Smart Tour Guide

Smart Tour Guide

Abstract

Table of Contents

List of Figures

List of Tables

1 Backend

1.1 Introduction to the MERN Stack

The MERN stack is a popular technology stack used for building modern web applications. It comprises four main technologies: MongoDB, Express.js, React, and Node.js. These technologies provide a powerful and efficient platform for developing full-stack web applications with a single language, JavaScript, across both the client and server sides.

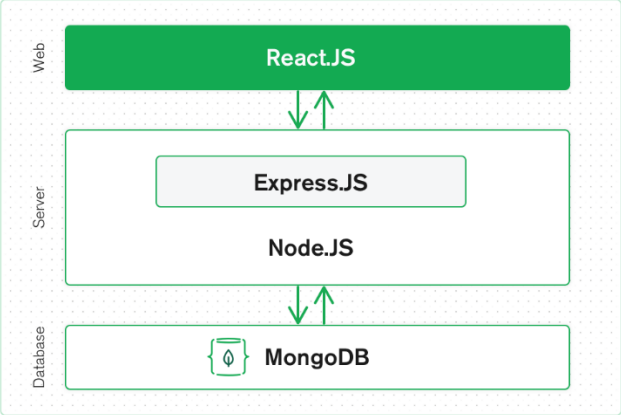


Figure 1-1 MERN architecture

Benefits of the MERN Stack

The MERN stack offers several advantages for web application development:

- **Full-Stack JavaScript**: By using JavaScript for both the client and server sides, the MERN stack enables seamless communication and consistency in code.
- **Rapid Development**: The combination of MongoDB, Express.js, React, and Node.js provides a comprehensive and efficient framework for developing modern web applications quickly.
- **Flexibility and Scalability**: Each component of the MERN stack is designed to be flexible and scalable, allowing developers to build applications that can grow and adapt to changing requirements.
- **Active Community and Ecosystem**: The technologies in the MERN stack have large, active communities and extensive ecosystems, offering a wealth of resources, libraries, and tools to support development.

Use Cases for the MERN Stack

The MERN stack is versatile and can be used to build a wide range of web applications, including:

- **Single Page Applications (SPAs)**: React's efficient rendering and component-based architecture make it ideal for SPAs that require dynamic, real-time user interactions.
- **E-commerce Platforms**: The stack's scalability and performance make it suitable for building robust e-commerce solutions with complex user and data management needs.
- **Content Management Systems (CMS)**: The flexibility of MongoDB and the simplicity of Express.js allow for the creation of customized CMS solutions that can handle various types of content and user roles.
- **Social Media Applications**: The stack's ability to handle high traffic and real-time interactions makes it a good choice for building social media platforms.

1.2 Tools and Technologies

1.2.1 Node.js and Its Ecosystem

Node.js is chosen as the runtime environment for this project due to its event-driven, non-blocking I/O model, making it efficient for handling concurrent requests. It allows us to build scalable and fast backend applications. Key features include:

Event-Driven Architecture: Utilizes asynchronous programming to handle multiple requests simultaneously, enhancing performance.

Package Management with npm: Centralized repository for Node.js packages, facilitating easy installation and management of dependencies like Express.js, Mongoose, and more.

Extensive Ecosystem: Abundance of libraries and frameworks that expedite development processes, such as authentication (jsonwebtoken), email handling (Nodemailer), and image processing (Multer).

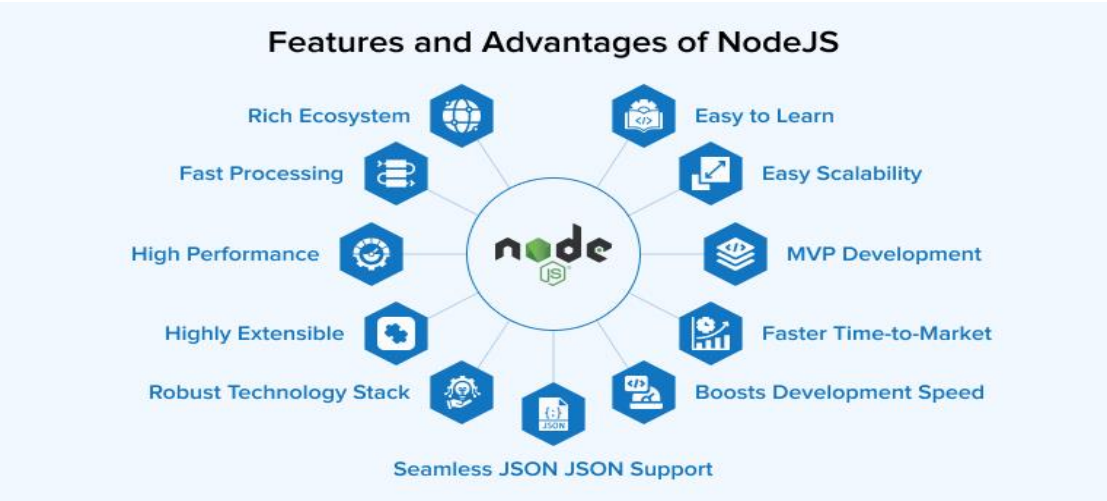


Figure 1-2 NodeJs Features

1.2.2 Express.js Framework

Express.js is chosen as the web application framework for Node.js due to its minimalist and flexible nature, allowing us to design and develop robust APIs efficiently. Key features include:

Routing: Simple yet powerful routing mechanism to handle HTTP requests and define endpoints for various functionalities (e.g., user management, tour operations).

Middleware Support: Extensive middleware options for handling authentication, logging, error handling, and more.

Template Engines: Integrates with popular template engines like Pug or EJS for server-side rendering of views (though our project focuses on building APIs, we acknowledge this capability).

1.2.3 MongoDB as the Database Choice

MongoDB is selected as the database due to its schema-less nature and flexibility, making it suitable for storing diverse data types related to tours, users, bookings, and more. Key features include:

JSON-Like Documents: Data stored in BSON format allows for easy representation of complex hierarchical relationships without extensive schema migrations.

Scalability: Supports horizontal scaling through sharding, enabling the system to handle growing data volumes.

Integration with Mongoose: ODM (Object Data Modeling) library for MongoDB and Node.js, providing a straightforward way to model application data and enforce schema validations.

1.2.4 Other Essential Tools and Libraries

Visual Studio Code: Chosen as the Integrated Development Environment (IDE) for its lightweight yet powerful features, including debugging, IntelliSense, and Git integration.

Postman: Used for API development and testing, facilitating easy creation and execution of API requests to verify endpoints and data flow.

1.2.5 Development Environment Setup

1.2.5.1 Installing Node.js and MongoDB Locally

Node.js Installation: Steps to download and install Node.js on the development machine.

MongoDB Setup: Instructions for installing MongoDB locally or using a cloud-based service like MongoDB Atlas.

1.2.5.2 Configuring Development Tools

Visual Studio Code Configuration: Setting up extensions (e.g., ESLint for code linting, Prettier for code formatting) to enhance productivity and maintain code quality.

Postman Setup: Configuring environment variables, setting up collections, and utilizing features like pre-request scripts for automated testing and validation of APIs.

1.3 Designing the Backend Architecture

1.3.1 Application Architecture Overview

1.3.1.1 Choosing an Application Architecture

The Smart Tour Guide application adopts a **Monolithic Architecture** due to its simplicity in development and deployment for the initial stages. This architecture style consolidates all functionality into a single unit, handling all processing responsibilities from user interface to data storage within a single codebase.

1.3.2 Considerations and Choices

1.3.2.1 API Design

RESTful API: Adopted for its simplicity, statelessness, and widespread adoption in web applications.

Benefits: Facilitates clear separation of concerns, enhances scalability through resource-based architecture, and simplifies client-server communication.

Drawbacks: Can lead to bloated codebase as the application scales, potentially impacting maintainability and agility.

