Atlas Itinerary Final Report

By Gabriel Taylor, Dustin Zook, Gabriyel Sorensen, & Sorelle Djuissi

# The Purpose and Introduction

Atlas Itinerary is a personal travel itinerary planner that helps users design detailed, time-slotted trips without the pressure of booking. After creating an account, users can build multiple trip plans, each focused on a specific destination. During a trip, they select lodging, restaurants, activities, and attractions, and then schedule them for specific days and times. The result is a daily schedule that shows what they will do and when they will do it instead of a loose collection of bookmarks and notes.

Many people start planning with sticky notes, random websites, and scattered phone reminders. They may know what hotel they want, where they want to eat, and which attractions are “must-sees,” but they struggle to see how everything fits together in a realistic timeline. It is easy to double-book a time slot, underestimate travel time between places, or simply forget a stop entirely. Atlas Itinerary addresses this problem by providing a single web application where users can search for places, select the ones they care about, and plug them into a calendar-like structure for their trip.

The audience for this system is casual travelers who want structure without committing to reservations yet, as well as more technical users who want a repeatable way to plan multiple trips. From a developer’s point of view, Atlas Itinerary is also a small but complete full-stack React and Next.js application with authentication, a Postgres-backed database, and server-side API routes. With the documentation in this report, a team of developers with similar skills should be able to understand, build, fix, and extend the system.

# Technology

## React and Next.js

The front-end is built with React using Next.js’s App Router. We chose React because several team members had prior JavaScript experience, and React’s component model made it easier to break the UI into reusable pieces like cards, modals, and layout components. Next.js provides server-side rendering and simple routing conventions (/app/home, /app/itineraries, /app/instructions, etc.), which reduced how much boilerplate routing code we had to write.

Coming up to speed on React and Next.js was a mix of familiar and new. Basic JSX and component logic felt close to JavaScript we already knew, but Next.js’s “server vs client component” rules and the new use client directive took some practice. We also had to learn Next.js patterns for page.tsx files, dynamic routes (for itinerary details), and how to use the built-in Image component correctly.

Overall, React and Next.js made development easier once the learning curve was behind us. We could reuse components like Header, itinerary cards, and modals across multiple pages, and the framework handled a lot of performance and routing details for us.

## Supabase (Postgres + Auth)

Originally, we planned to use Firebase as our backend. Several teammates had used Firebase before, and we expected to lean on that experience. After some research, though, we switched to Supabase. Supabase provides a Postgres database, row-level security, and a JavaScript client SDK that integrates very smoothly with React and Next.js. We realized that using Supabase would give us a relational schema and SQL while still providing an easy front-end API and built-in authentication.

The initial learning curve for Supabase was mostly about understanding its dashboard, how to define tables, and how row-level security policies worked. The client code ended up being straightforward: we used Supabase’s JavaScript libraries to read and write itineraries and locations, and we used the Supabase auth helpers to manage sessions. Once we had the first query working, adding additional CRUD operations felt repetitive in a good way.

Switching from the original Firebase plan to Supabase was the correct decision for this project. Supabase lets us model our data with proper primary keys and foreign keys and makes it easy to enforce that each itinerary is owned by a specific user. That structure made later features, like deleting an itinerary and cascading delete of all its locations, much simpler to implement.

## Google Places API

We also changed our location provider during development. The original proposal used the TripAdvisor API, but we ran into problems obtaining an API key and getting timely support responses. We switched to the Google Places API because Google’s documentation is strong, keys are easy to generate, and the API fits well with our UI needs: searching for destinations, displaying place details, and re-hydrating places later based on an ID.

Using Places introduced some complexity around API quotas and securing keys. On the client side, we used the Places Autocomplete and Details endpoints to power the search bar and the details view. On the server side, we stored only the Google Place ID inside our locations table, and then used that ID with the Places API when we needed to show details. That design kept our database schema simple and avoided duplicating place data.

## Vercel

We chose Vercel to host the React and Next.js app because it integrates directly with GitHub and Next.js. Every push to main triggers a new deployment, and the Vercel dashboard provides Web Analytics and deployment logs we used for debugging.

The learning curve for Vercel was fairly mild. The main tasks were connecting the repository, setting environment variables for Supabase and Google Places, and understanding the difference between preview deployments (for pull requests) and production. Vercel occasionally forced us to update dependencies (for example, bumping the Next.js version for security), but those pull requests were straightforward to merge.

## Development experience with the stack

* Overall, the technologies made development easier once we got through the initial ramp-up:
* React and Next.js gave us a comfortable way to structure pages and components.
* Supabase handled authentication and Postgres hosting so we did not need to manage our own database server.
* Google Places gave us high-quality location data with minimal UI changes after the switch.
* Vercel automated deployment and made it easy to share a running version of the app with the instructor.

The main difficulties were learning new framework conventions (Next.js’ server/client boundaries, Supabase auth flows) and coordinating environment variables so that the app worked the same way on every teammate’s machine and on Vercel.Design

## Overall architecture

Atlas Itinerary is built as a full-stack React and Next.js application with Supabase as the backend data store and Google Places as an external API. The system follows a standard layered approach:

**Presentation layer** – React components and pages under web/src/app, such as home, itineraries, itinerary-details, and instructions.

