Leaderboard Documentation

There are several pieces to this project. This document intends to cover what each piece is, how to use them, and how they may interact with one another.
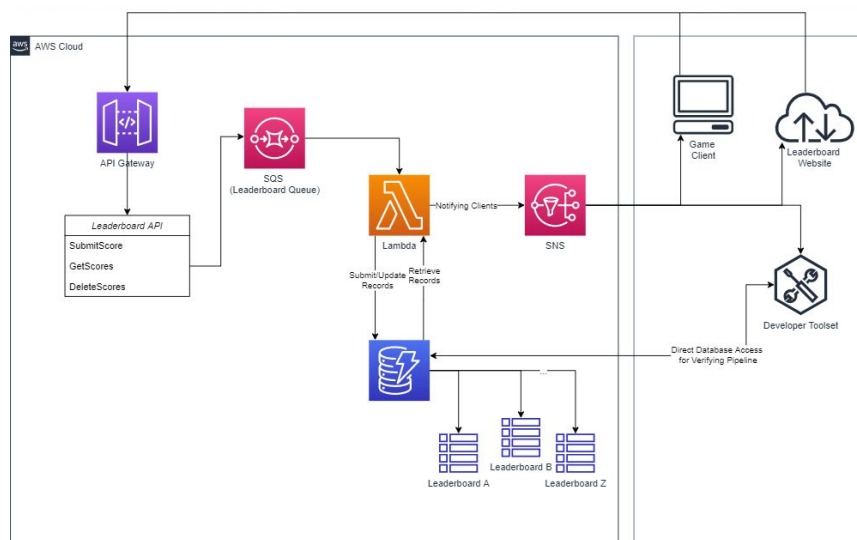
The hierarchy of the project looks as follows:
- LeaderboardLambda
  - This folder contains the Python server as well as a quick use test client.
- LeaderboardWPF
  - Contains the LeaderboardWPF solution, can be used to run a tool to interact with a running Leaderboard server.
- Postman Tests
  - Contains a Postman json file with several preset API calls and corresponding tests to interact with the server
- Whiteboard Photos
  - Contains a few images of my whiteboard detailing my thought process towards the beginning of the project

The submitted project has the following requirements:
- Python 3.8.2
- TinyDB (installed via 'pip install tinydb')
- .NET 4.7.2

The are two parts for the vision of this project: what it may look like as a project and what has actually been submitted.

AWS Vision

The diagram can be found here: (https://drive.google.com/open?id=1E9qsAYA-vTiRAWm9hfcu1YXTqUAMJoHh)

The vision for a "productized" version of the Leaderboard involves creating a communication pipeline in the cloud (in this case, AWS).

API Gateway acts as the communication point between any clients and the rest of the pipeline. The API would be defined in API Gateway and would use SQS as an integration point (though, perhaps not, more on this later). Lambda acts as a serverless server, responding to messages in the SQS queue as expected. Lambda stores and retrieves data from Dynamo (or other database/storage solution) and uses SNS to pass data back to clients if requested.

The case for SQS (but really against it)
Using SQS here turns this pipeline into an eventing queue, rather than a more traditional web service set up. The initial reason for this is because API Gateway requires a specific implementation for integrating with varying AWS services. A direct connection between Lambda and API Gateway does work, but if for some reason the decision were made to stop using Lambda and change to a different AWS service (or something outside of AWS), then the API Gateway integration would have to be changed. This runs the risk of breaking the API or changing its functionality.

However, using SQS in this scenario makes getting data back out of the pipeline a pain. Clients would need to subscribe to an additional SNS feed and wait after they make their request to get the event back. This means the clients have to implement a lot of extra code that is unnecessary.

In reality, it would be better to map the API to varying Lambda functions directly. Different endpoints can be mapped to different functions and the result from the function can be returned immediately back to the client and there's no need for an additional notification step. While dropping Lambda can result in some pain in reintegrating, it's likely not worth sacrificing what is gained by not using a queue.