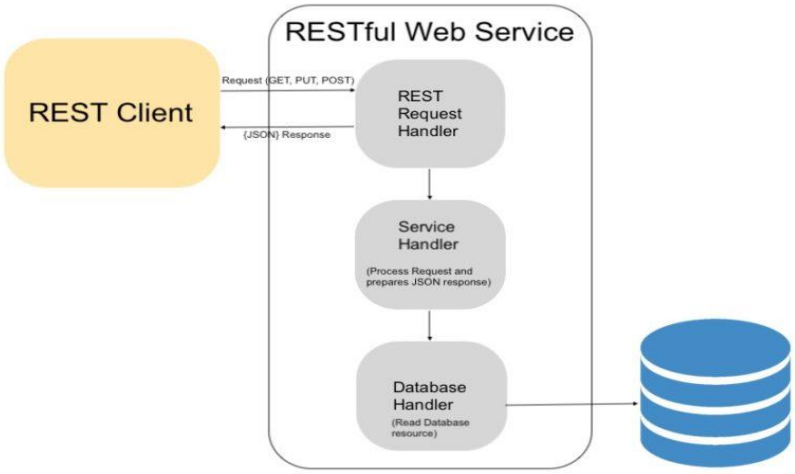


Figure 1-3 RESTful web service architecture

1.3.2.2 Data Management

Database Choice: Selected MongoDB for its flexibility with unstructured data and ability to scale horizontally.

Benefits: Supports agile development with schema-less design, enhances performance with document-oriented storage, and integrates well with Node.js ecosystem.

Drawbacks: Lacks transaction support across multiple documents, challenging complex relational queries, and requires careful schema design to avoid performance bottlenecks.

1.3.2.3 Security Measures

Authentication and Authorization: Implemented using JSON Web Tokens (JWT) for secure user sessions and role-based access control (RBAC) to manage user permissions.

Benefits: Enhances application security by preventing unauthorized access, facilitates seamless user sessions across multiple requests, and supports scalable user management.

Drawbacks: Requires careful management of tokens to mitigate security risks like token leakage or session hijacking.

This section outlines the design of the backend architecture for the Smart Tour Guide application using a Monolithic Architecture. We've discussed the adoption of RESTful APIs for clear client-server interaction, MongoDB for flexible data management, and JWT for robust authentication and authorization. These decisions aim to streamline development, ensure security, and support initial scalability as we build out the application.

1.4 Implementing Core Features

1.4.1 User Management

In the application, user management is foundational. This includes handling user authentication, authorization, and profile management.

1.4.1.1 User Authentication and Authorization using JWT

Authentication is the process of verifying the identity of a user attempting to access the application. In the Smart Tour Guide application, we use JWT-based authentication to securely manage user sessions.

Authorization determines the level of access granted to authenticated users based on their roles or permissions. RBAC (Role-Based Access Control) is employed to manage user privileges effectively.

JWT (JSON Web Tokens) serves as a pivotal mechanism for secure user authentication. Upon successful login, a JWT is generated and sent to the client, which is then included in subsequent requests to authenticate and authorize access to protected routes and resources.

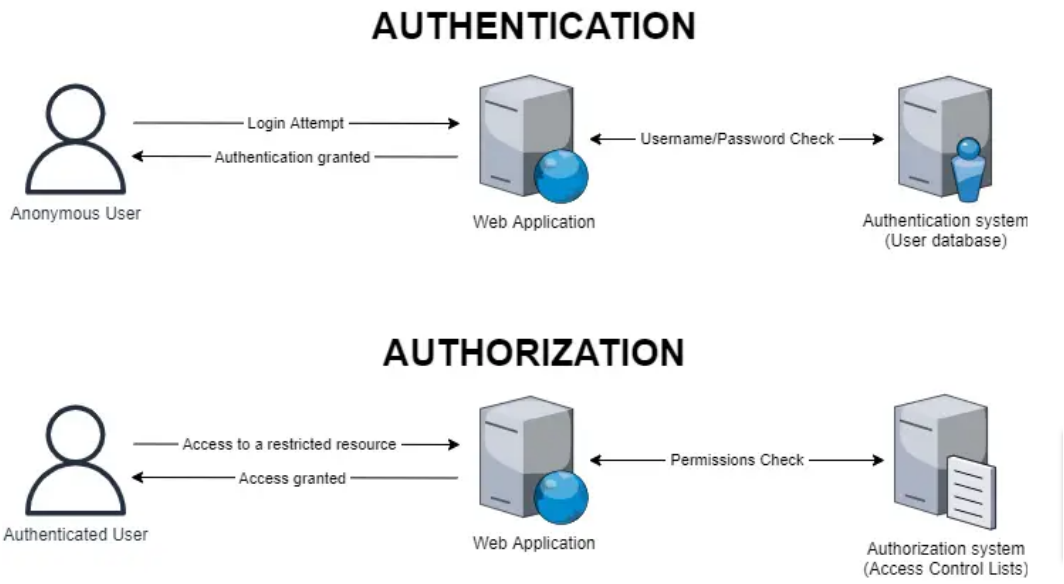


Figure 1-4 Authentication and Authorization

Implementation Details:

Authentication Flow: Users authenticate with their credentials (username/email and password), which are validated against stored credentials in your database (MongoDB).

Token Generation and Management: Upon successful authentication, a JWT containing user data (such as user ID and role) is signed and sent to the client.

RBAC Implementation: Users are assigned roles such as admin, guide, or tourist, each with predefined permissions.

Middleware Usage: Middleware functions in Node.js intercept incoming requests to validate JWT and authorize access to protected routes based on user roles.

1.4.1.2 User Profile Management and Settings

User profile management involves CRUD (Create, Read, Update, Delete) operations on user profiles and settings.

Implementation Details:

Profile Updates: Users can update their profile information, including name, email, and profile picture. Changes are reflected in the database and persisted.

1.4.2 Standard Tour Management

The application features both ready-made and customized tours, necessitating comprehensive management capabilities.

CRUD Operations for Standard Tours

Tour management encompasses the creation, retrieval, update, and deletion of tour data. This includes details about landmarks, tour itineraries, pricing, and availability.

Implementation Details:

Tour Creation: Admin or authorized users can create new tours, specifying details such as tour name, description, duration, and associated landmarks.

Tour Retrieval and Display: Tours are fetched from the database and displayed on the front end, allowing users to browse and select tours based on their preferences and requirements.

Tour Updates and Deletion: Admins manage tours dynamically, updating details or removing outdated tours as needed.

1.4.2.1 Standard Tour Booking Process

Tour Selection:

Browse and Choose: Tourists explore available standard tours listed in the application, categorized by location, type, and duration.

Selection: Upon choosing a tour, tourists proceed to book directly through the platform.

Payment Processing:

Integration with Stripe: Tourists make payment for the selected standard tour using the integrated Stripe payment gateway.

Transaction Security: Stripe ensures the security and integrity of payment transactions, safeguarding sensitive financial details.

Booking Confirmation:

Instant Confirmation: Upon successful payment, tourists receive immediate booking confirmation.

Details: The confirmation includes tour details, itinerary, meeting points, and payment receipt.

Tour Execution:

Tour Date: On the scheduled date and time, tourists join the standard tour at the designated starting point.

Tour Services: The tour proceeds with provided services such as guided sightseeing, transportation, and other specified inclusions.

Feedback and Review:

Post-Tour Feedback: Tourists have the opportunity to provide feedback and reviews about the standard tour experience.

Improving Services: Reviews contribute to enhancing service quality and aiding future tourists in making informed choices.

1.4.3 Customized Tour Management

In addition to ready-made tours, the application offers customized tour options, catering to personalized preferences and specific user requests.

1.4.3.1 Definition and Purpose of Customized Tours

Customized tours allow users to tailor their travel experiences based on unique interests, schedules, and preferences. This feature enhances user engagement by providing flexibility and personalized service.

1.4.3.2 Customized Tour Booking Process:

Tour Request Initiation:

User Initiation: Tourists log into the application and navigate to the customized tour creation page.

Tour Details: They select a city or landmarks of interest, preferred languages, group size, dates, and any specific requests or comments.

Submission: Upon submission, the tour request is saved in the database as a Customized Tour document and the user can browse available guides, chat with any of them to send them a request or wait for other guides to respond to his request.

Guide Proposal and Selection:

Guide Response: Guides review available tour requests and respond with proposals including pricing, additional services, and customized tour details.

Tourist Review: Tourists evaluate guide proposals based on factors like pricing, guide ratings, proposed itineraries, and compatibility.

