5CLCsCan

Gogle
Software Architecture Document

Version 2.0

Revision History

Date	Version	Description	Author
26/07/2024	1.0	Infomation and diagram fill in	Tuan Kiệt, Khuong Duy
28/07/2024	1.1	Review & Identify	Duy Anh
06/06/2024	2.0	Edit diagrams follow the feeback	Khuong Duy, Minh Man

Table of Contents

Software Architecture Document

1.Introduction

The introduction of the Software Architecture Document provides an overview of the entire document, ensuring that readers understand the purpose, scope, and context of the architecture being described. This section sets the stage for the detailed architecture descriptions that follow.

1.1.Purpose

The purpose of this Software Architecture Document is to outline the architectural design and structure of the Gogle Platform, a website that leverages AI technology to find, suggest, and create local trips. This document serves as a guide for stakeholders, developers, and other team members to understand the system's high-level design, components, and interactions. It ensures that the architecture aligns with the overall business objectives, requirements, and constraints of the Gogle Platform.

1.2.Scope

This document covers the architectural aspects of the Gogle Platform, including its key components, modules, and their interactions. It provides an abstract view of the system, highlighting the main architectural decisions, patterns, and frameworks used. The scope includes:

- High-level architecture overview of the Gogle Platform
- Detailed component descriptions and their responsibilities
- Inter-component relationships and data flow within the platform
- Technology stack and frameworks utilized
- Deployment and operational considerations specific to the Gogle Platform

1.3.References

A comprehensive list of all documents referenced within the Software Architecture document is presented in this subsection. Each reference is meticulously cataloged, including its title, report number (if applicable), date of publication, and the publishing organization. Additionally, sources for obtaining the referenced documents are provided to facilitate accessibility and further exploration, if necessary.

2.Architectural Goals and Constraints

- Architectural Constraints

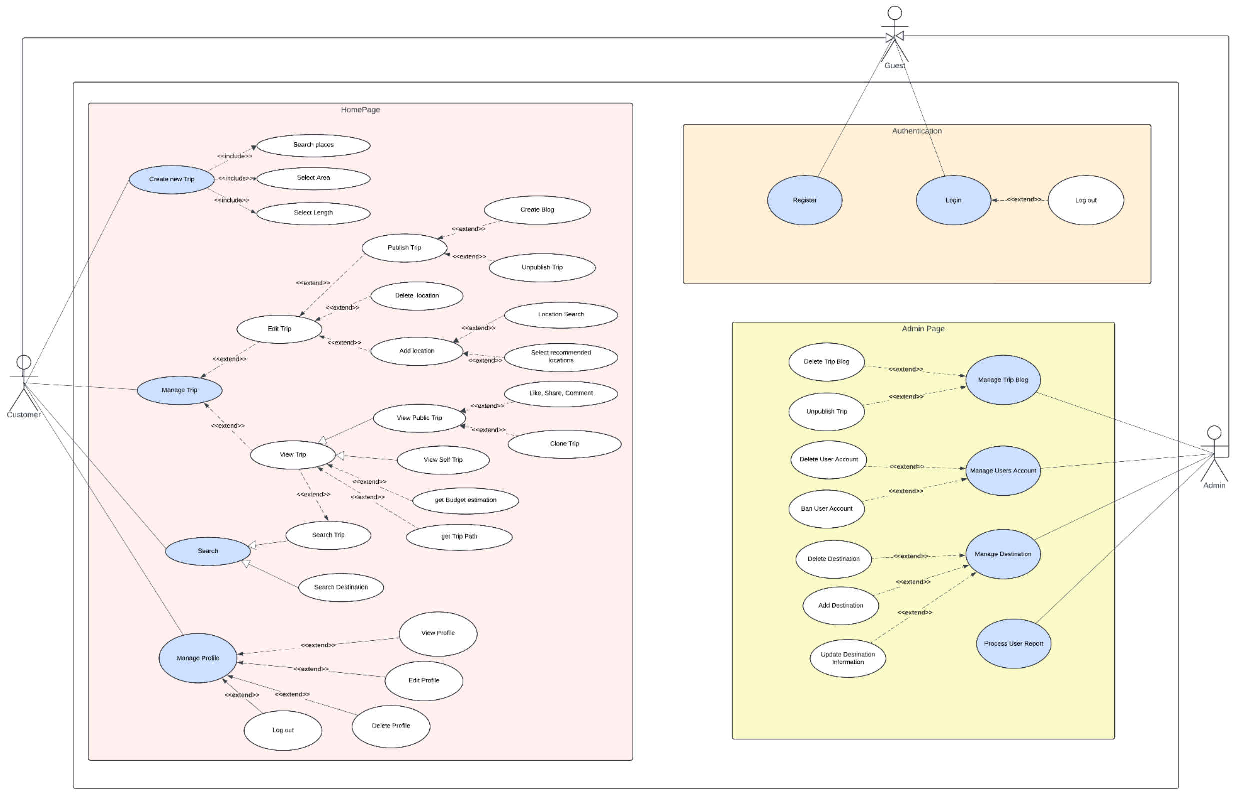
- **Running Environment**:
  - **Browser**: Google Chrome, version 116.0.5845.14 or newer.
  - **System**: Windows 10–11 with no more than 1GB of RAM in use.
- **Performance**: The website must serve at least 50 users at any given time, with services responding in no more than 10 seconds.
- **Programming Languages**: HTML, CSS, JavaScript.
- **Data**: The system uses MongoDB to securely store user account and product information.
- **User Interface Design**: Designs must be simple and clear to ensure a comfortable user experience.

- Additional Constraints

- **Security**: Must protect user data through encryption and secure authentication methods.
- **Maintainability**: The design must facilitate easy updates, bug fixes, and enhancements.

- **Compatibility**: The system must be compatible with major web browsers (Chrome, Firefox, Safari, Edge) and support responsive design for various devices (desktops, tablets, smartphones).
- **Portability**: The system should be easily transferable and deployable across different environments and cloud platforms without significant modifications.
- **Compliance**: The system must comply with relevant industry standards and regulations, such as GDPR for data protection and PCI-DSS for payment processing.
- **Backup and Recovery**: The system must implement automated daily backups with a recovery point objective (RPO) of 24 hours and a recovery time objective (RTO) of 2 hours in case of data loss.
- **Logging and Monitoring**: The system must have comprehensive logging and monitoring capabilities to track user activities, system performance, and errors, providing alerts for unusual or critical conditions.
- **Localization and Internationalization**: The system should support multiple languages and regional settings, including date formats, currency symbols, and time zones, allowing users to interact in their preferred language and format.
- **Energy Efficiency**: The system should be optimized for energy efficiency, minimizing server and resource usage to reduce environmental impact.
- **Accessibility**: The system must adhere to accessibility standards (e.g., WCAG 2.1) to ensure usability by individuals with disabilities, including support for screen readers and keyboard navigation.

