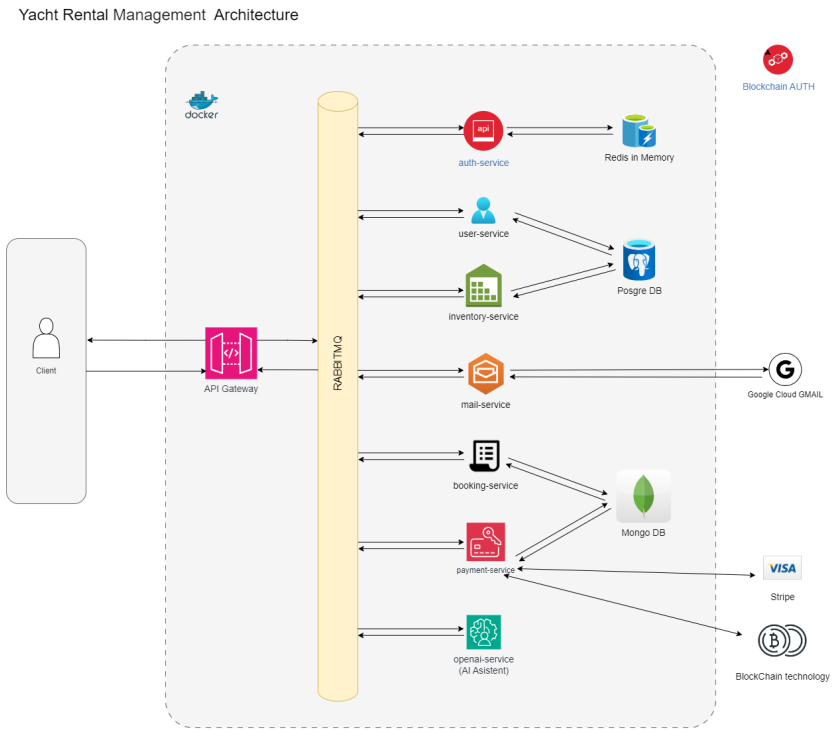

Figure 1: image

Yacht Rental Management System

Project Overview

The Yacht Rental Management architecture tackles the issues of scalability, modularity, and performance by distributing functionality across multiple specialized services. Each microservice independently manages aspects like user authentication, booking, and payments. This distributed setup improves both operational flexibility and fault isolation, ensuring that failures in one service don't affect the entire system.

Microservices Architecture

A microservices approach enables independent deployment, development, and scaling for each service. By segregating concerns into focused services, the project achieves:

- **Isolation and Independence**: Each service can be developed, updated, or scaled without impacting others.

- **Scalability**: Allows fine-grained scaling based on load (e.g., scaling booking services independently during high demand).
- **Fault Tolerance**: If one service fails (e.g., payment service), it doesn't bring down the entire application.

RabbitMQ

RabbitMQ is a message broker that facilitates reliable, asynchronous communication between microservices. It is particularly useful for operations that don't require immediate responses, such as:

- **Booking Confirmation**: The booking service can confirm reservations asynchronously.
- **Notification Dispatching**: Notifications (like booking confirmations or payment receipts) are sent without blocking other processes.

Why RabbitMQ?

- **Resilience**: Provides a fault-tolerant system where messages are reliably queued, processed in order, and retried if necessary.
- **Complex Routing**: Supports complex routing rules and retry mechanisms, vital for coordinating tasks across services.

Database Choices: PostgreSQL & MongoDB

The architecture employs a dual-database approach to address different data needs across services:

PostgreSQL with Prisma for Users and Yacht

- **Relational Integrity**: SQL is optimal for managing structured data with complex relationships, such as users and yachts.
- **ORM with Prisma**: Prisma's type-safe API improves development speed and code reliability, simplifying database operations and minimizing boilerplate code.
- **Data Consistency**: ACID compliance ensures data consistency, crucial for user profiles and booking transactions.

MongoDB for Payments

- **Document-based Flexibility**: MongoDB's schema-less nature is ideal for payment data, which can vary by transaction type.
- **High Throughput**: Suitable for handling a large volume of payment records, as MongoDB's design supports fast reads and writes.
- **Scalability**: Easily scales horizontally, fitting applications with high data ingestion requirements.

Frameworks and ORM Choices

- **NestJS**: The backbone of the microservices architecture, NestJS's modular structure and support for TypeScript align well with microservices development.

Prisma and TypeORM

- **Prisma for PostgreSQL Services**: Its developer-friendly API is useful for building high-performance queries and data modeling. It enhances type safety and ensures cleaner code.
- **TypeORM for MongoDB and Other Services**: TypeORM is flexible and integrates well with various SQL and NoSQL databases, making it valuable for managing MongoDB documents.

Security and Authentication

To safeguard the platform, OAuth2 with JWT (JSON Web Tokens) is employed for secure, stateless user authentication. Other key security measures include:

- **Redis Caching**: Redis caches JWT tokens and sessions, reducing the need for repeated authentication checks and improving response times.
- **bcrypt for Password Hashing**: bcrypt securely hashes passwords, providing resistance against brute-force attacks by increasing computational difficulty.
- **API Gateway Enforcement**: The API Gateway filters and authorizes requests before routing them to microservices, reducing exposure to unauthorized requests.