Acceptance: Tourists can accept a guide's proposal, which updates the tour status to "confirmed" for that guide.

Payment Processing:

Payment Gateway: Upon accepting a guide's proposal, tourists proceed to make payment through an integrated payment gateway such as Stripe.

Secure Transaction: Stripe ensures secure processing of payments, protecting sensitive financial information.

Tour Execution:

Scheduled Tour: On the agreed-upon date, tourists meet the guide at the designated location.

Services: The guide provides services as per the agreed itinerary, including transportation, narration, and guiding.

Feedback and Review:

Post-Tour Review: After the tour concludes, tourists provide feedback and reviews based on their experience.

Quality Assurance: Reviews help maintain service quality and assist future tourists in making informed decisions.

1.4.3.3 Cron Job to Automate Tour Completion

A cron job is a scheduled task that runs automatically at specified intervals. In our application, we use a cron job to automate the process of marking tours as completed once their end date has passed. This ensures that the status of tours is kept up-to-date without manual intervention.

Importance of the Cron Job

1. **Automation**: The cron job eliminates the need for manual checks and updates, saving time and reducing the risk of human error.
2. **Consistency**: It ensures that the status of all tours is consistently updated, providing accurate information to users and administrators.

3. **Efficiency**: By running the cron job at off-peak hours (midnight), we can update the tour statuses without affecting the performance of the system during peak usage times.

The cron job is configured to run daily at midnight. It scans the database for tours that have ended and updates their status to "completed" if they meet certain conditions, such as having a confirmed status and a completed payment.

1.4.4 Security Considerations

Security is paramount in your application to protect user data, prevent unauthorized access, and safeguard against potential vulnerabilities.

1.4.4.1 Secure JWT Handling

Token Storage: Tokens stored securely in client-side storage (e.g., HTTP-only cookies or local storage) to prevent XSS attacks.

Token Expiry: JWTs configured with an expiration time to mitigate the risk of token misuse or replay attacks.

1.4.4.2 Mongo-sanitize

Mongo-sanitize mitigates MongoDB Operator Injection risks by sanitizing user input. It ensures that user-supplied data does not include MongoDB query operators that could alter database operations.

Implementation Details:

Input Sanitization: Before processing user input, MongoDB queries are sanitized to remove any potential MongoDB operators embedded in user-supplied data.

Preventing Injection Attacks: By filtering out malicious input, Mongo-sanitize strengthens security measures, reducing the risk of injection attacks that could compromise database integrity

1.5 Enhancing Security and Performance

1.5.1 Security Measures

1.5.1.1 Implementing Password Hashing with bcryptjs

In our application, bcryptjs is used for securely hashing passwords. When users register or update their passwords, bcryptjs hashes the password before storing it in the database. This method involves generating a unique salt for each password and hashing it multiple times. During login, the entered password is hashed and compared with the stored hash. This approach ensures that

even if the database is compromised, plain text passwords are not exposed, significantly enhancing security.

1.5.1.2 Securing APIs with helmet and express-rate-limit Middlewares

To enhance the security of our APIs, we use two essential middleware libraries: helmet and express-rate-limit.

Helmet: Helmet sets various HTTP headers to protect the application from well-known web vulnerabilities like cross-site scripting (XSS) and clickjacking. By configuring these headers appropriately, Helmet provides a basic layer of security against common attacks.

Express-Rate-Limit: Express-rate-limit is used to limit the number of requests an IP address can make to the API within a specific timeframe. This helps protect the application from brute force attacks and denial-of-service (DoS) attacks by throttling excessive requests.

Both middleware functions are integrated into the Express app, with helmet setting security headers and express-rate-limit configured to restrict requests to sensitive routes, such as login endpoints.

1.5.1.3 Cross-Origin Resource Sharing (CORS) Implementation

Our application uses the cors middleware to handle Cross-Origin Resource Sharing (CORS). CORS allows web applications to request resources from another domain, which is crucial for enabling interactions with client-side applications hosted on different domains. By implementing CORS, we can control access from different origins while enforcing security policies. This ensures that only specified domains can interact with our resources, maintaining a balance between functionality and security. The cors middleware is configured with options like allowed origins, methods, and headers to manage these cross-domain interactions effectively.

1.6 Implementing Advanced Features

1.6.1 Email Notifications

One of the key features of the application is the ability to send automated email notifications to users. This functionality is critical for maintaining effective communication and ensuring that users are kept informed about important events and updates related to their tours.

1.6.1.1 Nodemailer Integration

We utilized Nodemailer, a powerful Node.js module, to handle all our email-sending needs. Nodemailer supports various transport methods, making it a versatile choice for sending emails.

For our project, we configured Nodemailer to use an SMTP transport, typically through a service like Bervo or mailtrap, which ensures reliable delivery of our transactional emails.

1.6.1.2 Email templates and customization

To ensure consistency and save development time, we implemented email templates using Pug (formerly known as Jade). Pug allows us to create dynamic email templates that can be customized with user-specific information. This means that every email sent to a user can be personalized with details such as the user's name, booking information, and other relevant data. By compiling these templates with Nodemailer, we can generate and send well-structured and visually appealing emails to users.

