CSC 667 Spring 2021 Term Project

Documentation

YOUNO

Team F

Benjamin Kao

Andrei Georgescu

Yangesh KC

Abishek Neralla

Github Repo:

https://github.com/sfsu-csc-667-spring-2021-roberts/term-project-team-aaby

Application:

https://murmuring-headland-53601.herokuapp.com/

Table of Contents

Project Architecture

Runtime Environment: NodeJs

Framework: Express

View Engines: PUG

CSS Framework: Bootstrap

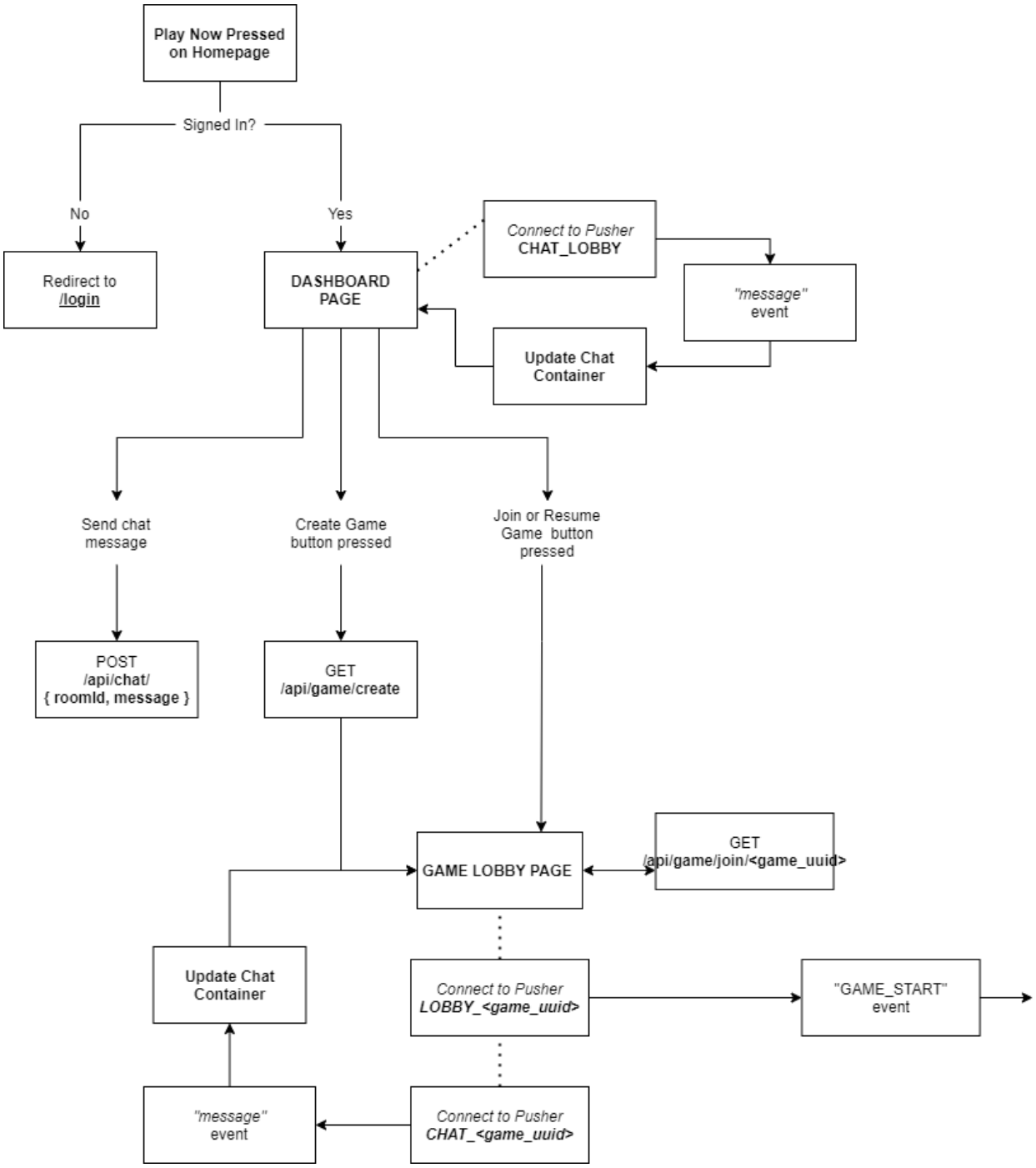
Database: Postgres

Deployment: Heroku

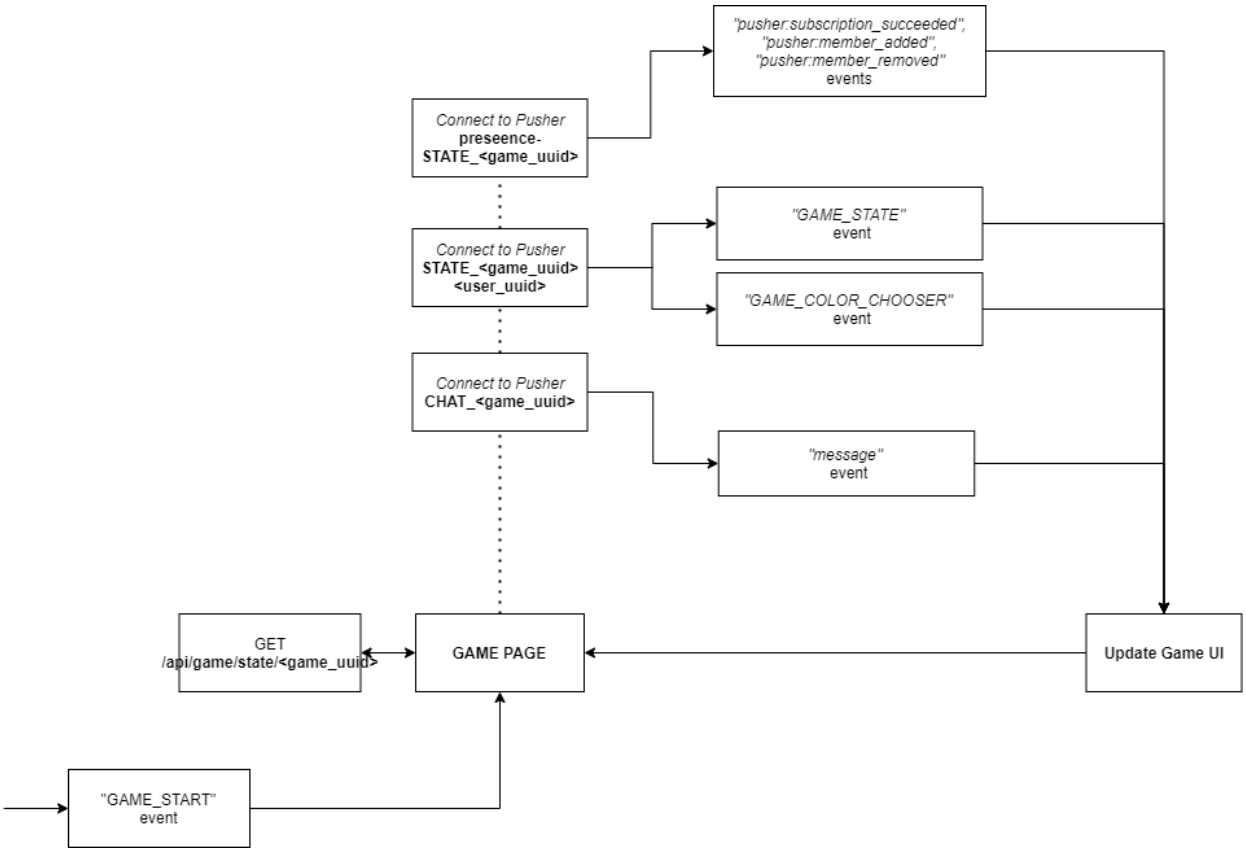
External API: Pusher, PassportJS

Front End

Frontend Architecture and Flow Diagram for the entire application except the Game page.

