Card Counting Program for Blackjack

Chan Ho Kim
Department of Information System
Hanyang University
Seoul, Korea
chanhokim95@gmail.com

Young Mo Koo
Department of Information System
Hanyang University
Seoul, Korea
kooym1@naver.com

Do Hyun Kim
Department of Information System
Hanyang University
Seoul, Korea
qawsed6518@gmail.com

Yeon Cheol Kang
Department of Information System
Hanyang University
Seoul, Korea
kyc3492@gmail.com

Chae Eun Lee
Department of Information System
Hanyang University
Seoul, Korea
421lilac@naver.com

***Abstract*—In this paper, we examine how the game of Blackjack is constructed and how gameplay decision with the help of card counting can increase the game's winning rate. We will make our own Blackjack game with programing and use this to collect and analyze statistics on card counting. This program provides you information in real time to help you make betting-gameplay decision. In the first half of the paper, we discuss how the game is constructed, and find out the way to provide an environment similar to a real casino on a computer. Next, we analyze the card counting algorithm and apply it to our Blackjack game. Finally, we propose possible extensions of our work in the form of new areas that command further research interest.**

Keywords—Blackjack, card-counting algorithm, gameplay decision, betting decision, static analysis.

I. Introduction

Recently, Movies such as Rain Man, 21 and The Hangover increase interest of card counting in Blackjack and have popularized the notion that tracking the number of "high" and "low" cards in play guarantees a high-payoff, but just watching the genius in the movies does not guarantee understanding card counting and does noting on your casino winning ratee. We want to make something more practical and usable than these movies.

First, we must know what Blackjack is. Blackjack, also known as twenty-one, is a comparing card game between usually several players and a dealer, where each player in turn competes against the dealer, but players do not play against each other. It is played with one or more decks of 52 cards, and is the most widely played casino banking game in the world. The objective of the game is to beat the dealer in one of the following ways: Get 21 points on the player's first two cards (called a "blackjack" or "natural"), without a dealer blackjack; Reach a final score higher than the dealer without exceeding 21; or Let the dealer draw additional cards until their hand exceeds 21.

We choose Blackjack because 1) It is the most famous playing-card game in all its aspect. Many studies have already studied Blackjack mathematically and have had perfect result. The only drawback is that these studies are really hard to understand for ordinary people like us. 2) Blackjack is the only game that guarantees a stable winning rate for player. Other games, like poker, winning rate of casino is set much higher than player. This gap cannot be reduced even if player uses perfect card counting. But Blackjack is different. If there are no special rules for casinos, you can record a winning rate of over 50%. This means, Blackjack is only game that you can make pos of money at the casino.

Card counting is a casino card game strategy used primarily in the blackjack family of casino games to determine whether the next hand is likely to give a probable advantage to the player or to the dealer. Card counters are a class of advantage players, who attempt to decrease the inherent casino house edge by keeping a running tally of all high and low valued cards seen by the player. Card counting allows players to bet more with less risk when the count gives an advantage as well as minimize losses during an unfavorable count. Card counting also provides the ability to alter playing decisions based on the composition of remaining cards. Card counting, also referred to as card reading, often refers to obtaining a sufficient count on the number, distribution and high-card location of cards in trick-taking games such as contract bridge or spades to optimize the winning of tricks.

The program we provide will have some special funtions. Like any ordinary flash web game, you can access our games and play blackjack games on the Internet. Additionally, You can get informations, like the probability of what the next open card will be and advice on gameplay-betting decision. It is possible, if you are a player with administrative right, that getting information for the next card of other players.

Roles	Name	Task description and etc.
User	Lee Chaeun Kim Dohyun	- determine if software is needed for users - make sure there are no inconveniences to use and feedback to the manager or developer - check if software was created as requested
Customer	Koo Youngmo	- considering that this program is necessary for me. - thinking about whether it is worth buying this program. - investigating if there is a program that offers better functionality
Software developer	Kang Yeoncheol	- researching and implement the Blackjack card-counting algorithm. - researching how to train the user. - thinking about necessity of multiplayer mode. - UI and UX, looks like the user is in real casino.
Development manager	Kim Chanho	- Organize every step in development - Make sure the program runs well

II. REQUIREMENTS ANALYSIS

Our primary goal is to develop the software that visually helps blackjack's card counting. We aim to make the program that let blackjack players know when the advantage shifts in their favor. When this occurs, they will increase their bets. When the advantage shifts in favor of the dealer, the players will make a smaller bet or no bet at all by not playing. Players can gain a positive advantage over the casino by varying bets referring to the probability information that we show visually. In order to clarify the functional specification, we investigated the level of necessity for the proposed system and uncovered the following needs:

- a necessity for, and an interest in automated card counting systems
- information about the card combination probability and betting decision which is useful for the players
- high accuracy and speed
- a need for multiple players to access the game at the same time

The requirements analysis also uncovered the following design requirements:

- common flash web game-like operations
- a graphical user interface (GUI) system that provides real-time information and guidance for the players

III. 3DEVELOPMENT ENVIRONMENT

A. Choice of software development platform

We will be developing on Mac and Linux using virtual machines. We will build a website and implement a blackjack counting program in it. The reason we chose the Web is because it is the easiest way for users to access and because they can be used by both the computer and the mobile phone.

