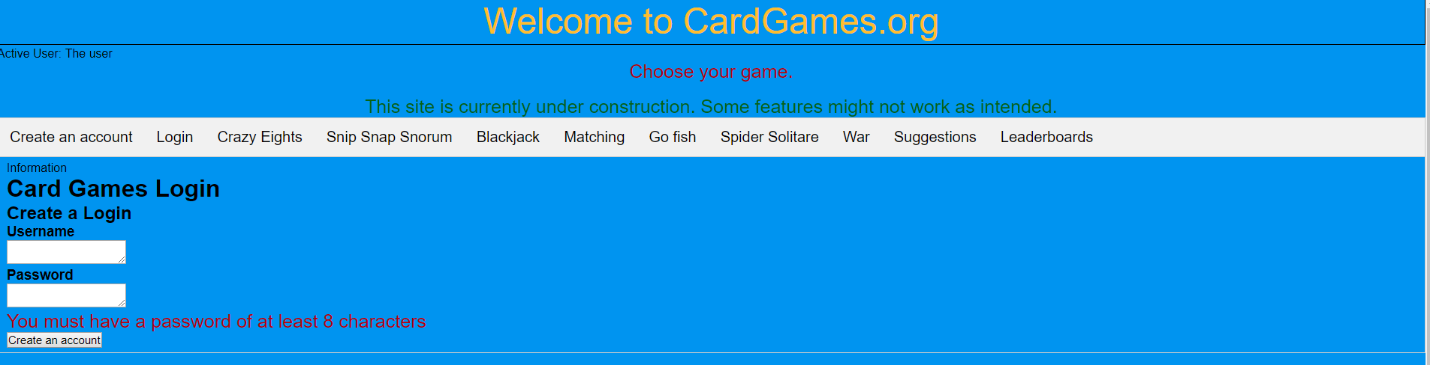
Zachary Kuchar CardGames.org 3-22-19

For my senior project, I extended a previous 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 their own website.

This project involved several different card games. Games like War, Go Fish, and a few others were included. These games have updated graphics and animations built into the gameplay. The website featured a login system too. Users can create a user ID and password, allowing them a login to the site. When a player wins a 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**

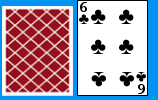


*Figure 1: Project look*

Using HTML and JavaScript, I created an original website. The website has a title and game tabs. A tab was created for a login, each game, and for the leaderboard. This setup allows users to maneuver from game to game easily.

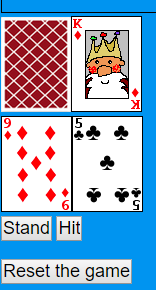
I started off with the graphical style for each game. Starting with Crazy Eights, I created the look for each of the cards and their layout. Then, I incorporated this layout into the other games, making modifications as needed. When making Go Fish and Snip Snap Snorum, the initial design did not need to change. For 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. To request a card, a player clicks on a specific card in the deck. This click checks to see if that card can be played. If you need to draw a card, you must click on the back-face card that represents the deck.

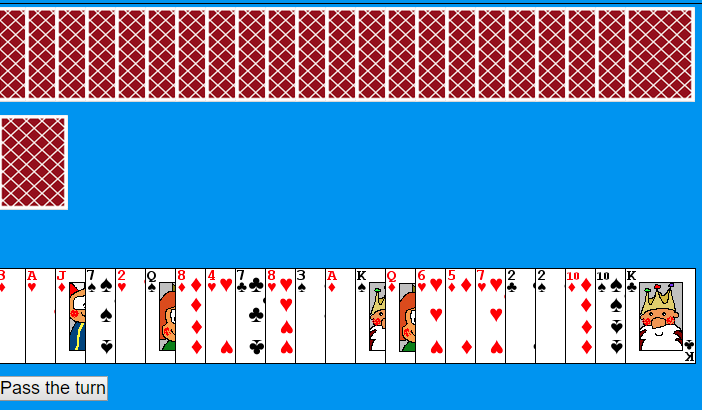
 *Figure 2: Back and front of cards*

The next game was War. The functionality of this game was much simpler. A player just needs to press a button to deal cards. The game would display the player’s cards and the computer’s cards, compare values and who was higher. The player with the higher valued scored a point. After the 52 cards are dealt, the winner is determined.

