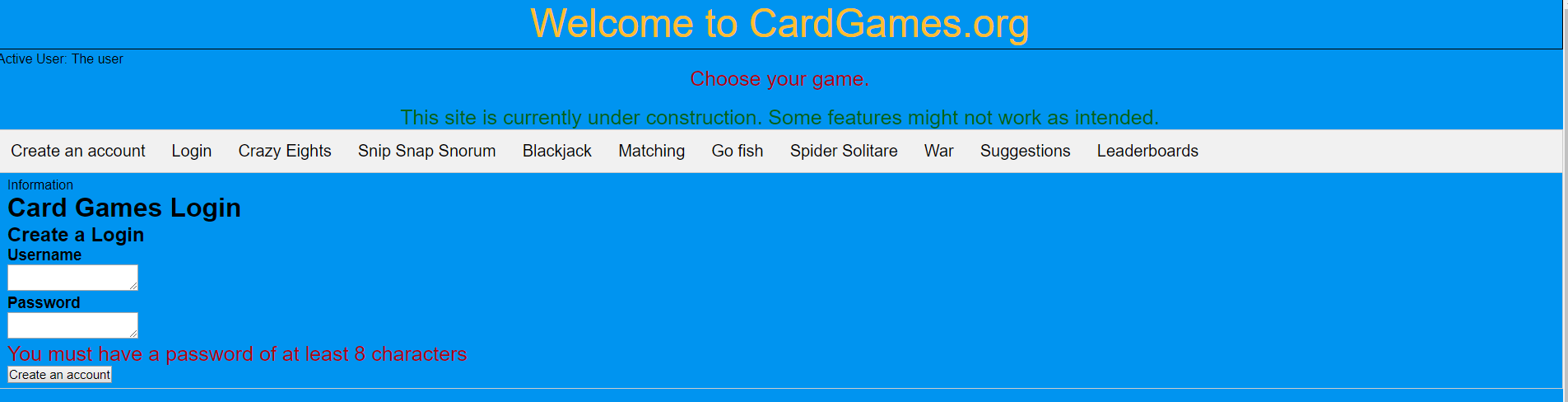
Zachary Kuchar CardGames.org 3-22-19

For my senior project, I plan to extend a class assignment. In my Web Systems class, I received an assignment for a Crazy Eights card game on the web. There were three different alterations of the assignment. One version played the game through alerts in the web page. The second version played the game with actual card graphics against a computer. You clicked on the card that you wanted to play, and you could click on the deck of cards to draw a card. The third version was played in the same way, but two different people connected to a server to play against each other. My project will involve taking this game, along with multiple other card games, and putting them on a fully fleshed out website.

This project will involve several different card games. Games like War, Go Fish, and many others may be included. These games will have updated graphics and some animations built into the gameplay. The website will also feature a login system. People can create a user ID and password, and then they can login to the site to play. When a person wins a particular game, their username, along with their score or best number of moves, will be uploaded to a database. This database will contain the scores of everyone who has achieved a good score on a certain game. You will also be able to view each of the scoreboards to compare yourself to others who are playing.

**Aesthetics and Functionality**

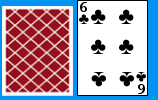


*Figure 1: Project look*

Using HTML and JavaScript, I wanted to create an original website. The website will have a title and tabs of the various card games. There will also be a tab for creating a login, one for each game, and one for a leaderboard. The setup will be easier for users to maneuver from game to game easily.

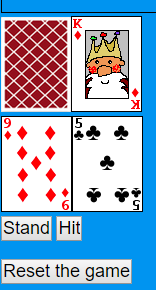
I started off with the graphical style for each game. Starting with Crazy Eights, I created the look for each of the cards and their layout. Then, I incorporated this layout into the other games, making modifications as needed. For games such as Go Fish and Snip Snap Snorum, the initial design did not need to change. For games such as War and Blackjack however, more modifications were needed more modifications.