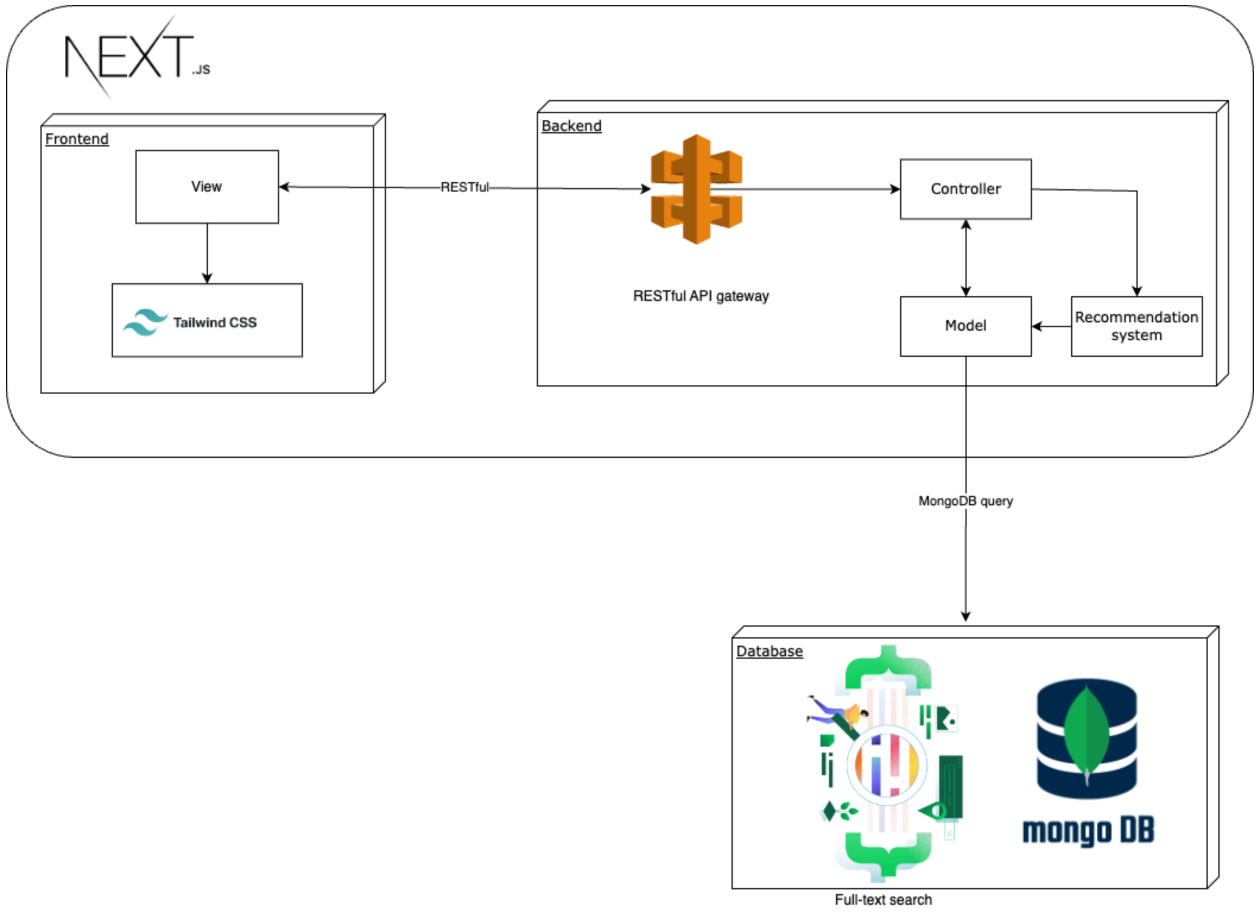
3.Use-Case Model



The Gogle platform is designed to streamline local trip planning through AI-powered recommendations, offering a range of functionalities to different user roles. Customers can create, manage, and search for trips, utilizing features such as selecting time, area, length, and category, as well as publishing and editing trips. Guests can register and login to access these features, while Admins have robust control over managing trip content, user accounts, and content web pages.

4.Logical View

4.1.Architecture diagram



The architecture diagram illustrates a system for recommending places for dating or hanging out, using Next.js for the frontend and a combination of various backend components and a MongoDB database. Here is a detailed description of each component and their connections:

Frontend:

1. **Next.js**: A React framework for building server-side rendered applications. It provides a structured way to develop web applications with built-in routing and performance optimizations.
2. **View**: This represents the user interface layer of the application, where users interact with the system. It receives and displays data from the backend.
3. **Tailwind CSS**: A utility-first CSS framework used to style the frontend components. It ensures that the user interface is responsive and visually appealing.

Backend:

1. **RESTful API Gateway**: This component acts as an entry point for client requests. It exposes various endpoints that the frontend can call to interact with the backend services. It follows REST principles, making the API stateless, scalable, and easy to maintain.
2. **Controller**: The controller handles incoming requests from the API gateway. It processes the requests, interacts with the model and the recommendation system, and sends back the appropriate response.
3. **Model**: This component represents the application's data structure. It handles the business logic and communicates with the database to perform CRUD (Create, Read, Update, Delete) operations.
4. **Recommendation System**: This subsystem processes user preferences and historical data to suggest suitable places for dating or hanging out. It leverages algorithms to provide personalized recommendations.

Database:

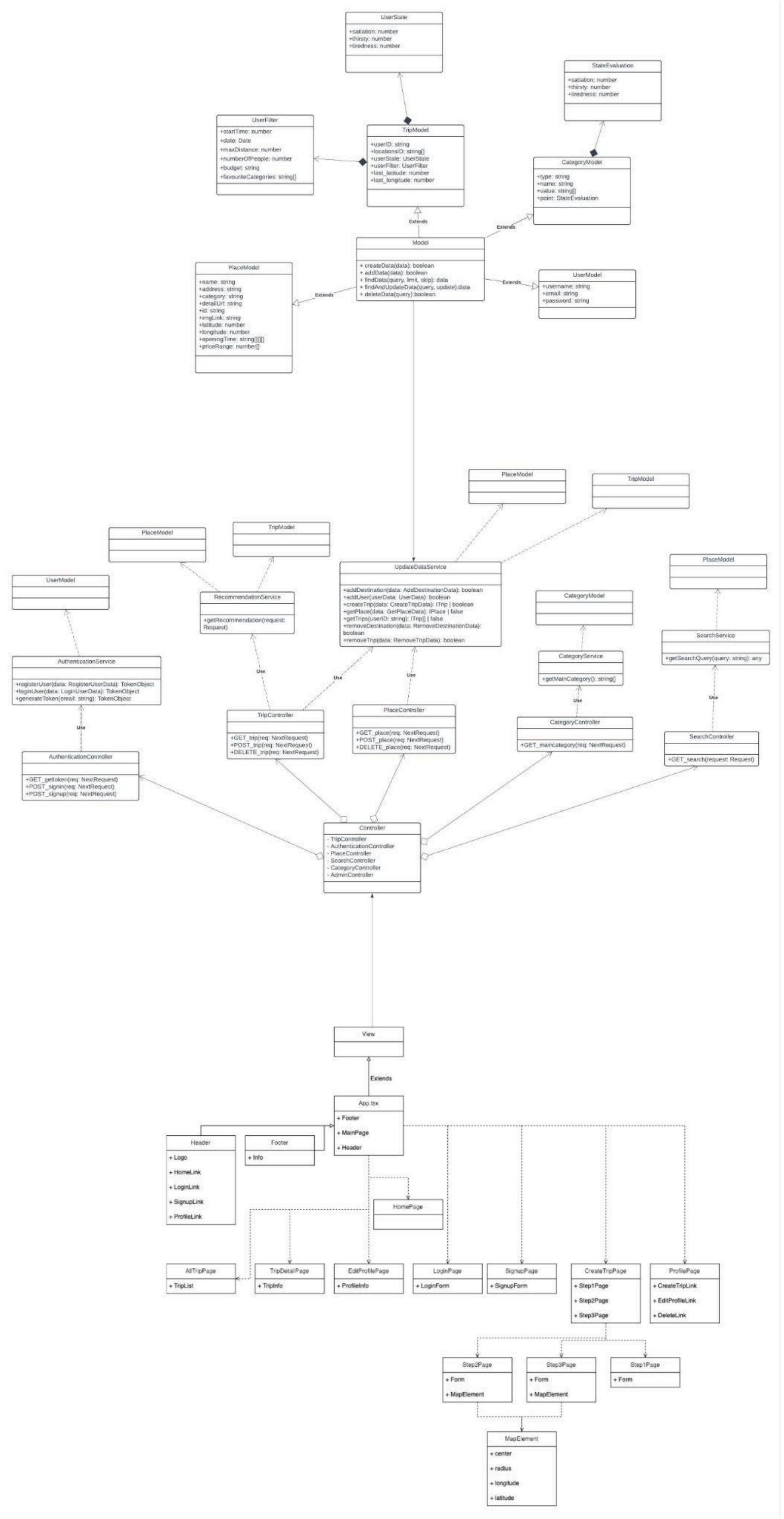
1. **MongoDB**: A NoSQL database used to store the application's data. It is chosen for its flexibility in handling unstructured data and its powerful querying capabilities, which are essential for full-text search functionalities.
2. **Full-Text Search**: This feature of MongoDB allows for efficient searching within text fields in the database. It is utilized to quickly retrieve relevant places based on user queries.

