Contents

[Problem 2](#_Toc127020810)

[Assessment Rules 2](#_Toc127020811)

[Software Requirements 2](#_Toc127020812)

[Thoughts: 2](#_Toc127020813)

[Frontend (REACT) 3](#_Toc127020814)

[Game State 3](#_Toc127020815)

[State Variables 3](#_Toc127020816)

[GUI 3](#_Toc127020817)

[/Game/ 3](#_Toc127020818)

[/Saves/ 4](#_Toc127020819)

[Backend (FastAPI) 5](#_Toc127020820)

[Relationship Overview 5](#_Toc127020821)

[Database 5](#_Toc127020822)

[Table 5](#_Toc127020823)

[Requests 6](#_Toc127020824)

[/Game/ 6](#_Toc127020825)

[/Saves/ 6](#_Toc127020826)

[Responses 7](#_Toc127020827)

[/Game/ 7](#_Toc127020828)

[/Saves/ 7](#_Toc127020829)

# Problem

The problem presented is to code a backend and frontend solution to the game “Rock, Paper, Scissors”.

# Assessment Rules

* No outside help.
* No copy-pasting from internet.
* Timebox to 3 hours (Recommended)
* Submit as Github repo.

# Software Requirements

* Allow input to get names of two players.
* Two modes: Player vs Player, Player vs Computer.
* Players will take sequential turns to choose their “weapon.”
* After both players go, announce winner and increment scores.
* Support saving games.

# Thoughts:

My initial steps were to rewrite the problem as I understood from the instructions provided. I also took the liberty of assuming some system requirements based on the wording of the requirements provided, such as the existence of at least two game modes.

Once I had my list of requirements, I proceeded to make mockups of the web app GUI. By getting the GUI planned out, I would then be able to see what functionality would need to be supporting, but also how to break down the GUI into smaller components.

With my diagrams and plan in hand, I decided to tackle the backend as that is where my strengths lie. I choose FastAPI because it is a Python based framework that I had recently been studying so it was the freshest in my mind. It also has a built-in interactable frontend that can be used for debugging.

I implemented the backend using a standard FastAPI RESTful setup for handling the saving feature. Regarding the game logic, I attempted to cutdown on the resources and time taken during comparison operations by primarily utilizing numerical variables.

Unfortunately, when it came time to code the frontend, I found myself running low on time. Most of my time went to planning and the backend so I was only able to startup a boilerplate React app and create my component files. I am also a bit rusty when it comes to frontend work, so it was taking me a little more time to get up and running to begin with.

Given more time, I would have like to finish frontend portion of assessment. However, I would have liked to add password functionality to the save files so that only the user who created the save file would be able to access it. This could be implemented by hashing the password the user submits as part of the POST request and inserted into the record. Upon a GET request, the user would have to supply the password and its hash would be checked against the record. Additionally, I would have like to cleanup the UX side of things.

# Frontend (REACT)

REACT will be the framework of choice as that is what I have experience in from doing personal projects and taking online courses.

## Game State

The game will need a client-side state. The state should be resettable to allow for the switching of game modes dynamically (on click event). When loading a save file, the state variables will be populated from the GET response from the backend.

When the user chooses to save their game, the state variables will be passed to the backend as a POST request. The backend will only care about the state variables when a “Save” is done.

The backend will handle the logic of the game based on the game mode and the player “weapon” choices.

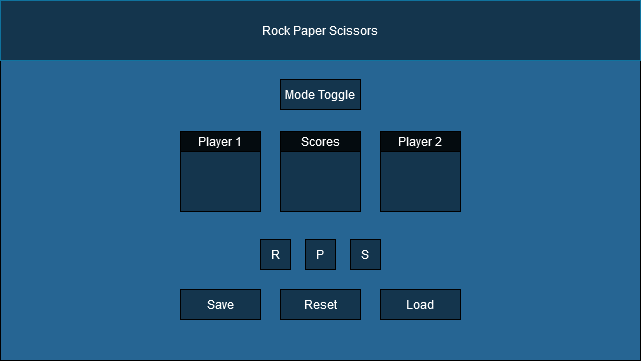
### State Variables

* Game Mode: PvP/PvC
* Active Player
* Player Names
* Player Scores
* Id (filled out when loaded)

## GUI

The GUI for will be composed of two pages. One primarily for playing the game and the other to handle save files.