After creating the layout of the cards, it was time to work on their functionality. Crazy Eights involved clicking on a card image to perform a specific action. On your turn it required you to click on a card in your hand. This would call a function to see if that card could be played. If you needed to draw a card, you must click on the back-face card that represents the deck.

 *Figure 2: Back and front of cards*

The next game was War. The functionality of this game was much simpler. All the player needed to do was press a button, and the game dealt out cards. From there the game would display the player’s cards and the computer’s cards. Then, the game would compare the card values and see which one was higher. The player with the higher valued card scored a point. Normally in War, you would keep the card all the cards that you won for later use again. For the purposes of this project, the functionality has been simplified for easier coding.

The final look of Blackjack involved a similar button intensive system. One button called for a function to deal a card, while the other allowed the computer to take its turn. This game required the need for the computer to keep track of the card values in each player’s hand. When the value was over twenty-one, the player would bust and lose. At the end of the game, the player who had the higher valued hand would win.

 *Figure 3: Blackjack*

Snip Snap Snorum invovled clicking on both an image and a button. The player selects a card in order to play, but the turn does not pass like it does with Crazy Eights. The first card played during a round can be anything, but the second and third cards must match the value of the first card played. Playing a third card allows for you to start the next round. If you do not have a legal card to play, you must pass. The computer will proceed to play until it no longer has a legal card.

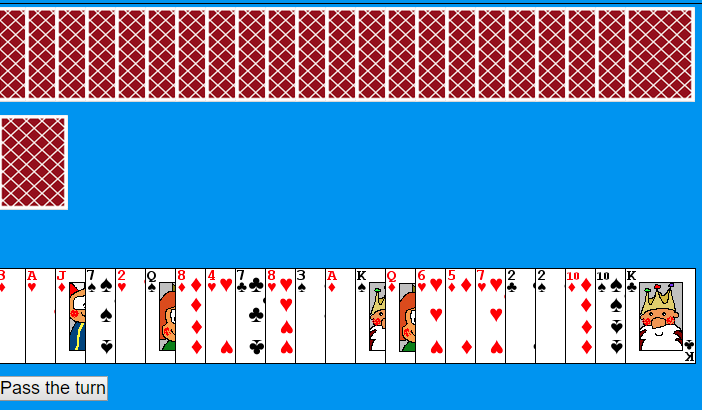


Fig 4. Snip Snap Snorum

Go Fish has two different functions that can occur when you click on a card. You can either click on a card to ask if the computer has it or give all necessary cards to the computer player. A Boolean value is used to determine which action is necessary for the player to take. If you do not have the required card, you can hit a button to tell the computer to Go Fish.

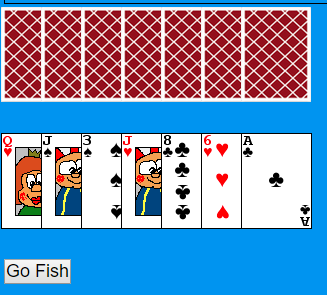


Fig 5 Go Fish

The Match Game is similar to other simple card matching games. You flip over two cards to see if they are the same. If the cards are the same you get a point, if not, you try again until you get a match. At the end, you will get to see how many moves it took.

