Northeastern

Project 2: TravelPal

Kimberly Nguyen, Longsheng Lin, Matt Dang, William Guo

April 21, 2018

Table of Contents

I. Introduction and App Description

TravelPal is a social flights application that allows users to search for and book flights and hotels, maintain a diary of past trips, and keep up with their friends' travels. In order to use the application, users must first register for an account. This allows users to create a public profile, add friends, and tailor search results to their travel preferences. Because users are required to provide an email upon registration, this also allows the application to send email notifications when a new account is registered and when an existing account is updated or deleted. The application aims to serve as a platform for travelers on a budget and to provide a smooth trip-planning experience.

Users can input their travel preferences in the form of travel date cards. A travel date card contains details about the desired destination, the date range that users would be available to travel, and the price limit for that specific trip. Users can then search for any fitting flights using the information from the travel date card. This allows users to save and organize all of the places they want to travel to in one place and start planning whichever trip when they are ready. After booking a flight, the application will then allow users to book a hotel if they so please.

Once a flight has been booked, the travel date card gets converted to a booked trip card, and users can use the "Booked Trips" page to keep track of all upcoming trips. As users go about their travels, the booked trip cards will migrate to the "Past Trips" page. Here, users can add and edit summaries of their trip and maintain something like a personal diary.

II. User Interface

When users first visit TravelPal, they will be prompted with a log-in form. If they do not have an account yet, they can create an account using the registration link below the form. Clicking this toggles the registration form, and users must enter a unique email and username, their full name, and a password that is at least eight characters long. Users are also asked to enter their usual budget so that the application can recommend trips within their budget. Once these requirements have been validated, users will be redirected to the log-in form after clicking the "Submit" button, or they will receive an alert if anything has gone wrong. After entering in their username and password, users will then be redirected to the home page.

There is a navigation bar at the top that users can use to navigate through the different pages: "Home," "Search," "Travel Dates," "Booked Trips," "Past Trips," and "Profile." The home page displays the user's social feed. This feed displays all of the booked trips of the user's friends. This allows users to stay-up-to-date with friends and coordinate trips if they happen to discover that a friend is mutually interested in a destination or has set aside the same travel dates. The feed also has a "Popular Flights" section, where the application recommends potential trips based on the user's budget. This feature only takes into account budget to help users discover new destinations that they might not have considered beforehand.

The "Search" page is where users can search for flights. Users are asked to enter a destination and the date range for which they will be travelling. After clicking the "Search" button, the application will display the results of the search from the flights API. For each search result, users are able to see the departure and arrival dates of the flight and the cost of the ticket. Once users find a satisfactory flight, they can use the "Book" button to book the flight. Users will then be shown a list of hotels near the destination that have rooms available for the booked dates. Users are allowed to skip this step if they do not wish to book a hotel. Once done, users will be shown a confirmation page indicating that the flight (and hotel if applicable) has been successfully booked.

The "Travel Dates" page is where users can organize potential trips. Users can click the "+ Add" button in the top right to create a new travel date card. Clicking this toggles a form that

asks for the destination, from date, to date, price limit (the maximum that users are willing to pay for the flight), and the number of passengers (self included). Once done, users can click the "Submit" button in the bottom-right to submit the information. Or, they can click the "Cancel" button beside the "Submit" button to close the form. If there are any invalid inputs, the user will see a red error message below the input that caused the error. If the inputs are valid, the application will render the travel date details as a card. Users can edit or delete individual trips by clicking the "Edit" or "Delete" button, respectively, in the bottom-right of the card. Clicking the "Edit" button will toggle the same form as when users click the "+ Add" button.

The "Booked Trips" page shows all of the trips that users have booked, organized by trips this month and future trips. Like travel dates, booked trips are rendered as cards. Each card displays the destination, from date, to date, total cost, departure time, arrival time, number of passengers, hotel, and number of rooms. Users can click the "View Itinerary" button in the bottom-right of the card to toggle the itinerary. The itinerary displays the origin from which the user will be flying, a list of the airlines that the user will take, and the round-trip, departure, and return trip durations. Clicking the "View Overview" button will toggle back the previous view.

Also in the bottom-right are "Edit" and "Unbook" buttons, which allow users to edit or delete booked trips, respectively. Clicking the "Edit" button toggles the edit form, where users can edit the total cost, number of passengers, hotel, and number of rooms. When the users click "Submit," the inputs are validated on the frontend and by the server. If there are any invalid inputs, users will see a red error message below the input that caused the error. If successful, the edit form will be closed, and the card will be updated with the new information. There is also a "Cancel" button on the edit form, which will close the form if users change their minds.