Connections:

- **Frontend to Backend (RESTful)**: The frontend View communicates with the RESTful API Gateway to request data or send user inputs. These requests are processed by the controller, which in turn interacts with the model and the recommendation system.
- **Backend to Database (MongoDB query)**: The model communicates with MongoDB to perform various data operations, such as retrieving user data, storing user preferences, or fetching recommended places.
- **Recommendation System to Model**: The recommendation system interacts with the model to fetch necessary data and provide recommendations based on user inputs and stored data.

This architecture ensures a modular and scalable system where each component has a specific responsibility. The use of Next.js for the frontend, combined with a RESTful API, a robust recommendation system, and MongoDB, allows for a responsive, efficient, and user-friendly application.

4.2. Connection overview



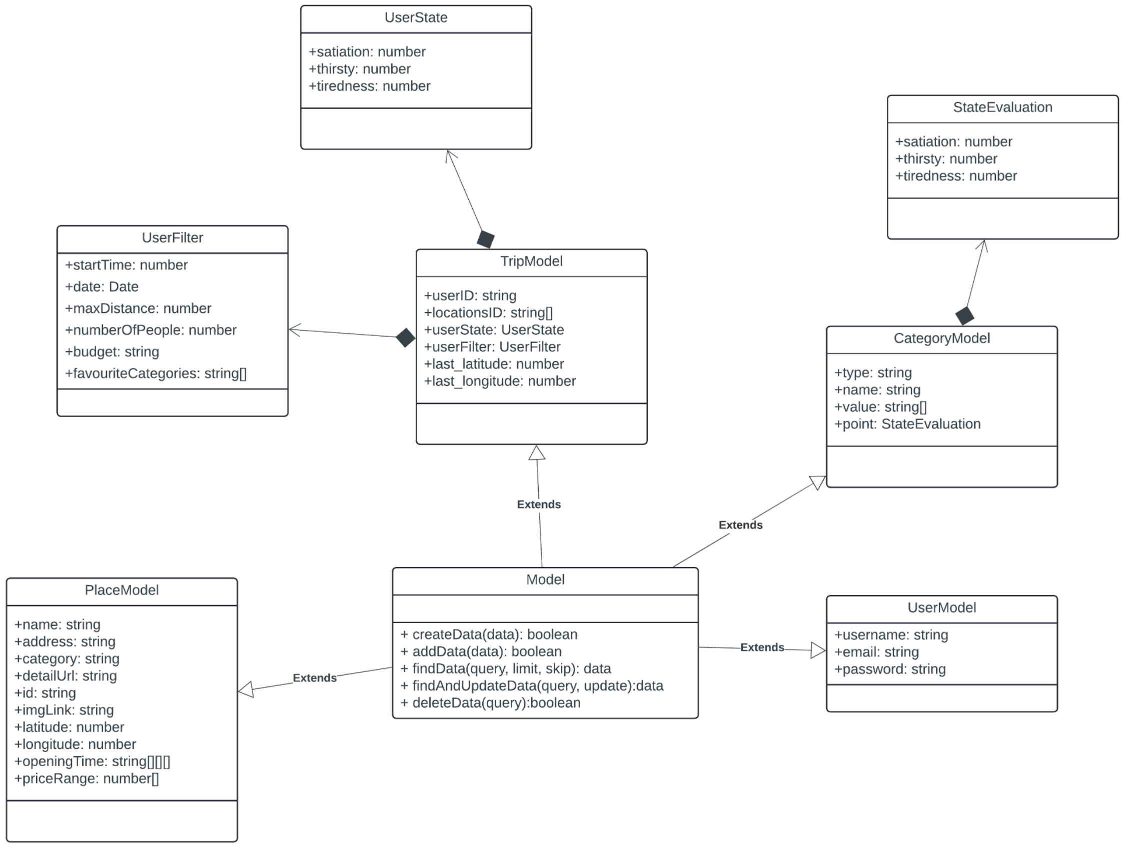
4.3. Component: Model

Description:

The Model component is responsible for managing and storing data within an application**.** It acts as a bridge between the user interface (UI) and the database, handling data retrieval, creation, modification, and deletion.

- UserState: Captures user's current state, including satiation, thirstiness, and tiredness levels.
- UserFilter: Defines user preferences for a trip, including start time, date, maximum distance, number of people, budget, and preferred categories.
- TripModel:Represents a trip, containing user ID, locations visited, user state, user filter, and potentially other details.
- CategoryModel: Defines categories for locations, with name, value, and an associated StateEvaluation.
- PlaceModel: Represents a place, including name, address, category, details URL, ID, image link, latitude, longitude, opening time, and price range.
- UserModel: Stores user information, such as username, email, and password.
- Model: An abstract class providing common CRUD (Create, Read, Update, Delete) operations for data management.
- StateEvaluation: Likely used to assess how well a location matches a user's current state.

Class Diagram:



Key-class explanation:

- **Model Class**
- Purpose
  - The Model class serves as a generic base class for data management operations, providing common functionalities for creating, adding, finding, updating, and deleting data. It acts as an abstraction layer, separating the core data operations from specific data models.
- Methods
  - createData(data): boolean
    - Creates a new data record based on the provided data.
    - Returns a boolean indicating success or failure.
  - addData(data): boolean
    - Adds an existing data record to a collection or database.
    - Returns a boolean indicating success or failure.
  - findData(query, limit, skip): data
    - Retrieves data based on the specified query, limit, and skip parameters.
    - Returns the found data.
    - The `query` parameter defines the search criteria.
    - The `limit` parameter specifies the maximum number of results to return.

