

Programming Project

hatchibombotar.com

Candidate Number: xxxx

Center Number: xxxxx

Table of Contents

Introduction.....	2
Project Analysis.....	3
My Audience.....	3
Game Analysis.....	3
Game Research (cardzmania.com).....	4
Game Research (skribbl.io).....	5
Game Research (Among Us).....	6
Conclusions from Game Research.....	9
Feature List.....	10
Technical Analysis.....	12
Web communication technologies.....	12
Software Requirements.....	13
Hardware Requirements.....	13
Computational Methods.....	13
Limitations.....	14
Success Criteria.....	14
Plan for Development and Testing.....	16
Development Plan.....	16
Testing Plan.....	16
Design Overview.....	19
Overall structure.....	19
Communication.....	20
Server Design.....	21
Client Design.....	26
Iterative Design & Development.....	31
Stage 1.....	31
Stage 2.....	45
Stage 3 / 4.....	55
Stage 5.....	69
Play Test 1.....	83
Play Test 2.....	87
Evaluation.....	89
Reviews.....	89
Video Showcase.....	90
Implementation of features.....	90
Integration Testing.....	92
Meeting Success Criteria.....	94
Evaluation Summary.....	98
Conclusion to the project.....	100
References.....	101

Introduction

Cheat is a card game where players try and get rid of all of their cards while trying to deduce if your opponents are lying or not. It is taught by oral tradition, meaning it comes under many names and the origin is largely unknown.

The entire deck of cards is split between the players and on each turn players place down a number of cards and state which cards they played or what they want their opponents to think they placed. If opponents believe the person was lying they can say “Cheat” and if they were lying they have to pick up the entire discard pile. Once a player has run out of cards, they are the winner.

In this project, I am going to apply a computational approach to the game of cheat to create an online alternative to the card game that allows friends to play together from anywhere in the world.

Project Analysis

In this section, I aim to outline the essential features of a computational approach to my game. By the end of it, I will have researched existing solutions to the problem and along with my stakeholders will decide what makes these solutions successful, condensing my research into a list of features.

My Audience

My target audience are students, where they will play against their friends in their free time for fun and where they are unable to meet in person.

To gain feedback for my project, I have asked people from my target audience to help who have played "Cheat" before. They are going to help by testing and evaluating my game during and after the development of it. Throughout my project I will refer to them by Person 1, Person 2, and Person 3.

Game Analysis

An analysis of how the "Cheat" card game is played in real life so that I have a secure understanding of what is required in the project.

Setting Up:

- The deck of cards is shuffled
- The cards are split between the players with cards being dealt one at a time in a clockwise direction starting from the person left of the dealer
- The person left of the dealer starts their turn by putting 1-4 cards down of the same rank

On a person's turn:

- The person places 1-4 cards from their hand onto the discard pile. The card should be one rank up from the previous card. If the player does not have the card needed to be placed, they can lie and place any 1-4 cards from their hand down.
- If there are no cards in the discard pile:
 - The person places 1-4 cards down, creating the discard pile for the next turn
- The person playing then calls out what cards they placed. They can either say the truth, or lie in order to put more cards down than they have available to play.
- After their turn, any other player can say "cheat". The top of the discard pile is shown to everyone and if the player cheated they pick up the whole discard pile. If they did not cheat, then the one who said "cheat" is the one that has to pick up the cards. The play continues from the person following the one that picked up the cards

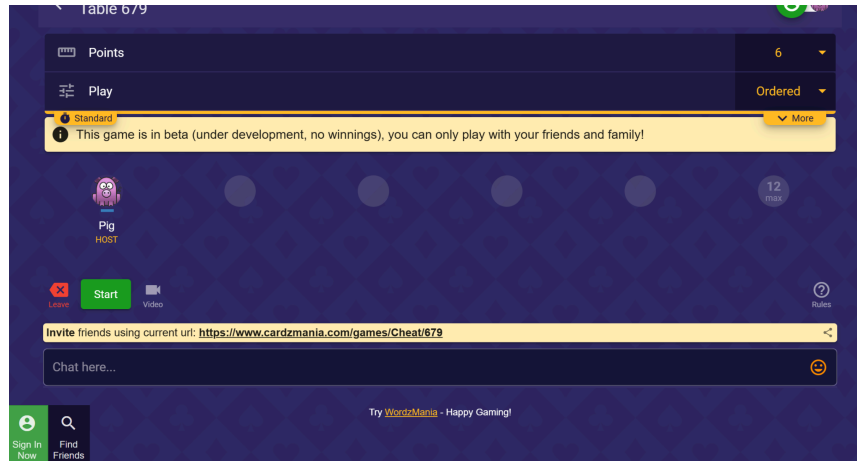
Ending the game:

- Once a player has placed all of their cards down, and they don't need to pick their cards up after cheating, they have won the game and it ends.

Game Research (cardzmania.com)

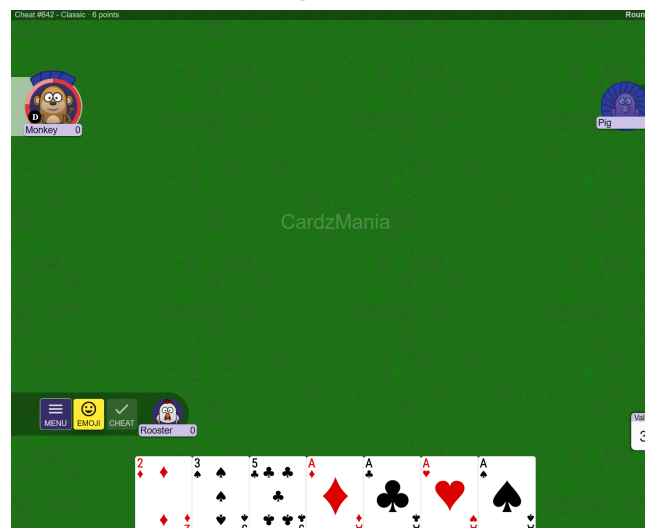
cardzmania.com is a website that hosts many different card games, one of them being Cheat.

Waiting Menu



- At the top of the screen, the host can choose how many rounds will be played before the game ends
- There is a list of all the players in the game as well as player icons
- Friends are invited to the game using a link that has to be shared
- There is a chat feature at the bottom of the screen for players to talk
- The host can press the start button when enough players have joined. If insufficient players have joined then an error will show.
- Players can leave at any time

Play Screen

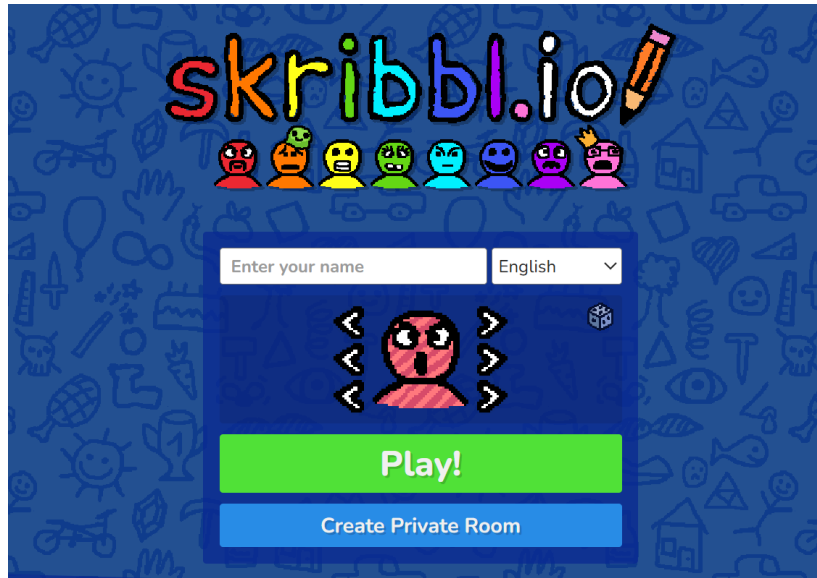


- Players are displayed around the screen with their name, icon, cards, and points won.
- A value counter that shows the value of the card the current player should be playing on their turn
- Cheat button to accuse the previous player of cheating
- A list of cards in a player's hand
- An emoji button to show emoji reactions to other players
- A menu button that allows players to leave the game
- A green background simulating the look of a card table.

Game Research (skribbl.io)

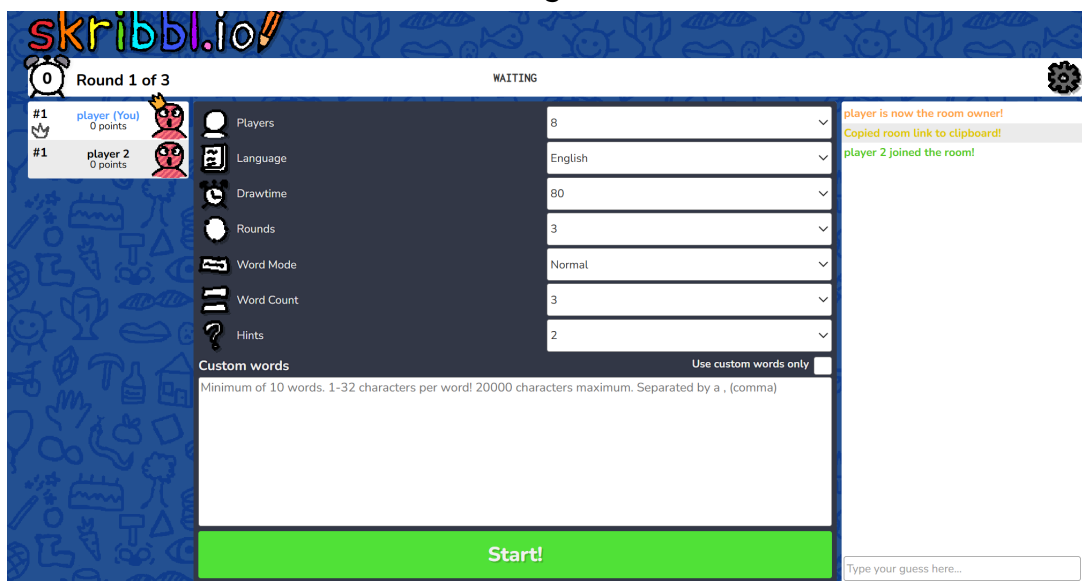
skribbl.io is a multiplayer online drawing game where players can play with their friends. In this game, players create a game and are able to share a room link with their friends in order to play together. Although the concept of the game is completely different to my project, the user interface and multiplayer system is something I will take inspiration from.

Main Menu



- The main menu for skribbl.io directly has buttons to play or create a private room
- Players enter their name into an input box before joining the game
- It also contains customisation features in the way of its character creator
- Users can select their language in order to get put into games with specific drawing prompts

Waiting Menu

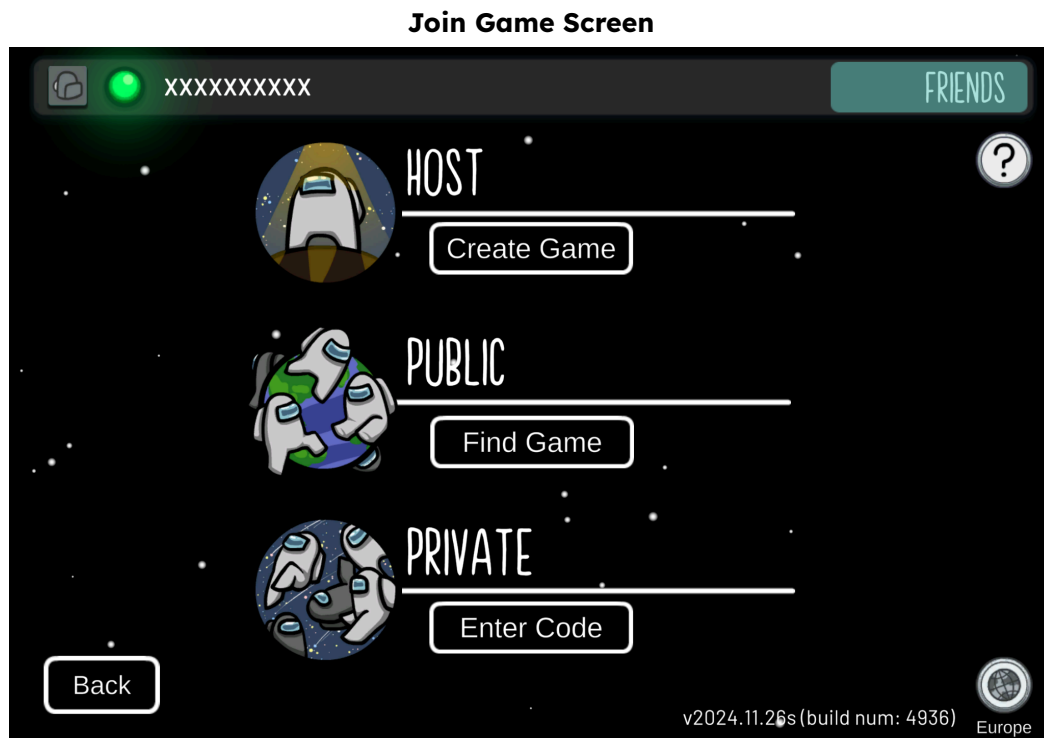


- The waiting menu is displayed before everyone joins the game and the host presses start.
- A list of all the players in the game is shown on the left of the screen
- The game options are shown in the middle of the screen to all players, and are editable by the host.

- The host player can press "Start!" as soon as there is more than one player in the game
- The cog icon on the top right allows players to edit player-specific options.
- There is a chat box on the right side of the screen, which is used for guessing later when the game begins.

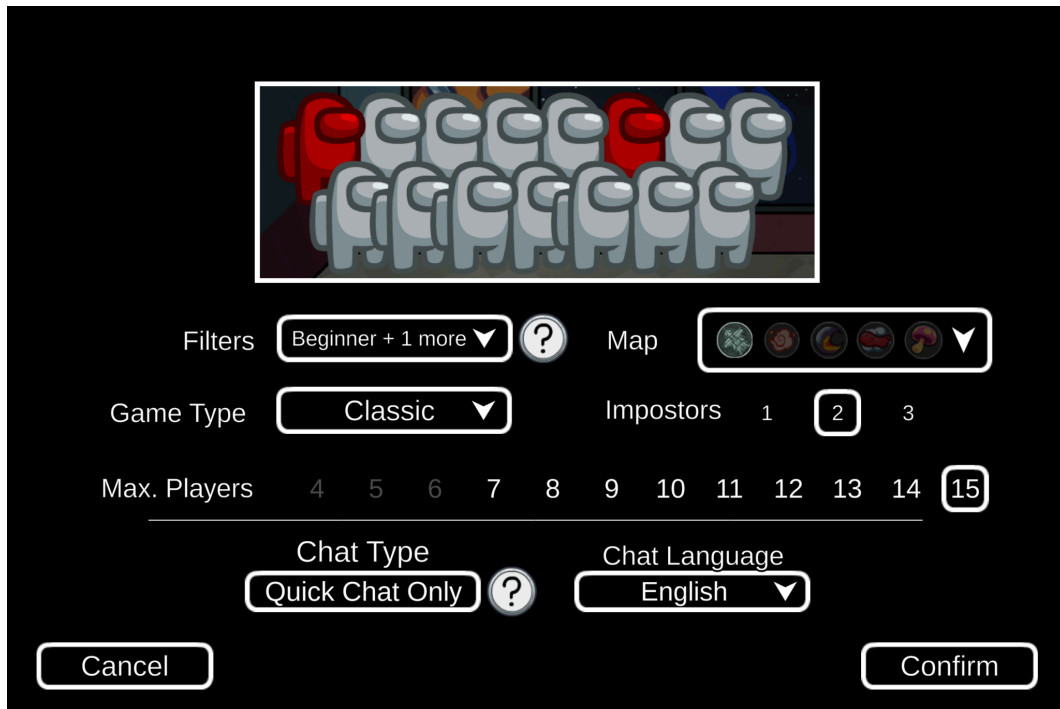
Game Research (Among Us)

Among Us is an online social deduction game where players work together to deduce which one of them is trying to wipe out the crew. The reason I am researching this is to look at a variety of multiplayer systems in order to take the best components of each.



- The menu for joining games in Among us contains three main flows that users can take: creating a game, joining a public game, or joining a private game.
- The "Create Game" button brings you to the options screen before putting you in a fresh game in the waiting screen.
- The "Find Game" button displays a panel which lists out public games that players have created
- The "Enter Code" box allows you to enter a game code of a game created by a friend. When pressing enter you are brought to the waiting screen.
- It also includes a friends feature where players can join games with people they have friended.

Options Screen



- When the "Create Game" button is pressed, an options screen comes up to change rules, such as the number of players in the game and the type of game being played.

Waiting Screen



- Unlike the other games I have researched, the waiting menu is a physical world that players can move around in before the game begins.
- Information about the game, such as the room code, players in the game, and game settings are displayed in a panel on the top right corner of the screen.
- Players can open the in-game chat to communicate with other players
- Per-player settings can be edited when the cog button is pressed, such as controls.

Conclusions from Game Research

After my research on all three games, I asked my target audience for their thoughts on all three games, aiming to get insight into what made the games good and bad, specifically when looking into multiplayer features for the second two games.

For the research on the cardzmania cheat game, the following points were mentioned by my stakeholders:

On the waiting screen:

- The menu is very cluttered and hard to understand
- The chat box is fun
- Sending a url is quite a lot of effort
- There is no point in the points feature
- It is good that you can start from a random person.
- The video feature is interesting as it could allow you to see the facial expressions of other people.

On the play screen:

- Ordering cards based on card numbers is good
- Again the UI is confusing
- The cheat button is too small for a crucial part of the game
- The value counter is confusing for players.

From this, I found that although the cardzmania game was functional, the user interface was exceedingly cluttered and confusing, and keeping the UI clear should be a key focus for my game. My stakeholders liked the video and chat features however given the time limitations I will be focusing on the core feature set of the game and so these features will be omitted.

In terms of multiplayer features, two out of my three main stakeholders said they preferred the system used by Among Us (where players had to enter a room code to join the game) and one preferred the system used by the other two games researched (where a url is shared to join the game). All players said they liked the chat systems built into the game however I have chosen not to include this as it does not match the level of abstraction I am looking for in my project and it is not necessary for the intended use of the game, in which people communicate while the game is being played.

Feature List

The list of essential features that will be required in the solution to the project.