B. Software in use

We will use HTML and CSS on frontend and will use Node.js on backend. It will also store user's database through MySQL. The language used to build the algorithm and program will use JavaScript.

Tools and language	Reason
HTML	HTML (Hypertext Markup Language) is the standard markup language for creating web pages and web applications. With CSS (Cascading Style Sheets) and JavaScript, it forms a triad of cornerstone technologies for the World Wide Web.
CSS	CSS (Cascading Style Sheets) is a style sheet language used for describing the presentation of a document written in a markup language like HTML. CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript.
Node.js	Node.js is an open-source, cross-platform JavaScript run-time environment that executes JavaScript code outside of a browser. We will use Node.js to develop out blackjack counting program. Node.js will help out program be easy to develop and implement well on the web. That's why we picked this.
Node.js express	Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.
jQuery	jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.

C. Task distribution

We are made up of five team members. Chan-ho Kim, Young-mo Koo, Do-hyoun Kim, Yeon-cheol Kang, Chae-eun Lee. We will have on person in charge of server construction and MySQL query production, three in blackjack card counting program, one in HTML and CSS production. Yeon-cheol Kang is responsible for the server and MySQl.. Chae-eun Lee takes on HTML and CSS to make the frontend. Young-mo Koo, DO-hyun Kim, Chan-ho Kim are responsible for blackjack card counting program. The blackjack card counting program aims to create an overall blackjack card game, including a count function.

IV. SPECIFICATIONS

A. Blackjack-Equipment

Blackjack is played with a standard international deck of cards with the Jokers removed, leaving 52 cards. Originally the game was played with a single deck. However, as a counter measure to card counting, casinos introduced multi-deck games, based on the false assumption that if there were more cards in play it would be harder for the card counter to keep track of them all. As a result, Blackjack is now usually offered in either single deck, double deck, 4 deck, 6 deck or 8 deck variants. It should be noted that there are exceptions in online casinos where far larger numbers of decks can be used than would be practical to manage offline.

Aside from the cards, the game requires a table, chips, a discard tray, cut card and a shoe.

After the dealer has shuffled a player will be selected at random and asked to take the cut card – a colored plastic card matching the playing cards in size – and place it at a random position within stack of cards. The dealer will then move the cards above the cut card to the back of the stack. This technique is intended to demonstrate to the players that the dealer cannot have rigged the deck. The cut card is then reinserted into the stack of cards by the dealer at a pre-defined position and when this card is reached this indicates the final deal of the game before the cards are shuffled.

Where multiple decks are used, after the shuffle the cards will be placed into a dispenser called a shoe. This piece of equipment has two purposes: to hold large stacks of cards in multi-deck games and make the practice of **hole carding** (cheating by catching a glimpse of the dealer's hole card) more difficult. In fact, hole carding is not illegal in the vast majority of jurisdictions. If the dealer is poorly trained or sloppy enough to fail to protect their down card from being seen ***by a player at the table*** this is not the player's fault and the player is not obliged to look away to prevent themselves seeing the down card. If however the player uses any form of device, for instance a metal lighter to observe the reflection in, or an accomplice off table signals the information to them, this is cheating. Hole carding is only legal where the player can see the card naturally from one of the player positions at the table.

B. Blackjack-Card Values

When playing Blackjack the numeral cards 2 to 10 have their face values, Jacks, Queens and Kings are valued at 10, and Aces can have a value of either 1 or 11. The Ace is always valued at 11 unless that would result in the hand going over 21, in which case it is valued as 1.

Any hand with an Ace valued as 11 is called a '**soft**' hand. All other hands are '**hard**' hands.

A starting hand of a 10 valued card and an Ace is called a **Blackjack** or **natural** and beats all hands other than another **Blackjack**. If both the player and dealer have Blackjack, the result is a **push** (tie): neither the player nor the bank wins and the bet is returned to the player.

C. BlackJack-Order of Play and Playing Options

Each player sitting at the table places their desired bet in the betting circle directly in front of them. In most casinos if there are untaken betting circles, the players sitting at the table can choose to play more than one hand at a time. The minimum and maximum bet size varies from casino to casino, generally with a ratio of 40 to 100 between them. For example, with a $25 minimum bet the maximum will usually be somewhere from $1000 to $2500. Once the bets are placed the dealer will move their hand across the table from their left to their right signaling that no further bets can be placed. The dealer then deals cards one at a time clockwise around the table, from the dealer's left to the dealer's right: first a card face up to each betting circle that has a bet in it, then a card face up to the dealer, and then a second card face up to each betting circle with a bet and finally a second card face down to the dealer.

In many places the dealer's first card is initially dealt face down. The dealer's second card is used to flip the first card face up and then slid underneath the first card. The exact dealing protocol varies from place to place as determined by the casino management.

If the dealer has a 10 or an Ace face up players are offered the option to place an **Insurance** bet. If a player chooses to take insurance they place an additional bet equal to half of their original bet. This insurance bet wins if the dealer has Blackjack.

The dealer now checks their down card to see if they have Blackjack. If they have Blackjack they expose their down card. The round is concluded and all players lose their original bet unless they also have Blackjack. If a player and the dealer each have Blackjack the result is a push and the player's bet is returned. Any insurance bets are paid out at 2:1.