**Application/service layer** – TypeScript helper modules in web/src/lib/ that contain business logic and data access functions (getItineraries, addItineraryLocation, removeItinerary, etc.).

**Data layer** – The Supabase Postgres database with three core tables: users, itineraries, and locations.

**External services** – Google Places API for searching for locations and retrieving place details, plus Supabase Auth and Vercel for hosting.

(see [Figure 1](#bookmark=id.l5zxtoklrzbp))

Requests from the browser first pass through Next.js middleware, which checks authentication via Supabase. Authenticated requests are allowed to reach protected pages such as Home, Itinerary List, and Itinerary Details; unauthenticated users are redirected to the Sign-in page.

## Major pages and components

### Home / Instructions page

The Home page provides an overview of the application and instructions for how to create and manage itineraries. It uses shared components like Header and a set of “step card” components that display numbered instructions and screenshots. These components are styled with CSS modules under web/src/css, and many screenshots are loaded using Next.js’s Image component (see [Figure 2](#bookmark=id.y7a3brf19q45)).

### Itinerary List

The Itinerary List page queries Supabase for all itineraries for the current user by calling getItineraries(user\_id) from lib/repos/itineraries.ts. It displays each itinerary in a card that shows the destination and date range. When a card is clicked, it calls getLocations() using the itineraryid of the selected itinerary to retrieve all locations associated with it. For each location, it uses the placeid and the Google Places API to fetch Places information about the scheduled location. Places information retrieved from the Google Places API are then stored within a record, caching the information to minimize the number of API calls. Reopening an itinerary that has already been processed will not result in repeated API calls as the Places information associated with the placeid already exists and has been cached (see [Figure 3.1](#bookmark=id.ln8cum92j0kx)).

Once it has acquired all of this location and Places information, it creates a calendar display using FullCalendar’s React component, passing the necessary information from each location to display them on the calendar according to their start date, start time, end date, and end time. When the location is clicked on the calendar, it uses the location’s placeid to retrieve the Places information from the record and displays it (see [Figure 3.2](#bookmark=id.3w7xsn3t1ca7) & [Figure 3.3](#bookmark=id.lb19xa65brl9))

When the location’s Places information is displayed, the user can press the delete button, calling removeItineraryLocation() with the given location’s locationid, removing the row from the locations table and updating the calendar display. When the calendar display of an itinerary is shown, the user can press the delete button to call removeItinerary with the currently displayed itinerary’s itineraryid, removing the row from the itineraries table and updating the UI of the Itinerary List page accordingly.

The itinerary List page utilizes several components to function properly. ItineraryCards are used to display basic information about itineraries and list them on the page for users to interact with. The ItineraryModal displays the itinerary associated with the clicked ItineraryCard, retrieving its locations and the Places information of those locations to create a calendar display. The CalendarEventModal takes the location and Places information of the location associated with the clicked event on the calendar and displays the start and end time of the location along with the Places information such as a photo, address, ratings, website link, phone number, etc.

### Itinerary Details

The Itinerary Details page is the core of the application; it’s where users can create itineraries and build them by adding locations. To create an itinerary, a user must click on the Create Itinerary Button and fill out the form that appears. Once they have given the new itinerary a name, city for its destination, start date, and end date, they can click the Create button. This button calls the createItinerary() function with the form values along with the user's id to create a new row within the itineraries table. Before a user can add locations, they must have created an itinerary (see [Figure 4.1](#bookmark=id.ezbp4hq2yla2) & [Figure 4.2](#bookmark=id.t23jr2ddpviv))

To begin the process of building an itinerary, users need to select the itinerary they wish to work on in the dropdown list which has all of their itineraries. After selecting an itinerary, the page will populate with a list of cards showing nearby locations to the destination of the itinerary. These locations are separated by categories, either being lodging, dining, or attractions. A user can change the selected category and the page will update to display nearby locations within that category. Users may also search for a specific city destination outside of their selected itinerary’s travel destination with a search bar connected to Google Places. More locations can be searched for by traversing through additional pages of locations and pressing the load more button. All the locations retrieved from Google Places API are cached, expanding as more locations are found. Each category has its own separate cache, so switching between the location categories does not result in repeated fetching and API calls (see [Figure 4.3](#bookmark=id.figitrlr3nlj)).

When a user selects a place, the place’s id is used to make a call to the Google Places API and retrieve additional information. A modal then appears to display this information and allow users to input a start date, start time, end date, and end time. After the user chooses the start and end times, the page calls addItineraryLocation with the given times along with the itineraryid of the selected itinerary to insert the new row into the locations table on Supabase. By repeating this process with various locations and scheduled times, a user can build an itinerary and plan a trip for the destination (see [Figure 4.4](#bookmark=id.gir0o66g548j)).

The Itinerary Details page, like the Itinerary List page, makes use of multiple components to fulfill its role in creating itineraries. The CreateItineraryButton that opens the form for creating itineraries. The CreateItineraryModal which acts as the display for the Create Itinerary Form, retrieving user input and handling the creation of itineraries. The ItineraryPicker that displays all the itineraries associated with the user in a selection dropdown, allowing users to select the itinerary they want to add locations to. The CategorySidebar that displays the Lodging, Dining, and Attractions categories, allowing users to change which category of locations are displayed on the page. The CitySearch which uses Google Maps Place Autocomplete so that users can search for and select a city to find nearby locations. The PlaceCard to display basic information of a nearby place that users can click. The PlaceModal retrieves additional information about the place associated with clicked PlaceCard using its placeid and displays details such as a photo of the location, the address, ratings, a link to the place’s website, etc. Along with this, the component handles adding the location to the itinerary with user input. Finally, the Pagination component which deals with traversing between the various pages of nearby places.