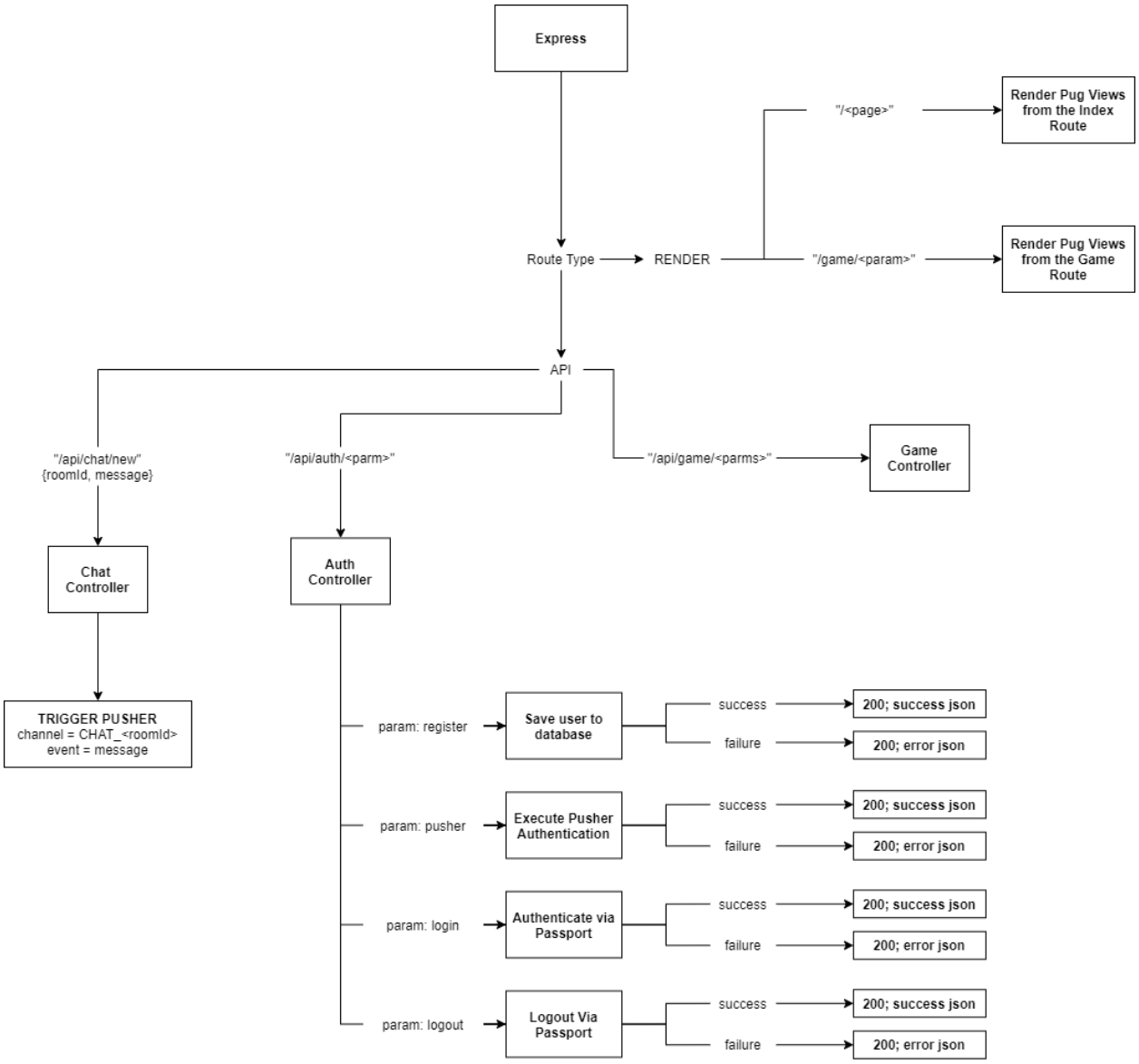


Frontend Architecture and Flow Diagram for the Game Page

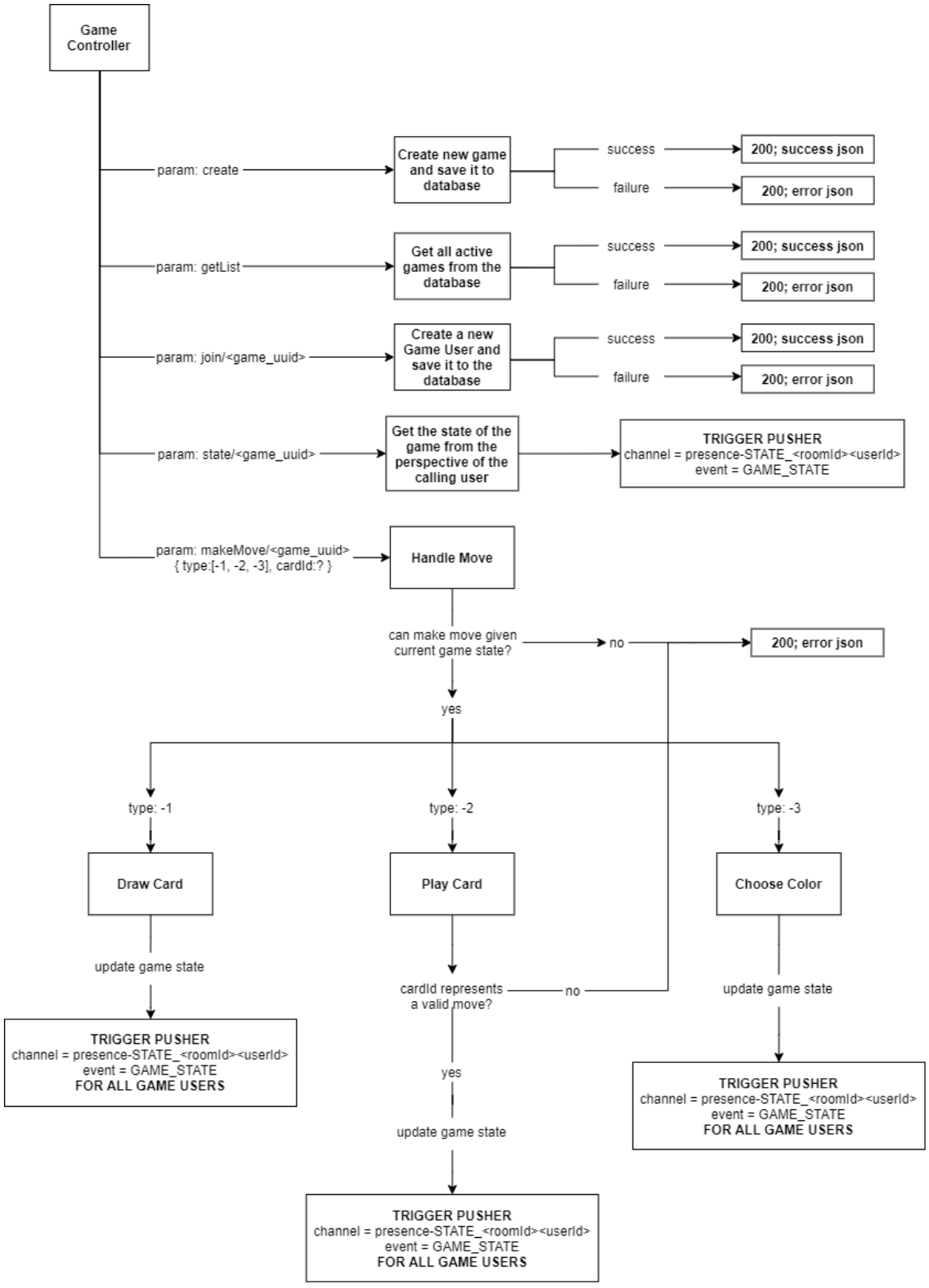


Back End

Backend Website Architecture and Flow Diagram (Not Including Game)



Backend Game Architecture and Flow Diagram



Project Implementation Difficulties and Problems

- **Asynchronous Promise Calls Causing Database Race Conditions**

One problem we ran into was database race conditions caused by asynchronous promise calls and chaining. When implementing the backend logic for the lobby, we noticed that there were a lot of GameUsers stored in the database that weren't doing anything, so we thought it was a waste of storage. This is why we switched to using Pusher's presence channels, thus, we had logic where once four users had joined a lobby, then we would create all of the Game Users and send all users in the game lobby. This way, we could avoid the wasted storage problem. However, a big problem was caused by this. In our Game User table, we have a playerNum attribute that the backend uses to keep track of which Game User will have the next turn. All 4 Game Users were assigned a number 1-4 depending on how many Game Users were already created for the specific game. However, because all 4 users in the game lobby were creating a Game User and we were assigning the Game User a playerNum on insert, this caused race conditions. When we checked how many Game Users there were in the database, the number of Game Users that was returned was not the actual number of Game Users. Due to all of the promise chaining and asynchronous promises, we would get the number of Game Users for one user's API call before another user's API call completed inserting a Game User. Thus, Game Users in the same game had to duplicate playerNum values. In order to solve this problem, we reverted back to creating Game Users on lobby join.

