CSC 667/867 Internet App Design - Term Project

UNO Multiplayer Card Game

Final Documentation

Submitted: December 14, 2020

Developed by:

Akhil Gandu, Allen Sun, Roberto Herman Valdivia & Katie Kennedy

Prof. John Roberts, SFSU

GitHub Repository

Our source code for this project can be found on the main branch at the following link:
https://github.com/sfsu-csc-667-fall-2020-roberts/term-project-akhil-allen-katie-roberto

Link to Deployed App

Our UNO game is hosted via Heroku and can be played by visiting the following link:
https://stormy-harbor-64543.herokuapp.com

Project Introduction & Overview

Objective

Our goal was to build a real time, multiplayer, online game that supports an arbitrary number of simultaneous games. Our team decided to recreate the classic card game UNO as a web application to be played by four people. We designed our user interface to be as simple as possible, so that joining a game is as easy as making an account and pressing play. Opponents are then chosen for the user to play against based on the time they entered the staging queue. Once four users are in the queue, they are all redirected to the game board and a new game id is created for their game.

Technologies Used

The front end of our application mostly uses grid-style CSS, while the lobby and chat feature use SCSS. The middleware and backend were developed using Node.js, JavaScript, and Express. For the views, Pug was used as a templating engine for HTML. PostgreSQL was used as a database server. All four of our team members used VS Code as an IDE. Development testing was done on localhost port 3000. Final testing of the multiplayer functionality was done after deployment on Heroku.

Assumptions

Our implementation assumes users will play on a desktop browser. Our game also assumes that all games will be exactly four players, though traditional UNO can be played with a minimum of two. There is no instructional feature in this game, so the app assumes users already know how to play. We also assumed that players could go back to the lobby to chat, which in hindsight, probably wasn't the best way to implement the feature.

Scope of Work

Category	Requirement	Completed
Authentication	Users may create an account.	X
	Users may login & log out.	X
	Pages only allow access to those users that should be able to see them.	X
Chat	Chat is enabled in a lobby for all users.	X
	Chat is enabled in each game room.	
Game States	Game state persists in a database.	X
	If a user closes a tab and reconnects to the game, the game is able to be reloaded for that user.	X
	Only relevant game state is sent to each user.	X
	Game state is updated in real time in response to user events and interaction with the game.	X
Number of Games	App supports an arbitrary (infinite) number of concurrent games.	X
	Any given user is permitted to participate in multiple games (in different tabs).	X
Design	User interface is visually appealing.	X