If the dealer does not have Blackjack any insurance bets are lost and any players who have Blackjack are paid. It is then the turn of the remaining players to take their actions. Starting with the player sitting furthest to dealer's left they have the following options:

Stand – If the player is happy with the total they've been dealt they can stand, taking no further action and passing to the next player. The player can take this action after any of the other player actions as long as their hand total is not more than 21. The hand signal to Stand is waving a flat hand over the cards.

Hit – If the player wishes to take another card they signal to the dealer to by scratching the felt beside their hand or pointing to their hand. A single card is then played face up onto their hand. If the hand total is less than 21 the player can choose to Hit again or Stand. If the total is 21 the hand automatically stands. If the total is over 21 the hand is bust, the player's bet is taken by the house and the turn to act passes to the next player.

Double Down – If the player considers they have a favorable hand, generally a total of 9, 10 or 11, they can choose to 'Double Down'. To do this they place a second wager equal to their first beside their first wager. A player who doubles down receives exactly one more card face up and is then forced to stand regardless of the total. This option is only available on the player's two-card starting hand. Some casinos will restrict which starting hand totals can be doubled.

Split – If the player's first two cards are of matching rank they can choose to place an additional bet equal to their original bet and split the cards into two hands. Where the player chooses to do this the cards are separated and an additional card is dealt to complete each hand. If either hand receives a second card of matching rank the player may be offered the option to split again, though this depends on the rules in the casino. Generally, the player is allowed a maximum of 4 hands after which no further splits are allowed. The split hands are played one at a time in the order in which they were dealt, from the dealer's left to the dealer's right. The player has all the usual options: stand, hit or double down.

A player who splits Aces is usually only allowed to receive a single additional card on each hand. Normally players are allowed to split two non-matching 10-value cards, for example a King and a Jack. If Aces are split and the player draws a Ten or if Tens are split and the player draws an Ace, the resulting hand does **not** count as a Blackjack but only as an ordinary 21. In this case the player's two-card 21 will push (tie with) dealer's 21 in three or more cards.

Surrender – Most casinos allow a player to surrender, taking back half their bet and giving up their hand. Surrender must be the player's first and only action on the hand. In the most usual version, known as Late Surrender, it is after the dealer has checked the hole card and does **not** have a Blackjack.

After all players have completed their actions the dealer plays their hand according to fixed rules. First they will reveal their down card. The dealer will then continue to take cards until they have a total of 17 or higher. The rules regarding Soft 17 (a total of 17 with an Ace counted as 11 such as A+6) vary from casino to casino. Some require the dealer to stand while others require additional cards to be taken until a total of hard 17 or 18+ is reached. This rule will be clearly printed on the felt of the table.

If the dealer busts all non-busted player hands are automatically winners.

D. Blackjack-Payouts

If the player and dealer have equal unbusted totals the hand is considered a push and the player's bet is returned.

If a player wins a hand they are paid out at 1:1 on the total bet wagered on that hand. For example. if the player wagered $10 and then doubled down placing a further bet of $10 on the hand and won, they would be paid a total of $40, their $20 bet back and $20 winnings.

If the player has Blackjack they are paid at 3:2, so that a wager of $10 the player would be paid a total of $25, their $10 bet back plus $15 winnings.

If the player has placed the Insurance bet and the dealer has Blackjack, the player's hand loses but the Insurance bet is paid out at 2:1. So if the player had bet $10 on the hand and $5 on the Insurance bet, they would lose the $10 and be paid a total of $15 – their $5 Insurance bet returned and $10 winnings. This effectively results in a push overall for the hand.

E. Blackjack-Variants

1) Deal

In some casinos the players' initial two-card hands are dealt face down. All additional cards dealy to the player are given face up. The initial cards are revealed by the player if the hand goes bust, or if the player wishes to split a pair. Otherwise the dealer reveals the cards at the end of the round when it is time to settle the bets. This style of game is rare nowadays: casinos don't like to allow players to touch the cards, because of the risk of card marking.

In **European style** games only the dealer's face up card is dealt the start of the round. Dealer's second card is dealt **after** all players have acted, and the dealer checks for Blackjack at this point. Player Blackjacks are paid at the end of the round if the dealer does not have Blackjack. If the dealer has Blackjack the rules regarding Doubled and Split hands vary from casino to casino. Some casinos will take both bets while others will only take the initial bet and return the other.

2) Blackjack payout

It should be noted that some casinos have started to offer a reduced payout on Blackjack, most commonly 6:5. This is very bad for the player, increasing the House Edge significantly. Any game offering a reduced payout on Blackjack should be avoided by players.

3) Splits

The maximum number of hands that can be created by splitting depends on the rules in the casino: some only allow one split.

When splitting 10 value cards, not all casinos will allow players to split non-matching 10 cards. For instance, in some casinos you could split two Jacks but could not split a King and a Jack. Also, some casinos will limit which card ranks can be split.

House rules will dictate whether the player is allowed to Double after splitting, and whether a player who splits Aces is allowed to receive more than one additional card on a hand.

4) Surrender

Not all casinos offer the Surrender option.

A few casinos may offer Early Surrender in which the player can take back half of their bet and give up their hand before the dealer checks for Blackjack. This is very rare nowadays

In European style games There is normally no Surrender option. If Surrender were offered it would of course have to be Early Surrender.