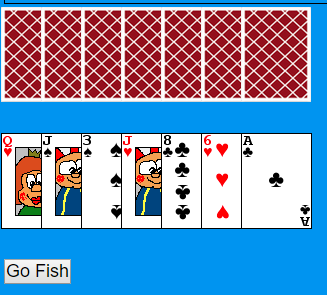
The final look of Blackjack involved a similar button intensive system. One button called for a function to deal a card, while the other allowed the computer to take its turn. This game required the need for the computer to keep track of the card values in each player’s hand. When the value was over 21, the player would bust and lose. At the end of the game, the player with the higher value won.

 *Figure 3: Blackjack*

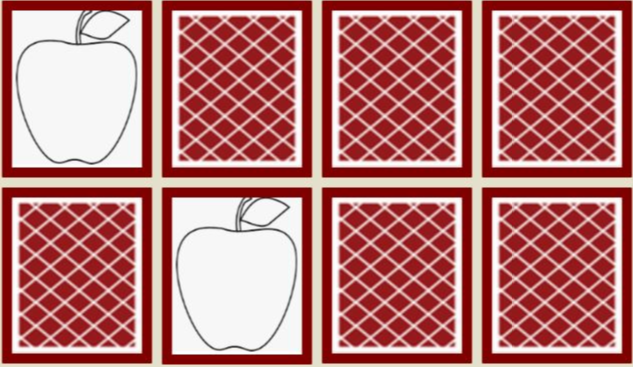
Snip Snap Snorum involved clicking on both an image and a button. The player clicks on a card, but the turn does not pass like it does with Crazy Eights. The first card played during a round can be anything, but the second and third cards must match the value of the first card. Playing a third card allows 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.

 *Figure 4. Snip Snap Snorum*

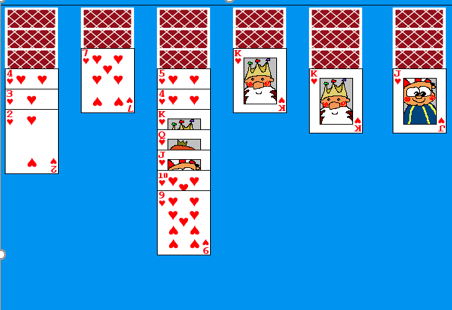
Go Fish has two different functions when clicking on a card. You can either click on a card to ask if the computer has it, or give all necessary cards to the computer player. A Boolean value is used to determine which action is necessary for the player to take. If you do not have the required card, you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

 *Figure 5 Go Fish*

The Match Game is a 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.

 *Figure 6 Match game*

Spider Solitaire required readjusting the layout and display of the game. To start, you click on a specific card you want to move, next you click on the row for placement. The game transfers it or the row. If legal, the move proceeds. If invalid, the user is alerted of an illegal move. If no moves are available, it requires clicking on a pile to have more dealt. When a row contains the correct order, they are removed, and the player is awarded a point.

 *Figure 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 arose with Blackjack when the values of an ace came into question. I needed 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 with Go Fish occurred when asking for a card, or receiving a card, along with 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 and 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 card/cards to be moved, while the second click found the new position of the card. This feature took some time to correctly execute. 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 and placed into the new one.

Other challenges involved moving a king into an empty row and removing the ordered row of descending cards from play. To move a king, the program needed to check if the row was empty. To do this, an empty card value was used with the value of empty. The program checked for an empty title, when determined, a king could be moved. When an empty-card image with a value of “empty” was displayed, the game could determine if the move was legal. Removing a row of cards required examining a row and verifying all of the cards were in the correct order. If they were, it removed the cards. Otherwise, nothing happened.

**Setting up the server**

Adding an online function required a Node.js project to be set up. My original game was created with PHP on Heroku. I converted the main project from PHP to Node.js. 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. It was disappointing that the process of learning the necessary requirements and getting everything working took as long as it did. I used 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 learned how to set up a WebSocket server to 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 the server 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.