### Shared components

Several React components are reused across pages:

* **Header / NavBar** – top navigation, branding, and how users check their user information and sign out. Displayed at the top of every major page within the application (see [Figures 2 - 4](#bookmark=id.y7a3brf19q45)).
* **ItineraryCard** – displays a single itinerary on the list page. Cards are dynamically generated according to how many itineraries the user currently has, reusing the component to display each unique itinerary and its information (see [Figure 3.1](#bookmark=id.ln8cum92j0kx)).
* **PlaceCard** - displays a single Place on the details page. These cards are dynamically generated according to the nearby locations of the searched destination, reusing the component to display basic information and a photo about each location retrieved from the Places APIs’ Nearby Search (see [Figure 4.3](#bookmark=id.figitrlr3nlj)).
* **PlaceModal** – shows details for a selected location and allows scheduling it into an itinerary (see [Figure 4.4](#bookmark=id.gir0o66g548j)).
* Layout components for consistent padding and responsive behavior.

Breaking the UI into these components kept the JSX for each page smaller and allowed us to test pieces in isolation.

## Database tables and scripts

The database for Atlas Itinerary is composed of three tables. This includes the users, itineraries, and locations tables. The users table represents all users of the application, storing their fullname and email, along with a user\_id column that is a Universally Unique Identifier (UUID) to identify an individual user. The itineraries table represents a created itinerary, having a name given by a user when created, a travelDestination which is the selected location the itinerary is created for, a startDate and endDate representing when the scheduled trip to the travelDestination is supposed to begin and end. Itineraries have a itineraryid UUID column to identify an itinerary. Itineraries also have a reference to a user\_id from the users table so that the application knows which user created the itinerary and is associated with it. The locations table represents all the scheduled locations within an itinerary. It includes a startdate, enddate, starttime, and endtime to determine when the location is scheduled to begin and end along with a placeid utilized to retrieve Place Details from Google’s Places API. locations have a locationid UUID column to identify a location. Locations also have a reference to an itineraryid from the itineraries table so that the itinerary that the location was selected and scheduled for can be known (see [Figure 5](#bookmark=id.5447v898br8b)).

To interact with the database, JavaScript functions utilizing Supabase’s libraries were written. In order to retrieve all itineraries stored within the database for a user, the getItineraries function is called in which the user\_id of the current user is taken and used to get all entries and their columns within the itineraries table where the user\_id within the table is equal to the user\_id of the current user trying to retrieve their stored itineraries. Retrieving all locations for an itinerary a user clicks on is achieved by calling the getLocations function and passing it an itineraryid, getting all entries and their columns within the locations table where the itineraryid within the table is equal to the itineraryid parameter. In order to delete a specific location, the locationid of the location is passed to the removeItineraryLocation function, deleting any entry with the locations table that has a locationid equal to the locationid parameter. The function removeItinerary, taking an itineraryid, deletes any entry within the itineraries table that has an itineraryid equal to the itineraryid passed to the function. As a result of how the foreign key constraint was configured for the locations table, once the itinerary with the passed itineraryid is deleted, all entries within the locations table associated to the itineraryid are deleted as well. This ensures that all information tied to the specific itinerary, including its scheduled locations, are removed when deleted.

For creating an itinerary, a name, traveldestination, startdate, and enddate, are passed to the createItinerary function. This function gets the user\_id of the current user creating the itinerary and inserts the values passed to the function along with the user\_id into the itineraries table. For building itineraries, the addItineraryLocation is called. Passing an itineraryid, placeid, startdate, enddate, starttime, and endtime, the function takes these values and inserts them into the locations table. Functions interacting with the locations and itineraries tables are in lib/repos/locations.ts and lib/repos/itineraries.ts respectively.

## Server-side scripts and authentication

Server-side logic lives mainly in three places:

1. **API routes** – Next.js route handlers that perform server-side calls to the Google Places API. These routes hide the server key (GOOGLE\_MAPS\_SERVER\_KEY) from the client and centralize error handling.
2. **Middleware (middleware.ts)** – Runs before certain routes and checks the Supabase session. If no valid session exists, the middleware redirects to /signin. If a user is already authenticated and tries to visit /signin, the middleware redirects them back to /home. This runs on Home, Itinerary Details, Itinerary List, protecting these pages from unauthenticated users and keeping them from potentially interacting with the database or any other functionalities within the application. Along with this, it runs on Signin, keeping users that are authenticated from ever seeing the Signin page again until they sign out or their session ends.
3. **Auth callback** – An API route that handles the OAuth callback from Google. When Google redirects back with an auth code, the server exchanges that code for a Supabase session and then redirects the user to /home.

These scripts together enforce that only authenticated users can reach pages that read or write itineraries and locations.