        - The `skip` parameter determines the offset for pagination.
    - findAndUpdateData(query, update): data
        - Finds data based on the query and updates matching records with the provided update data.
        - Returns the updated data.
    - deleteData(query): boolean
        - Deletes data matching the specified query.
        - Returns a boolean indicating success or failure.

- **UserState Class**

- **Purpose:**
    - The UserState class is designed to encapsulate a user's current physical and psychological condition. It serves as a data structure to store and manage attributes related to a user's state, such as hunger, thirst, and tiredness levels. This information can be used by other components of the system to personalize recommendations, experiences, or interactions based on the user's current needs.
- **Attribute:**
    - Satiation:
        - Represents the user's level of fullness or satisfaction.
        - Typically a numerical value (e.g., integer or float).
        - Range might be defined (e.g., 0-100) to indicate a scale from extremely hungry to completely full.
    - Thirsty:
        - Represents the user's level of thirst.
        - Similar to satiation, it's typically a numerical value.
        - Define a suitable range (e.g., 0-100) to indicate thirst level.
    - Tiredness:
        - Represents the user's level of fatigue or exhaustion.
        - Again, a numerical value is common.
        - Define a suitable range (e.g., 0-100) to indicate energy level.

- **UserFilter class:**

- **Purpose**
- The **UserFilter** class encapsulates a set of criteria used to filter and refine search results or recommendations based on user preferences. It defines parameters such as desired start time, date, maximum distance, number of people, budget, and preferred categories.
- **Attributes**
- **startTime:**
    - Represents the preferred start time for an activity or trip.
    - Typically a numerical value representing time in a specific format (e.g., milliseconds since epoch, hours in a day).
- **date:**
    - Represents the desired date for an activity or trip.
    - Typically a Date object or a string representing a date in a specific format.
- **maxDistance:**
    - Represents the maximum distance the user is willing to travel.
    - A numerical value representing distance (e.g., meters, kilometers, miles).
- **numberOfPeople:**
    - Represents the number of people involved in the activity or trip.
    - A numerical value (integer).

- **budget:**
  - Represents the user's budget for the activity or trip.
  - A string value to allow for flexible budget representations (e.g., "low", "medium", "high").
- **favoriteCategory:**
  - Represents a list of categories the user prefers.
  - A string array to store category names..

- **TripModel Class:**
- **Purpose**
- The **TripModel** class represents a user's trip or journey, encapsulating essential information about the trip, including the user who initiated it, the locations visited, the user's state during the trip, the filters applied to plan the trip, and the user's last known location.
- **Attributes**
- **userID:**
  - A unique identifier for the user associated with the trip.
  - Typically a string or a user object reference.
- **locationsID:**
  - An array of identifiers for the locations visited or planned to be visited during the trip.
  - The data type depends on the specific location identifier used (e.g., string, integer, object reference).
- **userState:**
  - A reference to a UserState object, capturing the user's physical and psychological condition at a specific point in the trip.
  - This allows for tracking changes in user state during the journey.
- **userFilter:**
  - A reference to a UserFilter object, representing the criteria used to plan the trip.
  - This provides insights into the user's preferences and constraints.
- **last_latitude:**
  - The latitude of the user's last known location.
  - A numerical value representing the latitude coordinate.
- **last_longitude:**
  - The longitude of the user's last known location.
  - A numerical value representing the longitude coordinate.

- **PlaceModel Class:**
- **Purpose**
- The **PlaceModel** class represents a physical location, providing essential information about the place, including its name, address, category, details, unique identifier, image, geographic coordinates, opening hours, and price range.
- **Attributes**
- **name:**
  - The name of the place.
  - Typically a string.
- **address:**
  - The physical address of the place.
  - Typically a string.
- **category:**

        - The category or type of the place (e.g., restaurant, hotel, attraction).
        - Typically a string.
    - **detailUrl:**
        - A URL linking to more detailed information about the place.
        - Typically a string.
    - **id:**
        - A unique identifier for the place.
        - Typically a string or a numerical value.
    - **imgLink:**
        - A URL to an image of the place.
        - Typically a string.
    - **latitude:**
        - The latitude coordinate of the place.
        - A numerical value.
    - **longitude:**
        - The longitude coordinate of the place.
        - A numerical value.
    - **openingTime:**
        - A complex data structure representing the place's opening hours.
        - A three-dimensional array where:
            - The first dimension represents days of the week (e.g., 0 for Monday, 1 for Tuesday).
            - The second dimension represents different opening time periods (e.g., morning, afternoon, evening).
            - The third dimension represents the start and end times for each period (e.g., ["10:00", "12:00"]).
    - **priceRange:**
        - An array of numerical values representing the price range of the place.
        - The array can contain two or more values to indicate the minimum and maximum prices.

- **CategoryModel Class**
- **Purpose**
    - The **CategoryModel** class defines a category or classification for places, providing information about the category type, name, associated values, and a potential evaluation point related to user state.
- **Attributes**
    - **type:**
        - The type or category of the category itself.
        - Typically a string.
    - **name:**
        - The name of the category.
        - Typically a string.
    - **value:**
        - An array of string values representing subcategories or specific options within the category.
        - Can be used for more filtering or categorization.
    - **point:**
        - A reference to a StateEvaluation object, potentially indicating how well places in

this category match a user's current state.
        - This could be used for personalized recommendations or filtering.
- **UserModel Class**
- **Purpose**
  - The **UserModel** class represents a user of the system, storing essential information for user identification and authentication.
- **Attributes**
  - **username:**
    - A unique identifier for the user.
    - Typically a string.
  - **email:**
    - The user's email address.
    - Typically a string.
  - **password:**
    - The user's password.
    - **Important:** For security reasons, passwords are never stored in plain text. They are hashed using a cryptographic algorithm.
- **StateEvaluation Class**
- **Purpose**
  - The **StateEvaluation** class represents the changes that affect the UserState if a place's category is chosen.
- **Attributes**
  - **satiation:**
    - **Type:** int
    - **Description:** Represents the level of hunger satisfaction provided by the place's category. A positive value indicates an increase in satiation (decrease in hunger), while a negative value might indicate a decrease in satiation.
- **thirsty:**
  - **Type:** int
  - **Description:** Represents the level of thirst satisfaction provided by the place's category. A positive value indicates an increase in thirst satisfaction (decrease in thirst), while a negative value indicates an increase in thirst.
- **tiredness:**
  - **Type:** int
  - **Description:** Represents the impact on tiredness after visiting the place. A positive value indicates an increase in tiredness (making the user more tired), while a negative value indicates a reduction in tiredness (making the user feel more rested).

4.4. Component: View

The view component displays related information in a structured and logical way so that the user can understand and interact with the whole system. It acts as a connector between the end user and the backend systems and services. In our application, this component also supports serializing data to match with other systems, providing a better user experience.