5) Five Card Charlie

The side rule is rarely offered. When it is in effect, a player who collects a hand of five cards (two cards plus three hits) without going bust is immediately paid even money, irrespective of the dealer's hand.

6) Home game blackjack

Blackjack can be played at home, rather than in a casino. In this case a fancy Blackjack table is not needed: just at least one pack of cards and something to bet with - cash, chips or maybe matches. Unless the players have agreed in advance that the host should deal throughout, to ensure a fair game the participants should take turns to be the dealer. The turn to deal can pass to the next player in clockwise order after every hand or every five hands or whatever the players agree. If playing with a single deck of cards, it is desirable to re-shuffle the cards after every hand.

7) Swedish Pub Blackjack

Nightclubs and pubs in Sweden often offer a Blackjack variant that is less favourable to the players. All the essential rules are the same as in the casino version unless the player and dealer have an equal total of 17, 18 or

19. In the casino version the player's stake is returned in these situations, but in Swedish pubs the house wins.

Although pub stakes may vary, they are often much lower than in casinos with a minimum stake of 20 or 40 Krona and a maximum of 60 Krona (about US$7) for each hand.

F. Card Counting

Card Counting provides the player a mathematically provable opportunity to gain an advantage over the house. It must be understood that this does not guarantee that the player will win. Just as a regular player may win though good luck despite playing at a disadvantage, it is perfectly possible for the Card Counter to lose through an extended period of bad luck even though playing with a small advantage over the House.

The basic premise of Card Counting is that mathematically speaking, low cards on average are beneficial to the dealer while high cards favour the player. There are many subtle reasons for this but the most significant are:

- A player who receives a Blackjack (a ten value card and an Ace – two high cards) is paid one and a half times their bet. The dealer however only receives the player's bet when dealt a Blackjack.

- While the player can stop taking additional cards at any time, rules require the dealer to continue drawing cards until they reach a total of 17. The the player can choose whether or not to take an additional card on a total of 16 whereas the dealer has to take one. In this situation small cards are less likely to cause the dealer to bust are thus favour the dealer, while big cards cause the dealer to bust more often and favour the player.

- The majority of situations where it is correct of the player to double are starting hands that would be made very strong by the addition of a ten value card or an Ace. Therefore, doubling becomes more favourable when there are more ten value cards and Aces left in the deck.

So the Card Counter looks for times when there are more high cards left to be played than a regular deck would have. Rather than trying to remember each card that has been played, the Card Counter will usually use a ratio system that offsets cards that are good for the player against cards that are good for the dealer.

The most commonly used Card Counting system is the **HiLo count**, which values cards as follows:

High cards: 10, J, Q, K, A:	-1
Medium cards: 7, 8, 9:	0
Low cards: 2, 3, 4, 5, 6:	+1

To keep track the player starts at zero, adds one to the total every time a low card is played and subtracts one from the total when a high card is played. This is called the 'Running Count'. It may seem counter-intuitive to subtract one for high value cards that are good for the player, but a high card that has been played is one less high card that is left to be played. Where the Running Count is positive the player knows that there are more player favourable cards remaining to be played.

When kept correctly the Running Count will start at 0 and, if all the cards were to be played out, would end at 0. This is because there are an equal number of high cards and low cards. The HiLo count is therefore referred to as a 'Balanced Counting System'.

Card Counting systems are generally not impeded by the addition of multiple decks to the game. At any rate multiple decks do not make it significantly more difficult for the Card Counter to keep track of the Running Count, since the Card Counter only needs to keep track of a single number, the Running Count. However many decks are used, the count begins at zero and would end at zero if there were no cards left, so no changes need to be made to the counting process.

Where multiple decks do make a difference is in how much impact a positive Running Count has to the player advantage. If the Running Count is +10 and there are two decks remaining to play, this means there are an extra 5 player favourable cards in each deck. If there are 5 decks remaining to be played there are only 2 extra player favourable cards in each deck. The higher the concentration of extra player favourable cards the stronger the player’s advantage. To estimate the strength of the player advantage the Running count therefore needs to be divided by the number of decks remaining to be played. This figure is called the True Count.

With the True Count the player has a consistent measure of how many extra player favourable cards are contained within the cards remaining to be dealt. The player can use this information to vary their bet and playing strategy. Deviations from Basic Strategy are far less important than placing big bets when the True Count is high and low bets (or preferably nothing) when the True count is low or negative.

While Card Counting is legal in most jurisdictions, for obvious reasons casinos do not like players that can consistently beat them. They therefore employ counter measures and any players they identify as Card Counters will be asked to leave the casino. The most common method used to identify Card Counters is to watch for a large bet spread (difference between the minimum and maximum bet a player uses) and to see whether large bets correlate with player favourable counts. Card Counters have developed several methods to help them avoid detection. The two most common are:

- **Wonging / Back Counting.** Named after Blackjack author Stanford Wong, this is the practice of watching the cards being played and only sitting down to play when there is a player favourable count. This practice reduces the bet spread the player uses as they only place bets in player favourable situations but casinos are now well aware of this strategy and watch out for players hanging around a table and not playing. The method is still useful, but not without its problems.
- **Team Play**. This involves several trained Card Counters working together. Most commonly there would be several 'Spotters' sitting at different tables keeping track of the count and either back counting or playing minimum bets. When a table reaches a positive count the Spotter would signal to the 'Big Player' who would come over and bet big during the player favourable count. This allows both players to make very little variation in their bets. Casinos are aware of this strategy and watch for groups of players working together.