## Required data files

In addition to the TypeScript and React code, the system depends on several configuration files:

* .**env.local** – holds environment variables like NEXT\_PUBLIC\_SUPABASE\_URL, NEXT\_PUBLIC\_SUPABASE\_ANON\_KEY, NEXT\_PUBLIC\_GOOGLE\_MAPS\_API\_KEY, and GOOGLE\_MAPS\_SERVER\_KEY. These are required for running the app locally.
* **next.config.mjs** – Next.js configuration (e.g., allowed image domains).
* **package.json** – lists dependencies such as next, react, @supabase/supabase-js, and testing tools.
* **CSS files in web/src/css** – define the layout for headers, instruction cards, itinerary cards, and modals.

# How to deploy or build the application

For an individual to build or deploy the code, they must first begin by downloading the code from GitHub. This can be done by having git installed and running the command “git clone https://github.com/DustinWZook/Atlas-Itinerary.git”. After the files have been downloaded, one must make sure they have Node.js installed and working correctly. From there, they will want to install the required dependencies in order for the program to function properly. Using Node.js, they can enter the /web directory (“cd web” for Windows users) and run the command “npm install”. This will install all the required packages by the project according to the package.json file.

Before an individual can build or deploy the code at this point, they must set the environment variables for Supabase and Google Maps. These variables include NEXT\_PUBLIC\_SUPABASE\_URL, NEXT\_PUBLIC\_SUPABASE\_ANON\_KEY, NEXT\_PUBLIC\_GOOGLE\_MAPS\_API\_KEY, and GOOGLE\_MAPS\_SERVER\_KEY. This will require them to set up a project within Supabase and obtain a Google Places API key. To obtain an API key for Google Places, they will need to create a Google Cloud Project. Once created, they will need to navigate to the APIs & Services page and click on “Enable APIs and Services”. From there, they will search for Places API (New) and enable it. Once enabled, they can create an API Key for the Places API (New) under credentials and use that key with this application.

For someone to set up a project in Supabase for this application, they will need to create a new project in Supabase along with a Google Cloud project if they have not yet made one. In the Google Cloud project, they need to use the Google Auth Platform console to register and set up the application’s targeted audience of google users and what the application can do with google user data. Once the Google Cloud project’s Auth Platform is properly configured, the Supabase project’s Google provider needs to be set up as well. This requires obtaining a Client ID and Client Secret from the Google Auth Platform. In the Clients section of the Google Auth Platform console, they will need to create a new OAuth client ID and set the application type to Web application. Afterwards, they will need to provide the application’s URL under Authorized JavaScript origins. They will then need to add the Supabase project’s callback URL under Authorized redirect URIs and finally create Client ID and Client Secret. Once set up, Supabase authentication can use Google as a provider, allowing users to sign in to the application with their Google accounts.

Once the Supabase project has a properly configured Google provider, setting up the tables must be done. This includes creating the itineraries and locations tables. They may also want to set up a separate users table in the public schema that has a primary key referencing the id from the users table in the auth schema. The itineraries table will have a uuid itineraryid as its primary key, text name, text traveldestination, date startdate, date enddate, and a uuid id that references the user\_id from the users table. The locations table will have a uuid locationid as its primary key, text placeid, date startdate, date enddate, time starttime, timeendtime, and a uuid itineraryid that references the itineraryid from the itineraries table (see [Figure 5](#bookmark=id.5447v898br8b)). They will also need to enable Row Level Security (RLS) and create RLS policies for the itineraries and locations table so that only the rows associated with the logged user can be selected, inserted, and updated.

After the database is configured properly, an individual can either run the code in development mode using “npm run dev” or they can compile the code locally using “npm run build” and start the program with “npm start”. In order to deploy the code on Vercel similarly to our group, they will have to configure Vercel and deploy it either from the command line using Vercel CLI or by creating their own git repository and linking it to Vercel, causing any commit to the connected repository to trigger a new deployment.

# Known issues

## The Create Itinerary search bar can disappear after navigation

On the Create Itinerary page, the search bar for choosing the destination is a React client component that wires up Google Places Autocomplete. After we added more styling and layout rules near the end of the project, the search bar started disappearing in some cases. If a user goes to another page (for example, Itinerary List) and then navigates back, the component is still in the JSX, but our CSS sometimes hides it or pushes it out of view. This did not happen before we changed the page’s layout, so the problem appears to be caused by our flex/grid/spacing rules rather than by the Places API itself.

A future fix would be to simplify the layout around the search bar, move its styles into a dedicated CSS module, and test it across route changes and different screen sizes. We would also use browser dev tools to confirm that the input element is still in the DOM and then adjust or remove any CSS rules (for example, display: none, overflow: hidden, or conflicting flex settings) that cause the search bar to disappear.

## The user’s location is never displayed in the account dropdown menu

In the Navigation Bar, the Account option reveals a dropdown menu displaying user information, including their avatar image, full name, and email. The dropdown menu was supposed to display the user’s current location if they allowed the website to use their location. Because we never got around to adding this functionality, whenever the dropdown is displayed, the user’s location is always unavailable.

To finish implementing this feature, we will need to gain access to some geocoding API. Once we have access, we will retrieve the user’s location through the navigator.geolocation object. Through this object, we can get their current position as a pair of coordinates, longitude and latitude.Finally, we then use those coordinates along with a geocoding API to search for a human-readable address or location tied to the coordinates, retrieving information such as the user’s street address, city, state/province, country, etc.