View Included:

1. **Home Page**

This view serves as a landing page for our application. It displays information about the project, including its aims and objectives. There are also other sections such as team introduction, project images, in-app examples, and testimonials. This page acts as a starting point for new users when they find out our project and have some questions about its functionality.

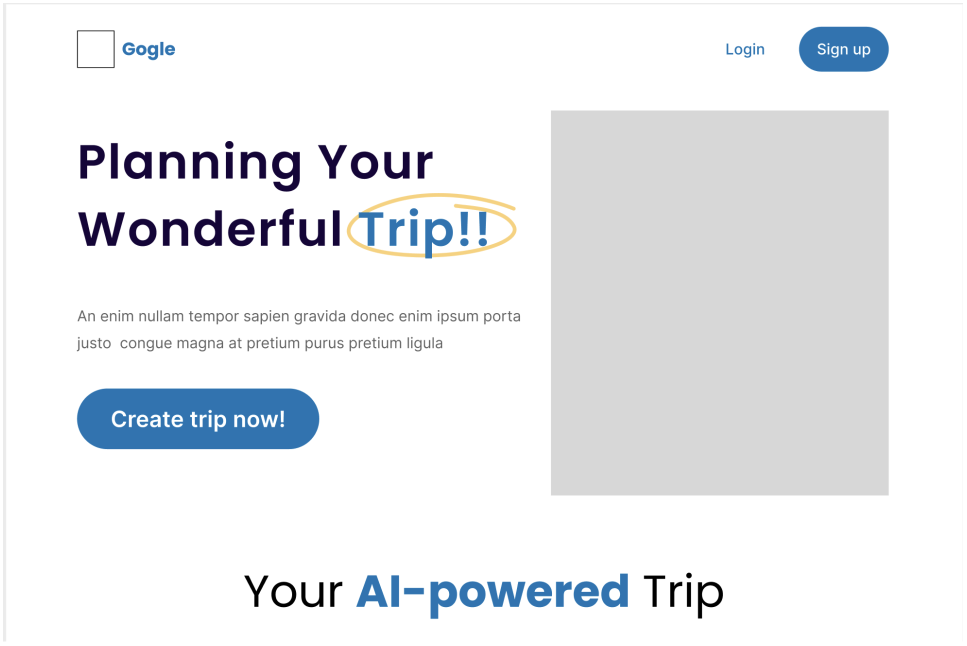

Gogle
Login
Sign up
Planning Your Wonderful Trip!!
An enim nullam tempor sapien gravida donec enim ipsum porta justo congue magna at pretium purus pretium ligula
Create trip now!
Your AI-powered Trip

Image 1. Homepage screenshot

2. Login Page

This view displays a curated form containing different fields and labels to direct users for the login process. It handles input validation: minimum characters for password, valid email,..., and allows users to view their password (only if needed). The page supports automatic redirection after successful login, and displays proper messages on errors.

This page also contains some handy links in case the users don't have an account yet, or they have forgotten their password. These links enhance the overall user experience, and create a complete flow for the authentication process.

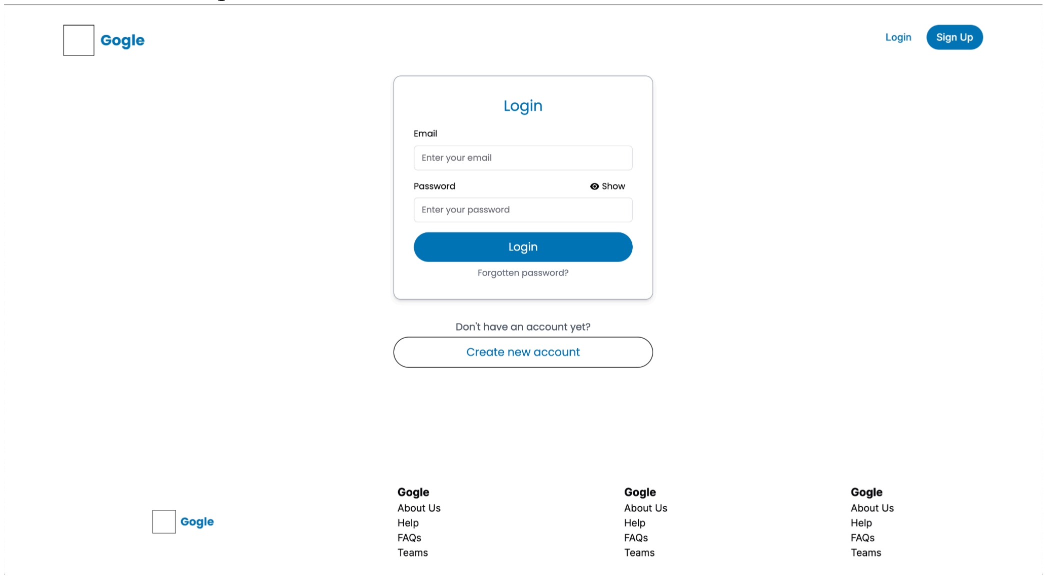
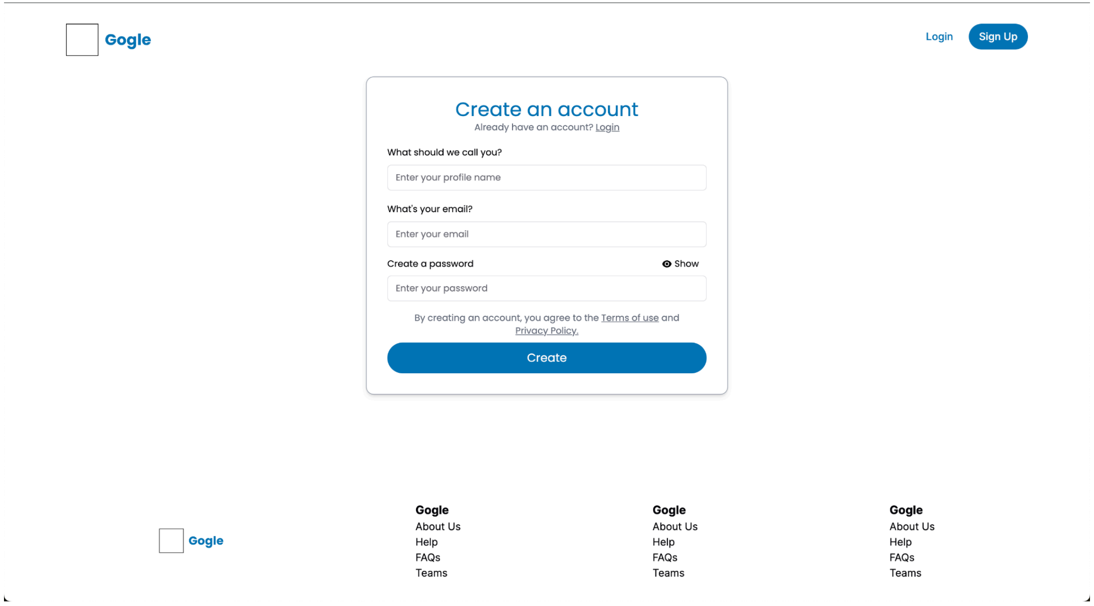


Image 2. Sign In Page

3. Signup Page

This view displays a form that contains different fields and labels to direct users for the signup process. It handles input validation: minimum characters for password, valid email,..., and allows users to view their password (only if needed). The page supports automatic redirection after successful signup, and displays proper messages on errors.