Requirement	Justification
A start screen with options to join or create a game	The player needs to be able to choose when they play and who they play with.
The ability for players to change their display name on the start screen	This is how it works in two of the games I have researched, in Among Us the display name must be changed in settings.
Game rooms are assigned a code by which players can join rooms by entering the code on the start screen.	This was the most popular system for joining games from my research (2/3 of my stakeholders preferred this system).
A "waiting" ui screen	All three of the games I have researched include some form of waiting ui screen.
A list of players on the waiting screen	So that players can see if and when their friends join the game
A start button	So that the host player can begin the game
An exit button visible on both the waiting and in-game screens	If players need to leave during the game they can press the exit button instead of refreshing the page
Before the game starts, the cards are placed in a random order and split evenly between players, one by one in the order that players join.	the game should be fair and cards should be evenly distributed.
After someone presses "Cheat", the game begins again from the person after the one who was challenged.	So that another round of gameplay can go ahead.
During the game, players are able to see how many cards their opponents have, as well as whose turn it is.	When playing the real game, you need to be able to see how many cards other players have to influence if you want to say cheat or not.
At the end of a player's turn, they select what cards they want to play and the correct value for these cards are shown to everyone.	When playing in real life, players say what cards they are playing out loud. As players may not be able to hear each other, this functionality needs to be built into the game. In addition, the game needs to know what cards they say they placed to check if a player was cheating.
On a turn, players can move their cards between their hand and the discard pile	It should be easy to choose what cards they play in a way similar to what it is like when playing the game in real life.

At any point in the game, players can press the “Cheat” button, and if the previous player was lying, they pick up the cards. Otherwise, the player who pressed the button picks up the cards.	the players need to be able to check if one of them is cheating.
When a game ends, the winner is displayed and players have the option to start another round.	It would be very annoying if the players would have to create a new game room from scratch every time they finish a round.
If a player leaves the game mid way through, their cards are placed on the discard pile.	It would ruin the game if some of the cards were out of play for the remainder of the game or if their turn was not taken and the game was stuck forever.

Technical Analysis

In this section, I will be researching the technical requirements of my project in order to begin to get an idea of the direction of my project, including what software/libraries I will be using.

To begin with, I need to decide on what architecture I will be using. If the architecture is client-server I will need to make a decision on a backend and frontend language, whereas if I go for a peer-to-peer architecture then I will only require a single language. I looked at some examples of web communication technologies in order to make my decision:

Web communication technologies

Technology	Description	
Websockets API	A websocket connection is opened which is kept open for the lifetime of the application being run. Messages can be sent across this channel between the clients and servers	Advantages: <ul style="list-style-type: none">- The server controls all of the logic so cheaters can not take advantage Disadvantages: <ul style="list-style-type: none">- A server always has to be running
WebRTC	A connection is opened between clients and data is sent between them.	Advantages: <ul style="list-style-type: none">- is that the server does not run the game logic, instead the logic is run off of the clients. This would mean less stress on the server Disadvantage: <ul style="list-style-type: none">- increased complexity over websockets as the API is less abstracted and
Polling	Clients make continuous requests to the server, if the server has a message for the client, it will respond with that message.	Advantages <ul style="list-style-type: none">- Based off of basic http requests Disadvantages <ul style="list-style-type: none">- Outdated- Continuous requests -> more stress on the server- More complicated to link
Long Polling	Long Polling is an improved technique over polling where instead of sending continuous requests, it instead sends a single request then sends another when a response is received.	Advantages <ul style="list-style-type: none">- Fewer requests over basic polling

From these, I decided to go with the websockets api because of its reduced scope of features combined with it being a fairly modern standard. The drawback of this will be that I need a server running to handle the games as they are played.

Software Requirements

Now that I have decided to go for a client-server model using the websocket api I am able to decide on what languages and libraries to use.

On the client side, I will be using raw javascript. I have chosen this as it will reduce the complexity of my project due to not needing a transpilation step which would be there if I were using a language like typescript. This should help speed up development of the project.

For the server, I have decided to use nodejs as it will allow me to use javascript on both the client and server which I hope will reduce the duplicate or redundant code as well as making sharing data easier.

I have identified two libraries that I will be using in my project:

- "express" - a minimalist web framework - this is a popular web framework and will allow me to run the server and it seems to integrate well with
- "ws" - a nodejs websocket implementation - this will allow clients to connect their websockets to the server to join and exchange information about ongoing games.

I will be using the windows operating system to develop my project however as far as I can tell it would not be locked into windows. NodeJS is a cross-platform runtime so a server could potentially be deployed on linux. Clients will connect to the server using their web browser which is not locked to a specific operating system.

Hardware Requirements

For the server, nodejs does not have any minimum requirements in the way of memory or cpu performance. The amount of memory required will depend highly on the number of players in the game concurrently. Javascript is a single threaded language so only one core is essential in terms of the, although it may benefit from additional cores as the runtime itself may have other processes that could work on multiple threads. Because of the high dependency of the program itself for memory and processor requirements I will not be able to find an accurate estimate of the minimum hardware requirements.

This said, I will be testing the server using:

- 16GB of RAM
- AMD Ryzen 7 7840U
 - Clock Speed 3.30 GHz
 - 8 Cores

My game is going to be primarily designed for keyboard and mouse users.

Computational Methods

Thinking abstractly

In terms of project structure, I will be using abstraction by representing the game, player, and cards as classes. Doing this will make my program more readable and maintainable as it creates a natural separation of logic.

For the design of the project, I am going to be abstracting the game state by only showing the face up cards in a player's hand instead of the entire state of the game. In addition, the cards will be represented by 2D shapes instead of an entire 3D object.

Thinking ahead

The game is going to need a screen for deciding to create or join a game, a screen that is shown while everyone is waiting for the game to begin, and of course the game itself.

In the game I will be using a mouse as the primary input method as it mimics how the card game would be played in real life and is also the most appropriate for a problem containing many ui elements.

Thinking procedurally & logically

In order to create a multiplayer environment for my game I am going to need a central place to handle the logic of the game. Because of this requirement, I will be using a client-server architecture where players interface with the server in order to join and play rounds of the game.

On the server side of things, its main functionality will be to handle incoming requests from clients: doing different things based on what the request is asking. In addition, it will be coordinating the game loop: sending and receiving appropriate requests based on the current stage of the game.

Clients will also need the ability to communicate with and handle requests from the server, running code conditionally based on the action. For example, an event that indicates the game has started would need to change the screen being shown from the waiting screen to the screen containing the game.

Thinking concurrently

My project will need to be able to do many things simultaneously. For example, multiple games being able to be played at the same time. In addition, I will need to keep in mind that players may complete actions in an unexpected order, for example at some points a player may either place cards or another player will press the cheat button.

To solve this, I will be using event listeners and asynchronous features in NodeJS such as promises and the async/await keywords. In addition, I will conduct tests of multiple games running simultaneously to check if requirements are reached.

Limitations

During the period of development I will be limited by time, meaning I will only be spending limited time on sections of the project that do not award marks, such as art and making all screens of the UI aesthetically pleasing. Despite this, I will aim for consistency in how UIs are designed and look in order to maximise the experience of my stakeholders. In addition, I do not have access to a large number of people so testing to check if my game can handle itself under pressure and with many concurrent games going on may be limited.

Success Criteria

Criteria	Test
The game includes all of the features specified in the project analysis.	All aspects of the feature list are completed.

The game is able to handle multiple games running simultaneously.	Two games can run simultaneously (Given access to a wider audience of testers I would have wished to run more extensive tests than this - see more in my limitations section above)
The game UIs are easy to navigate.	Two-thirds of testers follow the expected path navigating user interfaces and two-thirds say the UIs are easy to use
Players should be able to leave games at any point without breaking gameplay.	For each decision point, have a player leave the game at this point.
No information critical to gameplay (e.g. the cards in opponents hands) should be accessible to clients under any circumstance unless it is specifically for their use.	Check every event that is sent to the client during the lifespan of a game
The game should be able to prevent illegal moves (e.g. placing a card when it is not their turn)	Check each event handler, it should have appropriate safeguards and validation techniques to ensure the event is valid.
The game should be enjoyable	Two-thirds of testers say that they find the game to be enjoyable

Plan for Development and Testing

Development Plan

During the development of my project, I am going to be following an iterative development process according to the agile methodology as this will help create the best product alongside the assistance of my client. To begin with, I will be going through the below stages, gathering feedback on it as I go along:

1. Client: Create main menu and waiting screen
Server: Write game room logic and allow clients to create/join them.
2. Client: Start work on the main game interface. Display a list of cards for the player's "hand", let them select the cards they want to play and allow them to put them back in their hand.
Server: Tell the clients the game has started once the "host" player has pressed start.
3. Client: Only allow cards to be moved on your turn, and press a done button when the player is ready to place them.
Server: Write a basic game loop, going from one player's turn to the next. Tell clients when it is their turn to play.
4. Client: Add a button that allows players to accuse a cheater
Server: Check if players are cheating when a player presses cheat. Add the cards on the table to the hand of the cheating player. Start the round fresh with the player following the one challenged.
5. Client: Create the game over screen, show the winner.
Server: Once a player has placed down their last cards and are unchallenged, they win and the clients are told about this. Exit the game loop. Make sure that any players that leave are handled properly and the game can be continued to be played.

Following this, I will go through stages where I will:

- Check all the success criteria has been met
- Test the game with some of my stakeholders
- Make sure there are no critical issues or bugs with the game
- Otherwise make improvements to the game

Testing Plan

Testing plans for each stage of the project explained in the above development plan.

Stage 1

#	Test	Expected output	Justification
1	A valid username is entered and the create room button is pressed	the screen changes to the game waiting screen and a room code is displayed on the screen	To check if the players can join a game
2	A valid username and valid room code is entered and the user presses join game	They are able to join the game with no problems. the screen changes to the game waiting screen	Invalid data, check if it is handled correctly.
3	An invalid room code that does not exist is entered	An error is displayed stating the room does not	Invalid data, check if it is handled correctly.

	and the user presses join	exist	
4	An invalid username is entered with a valid room code	when the user tries to join a game an error is displayed saying the username is invalid	Invalid data, check if it is handled correctly.
5	When a new player joins the game, the player list updates	the player list updates to display the player that has joined	Check if the player list update event send when a player joins
6	A player leaves the game	They are removed from the player list	Check if the player list update event send when a player leaves
7	The host player leaves the game	A new player is chosen to be the host	Makes sure there is always a host player in the game

Stage 2

#	Test	Expected output	Justification
1	The host player presses the start game button	The waiting screen is hidden for all players and the game screen is now shown for all of them.	Checks if the game can be started
2	A player presses on one of the cards in their hand	The card is moved to the centre of their screen	checks that cards can be placed on the table
3	A player presses on one of the cards previously selected	It returns to their hand	checks that cards can be moved back to the hand from the table
4	A player presses on one of their cards when there are four already on the table	Nothing happens.	check to make sure that players cannot place more cards than they are allowed to

Stage 3

#	Test	Expected output	Justification
1	A player on their turn presses the done button	the cards in their hand are sent to the server. It is now the next player's turn and the previous player can no longer move their cards.	Check what happens when a player's turn is finished
2	A player on their turn presses the done button without placing cards	a message is shown that states they need to place cards down.	check that a turn can only be ended if cards have been placed
3	A player presses on a card without it being their turn	nothing happens	check that the server is only checking for the current player's events

4	The last player ends their turn	The game loops back to the first player's turn.	Check that the game loops correctly
---	---------------------------------	---	-------------------------------------

Stage 4

#	Test	Expected output	Justification
1	A player presses cheat when another player was cheating	the cards on the table go into the hand of the cheater	Check that the cheat detection code works correctly
2	A player presses cheat when the other player was not cheating	the cards on the table go into the hand of the accuser	Check that the cheat detection code works correctly
3	A player presses cheat	play continues from the one after the player challenged	Check that the turn order works correctly

Stage 5

#	Test	Expected output	Justification
1	A player places the last cards in their hand and the following player takes their turn	The player that placed the last cards down wins, show game over screen	Check that the end of game condition is correctly
2	A player leaves the game	The cards from their hand are moved to the discard pile	Check that players leaving mid-game will not break the game
3	A player leaves the game, bringing the total number of players below 3	The game ends. Show the game over screen.	Check that players leaving mid-game will not break the game
4	A player leaves the game on their turn	The next player's turn begins	Check that players leaving mid-game will not break the game
5	A player leaves the game after placing cards	cheat can not be called on them	Check that players leaving mid-game will not break the game

Design Overview

In this section, I will be giving an overview into the design of my program. It will contain information going into the structure, user interfaces and expected behaviours of my game. In addition to this section, I will design the in-depth stages of my program iteratively, written up preceding each development stage.

Overall structure

To begin with I decomposed my game, breaking it down into its core components which can be solved independently.

Structure Diagram

Card Game

- **Server**
 - Websocket connections
 - Request handling
 - Create a game
 - Join a game
 - Game loop
- **Client**
 - Main Menu UI
 - Name entry box
 - Room code
 - Join Game Button
 - Create Game Button
 - Waiting Screen UI
 - Start Game Button
 - Player list
 - Game UI
 - Hand
 - Selected Cards
 - Discard Pile
 - "Cheater" button
 - Exit Button
 - Game Over Screen UI
 - Game over text
 - Play again Button
 - Exit Button

I have separated the game into a client and server module. This is the required architecture for a websocket-based game (which is what I settled on in my technical analysis) and clients will connect to the server using a websocket in order to send and receive actions.

Within the client, I have split the structure out into individual UI screens. I have done this as the majority of the client has different requirements for each screen. In addition, I believe that I will be able to develop the client side code incrementally where one screen is done before the next.

On the server side of the project, I have broken the problem down into the three main components I believe I will have to deal with. The first is the initial connection with websockets, the second is the handling and responses of requests between the server and

clients and the last of which is the running of games, specifically the game loop which is where the logic behind everything will be contained.

From here I will be able to work on each section independently to create designs that I can use to guide my development.

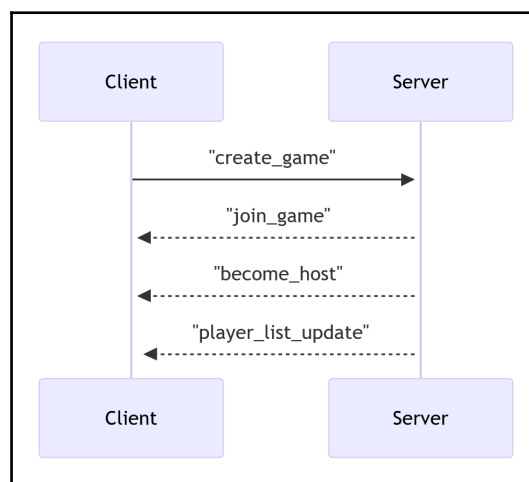
Communication

As stated in my technical analysis, I am going to be using the websockets api and so my project is split into a client and server.

In order to have an ongoing online game, data has to be sent between clients and my server. The websocket api supports sending data as a string, binary object or buffer. For my purposes I will be using strings containing JSON data. Before I send/recieve an action I will parse/stringify the data object I am sending. I will also have an "action" property on each object which states which action is being sent, for example:

```
{  
  "action": "join_game",  
  "code": "ABCDEF",  
  "name": "My Username"  
}
```

When the server receives an action it will handle it, and then send relevant response events. For example, when a client presses the create game button and valid input is given, the following responses will be sent back from the server. This will let the client know that the action was successful, they are the host player, and the content of the player list.



Below I have laid out all of the actions I am planning on initially implementing:

Action		Description	additional attributes
create_game	Client -> Server ▾	Send when a player wants to create a game room	name - string
join_game	Client -> Server ▾	Sent when a player wants to join a game	name - string room_code - string

leave_game	Client -> Server	Sent when a player wants to leave a game	
join_game	Server -> Client	Sent when the server has added a player to a room.	room_code - string
player_list_update	Server -> Client	Sent when a new player joins the game	player_list - Player[]
become_host	Server -> Client	Sent from the server to the client to indicate they are now the "host" player	
start_game	Client -> Server	Sent when the "Start Game" button is pressed	
start_turn	Server -> Client	Sent to all players when somebody's turn starts	your_turn - boolean - <i>set to true if it is this player's turn.</i> previous_card - Card
update_cards	Server -> Client	Send when a player's cards are updated, either due to picking cards up or starting their turn.	cards - Card[]
place_cards	Client -> Server	Sent once a player has selected the cards	cards - Card[]?
place_cards	Server -> Client	Sent when a player has selected their cards	
accuse_cheater	Client -> Server	Sent when a player presses the cheat button	

Now, when one of the above messages is received from a client, I need to trigger different event handlers depending on what action has been sent.

Server Design

Below are the event handlers that communicate with clients before the game begins. In the "create_game" and "join_game" handlers I check for the presence of a username in the request to make sure every player has a name. I also check for the existence of a game with the provided code and return if it doesn't exist.

SERVER - Event handlers

On action "join_game":

```

    Check player's name exists, return otherwise
    Get game object
    if game object does not exist, return
    Add player to game

```

On action "create_game":

```

    Check player's name exists, return otherwise
    Generate game code
    Create game object

```



```
Create player object
Add player to game

On action "leave_game":
    Remove player from game

On action "start_game":
    Start game loop
```

Other actions will solely be handled within the game loop.

Now for the actual data structures: players, games, and cards will be represented using classes containing data and methods relating to them.

For the game class, it will contain the game code associated with it, a list of the players, and the cards on the table.

