Tools and frameworks

Following is a list of tools used for the backend development and motivation("Why") of using them

Serverless framework

The serverless framework is an open source platform agnostic tool that facilitates development and deployment of serverless applications. See docs

The `serverless.yml` file in the backend root describes the configuration for each lambda function in our API. As a general rule, one lambda function is created per API endpoint.

The serverless framework has native support for community plugins which allow us to use custom bundlers(`serverless-jetpack` for our project), ('serverless-plugin-typescript' for typescript support)

The serverless framework dashboard allows for easy viewing of function invocations, logs, etc

Middy

Middy is a lightweight library that adds support for middlewares to serverless lambda functions. Adding middlewares allows us to separate business logic from boilerplate validations, etc. It is also an implementation of the well-known design principle Chain of Responsibility

Database communication via Prisma ORM

Prisma is an ORM(object relational mapper) that allows for quick database prototyping and clean communication with the database. A central config file(`prisma/schema.prisma`) is used to manage everything related to the database. It allows us to easily swap database provider and relieves us from performing repititive tasks like input sanitzation that add a lot of complexity.

Prisma also supports Typescript out of the box and generates types for our schema that can be used anywhere in the code.

When we generate a prisma client, it compiles a binary that uses a Rust runtime that enables memory safe and quick database access.

Event schema validation using Joi

A custom middleware `validateEventSchema` is created to clean up event validation and allow for direct and safe use of input variables

File structure

The handler functions for each lambda function are present in `src/*` . These handlers are designed to be thin wrappers around their core functions located in `src/utils/*` . Separation of business logic and boilerplate allows for easy testing and extension

Following is the general struture an API handler follows

```
 // Import the required modules
import { foo } from "bar";
...

// Initialize database connection outside handler to promote reuse between API calls
const prisma = new PrismaClient();

// Wrap the handler with middy() to allow for attaching middlewares
const handler = middy((event) => {
  // Parse parameters from event body
  // Call core API function

  // Return
})

handler
  .use(middlewareA())
  .use(middlewareB())
  ...
```

Backend functionality via REST APIs

1. Create Room

- API Relative Path : /create-room

- HTTP Type : Post
- Headers : authToken
- Usage : Host can create a private room with this API
- Input : PlayerID, Name
- Output : Room {id,hostid,key} , token (for authorization)
- Description: This is the very first API that will get hit once any user lands to Poll Me app. The PlayerID field is dynamic and is dependent on token. The fundamental logic here is to check if header has authtoken, if yes then decode playerID from authToken using prisma database and if not then we create a new entry in db and assign playerID. In cases wherein playerId is created afresh we send authToken in response of this API which is meant to be stored at clients local storage so that auth token header can be populated by client in subsequent requests. The response of this API suggests that a private room has been created hosted by {hostid} having Room ID as {id} and Room key as {key}.

2. Enter Room

- API Relative Path : /enter-room
- HTTP Type : Post
- Headers : authToken
- Usage : Player can enter a private room created by host using this API
- Input : PlayerID, Name, RoomKey
- Output : questionData { questionType: {qid,description,options} }, token (for authorization)
- Description: After the host shares the room key with required audience, players can start joining the room using RoomKey. The API does some processing on whether its a first request by a specific player OR its duplicate and accordingly fetches playerID from prisma db. Once this processing is done, a new entry is added to room table with the current player. There is also some error handling we have performed to check if RoomKey is non-existant OR we are unable to fetch player data due to issues with auth token. Once the player is added to the room successfully, we create a response object of questions present in that room and send it back to the client.

3. Create MCQ question

- API Relative Path : /create-mcq-question
- HTTP Type : Post
- Headers : authToken
- Usage : Host can create MCQ question to be displayed in Room using this API
- Input : RoomID, Title, Options
- Output : question {id,roomId,createdAt,isPublished,questionType} }, options
- Description: The host uses this API to create poll questions. First we verify if host is allowed to create question by checking if the room is created by same host. After that we create a prisma db entry for the question and its associated options. As part of API response we send the question id that was created, creation time and whether question is published yet. We also send the options associated with this question.

4. Vote MCQ option

- API Relative Path : /vote-mcq
- HTTP Type : Post
- Headers : authToken
- Usage : Player can vote for options in multiple choice question with this API
- Input : PlayerID , QuestionID, AnswerID
- Output : PlayerAnswer {pid,qid}
- Description: Player chooses the answer for a question and submits the answer. This API creates a playerAnswer object which internally points to playerId and questionID. Thus an entry is created on which player have voted for which answerID attached to which question. We use foreign key references to populate data in playerAnswer table in prisma.

5. Fetch Analytics

- API Relative Path : /fetch-analytics
- HTTP Type : Post
- Headers : authToken
- Usage : Players as well as Host can use this API to fetch analytics for poll questions in room
- Input : RoomID
- Output : answerId, qid, count
- Description: : Player and Host can view the statistics of the poll with this API. The API returns count for each answerID which can then be fed to client to display it via visuals like graph, piechaart etc

6. Fetch Question Data

- API Relative Path : /fetch-question-data
- HTTP Type : Post
- Headers : authToken
- Usage : Players can use this API to fetch questions whenever added by the admin in room
- Input : RoomKey
- Output : questionData
- Description: : Player can use this api to refresh question data incase new questions are posted by admin. The API returns question data which contains questions as well as correct answer. This can be fed to client to display it in the form of MCQ questions.