This page also contains some handy links in case the users already have an account. These links enhance the overall user experience, and create a complete flow for the authentication process.



Create an account
Already have an account? Login

What should we call you?
Enter your profile name

What's your email?
Enter your email

Create a password
Show
Enter your password

By creating an account, you agree to the Terms of use and Privacy Policy.

Create

Image 3. Sign Up Page

4. Dashboard Page

This page allows authenticated users to view their created trips, and some ongoing plans in a logical way. There're also some options on each trip information, helping users to perform actions on their existing trips. Moreover, there's also a link to the user settings page, which allows users to change their credentials, as well as some public information.

5. Create Trip Page

This page contains step-by-step forms, each step allows users to fill in the trip information, as well as filter out unwanted suggestions. There's also a map to give users an overview of their upcoming trip. It also supports automatically saving progress, even without Internet connection. This feature significantly increases the user experience, allowing them to come back and continue their trip's plan at any time.

6. Trip Detail Page

This page allows users to have a summary on the planned trip. In this view, users can quickly identify places, and detailed routes from their starting point to the first destination, and so on. Users can also edit selected places, and remove places to fit their plan better.

Note: there're more pages to be developed, and has been included in the below diagram.

Class Diagram:

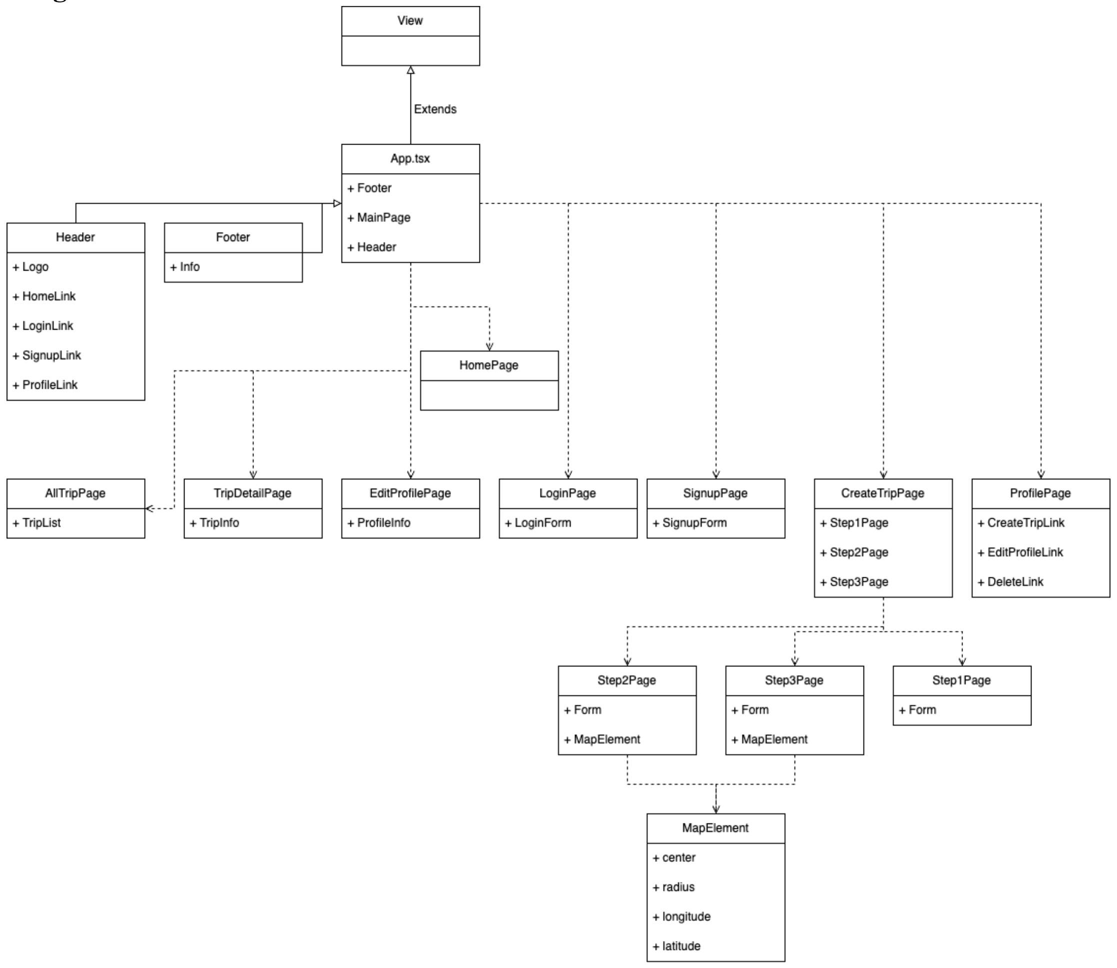


Image 4. Overview of View component

4.5.Component: Controller

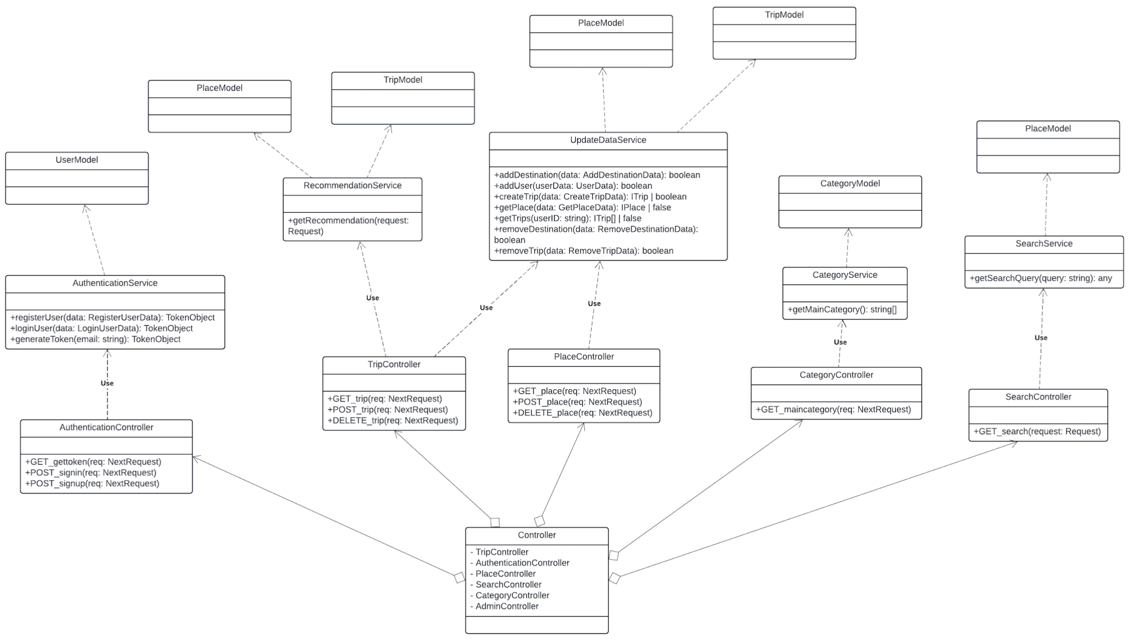
Description:

The controller component manages the interactions between the client requests and the services that handle the business logic. It ensures that the appropriate services are called and that the correct data is returned to the client. This component is composed of multiple controllers, each responsible for handling specific types of requests.