Game
code - string players - Player[] tablecards - Card[]
startGame() gameLoop() addPlayer(player) removePlayer(player)

SERVER - Game Class Methods**Add Player:**

- add player to players array
- tell clients the new players list

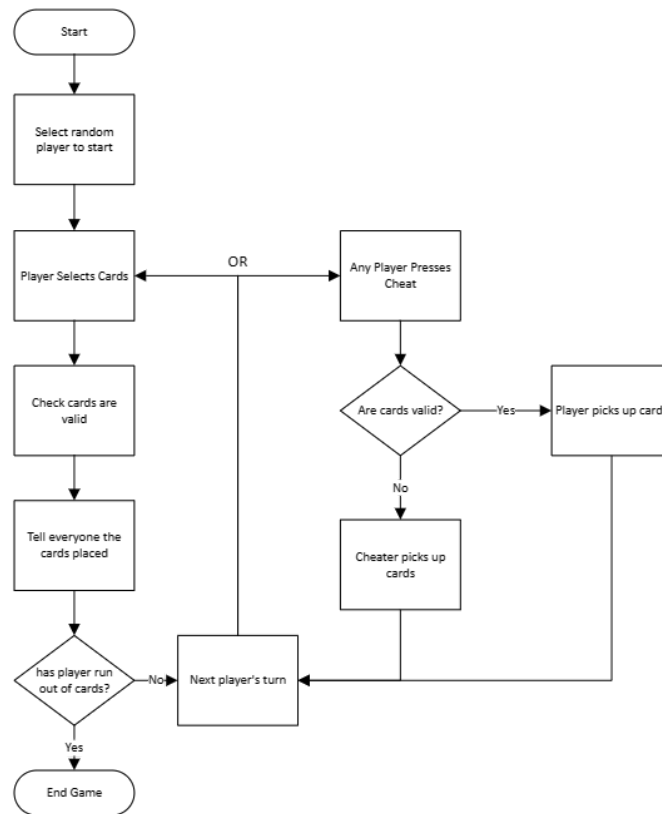
Remove Player:

- remove player from players list
- if player was host:
 - add new player as host

Start Game:

- Generate deck of cards
- Shuffle deck of cards
- while deck has cards:
 - for each player:
 - if deck has cards:
 - pop the top card off the deck
 - add the card to the player's hand
- for each player:
 - send "game_start" action
 - send "update_cards" action
- start game loop

For the actual game loop, I initially created a flow chart so I could visualise how it should work.

**SERVER - Game Class Methods (Continued)**

Game Loop:

```

table_cards = []
just_placed_cards = []
previous_rank = 0
while game ongoing:
    for each player:
        broadcast action "turn_start"
        send "update_cards" event to current player
        wait for "place_cards" event or any "accuse_cheater" event

        if event is "place_cards" then
            table_cards = table_cards + just_placed_cards
            set just_placed_cards to cards in event

        else if event is "cheat" and len(just_placed_cards) > 0 then
            cards_to_pickup = table_cards + just_placed_cards

            if all of just_placed_cards == previous_rank then
                give cards_to_pickup to accuser hand
            else
                give cards_to_pickup to cheater hand
            endif

            table_cards = []
            just_placed_cards = []
        endif

    previous_rank += 1
    if previous_rank > 12 then
        previous_rank = 0
    endif
  
```

next player

For players within the game, each of them is to have their own object. It will contain an id to identify them, a hand array containing a list of cards, a reference to the game they are playing within, and a reference to the websocket that is used to communicate with them.

Player
id - number hand - Card[] game - Game ws - websocket connection

Cards will also be stored as objects. they will contain a number that is associated with their rank and a string for their suit.

Card
rank - number suit - string

For simplicity when comparing card values I have reduced their ranks into numbers representing each rank:

A	2	3	4	5	6	7	8	9	10	J	Q	K
0	1	2	3	4	5	6	7	8	9	10	11	12

Below are the functions that I am going to be implementing in order to shuffle and handle cards.

For the shuffling algorithm, I will be using the Fisher-Yates algorithm as it is unbiased.

For the sorting algorithm I will just be using the built in javascript sort method as there is no reason to reinvent something that already exists for me to use.

SERVER

```
function generateDeck()  
  deck = []  
  
  for each rank:  
    for each suit:  
      add new Card(rank, suit) to deck  
  
  return deck
```

```
function shuffle(deck)
  for i = n-1 to 1 step -1
    j = randomint(0, i)
    temp = deck[i]
    deck[i] = deck[j]
    deck[j] = temp
  next

function sort(deck)
  // sort using the built in javascript array sort method
```

File Structure

I am going to separate each class into its own file/module. For example, the game class will have its own module, the player class will have a module, and the card class will have its own file. Related functions will also be contained in these modules.

Client Design

Now onto the client side of my project, the structure will be quite similar to the server. The main logic of the client will be concentrated on communication between itself and the server. Like the server, it will have a variety of server actions it will respond to through websocket event listeners.

In order to create and join game rooms the following event handlers will be used:

CLIENT - Event handlers

```
On action "join_game":
  Hide main menu screen
  Show waiting screen
  set game code element

On action "update_player_list":
  Remove player list content
  For each player:
    Add name to player list

On action "become_host":
  Show "Start Game" button

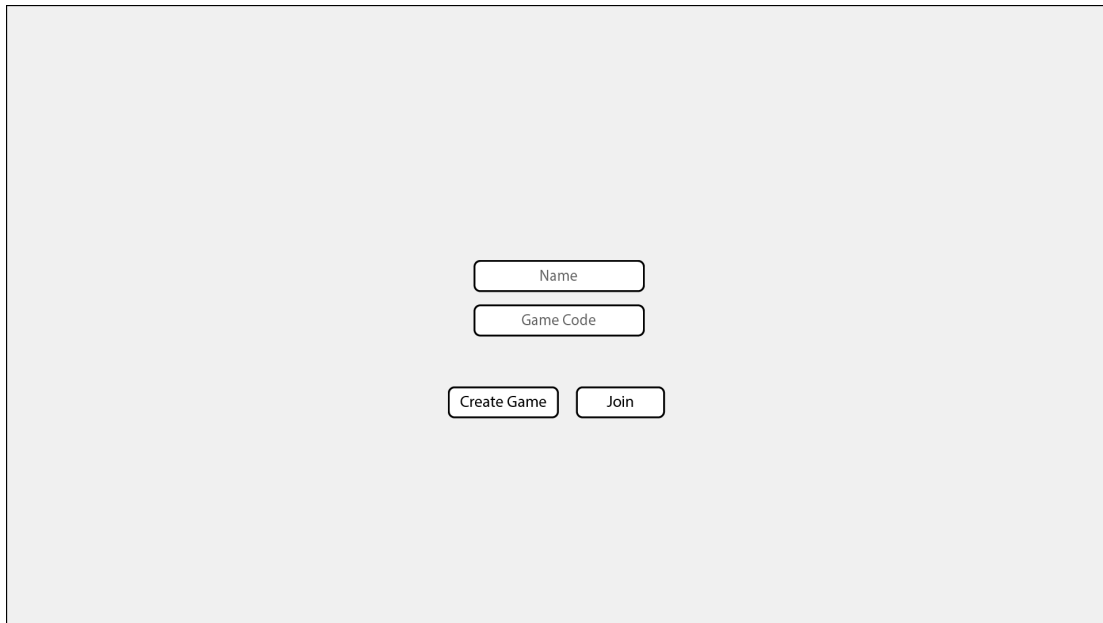
On action "start_turn":
  show "Your turn" text
  allow moving cards between hand and table
  enable end turn button

On action "update_cards":
  update card list

On action "place_cards":
  Update just placed text
```

Each "screen" on the client will be contained in an html div element. The client code will have a reference to each of them saved in variables and will show/hide these elements when needed in response to actions or events from the server.

Main Screen



The Main Screen is a light gray rectangular area. In the center, there are four input elements. At the top is a text input field labeled "Name". Below it is another text input field labeled "Game Code". At the bottom, there are two buttons: "Create Game" on the left and "Join" on the right.

CLIENT

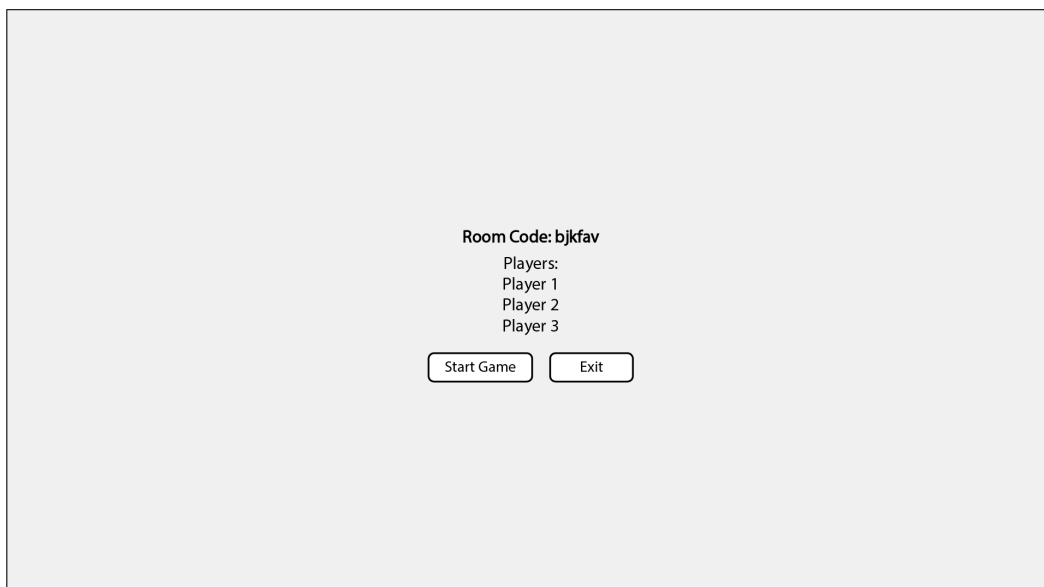
Press "Create Game":

send "create_game" event with name from "Name" input element

Press "Join"

send "join_game" event with code from "Game Code" input element

Waiting Screen



The Waiting Screen is a light gray rectangular area. In the center, the text "Room Code: bjkfav" is displayed. Below it, the text "Players:" is followed by a list of three players: "Player 1", "Player 2", and "Player 3". At the bottom, there are two buttons: "Start Game" on the left and "Exit" on the right.

The waiting screen will also be created with html/css elements.

CLIENT

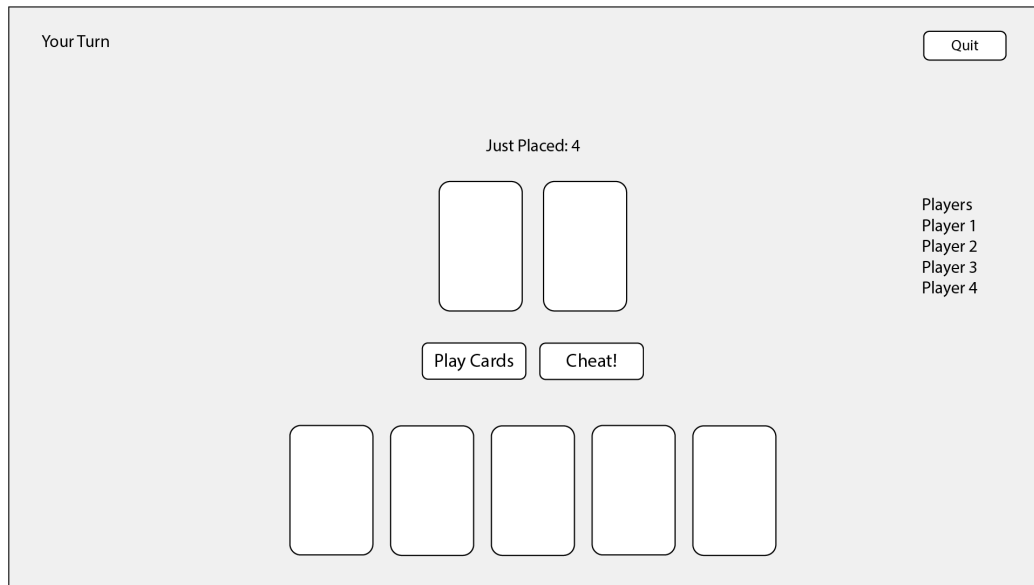
Press "Start Game":

```
send "create_game" event with name from "Name" input element
```

Press "Exit":

```
send "leave_game" event  
hide waiting screen  
show main menu
```

Play Screen



For creating the play screen for my project, I had to decide if i wanted to use a canvas or html dom elements for representing my UI.

Overview of using a canvas

If I were to use a canvas for the game, I would be able to control every element of what is displayed on the screen. For UI elements, I would be drawing ui using an immediate mode paradigm. This means that I would have to explicitly draw every card, the text, etc every frame. The advantage of this approach is that I would be able to control every pixel on the canvas.

Overview of using html elements

If I were to use just html dom elements, I would be using a retained mode GUI paradigm. This means I would only reference and modify ui elements when an event or server action occurs. In addition, I would be able to leverage the features built in to html/css to create a responsive game that adapts to the screen it is being played on.

Overall, I decided to go for the approach of using html elements as it would work the best with the push-based action system described in the server design.

CLIENT

Press card in hand:

```
if handCards length < 4 then  
  remove card from handCards  
  add card to selectedCards
```

Press selected card:

```
remove card from selectedCards  
add card to handCards
```

Press "Play Cards":

```
send "play_cards" event with contents of selectedCards
```

Press "Cheat":

```
send "accuse_cheater" event
```

Press "Exit":

```
send "leave_game" event  
hide play screen  
show main menu
```

CLIENT

function Update Cards:

```
remove contents of selected cards element, hand cards div  
for card of handCards:  
    add card to hand cards div
```

for card of selectedCard:

```
    add card to selected cards div
```

Game Over Screen



This is the screen shown once a player has won the game.

CLIENT

Press "Play Again":

```
send "play_again" event
```

Press "Exit":

```
send "leave_game" event  
hide play screen  
show main menu
```


Client - Main Variables

Variable	Type / Structure	Description
socket	websocket	the websocket used to communicate with the backend server
joinGameScreen	html element	the html element containing the join game screen
waitingScreen	html element	the html element containing the waiting screen
playScreen	html element	the html element containing the play screen
endScreen	html element	the html element containing the end screen
handCards	ClientCard[]	All the cards currently in the player's hand
selectedCards	ClientCard[]	The cards the player has selected

Client - Functions

Function	Description
joinGame	Ran when someone presses the join game button on the main menu
startGame	Ran when someone presses start game
receiveMessage	Triggered when a message is sent from the server

Iterative Design & Development

Stage 1

Create the main structure of the program, display the webpage, and allow clients to join a game "room".

Development

1.1

To begin with, I initialised my node js project, and created a base structure and the required files to

1. Start a server
2. Host html/js files which are accessible locally
3. Setup, send and receive websocket messages

server/main.js

```
server >  main.js
1  import express from "express"
2  import http from "http"
3  import { WebSocketServer } from "ws"
4
5  const app = express()
6  const server = http.createServer(app)
7  const websocketServer = new WebSocketServer({ server, path: "/ws" })
8
9  // make all files within the client directory accessible
10 app.use(express.static('client'))
11
12 // handle the websocket
13 websocketServer.addListener("connection", (websocket) => {
14   console.log("Recieved websocket connection.")
15
16   websocket.send(JSON.stringify({
17     action: "TestMessageToClient"
18   }))
19
20   websocket.addListener("message", (message) => {
21     console.log("Received message:", message.data)
22     // TODO: handle messages
23   })
24
25   websocket.addListener("close", () => {
26     console.log("Websocket closed.")
27     // TODO: handle closing of the websocket
28   })
29 })
30
31 server.listen(3000, () => {
32   console.log(`Server started.`)
33 })
```

client/index.html

client >  index.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Cheat</title>
7 </head>
8 <body>
9   <h1>Hello, World!</h1>
10  <script src="index.js"></script>
11 </body>
12 </html>
```

client/index.jsclient >  index.js

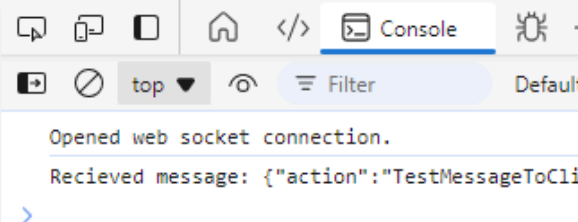
```
1 const websocket = new WebSocket(`https://${window.location.host}/ws`)
2
3 websocket.addEventListener("open", () => {
4   console.log("Opened web socket connection.")
5
6   websocket.send(JSON.stringify({
7     action: "TestMessageToServer",
8   }))
9 })
10
11 websocket.addEventListener("message", (message) => {
12   console.log("Recieved message:", message.data)
13 })
```

Note: I am going to leave the message receive function inlined here rather than creating a separate receiveMessage function as stated in my designs as it will only be called from this context as an event handler and so is not needed elsewhere.

This worked well, with the test message being sent both to the client and the server. When a page with an active websocket is closed, the server is also told about this.

Client Output:

Hello, World!

**Server Output:**

```
Restarting 'server/main.js'  
Server started.  
Recieved websocket connection.  
Received message: {"action":"TestMessageToServer"}  
Websocket closed.  
□
```

1.2

I then created the necessary ui elements for the main menu and waiting screen, as below:

client/index.html

```
snippets > index.html  
1  ...  
2  <body>  
3    <div id="joinGameScreen">  
4      <h1>Cheat</h1>  
5      <p>By Ted Brunner</p>  
6  
7      <input id="nameEntry" spellcheck="false" type="text" placeholder="Name">  
8      <input id="gameCodeEntry" spellcheck="false" type="text" placeholder="Game Code">  
9      <div class="buttonRow">  
10       <button id="joinRoom">Join Room</button>  
11       <button id="createRoom">Create Room</button>  
12     </div>  
13     <p id="joinGameErrorText" class="hidden"></p>  
14   </div>  
15   <div id="waitingScreen" class="hidden">  
16     <p>  
17       Room Code:  
18       <span id="roomCodeText"></span>  
19     </p>  
20     <p>Players:</p>  
21     <ul id="playerList">  
22     </ul>  
23     <div class="buttonRow">  
24       <button id="startGame">Start Game</button>  
25       <button id="exitGame">Exit</button>  
26     </div>  
27   </div>  
28   ...  
29 </body>
```

And once styled, the main menu looked like this:

Cheat

Name

Game Code

Join Room Create Room

1.3

To add functionality to it, I needed to create classes for Player and Game as below:

server/player.js

```
export class Player {
  // id: number
  // name: string
  // ws: Websocket
  // hand: Card[]
  // game: null or Game
  // isHost: boolean
  constructor(ws, name, isHost) {
    this.name = name
    this.ws = ws
    this.isHost = isHost
    this.hand = []
  }
}
```

server/game.js

```
export class Game {
  constructor(gameCode) {
    this.code = gameCode
    this.players = []
  }
}
```

1.4

I decided to use a map data structure instead of a list (as I stated in my design) for storing the game objects. This is because using a list would mean that I would have to loop over them until the correct game is found. In addition, each game already has a unique id so this can be used again for the map key.

Accessing games using the game Map:

```
const games = new Map()
// ...
if (games.has(code)) {
  // ...

  const game = games.get(code)
  game.addPlayer(player)
}
```

1.5

To respond to actions on the server, I added the following logic to the server's message event listener:

server/main.js (join game message handler)

```
if (action === "join_game") {
  const code = messageData.room_code
  const name = messageData.name
  if (name.length === 0) {
    websocket.send(JSON.stringify(
      {
        action: "join_game_error",
        message: "The provided username is invalid."
      }
    ))
    return
  }

  if (games.has(code)) {
    player = new Player(websocket, name, false)

    const game = games.get(code)
    game.addPlayer(player)
  } else {
    websocket.send(JSON.stringify(
      {
        action: "join_game_error",
        message: "The game with the provided code does not exist."
      }
    ))
  }
}
```

This checks if the client has sent valid information, and if so it will call the addPlayer method on the required game object, otherwise it will send an error to the client.

Below is the mentioned addPlayer method defined on the Game objects.