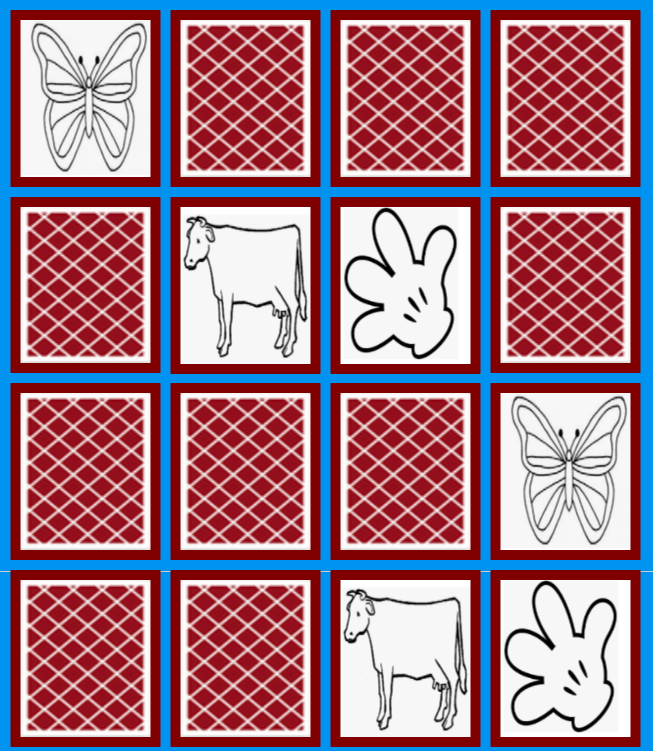


Fig 6 Match game

Spider Solitaire required readjusting the layout and display of the game. To make a move, you click on a specific card, and then the game transfers it along with the row. The next click attempts to move the chosen cards. If legal, the move proceeds. If not valid, the user is alerted of an illegal move. If no moves were available, it requires clicking on a new deck to deal out extras. If a row contains the correct order, they are removed, and the player is awarded a point.

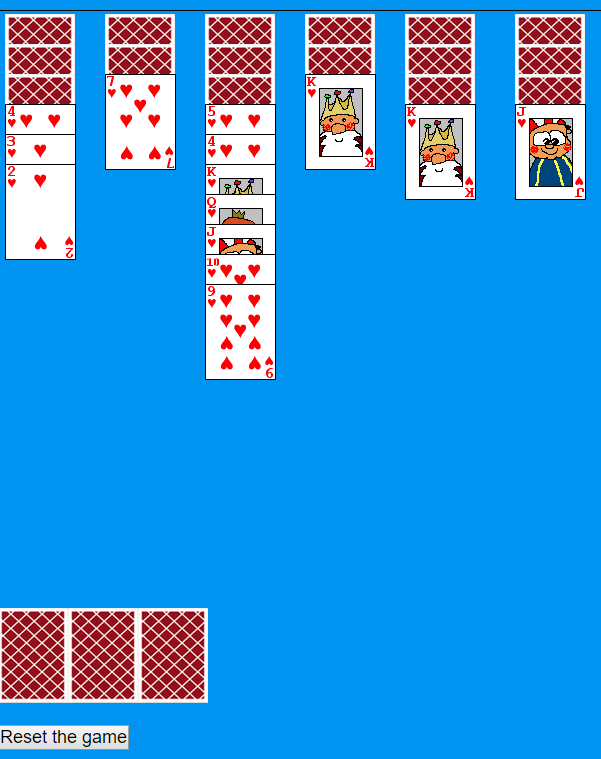


Fig 7 Spider Solitare

**Challenges/Problems**

Working on this project did come with its fair share of difficulties. Calculating the values of the cards in games like Blackjack was time consuming and had its struggles. An error involving the way the values were being concatenated together took some time to figure out. I was attempting to use parenthesis and a plus sign to add the values of the numbers up for a comparison. Instead of the desired result, this would cause the numbers would be combined into one big number instead of being added together separately (7+3 would make 73, not 10). I realized my mistake with the corrected code below. I needed to put a “+” sign in front of the values to be added. This added the numbers instead of making one number.



*Figure 8: code snipit. Using + sign to add numbers instead of concatenation*

A technical problem that arose with Blackjack was when the value of an ace came into question. I needed to determine how to properly incorporate both values of a one and an eleven. It proved to be difficult to effectively switch between the two values. As I tested different methods, I settled on the value of eleven and subtracted ten if the total hand value went over twenty-one. This method worked.

A few snags while working with Go Fish involved clarification whether asking for or receiving cards, as well as making sure the user could not play extra cards. I set up a Boolean variable to determine if the player needed to ask for a card or give a card to the computer. This allowed the game to determine the necessary action for the player to take.