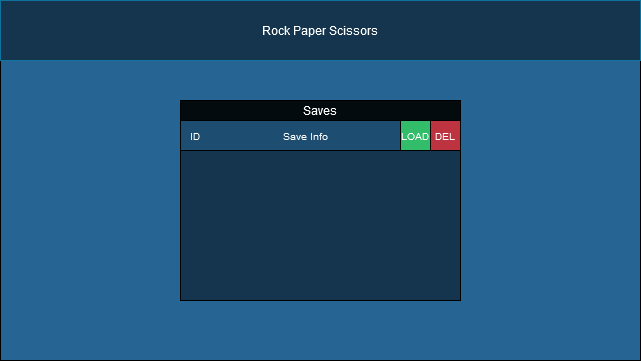
### /Game/



#### Components

* Header
* Mode Toggle: Click event shared with Reset Button
* Player
* Score Card: Display number of wins and match result message
* Weapon Choice
* Save Button
  + POST Request: if game state not loaded from save
  + PUT Request: if game state loaded from save (ID will not be NULL)
* Reset Button
* Load Button: Reroute to /Saves/.

### /Saves/



#### Components

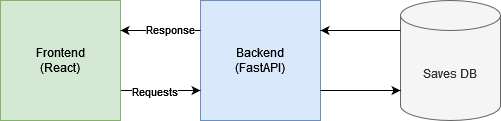
* Save List (Container)
* Save Item
* Load Button: Same as /Game/ different class?
  + GET Request
  + Routes to /Game/
* Delete Button
  + DELETE Request

# Backend (FastAPI)

FastAPI was chosen due to being the most recent backend framework I have studied. Additionally, it allows for easy integration with databases and RESTful API setups. Furthermore, FastAPI has a wonderful debugging/testing frontend located at: <http://127.0.0.1:8000/docs>.

The backend is to be stateless to adhere to RESTful principles. All game logic will be handled by the backend. Match results will be served to the frontend where the data will be parsed and handled accordingly.

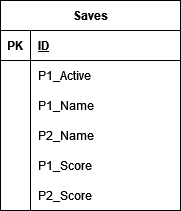
## Relationship Overview



## Database

SQLite will be used as the database because it is lightweight and is natively supported by Python. Due to the nature of the problem, only one table will be used for Save files. An autoincrementing ID will be used as the Primary Key.

### Table



The Primary Key will be autoincremented, the rest of the data will be supplied by the frontend when saving.

## Requests

The following are the requests and response data formats.

### /Game/

GET: Get the result of the current match in a game

Graphical user interface

Description automatically generated

### /Saves/

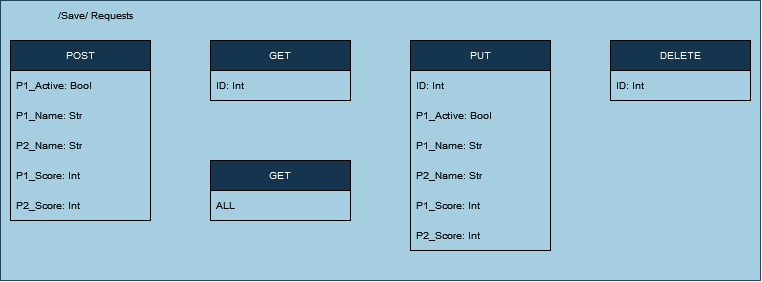
POST: Save game state to database

GET: Get save file with matching ID

GET (ALL): Used to populate save list.

PUT: Update loaded save file

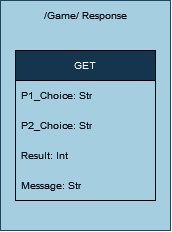
DELETE: Delete save file with matching ID



## Responses

### /Game/

GET: Match result



### /Saves/

POST: Copy of game state forwarded

GET: Save file record

PUT: Copy of updated game state used for updating a save file record

DELETE: Save file deleted