```
addPlayer(player) {
  this.players.push(player)
  player.game = this
  player.ws.send(JSON.stringify({
    action: "join_game",
    room_code: this.code
  })))

  const playerList = []
  for (const player of this.players) {
    playerList.push({
      name: player.name,
      isHost: player.isHost,
      isMe: player === player
    })
  }
  for (const player of this.players) {
    player.ws.send(JSON.stringify({
      action: "player_list_update",
      players: playerList
    })))
  }
}
```

Also in the join game message handler is the use of a new message type "join_game_room" which I added when I realised that there was no way for the server to communicate to the client that an error occurred if incorrect input was provided. The action has one attribute containing the required error message.

Action		attributes
join_game_error	Server -> Client ▾	message - string

When sent to the client the error looks like this:

Cheat

By Ted Brunner

The game with the provided code does not exit.

1.6

server/main.js (create game message handler)

```
} else if (action === "create_game") {
  const name = messageData.name
  if (name.length === 0) {
    websocket.send(JSON.stringify(
      {
        action: "join_game_error",
        message: "The provided username is invalid."
      }
    ))
    return
  }

  const code = generateCode()
  const game = new Game(code)
  games.set(code, game)

  player = new Player(websocket, name, true)

  game.addPlayer(player)
}
```

While conducting end of stage testing, I found that I had no way to test if a player was indeed the "host" player. I added an attribute to the client player objects sent "isHost" to tell clients about it.

I filled in the generateCode function which generates a random code out of 5 alphabet characters. It does this by creating a random offset from the "a" character and then adding it to the "a" character's character code, before turning the character code back into text. It puts these all together and then returns the code.

```
export function generateCode() {
  let code = ""
  for (let i = 0; i < 5; i++) {
    code += String.fromCharCode("a".charCodeAt(0) + Math.floor(Math.random() * 26))
  }
  return code
}
```

1.7

Finally, I added the leave_game message handler, which calls the removePlayer method on the game.

server/game.js

```
else if (action === "leave_game") {
  const game = player?.game
  if (game) {
    game.removePlayer(player)
  }
}
```



```
removePlayer(playerToRemove) {
  const remainingPlayers = []
  for (const player of this.players) {
    if (player === playerToRemove) {
      continue
    } else {
      remainingPlayers.push(player)
    }
  }

  this.players = remainingPlayers

  if (this.players.length === 0) {
    return
  }

  if (playerToRemove.isHost) {
    this.players[0].isHost = true
  }

  this.updatePlayerList()
}
```

I also extracted the logic for updating the player list into its own method as it was duplicated in both addPlayer and removePlayer.

```
updatePlayerList() {
  for (const player of this.players) {
    const playerList = []
    for (const player2 of this.players) {
      playerList.push({
        name: player2.name,
        isHost: player2.isHost,
      })
    }
    player.ws.send(JSON.stringify({
      action: "player_list_update",
      players: playerList
    }))
  }
}
```

End of Stage Testing

#	Test	Expected output	Output	Output Description
1	A valid username is entered and the create room button is pressed	the screen changes to the game waiting screen and a	1.	1. Before pressing the "Create Room" button 2. After Pressing "Create Room"

		room code is displayed on the screen	<p>Cheat</p> <p>Player</p> <p>Game Code</p> <p>Join Room Create Room</p> <p>2.</p> <p>Room Code: sbzeej</p> <p>Players:</p> <ul style="list-style-type: none"> • Player <p>Start Game Exit</p>	
2	A valid username and valid room code is entered and the user presses join game	They are able to join the game with no problems. the screen changes to the game waiting screen	<p>1.</p> <p>Cheat</p> <p>Player 2</p> <p>sbzeej</p> <p>Join Room Create Room</p> <p>2.</p> <p>Room Code: sbzeej</p> <p>Players:</p> <ul style="list-style-type: none"> • Player • Player 2 <p>Start Game Exit</p>	<p>1. Before pressing the "Join Room" button</p> <p>2. After Pressing "Join Room"</p>
3	An invalid room code that does not exist is entered and the user presses join	An error is displayed stating the room does not exist	<p>1.</p> <p>Cheat</p> <p>Some Player</p> <p>invalid room code!!!!</p> <p>Join Room Create Room</p> <p>2.</p>	<p>1. An invalid room code is entered</p> <p>2. An error message is displayed</p>

			<p style="text-align: center;">Cheat</p> <div> <input type="text" value="Some Player"/> <input type="text" value="Invalid room code!!!!"/> <input type="button" value="Join Room"/> <input type="button" value="Create Room"/> </div> <p style="text-align: center; color: red;">The game with the provided code does not exist.</p>	
4	An invalid username is entered with a valid room code	when the user tries to join a game an error is displayed saying the username is invalid	<p style="text-align: center;">1.</p> <p style="text-align: center;">Cheat</p> <div> <input type="text" value="Name"/> <input type="text" value="sbzeej"/> <input type="button" value="Join Room"/> <input type="button" value="Create Room"/> </div> <p style="text-align: center;">2.</p> <p style="text-align: center;">Cheat</p> <div> <input type="text" value="Name"/> <input type="text" value="sbzeej"/> <input type="button" value="Join Room"/> <input type="button" value="Create Room"/> </div> <p style="text-align: center; color: red;">The provided username is invalid.</p>	<ol style="list-style-type: none"> Before pressing "Join Room" After pressing "Join Room"
5	When a new player joins the game, the player list updates	the player list updates to display the player that has joined	<p style="text-align: center;">1.</p> <p style="text-align: center;">Room Code: sbzeej</p> <p style="text-align: center;">Players:</p> <ul style="list-style-type: none"> • Player <div> <input type="button" value="Start Game"/> <input type="button" value="Exit"/> </div> <p style="text-align: center;">2.</p> <p style="text-align: center;">Room Code: sbzeej</p> <p style="text-align: center;">Players:</p> <ul style="list-style-type: none"> • Player • Player2 <div> <input type="button" value="Start Game"/> <input type="button" value="Exit"/> </div>	<ol style="list-style-type: none"> Before "Player2" joins After "Player2" joins
6	A player leaves the game	They are removed from the player list	<p style="text-align: center;">1.</p>	<ol style="list-style-type: none"> Before player 2 joins After player 2 joins

			<p>Room Code: gduijc</p> <p>Players:</p> <ul style="list-style-type: none"> • Player • Player 2 <p>Start Game Exit</p> <p>2.</p> <p>Room Code: gduijc</p> <p>Players:</p> <ul style="list-style-type: none"> • Player <p>Start Game Exit</p>	
7	The host player leaves the game	A new player is chosen to be the host	<p>1.</p> <p>Room Code: ozbber</p> <p>Players:</p> <ul style="list-style-type: none"> • Player 1 (host) • Player 2 <p>Start Game Exit</p> <p>2.</p> <p>Room Code: ozbber</p> <p>Players:</p> <ul style="list-style-type: none"> • Player 2 (host) <p>Start Game Exit</p>	<p>1. before the host leaves</p> <p>2. after the host leaves</p>

User Feedback

I asked some of the members of the people I identified as my target audience from my analysis section for their thoughts on the project so far and they said:

1. Person 1: It is difficult to see who you are in the waiting screen
2. Person 1: The start game button should only be visible for the host player
3. Person 2 & 3: They had trouble with the room code system, both of them entered their own code into the game code box.

Feedback Point 1

To show players who they are I added an additional property to the player list, "isMe", which is sent whenever the player list updates.

```
updatePlayerList() {  
  for (const player of this.players) {  
    const playerList = []  
    for (const player2 of this.players) {  
      playerList.push({  
        name: player2.name,  
        isHost: player2.isHost,  
        isMe: player === player2  
      })  
    }  
    player.ws.send(JSON.stringify({  
      action: "player_list_update",  
      players: playerList  
    })))  
  }  
}
```

And then on the client:

```
case "player_list_update":  
  playerList.innerHTML = ""  
  for (const player of messageData.players) {  
    const element = document.createElement("li")  
    let text = player.name  
    if (player.isHost) {  
      text = text + " (host)"  
    }  
    if (player.isMe) {  
      text = text + " - me"  
    }  
    element.innerText = text  
    playerList.appendChild(element)  
  }  
  break
```

Feedback Point 2

For this point, I made it so that the "become_host" event triggered and then changed the visibility of the Start Game based on the value provided in it.

Room Code: bqixhn

Players:

- player 1 (host)
- player 2 - me

Exit

Room Code: bqixhn

Players:

- player 2 (host) - me

Start Game

Exit

Feedback Point 3

As two of my target audience had issues understanding the UI I decided I had to change it. Two out of the three people that tested this stage put their own code in the game code box before joining the game, and then when they went to add another player into the game they put their code in instead of the code that was generated for them

I came up with three options:

- Require users to create a game code when they are making a game and then make this the code that people have to put in
- Separate the main screen into one section for joining a game and one section for creating a game
- Create an intermediary screen which only shows when the join room button is clicked which is where the game code entry exists.

I decided to split the sections where players join and create games because it lets players see all the pathways they can take and will reduce the number of clicks needed to join the game. Firstly, I created a modified version of a screenshot using paint to get an idea of what it should look like:

Cheat

Join a Game

Name

Game Code

Join Room

Create a Game

Name

Create Room

which I then used as guidance for modifying the user interface:

Cheat

Join a Game

Create a Game

I asked my stakeholders if this version of the ui was clearer and they all said yes.

Stage 2

Create the main interface where most of the game takes place and show the cards in the player's hand.

Development

2.1 - Creating the cards module

As cards will be used on both the client and server I am creating one javascript module that can be used for both. I am putting it in a new folder named "common".

In order for this new file to be accessible in the client I added "common" as a new static directory:

```
app.use(express.static('client'))
app.use(express.static('common'))
```

Now in common/card.js I created the card class:

```
export class Card {
  constructor(rank, suit) {
    this.rank = rank
    this.suit = suit
  }
}
```

As well as the functions I specified in my design phase

```
export function generateDeck() {
  const deck = []
  for (let suit = 0; suit < 4; suit++) {
    for (let rank = 0; rank < 13; rank++) {
      const card = new Card(rank, suit)
      deck.push(card)
    }
  }
  return deck
}

export function shuffleCards(deck) {
  for (let i = deck.length - 1; i >= 1; i--) {
    const j = Math.floor(Math.random() * i)
    const temp = deck[i]
    deck[i] = deck[j]
    deck[j] = temp
  }
  return deck
}
```

I also added constants that directly correlate to the numeric versions of ranks and suits:

```
export const SUITS = ["hearts", "diamonds", "spades", "clubs"]
export const SUIT_SYMBOLS = ["♥", "♦", "♠", "♣"]
export const RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]
```

2.2

I then added a handler for the "start_game" action on the server.


```

} else if (action === "start_game") {
  const game = player?.game

  if (player.isHost && game) {
    game.startGame()
  }
}

```

It checks if the player sending the action is inside a game and if they are the host of the game. If either of these conditions are not true, the game will not start. If the conditions are both true, it will execute the runGameLoop method on the game, which for now just tells all the players that the game has begun.

```

startGame() {
  let deck = generateDeck()
  deck = shuffleCards(deck)

  while (deck.length > 0) {
    for (const player of this.players) {
      const card = deck.pop()
      if (card !== undefined) {
        player.hand.push(card)
      }
    }
  }

  for (const player of this.players) {
    player.ws.send(JSON.stringify({
      action: "start_game",
    }))
    player.ws.send(JSON.stringify({
      action: "update_cards",
      cards: player.hand.map(card => card.getData())
    }))
  }

  this.gameLoop()
}

gameLoop() {
  // TODO
}

```

2.3

When the "start_game" action is received on the client, It hides the waiting screen and shows the play screen.

```

} else if (action == "start_game") {
  waitingScreen.classList.add("hidden")
  playScreen.classList.remove("hidden")
}

```

I created arrays for the hand and selected cards:

```

let handCards = []
let selectedCards = []

```

and then added the response handler for it:

```
} else if (action == "update_cards") {  
  handCards = []  
  selectedCards = []  
  for (const {id, rank, suit} of messageData.cards) {  
    handCards.push(  
      new Card(id, rank, suit)  
    )  
  }  
  
  updateCards()  
}
```

I created the updateCards() function that gets called whenever there is a change in the hand or selected cards array.

```
function updateCards() {  
  handCardsContainer.innerHTML = ""  
  selectedCardsContainer.innerHTML = ""  
  
  for (const card of handCards) {  
    const cardElement = createCardElement(card)  
    handCardsContainer.appendChild(cardElement)  
  }  
  
  for (const card of selectedCards) {  
    const cardElement = createCardElement(card)  
    selectedCardsContainer.appendChild(cardElement)  
  }  
}
```

Instead of creating the card elements within the updateCards function I created a new function, reducing duplicated code:

```
function createCardElement(card) {  
  const cardElement = document.createElement("div")  
  cardElement.classList.add("card")  
  
  if (card.suit == 0 || card.suit == 1) {  
    cardElement.classList.add("redCard")  
  }  
  
  const cardElementText = document.createElement("p")  
  cardElementText.classList.add("rankIcon")  
  cardElementText.innerText = RANKS[card.rank]  
  
  const suitIconTop = document.createElement("p")  
  suitIconTop.classList.add("suitIcon")  
  suitIconTop.classList.add("top")  
  suitIconTop.innerText = SUIT_SYMBOLS[card.suit]  
  
  const suitIconBottom = document.createElement("p")  
  suitIconBottom.classList.add("suitIcon")  
  suitIconBottom.classList.add("bottom")  
  suitIconBottom.innerText = SUIT_SYMBOLS[card.suit]  
  
  cardElement.appendChild(suitIconTop)  
  cardElement.appendChild(suitIconBottom)  
  cardElement.appendChild(cardElementText)  
  
  return cardElement  
}
```

Each card is represented by an html div element with text (p elements) contained within it to show what card and suit it belongs to.

This function creates a card element with markup similar to this:

```
<div class="card redCard">  
  <p class="suitIcon top">♥</p>  
  <p class="rankIcon">K</p>  
  <p class="suitIcon bottom">♥</p>  
</div>
```

Which looks like this when displayed on the page:



I then needed to add event listeners to listen to click events for swapping cards between the selected and unselected lists.

```

function updateCards() {
  handCardsContainer.innerHTML = ""
  selectedCardsContainer.innerHTML = ""

  for (const cardIndex in handCards) {
    const card = handCards[cardIndex]
    const cardElement = createCardElement(card)
    cardElement.addEventListener("click", () => {
      handCards.splice(cardIndex, 1)
      selectedCards.push(card)
      updateCards()
      console.log(handCards, selectedCards)
    })
    handCardsContainer.appendChild(cardElement)
  }

  for (const card of selectedCards) {
    const cardElement = createCardElement(card)
    selectedCardsContainer.appendChild(cardElement)
  }
}

```

In green is the added event listener which was intended to move cards from my hand to the table.

Before pressing on a card:



After pressing on a card:



As you can see, the cards didn't move from the hand, they just duplicated at the top. To debug this I logged the values of the hand and selected arrays in the console (highlighted in yellow).

```
► (3) [Card, Card, Card] ► [Card]
```

```
► (2) [Card, Card] ► (2) [Card, Card]
```

But the result seemed correct. I then identified the issue: I was setting `innerHTML` instead of the correct name `innerHTML`.

I fixed this issue and copied it for moving cards back from the selected list to the hand.

Below is the final version of this function, with all the changes made and added events on the selected cards array of cards to move them back into the hand as well as a restriction on the number of cards that can be selected at one time.

```
function updateCards() {
  handCardsContainer.innerHTML = ""
  selectedCardsContainer.innerHTML = ""

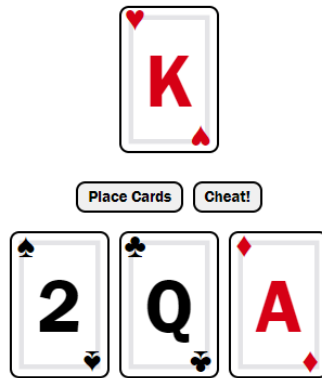
  for (const cardIndex in handCards) {
    const card = handCards[cardIndex]
    const cardElement = createCardElement(card)
    cardElement.addEventListener("click", () => {
      if (selectedCards.length >= 4) {
        return
      }
      handCards.splice(cardIndex, 1)
      selectedCards.push(card)
      updateCards()
    })
    handCardsContainer.appendChild(cardElement)
  }

  for (const cardIndex in selectedCards) {
    const card = selectedCards[cardIndex]
    const cardElement = createCardElement(card)
    cardElement.addEventListener("click", () => {
      selectedCards.splice(cardIndex, 1)
      handCards.push(card)
      updateCards()
    })
    selectedCardsContainer.appendChild(cardElement)
  }
}
```

Before clicking a card:



After clicking on a card:



End of Stage Testing

#	Test	Expected output	Output	Output Description
1	The host player presses the start game button	The waiting screen is hidden for all players and the game screen is now shown for all of them.	<p>1.</p> <p>Room Code: ojwrhy</p> <p>Players:</p> <ul style="list-style-type: none"> • Player 1 (host) - me • Player 2 • Player 3 <p>Start Game Exit</p> <p>2.</p> <p>Your Turn Exit</p> <p>Place Cards Cheat!</p>	<p>1. Before pressing start button</p> <p>2. After pressing start button</p>
2	A player presses on one of the cards in their hand	The card is moved to the centre of their screen	<p>1.</p> <p>Your Turn Exit</p> <p>Place Cards Cheat!</p> <p>2.</p>	<p>1. Before pressing on card</p> <p>2. After pressing on the 2 of spades card</p>

			<p>Your Turn EXIT</p>	
3	A player presses on one of the cards previously selected	It returns to their hand	<p>1.</p> <p>Your Turn EXIT</p> <p>2.</p> <p>Your Turn EXIT</p>	<ol style="list-style-type: none"> 1. Before pressing on the selected card 2. After pressing on the selected card
4	A player presses on one of their cards when there are four already on the table	Nothing happens.	<p>1.</p> <p>Your Turn EXIT</p> <p>2.</p> <p>Your Turn EXIT</p>	<ol style="list-style-type: none"> 1. Before pressing on card in hand 2. After pressing on card in hand (no result)

User Feedback

For this stage, I again asked my stakeholders for feedback. From their feedback I got the following points:

1. Make the cards sort when they are moved between the hand and the selected cards
2. It's a bit vague what the name input boxes mean
3. When copy and pasting codes, it does not always seem to work

Feedback Point 1

To make the cards sort, I implemented the sort function specified in my design, however I renamed it to sortCardsByRank for clarity.

