Nucamp Python Fundamentals

Final Report

May 18, 2022

Gaming Mini-Server with Quiz

Game Mini-Server Version 2.0

As a follow-on from version one, version 2.0 contains a custom, Python-based HTTP Server framework. A networking module was updated to include the modeling of HTTPRequests and the required data structures for responses. The Game Server allows registration of users as well as authentication for registered users. The current implementation does not use database persistence for games or user data.

The architecture consists of a Service model, which allows a single Game Service to encapsulate a user authenticated to the Game Server. This model allows for session control, which was necessary due to the stateless nature of HTTP. Requests to the Server are routed on RequestRouter threads, which either distributes the request to an existing Service object or has the Server create a new GameService object.

The server will render HTML to a browser, which has been tested with Firefox, Google Chrome, and Safari. Static HTML documents are utilized as templates, which are dynamically populated with server and game result data. All game logic is implemented with Python. In this architecture, games may be written with separate logic and view classes to implement game play and visualization.

The Game Server incorporates a Game Engine, which uses reflection and introspection on Game and Game View classes to dynamically load games and conduct game play. Game data is stored in an init file in a json-encoding format. The architecture allows for instantiation of the Server with DataAccess objects, for persistence and data retrieval. With this polymorphic nature of this architecture, database access could be achieved with derivation of database-specific classes for the DataAccess objects.

A unique messaging model was created to allow a consistent interface for communication between the Game and Game View objects. Upholding the messaging interface allows simplified construction of separate Game and Game View classes. The Game Server processes requests based on CRUD interfacing. At this stage, only GET request processing is implemented.

This project does **not** rely on any outside, published web framework, such as Flask or Django. While such frameworks for web servers and http requests/communication are readily accessible, the purpose of this project was to delve more deeply into the socket and threading modules of Python.

Quiz Version 1.0

Quiz is a web-based game built for the RJM GameServer architecture. It renders a quiz to a web browser, which allows the user to navigate based on the logic of a server-based. A separate terminal/console-based tool, QuizMaker, allows for the creation and modification of Quizzes. Between runs of the Quiz, the quiz to be played can be modified by QuizMaker. These changes will be reflected in the next play of Quiz. Please NOTE, at this stage of implementation, only one quiz file is used, called test\_quiz.

During play, the user can answer, skip, or return to questions. To submit the quiz, the Submit Quiz button should be pressed. Upon submission, the results are determined by the Python-based Quiz object on the Game Server. The results are rendered to the browser, after which the player may exit and return to the menu.

Installation

To ensure that the Server, Games, and GameMaker work properly, the PYTHONPATH must include the following. NOTE the “install” directory is a directory in which rjm\_games\_V2\_0 is contained.

/install/rjm\_games\_V2\_0/rjm\_gaming

/install/rjm\_games\_V2\_0/games

QuizMaker is in and should remain in the directory /install/rjm\_games\_V2\_0/games/quiz/tools

Architecture

The main package is rjm\_gaming.

**rjm\_gaming modules:**

config.py

class Config

game\_authentication.py

class ServerClient

class Authenticator

game\_base.py

class GameInvalidError(Exception)

class GameResult

class Player(ServerClient)

class Game(abc.ABC)

class ClassInitMeta

game\_engine.py

class GameEngine

game\_network.py

function paerse\_query

function remove\_http\_pluses

class HTTPStatusCode(enum.Enum)

class HTTPHeader

class HTTPCommsModule

class HTTPSession

class HTTPRequest

game\_registry.py

class Registry

game\_server.py

class ServiceClosedError(RuntimeError)

class ServerClientService(threading.Thread)

class GameClientService(ServerClientService)

class GameServer

class RequestRouter(threading.Thread)

game\_utilities.py

class GameCommsError(Exception)

class DataAccess(abc.ABC)

class FileDataAccess(DataAccess)

class ClassLoader

game\_view.py

class GameView(abc.ABC)

**games modules:**

quiz.py

class Question

class Answer

class QuizSubmission

class Quiz(Game)

quiz\_view.py

class QuizView(GameView)

Flow of Control

A GameServer object is instantiated, creating a GameEngine object of which it is composed. The GameServer listens on a port entered by the manager of the server at start up. Once a connection is received, a RequestRouter object is instantiated, which acting as its own thread (though still bound by the Global Interpreter Lock, as it is mostly non-I/O processing), routes the current request. The RequestRouted instantiates an HTTPRequest object. This object serves to instantiate an HTTPCommsModule object from the connection it is passed in its initializer by the RequestRouter. The HTTPRequest uses the HTTPCommsModule to read the request sent to the server.

