Design Architecture for Hotel Booking Web Application

1. Overview

The Web Application is a backend service built with Node.js and Express.js. It provides APIs for hotel booking functionalities, including user authentication, hotel management, and booking management. The application uses MongoDB as its database and is containerized using Docker for deployment on AWS infrastructure.

2. Architecture Diagram

- Client (Browser/Postman) --> [Express.js API Server] --> [MongoDB Database]
  For deployment:
- Client --> AWS EC2 Instance --> Docker Container --> [Express.js API Server + MongoDB]

3. Components

Backend (Node.js + Express.js)

Purpose: Handles HTTP requests, processes business logic, and interacts with the database.

Key Features:

- API endpoints for hotel booking functionalities.
- Middleware for request handling.
- Error handling and logging using `winston`.

Database (MongoDB)

- **Purpose:** Stores data for hotels, users, and bookings.

Key Features:

- NoSQL database for flexible schema design.
- Connection retries implemented in case of failure.

Environmental Variables

- **Purpose:** Securely store sensitive information like database connection strings.

Key Variables:

- `MONGO`: MongoDB connection string.
- `JWT_SECRET` (if authentication is implemented).

Logging (Winston)

- Purpose: Logs application events for debugging and monitoring.
- Key Features:
- Logs to both console and a file (`app.log`).

Deployment (Docker + AWS)

- **Purpose**: Containerizes the application for consistent deployment and runs it on AWS EC2.
- **Key Features**:
- Dockerfile for building the application image.
- Docker Compose for managing services (e.g., MongoDB).
- Terraform for provisioning AWS infrastructure.

4. Application Flow

Startup Process

1. The application starts by initializing the Express server.
2. Environment variables are loaded using `dotenv`.
3. A connection to MongoDB is established using `mongoose.connect()`.
4. Logging is initialized using `winston`.

Request Handling

1. A client sends an HTTP request to the API server.
2. The Express server routes the request to the appropriate handler.
3. The handler processes the request, interacts with MongoDB if needed, and sends a response back to the client.

Database Connection

- The application attempts to connect to MongoDB on startup.
- If the connection fails, it retries every 5 seconds using `setTimeout`.

Logging

- Logs are generated for:
- Application startup.
- MongoDB connection events (connected/disconnected).
- API requests and errors.

5. Deployment Architecture

Local Development

- The application is run locally using:
  1. Bash
  2. node index.js

Containerized Deployment

- The application is containerized using Docker:
- **Dockerfile**: Builds the application image.
- **Docker Compose**: Manages the application and MongoDB services.

AWS Deployment

- The application is deployed on an AWS EC2 instance:
- Terraform provisions the EC2 instance and security groups.
- Docker runs the application inside a container.

6. Key Files

- `index.js`
- Entry point of the application.
- Initializes the Express server, connects to MongoDB, and sets up API routes.

`.env`

- Stores environment variables like the MongoDB connection string.

Dockerfile

- Builds a Docker image for the application.

`docker-compose.yml`

- Defines services for the application and MongoDB.

`main.tf`

- Terraform configuration for provisioning AWS infrastructure.

7. API Design

Endpoints

- **GET** `/`: Returns a welcome message.
- **Response**: `"Beginning of something spectacular"`

Future Endpoints (if applicable):

- **POST `/api/auth/register`:** Register a new user.
- **POST `/api/auth/login`:** Log in and receive a JWT.
- **GET `/api/hotels`:** Retrieve all hotel listings.
- **POST `/api/hotels`:** Add a new hotel (admin only).
- **POST `/api/bookings`:** Create a new booking.

8. Error Handling

- Database Connection Errors:
- Retries connection every 5 seconds if MongoDB is unavailable.

API Errors:

- Logs errors using `winston`.
- Sends appropriate HTTP status codes and error messages to the client.

9. Security

Environment Variables:

- Sensitive information like the MongoDB connection string is stored in .env.

Access Control (Future Implementation):

- Use JWT for user authentication and authorization.

Network Security:

- Security groups restrict access to port `22` (SSH) and `2704` (API).

10. Scalability

Horizontal Scaling:

- Use a load balancer (e.g., AWS ELB) to distribute traffic across multiple EC2 instances.

Database Scaling:

Use MongoDB Atlas for managed database services with auto-scaling.

Container Orchestration:

Use Kubernetes or AWS ECS to manage containers in production.

11. Monitoring and Logging

Monitoring:

- Use tools like Prometheus and Grafana to monitor application performance.
- Use AWS CloudWatch to monitor EC2 instance metrics.

Logging:

- Logs are written to both the console and a file (`app.log`) using `winston`.

12. Future Enhancements

- Add user authentication and authorization using JWT.
- Implement a frontend for user interaction.
- Integrate payment gateways for booking payments.
- Add CI/CD pipelines using GitHub Actions or AWS CodePipeline.