- **Rendering Individual Uno Cards Using SVG File**

Another problem we ran into was rendering individual UNO cards using an SVG file. Because the SVG file was so large, when we tried rendering individual UNO cards using just the CSS background-position style, we could not see the entire UNO card. In order to solve this problem, we used CSS Custom Properties to store the height and width of an individual UNO card that we wanted. Then, because the SVG file contained an 8x14 grid of UNO cards, we figured out the background size of the SVG file had to be changed to (14 * UNO card width) (8 * UNO card height). This made one UNO card the exact side that we needed.

- **Rendering All Game User's Cards with Individualized Game State Data**

Another problem we ran into was rendering the Game State for each Game User using the individualized Game State object they received from the backend. Because we wanted to have the bottom of the screen to contain the user's card hand, coupled with keeping the position of the Game Users consistent with how the database has ordered the Game Users using a player number field, the logic to make the frontend choose which position each Game User's hand would be rendered at was tricky. In order to solve this problem, we all met where we discussed the Game State data structure that the backend was sending and we realized that since the backend was sending all of the other player's hand lengths in a separate array from the user's hand, we could use the modulus operator to implement a circular array logic that would allow us to easily determine the position of each Game User for every Game User's screen.

- **Trouble Deploying on Heroku:**

During the initial phase of our project setup, we had issues with deploying to Heroku where our website could not connect to our PostgresQL Heroku database. Through the help of our classmates in the class Discord server, we figured out that it had to do with not using SSL to connect to our database. After adding an environment variable and some code that made our website not have to use SSL to connect to our database, everything worked perfectly. In addition, when we deployed our entire YOUNO application, we ran into no issues whatsoever.

- **Dealing with Game Logic Edge Cases**

We ran into a few problems when dealing with game logic edge cases such as if there were no cards left in the deck and there was only one card in the discard pile, and the current player could not make a move. Honestly, we could not solve this problem because we don't even know how we would handle this in real life, but since we are assuming that all players are actually playing and not drawing cards just for fun, this specific edge case/game bug should never appear. Another game logic edge case that we ran into was reshuffling the played cards back into the deck when the deck was empty. The problem was that the way we implemented our draw card logic for the "Draw 2" card and the "Draw 4 Wildcard" card, we grabbed 2 or 4 cards from the top of the deck, instead of simulating drawing individual cards like in real life. However, if there were less than 2 or 4 cards when one of these cards were played, there was no way for us to check if the game deck was empty and had to be reshuffled before drawing the rest of the cards. Therefore, in order to fix this problem, which is kind of a hacky way of solving the problem, whenever the game deck had less than 5 cards left, we reshuffled all of the played cards into the game deck. This way, we avoided the possibility of having to draw 2 or 4 cards when there were not enough cards in the game deck.

Testing

No.	Description	Test Input	Expected Output	Pass/Fail
1	**Registering**	Username: “john” Email : john@email.com Password : Uno1011@	Registration Successful. Redirect back to the Registration page with alert message saying that user has successfully registered and must log in.	**Pass**
2	**Logging In:**	Username: “john” Password : Uno1011@	Successful Login. Navigates to Landing Page	**Pass**
3.	**Login Form Validation**	-Password < 8 Character -Invalid Credentials	Flags and Prevents from advancing.	**Pass**
4	**Storing Authentication Data in Session**	Start at home page after login. Press “Play Now” button.	Users should still be logged in and see their name at the top right hand corner of the navbar.	**Pass**

CHAT

No.	Description	Test Input	Expected Output	Pass/Fail
1	**Send Message** **Lobby Chat:**	“Hi”	Message shall be displayed in the Lobby Chatbox for all active users.	**Pass**
2	**Send Message** **Game Chat:**	**“**Hello”	Messages shall be displayed in the game chat box for all active users.	**Pass**
3	**Send Websocket** **Event Message** **to Update Chat** **UI:**	Press the “Send” button with any text in the chat box	UI shall be updated with the message data sent in the Websocket Event	**Pass**

Note: For every test done below, We have "Test input" for "Current Player" and "Non current player". For "Non current player" Expected output is "nothing happens" for every card.