```
export function sortCardsByRank(deck) {
  return deck.sort((card1, card2) => card1.rank - card2.rank)
}
```

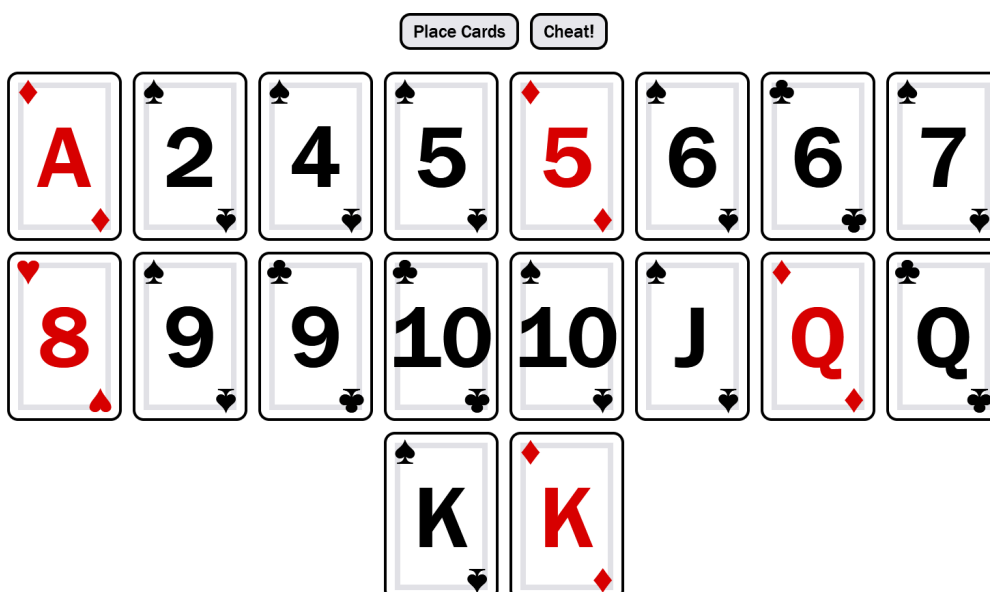
I then made the player's hands sort before being sent to them in the startGame method.

```
for (const player of this.players) {
  player.hand = sortCardsByRank(player.hand)
  player.ws.send(JSON.stringify({
    action: "start_game",
  }))
  player.ws.send(JSON.stringify({
    action: "update_cards",
    cards: player.hand.map(card => card.getData())
  }))
}
```

When playing the game, cards get moved back and forth between the player's hand and the selected hand. I also added the sort function here that is ran when cards are moved back:

```
cardElement.addEventListener("click", () => {
  selectedCards.splice(cardIndex, 1)
  handCards.push(card)
  handCards = sortCardsByRank(handCards)
  updateCards()
})
```

I decided not to add it for the selected cards array as this should make it easier for the player to track where the cards they place are going.



Feedback Point 2

When creating games, there was some confusion over if the Name input box set your username or the name of the game room. To fix this I renamed the placeholder text to Username

Cheat

Join a Game	Create a Game
<input type="text" value="Username"/>	<input type="text" value="Username"/>
<input type="text" value="Game Code"/>	
<input type="button" value="Join Room"/>	<input type="button" value="Create Room"/>

Feedback Point 3

I found when players copy the game code from the waiting screen there is sometimes some whitespace either side of the game code. When pasting this into the game code entry box it will not recognise the code as valid.

To fix this, I applied the trim method to the game code string before sending it to the server.

```
room_code: gameCodeEntry.value.trim()
```

This fixed the issue.

Stage 3 / 4

Although initially I split the development of the game loop into two stages, as I began I realised that this would not be a good approach as the game loop logic relies on there being two events that could occur: a card being placed or someone pressing cheat. For this reason, I merged the two stages into one.

Development

To begin with, I began writing the gameLoop function.

```
async gameLoop() {
  let previousRank = null
  let currentRank = 0

  let tableCards = []
  let justPlacedCards = []

  let playerIndex = 0
  let previousPlayer = null

  [3.1]
  while (true) {
    const player = this.players[playerIndex]

    [3.2]
    this.currentPlayer = player
    this.updatePlayerList()

    [3.3, 3.4, 3.5]
    for (const otherPlayer of this.players) {
      otherPlayer.ws.send(JSON.stringify({
        action: "start_turn",
        previous_rank: previousRank,
        current_rank: currentRank,
        can_accuse_cheater: previousPlayer !== null && previousPlayer !== otherPlayer,
        your_turn: otherPlayer === player,
      }))
    }

    [3.6]
    let messageData;
    let messageSource;
    // wait for either:
    // - place_cards event from current player
    // - accuse_cheater event from any other player
    await new Promise((resolve) => {
      const placeCardsListener = (message) => {
        const possibleMessageData = JSON.parse(message.data)
        const action = possibleMessageData.action
        if (action === "place_cards") {
          player.ws.removeEventListener("message", placeCardsListener)
          resolve()
          messageData = possibleMessageData
          messageSource = player
        }
      }
      player.ws.addEventListener("message", placeCardsListener)

      for (const player of this.players) {
        const accuseCheaterListener = (message) => {
          const possibleMessageData = JSON.parse(message.data)
```

```
        const action = possibleMessageData.action
        if (action === "accuse_cheater") {
            resolve()
            messageData = possibleMessageData
            messageSource = player
        }
    }
    player.ws.addEventListener("message", accuseCheaterListener)
}
})

const action = messageData.action
if (action === "place_cards") {
    const cards = []
    // remove cards from player hand
    for (const cardData of messageData.cards) {
        cards.push(new Card(cardData.rank, cardData.suit))
        for (const cardIndex in player.hand) {
            const cardInHand = player.hand[cardIndex]
            if (cardInHand.rank === cardData.rank && cardInHand.suit === cardData.suit) {
                player.hand.splice(cardIndex, 1)
                break
            }
        }
    }
    tableCards.push(...justPlacedCards)
    justPlacedCards = cards
    player.ws.send(JSON.stringify({
        action: "update_cards",
        cards: player.hand.map(card => card.getData())
    }))
    previousPlayer = player
    previousRank = currentRank

    currentRank += 1
    if (currentRank > 12) {
        currentRank = 0
    }

    playerIndex += 1
    if (playerIndex ≥ this.players.length) {
        playerIndex = 0
    }
} else if (action === "accuse_cheater") {
    let allCardsValid = true
    for (const card of justPlacedCards) {
        if (card.rank ≠ previousRank) {
            allCardsValid = false
        }
    }
}

[3.7]
let playerToPickup = null
if (allCardsValid) {
    playerToPickup = messageSource
} else {
    playerToPickup = player
}
playerToPickup.hand.push(...tableCards, ...justPlacedCards)
playerToPickup.hand = sortCardsByRank(playerToPickup.hand)
playerToPickup.ws.send(JSON.stringify({
    action: "update_cards",
```

```

        cards: playerToPickup.hand.map(card => card.getData())
    )))

    tableCards = []
    justPlacedCards = []
    previousPlayer = null
}

[3.x]
if (previousPlayer !== null && previousPlayer.hand.length === 0) {
    // TODO: start end game sequence
    console.log(previousPlayer.name, "wins")
    return
}
}
}

```

3.1.1 - swapping to one loop

Instead of using a for loop nested inside a while loop as specified in my design I decided to use a single loop, instead tracking the current player and turn order manually. The advantages of this approach are:

- The game itself doesn't lend itself to this approach in the way that it is a continuous loop of turns, not a loop of rounds
- If one player leaves in front of the current one playing we no longer need to go over that iteration
- When someone presses cheat, we don't want to skip the current player's turn. Doing it with one loop means we can start the same player's turn again without looping through everyone.

3.1.2 - making current player clear

I wanted it to be clear whose turn it was when the game was being played. To do this, I decided to make the name of whoever was currently playing bold on the player list.

For the adding the player list to the game screen, I updated the updatePlayerList method of the Game class.

Before

```

updatePlayerList() {
    for (const player of this.players) {
        const playerList = []
        for (const player2 of this.players) {
            playerList.push({
                name: player2.name,
                isHost: player2.isHost,
                isMe: player === player2
            })
        }
        player.ws.send(JSON.stringify({
            action: "player_list_update",
            players: playerList
        }))
    }
}

```

After

```

updatePlayerList() {
    for (const player of this.players) {
        const playerList = []
        for (const player2 of this.players) {
            playerList.push({
                ...player2.getData(),
                isMe: player === player2
            })
        }
        player.ws.send(JSON.stringify({
            action: "player_list_update",
            players: playerList
        }))
    }
}

```

I changed the implementation of the function to instead get the data from a new method `getData` defined on the `Player` class, copying the system i am using on the `card` class to get the data to be sent publicly to clients. Below is that new `getData` method:

```
export class Player {
  // name: string
  // ws: Websocket
  // hand: Card[]
  // game: null or Game
  // isHost: boolean
  constructor(ws, name, isHost) {
    this.name = name
    this.ws = ws
    this.isHost = isHost
    this.hand = []
  }

  getData() {
    const data = {
      name: this.name,
      isHost: this.isHost,
    }
    if (this.game != null) {
      data.myTurn = this.game.currentPlayer == this
    }

    return data
  }
}
```

I added the `currentPlayer` attribute to the `Game` class to allow for this functionality

3.1.3 - player ids

When sending the "start_turn" action to other players, I had to make sure the current player did not receive it. I added a check to see if the player ids were the same. This did not work as expected, and logging the player ids it became clear why:

```
for (const otherPlayer of this.players) {
  console.log(player.id, otherPlayer.id)
  if (otherPlayer.id == player.id) {
    continue
  }
  otherPlayer.ws.send(JSON.stringify({
    action: "start_turn",
    your_turn: false,
    previous_rank: previous_rank
  })))
}
```

output:

```
undefined undefined
```

I had not implemented player ids. I was going to add them, but I then realised that I might not need player id. Instead, I decided to compare the references to the player objects directly, as each player will only ever have one player object.

```
for (const otherPlayer of this.players) {  
  if (otherPlayer == player) {  
    continue  
  }  
  otherPlayer.ws.send(JSON.stringify({  
    action: "start_turn",  
    your_turn: false,  
    previous_rank: previous_rank  
  })))  
}
```

I have made the decision that I will no longer give any player an id as it would add unnecessary extra complexity to my project.

3.1.4 - changes to start_turn (server -> client) event

Before (Design)	After (after development)
your_turn - boolean previous_card - Card	your_turn - boolean previous_rank - number current_rank - number can_accuse_cheater - boolean

your_turn
no changes

previous_rank

Multiple cards can be played and other information about last cards played does not need to be stored.

current_rank

In my design, I said that above the selected cards there would be text saying what the previous card placed was. Whilst adding this text, I realised that it could be confusing for players. I decided that the best option was to add both the text displaying the last card placed and the card that is now supposed to be placed as:

- The previous card text is useful for players wondering if the last player cheated
- The previous card text may not always be displayed - for example when a round is starting
- The current card text is useful for players on their turns and for anticipating what card is being placed.

can_accuse_cheater

There are two positions in the game where it would not be possible or make sense to accuse someone of cheating:

- When there was no previous player, for example at the start of the game or after someone has already been accused to cheating
- When you just had a turn you can't accuse yourself of cheating.

Instead of telling the clients who had or hadn't just played I added this new attribute to the start_game object.

3.1.5 - cleaning up start turn message

Instead of sending the start_turn message separately for the current player and all other players I merged this into one loop.

Before it looked like this:

```
const startTurnBaseMessageData = {
  action: "start_turn",
  previous_rank: previousRank,
  current_rank: currentRank,
  can_accuse_cheater: previousPlayer !== null,
}

player.ws.send(JSON.stringify({
  ...startTurnBaseMessageData,
  your_turn: true,
}))

for (const otherPlayer of this.players) {
  if (otherPlayer === player) {
    continue
  }
  otherPlayer.ws.send(JSON.stringify({
    ...startTurnBaseMessageData,
    your_turn: false,
  }))
}
```

And now it looks like this:

```
for (const otherPlayer of this.players) {
  otherPlayer.ws.send(JSON.stringify({
    action: "start_turn",
    previous_rank: previousRank,
    current_rank: currentRank,
    can_accuse_cheater: previousPlayer !== null && previousPlayer !== otherPlayer,
    your_turn: otherPlayer === player,
  }))
}
```

"your_turn" is now set individually for each player based on if it is the one who has a turn.

3.1.6 - waiting for events

In order to wait for the place cards or accuse cheaters events to arrive while remaining in the same point of the loop I decided to use async promises. When creating the promise we also await the new object we have created. Once the resolve function is called it returns back out into the main game loop. Inside the promise it creates event listeners that listen for cheat and place card events.

3.1.7 - removing duplicate code

For the check to make sure that all cards in a hand are valid I flattened it out so that the code is no longer duplicated.

Before:

```

if (allCardsValid) {
    messageSource.hand.push(...tableCards, ...justPlacedCards)
    messageSource.hand = sortCardsByRank(messageSource.hand)
    messageSource.ws.send(JSON.stringify({
        action: "update_cards",
        cards: messageSource.hand.map(card => card.getData())
    }))
} else {
    player.hand.push(...tableCards, ...justPlacedCards)
    player.hand = sortCardsByRank(player.hand)
    player.ws.send(JSON.stringify({
        action: "update_cards",
        cards: player.hand.map(card => card.getData())
    }))
}

```

After:

```

let playerToPickup = null
if (allCardsValid) {
    playerToPickup = messageSource
} else {
    playerToPickup = player
}
playerToPickup.hand.push(...tableCards, ...justPlacedCards)
playerToPickup.hand = sortCardsByRank(playerToPickup.hand)
playerToPickup.ws.send(JSON.stringify({
    action: "update_cards",
    cards: playerToPickup.hand.map(card => card.getData())
}))

```

3.2**Client**

I made the place cards and accuse cheater buttons send events to the client

```

endTurnButton.addEventListener("click", () => {
    websocket.send(JSON.stringify(
        {
            action: "place_cards",
            cards: selectedCards.map(card => card.getData())
        }
    ))
})

accuseCheaterButton.addEventListener("click", () => {
    websocket.send(JSON.stringify(
        {
            action: "accuse_cheater",
        }
    ))
})

```

I also wrote the handling for the two main events that would be required for the game to run.

Firstly I added the following to the player_list_update event to update the player list in the player screen.


```
// in game list
inGamePlayerList.innerHTML = ""
for (const player of messageData.players) {
  const element = document.createElement("li")
  let text = player.name
  if (player.isMe) {
    text = text + " (You)"
  }
  if (player.myTurn) {
    element.classList.add("myTurn")
  }
  element.innerText = text
  inGamePlayerList.appendChild(element)
}
```

And then for the start_turn event:

```
} else if (action == "start_turn") {
  let justPlacedTextContent = ""
  if (messageData.previous_rank != undefined) {
    justPlacedTextContent += "Just Placed: " + RANKS[messageData.previous_rank] + " | "
  }
  justPlacedTextContent += "Next Card: " + RANKS[messageData.current_rank]
  justPlacedText.innerText = justPlacedTextContent

  if (messageData.your_turn) {
    endTurnButton.disabled = false
    handCardsContainer.classList.remove("disabled")
    playerTurnText.innerText = "Your Turn"
  } else {
    endTurnButton.disabled = true
    handCardsContainer.classList.add("disabled")
    playerTurnText.innerText = ""
  }

  if (messageData.can_accuse_cheater) {
    accuseCheaterButton.disabled = false
  } else {
    accuseCheaterButton.disabled = true
  }
}
```

3.3

In order to display the message that appears when players have pressed the done button without placing cards I added a new text element:

```
<p id="inGameStatusText">You have to place a card to end your turn.</p>

if (selectedCards.length == 0) {
  inGameStatusText.innerHTML = "You have to place a card to end your turn."
  return
}
```

I am also going to use this element to display messages that indicate if a player was cheating or not, with the use of a new action type, "info_text_update".

Action		attributes
info_text_update	Server -> Client	message - string

```


let playerToPickup = null
if (allCardsValid) {
  playerToPickup = messageSource
  for (const somePlayer of this.players) {
    somePlayer.ws.send(JSON.stringify({
      action: "info_text_update",
      message: messageSource.name + " wrongly accused " + player.name + " of being a cheater."
    }))
  }
} else {
  playerToPickup = previousPlayer
  for (const somePlayer of this.players) {
    somePlayer.ws.send(JSON.stringify({
      action: "info_text_update",
      message: previousPlayer.name + " was a cheater!"
    }))
  }
}
}