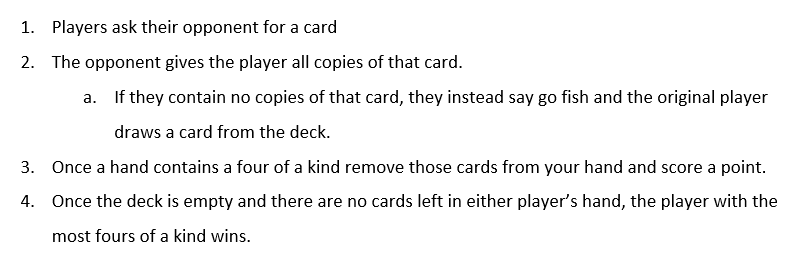
*Figure 10: Websocket code, Websocket receives server messages and passes info off to correct game*

My project was set up to send information through the web page itself since the WebSocket connection resided there. When a message was sent to the webpage, it verifies which game is being played online. From there, it sends the information to the correct game for updates to be made. Lastly, I added a button to toggle between on/offline modes. Selecting the offline mode notifies your opponent that you left the game, and they have won 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 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:



*Figure 11: Set of Rules for remodified version of Go Fish*

I needed to remove some of the original functionality for new functionality. I changed how the game determined if multiple copies of a specific card were in a player’s hand. I did this by looping through the player’s hand and counting the cards. If four-of-a kind was found, it removed those cards and a placed them in a pile to the side. Each pile was given a point. Although, the game would lock up when attempting to remove the cards from the hand and correctly scoring the piles. I discovered the error for this code was in an infinite loop. When the cards were being removed, it was not updating the copy of the array in the program. 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 modified version of Go Fish into the online game. The main change to this required that I 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 fair share of problems. Once again, I had to remove the cards effectively by restructuring the code preventing an infinite loop. When attempting to look at features of either the deck, or a player’s cards they were not being registered as card objects. Happily, this was fixed by using JSON’s methods of stringing and parse to turn the objects into actual card objects. Finally, Go Fish was working properly.

**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.

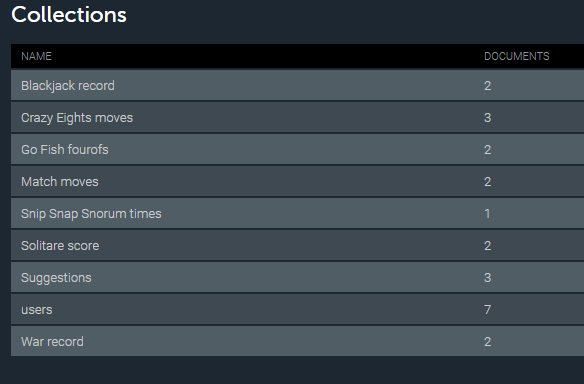
*Figure 12: Crazy Eights with four different players and two different games going on at the same time*

In each of the game’s functions, I had to have the server calculate which game the user was playing. I would make this determination by looking at the player’s ID. If it was an even number, I divided it by two to determine the number for the game that the current user was 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 the data. 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 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 more work, because of the way 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 with the other player’s hand. Because of this, Go Fish required some very tedious work with multiple people. Once all this 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 was to handle players quitting and removing them from the array lists for the respective games. When a player leaves a game, not only does the server send information to the other player saying their opponent has left, the other user 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 games that were now no longer in use. My initial idea was to remove the game essentials for a game when both socket connections for those games had been set to null. This means that they disconnected from the game. However, I would have to write code to consider scenarios such as a player two being added after a player one already left. In the end, I decided on 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 when the next set of players joins. 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 found, 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. I spent some time learning about how to connect to a database, add/delete an entry, and update entries. I set up a local MongoDB database on my machine to test it. I discovered the code for connecting to the local database would not be satisfactory when running on Heroku. I made changes to the server to recognize the MongoDB plugin, but it failed to connect to my local database.



*Figure 13: The collections in my Mongodb database.*

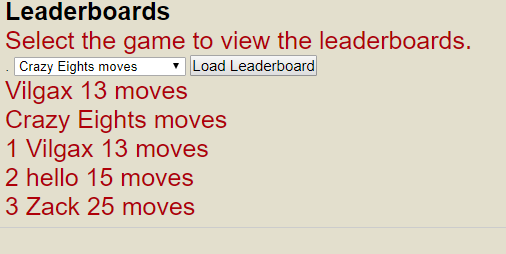
After creating the database on mLab, I integrated a login function, which allowed users to create a login for the database. If an account name already existed, it would not let you use that name. For example, if one user set up the name “gamer23”, the system would not allow another use of that name. 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 the player acquired to complete the game. The game records the number of moves the player made while playing. If the player wins, information is sent to the server, and the server sends it to the database. The server checks to see if the user has an entry on the leaderboard for that game. If not, an entry is made. However, if an entry with the user is found, the server will have that entry updated with the new information, only recording the best results.