The Actual Submission
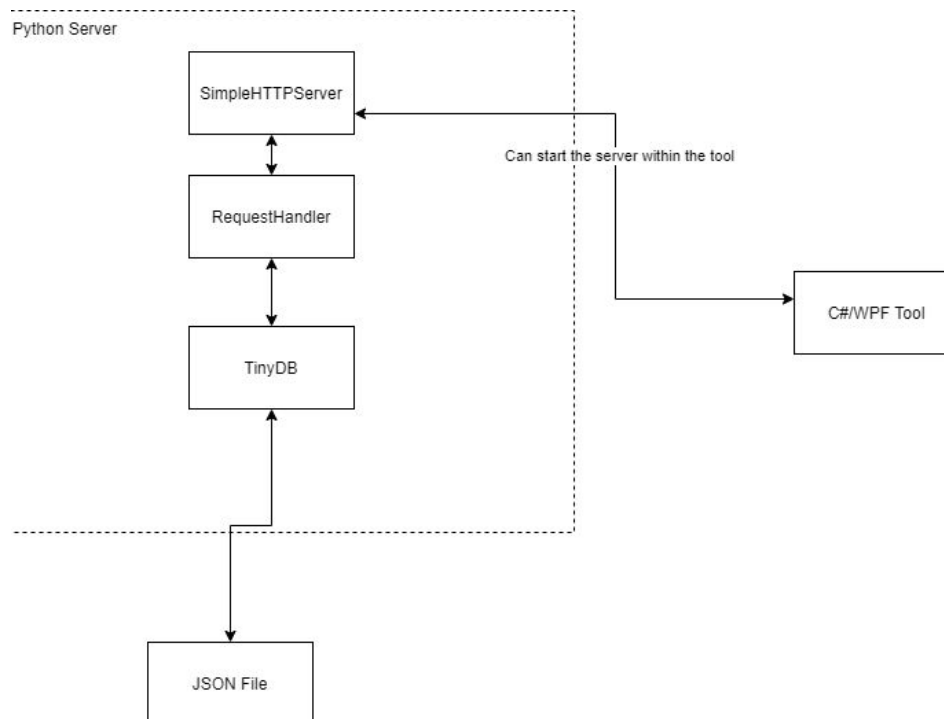
The following diagram can be found here: (https://drive.google.com/open?id=1KSCUMxaeo6uEcx4hTQezD9BZIqfDgAC7)

The submitted project is overall much simpler than the "product" vision, but tries to mimic the pipeline the best that it can.

The Server (LeaderboardLambda)

The server that handles the leaderboard requests is written in Python 3.8.2 and uses tinydb as its storage layer.

The basic loop of the server is simple. TinyDB is initialized with either a default or argument provided json file, request handlers are created, and a python HTTPServer is started. It serves forever until a keyboard interrupt is detected.

Running the Server

The server can be started as follows:

```
python leaderboard.py
```

This will start server with the default host (localhost), the default port (8080), and the default path for the json database (database.json).

The server supports the following arguments:
- --host (str)
    - The hostname to start the server with
- --port (int)
    - The port to start the server with
- --database (str)
    - The filepath to either an existing json database or where to create a new one

An example of starting the server with these arguments is as follows:

```
python leaderboard.py --host localhost --port 8080 --database database.json
```

Server Overview

This section details a general look at the areas of interest for the server.

The bulk of the work done by the server is done in a series of classes derived from a base RequestHandler. Each request handler maps to a specific action in the API (ie GetScores, SubmitScore, etc). The handlers implement a HandleRequest function that validates the request data, performs any necessary queries on the database, and returns a response.

If an error occurs, the exception is caught, logged, and an error response is sent back to the client.

The Leaderboard Viewer (LeaderboardWPF)

The viewer is a WPF application that is intended to replicate what a tool might look like for developers working with the Leaderboard server.

The solution contains two projects:
- LeaderboardWPF
    - The WPF application and the varying views and view models needed
- SharedLibrary
    - A small library of common elements that was ported over from a personal project. At one point, SharedLibrary contained many base elements one would use across several WPF projects.

The viewer is broken up into two pieces: a view to connect to a running server and interact with it and a view to start the server and watch its output (if desired).