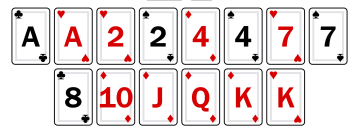

```



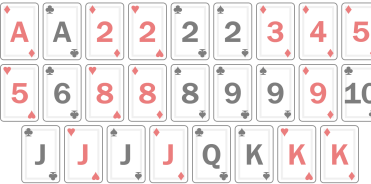
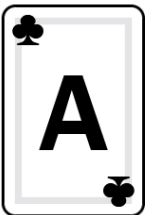
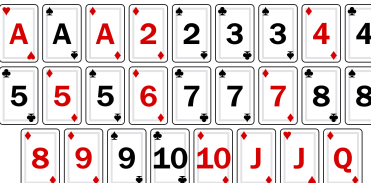
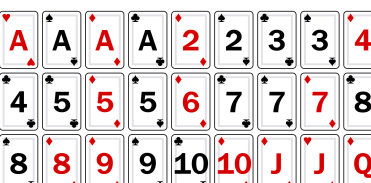
When testing this I also found a bug with the above code. Instead of the player who cheated getting the cards, the player whose turn is currently was got the cards. I fixed this by changing "playerToPickup = player" to "playerToPickup = previousPlayer"


End of Stage Testing

Below is a merged table from the two testing tables from stages 3 and 4

#	Test	Expected output	Output	Output Description
3.1	A player on their turn presses the done button	the cards in their hand are sent to the server. It is now the next player's turn and the previous player can no longer move their cards.	<p>1.</p> <p>Just Placed: 3 Next Card: 4</p>  <p>Place Cards Cheat!</p> <p>2.</p> <p>Just Placed: 4 Next Card: 5</p> <p>Place Cards Cheat!</p>	<p>1. Before pressing done button</p> <p>2. After pressing done button</p>
3.2	A player on	a message is shown	<p>1.</p>	1. Before pressing

	their turn presses the done button without placing cards	that states they need to place cards down.	<p>Just Placed: 4 Next Card: 5</p> <p>Place Cards Cheat!</p> <p>2.</p> <p>Just Placed: 4 Next Card: 5</p> <p>You have to place a card to end your turn.</p> <p>Place Cards Cheat!</p>	<p>done button</p> <p>2. After pressing done button</p>
3.3	A player presses on a card without it being their turn	nothing happens	<p>1.</p> <p>Place Cards Cheat!</p>  <p>2.</p> <p>Place Cards Cheat!</p> 	<p>1. Before pressing on a card</p> <p>2. After pressing on a card</p>
3.4	The last player ends their turn	The game loops back to the first player's turn.	<p>1.</p> <p>Your Turn</p> <p>Just Placed: 5 Next Card: 6</p> <p>6</p> <p>Place Cards Cheat!</p>  <p>2.</p> <p>Just Placed: 6 Next Card: 7</p> <p>Place Cards Cheat!</p> 	<p>1. Before ending last player's turn (it is player 3's turn indicated by their bold name)</p> <p>2. After ending last player's turn (it is player 1's turn indicated by their bold name)</p>

4.1	A player presses cheat when another player was cheating	the cards on the table go into the hand of the cheater	<p>1. Next Card: A</p>  <p>Place Cards Cheat!</p> <p>2.</p>  <p>3.</p> 	<ol style="list-style-type: none"> 1. The invalid cards played by "Player 1" 2. The hand of "Player 1" before they are accused of cheating 3. The hand of "Player 1" after being accused of cheating
4.2	A player presses cheat when the other player was not cheating	the cards on the table go into the hand of the accuser	<p>1. Next Card: A</p>  <p>Place Cards Cheat!</p> <p>2.</p>  <p>3.</p> <p>Player 2 wrongly accused Player 1 of being a cheater.</p> <p>4.</p> 	<ol style="list-style-type: none"> 1. The valid cards played by "Player 1" 2. The hand of "Player 2" before they accuse "Player 1" of cheating 3. Message sent to clients when the "Player 2" pressed the cheat button. 4. The hand of "Player 2" after they accuse "Player 1" of cheating
4.3	A player presses cheat	play continues from the one after the player challenged	<p>1.</p>	<ol style="list-style-type: none"> 1. Invalid cards played by player 1 2. Turn order after player 4 pressed

			<p>Next Card: A</p>  <p>Place Cards Cheat!</p> <p>2.</p> <p>Players:</p> <ul style="list-style-type: none"> • Player 1 • Player 2 • Player 3 • Player 4 (You) 	"cheat!"
--	--	--	---	----------

User Feedback

1. Players should be able to see how many cards are in their opponent's hands
2. There should be a way to see how many cards have been placed and how many cards are on the table
3. Needs to add a way to call cheat even if the player does not have cards so that they can't lie on their last turn.

Feedback Point 1

To begin with, I wanted the cards to fan out however I decided that it was infeasible due to the number of cards that players may have in their hand.

Using paint, I created a concept for what it might look like with a number indicating how many cards were in a hand

Players:

Player 1 (You) 12

Player 2 16

Player 3 11

I then tried it on a card background so that it was clearer that it was cards that were being counted.

Players:

Player 1 (You) 12

Player 2 16

Player 3 8

To add this, I added a new item in the player > getData function called handSize.

```
const data = {
  name: this.name,
  isHost: this.isHost,
  handSize: this.hand.length
}
```

and now on the client, I added a new element to the in-game player list when the `player_list_update` event is received.

```
const cardIndicator = document.createElement("div")
cardIndicator.classList.add("cardCount")
cardIndicator.innerHTML = player.handSize

element.appendChild(cardIndicator)
```

Instead of the background being red, I decided to make it grey so that it was more readable.

Players:

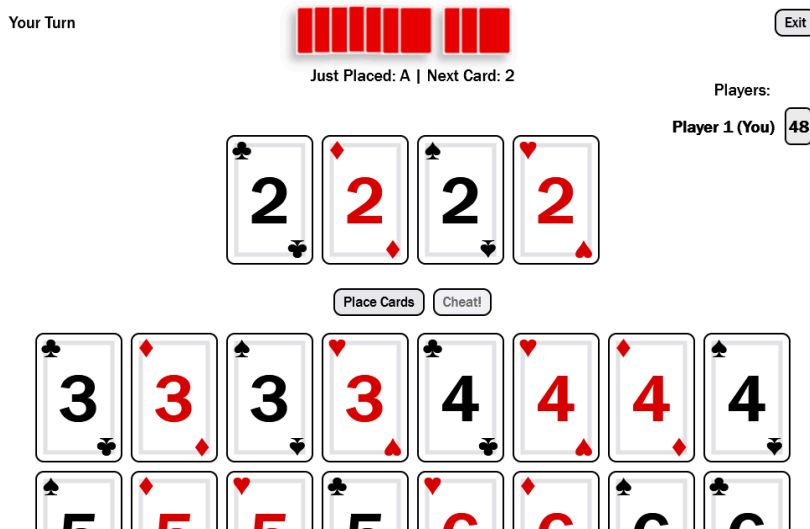
Player 1 (You) 18

Player 2 17

Player 3 17

Feedback Point 2

I again created a concept for this



I also added a new action "place_cards" which triggers after cards have been placed by some player.

Action		attributes
place_cards	Server -> Client ▾	cards_on_table - number cards_just_placed - number

On the server, I send the action after cards have been placed:

```
for (const otherPlayer of this.players) {  
  otherPlayer.ws.send(JSON.stringify({  
    action: "place_cards",  
    cards_on_table: tableCards.length,  
    cards_just_placed: justPlacedCards.length  
  })))  
}
```

And then on the client I add the appropriate number of cards to the container.

```
} else if (action == "place_cards") {  
  tableCardsPreview.innerHTML = ""  
  justPlacedCardPreview.innerHTML = ""  
  for (let i = 0; i < messageData.cards_on_table; i++) {  
    const card = document.createElement("div")  
    card.classList.add("placedCard")  
    tableCardsPreview.appendChild(card)  
  }  
  for (let i = 0; i < messageData.cards_just_placed; i++) {  
    const card = document.createElement("div")  
    card.classList.add("placedCard")  
    justPlacedCardPreview.appendChild(card)  
  }  
}
```

which when implemented looked like this:



Just Placed: 9 | Next Card: 10

Place Cards

Cheat!

Feedback Point 3

I will be deferring the response to this feedback point until the next stage so I can get an idea of how the game should play and as I am focusing on the end-of-game part of the project.

Stage 5

At the end of this stage I aim to have close to a feature complete version of my game. I will add the end of game logic and ensure that players that leave are handled gracefully, preventing errors and allowing the game to continue.

Development

5.1 - end game event

To begin with, I needed a way to indicate to clients that the game had ended. To do this I created a new action type:

Action		attributes
end_game	Server -> Client ▾	game_over_text - string can_play_again - boolean

The game over text will say who won e.g. "Player 1 won the game."

```
} else if (action == "end_game") {  
    endScreen.classList.remove("hidden")  
    playScreen.classList.add("hidden")  
  
    gameOverText.innerText = messageData.game_over_text  
  
<div id="endScreen" class="hidden">  
    <h2>Game Over</h2>  
    <p id="gameOverText"></p>  
  
    <div class="buttonRow">  
        <button class="hidden" id="playAgain">Play Again</button>  
        <button id="endScreenExit">Exit</button>  
    </div>  
</div>
```

Game Over

Player 1 has won the game

Play Again

Exit

5.2 - exiting games

To implement the exit buttons, I began by making the exit game buttons send the `leave_game` action as described in my design.

```
waitingScreenExit.addEventListener("click", () => {
    websocket.send(JSON.stringify(
        {
            action: "leave_game",
        }
    ))
})
playScreenExit.addEventListener("click", () => {
    websocket.send(JSON.stringify(
        {
            action: "leave_game",
        }
    ))
})
endScreenExit.addEventListener("click", () => {
    websocket.send(JSON.stringify(
        {
            action: "leave_game",
        }
    ))
})
```

I also moved the code that hid all the screens and showed the start screen to trigger when a new action was triggered from the server "leave_game"

Action		Description
leave_game	Server -> Client	Sent from a server to the client when the player is to leave a game

```
} else if (action == "leave_game") {
    joinGameScreen.classList.remove("hidden")
    waitingScreen.classList.add("hidden")
    playScreen.classList.add("hidden")
    endScreen.classList.add("hidden")
} else {
```

Below: the server "leave_game" handler

```
} else if (action === "leave_game") {  
  const game = player?.game  
  if (game) {  
    game.removePlayer(player)  
    player = null  
  }  
}
```

5.3 - play again button

Now for the play again button, I decided that the button would only appear for the host player. This was because if other players pressed it before the host did then they would be in a game that they couldn't start.

Action	
play_again	Client -> Server ▾

```
if (player.isHost) {  
  player.ws.addEventListener("message", (message) => {  
    const possibleMessageData = JSON.parse(message.data)  
    const action = possibleMessageData.action  
  
    if (action == "play_again") {  
      for (const player of this.players) {  
        this.removePlayer(player)  
      }  
      this.addPlayer(player)  
      player.ws.send(JSON.stringify({  
        action: "become_host"  
      })))  
    }  
  })  
}
```

And then on the client, I toggle the visibility of the play again button based on the attribute set on the "end_game" event

```
if (messageData.can_play_again) {  
  playAgainButton.classList.remove("hidden")  
} else {  
  playAgainButton.classList.add("hidden")  
}
```

5.4 - event listener warning

I noticed that I was getting these warnings in my project.

(node:19572) MaxListenersExceededWarning: Possible EventEmitter memory leak detected. 11 message listeners added to [WebSocket]. MaxListeners is 10. Use emitter.setMaxListeners() to increase limit

```
for (const player of this.players) {
  const accuseCheaterListener = (message) => {
    const possibleMessageData = JSON.parse(message.data)
    const action = possibleMessageData.action
    if (action == "accuse_cheater") {
      resolve()
      messageData = possibleMessageData
      messageSource = player
    }
  }
  player.ws.addEventListener("message", accuseCheaterListener)
}
```

To fix this, I added a new variable "listenersToRemove" which I push references to event listeners and their source websockets. After the listeners have returned, I remove them all from their sources. This fixed the issue.

```
// list of arrays containing two elements: a websocket and an event listener to be removed e.g. [[websocket, listener]]
const listenersToRemove = []

// wait for either:
// - place_cards event from current player
// - accuse_cheater event from any other player
await new Promise((resolve) => {
  const placeCardsListener = (message) => {
    const possibleMessageData = JSON.parse(message.data)
    const action = possibleMessageData.action
    if (action == "place_cards") {
      player.ws.removeEventListener("message", placeCardsListener)
      resolve()
      messageData = possibleMessageData
      messageSource = player
    }
  }
  player.ws.addEventListener("message", placeCardsListener)
  listenersToRemove.push([player.ws, placeCardsListener])

  for (const player of this.players) {
    const accuseCheaterListener = (message) => {
      const possibleMessageData = JSON.parse(message.data)
      const action = possibleMessageData.action
      if (action == "accuse_cheater") {
        resolve()
        messageData = possibleMessageData
        messageSource = player
      }
    }
    player.ws.addEventListener("message", accuseCheaterListener)
    listenersToRemove.push([player.ws, accuseCheaterListener])
  }
})

for (const [websocket, listener] of listenersToRemove) {
  websocket.removeEventListener("message", listener)
}
```

Console output after playing a round:

```
Server started.
```

5.5 - restricting gameplay to valid numbers of players

Firstly, I added a new element on the waiting screen to display a new error message.

```
<p id="waitingScreenErrorText" class="hidden"></p>
```

I created a new action which triggers this text to show:

Action		attributes
waiting_screen_error	Server -> Client ▾	message - string

Now, when the start_game event is ran, the numbers of players in the room are checked:

```
if (game.players.length < 3) {  
    websocket.send(JSON.stringify({  
        action: "waiting_screen_error",  
        message: "Three or more people required to start the game."  
    })))  
    return  
}
```

and an error message is sent to the host.

Room Code: afdoue

Players:

- Player 1 (You) - Host

Start Game

Exit

Three or more people required to start the game.

5.6 - end of stage 4 feedback point 3

I now came back to the third feedback point given by my target market in the previous stage.

"Needs to add a way to call cheat even if the player does not have cards so that they can't lie on their last turn."

To complete this point, I moved the code that checked if the previous player has won into the if statement for the "place_cards" event, before the previousPlayer variable gets set to the currentPlayer and the game loop continues.

```
// checks if the previous player has now won, end of game logic
if (previousPlayer != null && previousPlayer.hand.length == 0) {
  for (const player of this.players) {
    player.ws.send(JSON.stringify({
      action: "end_game",
      game_over_text: previousPlayer.name + " won the game.",
      can_play_again: player.isHost
    })))
  }

  if (player.isHost) {
    const playAgainListener = (message) => {
      const possibleMessageData = JSON.parse(message.data)
      const action = possibleMessageData.action

      if (action == "play_again") {
        for (const player of this.players) {
          this.removePlayer(player)
        }
        this.addPlayer(player)
        player.ws.send(JSON.stringify({
          action: "become_host"
        })))
        player.ws.removeEventListener("message", playAgainListener)
      }
    }
    player.ws.addEventListener("message", playAgainListener)
  }
}

return
}
```

previousPlayer = player

5.7 - Handling players leaving mid-game

For players who left the game, I found these scenarios where some action would need to be taken:

- When any player leaves:
 - remove player from player list
 - move cards from the hand of the player into the discard pile
- When a player leaves on their turn.
Solution:
 - skip to next iteration of the while loop (playerIndex although the same value will now reference the next player in the array)
 - I also needed to move the playerIndex check to the start of the while loop as it was now possible that if the last player in the players array leaves then playerIndex will now be outside of the bounds of the array
- When the previous player leaves and cheat is called on them
 - set previousPlayer to null when they leave
- When a player before the current one in the turn order leaves
 - subtract one from playerIndex
 - Also add a negative player index check to the while loop
- When a player after the current one in the turn order leaves
 - no extra logic is required here
- If a player leaves and there are only two people left in the game
 - end the game with the message "Not enough players to continue"

For the first point, I needed to add a new pathway for the branching on each game turn:

```
const action = messageData.action
if (action == "place_cards") {...
} else if (action == "accuse_cheater") {...
}
```

In the promise preceding this if statement, I had to add two event listeners. One for the "leave_game" action and the other for the websocket close event. I had a concern about the order of event listeners: if the order was random then I would have to design the program to expect either one first. To check this I logged a message in the console and checked which one triggered first. Each time I tested it the listener in the main.js was triggered first.

```
main close listener
gameLoop close listener
```

I renamed the placeCardsListener function to currentPlayerListener and the accuseCheaterListener to inGameActionListener, as it now contains both "accuse_cheater" and "leave_game" events.

I changed the action variable's name to messageAction and this is now assigned within the event listeners. I did this to make both the "leave_game" and websocket close events lead to the same result.

```
const currentPlayerListener = (message) => {
  const possibleMessageData = JSON.parse(message.data)
  const action = possibleMessageData.action
  if (action == "place_cards") {
    player.ws.removeEventListener("message", currentPlayerListener)
    messageData = possibleMessageData
    messageSource = player
    messageAction = "place_cards"
    resolve()
  }
}
```

```
if (action == "place_cards") {
  if (messageAction == "place_cards") {
```

As I was now adding a new event listener type, I had to restructure the listenersToRemove variable. Instead of a list of arrays [websocket, listener] I now made it [websocket, eventName, listener] where eventName is either "close" or "message".

```
const websocketCloseListener = () => {
  messageSource = player
  messageAction = "leave_game"
  resolve()
}
player.ws.addEventListener("close", websocketCloseListener)
listenersToRemove.push([player.ws, "close", websocketCloseListener])
```

```
for (const [websocket, eventType, listener] of listenersToRemove) {  
  websocket.removeEventListener(eventType, listener)  
}
```

I moved the out of bounds player index check to the beginning of the game loop.

```
while (true) {  
  if (playerIndex >= this.players.length) {  
    playerIndex = 0  
  }  
  
  const player = this.players[playerIndex]
```

Here is the code that runs after a "leave_game" action or websocket close event is received:

```
} else if (messageAction == "leave_game") {  
  if (this.players.length < 3) {  
    this.endGame("Not enough players to continue.")  
    return  
  }  
  tableCards.push(...messageSource.hand)  
  
  if (messageSource == previousPlayer) {  
    previousPlayer = null  
  } else {  
    const currentPlayerIndex = this.players.indexOf(currentPlayer)  
    if (currentPlayerIndex != -1) {  
      playerIndex = currentPlayerIndex  
    }  
  }  
  
  for (const player of this.players) {  
    player.ws.send(JSON.stringify({  
      action: "place_cards",  
      cards_on_table: tableCards.length,  
      cards_just_placed: justPlacedCards.length  
    })))  
  }  
  
  for (const player of this.players) {  
    player.ws.send(JSON.stringify({  
      action: "info_text_update",  
      message: messageSource.name + " left the game."  
    })))  
  }  
}
```

Instead of subtracting one from the current index if the player that left was before the current one, I check the index of the current player. The benefit of this method is that I firstly do not need to implement any new variable to track the index of the players that trigger events and also if multiple players left, the index could reference a player that doesn't exist.

5.8 - Joining ongoing games

I also realised that players could join the game mid way through. To prevent this I needed to add a way for the main.js event listener to check if a game was ongoing or not. To do this, I added a new attribute to the game class:


```
export class Game {  
  constructor(gameCode) {  
    this.code = gameCode  
    this.players = []  
    this.ongoing = false  
  }  
}
```

I then set this to true in the startGame method:

```
startGame() {  
  this.ongoing = true  
}
```

Now in the main.js start_game action handler, if the game is found to be ongoing then an error is sent to the client and the handler returns:

```
const game = games.get(code)  
if (game.ongoing) {  
  websocket.send(JSON.stringify(  
    {  
      action: "join_game_error",  
      message: "The game with the provided code is ongoing."  
    }  
  ))  
  return  
}
```

Cheat

Join a Game	Create a Game
<input type="text" value="Player 4"/>	<input type="text" value="Username"/>
<input type="text" value="nywvfb"/>	
<input type="button" value="Join Room"/>	<input type="button" value="Create Room"/>

The game with the provided code has already started.

And now when the game ends, the ongoing flag is set to false.

```
endGame(gameOverText) {  
  this.ongoing = false  
}
```

5.9 - Ensuring request validity



To make sure no malicious players could send a cheat action when they were not supposed to be able to, I added a check for previousPlayer being null or the previousPlayer being the one sending the request.


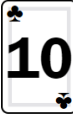
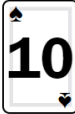



```

const inGameActionListener = (message) => {
  const possibleMessageData = JSON.parse(message.data)
  const action = possibleMessageData.action
  if (action == "accuse_cheater") {
    if (previousPlayer == null || previousPlayer == player) {
      return
    }
    messageData = possibleMessageData
    messageSource = player
    messageAction = "accuse_cheater"
    resolve()
  } else if (action == "leave_game") {
    messageData = possibleMessageData
    messageSource = player
    messageAction = "leave_game"
    resolve()
  }
}

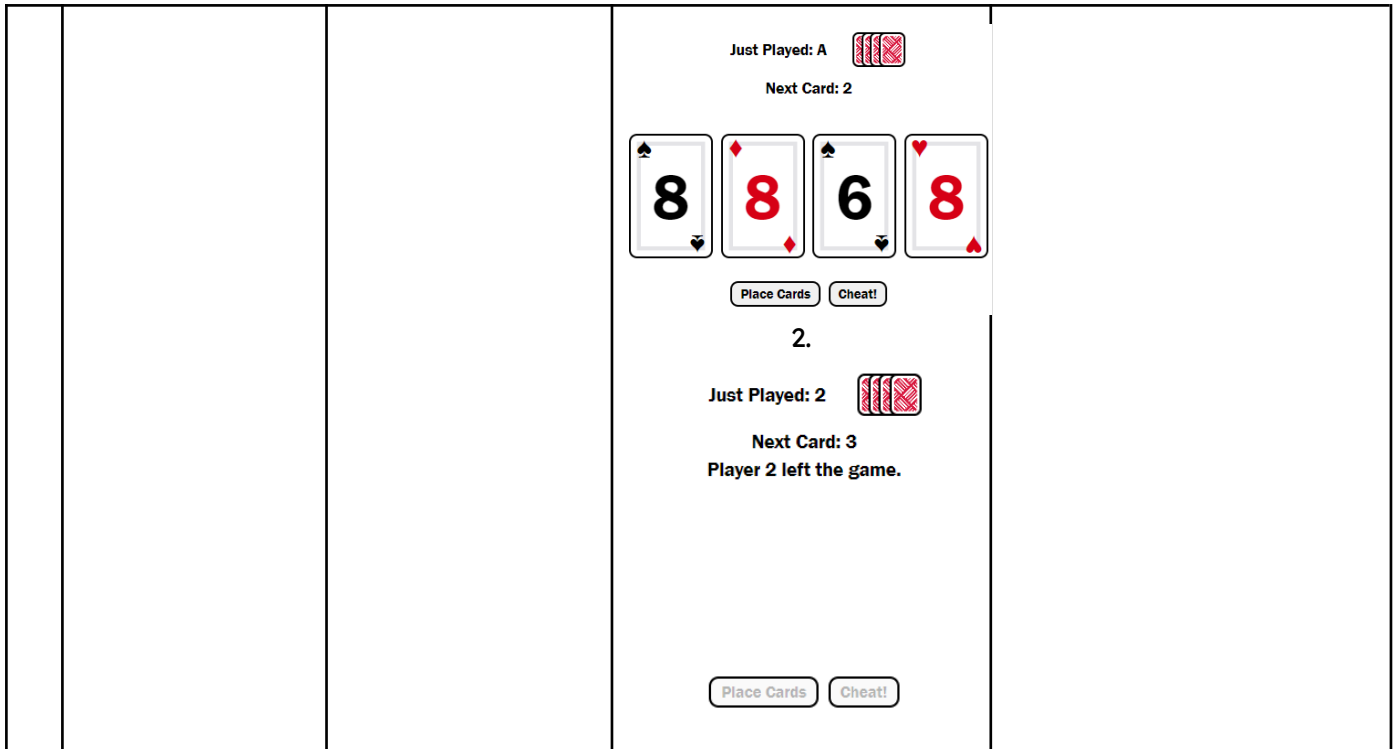
```

End of Stage Testing

#	Test	Expected output	Output	Output Description
1	A player places the last cards in their hand and the following player takes their turn	The player that placed the last cards down wins, show game over screen	<p>1.</p> <p>Just Played: A </p> <p>Next Card: 2</p>  <p><input type="button" value="Place Cards"/> <input data-bbox="949 1473 1018 1505" type="button" value="Cheat!"/></p> <p>2.</p> <p>Players:</p> <p>Player 1 (You) 16</p> <p>Player 2 25</p> <p>Player 3 0</p> <p>3.</p>	<p>1. Player 3 places their last card</p> <p>2. List of players after player 3's turn</p> <p>3. Player 1 places some cards</p> <p>4. The game over screen is shown</p>

			<div> <p>Just Played: 2 </p> <p>Next Card: 3</p> <div>     </div> <div> Place Cards Cheat! </div> <p>4.</p> <p>Game Over</p> <p>Player 3 won the game.</p> <div> Play Again Exit </div> </div>	
2	A player leaves the game	The cards from their hand are moved to the discard pile	<div> <p>1.</p> <p>Players:</p> <div> Player 1 (You) 13 </div> <div> Player 2 13 </div> <div> Player 3 13 </div> <div> Player 4 13 </div> <p>2.</p> <p>Players:</p> <div> Player 1 (You) 13 </div> <div> Player 3 13 </div> <div> Player 4 13 </div> <p>3.</p> <div>  </div> <p>Your Turn</p> </div>	<ol style="list-style-type: none"> The player list before player 2 leaves the game The player list after player 2 leaves the game The discard pile after player 2 leaves the game
3	A player leaves the game, bringing the total number of players below 3	The game ends. Show the game over screen.	<ol style="list-style-type: none"> 	<ol style="list-style-type: none"> The player list before player 3 leaves the game The game over screen shown after player 3 leaves the game

			<p>Players:</p> <p>Player 1 (You) 18</p> <p>Player 2 17</p> <p>Player 3 17</p> <p>2.</p> <p>Game Over</p> <p>Not enough players to continue.</p> <p>Exit</p>	
4	A player leaves the game on their turn	The next player's turn begins	<p>1.</p> <p>Players:</p> <p>Player 1 (You) 10</p> <p>Player 2 13</p> <p>Player 3 13</p> <p>Player 4 13</p> <p>2.</p> <p>Players:</p> <p>Player 1 (You) 10</p> <p>Player 3 13</p> <p>Player 4 13</p>	<p>1. The player list before player 2 leaves the game - it is player 2's turn.</p> <p>2. The player list after player 2 leaves the game - it is now player 3's turn</p>
5	A player leaves the game after placing cards	cheat can not be called on them	<p>1.</p>	<p>1. Player 2 places 4 incorrect cards and then leaves</p> <p>2. Player 1's screen after player 2 leaves. The cheat button is disabled.</p>



User Feedback

Feedback Point 1

Names should have a maximum length.

I previously had a check to make sure that the username was at least one character. After testing with my stakeholders I found that users could make their usernames very long and this would cover up the whole screen. To fix this issue I created a maximum number of characters that usernames could be. I settled at 16 characters as anything past this looked very long.

Along with adding a new validity check, I rewrote the previous check's error message to make it more descriptive.

This is the check before:

```
const nameValid = name.length !== 0
if (!nameValid) {
  websocket.send(JSON.stringify(
    {
      action: "join_game_error",
      message: "The provided username is invalid."
    }
  ))
  return
}
```

And this is the check after:

```
if (name.length === 0) {  
    websocket.send(JSON.stringify(  
        {  
            action: "join_game_error",  
            message: "The provided username is too short."  
        }  
    ))  
    return  
}  
  
if (name.length > 16) {  
    websocket.send(JSON.stringify(  
        {  
            action: "join_game_error",  
            message: "The provided username is too long."  
        }  
    ))  
    return  
}
```

Play Test 1

I completed a play test with my stakeholders.

From the experience, I noted down the following points of feedback and observations:

1. On some devices the player list was always in bold:

Players:

Player 1 (You)	18
Player 2	17
Player 3	17

My solution for this was to make the text a lighter colour for the players who did not have a turn.

Players:

Player 1 (You)	18
Player 2	17
Player 3	17

2. The cards at the top of the screen should disappear when someone has picked up cards from cheating.

To fix this I sent a "place_cards" event when the cheater has been identified

```
for (const player of this.players) {  
  player.ws.send(JSON.stringify({  
    action: "place_cards",  
    cards_on_table: 0,  
    cards_just_placed: 0  
  })))  
}
```

3. When a player has placed all of their cards and another player presses cheat, the game does not check if the last player has won.

I worked out that the reason for this was that previousPlayer gets assigned to null if a cheater is called:

```
tableCards = []  
justPlacedCards = []  
previousPlayer = null
```

To fix this I could have created another variable and checked this instead of checking previousPlayer. Instead I decided to extract the end of game logic into its own method.

```
// checks if the previous player has now won, end of game logic
if (previousPlayer != null && previousPlayer.hand.length == 0) {
  ...for (const player of this.players) {
  ...  player.ws.send(JSON.stringify({
  ...    action: "end_game",
  ...    game_over_text: previousPlayer.name + " won the game.",
  ...    can_play_again: player.isHost
  ...  }))
  ...
  ...if (player.isHost) {
  ...  const playAgainListener = (message) => {
  ...    const possibleMessageData = JSON.parse(message.data)
  ...    const action = possibleMessageData.action
  ...
  ...    if (action == "play_again") {
  ...      for (const player of this.players) {
  ...        this.removePlayer(player)
  ...      }
  ...      this.addPlayer(player)
  ...      player.ws.send(JSON.stringify({
  ...        action: "become_host"
  ...      }))
  ...      player.ws.removeEventListener("message", playAgainListener)
  ...    }
  ...    player.ws.addEventListener("message", playAgainListener)
  ...  }
  ...}
  ...}
  ...return
}
```

Extracted endGame method

```
endGame(gameOverText) {
  for (const player of this.players) {
    player.ws.send(JSON.stringify({
      action: "end_game",
      game_over_text: gameOverText,
      can_play_again: player.isHost
    }))
  }

  if (player.isHost) {
    const playAgainListener = (message) => {
      const possibleMessageData = JSON.parse(message.data)
      const action = possibleMessageData.action

      if (action == "play_again") {
        for (const player of this.players) {
          this.removePlayer(player)
        }
        this.addPlayer(player)
        player.ws.send(JSON.stringify({
          action: "become_host"
        }))
        player.ws.removeEventListener("message", playAgainListener)
      }
    }
    player.ws.addEventListener("message", playAgainListener)
  }
}

}
```

which can now be called from both the end of turn and cheat actions.

End of game action:

```
// checks if the previous player has now won
if (previousPlayer != null && previousPlayer.hand.length == 0) {
  this.endGame(previousPlayer.name + " won the game.")
  return
}
```

Cheat action:


```
playerToPickup.hand = sortCardsByRank(playerToPickup.hand)
playerToPickup.ws.send(JSON.stringify({
  action: "update_cards",
  cards: playerToPickup.hand.map(card => card.getData())
}))

for (const player of this.players) {
  player.ws.send(JSON.stringify({
    action: "place_cards",
    cards_on_table: 0,
    cards_just_placed: 0
  }))
}

if (previousPlayer.hand.length == 0) {
  this.endGame(previousPlayer.name + " won the game.")
  return
}

tableCards = []
justPlacedCards = []
previousPlayer = null
}
```

4. The UI with cards have been placed is confusing. See below: The card that have been placed are not next to the text that says what has been placed



To fix this I again created a concept in paint to see what it might look like if I rearranged the UI elements.



This version is better as the "Just Placed" text is now next to the card indicator. The cards already placed are moved off to the left of the screen and the "Your Turn" text has been moved downwards to accommodate this.

```
const justPlacedText = document.querySelector("#justPlacedText")
const placeNextText = document.querySelector("#placeNextText")
```

I separated the just placed and place next texts out into their own elements:

```
} else if (action == "start_turn") {
  if (messageData.previous_rank !== null) {
    justPlacedText.innerHTML = "Just Played: " + RANKS[messageData.previous_rank]
  } else {
    justPlacedText.innerHTML = ""
  }
  placeNextText.innerHTML = "Next Card: " + RANKS[messageData.current_rank]
```

And rearranged the dom elements to closer represent the layout, making css styling easier

```
<div id="playScreen" class="hidden">
  <div class="topLeft">
    <div id="tableCardsPreview"></div>
    <p id="playerTurnText">Your Turn</p>
  </div>
  <div class="topBar">
    <div class="cardPreview">
      <p id="justPlacedText"></p>
      <div id="justPlacedCardPreview"></div>
    </div>
  </div>
  <div class="topRight">
    <button id="playScreenExit">Exit</button>
  </div>
  <div class="inGamePlayerListContainer">
    <p>Players:</p>
    <ul id="inGamePlayerList">
    </ul>
  </div>

  <p id="placeNextText">Place Next: X</p>
  <p id="inGameStatusText"></p>
  <div id="selectedCards"></div>
  <div class="buttonRow">
    <button id="endTurn">Place Cards</button>
    <button id="cheat">Cheat!</button>
  </div>
  <div id="handCards"></div>
</div>
```

After implementing the above changes, the game UI now looks like the following:



Play Test 2

I conducted one last play test and set of changes before finishing the project.

1. The "Just Played" text does not disappear when someone has pressed cheat

Just Played: A

Next Card: 2

Player 2 wrongly accused Player 3 of being a cheater.

Just Played: 7

Next Card: 8

Player 3 was a cheater!

I fixed this by setting the previous rank to null when someone presses it

```
for (const player of this.players) {  
  player.ws.send(JSON.stringify({  
    action: "place_cards",  
    cards_on_table: 0,  
    cards_just_placed: 0  
  })))  
}  
  
if (previousPlayer.hand.length == 0) {  
  this.endGame(previousPlayer.name + " won the game.")  
  return  
}  
  
tableCards = []  
justPlacedCards = []  
previousPlayer = null  
previousRank = null
```

2. When a new game happens, the status text is not reset

Next Card: A

Player 3 wrongly accused Player 1 of being a cheater.

I fixed this by setting the content of this stats text to an empty string when the start_game event was sent.