The Request Router object distributes the request to the proper ServiceClientService object, through the GameServer object. If it is an initial request (one without an assoiated session), a new GameClientService is created. Also with this initial request, because browser tend to send a favicon.ico fetch request, the HTTPCommsModule kills this at the proper time. The favicon request is killed after the login page is sent to the client in the ServerClientService base class \_\_init\_\_ method. A session for this browser, is also set at this time.

After successful authentication, the GameClientService sends appropriate HTML reponses, populated with dynamic data as needed, based on the continued requests of the client. Only the client with the associated session receives the response.

The Game Engine instantiates both the Quiz and QuizView objects without explicit reference to either. It relies on reflection and the init files to create these objects. Duckt typing is utilized to play the game object, once the start request is sent to the server by the user.

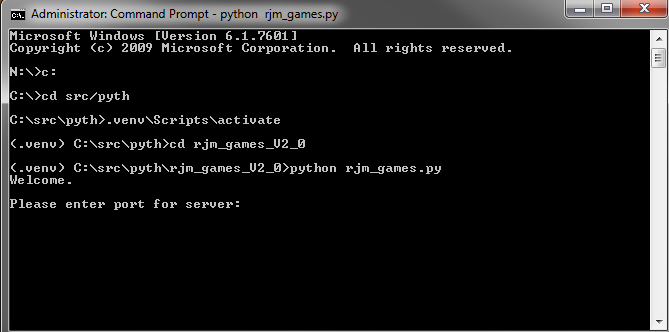
All of the GET requests from the client processed by the GameClientService object. Requests are placed into the request queue by a RequestRouter object. The GameClientService loops continuously until a request arrives or if one remains at the head of the queue. It is then directed to the proper handler.

Quiz game play occurs through one of these handlers. The handler is written to be non-specific, such that, if any game is created to send the proper request format from the browser, it will be handled. The polymorphism of the Game and GameView subclasses applied here.

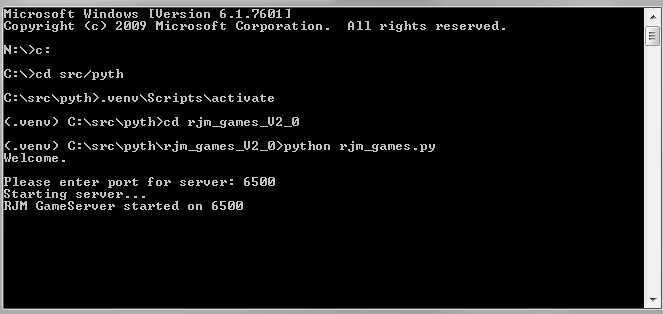
One other interesting feature built into the Game architecture is the ClassInitMeta class. This allows for dynamic interpretation of the \_\_init\_\_ methods of classes instantiated reflectively. This class, as well as the Game, GameView, and Question class, utilize json encoding and decoding for serialization. A unique approach of creating an “encoding” property on these classes allows for easy translation to json-required data. Inversely, a static method “decode” on these objects can be passes to json.loads as the object\_hook key-word argument, which instantiates an object of the proper type.

Demonstration Images

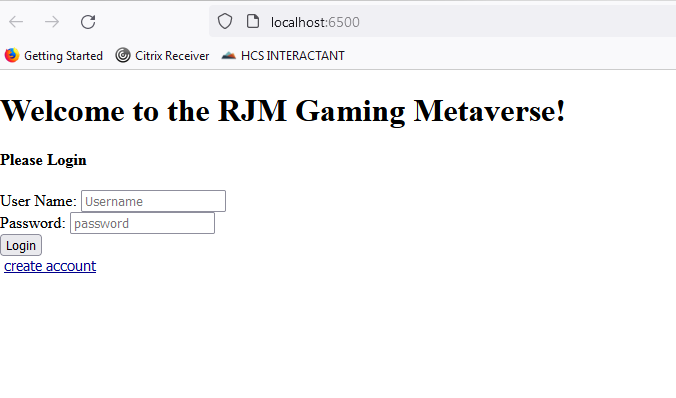
Starting the server:

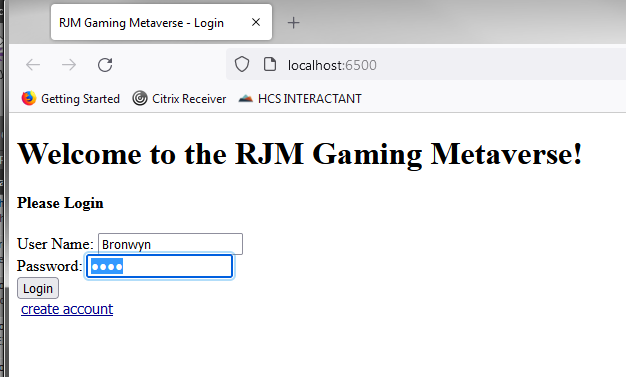


Setting server port:

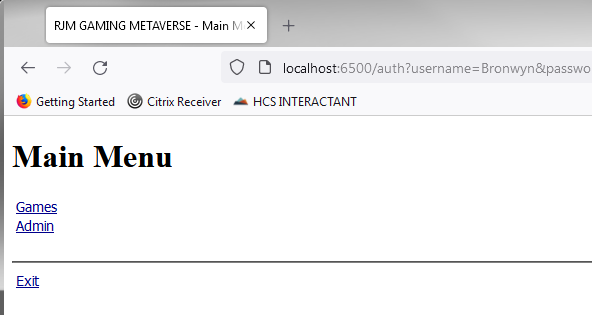


Login screen:

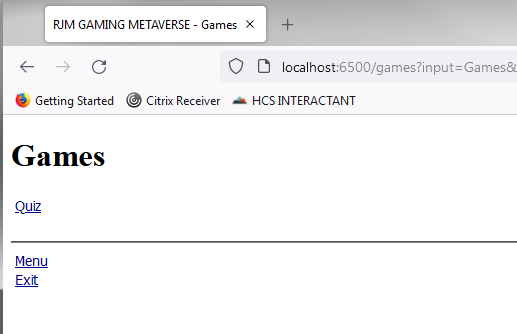




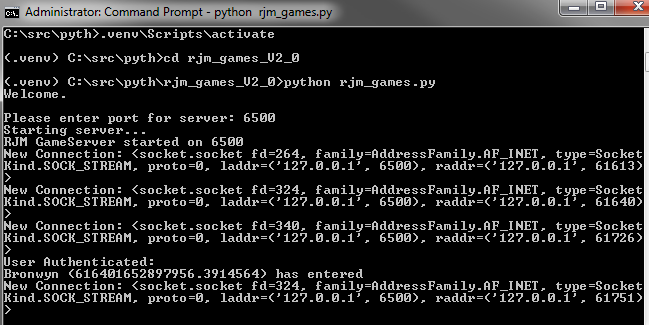
Post login, main menu:



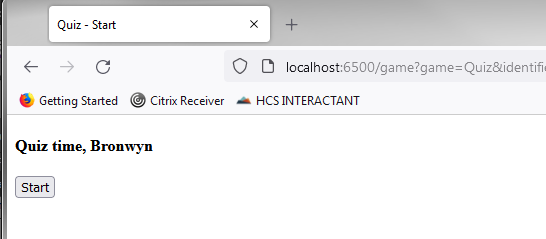
Game menu (Games dynamically loaded, based in init file):



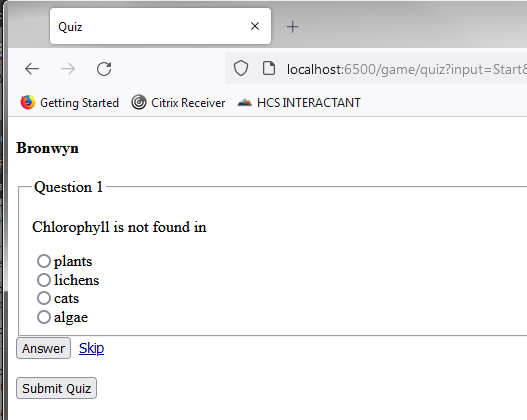
Server being hit by requests:

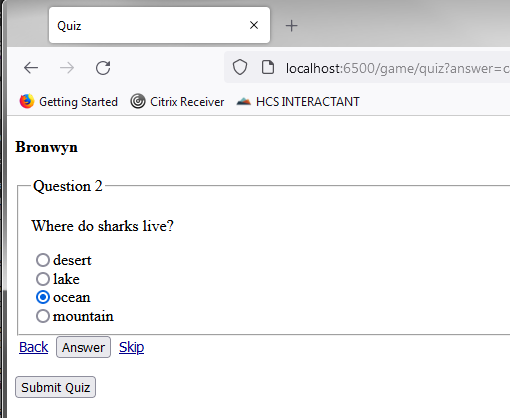


Quiz start page:

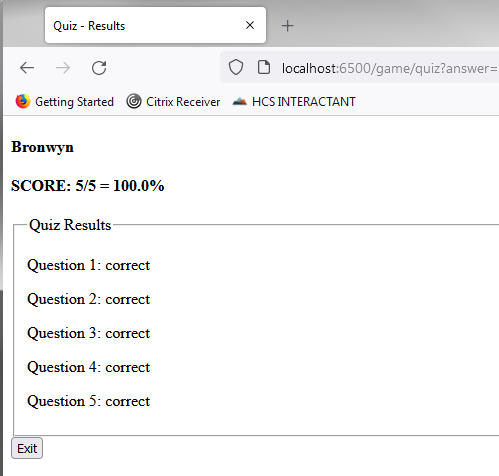


Quiz questions:





Quiz results:



QuizMaker:

