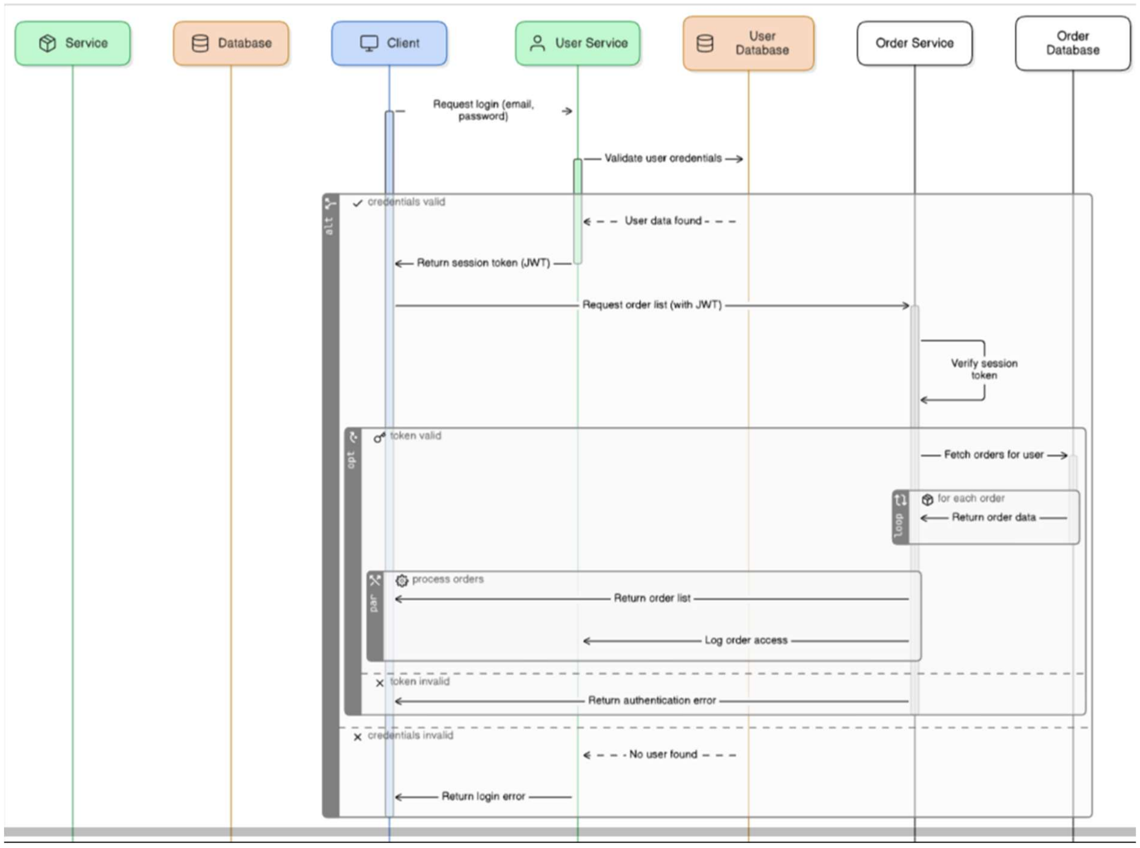
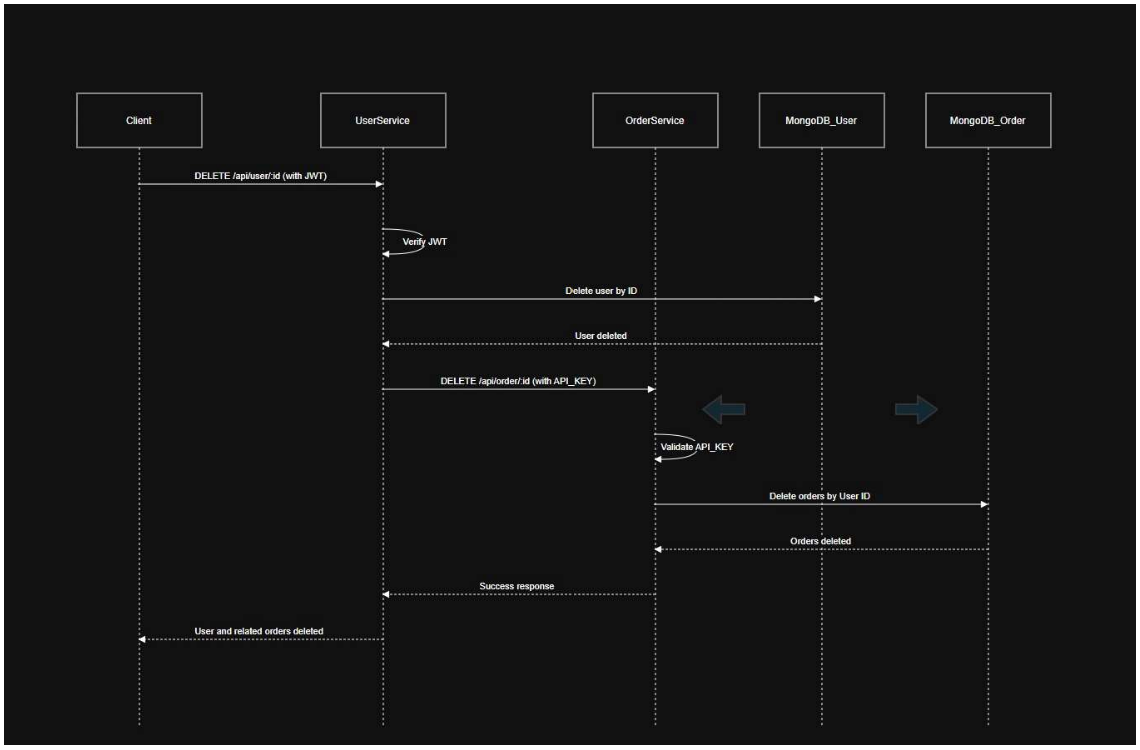
Architecture Decision