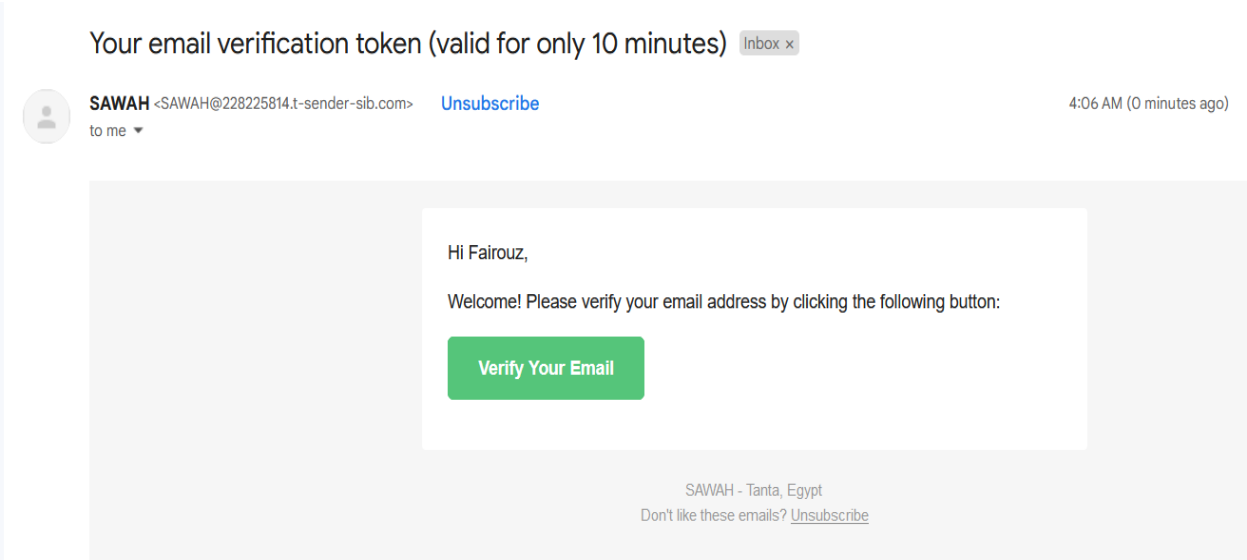


Figure 1-5 Email Example

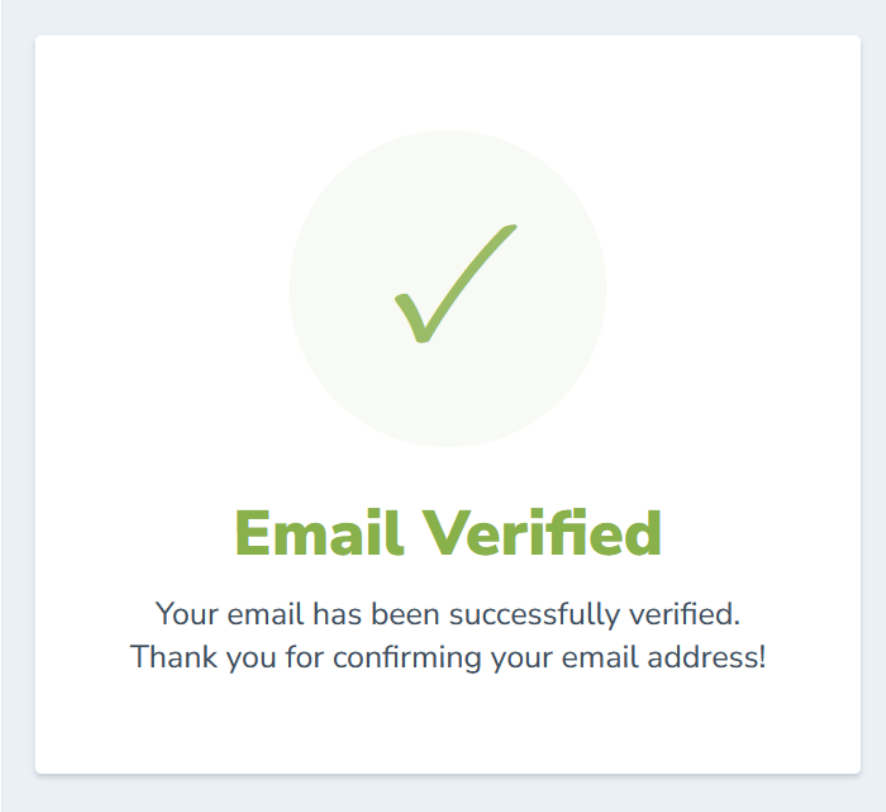


Figure 1-6 PUG Template

1.6.2 Real-time User Messages to Admin

In our application, real-time communication is facilitated between users and the admin to handle inquiries, issues, or feedback promptly.

WebSocket Integration: To enable real-time messaging, we integrated WebSockets using the `socket.io` library. This allows users to send messages directly to the admin in real time. When a user sends a message, it is instantly delivered to the admin's interface, where they can view and respond to it immediately. This setup ensures that any user queries or concerns are addressed quickly, enhancing user satisfaction and providing a seamless communication experience.

The real-time messaging system is particularly useful for scenarios where users need immediate assistance or have urgent questions. By using WebSockets, we ensure that the communication is instantaneous, reducing the wait time for users and enabling the admin to handle multiple conversations efficiently.

1.6.3 Booking and Payment Processes

Ensuring a smooth and secure booking and payment process is crucial for the success of our application. We implemented robust systems to handle both customized and standard tour bookings, as well as secure payment processing using Stripe.

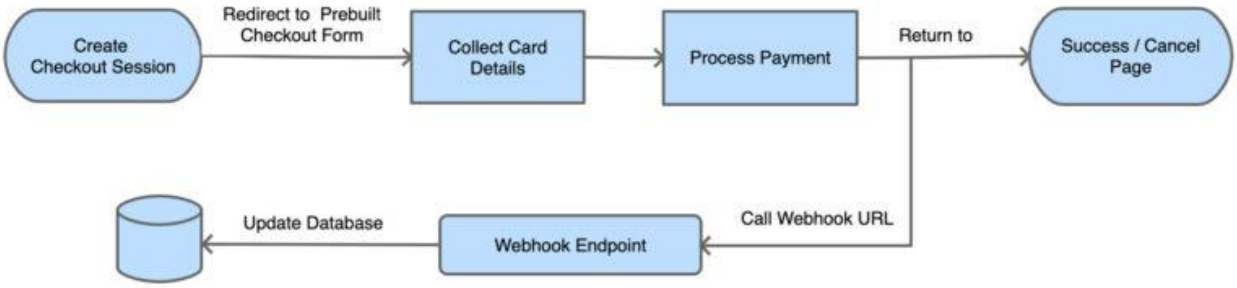


Figure 1-7 Stripe Payment Flow

1.6.4 Logging

Effective logging is crucial for maintaining and debugging an application. It helps developers monitor application behavior, diagnose issues, and maintain a reliable service. In our application, we employed two powerful logging tools: Winston and Morgan.

Winston: Winston is a versatile logging library for Node.js, known for its flexibility and feature-rich capabilities. We used Winston to create detailed logs of various levels, such as info, warn, error, and debug. This granularity allows us to capture comprehensive logs that provide insights into the application's operations and errors. Winston's ability to log to multiple transports (such as files, databases, or third-party services) makes it a robust choice for centralized logging. In our project, Winston is configured to log errors and important events to log files, ensuring that we have a permanent record of critical information.

Morgan: Morgan is a middleware for logging HTTP requests in Node.js applications. It is specifically designed to capture and log incoming requests and their responses, making it an invaluable tool for monitoring API interactions. We integrated Morgan to log each request made to our application, including details like request method, URL, status code, and response time. This helps us track the application's usage patterns, identify performance bottlenecks, and quickly respond to issues related to API endpoints. By combining Morgan with Winston, we ensure that our HTTP request logs are comprehensive and easy to analyze.

1.7 Admin Dashboard and Analytics

The admin dashboard in our application provides a comprehensive overview of various statistics and performance metrics, crucial for monitoring the system and making informed decisions. The dashboard is implemented with a combination of MongoDB aggregations and Redis caching for optimal performance and scalability.

1.7.1 Real-time Statistics

1.7.1.1 Overview of Key Metrics

The dashboard offers real-time statistics on key metrics, including the number of users, guides, reviews, landmarks, tours, and bookings. These metrics provide a snapshot of the system's current state, essential for tracking growth and engagement. By counting documents in the MongoDB collections corresponding to each entity, the system updates these statistics regularly.

1.7.1.2 Caching with Redis

To reduce the load on the database and improve response times, Redis is used to cache frequently accessed statistics. This cached data is refreshed daily, ensuring the statistics are relatively up-to-date while minimizing database queries. If Redis caching is enabled, the application first checks the cache for existing statistics before querying the database.

1.7.2 Advanced Analytics

1.7.2.1 Monthly Sign-Ups

We track user sign-ups over the past year, grouped by month, using MongoDB's aggregation framework. This helps in understanding user acquisition trends and identifying peak periods. By grouping users based on their registration date, the system can display the number of new users each month.

1.7.2.2 Monthly Revenue

Revenue statistics are aggregated by month, providing insights into the application's financial performance. This includes total revenue generated each month, which is crucial for financial planning and analysis. By grouping bookings based on their creation date and summing up the total price, the system can present monthly revenue figures.

1.7.2.3 Overall Revenue Statistics

In addition to monthly revenue, we calculate overall revenue statistics, including total and average revenue. This provides a comprehensive view of the application's financial health. The system aggregates all bookings to compute these figures, offering insights into both total earnings and average booking values.

1.7.2.4 Popular Landmarks

Identifying the most popular landmarks helps in understanding user preferences and planning future tours. This is achieved by aggregating the number of bookings for each landmark. By analyzing customized tour requests, the system can determine which landmarks are most frequently requested and booked.

1.7.2.5 Top Guides and Tours

The dashboard also highlights the top-performing guides and tours based on booking counts. This information is valuable for recognizing and rewarding high-performing guides and understanding which tours are most popular among users. By aggregating bookings, the system can identify guides with the most tours and tours with the highest booking rates.

1.7.3 Real-Time Messaging

The application implements a real-time messaging system allowing users to send messages to the admin. This feature uses WebSocket technology to provide instant communication between users and administrators. When a user sends a message, it is immediately delivered to the admin, ensuring timely responses and efficient issue resolution.

