creation Microservice

- A user initiates a chat through the system.
- The **Chat API** service saves the chat in the database and publishes a message to the main RabbitMQ queue.
- This design ensures fast response time to users, as chat assignment happens asynchronously.

---

2. Chat Assignment Microservice

- The **ChatAssignmentWorker** (a hosted service running inside its microservice) consumes chat messages from the main queue.
- It checks current capacity and assigns chats to available agents using a round-robin order (Junior → Mid → Senior → Team Lead).
- If the system is full, chats are either queued for overflow or marked as refused.
- Once assigned, the service updates the database and publishes a message to the **Agent Exchange**, routed to the specific agent's queue using RabbitMQ Topic exchange.
- The **QueueMonitorHostedService** runs continuously (check every 1s), Detects inactive chats (no polls for > 3 seconds). Marks those sessions as Inactive. Frees the agent's slot so they can take a new chat.

---

3. Agent Handling Microservice

- An agent handler microservice listens to their dedicated RabbitMQ queue (for example, agent.{AgentId}.queue).
- The **AgentHandlerService** (a hosted service running inside the Agent microservice) consumes messages from this queue.
- Upon receiving a message, it updates the database and notifies the assigned agent's front-end in real time through SignalR or WebSocket.
- The agent can then immediately begin communication with the user.

Tools & Technologies

- **.NET Core 8.0.21 SDK – for building and running the application**
- **Visual Studio 2022 – integrated development environment**
- **SQL Server & SSMS – database management and query tools**
- **Entity Framework Core – ORM for database access and migrations**
- **RabbitMQ – message broker for asynchronous communication**
- **SignalR – real-time messaging and WebSocket support**

- **HttpClient – for internal API calls between micro services**
- **Serilog – structured logging framework**

Message Flow

1. **User → Chat API**
   The user starts a chat; it is stored and published to the main RabbitMQ queue.
2. **Chat API → RabbitMQ**
   RabbitMQ decouples the creation process from assignment to keep the system responsive.
3. **RabbitMQ → Chat Assignment Microservice**
   The ChatAssignmentWorker consumes the chat message, assigns an agent, and publishes to the agent's specific queue.
4. **RabbitMQ → Agent Handling Microservice**
   The AgentHandlerService consumes the message, updates the database, and notifies the assigned agent.
5. **Agent → SignalR**
   The agent receives the chat instantly in their front-end application.

Scalability and Design Notes

- **Decoupled Microservices:** Each major function (Chat API, Assignment, Agent Handling) operates as an independent microservice.
- **Hosted Services:** Each microservice runs its core logic through a background hosted service (e.g., ChatAssignmentWorker or AgentHandlerService).
- **Asynchronous Processing:** RabbitMQ ensures non-blocking communication between microservices.
- **Real-Time Updates:** Agents receive new chat notifications instantly through SignalR.
- **Scalable Design:** Each service can be scaled individually depending on load (e.g., more ChatAssignmentWorker instances).

Simplified Flow Diagram

User → Chat API → Saves chat in DB → RabbitMQ → Chat Assignment Microservice → Updates DB → Agent Handling Microservice → Updates DB → SignalR → Agent UI

Also, front end can poll every I second by called **poll API in Chat micro service.**

Key Benefits

- Improved performance through asynchronous message-driven processing.
- Clear separation of responsibilities across microservices.
- Independent scaling for high availability and better load management.
- Real-time communication for a seamless agent experience.
- Reliable message handling with RabbitMQ ensuring durability and delivery guarantees.

Setup Instructions for the Task

1. Database Setup

- Create a new database (SQL Server)
- Apply the necessary **migrations**:

  ```
  dotnet ef database update
  ```

- Verify that tables such as Agents, Chats are created.
- Ensure connection strings in appsettings.json or environment variables are correctly configured.

2. RabbitMQ Setup

- Install RabbitMQ locally or use a Docker container:

  ```
  docker run -d --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management
  ```

- Configure the connection in appsettings.json:

```
"RabbitMQ": {
    "Host": "localhost",
     "Username": "guest",
     "Password": "guest"
 }
```

3. SignalR Configuration

- **Hub URL: https://localhost:5000/chatHub**
- **Hub Mapping in Code:**

```
app.MapHub<ChatHub>("/chatHub");
```

- **Used for real-time messaging between clients and the server.**