Advantages

- **JWT's Stateless Nature**: Enables lightweight, decentralized authentication.
- **Redis Performance**: Enhances performance by caching token data, reducing database calls for frequent logins.

Deployment and Scalability

Each microservice is containerized with Docker, facilitating straightforward deployment, replication, and scaling. Docker Compose is used to orchestrate multi-container setups, streamlining development and testing. In production, orchestration tools like Kubernetes can be leveraged to automate container scaling and resilience.

Architecture & Technologies

- **API Gateway**: The entry point for all services, consolidating multiple endpoints and managing traffic.

- **Microservices**:
  - ☒ **Authentication Service**: Manages user authentication and authorization.
  - ☒ **User Management Service**: Handles user data and profiles.
  - ☒ **Booking Service**: Manages reservations, cancellations, and availability.
  - ☒ **Payment Processing Service**: Integrates with payment gateways for transactions.
  - ☒ **Inventory Service**: Manages yacht listings and availability using PostgreSQL with Prisma.
  - ☐ **OpenAI Service**: Integrates ChatGPT to assist users with queries in natural language (Not started yet).
  - ☐ **Blockchain authentication**: Traditional authentication systems are centralized and vulnerable to breaches, while blockchain authentication is decentralized, enhancing security and data integrity. It uses cryptographic techniques to prevent unauthorized access and ensures immutability, making tampering difficult. This transparency fosters accountability and reduces fraud, making blockchain a superior option for secure authentication compared to conventional methods (Not started yet).
  - ☐ **Cryptocurrency payments**: As digital currencies gain popularity, integrating them can improve competitiveness and adaptability in the market (Not started yet).
- **Asynchronous Communication**: Utilizes RabbitMQ for reliable, non-blocking service communication.

Security Measures

- **SQL Injection**: Prisma uses parameterized queries, automatically sanitizing inputs to prevent SQL injection attacks.
- **Cross-Site Scripting (XSS)**: OAuth2 promotes best practices for secure token management, while Prisma encourages input validation, reducing the risk of XSS.
- **Cross-Site Request Forgery (CSRF)**: OAuth2 uses token-based authentication, which can help prevent CSRF by ensuring that requests are made by authenticated users.

Features

Authentication Service

- **User Signup**: Create new accounts with personal details.
- **User Signin**: Access accounts for registered customers.
- **Password Management**: Securely reset and update passwords.

- **User Session Management**: Maintains secure access to protected resources.

Booking Service

- **Yacht Rent Booking**: Enables users to book a yacht to rent.
- **Yacht Rent Cancellation**: Allows users to cancel booked a yacht.
- **Yacht Rent Update**: Allows users to update booked a yacht.

Additional Features

- **RabbitMQ Communication**: Reliable and asynchronous message processing.
- **CQRS Implementation**: Separates read/write operations for scalability.
- **Stripe Integration**: Secure payment processing (In progress).
- **Redis Caching**: Enhances response times and reduces database load.
- **Compression**:This middleware reduces the size of the response payloads, optimizing bandwidth usage and improving load times for users. It can significantly enhance the user experience, especially for clients with slower internet connections.
- **Helmet**: This middleware sets various HTTP headers to help protect the application from known web vulnerabilities. It includes protections against cross-site scripting (XSS), clickjacking, and other attacks, making it an essential security feature.
- **Rate Limiting**: This feature is implemented to control the number of requests a user can make to the API in a given timeframe. It protects the application from abuse and prevents overload, ensuring fair usage across all users and maintaining overall performance.
- **CORS**: Enabling CORS allows your API to be accessed from different domains securely. It prevents issues related to same-origin policy restrictions in browsers, facilitating seamless integration with various frontend applications.
- **Dockerization**: Ensures consistent service deployment.
- **Husky for Development**: Streamlines development workflows.
- **Inter-service Communication (Intersop)**: It'is crucial in microservices architecture, as it allows independent services to interact seamlessly. Effective communication ensures that data flows smoothly between services, facilitating complex operations like booking and payments.
- **Error Filtering**: Adding a filter for error handling is essential to maintain the robustness of the microservices architecture. This filter intercepts incoming requests, allowing for centralized management of error responses, logging, and exception handling. By doing so, it ensures that consistent error messages are sent to the clients, improving the user experience. It also aids in debugging and monitoring by logging detailed error information, which helps identify issues early in the request processing cycle.

- **OpenAI Service**: Integrates ChatGPT for optimal user assistance (Not started yet).

Technologies Used

- **TypeScript**
- **Node.js**
- **Nest.js**
- **MongoDB (TypeORM)**
- **PostgreSQL (Prisma)**
- **RabbitMQ (Message Queue)**
- **JSON Web Tokens (JWT)**
- **bcrypt.js**
- **Google Gmail (for Notification Service)**
- **Docker**
- **Swagger**

For more details and to access the code, please visit the Yacht Rental Management GitHub Repository.