1.8 Documentation and API Design

1.8.1 API Documentation with Postman

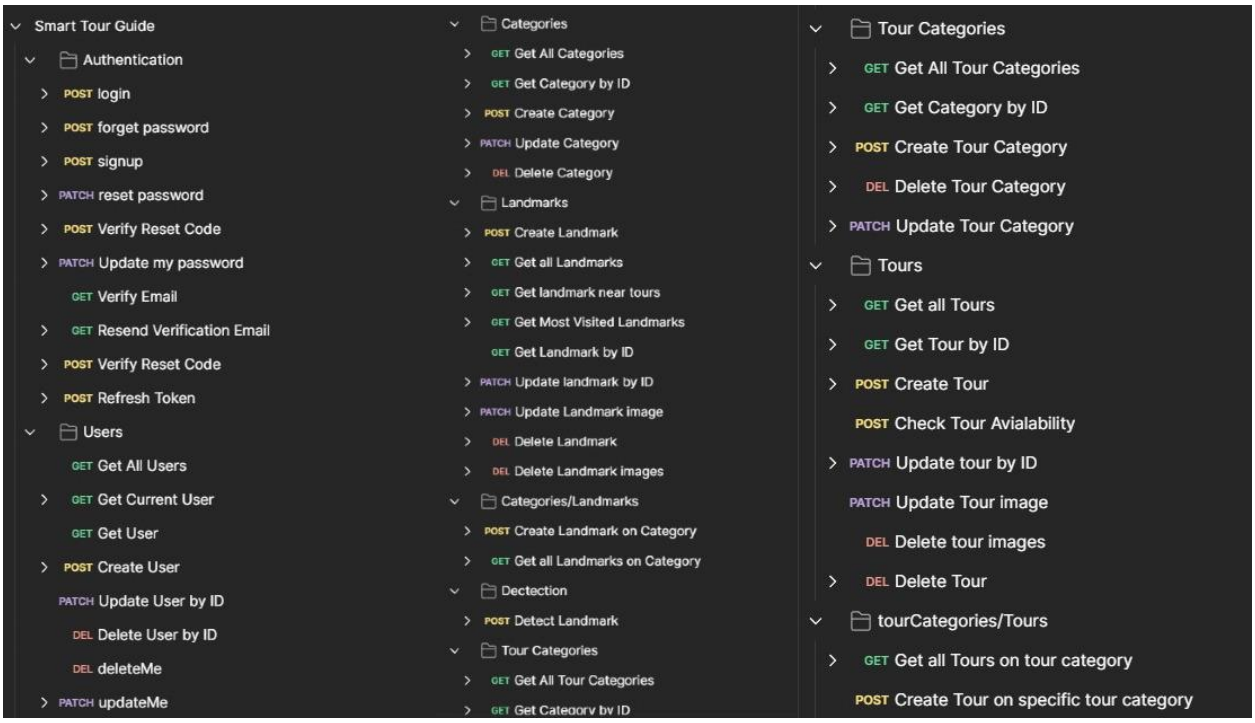
API documentation is crucial for developers who interact with your API, providing clear and comprehensive information about the endpoints, request/response formats, authentication methods, and error handling. Postman, a popular API development tool, offers features to document APIs effectively.

Postman Collections

Postman Collections allow you to group related API endpoints and provide detailed descriptions for each one. These collections can include examples of requests and responses, making it easier for developers to understand how to use the API. Each endpoint can be documented with parameters, headers, body data, and authentication requirements.

Sharing and Collaboration

Postman enables sharing collections with team members or external developers. By providing a shared URL or exporting the collection, you ensure that everyone has access to the latest API documentation. Postman also offers collaboration features, allowing team members to comment on and update documentation in real-time.



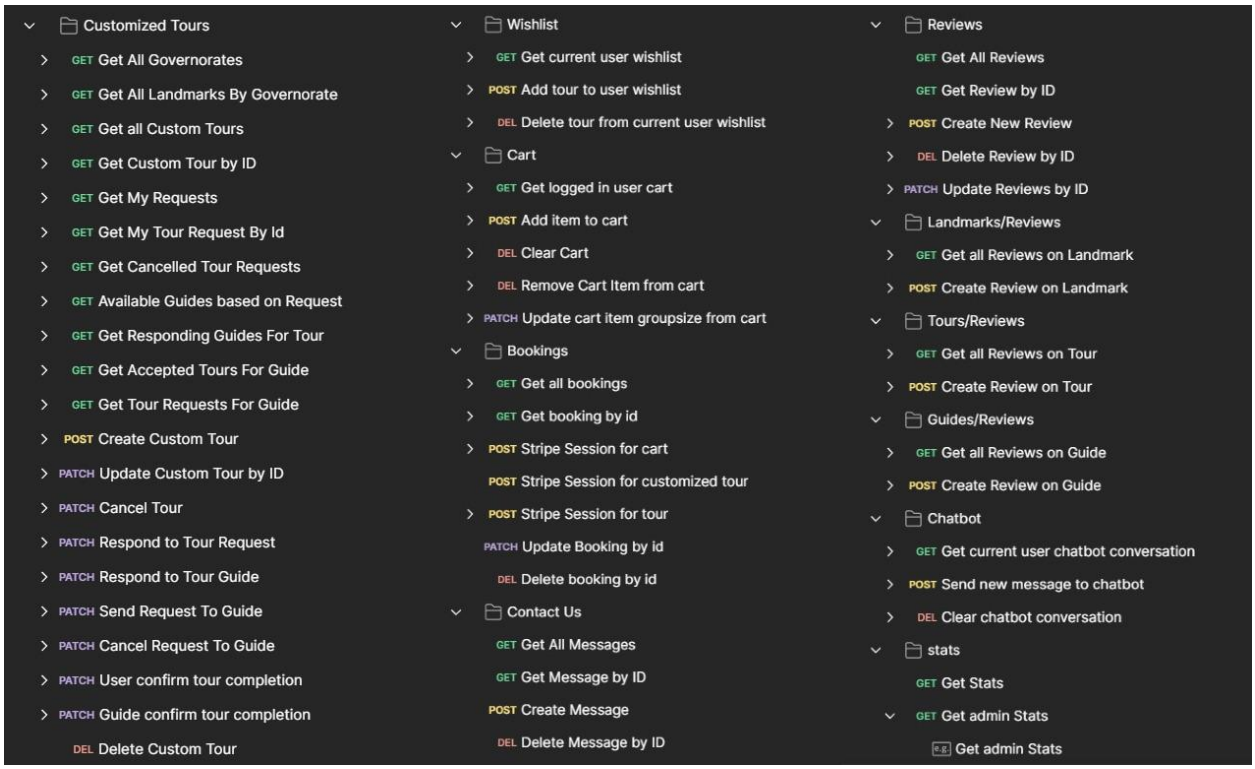


Figure 1-8 API Endpoints

Automated Documentation Generation

Postman can automatically generate API documentation based on the collections you create. This documentation can be hosted on Postman's public or private servers, providing an interactive interface for developers to explore and test the API endpoints directly from the documentation.

1.9 User Experience Enhancements

Improving user experience is a critical aspect of modern web applications. For our project, we've implemented various features to ensure a smooth and efficient user experience. These include pagination, sorting, and search functionality for endpoints that return large datasets.

1.9.1 Pagination

Pagination divides large datasets into smaller, more manageable chunks or pages. This approach reduces the amount of data sent to the client at once, resulting in faster load times and a more responsive application. In our application, pagination is implemented by accepting query parameters such as `page` and `limit` in API requests.

- **Page**: Indicates the current page number.
- **Limit**: Specifies the number of items per page.

1.9.2 Sorting

Sorting allows users to organize data based on specific fields, such as name, date, or rating. This feature is implemented by accepting a `sort` parameter in API requests, which specifies the field to sort by and the order (ascending or descending).

1.9.3 Search

We have also added search functionality to enhance the user experience. This feature allows users to search for specific items using keywords. The search parameter is processed on the server side to filter the dataset based on the provided search term.

1.9.4 Integrating a Virtual Tour Guide

The chatbot integration in our application provides users with an interactive and engaging way to get information about Egypt and its landmarks. Leveraging Google's Generative AI, the chatbot acts as a virtual tour guide and historian, answering user queries and providing detailed information about tours and historical sites.

1.9.4.1 Key Features of the Chatbot

Interactive Conversations: Users can ask questions about Egypt's history, landmarks, and tours, and receive informative responses from the chatbot.

Personalized Experience: The chatbot maintains a history of conversations, allowing it to provide contextually relevant responses and continue ongoing discussions.

Scalability: The chatbot can handle multiple user interactions simultaneously, making it a scalable solution for providing information and enhancing user engagement.

1.9.4.2 Use Cases

Tour Information: Users can inquire about specific tours, including details about landmarks, tour schedules, and pricing.

Historical Insights: The chatbot can provide historical context and interesting facts about various Egyptian landmarks and sites.

User Assistance: It can assist users in navigating the application, booking tours, and addressing common queries.

