Senior Backend Engineer

Introduction

The purpose of this case study is to provide a basis for being able to gauge the candidate's seniority and also technical and communication abilities.

The Case

The case consists of 2 parts:

- The first part looks to investigate the candidate's reasoning ability, their decision-making process as well as their prior experiences.
- The second part looks to investigate their technical ability to provide a production-grade service.

Both parts have required deliverables.

Part 1

You have been hired as a Senior Backend Engineer in a small team for a successful mobile games studio called Studio808. This studio has been known to make very good games that are played by hundreds of millions of players every month. Unfortunately, their backend has either been handled by an external publisher or kludged together due to a lack of time (and people) to do it properly.

Now, Studio808 wants to spread its wings - they've got 2 new games in the pipeline that they're looking to soft-launch within the next year. For that, they need a new robust and modern backend for both of these upcoming games, each requiring many game services.

You are tasked with deciding on the choice of cloud infrastructure/backend solution and designing an architecture setup that will scale easily, remain secure, highly available and is generally easy to work with. You have a team of 2 experienced backend engineers who will help you and are waiting for your new architecture to be presented.

DELIVERABLE 1:

Please deliver a presentation (lasting up to 15 mins max.) describing your architecture/backend. Include any prior experience that helped come up with these answers. You will present this architecture to us at a meeting if we proceed.

The following content should be included in the presentation:

- A diagram explaining how traffic is being routed into the backend along with a short text describing the flow;
- A summary of the tech stack and reasons for the decisions;
- Potential issues that you see with the chosen tech stack.

Part 2

It just so happens that a friend of yours has created a game they are confident will go viral and blow all other games out of the water. They've very kindly asked you to develop the backend for this.

Your friend has already created the game client and put together the API specifications as a simple JSON-based REST API that allows the game client to interact with the backend server.

While your friend asked you to implement the REST API in its current form and get it to work with the game as is, they also pointed out that he would be interested in hearing any suggestions you might have to improve the API for future versions of the game. Your friend has no opinion on what technology you should use - use whatever you feel the most confidence with.

DELIVERABLE 2:

Please submit a **production-quality** implementation of the proposed REST API.

What we are looking for:

- The service should be implemented in either Go, JavaScript, Python or Java;
- A well-tested code;
- The service should provide ways to run in a Unix-like system;
- Create a README file that contains information about how to run the service locally and general application usage;
- Implementation of a database layer that is not an in-memory store.

The REST API

The game uses a simple REST-based API and passes data using JSON. Note that it is assumed that ids used are UUIDs.

Create User

When the game is started up the first (or reset) time it will create a new user. The game expects this call to return a user id.

```
POST /user

REQUEST EXAMPLE:
{
  "name": "John"
}

RESPONSE EXAMPLE:
{
  "id": "18dd75e9-3d4a-48e2-bafc-3c8f95a8f0d1",
  "name": "John"
}
```

Save Game State

Whenever a game is played it will try to save its game state using this request.

```
PUT /user/<userid>/state

REQUEST EXAMPLE:
{
  "gamesPlayed": 42,
  "score": 358
}
```

Load Game State

On starting up the game it will load the game state. The score field should be considered to be the player's highest score.

```
GET /user/<userid>/state

RESPONSE EXAMPLE:
{
  "gamesPlayed": 42,
  "score": 358
}
```

Update Friends

This request will only happen when the friend list actually changes. The request contains the full list of friends.

```
PUT /user/<userid>/friends

REQUEST EXAMPLE:
{
  "friends": [ "18dd75e9-3d4a-48e2-bafc-3c8f95a8f0d1",
               "f9a9af78-6681-4d7d-8ae7-fc41e7a24d08",
               "2d18862b-b9c3-40f5-803e-5e100a520249" ]
}
```

Get Friends

This request is called upon starting the game and must return the user's list of friends as well as their high scores (the same score as listed in the game state) to allow populating the high score table.

```
GET /user/<userid>/friends

RESPONSE EXAMPLE:
{
  "friends": [
    {
      "id": "18dd75e9-3d4a-48e2-bafc-3c8f95a8f0d1",
      "name": "John",
      "highscore": 322
    },
    {
      "id": "f9a9af78-6681-4d7d-8ae7-fc41e7a24d08",
      "name": "Bob",
      "highscore": 21
    },
    {
      "id": "2d18862b-b9c3-40f5-803e-5e100a520249",
      "name": "Alice",
      "highscore": 99332
    }
  ]
}
```

Debug: Get All Users

Ok, so we didn't quite have time to figure out how people actually would get friends in the game yet. So for now the game uses this request to get **all users** in the system so the player can select who of these that he wants to be friends with.

Obviously, this will not scale at all; we will fix this later. For now, just implement it as is. Feel free to improve this if you got time.

GET /user

RESPONSE EXAMPLE:

```
{
  "users": [
    {
      "id": "18dd75e9-3d4a-48e2-bafc-3c8f95a8f0d1",
      "name": "John"
    },
    {
      "id": "f9a9af78-6681-4d7d-8ae7-fc41e7a24d08",
      "name": "Bob"
    },
    {
      "id": "2d18862b-b9c3-40f5-803e-5e100a520249",
      "name": "Alice"
    }
  ]
}
```