With Spider Solitaire, the obstacles with card positions as well as functionality with card movement created multiple challenges. Making the card’s position absolute allowed the cards to be displayed in vertical rows. The next issue involved the placement of cards properly. A two-click system seemed to correct the problem. The first click selected the cards to be moved, while the second found the section that they were to be moved to. This feature took some time to correctly implement. In the end, I opted for an implementation that noted the cards to be moved, and then I determined if the move was valid. If it was legal, the cards were removed from their old row and placed into the new one.

Other challenges that needed attention were the ability to move a king into an empty row and remove the ordered row of descending cards from play to add a point. To move a king, we needed to see if a row was empty. To do this, a card back face would be used. This image would use an image title of empty, instead of any other string used for a card. The program would check for the card title to say empty, when determining if a king could be moved. By displaying the image of a card back with a value of “empty”, I was able to determine a legal move for a king. Removing a row of cards required examining a row and verifying if all of the necessary cards were in the correct order. If they were, it removed the cards. Otherwise, nothing happened.

Setting up the server

Adding online functionality required setting up the project in Node.js. I created a test server on Heroku, and then I converted the main project from PHP to Node.js. I previously used some simple PHP code so that the main project could run on Heroku. Now that I am switching over to Node.js, I no longer need to use this PHP code anymore. This enabled the setup of a WebSocket server, as well as the rendering of the webpage with the necessary CSS and JavaScript files. While I was familiar with setting up WebSocket servers with Node.js, I was not familiar with running a node server under Heroku. I was disappointed that the process of learning the necessary requirements and getting everything working took as long as it did. I had to use Express to load the webpage, and then do some finagling in order to load any extra files that were necessary. When this was solved, I needed to learn how to set up a WebSocket server so it could interact with Heroku. The only real change needed was the way that the WebSocket on the client end finds the server. Heroku’s forums suggested this code to allow my webpage to interact with my server properly.



*Figure 9: WebSocket client code, replaces http in url with ws for compatibility with websockets*

Once set up, I was able to test if the server was working with some simple communication scenarios. I set up functions to send messages to the server, and have the server send messages back. An error occurred with the correct message being sent back, but this issue was just a simple if-statement problem. Once fixed, the server appeared to send and receive messages correctly.

Online mode

With the main server now up and running, I introduced an online mode for Crazy Eights, Sip Snap Snorum, and Go Fish. I did not give all the games this option. When playing Crazy Eights, the game would notify the server if it was going to play a card or draw a card from the deck. The server would send a message back with the essential information.



*Fig 10 Websocket code, Websocket receives server messages and passes info off to correct game*

My project was set up to send information through the web page itself since the WebSocket connection resided there. When a message is sent to the webpage, it checks a portion of the message to see which game is being played online. From there, it sends the information off to the correct game in order for an update to be made. Lastly, I added a button to switch back to offline mode. When choosing this option, the current user could switch back to a normal version of the game. Selecting the offline mode sends a message to your opponent that you have left the game, and they will win by forfeit.

Next on the agenda was getting the online capabilities for Snip Snap Snorum. The main change in this game from the offline version is removing the functionality for the computer player. This allowed for a much simpler look to the main presenter class. The only other necessary changes were the addition of the update function, to receive updates for the game from the server, and a function to perform the necessities before switching back to offline mode. To make things even simpler, the game only required an update when play passed to the other player. From there, it became the server’s job to determine the correct pass along the correct information to the other player. The only exception to this was when a player ran out of cards in their hand. If this was the case, then the server would perform the necessary actions to say that a player has won the game.