The same method was taken for the other games, however, the results recorded are not always the same. For example, in Snip Snap Snorum, I decided to record how long it took the user to win since it takes the same amount of moves every time. This would result in every user put into the database to have the exact same number of moves, which would not be good for comparing yourself with others.

For the games that did not have online functionality, I set up their program to communicate to the server. This involved adding the socket to the presenter as part of its components. When the time was right, the WebSocket sent a message with the appropriate information to the server. Now, the server would check if the WebSocket connection had an appropriate login name added. If it did, the socket’s login name, along with the appropriative information was added to the database. I did this for each of the offline games. However, as these implementations were being made, the information being 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. In war, this did not seem logical, since it only required one or two moves. The webpage was modified so the leaderboard tab could load the database.

Using a list of options with each of the database names, the webpage can poll the updated leaderboard on the server to display information. The top 20 entries would be displayed. If the user was logged in, the player could compare their standing with the other players. Each database had a specific way that the results would be recorded. For example, the Crazy Eights database was ranked based on number of moves to win. This displayed the players who won with the fewest moves.



*Figure 14: Displaying leaderboard for Crazy Eights. Goes by number of moves taken to win*

**Fixing some issues**

With the functionality of the database operational, it was time to fix another issue. When using Heroku, if a user would remain inactive for too long, the node would go to sleep, causing the WebSocket connection to be dropped. After researching some techniques, I found a simple fix by having the window ping the server every so often. Now that the server was getting a message, the game remained active.

Moving on to the Matching game, I needed to add a reset button to the game and update the scores to the leaderboard. Since, the game had no constructor to easily pass the game code into, I decided to move the JavaScript code for the game into the main HTML file. This made it much easier to insert code that would allow the game to communicate with the database without having to have any fancy setups. This involved setting up all the cards so that they were flipped face down once again, and then once again randomizing their locations. If the reset happened to fast, the player were able to see the new positions of the cards before they flipped. I set a delay in between flipping the cards back over, and the card positions being randomized again by doing a time out.

**Animations**

Now to make the cards move. These methods would differ according to each game. The trick was to change the left and top attributes of each image appropriately. The delicate part was accurately delaying certain elements to make the card looked like it was moving. First, I needed to set a method to delay the movement. Without it, the cards would snap into place, preventing the effect I desired. Here is the method I choose.



*Figure 15: Code to have cards move across the screen*

For Crazy Eights, there were two items to do. The first issue was fixing the addition of a card to a players hand after they drew one. The second was whenever a card was placed into the pile. I started with adding cards to the player’s hand. I decided to add two new functions that placed a card offscreen. From there, the above function was called in order to move the card into the correct position. I wanted the cards to appear as if they were moving across the row. It took some time, but I managed to get it working.

Next hurtle, to get the Timeout function working, since it did not stop other process from starting. Eventually, I modified the display pile function so that the card would be first put down off screen. Then, with the help of the above function, I moved the card into its proper place. To allow the card to properly move into place before anything else happened, I delayed the computer from taking its turn right away. This allows for a clear view of what card you put down, and what card the computer put down.

It was tricky to get the same functions to work with the online version of Crazy Eights, since the functions for displaying the cards were different. A different setup of functions was being used to render and display each card. On top of that, it is not as easy to determine if a card needs to be drawn, since one function in the presenter is what handles all the updates. Because of this, I had to make modifications to in the server to send information to determine if we needed to draw a card. Then, in the update function, I had to make some changes not just to determine if a card needed to be drawn from the deck, but which player needed to draw the card. Displaying the card in the pile, however, was just as easy as in the offline mode. No real change was necessary in order to get the desired effect of the card moving.

With Snip Snap Snorum, the main card movement was putting a card onto the pile. For the human player, this was rather easy. All I had to do was recreate how I moved it in the Crazy Eights. For the computer player, this was not so simple. I had to figure out the best way in order to have each card that the computer put down each one of its cards. The function that handled the computer’s turn underwent multiple tweaks, starting with the removal of its while loop. It took some time to figure out how to properly and integrate a delay into. After some tinkering, I opted to have the function be recursively be called. So long as it was the computer’s turn, the function would be recursively called with a delay. If the computer could no longer play a card, the player would then be alerted that it was now his or her turn. This process allowed for the player to see each card that the computer put down, along with the entire process of the computer’s turn.