```
} else if (action == "start_game") {  
  waitingScreen.classList.add("hidden")  
  playScreen.classList.remove("hidden")  
  inGameStatusText.innerHTML = ""  
}
```

Evaluation

Reviews

I got three reviews. Two from people that were familiar with the game and one that wasn't.

1. Person One

Is the game enjoyable?

yes, very fun to play against your friends and try and make them look like fools

What are the positive aspects of the game?

the icons for the cards, the speed of the processing time, the fact that names can be unique

What are the parts of the game that could be improved?

knowing what card has been place and what card is next makes it a bit confusing sometimes for some people

Are the UIs easy to navigate?

yes apart from knowing the cards number that have been placed and what is next can be a bit confusing

Is the game aesthetically pleasing?

The card designs are sleek and pleasing to the eyes, how the cards are arranged into groups of the same number value helps.

2. Person Two

Is the game enjoyable?

Yes, the game is quite enjoyable as it is a very easy concept to understand

What are the positive aspects of the game?

One positive aspect of the game is the card design, it is very easy to tell what card is what. Another positive aspect of the game is that you are able to tell who's turn it is by highlighting the name of the person and how many cards each person has.

What are the parts of the game that could be improved?

One area that could be improved is that you should be able to place cards -1 to the card that has just been played because if you could only go up it is obvious what cards have been played and easy to who is lying later on in the game.

Are the UIs easy to navigate?

The UI is easy to navigate as it is very simple.

Is the game aesthetically pleasing?

The game is somewhat aesthetically pleasing with how simple the design is. However, some would argue that the game is too simplistic and lacks character with how simplistic it is, with it being on a plain white background and the only bit of colour coming from the cards.

3. Person Three

Is the game enjoyable?

Yes however the game may also require a side application such as discord or xbox so that players may communicate with one another.

What are the positive aspects of the game?

The UI is very intuitive making the game easy to navigate and to learn if the user is new. The game functions as expected without errors

What are the parts of the game that could be improved?

Implement a copy and paste button for the room code to make it easier to send the code to other players. Add sounds to the card movements, winning sounds, a sound for when somebody is found to have cheated and relaxing music for the main menu and in game which can be disabled in a settings menu. As a new player I suggest a tutorial level be implemented in order to visually show the users how to play the game

Are the UIs easy to navigate?

Yes very intuitive

Is the game aesthetically pleasing?

No, the lack of colour and detail in the background give a monotonous tone, by effectuating a more colourful background and menu, this may create a more inviting and exciting game for people to play

Video Showcase

In order to gather evidence for my evaluation, I recorded some video footage. These videos are available in both my project file within the videos directory and on youtube, where the link has been below:

Video Title	video path youtube link	Explanation
4 Players Side-By-Side	"4-players-side-by-side.mp4"	In this video I played an example round of my game.
Integration Testing	"integration-testing.mp4"	In this video I performed each one of my tests.

Implementation of features

In this section I went through every feature stated in my analysis, checking that it is present in the final game and evaluating its implementation. For this, I used the footage recorded in my video showcase titled "4 Players Side-By-Side"

Feature	Completed & Timestamp	Evaluation
A start screen with options to join or create a game	Completed ▾ 0:00-0:15	This is the default screen and appears when the webpage is loaded.
The ability for players to	Completed ▾	I have added this, however when

change their display name on the start screen	0:03-0:07	players are in the waiting screen for the game it would be useful if they could also change it here
Game rooms are assigned a code by which players can join rooms by entering the code on the start screen.	Completed ▾ 0:07-0:14	The game code is displayed in the waiting room. To make it easier for the code to be shared, it could be copied to the clipboard when a player clicks on it.
A "waiting" ui screen	Completed ▾ 0:02-0:14	Fully implemented.
A list of players on the waiting screen	Completed ▾ 0:02-0:14	The player list works correctly according to the design. If I were to continue development I would want to make the indicators for "me"/"host" clearer and separated from the player name as well as allowing the host player to kick people from the game.
A start button	Completed ▾ 0:02-0:14	The start button has been added and is only displayed for the host player. To make it clearer for other players I could have made it appear but be greyed out, with a message on the screen that appears
An exit button visible on both the waiting and in-game screens	Completed ▾ 0:02-1:15	The exit button is shown on both screens. The only potential improvement I could make is that maybe a pop-up should appear to ensure the player wants to leave and it is not accidentally clicked.
Before the game starts, the cards are placed in a random order and split evenly between players, one by one in the order that players join.	Completed ▾ 0:16	This has been implemented. If I were to allow the turn order to start from anyone then I might need to change the first player that the cards are given to.
After someone presses "Cheat", the game begins again from the person after the one who was challenged.	Completed ▾ 0:26, 0:31, 0:48, 1:03	Fully completed. No improvements to be made.
During the game, players are able to see how many cards their opponents have, as well as whose turn it is.	Completed ▾ 0:16	Fully completed. There is a player list on the side of the screen that includes card counters. To improve this, I could make it more clear when it was someone's turn.
At the end of a player's turn, they select what cards they want to play and the correct	Completed ▾ 0:20, 0:28, ...	I have added this. Given more time I would have liked to create more concepts for this part of the UI in

value for these cards are shown to everyone.		order to make it clearer that X amount of Y card have been placed.
On a turn, players can move their cards between their hand and the discard pile	Completed ▾ 0:17	In the current version of my project, cards are moved between the hand and discard pile by dragging. If I were to improve this, I would want to make the cards draggable between the deck and hand. Currently I do not know how I would implement this.
At any point in the game, players can press the "Cheat" button, and if the previous player was lying, they pick up the cards. Otherwise, the player who pressed the button picks up the cards.	Completed ▾ 0:23, 0:30, ...	This has been added in full. If I were to improve this, I would want to make it clearer who was cheating and who was telling the truth, for example by highlighting the player names red or green with texts "cheating" or "not cheating" depending on the state of the game.
When a game ends, the winner is displayed and players have the option to start another round.	Partially Compl... ▾ 1:16	The winner of the game is displayed. Only the host player has the option to start another round as it would have been too complicated given the time I had to wait for non-hosts to join the game before the host player.
If a player leaves the game mid way through, their cards are placed on the discard pile.	Completed ▾ (within integration testing)	This has been added. The only improvement to this I can think of could be that it keeps the name in the player list but the text becomes faint.

Integration Testing

I compiled every test from each previous section and tested them all together to make sure it all worked in the final state of the project. The timestamps refer to the time of the test in the "Integration Testing" video.



#	Test	Expected output	Completed & Timestamp
1.1	A valid username is entered and the create room button is pressed	the screen changes to the game waiting screen and a room code is displayed on the screen	Completed ▾ 0:00
1.2	A valid username and valid room code is entered and the user presses join game	They are able to join the game with no problems. the screen changes to the game waiting screen	Completed ▾ 0:04

1.3	An invalid room code that does not exist is entered and the user presses join	An error is displayed stating the room does not exist	Completed ▾ 0:10
1.4	An invalid username is entered with a valid room code	when the user tries to join a game an error is displayed saying the username is invalid	Completed ▾ 0:18
1.5	When a new player joins the game, the player list updates	the player list updates to display the player that has joined	Completed ▾ 0:23
1.6	A player leaves the game	They are removed from the player list	Completed ▾ 0:30
1.7	The host player leaves the game	A new player is chosen to be the host	Completed ▾ 0:35
2.1	The host player presses the start game button	The waiting screen is hidden for all players and the game screen is now shown for all of them.	Completed ▾ 0:40
2.2	A player presses on one of the cards in their hand	The card is moved to the centre of their screen	Completed ▾ 0:44
2.3	A player presses on one of the cards previously selected	It returns to their hand	Completed ▾ 0:46
2.4	A player presses on one of their cards when there are four already on the table	Nothing happens.	Completed ▾ 0:49
3.1	A player on their turn presses the done button	the cards in their hand are sent to the server. It is now the next player's turn and the previous player can no longer move their cards.	Completed ▾ 0:53
3.2	A player on their turn presses the done button without placing cards	a message is shown that states they need to place cards down.	Completed ▾ 0:57
3.3	A player presses on a card without it being their turn	nothing happens	Completed ▾ 1:00
3.4	The last player ends their turn	The game loops back to the first player's turn.	Completed ▾ 1:07
4.1	A player presses cheat when another player was cheating	the cards on the table go into the hand of the cheater	Completed ▾ 1:11
4.2	A player presses cheat when the other player was not cheating	the cards on the table go into the hand of the accuser	Completed ▾ 1:14

4.3	A player presses cheat	play continues from the one after the player challenged	Completed ▾ 1:14
5.1	A player places the last cards in their hand and the following player takes their turn	The player that placed the last cards down wins, show game over screen	Completed ▾ 1:21
5.2	A player leaves the game	The cards from their hand are moved to the discard pile	Completed ▾ 1:30
5.3	A player leaves the game, bringing the total number of players below 3	The game ends. Show the game over screen.	Completed ▾ 1:36
5.4	A player leaves the game on their turn	The next player's turn begins	Completed ▾ 1:36
5.5	A player leaves the game after placing cards	cheat can not be called on them	Completed ▾ 1:42

Meeting Success Criteria

#	Criteria	Test	Evaluation
1	The game includes all of the features specified in the project analysis.	All aspects of the feature list are completed.	Partially Completed ▾ All of the features in the feature list have been implemented to some extent, with the only partially completed feature being the play again button only showing for the host player.
2	The game is able to handle multiple games running simultaneously.	Two games can run simultaneously (Given access to a wider audience of testers I would have wished to run more extensive tests than this - see more in my limitations section above)	Completed ▾ See test below.
3	The game UIs are easy to navigate.	Two-thirds of testers say that they find the UIs to be easy to use	Completed ▾ All three of my testers found the UIs easy to use
4	Players should be able to leave games at any point without breaking gameplay.	For each decision point, have a player leave the game at this point.	Completed ▾ All tests relating to player leaves passed correctly (stage 5 tests)
5	No information critical to gameplay (e.g. the	Check every event that is sent to the client during	Completed ▾

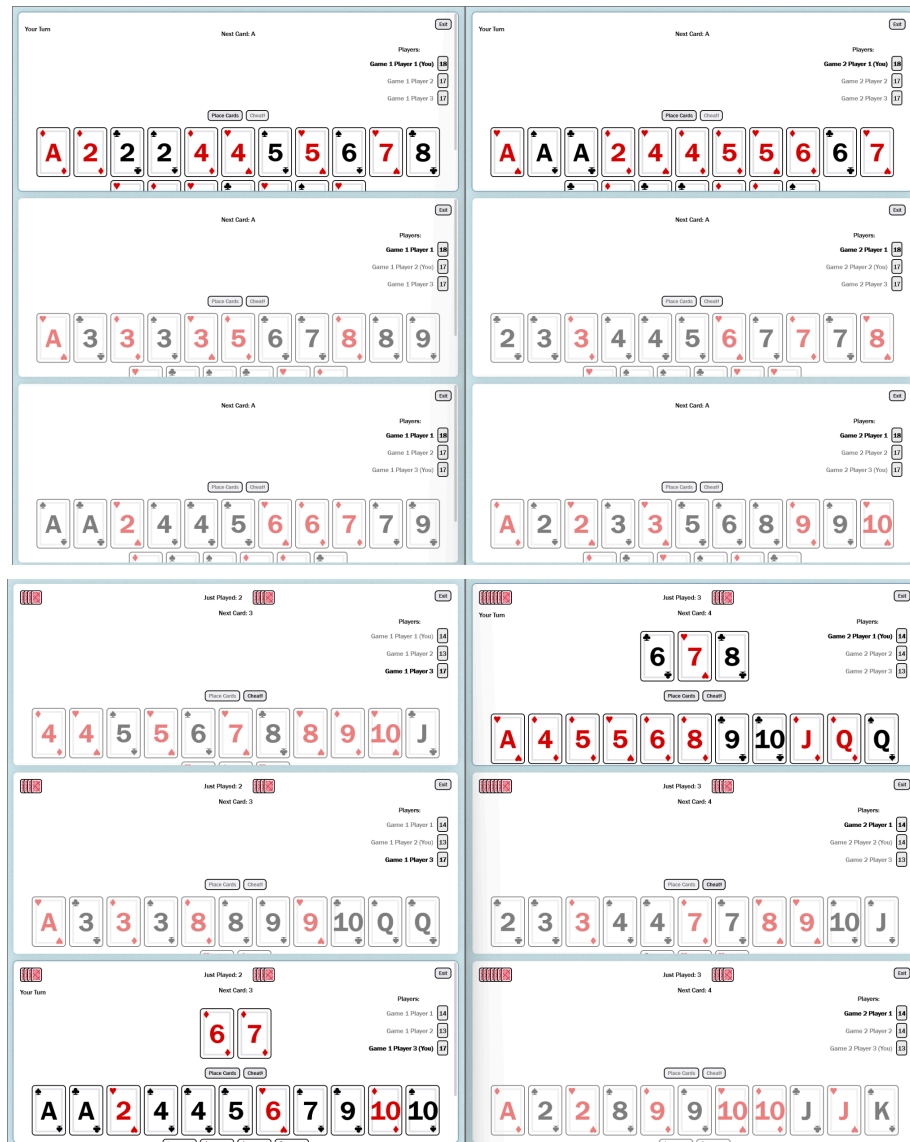
	cards in opponents hands) should be accessible to clients under any circumstance unless it is specifically for their use.	the lifespan of a game	See test below. None of the events sent to the clients from the page being loaded to the game finishing contain any information that they shouldn't have access to.
6	The game should be able to prevent illegal moves (e.g. placing a card when it is not their turn)	Check each event handler, it should have appropriate safeguards and validation techniques to ensure the event is valid.	Partially Completed  See test below. All events can only be triggered at expected times by the users that send them. The reason I have not said this is fully completed is as there is minimum validation for the content of events themselves. For example, a malformed object could cause the game to crash.
7	The game should be enjoyable	Two-thirds of testers say they found the game enjoyable to play	Completed  As evidenced by the reviews above, all three of my stakeholders found the game enjoyable.

Success Criteria Extended Tests

(2) Multiple Simultaneous Games

I tested joining the game, starting the game, and taking actions in the game. There was no problem with any of these and all parts of the game worked independently as if they were separate instances of the game.





(5) Checking for information leaking

event	event contents	Contains leaked information?
join_game	{"action":"join_game","room_code":"ytdjna"}	No
player_list_update	{ "action":"player_list_update", "players":[{"name":"Player 1","isHost":true,"handSize":14,"myTurn":false,"isMe":true}, {"name":"Player 2","isHost":false,"handSize":17,"myTurn":true,"isMe":false}, {"name":"Player 3","isHost":false,"handSize":14,"myTurn":false,"isMe":false}] }	No The only information about other players are the things that are displayed and the number of cards in their hand.
become_host	{"action":"become_host"}	No

start_game	<code>{"action":"start_game"}</code>	No ▾
update_cards	<code>{ "action":"update_cards", "cards":[{"rank":1,"suit":2},{ "rank":1,"suit":0}, {"rank":2,"suit":0},{ "rank":2,"suit":1},{ "rank":3 ,"suit":2},{ "rank":3,"suit":1},{ "rank":4,"suit":0 },{ "rank":4,"suit":3},{ "rank":5,"suit":1},{ "rank" :5,"suit":3},{ "rank":6,"suit":2},{ "rank":6,"suit" :0},{ "rank":6,"suit":1},{ "rank":7,"suit":1},{ "ran k":9,"suit":0},{ "rank":10,"suit":0},{ "rank":11,"s uit":3},{ "rank":11,"suit":1}] }</code>	No ▾ Only contains the cards in the client's hand
start_turn	<code>{ "action":"start_turn", "previous_rank":0, "current_rank":1, "can_accuse_cheater":false, "your_turn":false }</code>	No ▾ Only contains information that should be visible.
info_text_update	<code>{"action":"info_text_update","message":""}</code>	No ▾
end_game	<code>{ "action":"end_game", "game_over_text":"Player 2 won the game.", "can_play_again":true }</code>	No ▾
leave_game	<code>{"action":"leave_game"}</code>	No ▾

(6) Preventing illegal moves

Event	File	Correct response?	
join_game	main.js	Yes ▾	Any player can trigger this action, there is no need to check the validity of the request.
create_game	"	Yes ▾	Any player can create a new game.
start_game	"	Yes ▾	There is a check for the player.isHost flag
leave_game	"	Yes ▾	Any player can leave a game.
place_cards	game.js	Yes ▾	The place_cards event listener is only set on the
accuse_cheater	"	Yes ▾	An if statement checks if the accuse_cheater event can be sent and if it can't then it returns.

play_again	"	Yes ▾	The play again listener is only set on the host player
leave_game	"	Yes ▾	Any player can leave a game.

Evaluation Summary

Aesthetics

It is clear from my reviews that there is still a bit of a way to go in terms of how my project looks. Two out of three agreed that the design was too simplistic although one person said the clean design was good.

I do not consider this as a large problem for this first version of the game, as I was primarily focusing on UIs thinking first about functionality and usability. If I were to continue working on this project, I would further look at both online and offline card games and would think about adding a new background colour.

Code Quality & Project Structure

I have split the project's javascript code up into the following files:

server:

- main.js
- player.js
- game.js

client:

- index.js

common:

- card.js

This has been quite modular, with the card.js module being used across the client and server side code. In addition, the player and game classes are stored in their own individual modules along with their related functions.

I could have split the client side source into multiple files. For example, one contains event listeners, one contains the server communication code, and one contains client-side functions. This would have made the solution more maintainable.

Functionality

I would like to swap from using javascript asynchronous features and promises api in the game loop to using queues as it would bring these benefits:

- No events could be missed in dead time where the event listener is not applied yet as they would always be caught
- Games could be more easily recorded, paused, replayed, and errors if occurring could easily be recovered from without breaking the flow of the game loop.

The disadvantage to this approach would be that firstly I would need to re-write the mass of the game logic and secondly I may no longer be able to structure the game loop logically. Instead I may need to flatten the state of the game out into a variety of procedures that respond to different types of requests coming in taking into account the current state of the game, more of which would have to be recorded in game class attributes for example.

Despite this, I believe that making this change would be better overall for the game as it would no longer rely on promises and could better respond to different events coming in at different times. For example: people leaving, a turn time limit, and gameplay events all happening simultaneously and the main game "loop" having to account for all of these in making its decisions.

Robustness & Error Checking

Although there is some prevention against invalid requests through the validation of requests to only come from the player it is expected from, the general coverage of error handling is extremely minimal throughout the project. Throughout the project there are opportunities for errors to occur. For example, when a "join_game" is sent there could be an error parsing the JSON object, it may then have missing properties which causes the game to crash.

In order to improve this, I would firstly need to add try/catch blocks more frequently across the project - restarting the game loop if it catches one.

User Experience

One feature I think would be greatly beneficial to my stakeholders is a settings screen. In Harrison's review, he stated he would like the option to place cards below the previous one, which is a variation on the game of Cheat. Adding an options menu would allow for players to choose which rules they want to play with.

Another feature I would consider adding would be sound effects. This was mentioned by my stakeholders as something the game was lacking.

Finally, if I had more time I would improve the player list as stated in the evaluation for the waiting screen feature. I would make indicators for "me"/"host" clearer and separated from the player name as well as allowing the host player to kick people from the game.

Enhancing Usability

In terms of usability, my UIs seem simple to use and understandable. This is evidenced by all three of my stakeholders finding the UI understandable.

Despite this, I believe that there are still some improvements that could be made:

- To replace the error and information messages with a "toast" popup element.

This would replace:

- the text on the start screen e.g "The game with the provided code does not exist"
- errors saying "Three or more people are required to start the game."
- status messages like "Player 1 left the game" and "Player 4 was a cheater"

It could also pop up when a turn starts or players place cards down

This would make it easier to tell when something happens in the game and would make it clearer when you have done something wrong.

In addition, although none of my stakeholders said that touch controls were something they required from the solution I think this could be an interesting market to consider, especially as my project is built using web pages and adding support for touch events shouldn't take a large amount of work.

Limitations in Scalability

As stated in my success criteria and the limitations section of my analysis I do not have access to a large number of people to test my game at scale however given the testing I conducted on two games running simultaneously, I can assume that it would work for at least a dozen more instances. This assumption becomes limited past small numbers of concurrent games as it is much more difficult to predict whether or not it would handle the traffic fine or if it would slow down/memory would grow too quickly.

I do not know how I would have gone about stress-testing the system as simulating many games going on would be quite a complex problem to solve, especially given the limited amount of time I had.

If I were to continue the project and explore the problem of scalability further I might look at measuring memory growth, simulating large numbers of clients joining the game and taking actions and completing further development cycles with this as a focus.

Conclusion to the project

Overall, as most of the success criteria have been completed to some extent, my stakeholders were satisfied with my solution, and the majority of features have been implemented I would say that I was successful in my initial goal of applying a computational approach to the game of cheat, creating an online alternative to the card game that allows friends to play together from anywhere in the world.

Bringing this project into the further development however, I would focus on the limitations I have identified in my evaluation being:

- Look into other designs for the game, such as different background colours, in order to add personality to my game.
- Ditching the current system where event listeners are continuously added and removed in order for the game to operate in favour of an event queue based system.
- Adding options and settings screens for players in order for them to enhance and modify their experience playing the game.
- Consider scalability as more of a concern, researching ways to simulate large amounts of traffic and checking for places where errors could be thrown.

I would also consider expanding the scope of the project. It could support multiple games, and the player selects which game they want to play on the start screen. This would make it more engaging for players as they have multiple options to choose from.

References

Cheat (game) - Wikipedia - [https://en.wikipedia.org/wiki/Cheat_\(game\)](https://en.wikipedia.org/wiki/Cheat_(game))

Cheat rules - <https://www.pagat.com/beating/cheat.html>

Cardmania card game - <https://www.cardzmania.com/games/Cheat>

Skribbl.io - <https://skribbl.io>

Http long polling - <https://ably.com/topic/long-polling>

NodeJS web server libraries -

<https://reintech.io/blog/best-libraries-for-using-nodejs-with-web-server>

express package - <https://www.npmjs.com/package/express>

ws package - <https://www.npmjs.com/package/ws>

NodeJS hardware requirements - <https://github.com/nodejs/help/issues/1323>

Websocket API documentation -

https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_client_applications

Fisher-Yates shuffle - https://en.wikipedia.org/wiki/Fisher-Yates_shuffle