**Reworking Go Fish**

Before working on an online mode for Go Fish, I decided to restructure the offline version that I currently had. I discovered that the original set of rules that I programed the game off of were not correct. I originally had players asking for cards, and then removing card pairs from each player’s hand when they contained one. The player would then win by running out of cards in hand. To have the game play out like a more normal game of Go Fish, I implemented the following set of rules:

1. Players ask their opponent for a card
2. The opponent gives the player all copies of that card.
   1. If they contain no copies of that card, they instead say go fish and the original player draws a card from the deck.
3. Once a hand contains a four of a kind remove those cards from your hand and score a point.
4. Once the deck is empty and there are no cards left in either player’s hand, the player with the most fours of a kind wins.

*Fig 11 Set of Rules for remodified version of Go Fish*

I needed to remove some of the original functionality for new functionality. First of all, I needed to change how the game determined if there were multiple copies of a card in a player’s hand. I did this by actually looping through the player’s hand and counting the cards. If it found a four of a kind, it would proceed to remove all copies of that card. As an extra addition, a copy of the card removed was placed off to the side to show that a point had been scored. This feature to count the score correctly ended up giving me some issues. The program would appear to lock up when attempting to remove cards. On further inspection, it was discovered that the code the I was using was creating an infinite loop. As I was removing the cards, I was not updating the copy of the array that I was looking at. I also worked on a method for not just moving one card at a time, but multiple cards at a time. I did this by simply passing all the cards to be moved into an array and passing along the contents to the other player. After making some other changes and bug fixes, the game was now working properly.

I implemented the features of the modified version of Go Fish into the online version of the game. The only difference was that I now had to upload data to the server in order to pass it along to the other player. This version of the game also presented its own fait share of problems. One problem was once again, removing cards effectively. However, I once again solved it by restructuring my code so that there was not an infinite loop. Another problem presented itself when attempting to look at features of either the deck, or one of the players. For some reason, the lists of cards for the players would not register as card objects. Luckily, I was able to fix this problem by using JSON’s methods of stringify and parse to turn the objects into actual card objects. With these problems fixed, I was able to get Go Fish to work properly with two players.

**Fixing online issues**

One of the main problems that I encountered with the setup that I was working with the online modes was the inability to play with more than two players. It would also crash if two players would get on the server before one would switch over to an online game. Fixing this issue would mean tinkering with my current setup on the server. I started off by setting up different arrays for the different games. One for the WebSockets that would join the game, while others would be for any of the game’s necessities. When a player would switch over to an online version of a game, their socket connection would be added to the array of sockets for that game. Also, once a second player joined, the server would ready the necessities for another game, should another pair of players decide to join.

In each of the game’s other functions, I had to calculate which game to refer to when passing along information, as well as what player to send the correct information off to. I would make this determination by looking at the player number. If it was an even number, I would divide the number by two to determine the number for the game that the current users were playing. This meant that we would take the game number and use the values of the game’s array necessities at the point. The trouble came when implementing which player to send off the data to. Depending on whether the player had an odd position, or an even position in the WebSockets array, I would have to look at the socket and player, directly to its right or it left. This became very tedious to do, because I had to set up multiple cases for how to look at different players and pass, and pass back information to different Web Sockets. I had a case for the current player being even, and a case for the current player being odd. For Crazy Eights and Snip Snap Snorum, all that I really had to do was make a second copy of the code. Go Fish on the other hand, required much more work to be put in because of how I originally programed the game. I not only had to make a second copy of the code to send back to the server, I had to make a second copy of code for calculations done on the other player’s hand. Because of this, Go Fish required some very tedious work to get working correctly with multiple people. Once all this work this work was done, the server side of the project was now able to handle multiple people logging onto the game and then playing against each other.