There are several variations on team play designed to be employed in different situations and to different effects. These are covered more fully in the reading resources detailed below.

Successful Card Counting is generally only profitable in land based casinos, not in online games. The strategy relies on the game having a "memory" in that cards are dealt from the cards remaining after previous rounds have been played. Online Blackjack games are dealt by computer and normally use a random number generator to shuffle the whole deck after every round of play. Games of this sort are not countable.

There are some Live Blackjack games online, which are played over a video feed with a human dealer. These could technically be counted but there are several significant disadvantages that make this difficult or not worth the player's time:

1) Games of this type are very slow to play. A slow game means less money made.

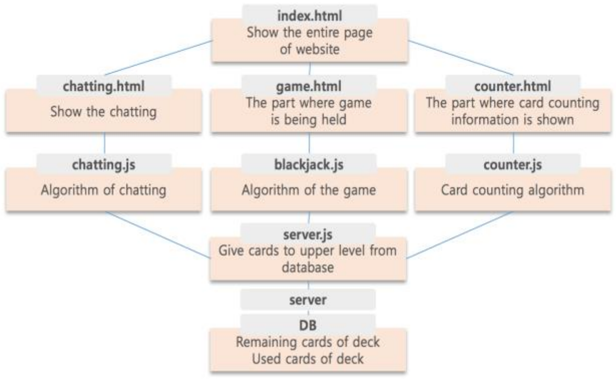
2) The games generally offer poor "penetration". This means that the decks are shuffled early, not allowing enough cards to be dealt out for many player favourable situations to develop. (The most favourable situations for the player tend to occur further into the shoe.)

3) The casino's software records every player bet and all the cards dealt. This makes it relatively easy for a casino to employ software to track the count and watch for players raising their bet or only playing when the count is favourable.

For the above reasons Card Counting has not become commonplace online.

V. Architecture Design & Implementation

A. Overall Archotecture

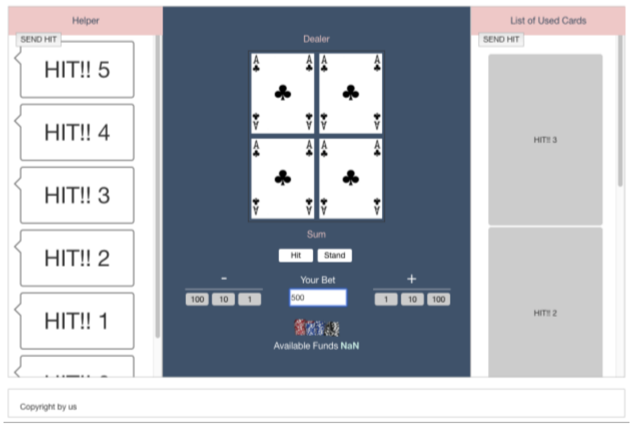


B. Directory Organization

Directory	File names
/project/index.html	index.html
/project/src_html	counter.html, game.html, chatting.html
/project/src_js	counter.js, blackjack.js, chatting.js
/project/server/server.js	server.js

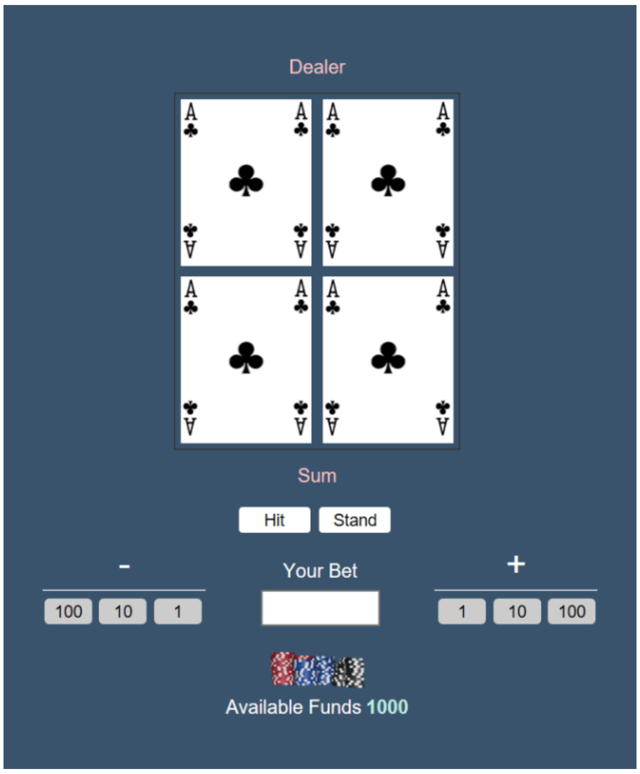
C. Index.html

We first created a parent file, index.html, to group all the parts(game.html, counter.html, helper.html) together. We created a code that divides the page into three parts and then loads each three different codes. Collaboration was easier because we had three separate parts and we were able to implement the behavior as if there were three different pages in one page.