Command Line Instructions to Compile & Execute

To initialize the database before running app:

```
node initialize_db.js
```

To run the app on localhost:3000:

```
npm run start:dev
```

Implementation Discussion

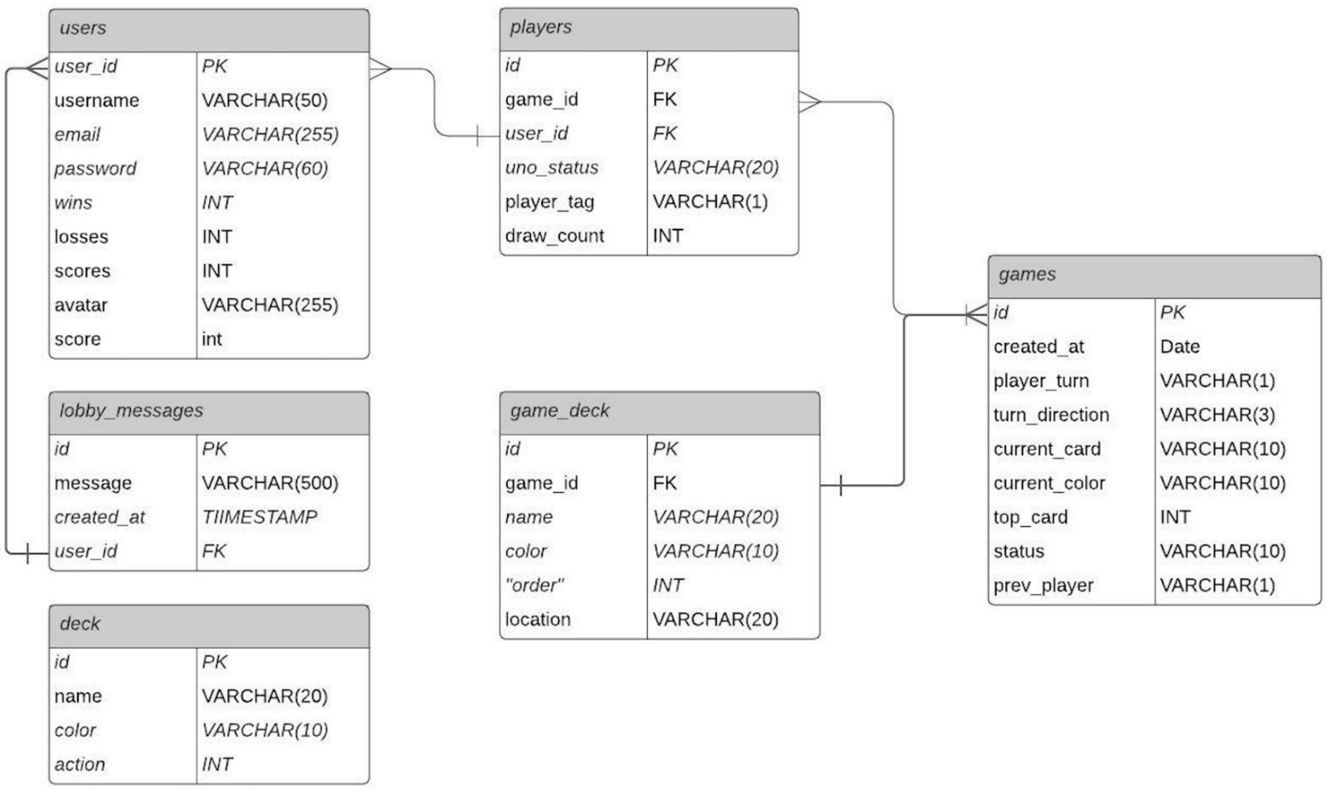
Gameplay

Our source code organization was initiated using the Node.js default file structure. The active game state persists in the database to make sure each game update is distributed to all players. Using cookies and sessions, players may rejoin their active game if they leave the page or close their browser. However, the chat feature does not store messages in the database and instead uses sockets to grab user input and then update the chat window globally with the new message.

Multiplayer games require frequent updates synchronized across all players in their game instance. We implemented state objects and event objects to update the gameplay as players discarded a card, picked up cards, or called UNO. We used an observer design pattern to get events from users, serve them to the database via Postgres, then send back the appropriate game updates to the "subscribers," which are the four game players. Finally, we tried to make sure each service of the app was in its own file to make the code better organized and easier to maintain.

Database Entity Design

The following diagram describes the logical relationship between tables used for our app:



users	
user_id	PK
username	VARCHAR(50)
email	VARCHAR(255)
password	VARCHAR(60)
wins	INT
losses	INT
scores	INT
avatar	VARCHAR(255)
score	int

players	
id	PK
game_id	FK
user_id	FK
uno_status	VARCHAR(20)
player_tag	VARCHAR(1)
draw_count	INT

games	
id	PK
created_at	Date
player_turn	VARCHAR(1)
turn_direction	VARCHAR(3)
current_card	VARCHAR(10)
current_color	VARCHAR(10)
top_card	INT
status	VARCHAR(10)
prev_player	VARCHAR(1)

lobby_messages	
id	PK
message	VARCHAR(500)
created_at	TIIMESTAMP
user_id	FK

game_deck	
id	PK
game_id	FK
name	VARCHAR(20)
color	VARCHAR(10)
"order"	INT
location	VARCHAR(20)

deck	
id	PK
name	VARCHAR(20)
color	VARCHAR(10)
action	INT

The "games" table contains a primary key called "id" that is used as a foreign key in the players and table to map the correct players to the correct game. Each entry in the "players" table contains a foreign key to each entry in the users table, as all players entries must have a corresponding row in the user table. An instance of the UNO deck is also assigned to a game id via a foreign key in the “game_deck” table to make sure each game is playing with its own deck information. Finally, the chat feature in the lobby maps each message to a user via a foreign key in the “lobby” table.

Results & Conclusions

What We Learned

In the end, we learned to develop a fully functional multiplayer UNO game for the web. Designing UNO for the web exposed how wildly complicated the backend logic is for a simple card game. Our team had to do a lot of reading about JavaScript sockets, grid css design, state management, and much more to make a successful end product.

One of the most important things we learned was how to split each service up into its own file. Decoupling code makes the app more reusable, maintainable, and extensible, and will be a useful skill down the road.

Challenges

One of the first challenges we encountered was trying to design the database entities in a way that made sense logically so we could draft our models effectively.

It also took us a while to map each user to the correct position in the game board UI so that the rotation of the players’ turns were consistent and made logical sense. UNO requires that the rotation be reversed if a “reverse” card is played, so the turn counter was a tad tricky to implement. We also had trouble initializing the game board so that the correct cards were dealt to the correct player’s hand after they were “shuffled.”

Future Work

If we had more time, we could have improved the graphics with card animations and pop-up banners, but we chose to focus on the game logic which was way more intense than we thought it would be originally. We could also add support for playing with as little as two people and with more than four. We originally wanted to allow users in the lobby to create games with other users online of their choice, but our current API only allows games to be created with arbitrary users based on when they enter the staging queue.

Our cheat feature is only included in the lobby, but should also be included in the gameplay UI so that players could chat with other players. We completed the front end for a slide-up chat window, but did not get the backend running in time.

We also could improve the scoring method of the game to be more consistent; currently the player with the lowest score at the end of the game is the winner, but scores listed on the lobby page are organized by high score.

Finally, we need to improve the security of the game should it be deployed to actual users. It is currently possible to cheat in the game by changing the user id via a cookie, allowing them to make moves as another user in the game.