## Editing Itineraries and Locations was not implemented

When in the Itinerary List page, users can select itineraries to display them and their associated locations. While users can delete specific locations from an itinerary and delete entire itineraries, making edits to the locations and itineraries is not currently feasible.

For implementing the ability to edit itineraries and locations, we could add an edit button to the ItineraryModal that handles edits to the itinerary’s attributes name, startdate, and enddate. We could then add an edit button to the CalendarEventModal that handles edits to the location’s attributes startdate, starttime, enddate, and endtime. When these buttons are clicked, they will update the UI to have boxes so the user can input and confirm their changes. Once those changes are made, they will be prompted to confirm their changes. After confirmation, the input values will be passed to functions that update the database rows with the associated itinerary or location id with the passed values. Afterwards, the Calendar display will be updated to reflect the change to either the itinerary itself or the locations.

# Figures for Design Description

Figure 1: Overall Architecture (Block Diagram)

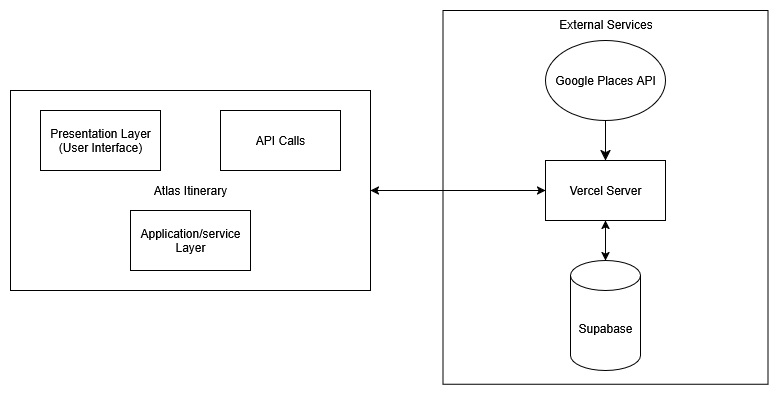


Figure 2: Home Page

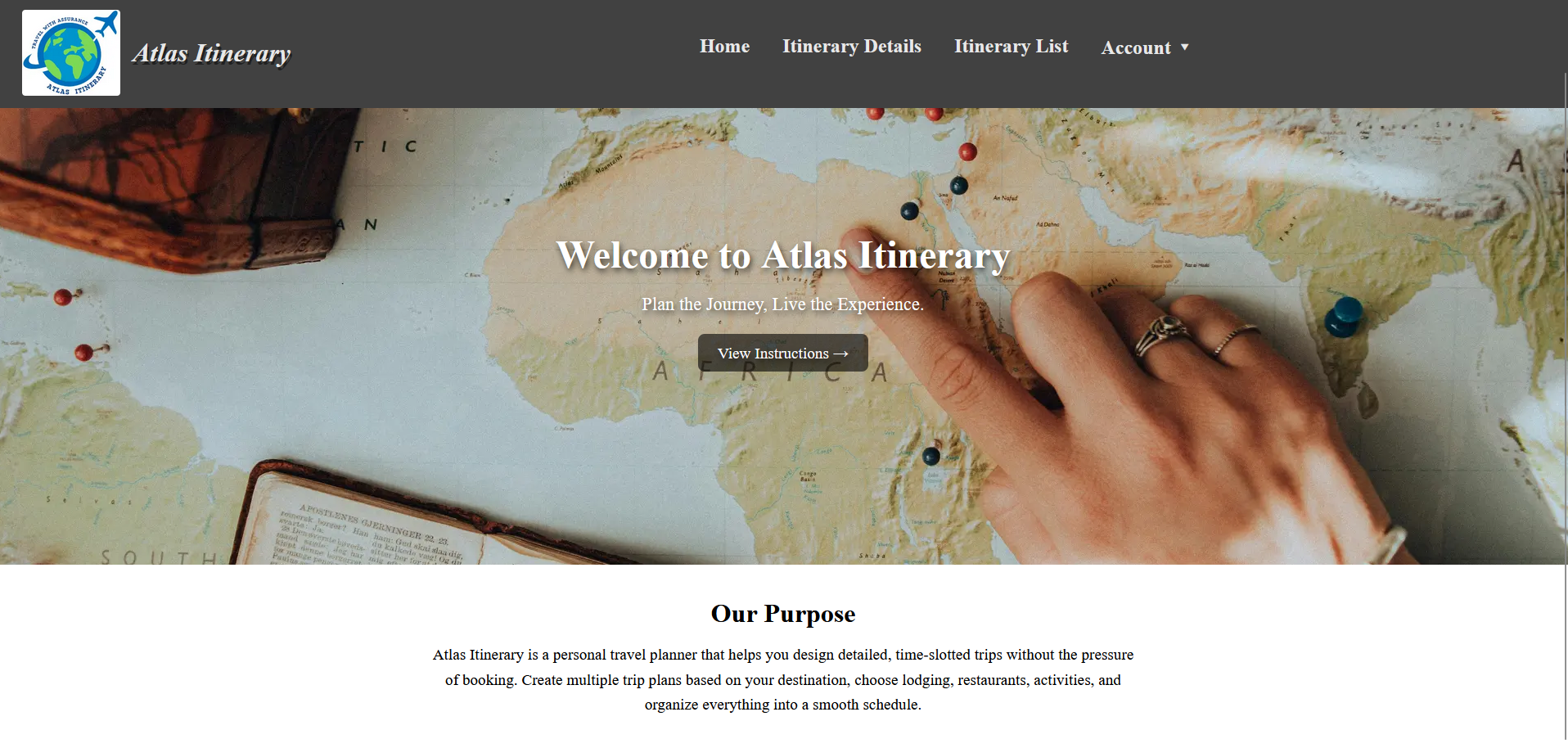


Figure 3.1: Itinerary List Page after calling getItineraries and generating the ItineraryCard components for each itinerary



Figure 3.2: ItineraryModal display after clicking on an itinerary and retrieving all of its locations with getLocations along with fetching all places information from Google Places using the location’s placeid

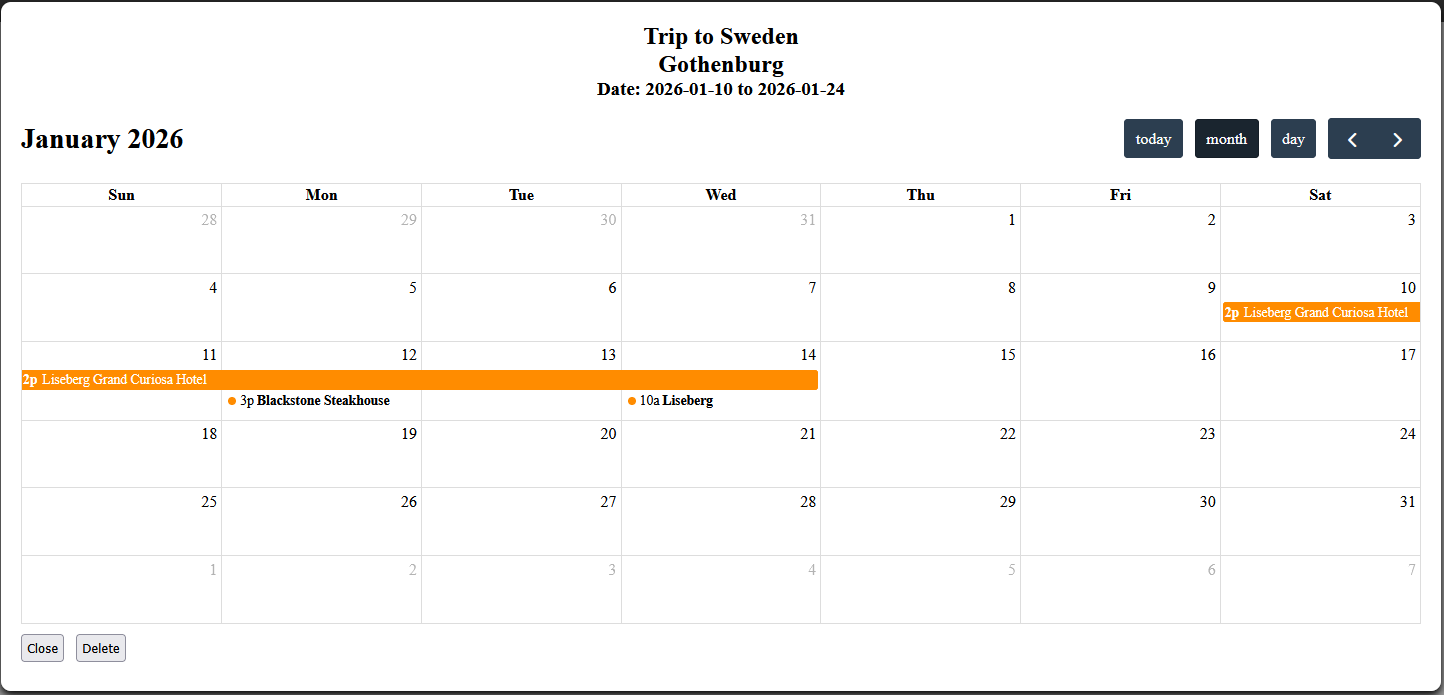


Figure 3.3: CalendarEventModal display after clicking on an event on the calendar and fetching the places information for the location from the stored record



Figure 4.1: The Itinerary Details Page when first entered

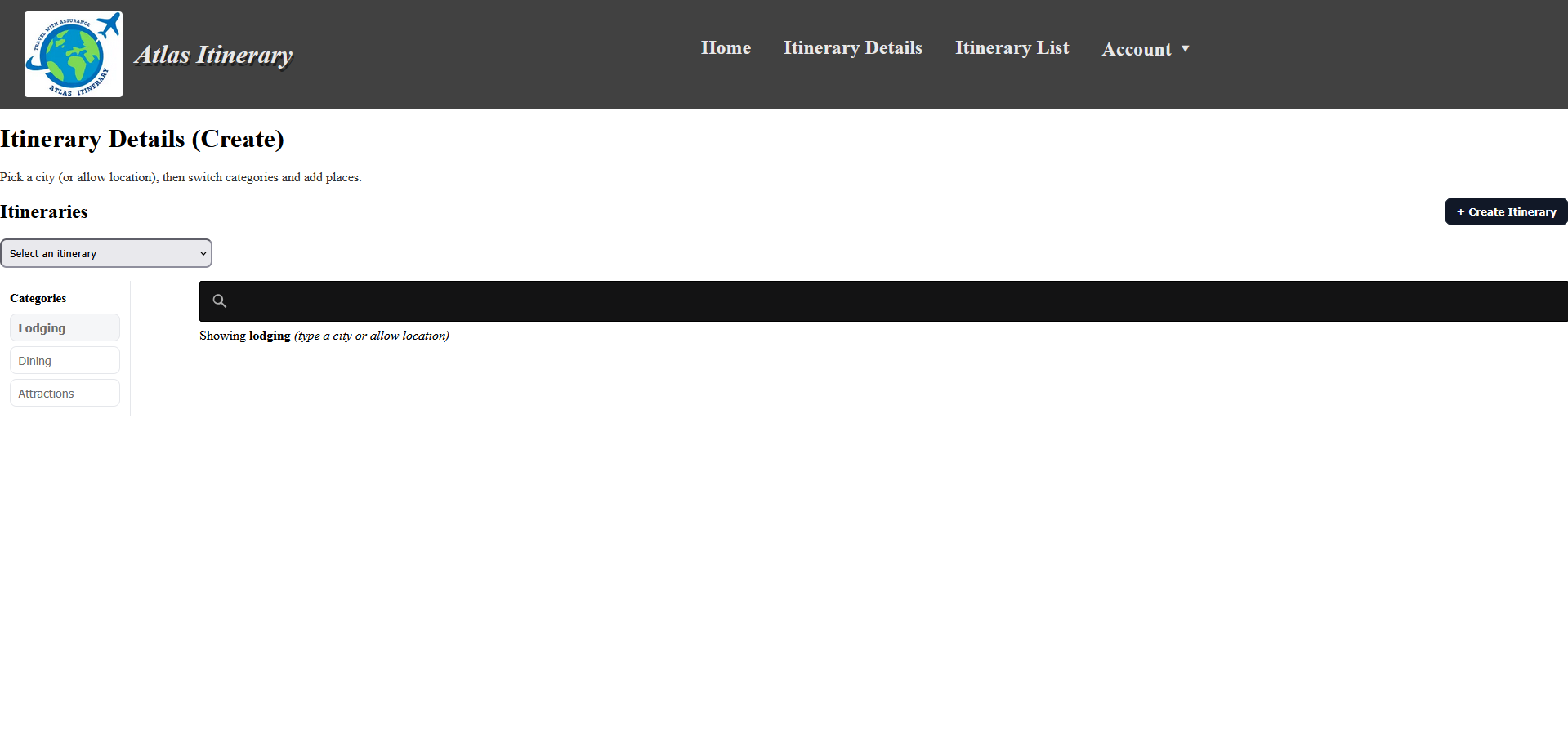


Figure 4.2: CreateItineraryModal displays when the CreateItineraryButton is clicked, getting user input to create a new itinerary and store it within the database

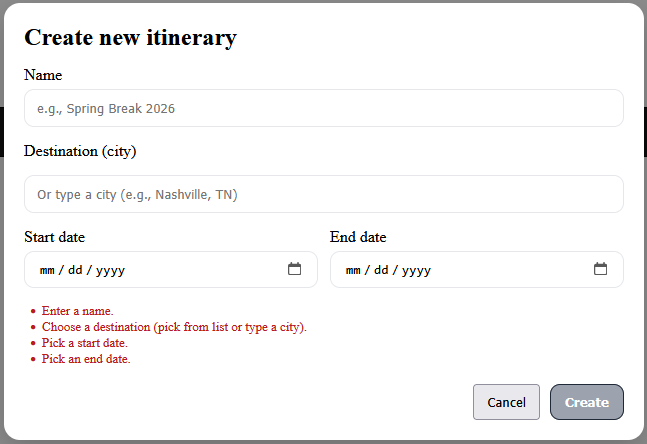


Figure 4.3: Itinerary Details Page after an itinerary is selected with the ItineraryPicker and fetches nearby locations to the Itinerary’s travel destination from Google Places

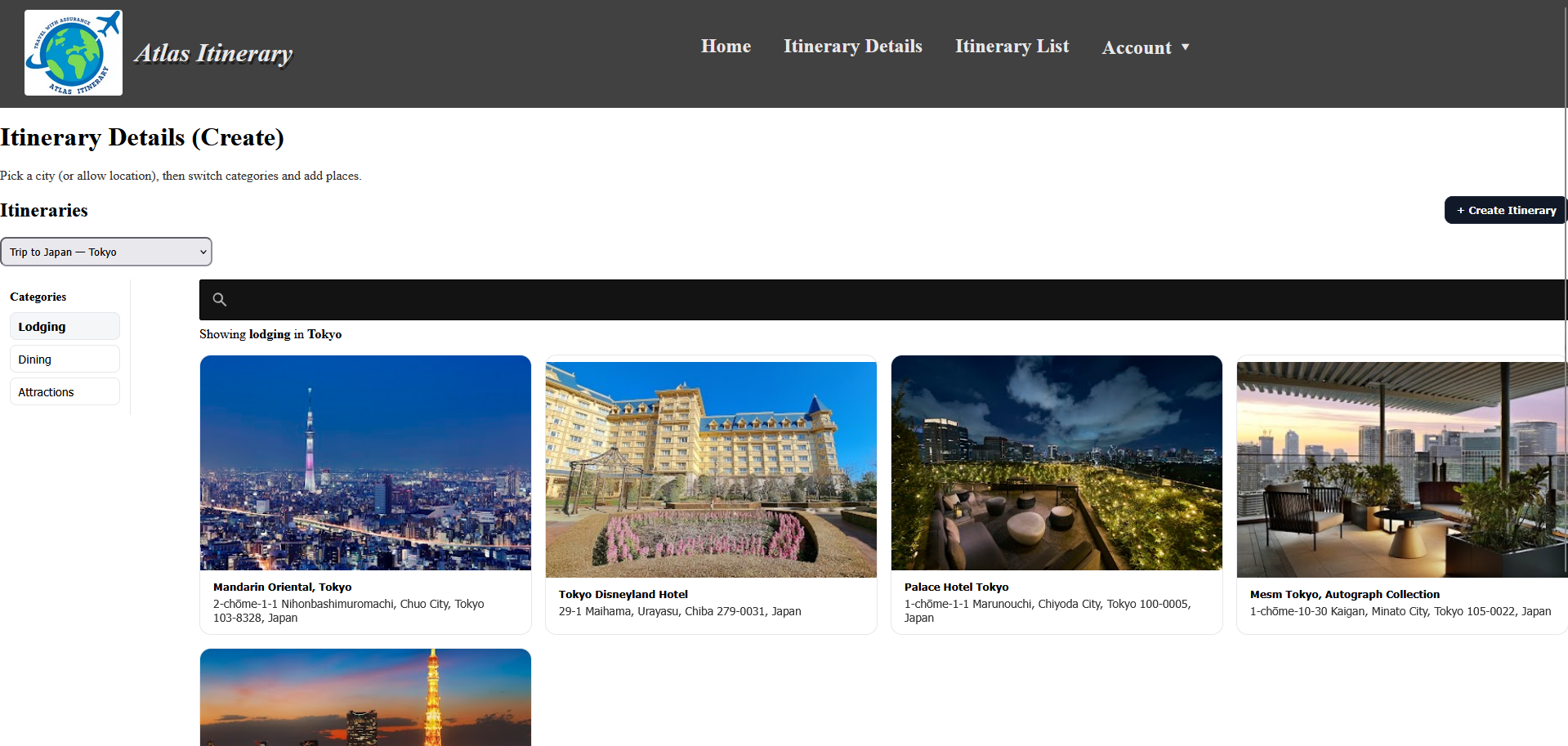


Figure 4.4: PlaceModal display after clicking on a PlaceCard and fetching additional details about the location from Google Places using its placeid. Handles user input to add the selected location to the currently selected itinerary and store it within the database

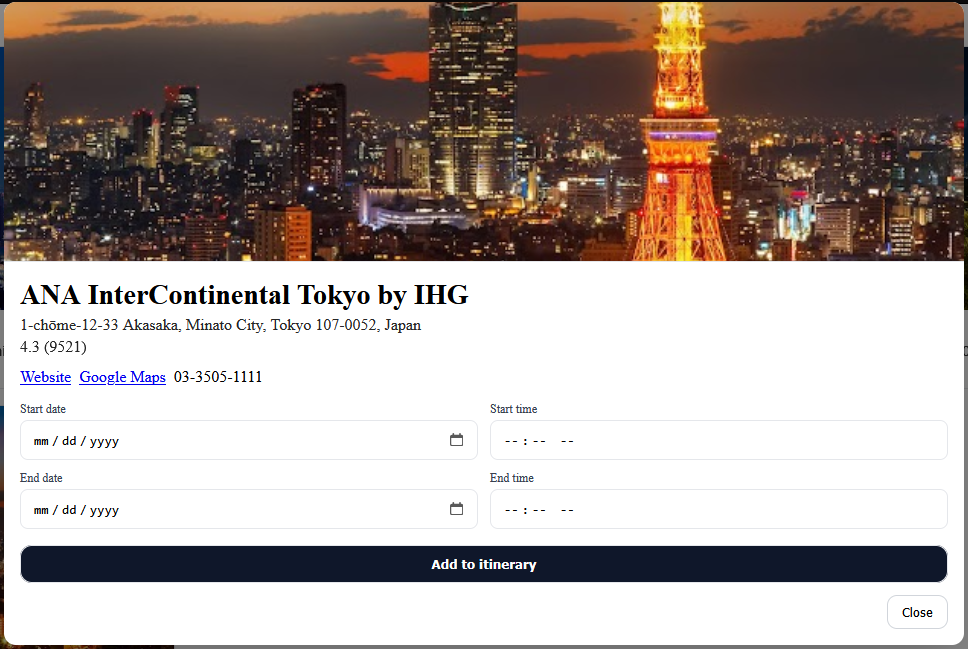


Figure 5: Supabase’s Schema Visualizer of Atlas Itinerary’s public schema