The next step to handling online play was to handle players quitting and removing them from the array lists for the respective games that they were playing. Starting off, when a player quits a game, not only does the server send information to the other player to tell them that their opponent has left the game, the player that left has their WebSocket removed from the list of sockets playing that game. Now the trick was to determine how to properly remove null sockets, and game essentials for the games that were being played by these connections. My initial idea was to remove the game essentials for a particular game when both socket connections for those games had been set to null. This means that they either disconnected from the game or left online mode for that game. However, this meant that I would have to write code to take into account scenarios such as a player two being added after player one has already left. In the end, I decided to go with a setup that would delete all game necessities for a game in which just one player has left. Afterwards, if no game necessities remain, then a new setup for a game will be created for the next set of players when they join. Also, if a player decides to leave the site entirely, the server will check if that player’s socket connection exists in each game. If it finds it, it sets that connection to null. Once all this was done, the website was cleanly able to handle what happens with online games when players disconnect or leave.

**Handling Databases**

The next task for the server was an online database. After some online research of how to incorporate databases into Node.js, I decided to use MongoDB for my database. I spent some time learning about how to connect to a database, how to add an entry, delete an entry, and so on. I set up a local MongoDB database on my machine to test it. An early problem I discovered, is that the code for connecting to the local database will not be satisfactory when running it on Heroku. I managed to make changes so that it could recognize the MongoDB plugin, but it fails to connect to the local database that I have set up.

After creating the database and the collections, I started off by integrating a login function. You could create a login in order to add your name to the database. However, you could not choose a name that was already in use. For example, if one user set up the name “gamer23”, the system would not allow another to use that name. I also had functionality for logging in. You would enter a username and password, and the server would attempt to find an entry in the database with the given information. If that information was found, then the user’s WebSocket connection was given a username with the name found. If not, then it would inform the user that the login attempt had failed.

For Crazy Eights, the database and the game were set up to record the number of moves that the player took in order to complete the game. The game itself will keep track of the amount of moves that the player made while playing. If the player wins, that information is then sent to the server, and the server will send it into the correct database. The server will first check to see if the currently logged in user has an entry into the database for that game. If not, then an entry is put into the database. However, if an entry with the user logged in is found, the server will have that database entry updated with the new information. Since we want to record the best result of the games only, the result is only updated if it is better than the existing result that existed in the database.

The same methods are taken for the other games to upload data, however, the results recorded for each game are not always the same. For example, in Snip Snap Snorum, I decided to record the amount of time that it took the user to win. I made this decision because it takes the same amount of moves for every player to win the game, this would result in every user put into the database to have the exact same number of moves, which would not be good for you to compare yourself with others.

For the games that did not already contain functionality to communicate with the server, I began to set up use so that their main presenter functions could. This simply involved adding the socket to the presenter as part of its components. Then when the time was right, the WebSocket sent a message with the appropriate information to pass to the server. From there, the server would check if the WebSocket connection had an appropriate login name added. If it did, then the name the socket was using, along with the appropriative information into the database. I proceeded to do this for each of the offline games. However, as these implementations were being made, the information that was uploaded to the database was changed to reflect how the game was played. For example, in war, the number of times that the human player had won and lost is recorded, versus in Crazy Eights where the number of moves taken to win was recorded instead. These decisions were made based on what made sense for each game. In war, it did not make any sense to count the number of moves to win, since it only required one move to deal a card, and possibly more if a war is declared. As each of these databases were created, the webpage itself was modified to pull information from the database. Using a list of options with each of the database names, the webpage can poll the server to display information from the database. The first 20 entries of a particular database will be displayed, along with the entry from the user, if they were logged in. That way, the user could display how well that he is playing versus the rest of the field. Each leaderboard will continually challenge players to continue to climb to the tops of each rank. Each database had a specific way that the results would be ordered in. For example, for the Crazy Eights database is ranked based on number of moves it takes to win. The resulting printout would display users who have won with fewer moves before those who have won with more moves.

**Fixing some issues**

With the functionality of the database operational, I went to fix some issues that still existed. One such issue involved Heroku. When using Heroku, if a user would remain inactive for too long, node would go to sleep, resulting in the WebSocket connection being dropped. I began researching some techniques that I could use in order to resolve this issue. I eventually opted to go with a simple setup, where the window would ping the server every so often, so that the connection would not time out.