Design and Implementation

Frontend

List the key libraries, languages, components used by the MVP.

If applicable, describe essential screens.

Backend (Application Server Design)

In our POC, the burden of managing business logic was placed largely on the app, namely database accesses and API requests to different endpoints. We would like to centralize all of this functionality in our application servers, allowing the frontend application to communicate via CRUD requests to a single endpoint that we host (for most requests) This will imply

- A more performant and reactive application frontend: the user's device won't have to handle and complicated business logic such as filtering a large number of public itineraries.
- Clear separation between the frontend and backend: the backend can be updated independently of the frontend, provided that the general API format is well-defined and consistent

We envisage implementing core backend functionality in blazingly-fast and memory-safe Rust.

Authentication

We would like to use an authentication library compliant with OAuth 2.0 standards, offloading sign-in to third-party applications instead of building our own authentication from the ground up. This also enables the user to use their account from another service instead of going through the process of creating a new username-password combination that they can forget.

We will opt for a session-token authentication approach as opposed to JWT. Given the stateless nature of JWT, we wouldn't be able to manage the lifetime of a user session (for example, in the case that they wish to sign out). This adds a certain burden to our application servers, as authenticated actions will require additional database interaction.

Responding to a Request from the Frontend

1. Receive authentication code (unstored) and make API request to relevant authentication API (e.g. Google Auth). Once validity is confirmed, generate a session token for the frontend. Only actions that require specific access rights (`DELETE` an itinerary, `GET` user's private itineraries) will require that the token be provided in the request header.
    a. Token is queried in the `USER` table of our database

   b. If no row exists, then respond with a `401: Unauthorized`, else we can check if the token corresponds to a user that has correct access rights for the action they are performing (e.g. a user should only be able to `DELETE` their own itinerary)
2. The application server makes API requests to Google Directions API, parsing and caching the resulting polyline (as we discussed in chapter 7). Future previews of the same itinerary won't require further API requests.
3. The application server handles access of an itinerary, querying first the cache and then the main database if needed.
4. The application server maintains lightweight metrics of frequently accessed itinerary `uid`s for cache maintainance. We don't want to be polluting an in-memory cache with *"one-hit-wonder"* itineraries, for example.

Data Model

What data are you collecting / managing?

How is it organised?

Where is it stored?

How is it shared/copied/cached?

Security Considerations

Infrastructure and Deployment

How is the application developed, tested and deployed?

Any special infrastructure requirements.

Test Plan

How is the application developed, tested and deployed?

Any special infrastructure requirements.