For Blackjack, all I really had to do was move the cards into their proper places. This mean just replacing the display functions with ones to place the cards on the screen, and then move them into their proper place. From there, I just had to call the functions at the correct time. If a card was ever added to either hand, the game would then proceed to call the correct function to add the card to the appropriate player’s hand.

The next game on the list was Matching. There was not really any work required to be done here. Through the in-game code, the cards could already be seen flipping over when they were clicked on. With this movement already present, there remained no other need for addition animations.

War’s animations were very similar to those in Blackjack. All I had to do was setup the cards to move in from offscreen. This was also the easiest to setup. All I had to do was replace the calls to the display functions. Once all that was done, the cards would all just move onscreen whenever the play a round button was pressed. The new functionality allowed me to fix a bug that was present when clicking on the reset button. Originally, when resetting the game, it would cause the cards to display incorrectly until the player clicked the play a round button again. With the new animations, I decided to start the game off with no cards on screen. This made hitting the reset button simply remove all of the on-screen cards. This way, any issues with the display would be prevented.

For Spider Solitaire, the main animation that I decided to go with is to have the cards fall into place when being moved. This required taking the set of cards to be moved, removing them from their original row, and then having them drop down into their new row. Originally, I wanted to have the cards rise up out of the rows that they were being removed from. However, this feature ended up providing more trouble than was necessary. I had to find a good way in order to grab a set of cards that were already in existence on the screen. The way that Spider Solitare was already set up and with how I was already moving the cards around, there was no easy way to do this. In the end, I opted for simply removing the card from the first row, and letting it slide into place in the second row. That way, I did not spend time making a part of the project harder than it needed to be when there was not a significant portion of time left.

The last game to work on was Go Fish. There were two animations to be implemented in this game. One was having a set of cards be added to a player’s hand. This happened anytime that a card was drawn from the deck. This could also happen when a player is given cards from the other player. This meant there were multiple place for the animations to be called. I had to make sure the proper set of cards were being moved at the proper time. It took some tinkering for it to get working correctly, but after some time, I managed to get it done. The harder part was setting up the online version of the game to also work with the animations. Go Fish ended up being the hardest online game to have all the animations working correctly. The code on the server for Go Fish ended up being the most complex. Because of this, I had to really think about the best way to restructure the code so that I could pass the correct information to each player. That way, the game could be played smoothly, and the animations would also work as well. I set up the code on the server side, so that each function in Go Fish determined which set of players to send the correct info to once. The program will add cards to the correct players hand and send data back. On the client side, the web page will then process the correct animations that need to play out. While this was happening, neither player would be able to play, so that no problems would occur. Then, after a short delay to allow the animation to play out, the server sends out data to each player to allow play to continue. With some time to tinker and fix any bugs with information sent back, the last of the animations for each game was up and running.

**Future Additions**

I am amazed with how much I learned with this project, and I am happy with the final result. That said, there are some features that I would have liked to implement. One of the features is a chat room, where players could interact with each other. There would be a different chat room for each game. This would add one additional feature to the website for users to interact with each other.

One thing that I would have done differently would be the ability to drag and drop the cards in Spider Solitaire. The current method of moving cards in the game, while sufficient to play, is not the best system to use. The ability to drag and drop the cards would allow for easier play. A player could drag a certain card along with all the cards underneath it, and then drag them to the row that they want to move the cards to. The cards they moved would then snap into place if the move was legal. Otherwise, the cards would move back to their original positions. This would give users a more friendly control scheme while playing.

**Conclusion**

Overall, this project has been a huge success. I learned a lot HTML, Javascript, and Node.js, as well as how to incorporate databases and animations. I was able find the best way to incorporate all these elements together, to form a fully functional website. There were moments where work took longer than initially suspected, which disappointed me. If I could have better allotted my time and prepared for these sections that slowed me down, I could have had more time to better perfect the project overall. In the end, the project became one of my most greatest programing accomplishments, and something for everyone to logon and enjoy.