Game

No.	Description	Test Input	Expected Output	Pass/Fail
1	**Game Start**	4 users join lobby	Game Users are added to database and Game Deck is created. All 4 users in lobby are redirected to game page	**pass**
2	**Draw**	current Player	When clicked, the current player is given 1 extra card. Current players can draw unlimited cards.	**pass**
3	**PlayCard**	current Player	The played card will be placed on the play card deck face up.	**pass**
4.	**Reverse Card**	current Player	The Rotation of the game Should change and shall go to the previous player	**Fail, whenever reverse card is played and it involves player number 1, it messes up.**
5.	**Draw2 Card**	current Player	When the draw 2 is played, the following player is given 2 cards automatically and then skipped	**pass**
6.	**Skip Card**	current Player	When played the following players turn gets skipped	**pass**
7	**Wild Card**	current Player	When played, the current player has an option to change the color of the cards being played with a color switcher prompter	**pass**

8.	**Draw 4 Wild Card**	current Player	When played the current player has an option to switch the color of the cards being played like a wild card, but also will draw 4 cards automatically to the following player.	**pass**
9.	**Reshuffle Deck**	Game User draws card(s) from deck and the number of cards in deck is less than 5	Deck is reshuffled and can be seen after 5 more cards are drawn	**pass**
10.	**Websocket Send Event to Update Game UI**		Triggers Pusher Event to that channel	**pass**
11.	**Individualized Game State Sent to Each Game User**		In Browser Console, Individual gamestate is visible as being sent to Each game user	**Pass**
12.	**Game Winner/End**	Player plays their last card	Trigger Pusher Event with game over set to true in Game State object.	**Pass**

Project Rubric

Category	Description	Points Available	Points Received	Comments
Code Quality	Code is clean, well formatted (appropriate white space and indentation)	5	5	None
	Classes, methods, and variables are meaningfully named (no comments exist to explain functionality - the identifiers serve that purpose)	5	5	None
	Methods are small and serve a single purpose	3	2	With some of our methods containing long promise chains, some of our methods are not small, however, we tried to keep all methods to serve a single purpose.
	Code is well organized into a meaningful file structure	2	2	None
Documentation	A PDF is submitted that contains:	3	3	None
	Full names of team members	3	3	None
	A link to github repository and application deployed on Heroku	3	3	None
	A copy of this rubric with each item checked off that was completed (feel free to provide a suggested total you deserve based on completion)	3	3	None
	Brief description of architecture	1	1	None
	Problems you encountered during	5	5	None

	implementation, and how you solved them			
	A discussion of what was difficult, and why	5	5	None
	A thorough description of your test plan (if you can't prove that it works, you shouldn't get 100%)	5	5	None
Functionality - Authentication	Registration	2	2	None
	Login	2	2	None
	Keep User Logged In using Sessions	2	2	None
Functionality - Chat	Lobby Chat Send To Backend	2	2	None
	Game Chat Send To Backend	2	2	None
	Websocket Event to Update UI	5	5	None
Functionality - Game	Game Start	4	4	None
	Draw Card	2	2	None
	Play Card	5	5	None
	Reverse Card	2	1.5	When the reverse card is played and player number 1 is involved, there is a bug where the reverse card does not work immediately. Otherwise, reverse card logic works perfectly.
	Draw 2 Card	2	2	None
	Skip Card	2	2	None
	Wild Card	2	2	None
	Draw 4 Wild Card	2	2	None

	Reshuffle Deck	2	2	None
	Websocket Event Sent to Update UI	5	5	None
	Individualized Game State sent to each Game User	10	10	None
	Game Can End	4	4	None
Total		100	98.5	