1.10 Utilizing Docker for Development and Deployment

To streamline the development and deployment process of the Smart Tour Guide application, Docker is utilized. Docker allows for the effective packaging, shipping, and running of the application in a consistent environment, ensuring that both the frontend and mobile developers do not need to install all dependencies locally.

Benefits of Using Docker

1. **Consistency:** Docker containers ensure that the application runs in the same environment across all stages, from development to production.
2. **Isolation:** Each component of the application can run in its isolated container, preventing conflicts and simplifying dependency management.
3. **Reusability:** Docker images can be reused across different environments, reducing the need for repeated setup and configuration.

Development Setup

- **Frontend and Mobile Development:** Developers can use Docker to spin up the necessary services, such as the backend API, without installing Node.js, MongoDB, or Redis on their local machines. This minimizes the setup time and avoids potential conflicts with other projects.

- **Shared Development Environment:** Docker Compose is used to define and run multi-container Docker applications. This setup includes the backend server, database, and any other services required for development, providing a consistent environment for all developers.

For future additions docker can be used in deployment process

Deployment Process

- **Containerization:** The backend application, along with its dependencies, is packaged into a Docker image. This image can be deployed to various environments, ensuring consistency and reliability.
- **Scalability:** Docker makes it easier to scale the application by running multiple instances of containers, which can be managed using orchestration tools like Kubernetes.
- **Portability:** Docker images can be easily moved between different cloud providers or on-premise servers, providing flexibility in deployment options.

```yaml
version: "3.8"
services:
  app:
    build: .
    ports:
      - "8000:8000"
    environment:
      DATABASE: mongodb://db:27017
      REDIS_URL: redis://redis:6379
    volumes:
      - .:/app
  db:
    image: mongo:5.0.28-rc0-focal
    ports:
    volumes:
      - mongo-data:/data/db
  redis:
    image: redis:latest
    ports:
      - "6379:6379"
    volumes:
      - redis-data:/data

volumes:
  mongo-data:
  redis-data:
```

Figure 1-9 - Docker Compose

1.11 Redis Caching

Redis is employed to cache frequently accessed data, enhancing performance and reducing the load on the database. It serves as an in-memory data store that allows quick read operations, which is crucial for improving the responsiveness of our application.

1.11.1 Purpose of Redis Caching

Performance Enhancement: By storing frequently accessed data in memory, Redis significantly reduces the time required to fetch data compared to querying the database each time. This is particularly useful for data that does not change frequently, such as landmark details and tour information.

Reduced Database Load: By caching database query results, Redis reduces the number of direct queries made to the MongoDB database, thereby decreasing the load and improving overall system performance.

Scalability: The caching mechanism scales with increasing user traffic, maintaining efficient data retrieval even during peak usage periods.

1.11.2 Implementation

The caching strategy involves storing the results of frequently accessed API responses in Redis. When a request is made, the application first checks if the data is available in the Redis cache. If it is, the cached data is returned, thereby reducing the latency. If not, the data is fetched from the database, stored in Redis for future requests, and then returned to the client.

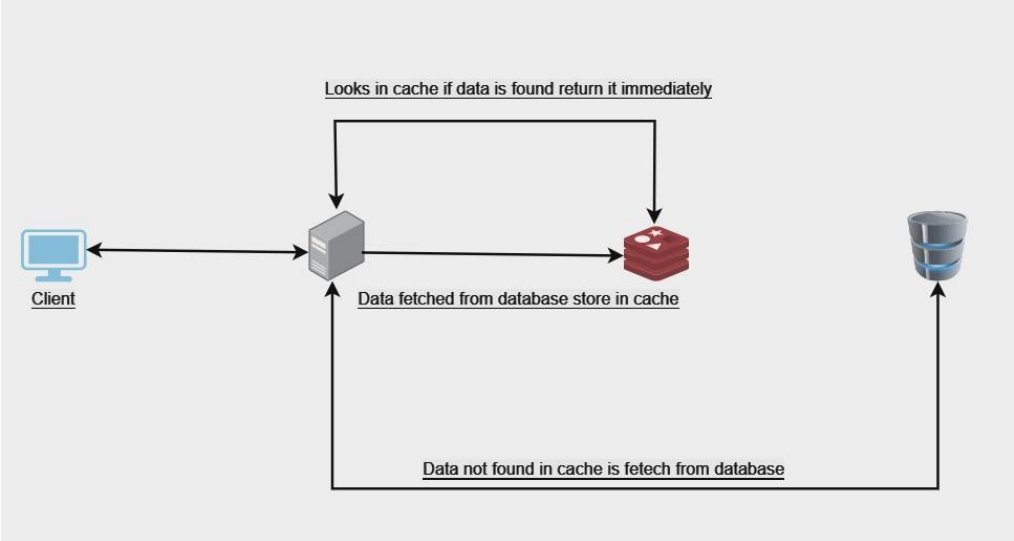


Figure 1-10 Redis Flow

1.11.3 Example Use Cases

Landmark Details: When the user requests all landmarks, with any query type the application first checks if the data is available in the Redis cache. If found, the data is returned immediately. If not, it is fetched from the MongoDB database, cached in Redis, and then returned to the user.

Tours, Categories, Tours Categories: Similar caching is applied for frequently accessed tour information, ensuring quick response times for users browsing tours.

By incorporating Redis caching into the Smart Tour Guide application, the backend achieves better performance, reduced latency, and a more efficient handling of high volumes of requests, especially for read-heavy operations.

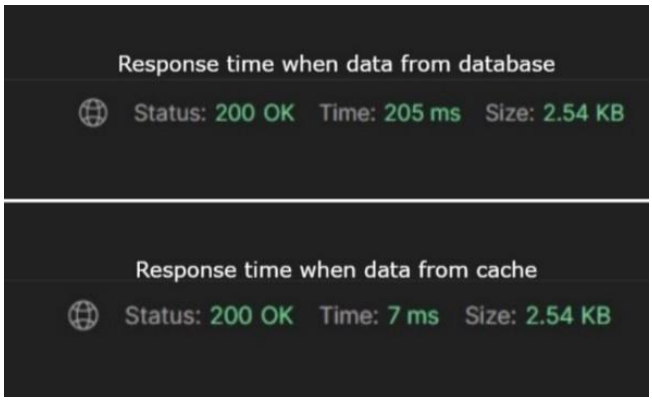


Figure 1-11 Effect of Redis

1.12 Deployment and Scalability:

1.12.1 Why Google Compute Engine

Google Compute Engine (GCE) is a Infrastructure as a Service (IaaS) offering from Google Cloud Platform (GCP) that allows users to run virtual machines (VMs) on Google's global infrastructure. Here are key points about GCE:

Virtual Machines (VMs): GCE provides virtual machine instances that users can customize and control. These VMs can run various operating systems and applications.

Scalability: GCE offers scalability by allowing users to dynamically adjust the number and size of their VM instances based on workload demands.

Global Infrastructure: Google's global network infrastructure supports GCE, providing high performance, low-latency connectivity across regions and availability zones.

Management Tools: GCE provides management tools such as Cloud Console, CLI (Command Line Interface), and APIs for managing and monitoring VM instances and resources.

Integration: GCE integrates seamlessly with other Google Cloud services like Google Kubernetes Engine (GKE), Cloud Storage, BigQuery, and more, enabling a comprehensive cloud computing ecosystem.

Security: Google Cloud's security model ensures data protection, compliance, and robust network security features for VM instances deployed on GCE.

Use Cases: GCE is suitable for various use cases including website hosting, application development and testing, data processing and analytics, machine learning, and more.