Controllers Included:

1. AuthenticationController
2. TripController
3. PlaceController
4. CategoryController
5. SearchController

Class Diagram:



PlaceModel
TripModel
PlaceModel
TripModel
UpdateDataService
+addDestination(data: AddDestinationData): boolean
+addUser(userData: UserData): boolean
+createTrip(data: CreateTripData): ITrip | boolean
+getPlace(data: GetPlaceData): IPlace | false
+getTrips(userID: string): ITrip[] | false
+removeDestination(data: RemoveDestinationData): boolean
+removeTrip(data: RemoveTripData): boolean
PlaceModel
UserModel
RecommendationService
+getRecommendation(request: Request)
CategoryModel
SearchService
+getSearchQuery(query: string): any
AuthenticationService
+registerUser(data: RegisterUserData): TokenObject
+loginUser(data: LoginUserData): TokenObject
+generateToken(email: string): TokenObject
CategoryService
+getMainCategory(): string[]
Use
Use
Use
Use
Use
Use
TripController
+GET_trip(req: NextRequest)
+POST_trip(req: NextRequest)
+DELETE_trip(req: NextRequest)
PlaceController
+GET_place(req: NextRequest)
+POST_place(req: NextRequest)
+DELETE_place(req: NextRequest)
CategoryController
+GET_maincategory(req: NextRequest)
SearchController
+GET_search(request: Request)
AuthenticationController
+GET_gettoken(req: NextRequest)
+POST_signin(req: NextRequest)
+POST_signup(req: NextRequest)
Controller
- TripController
- AuthenticationController
- PlaceController
- SearchController
- CategoryController
- AdminController

Key-class Explanation:

1. AuthenticationController:

Handles all authentication-related requests, such as user login, signup, and token generation.

- Methods:
  - `+GET_gettoken(req: NextRequest)`: Generates and returns a new authentication token.
  - `+POST_signin(req: NextRequest)`: Authenticates a user based on provided credentials.
  - `+POST_signup(req: NextRequest)`: Registers a new user in the system.

2. TripController:

Manages trip-related requests, such as creating, retrieving, and deleting trips.

- Methods:
  - `+GET_trip(req: NextRequest)`: Retrieves details of a specific trip.
  - `+POST_trip(req: NextRequest)`: Creates a new trip based on provided data.
  - `+DELETE_trip(req: NextRequest)`: Deletes an existing trip.

3. PlaceController:

Handles requests related to places, including retrieving, adding, and removing places.

- Methods:
  - `+GET_place(req: NextRequest)`: Retrieves details of a specific place.
  - `+POST_place(req: NextRequest)`: Adds a new place to the system.
  - `+DELETE_place(req: NextRequest)`: Removes an existing place from the system.

4. CategoryController:

Manages category-related requests, primarily retrieving the main categories.

- Methods:
  - `+GET_maincategory(req: NextRequest)`: Retrieves the main categories available.

5. SearchController:

Handles search queries within the system.

- Methods:
  - `+GET_search(request: Request)`: Processes search queries and returns relevant results.

Integration with Services:

In this section, we detail the specific roles and functionalities of each service that the controllers interact with. Each service encapsulates a distinct set of business logic and operations, ensuring a clean separation of concerns and promoting maintainability.

AuthenticationService

Role: Manages user authentication, registration, and token generation.

- registerUser(data: RegisterUserData): TokenObject
  - Registers a new user and returns a token object.
- loginUser(data: LoginUserData): TokenObject
  - Authenticates a user with the provided credentials and returns a token object.
- generateToken(email: string): TokenObject
  - Generates a new authentication token for the given email.

Interacts With:

- AuthenticationController: For handling requests related to user sign-in, sign-up, and token management.

UpdateDataService

Role: Manages updates to trip and place data, including adding, removing, and retrieving destinations and trips.

- addDestination(data: AddDestinationData): boolean
  - Adds a new destination to a user's trip.
- addUser(userData: UserData): boolean
  - Adds a new user to the system.
- createTrip(data: CreateTripData): ITrip | boolean
  - Creates a new trip with the provided data and returns the trip object or a boolean status.
- getPlace(data: GetPlaceData): IPlace | false
  - Retrieves place information based on the provided data.
- getTrips(userID: string): ITrip[] | false
  - Retrieves all trips associated with the specified user ID.
- removeDestination(data: RemoveDestinationData): boolean
  - Removes a destination from a user's trip.
- removeTrip(data: RemoveTripData): boolean
  - Deletes an existing trip based on the provided data.

Interacts With:

- TripController: For creating, retrieving, and deleting trips.
- PlaceController: For retrieving, adding, and removing places.

CategoryService

Role: Manages operations related to categories, including retrieving the main categories.

- getMainCategory(): string[]
  - Retrieves the main categories available in the system.

Interacts With:

- CategoryController: For handling requests to retrieve main categories.

SearchService

Role: Handles search queries within the system, ensuring that relevant data is retrieved based on the search parameters.

- getSearchQuery(query: string): any
  - Processes the provided search query and returns the relevant results.

Interacts With:

- SearchController: For handling search requests and returning the results to the client.

-

5. Deployment

Overview

The deployment strategy for the website leverages Next.js for both frontend and backend functionalities, and MongoDB for the database layer.

Vercel Managed Deployment (Frontend and Backend)

The Next.js application, which includes both the frontend and backend, is deployed using Vercel. Vercel is a cloud platform for static sites and Serverless Functions that fits perfectly with Next.js. It provides features such as automatic HTTPS, continuous deployment from Git, and global CDN.

Deployment steps:

1. Push the code to the Git repository.
2. Vercel automatically triggers the deployment upon detecting the new push.
3. Vercel builds the Next.js application and deploys it across its CDN network.

Alternatively, for self-hosting, the Next.js application can be deployed on a Node.js server, as a Docker image, or even as static HTML files using `next start`. All Next.js features are supported when deploying using `next start`.