D. game.html

This part is located at the center of the page and is where the Blackjack game progresses. No data is displayed in this section and the player only plays the game. At the top of the page, there is a 2x2 size table with the text telling player that the upper side is the dealer. In each column of the table, the card information of the dealer and the player running on the back-end blackjack.js will be received and the corresponding card image will be displayed. The card image will change in real time as the game progresses. And below is the text that tells the player the sum of the current card numbers. Player is able to know the current sum of their cards in real time by checking this number and decide whether to hit or stand. Next, there is a section to adjust the bet amount. We configured the three buttons on both sides so that the player can add or subtract bet in units of 1, 10, or 100. The player can also enter his own bet in the text box. And after adjusting the bet, player should press the button below to see the remaining funds. Changes in bet amount are immediately applied to the available funds below.



E. counter.html

'counter.html' is designed to show which cards are appeared already. User can seize which has been used and which cards are left, receiving data from 'counter.js'. For example, let's suppose there are 4 aces in the list of counter.html and the game is using only 1 set of deck. The player can be at ease because he or she knows there is no case of dealer's blackjack. We considered a lot of designs to show the list to the player effectively. At the end, we thought showing the latest data on the top of the list is the best way to show the order of the appeared cards.

F. chatting.html

'chatting.html' is a page that shows the chatting between multiple players. The chatting information between players comes from the 'chatting.js'

G. counter.js

The counter.js is directly connected with counter.html and server.js. The counter.js does card counting so it can give the information of card counting to counter.html. The information of cards is given by server.js. The logic of card counting is simple enough to understood by anybody. There are three values which are running counts, the number of remaining decks, and true counts. For running counts, values are +1 for 2, 3, 4, 5, 6 cards, +0 for 7, 8, 9 cards, and -1 for 10, 'J', 'Q', 'K', 'A' cards. True counts can be calculated as running counts divided by the number of remaining decks. Considering basic concept of card counting, we included fucntions which are countCards(card), remainingDecks(deck), trueCounts(countCards(), remainigDecks()), sendUsedCard(card). Argument values for countCards() and remaininDecks() will be given from the server. In addition, the information of used cards will be sent to counter.html by sendUsedCard(card). After the calculation, values will be sent to counter.html, and the information of card counting will be shown on it.

H. blackjack.js

blackjack.js is responsible for the functionality that runs in game.html. blackjack.js connects game.html and server.js to make game functions and values work properly. blackjack.js consists of reset function, deal function, and ready function. The reset function returns all values in game.html to their initial values. Whenever the deal function is called, the reset function is first called to initialize the blackjack game running on game.html.

The deal function basically makes the game run. The first thing to do is to get the card value from the database and configure the deck. Always calculate dealer and player card values. The dealer will hit and stand in accordance with established rules. The player has two functions: hit function and stand function. When the hit function is executed, a new card is added to the existing card in the database. At this time, the new card comes from the remaining card deck table. The used deck stand function in the database will terminate the last card in the game. When the game ends, all used cards are stored in the used card deck in the database. After the stand function, dealer and play are determined by the value of the card they have. If player's result exceeds 21, player is defeated regardless of dealer's result value. If the result value of the player is 21 or less, if the result value of the dealer exceeds 21, the player will win the game unconditionally. The dealer with the highest value among the player wins. If the values are the same, they are drawn. The player receives twice the amount of the bet when it wins and reads all the bets if it is defeated. In the case of a draw, the bet money is returned to the player.

The ready function runs the bet function. The bet function allows the player to raise and lower money. The player is given a button to bet, and the amount of the bet is determined by the bet amount each time the button is pressed. The amount of the bet is increased by the amount that the player has. When executing the ready function, the deal function is executed first.

I. chatting.js

The 'chatting.js' is directly connected to the 'chatting.html' the algorithm of the chatting (between client and server) is included in this file.

J. server.js

BLACKJACK CARD COUNTER's server runs on node.js – which has powerful modules. The program sends and receives data using these modules. 1)HTTP module is designed to support many features of the protocol which have been traditionally difficult to use. In particular, large, possibly chunk-encoded, messages. The module is careful to never buffer entire requests or responses — the user is able to stream data. 2) Express module is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. With a myriad of HTTP utility methods and middleware at your disposal, creating a robust API is quick and easy. Express provides a thin layer of fundamental web application features, without obscuring Node.js features that you know and love. Many popular frameworks are based on Express. 3) Socket.io module is a library that enables real-time, bidirectional and event-based communication between the browser and the server. It consists of a Node.js server and a Javascript client library for the browser (which can be also run from Node.js). Its main features are reliability, auto-reconnection support, disconnection detection and multiplexing support.

Javascript client and node.js server communicate in the form of 'Data' class. 'Data' class consists of 1) array of player and dealer's current hands 2) player's available fund 3) player's bet and 4) Card Counting information. Basically, node.js uses transmission control protocol(socket.io). The program sets connection between server and client, and then transports data. Client and server each have same "Event names". They work in groups of the same name. For example, client's socket.emit method of "GAME_START" event send data. And server's socket.on method of "GAME_START" event receives this data.