We chose a microservices architecture using two separate services—**User Service** and **Order Service**—to promote modularity, scalability, and separation of concerns. Each service owns its own data and operates independently, communicating via RESTful APIs. This approach enables parallel development, independent deployment, and better fault isolation. HTTP-based communication was selected for its simplicity and compatibility with tools like Postman and frontend clients.

Sequence Diagram





2. Failure Scenarios

We anticipated several failure points and handled them as follows:

- **Invalid User ID**: When creating or fetching orders, we validate the userId using Mongoose's ObjectId validation. Invalid requests return a 400 Bad Request response with clear messages.
- **User Not Found**: The Order Service first confirms the user's existence by querying the User Service before proceeding with order creation. If the user doesn't exist, it returns a 404 Not Found.
- **Service Downtime**: If the User Service is unreachable, the Order Service returns a 503 Service Unavailable and logs the issue for monitoring.
- **Database Errors**: Try-catch blocks are implemented to handle database-related exceptions, returning appropriate error codes (500 Internal Server Error) with safe messaging.
- I encountered issues with failed or unauthorized API calls between services. To fix this, I used JWT authentication for secure user requests—tokens are issued at login and verified on each request. For internal actions like user deletion notifications, I added an

internal API key sent via the x-api-key header. The receiving service validates this key to ensure only trusted services can perform sensitive operations.

3. Scaling Considerations

For scaling to support thousands or millions of users:

- **Horizontal Scaling**: Services are stateless and can be scaled horizontally behind a load balancer.
- **Caching**: Frequently requested data (like user info) can be cached using Redis to reduce inter-service calls.
- **Rate Limiting and Queues**: To handle spikes, rate limiting can prevent abuse, and asynchronous message queues can be introduced for background tasks.

4. Alternative Approaches:

In the future, I can replace direct REST communication between services with a message queue like RabbitMQ or Kafka. Instead of making HTTP calls (e.g., notifying the Order Service on user deletion), the User Service can publish an event (like UserDeleted) to a queue. The Order Service can listen to this