MongoDB database

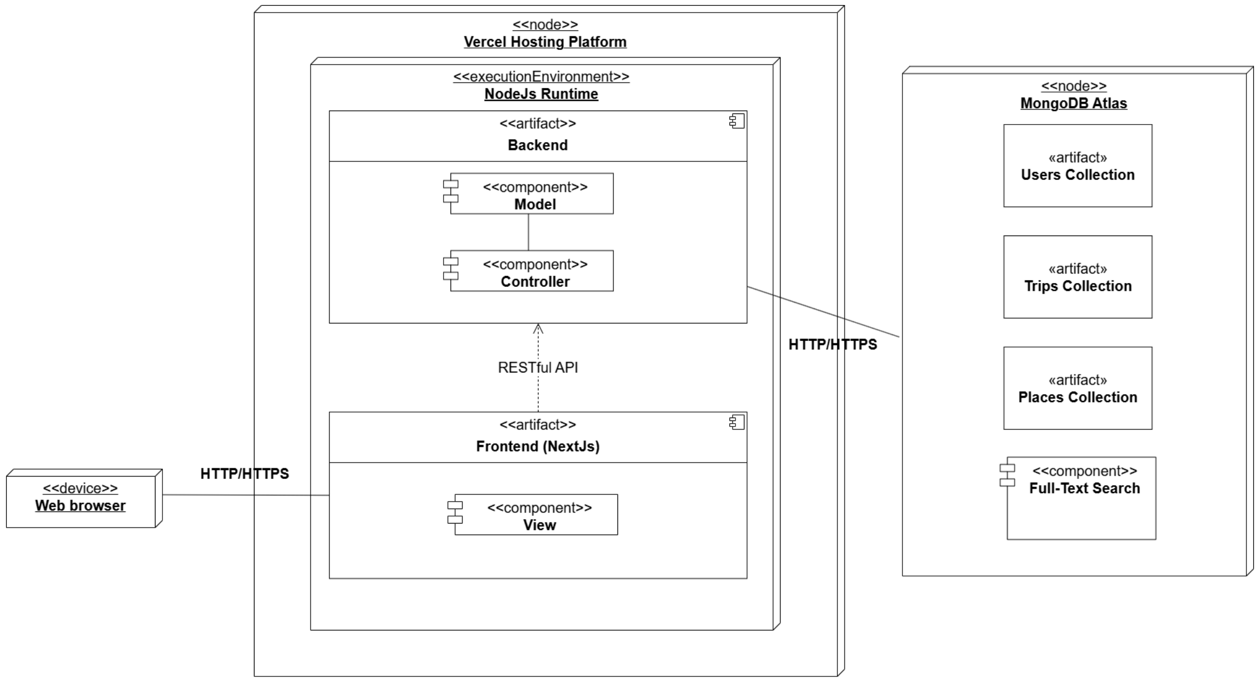
The MongoDB database is hosted on MongoDB Atlas, a fully-managed cloud database developed by the same people that build MongoDB. MongoDB Atlas handles all the complexity of deploying, managing, and healing your deployments on the cloud service provider of your choice (AWS, Azure, and GCP).

Deployment steps:

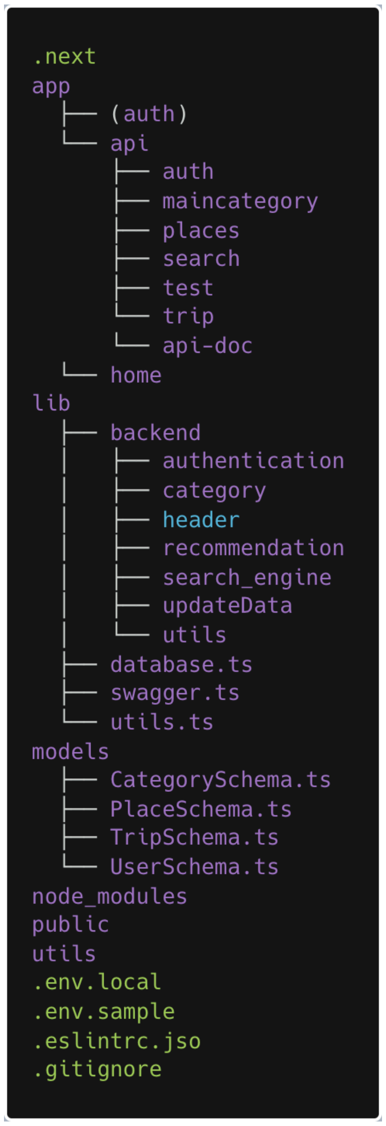
1. Create a new cluster on MongoDB Atlas.
2. Configure the security settings and network access.
3. Load the data into the database.
4. Connect the Next.js application to the MongoDB Atlas cluster using the provided connection string.

This deployment strategy ensures that our application is scalable, reliable, and secure.

Deployment diagram



6.Implementation View



```
.next
app
  ├── (auth)
  └── api
      ├── auth
      ├── maincategory
      ├── places
      ├── search
      ├── test
      └── trip
      └── api-doc
  └── home
lib
  ├── backend
  │   ├── authentication
  │   ├── category
  │   ├── header
  │   ├── recommendation
  │   ├── search_engine
  │   ├── updateData
  │   └── utils
  ├── database.ts
  ├── swagger.ts
  └── utils.ts
models
  ├── CategorySchema.ts
  ├── PlaceSchema.ts
  ├── TripSchema.ts
  └── UserSchema.ts
node_modules
public
utils
.env.local
.env.sample
.eslintrc.jso
.gitignore
```

The provided folder structure is organized to follow the Next.js conventions, ensuring a clear separation of concerns and modularity. Here's an explanation of each part of the structure:

Root Level

- **.next**: This folder is created automatically by Next.js when the project is built. It contains the compiled output and is used during server-side rendering and client-side hydration.
- **node_modules**: This directory contains all the npm packages installed for the project.
- **public**: This folder is for static assets like images, which can be accessed directly via the URL path.

Application Code (`app` Directory)

- **app**: This is the main directory containing all application-specific code.
  - **(auth)**: This folder contains components or pages related to authentication. The parentheses indicate that this might be an optional feature or route.
  - **api**: This directory contains API route handlers, structured according to Next.js conventions.
    - **auth**: Handlers for authentication-related API endpoints.
    - **maincategory**: API endpoints related to main categories.
    - **places**: API endpoints related to places.
    - **search**: API endpoints related to search functionality.
    - **test**: API endpoints for testing purposes.
    - **trip**: API endpoints related to trips.
    - **api-doc**: This might contain API documentation routes or Swagger documentation.
  - **home**: This folder likely contains components or pages for the home route.

Config and Styles

- **favicon.ico**: The favicon for the website.
- **globals.css**: Global CSS styles that apply to the entire application.
- **layout.tsx**: The layout component for wrapping pages to provide a consistent look and feel.
- **page.tsx**: The main page component.

Components and Hooks

- **components**: This folder contains reusable React components used throughout the application.
- **hooks**: This directory contains custom React hooks for encapsulating reusable logic.

Library Code (`lib` Directory)

- **lib**: This directory contains backend logic and utilities.
  - **backend**: Contains backend-specific logic, possibly organized by feature or domain.
    - **authentication**: Code related to user authentication.
    - **category**: Code related to category management.
    - **header**: Logic related to the header component or data.
    - **recommendation**: Recommendation engine logic.
    - **search_engine**: Search engine related logic.
    - **updateData**: Code for updating data.
    - **utils**: Utility functions used by backend code.
  - **database.ts**: Database connection and configuration logic.
  - **swagger.ts**: Swagger configuration for API documentation.
  - **utils.ts**: General utility functions.

Models

- **models**: This directory contains Mongoose schemas or other data models.
  - **CategorySchema.ts**: Mongoose schema for categories.

    - **PlaceSchema.ts**: Mongoose schema for places.
    - **TripSchema.ts**: Mongoose schema for trips.
    - **UserSchema.ts**: Mongoose schema for users.

Configuration and Environment

- **.env.local**: Local environment variables.
- **.env.sample**: Sample environment variables file for reference.
- **.eslintrc.json**: Configuration file for ESLint, a linting tool for identifying and fixing problems in JavaScript code.
- **.gitignore**: Git configuration file to specify intentionally untracked files to ignore.