To request "GAME_START" event, client send 'BET' (how much player put money on). After receives 'Bet', server create new "Data" and write back. As mentioned above, "Data" contains all information that client should know. As Blackjack game progresses, server and client exchange information in "Data" format each time the information is modified. Player needs information of what cards he and dealer have on hand and Card counting status to get help in game play decision making. Every time program draws a card, Card counting status should be changed, so it must be modified at every event. The cards sent to player are also sent to list of used cards. If player presses 'HIT' button, event reasoner names 'Hit' is connected. Server operates almost the same as before. When the player finishes making gameplay decision, 'Dealer' will operate automatically, and show results at the same time.

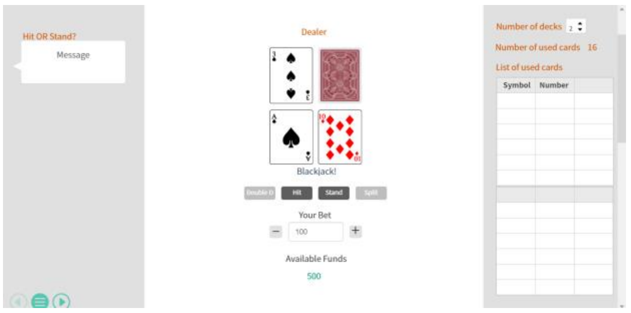
```
<pseudocode of server.js>
var io = socketio.listen(server);
var bj = require('./js/blackjack.js');
deck.shuffle();

io.sockets.on('connection', function (socket) {
   socket.on('game_start', function(bet){
      client_card = bj.start_game(); // client_card = [dealer_card[], player_card[]]
      usedcard += client_card;
      deck -= usedcard;

      data.counting_info = bj.cardcounting(usedcard);
      data.card = client_card;
      data.fund = data.fund - bet
      data.bet = bet

      io.sockets.emit('game_start', data);
   });

   socket.on('hit', function(data){
      draw_card = deck.drawscard();
      usedcard += draw_card;
      deck -= draw_card;

      client_card.player += draw_card;

      data.counting_info = bj.cardcounting(usedcard);
      data.card = client_card;

      io.sockets.emit('hit', data);
   });

   socket.on('dealer', function(data){
      draw_card = deck.drawscard();
      usedcard += draw_card;
      deck -= draw_card;

      client_card.dealer += draw_card;

      data.counting_info = bj.cardcounting(usedcard);
      data.card = client_card;

      io.sockets.emit('dealer', data);

   });
```

```
    socket.on('win', function(data){
        data.fund = data.bet * 2;
        data.bet = 0;
        io.sockets.emit('dealer', data);
    });

    socket.on('blackjack', function(data){
        data.fund = data.bet * 2.5;
        data.bet = 0;
        io.sockets.emit('dealer', data);
    });


});
```

VI. Use Cases

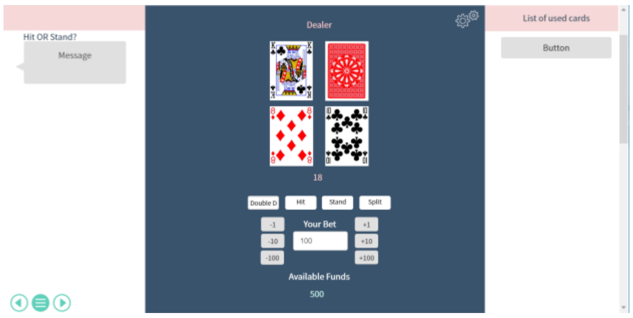
A. *UI Design*

We used the ‘Oven’ to frame the overall layout of the page. It is a HTML5-powered free web/app prototyping tool by *Kakao and we* used this tool to specify the color of the background, the components of each part, and the size of the objects. Also, since team members share the same account and collaborate, we could implement each of our own designs in real time and visually check them all right away.



<initial design>

The initial design was based on blackjack open source web page code. A blackjack game is played in the game part located at the center of the page, and both sides provide card counting information. We have considered various ways of providing information. We also had to design the button, how it works, and the color of each element. All these decisions were made in the Oven and after all the team members agreed, we started to do full-scale programming. The final design with feedback on the card images, both side implementations, and the format of the card counting information to provide is as follows.



<final design>

VII. Software Installation Guide

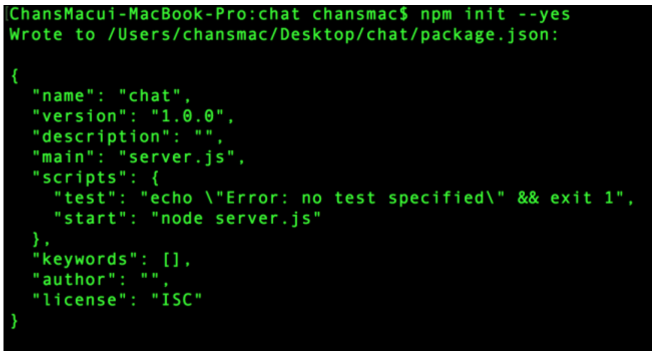
A. *Package.json*

We used node.js to run our own server. To run the server using node.js, the following instruction is preferred.

npm init <initializer> can be used to set up a new or existing npm package. Initializer in this case is an npm package named **create-<initializer>**, which will be installed by npx, and then have its main bin executed – presumably creating or updating **package.json** and running any other initializer-related operations. The init command is transformed to a corresponding npx operation as follows:

1) npm init foo → npx create-foo
2) npm init @usr/foo → npx @usr/create-foo
3) npm init @usr → npx @usr/create

