Creating a Virtual Reality Meditation Visualisation System

Server Documentation

Rosie Bartlett

February 8, 2023

Introduction

This document will outline the server used to store user data. This is not internal documentation, but a more general overview of database design and public API for use as reference.

Overview

The server is split in two parts:

- API call handler
- Database storage

These will each be handled separately with references made in the former to the latter.

Contents

1 API Call Handler

The API call handler is written in Rust using the Actix crate. Database interaction is handed with `tokio-postgres` and `deadpool-postgres` used for database interaction.

At the moment this only handles API requests but can be expanded to include a simple frontend to allow for downloading relevant data with user validation.

Crate	**Version**	**Features**	**Usage**
`actix-web`	4	Default	Web request handling
`futures-util`	0.3.26	Default	Logger trait requirements
`chrono`	0.4.23	`serde`	Datetime handling
`serde`	1.0.152	`derive`	Serialising and deserialising structs into and out of JSON
`dotenv`	0.15.0	Default	Adding variables specified in a `.env` file at run time to the environment variables
`config`	0.13.1	Default	Database configuration
`deadpool-postgres`	0.10.2	`serde`	Database interations
`tokio-pg-mapper`	0.2.0	Default	
`tokio-pg-mapper-derive`	0.2.0	Default	
`tokio-postgres`	0.7.6	Default	

Table 1: API call handler dependencies

1.1 API methods

The following gives the public API paths with the general form:

1. Description of the request
2. Request information and response codes
3. Expected request data
4. Successful response data format

All API paths require the `API_KEY` cookie to be set with the correct API key. All data should be sent as valid JSON not JSON5 sorry. Any data in JSON5 will not be able to be serialised and result in an unprocessable entity (422) response from the API call.

Bug reports should be submitted to the GitHub repository.

1.1.1 Adding a new user

Add a new user to the database. User will be assigned a unique user ID (UUID) by the server upon a successful request.

Request type	POST
Path	`/api/new`

Response	**Reason**
200	User added to database
403	Missing or incorrect API key
409	Username already exists
422	Request body cannot be serialised
500	Internal server error. Please submit a bug report

Expected data format:

```
{ "uname": "usename" }
```

Usernames are stored as a `varchar[30]` so will be truncated to 30 characters. For more detail see section 2.1.

Successful returned data format:

```
{ "uname": "stored username", "uuid": "integer UUID" }
```

1.1.2 Submitting session data

Submit session data for a given user.

Request type	POST
Path	`/api/submit`

Response	**Reason**
200	Session recorded
403	Missing or incorrect API key
410	UUID does not exist or has been removed
422	Request body cannot be serialised
500	Internal server error. Please submit a bug report

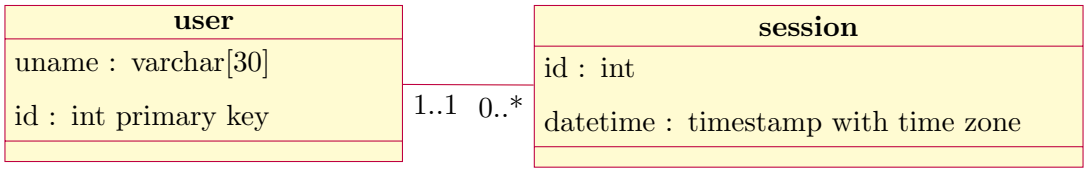
Expected data format:

```
{
  "uuid": "integer UUID",
  "time": "start time in RFC 3339 format"
}
```

No data is returned from this path

2 Database

The server uses a PostgreSQL 13 database with the following schema



The tables are generated with the following

```
CREATE TABLE users (
    uname    varchar(30) unique,
    id       int primary key CHECK ( id >= 0 )
);
CREATE TABLE data (
    time_start  timestamp with time zone,
    id         int REFERENCES users,
);
```

2.1 A note on data types and security

The type used to store usernames is a `varchar(30)` meaning 30 characters at most. This means that assigning a username with more than 30 characters will result in a truncated username being stored which, if not properly handled in a frontend could result in user account collisions; that is allowing different usernames to be treated as the same because the truncated versions are the same.