Leaderboard View

The Leaderboard Tab (not to be confused with the other Leaderboard tab, whoops) contains the UI to interact with the server.

File Menu
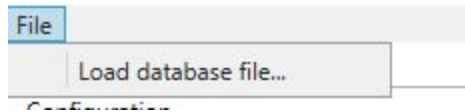
The only option in the file menu is the load database item. This allows the user to parse a server's JSON database file and display it in the Database tab. This is intended to allow users to see exactly what is in the database and ensure it is updated properly when commands are made.

Configuration

The configuration section allows the user to specify the URL of the server. If the host is empty and the user attempts to interact with the server, nothing happens (there should probably be an error message). If the host is invalid or the host cannot be connected to, an error will display in the Server Response section.

API Calls
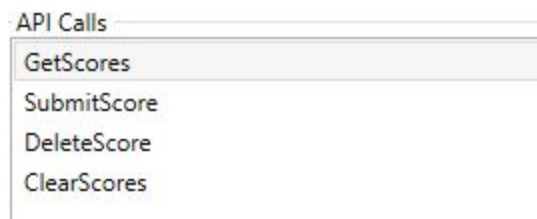
This ListBox contains all API request definitions that have been implemented. These derive from a RequestDefinition base class and are automatically added to the list at start up. Selecting one of the items in the list updates the section below.

Selected API Actions

This area allows the user to configure their API call before sending it. Blank options are nulled where possible and otherwise have default values as expected. The tool only does minimal amounts of verification of the data, rather allowing for bad requests to be made and tested.
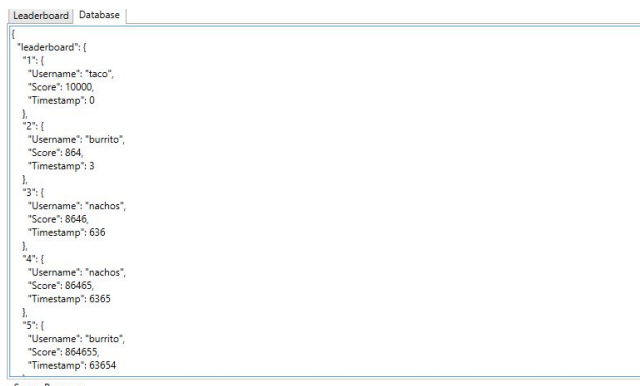
The 'Send' button sends the API call to the provided host. The 'Refresh Database' button reloads and parses the JSON database file if one was provided from the file menu. Additionally, any time an API call is sent the database view is automatically refreshed.

Leaderboard View

The Leaderboard tab (not to be confused with the other leaderboard tab) is automatically updated when the GetScores API call is made. Other API calls do not reflect changes in this view. The user can locally sort any of the columns.
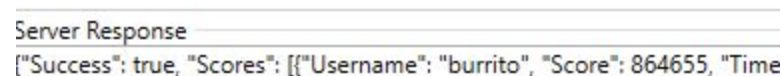
Database View

This view shows the JSON data in the file provided by the user. It is automatically updated whenever a API call is made and can be manually refreshed in the UI. The TextBox in this tab is set to readonly, the database cannot be directly edited in this tool
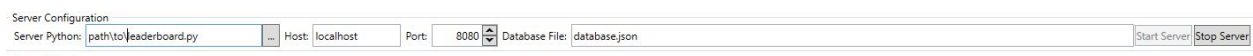
Server Response

This section contains the JSON response from the server when API calls are made. It does not keep a timeline, rather it overwrites whatever was in the TextBox previously. Additionally, in some cases handled exceptions are also displayed in this TextBox when a user's action fails locally in the tool.

Server View

This tab allows the user to manually start the Leaderboard python server.

Server Configuration

This section allows the user to provide the location of the 'leaderboard.py' file and set the arguments for the server. When 'Start Server' is clicked by the user, a separate python process is started and its output is redirected to the server output section. Python is currently assumed to be in the PATH environment variable. The user can stop the server by clicking 'Stop Server' or by closing the tool.