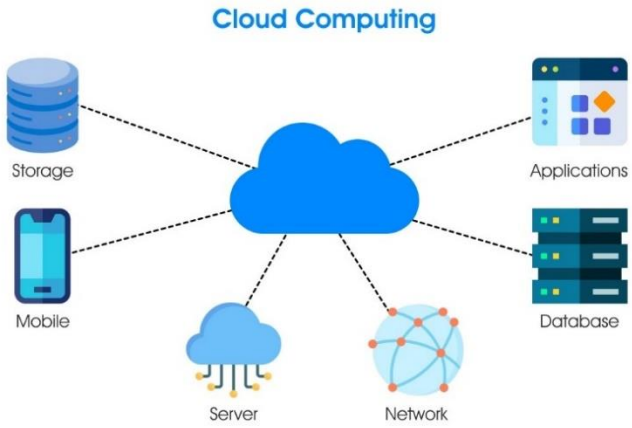


Figure 1-12 Cloud

1.12.2 What is SSL/TLS

SSL/TLS (Secure Sockets Layer/Transport Layer Security) is a cryptographic protocol designed to provide secure communication over a computer network. It ensures that the data transmitted between a client (such as a web browser) and a server (such as a web server) remains private and integral. Here's a detailed explanation of SSL/TLS:

SSL (Secure Sockets Layer)

SSL was developed by Netscape in the 1990s as a protocol to secure communications over the internet. It operates at the application layer of the OSI model and uses cryptographic techniques to encrypt data exchanged between clients and servers. The primary goals of SSL are:

- Encryption: Encrypts data to prevent unauthorized access during transmission.
- Data Integrity: Ensures data remains intact and has not been tampered with during transmission.
- Authentication: Verifies the identity of the communicating parties to prevent impersonation and ensure trustworthiness.

TLS (Transport Layer Security)

TLS is the successor to SSL and provides enhanced security features and improvements over SSL. It operates in a similar manner but includes enhancements to address vulnerabilities found in SSL. TLS also supports various cryptographic algorithms and protocols for securing data transmission.

Key Components and Functions

Encryption Algorithms: SSL/TLS uses symmetric and asymmetric encryption algorithms to secure data. Symmetric encryption is used for data transmission, while asymmetric encryption ensures secure key exchange and authentication.

SSL/TLS Handshake: Before secure communication begins, SSL/TLS performs a handshake process. During the handshake, the client and server authenticate each other, negotiate encryption algorithms and keys, and establish session keys for secure data transmission.SSL Certificates: SSL/TLS certificates are issued by trusted Certificate Authorities (CAs) and contain information about the identity of the certificate holder (such as a website). They include a public key used for encryption and a private key kept secure on the server.

HTTPS (HTTP Secure): HTTPS is the secure version of HTTP. It uses SSL/TLS to encrypt HTTP requests and responses between clients and servers. HTTPS URLs are identified by "https://" in web browsers and typically display a padlock icon to indicate a secure connection.

Security and Trust: SSL/TLS ensures data security and enhances user trust by providing visual indicators of a secure connection in web browsers.

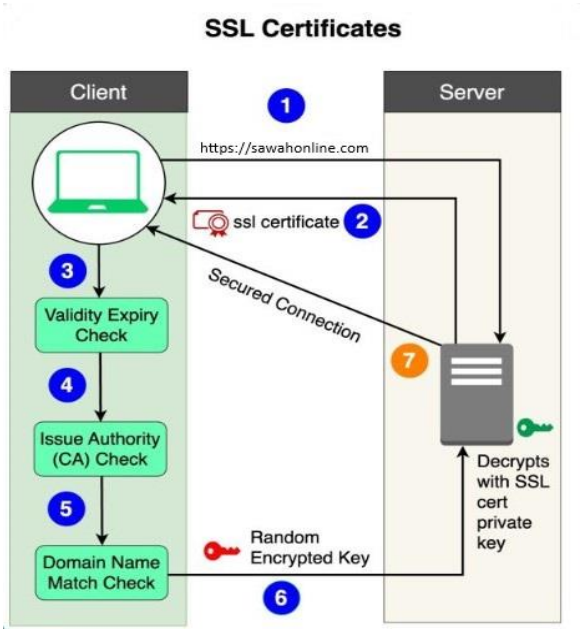


Figure 1-13 SSL Flow

1.12.3 What is Nginx

Nginx (pronounced "engine-x") is a powerful and popular open-source web server and reverse proxy server. Originally developed to solve the C10k problem (handling a large number of concurrent connections), Nginx has grown to become one of the most widely used web servers and proxy servers in the world. Here are some key aspects of Nginx:

Key Features and Functions of Nginx

Web Server: Nginx can serve static content such as HTML, CSS, JavaScript, and media files directly to clients (browsers) without needing to involve backend applications for each request. It is designed for high performance, handling many concurrent connections efficiently.

Reverse Proxy: Nginx can act as a reverse proxy server, sitting in front of web servers and forwarding client requests to those servers. It can distribute incoming requests among multiple backend servers to balance the load and improve server performance.

Load Balancer: As a reverse proxy, Nginx can perform load balancing across multiple backend servers, distributing client requests based on various algorithms (e.g., round-robin, least connections).

HTTP Server and HTTPS Proxy: Nginx supports HTTP and HTTPS protocols, allowing it to handle secure communication (SSL/TLS) with clients. It can terminate SSL/TLS connections, offloading this resource-intensive task from backend servers.

Caching: Nginx can cache static content and even dynamic content in some configurations, improving response times for frequently requested resources. This caching capability helps reduce the load on backend servers and speeds up content delivery to clients.

Security Features: Nginx includes features for access control, IP filtering, and request rate limiting to protect web applications from malicious attacks and unauthorized access. It can also be configured to handle various security headers to enhance web application security.

High Availability and Scalability: By serving as a load balancer and reverse proxy, Nginx contributes to creating highly available and scalable web architectures. It supports scaling web applications horizontally by distributing traffic across multiple servers.

Configuration Flexibility: Nginx uses a declarative configuration syntax that allows administrators to define complex server behavior and routing rules with clarity and ease.

Use Cases for Nginx

Web Hosting: Serving static content and handling high traffic websites efficiently.

Reverse Proxy: Load balancing and routing requests to backend application servers.

SSL/TLS Termination: Managing secure connections and offloading encryption tasks.

Caching: Improving performance by caching static and dynamic content.

API Gateway: Proxying and managing API requests from clients to backend services.

Microservices Architecture: Orchestrating requests between microservices and providing routing and load balancing.

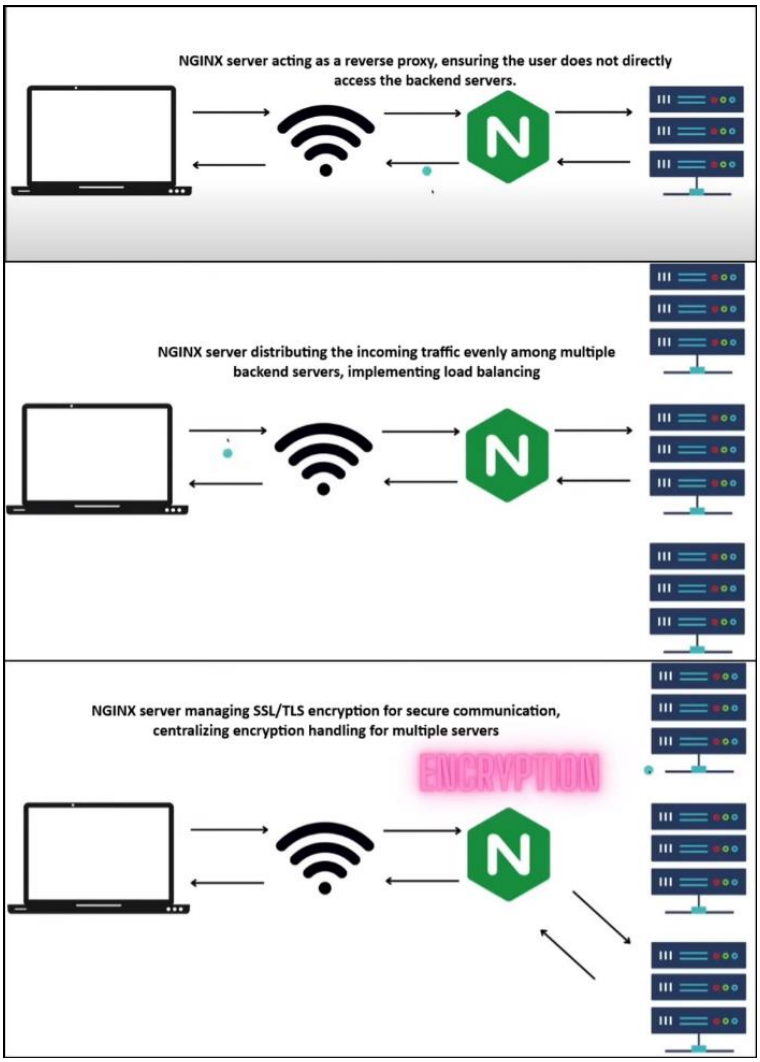


Figure 1-14 Nginx

1.12.4 Setting up our server

Compute Engine Setup

Deployed the backend application on Ubuntu.

