CSC 667 Term Project Documentation

Github Link : https://github.com/csc-667-fall-2022-sfsu-roberts/term-project-UnoDosTres

# Uno,Dos,Tres

11/16/2022
(Uno Card Game)


Team Members :

Michael Almeda

Julio Murga

Jonathan Chen

Yuwei Liu

Within our documentation we will describe the usable features within our project by using a table for specification
First we will list all actions a user can perform that your application will need to handle
a. Second we will Enumerate the inputs for the action
b. Third we will define the outcome of the action

Actions that a user can perform with our platform:

- Create a game by player
- Shuffle cards by dealer
- Deal cards by dealer
- Draw cards by player
- Drop cards by player

Endpoint Table:

| | Inputs | Post Condition | API Endpoint |
|---|---|---|---|
| Play a Card | card_id, player_id, game_id | If<br>1. player_id is in game_id,<br>2. it is player_id's turn,<br>3. card_id is in their hand, and<br>4. playing card_id is a legal move,<br>Then<br><br>1. the discard pile will be updated with card_id,<br>2. the next player will become the current player<br>3. all users in the game will receive their updated game state. | POST /games/:id/play<br>Body:<br>{ card_id }<br>(game_id is in the URL, and player_id will be in the session since the player will be authenticated) |

| | | | |
|---|---|---|---|
| Create a game | player_id, title, number_of_players player_id, title, number_of_players | 1. A new game is created<br>2. The game list in the lobby is updated to show the new game<br>3. The player is redirected into the game room | POST /games/create Body { title, number_of_players } |
| Deal cards | card_id, dealer_id, game_id; | 1. dealer_id is in game_id<br>2. It's dealer_id's turn to shuffle cards<br>3. Shuffle card_id is legal | POST/games/:id/Shuffle Body: { card_id } |
| Draw cards | card_id, player_id, game_id; | 1. If player_id is in game_id<br>2. Card_id is in their hand | |
| Drop cards | card_id, player_id, game_id; | 1. Update the discard_pile with card_id | |

Rubric:
Code Quality:good
Documentation: completed
Functionality -server: completed
Functionality -responses: completed
Functionality-gameplay:completed
Functionality-config:completed
Functionality-render:completed

Description of Architecture::
For our Uno Term Project, we created multiple classes that represented most of the milestones given to us to implement and test. The classes in the project are separated by packages grouped to serve specific purposes. Specifically, the project contains the “App” folder containing the files for configuring our game and reading the configured files; the “views” folder for multi-threads in order to the “Milestones” folder for milestones storage and responses to the console after requests are received; and the “migrations” folder for reading migrating files to render files. The server starts with the HttpServer class containing the main class necessary for us to run our code. Inside the class, we created a Configuration object for us to read a resource file necessary to run the program. The Configuration class contains our logic for reading and parsing files; after reading our resource file, we return a new instance of the class similar to Singleton, then get the port and webroot for our server. We created a Server on render and ConnectionThread class to initialize the socket we will use for the server (port 8080). The ServerThread, extending the Thread class, consists of a run method where we create a new ConnectionThread object in order to retrieve the requests and process them accordingly. The ConnectionThread class contains two functions that represent these respective tasks, along with a constructor to initialize the socket, requests and redirects. In the first method we read lines from the request, split the requests into subarrays, then store the values of the requests in key/value pairs inside the application. The second method parses through these pairs from the application to the uno game to see if it’s valid, then returns a response accordingly. The ConnectionThread class also contains multi-threading for returning these responses. Finally, the Application folder contains our classes to parse Application requests along with classes to return the Application responses. For returning the status code numbers, we created the HttpStatusCode class with a public enum of the responses which gets returned via a Status Code constructor, and for parsing the requests we took advantage of the InputStream, InputStreamReader and StringBuilder functionalities, checking to see if the responses were valid. We tested our server via Render and through our

website, and besides an error in game.pug which was having trouble linking our game to the lobby, home and other classes explained below in our Problems discussion, the server was responsive with our requests via GET, POST and HEAD. We didn't get to test our application due to our time constraints.

Problems we encountered and discussion:

When deploying our database we had a few struggles

- As mentioned in the specifications there were differences when deploying on different systems such as Windows and Linux
- We ended up using a virtual machine in order to distinguish which one we wanted to use
- Increase meeting times and give tasks sooner into our projects
- Improve our communication within discord and hold each other accountable to deadlines prior to our due dates
- Since none of our team had much experience using (PUG) we had a bit of trouble when implementing things such as the lobby and more so when implementing our game.pug

How we solved our errors:

- After realizing how much trouble it was to run it locally on everyone's locally
- Our way of fixing this issue was through using a virtual machine that would run our project since some of our OS's were not operating properly
- After using multiple OS's we were finally able to to run the application locally through Virtual machines
- After some trial and error we were able to implement the game.pug page by linking it to our other pages such as lobby and our main application

Game pug
render

Our test Plan:
We will use render to test our server and we also need the public.html file given to us, the server responds to our request via GET, POST and HEAD.