The "Past Trips" page shows all of the trips that users have booked and gone on. Each past trip is rendered as a card, showing the same information as on the booked trip cards. However, there is an additional section called "Trip Summary," which displays the description that the user has included about the trip. Users can edit or delete past trips by clicking the "Edit" or "Delete" button, respectively, in the bottom-right of the card. Clicking the "Edit" button toggles the edit form, where users can edit only the trip summary. This is because the "Past Trips" page is meant to serve as a sort of personal diary for users. Once users have entered a

summary, they can click the “Submit” button in the bottom-right to see their changes. Or, they can click the “Cancel” button to the left of the “Submit” button if they wish to cancel their changes.

The “Profile” page shows the user’s information and the user’s friends. In the “About” section, users can see their name, username, email, and budget. Since username and email are unique identifiers, users can only edit their name or budget. They can do so by clicking the pencil icon beside these fields. Clicking this toggles an input field where users can enter new values and click “Save,” or they can click “Cancel” if they change their minds.

The “Friends” section shows all of the user’s accepted friend requests and pending friend requests. The details of each friend is rendered as a card that displays the friend’s name, username, and email and the year in which the user and became friends with that person. If users have already accepted the friend request, then they will see an “Unfriend” button in the bottom-right of the card. If the request is pending, and the user was the one who sent the request, then the user will see a “Cancel Request” button. If the user was the one who received the request, then the user will see “Accept” and “Deny” buttons instead. Clicking “Unfriend,” “Cancel Request,” or “Deny” will remove the friend from the user’s profile.

There is also a “Find Friends” button in the top-right of the “Friends” section, where users can search for other users by name or username. This search feature will match by full name/username, the beginning of the name/username, or any part in the middle of the name/username. If there is a match, the results will be displayed below the search input. Clicking a result will redirect users to that user’s profile. Here, users can see the user’s name and username and buttons that will allow users to friend or unfriend the person, depending on the current relationship between them.

III. Server-Side State

The main functionality of our server-side is to handle retrieval of data from external APIs and manage the database. For handling state, our application utilizes a database with multiple tables for storing data. Each table has a defined schema, and some have relations to other tables by way of a foreign key. There is also a table for each external API we have that stores the data from the requests made.

The `Weather` table has a schema containing the following fields: `city`, `date`, `high`, `low`, `text`, and `forecast`. The `city` field is a string representing the location that the weather information is pertaining to. The `date` field is a date object representing the date to which the information applies. The `high` and `low` fields are integers representing the high and low temperatures, respectively, for the day. The `text` field is a string description of the weather. The `forecast` field is an array of maps containing the weather data for the next six days.

The `Flight` table has a schema containing the following fields: `origin`, `dest`, `date_from`, `date_to`, `price`, `airlines`, and `duration`. The `origin` and `dest` fields are strings representing the origin and destination city, respectively, for the flight. The `date_from` and `date_to` fields are date objects for the dates that the flight is from. The `price` field is a float for the total cost of the flight. The `airlines` field is an array of strings representing all of the airlines involved in the flight. The `duration` field is a map containing the times for the arrival and departure flights, as well as the total time. The times are measured in seconds.

The `Hotel` table has a schema containing the following fields: `district`, `link`, `name`, `price`, `rating`, `image_src`, and `result_from`. The `district`, `name`, `price`, and `rating` fields represent general information about the hotel. The `link` and `image_src` fields provides users with a link to the hotel on Booking.com as well as an image of what the hotel looks like. The `result_from` field is the keyword that was searched that resulted in the hotel being inserted into the database so that we do not have to scrape for information more times than we have to.

Along with those tables and schemas for our external API data, we also have additional ones for other parts of our application. These include tables for booked trips, friends, travel

dates, and users. There are relationships between the tables, and each have controllers that manage operations on the data stored in them. When the front-end makes requests for the data, the router is configured to point at the controllers that handle certain data to return.

IV. APIs

For our project, we utilized two different external APIs for retrieving weather and flight information. We used Yahoo's Weather API for our source of weather data and Kiwi's Flight API for our source of flight data. Yahoo's weather API was chosen primarily because one of our team members had some prior experience using it and thought it would be able to fit our needs. An interesting feature of the API is that it requires queries to be made in YQL, which stands for Yahoo Query Language. YQL was not too difficult to use since it has SQL-like syntax. With YQL, we were able to make queries specifying what data we required and for which location. The requests returned data containing forecast information for seven days of a specific city. It was returned in JSON format, which we then converted to an Elixir list of map objects using the Poison library. Once converted, the data was not able to be immediately returned because there was a date field on each forecast map that had to be properly formatted. The date was originally a string in a non-standard format, but we wanted to store it as an Elixir date object to be easier to work with later. So, we used a map function to format each date string to an Elixir date.

The other external API that we incorporated into our application was one for flights, provided by Kiwi. This API allowed us to request for flight information by sending queries with a variety of parameters to tune the data we received back. We mainly tailored our requests to include an origin, destination, start date, and end date. The data we received from the requests were in JSON format and included a list of flights that matched our query parameters. After making a few test requests, we decided to attach a limit to the number of flights that the API returned because it was a very large response and was difficult to work with. Just as with the weather API data, we had to format the data before we could return it. Also, we utilized the database to store flight data and return that data if any calls matched the same parameters. This way, we could reduce the number of external API requests that were made.

One consideration that we had to keep in mind while having the external API requests working in our code was the rate limits. The APIs that we used only allowed us to make a certain number of requests per day or month, so we had to make sure that we did not exceed those numbers in the development stage, where we would be frequently running the program to test the

functionality. As a workaround to that, we set up the back-end to return dummy data that reflected what the actual API request would return. This way, we could prevent any additional requests called during development when it was not necessary for the data to be entirely accurate.

Another method we used to reduce our external API requests was to store the data from those requests in our database and then return that for any subsequent calls with the same parameters. For example, if there was a call for weather information in Boston, then an external API request would be made. However, it would be stored in the database first before being returned to the front-end. This way, if there were any calls for weather information in Boston made after that, then we could use the data in the database, saving us an API request. Periodically, we would replace outdated data in the database with new data from other external requests. An added benefit of using the database was that it required less query time compared to making an HTTP request.

V. Complexity

Our goal was to create an application with a fluid and dynamic user interface, and we decided to use React/Redux to create a single-page application as a result. The complex part of our application was making sure that information could be passed around between the front-end and the back-end seamlessly. We handled all of the interactions with external APIs as well as with a web scraper through logic implemented in our Phoenix controllers. We created API endpoints to connect our frontend and backend and facilitate the flow of information. In order to use the APIs, we had to first preprocess data into parameters for an API call and then process data again from the response to display to the user or store into our database. We had to decide what we wanted to keep in our database, whether it be information that we needed or records that would help us reduce the amount of API calls that we would need to make in the future. Since API calls are not as fast as querying our database, and we have extrinsic limits imposed, we used the database to our advantage. We also had to model and maintain the relations between our entities. On the user's side, we had to present information in a way that makes sense. For example, only showing a list of flights when a user performs a certain action and only showing a list of hotels after a user has confirmed a trip by selecting a flight. This leads back to reducing the amount of unnecessary work done in the backend.

Another layer of complexity we added was email functionality. We used the Bamboo library using the SendGrid adapter for sending emails to registered users. This required us to create an account on SendGrid to acquire an API key. After we received our key, we could integrate the functionality into our application and configure the email capabilities. Although we only set up the email feature to send emails to users on account creation, account updates, and account deletion, it would have been easy to integrate that feature into other parts of our application if we wanted to.

VI. Challenges and Solutions

One challenge we faced at the beginning of the project was deciding on which external APIs to use. This was especially difficult when trying to find a hotels API. Even after extensive research, the ones that were deemed usable required some sort of membership, fee, or partnership. This would not have been possible due to time constraints. Some of these APIs also required that we have a certain amount of traffic going through our website and that we demonstrate the ability to bring in revenue before allowing us to use their APIs. This narrowed the total amount of possible APIs down to a select few. However, we still could not find a hotels API that allowed us to use it for free. We solved this problem by scraping booking.com for their hotel postings. We confirmed that this was okay by checking the robot.txt file, which did not specifically disallow this. One downside is that this introduces a bit of a wait time (approximately three seconds) before we can display any hotels.

Another somewhat challenging problem was that the APIs had some limit either per minute, hour, or day. Since we were just using the free tier for these APIs, we had to keep to low limits or risk having our access revoked. This meant that we could not allow the users to dictate at all times when API calls are made. Instead, when users search for destinations, we query our database first for any previous information that was looked up about the same destination. If the information exists in our database, then we do not need to use one of our limited API calls to query the external API. If we do not have the information already in the database, then we call the external API, return to the user the required information, and store all of the data from that call into our database in case someone in the future requests the same information again.