Installed Node.js, Redis and Git for running the application.

Table 1 Machine Specs

Machine information	
Series	E2
Family	General-purpose
vCPU	4
Memory	16GB
CPU platform	Intel Broadwell
Network Bandwidth	8 Gbps
Architecture	x86/64

Node.js Service Configuration

Created a systemd service file (sawah-app.service) to ensure the Node.js server (server.js) starts automatically and runs continuously in the background.

Nginx as Reverse Proxy:

Configured Nginx to act as a reverse proxy to handle incoming HTTP and HTTPS requests.

Nginx routes all incoming requests to the backend Node.js application running on port 8000.

Ensured that Nginx routes all HTTP requests (port 80) to HTTPS (port 443) and securely communicates with the backend server using SSL/TLS.

SSL/TLS Configuration: I used Certbot, a tool from Let's Encrypt, to obtain and manage SSL certificates. This ensures that all communications between clients and the server are encrypted, providing security and privacy. Nginx is configured to route all HTTP traffic to HTTPS, ensuring secure communication.

Proxy Pass: The proxy pass directive in the Nginx configuration specifies that incoming requests should be forwarded to the Node.js application running on localhost:8000.

Headers: Additional headers are set to support WebSocket connections and handle various proxy settings.

1.12.5 Smart Tour Guide Application Request Flow

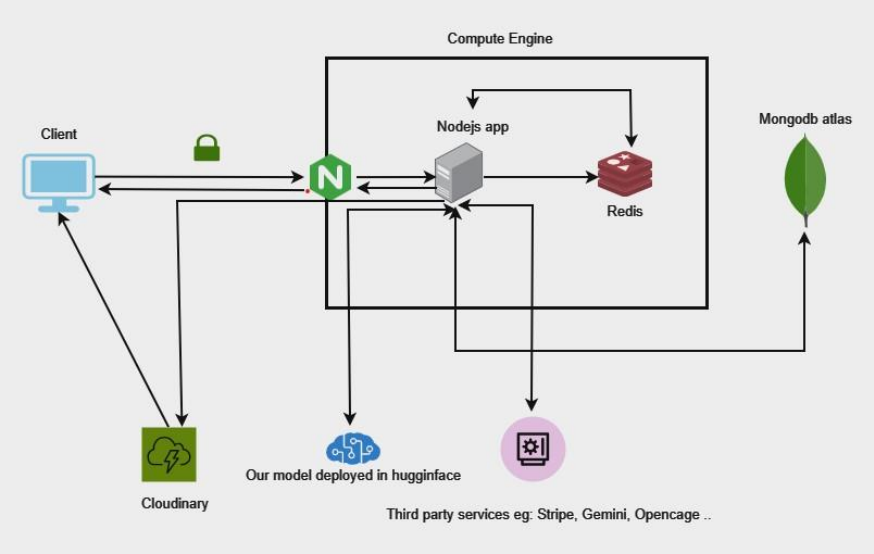


Figure 1-15 Request Flow

1. User Request: The user sends a request to the domain.
2. Nginx (Reverse Proxy): Nginx receives the request and handles SSL termination using the certificates obtained from Certbot. It then forwards the request to the Node.js server running on localhost:8000.
3. Node.js Server: The Node.js application receives the request from Nginx. If the request involves fetching data that might be cached:
    - **Cache Check:** The Node.js application first checks if the requested data is available in Redis.
    - **Cache Miss:** If the data is not found in Redis, it fetches the data from MongoDB.
    - **Cache Write:** The fetched data is then written to Redis for future requests.
    - **API Calls:** If the request involves calling third-party APIs like Gemini, or Opencage, or using Stripe after payment the Node.js application handles these calls. For instance, Opencage might be used to convert latitude and longitude into a city name before querying your model.
4. AI Model Integration: If the request involves using our AI model deployed on Huggingface, the Node.js application will make the necessary calls to this model after obtaining the required location data from Opencage.
5. Response: Once the data is fetched (either from cache, database, or a third-party service), the Node.js application sends the response back to Nginx.
6. Nginx: Nginx forwards the response back to the client.

7. Image Handling: If the response includes URLs for images stored on Cloudinary, the client can use these URLs to directly access the images.

Future Enhancements on our server

Dedicated Database Serve: Move the database to its dedicated server.

Benefit: By isolating the database on its own server, we can optimize its performance, enhance security, and improve resource allocation.

Scalable Server Architecture: Scale the application horizontally by adding multiple server instances.

Benefit: Horizontal scaling allows the application to handle increased traffic and load more efficiently by distributing requests across multiple servers.

Nginx Load Balancer Integration: Implement Nginx as a load balancer to manage incoming traffic.

Benefit: Nginx will evenly distribute requests among the available server instances, ensuring no single server becomes a bottleneck. This improves the application's availability, fault tolerance, and response times. Additionally, Nginx can handle SSL termination, caching, and other useful features to further enhance performance and security.

1.13 Future Enhancements and Scalability

1.13.1 Planning for Scalability

As the user base of your application grows, it's essential to plan for scalability to ensure the application remains responsive and reliable. Scalability involves designing the application to handle increased traffic and data volume without compromising performance.

Database Scaling

Database scaling can be achieved through vertical scaling (upgrading the hardware resources of the database server) or horizontal scaling (distributing the database across multiple servers). Horizontal scaling can be implemented using database sharding or replication techniques. Sharding involves dividing the database into smaller, manageable pieces, while replication involves creating multiple copies of the database for load balancing and redundancy.

Horizontal Scaling of Application Servers

Horizontal scaling of application servers involves adding more server instances to handle increased traffic. Load balancers can distribute incoming requests across multiple servers, ensuring no single

server becomes a bottleneck. This approach enhances the application's availability and reliability by providing redundancy and fault tolerance.

Microservices Architecture

While the current application follows a monolithic architecture, transitioning to a microservices architecture can enhance scalability and maintainability. Microservices involve breaking down the application into smaller, independent services that communicate through APIs. Each service can be developed, deployed, and scaled independently, allowing for greater flexibility and resilience.

1.13.2 Future Features and Improvements

As the application evolves, consider implementing new features and improvements to enhance functionality and user experience. Potential future enhancements include:

1.13.2.1 Enhanced User Profiles

Expanding user profiles with additional customization options, such as preferences, travel history, and saved tours, can provide a more personalized experience

1.13.2.2 Integration with Additional Services

Integrating the application with additional third-party services, such as travel booking platforms, transportation providers, and local attractions, can offer users a more comprehensive travel planning experience.

By planning for scalability and continuously enhancing the application with new features, you can ensure that your project remains competitive and meets the evolving needs of users.

1.13.2.3 Advanced Tour Customization

1.13.2.3.1 AI-Powered Tour Recommendations

Utilizing machine learning algorithms, the application will offer personalized tour recommendations based on user preferences, past bookings, and browsing behavior. This feature aims to enhance user engagement by suggesting tours that match their interests.

1.13.2.3.2 Dynamic Pricing Models

Introducing dynamic pricing models that adjust tour prices based on demand, seasonality, and booking trends. This will help optimize revenue and provide competitive pricing for users.

1.13.2.4 Improved Security and Compliance

1.13.2.4.1 Two-Factor Authentication (2FA)

To enhance account security, we plan to implement two-factor authentication (2FA). Users will be required to verify their identity using a second method, such as a mobile app or SMS code, in addition to their password.

1.13.2.5 Expanded Payment Options

1.13.2.5.1 Multiple Payment Gateways

Integrating multiple payment gateways, such as PayPal, Apple Pay, and Google Wallet, to provide users with more payment options. This will improve the convenience and flexibility of the payment process.

1.13.2.6 Localization

1.13.2.6.1 Multi-Language Support

Expanding the application to support multiple languages, making it accessible to a broader audience. This will involve translating content and providing language-specific features to enhance user experience.