Any additional options will be passed directly to the command, so **npm init foo –hello** will map to **npx create-foo –hello**. If the initializer is omitted (by just calling **npm init**), init will fall back to legacy init behavior. It will ask you a bunch of questions, and then write a package.json for you. It will attempt to make reasonable guesses based on existing fields, dependencies, and options selected. It is strictly additive, so it will keep any fields and values that were already set. You can also use **–y / --yes** to skip the questionnaire altogether. If you pass **--scope**, it will create a scoped package.



```
ChansMacui-MacBook-Pro:chat chansmac$ npm init --yes
Wrote to /Users/chansmac/Desktop/chat/package.json:

{
  "name": "chat",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

The following shows all you need to know about what's required in your package.json file. It must be actual JSON, not just a JavaScript object literal.

A lot of the behavior described in this document is affected by the config settings described in **npm-config.**

1) name

If you plan to publish your package, the *most* important things in your package.json are the name and version fields as they will be required. The name and version together form an identifier that is assumed to be completely unique. Changes to the package should come along with changes to the version. If you don't plan to publish your package, the name and version fields are optional.

The name is what your thing is called.

Some rules:

- The name must be less than or equal to 214 characters. This includes the scope for scoped packages.
- The name can't start with a dot or an underscore.
- New packages must not have uppercase letters in the name.
- The name ends up being part of a URL, an argument on the command line, and a folder name. Therefore, the name can't contain any non-URL-safe characters.

Some tips:

- Don't use the same name as a core Node module.
- Don't put "js" or "node" in the name. It's assumed that it's js, since you're writing a package.json file, and you can specify the engine using the "engines" field. (See below.)
- The name will probably be passed as an argument to require(), so it should be something short, but also reasonably descriptive.
- You may want to check the npm registry to see if there's something by that name already, before you get too attached to it. https://www.npmjs.com/

2) version

If you plan to publish your package, the *most* important things in your package.json are the name and version fields as they will be required. The name and version together form an identifier that is assumed to be completely unique. Changes to the package should come along with changes to the version. If you don't plan to publish your package, the name and version fields are optional.

Version must be parseable by node-semver, which is bundled with npm as a dependency. (**npm install semver** to use it yourself.)

More on version numbers and ranges at semver.

3) description

Put a description in it. It's a string. This helps people discover your package, as it's listed in **npm search**.

4) keywords

Put keywords in it. It's an array of strings. This helps people discover your package as it's listed in **npm search**.

5) homepage

The url to the project homepage.

Example:

"homepage": "https://github.com/owner/project#readme"

6) bugs

The url to your project's issue tracker and / or the email address to which issues should be reported. These are helpful for people who encounter issues with your package.

It should look like this:

{ "url" : "https://github.com/owner/project/issues"

, "email" : "project@hostname.com"

}

You can specify either one or both values. If you want to provide only a url, you can specify the value for "bugs" as a simple string instead of an object.

If a url is provided, it will be used by the **npm bugs** command.

7) license

You should specify a license for your package so that people know how they are permitted to use it, and any restrictions you're placing on it.

If you're using a common license such as BSD-2-Clause or MIT, add a current SPDX license identifier for the license you're using, like this:

```
{ "license" : "BSD-3-Clause" }
```

You can check the full list of SPDX license IDs. Ideally you should pick one that is OSI approved.

If your package is licensed under multiple common licenses, use an SPDX license expression syntax version 2.0 string, like this:

```
{ "license" : "(ISC OR GPL-3.0)" }
```

If you are using a license that hasn't been assigned an SPDX identifier, or if you are using a custom license, use a string value like this one:

```
{ "license" : "SEE LICENSE IN <filename>" }
```

Then include a file named **<filename>** at the top level of the package.

Some old packages used license objects or a "licenses" property containing an array of license objects:

```
// Not valid metadata

{ "license" :

  { "type" : "ISC"

  , "url" : "https://opensource.org/licenses/ISC"

  }

}

// Not valid metadata

{ "licenses" :

  [

    { "type": "MIT"

    , "url": "https://www.opensource.org/licenses/mit-license.php"

    }

  , { "type": "Apache-2.0"

    , "url": "https://opensource.org/licenses/apache2.0.php"

    }

  ]

}
```

Those styles are now deprecated. Instead, use SPDX expressions, like this:

```
{ "license": "ISC" }

{ "license": "(MIT OR Apache-2.0)" }
```

Finally, if you do not wish to grant others the right to use a private or unpublished package under any terms:

```
{ "license": "UNLICENSED" }
```

Consider also setting **"private": true** to prevent accidental publication.

8) people fields: author, contributors

The "author" is one person. "contributors" is an array of people. A "person" is an object with a "name" field and optionally "url" and "email", like this:

```
{ "name" : "Barney Rubble"

, "email" : "b@rubble.com"

, "url" : "http://barnyrubble.tumblr.com/"

}
```

Or you can shorten that all into a single string, and npm will parse it for you:

```
"Barney Rubble <b@rubble.com> (http://barnyrubble.tumblr.com/)"
```

Both email and url are optional either way.

npm also sets a top-level "maintainers" field with your npm user info.

9) files

The optional **files** field is an array of file patterns that describes the entries to be included when your package is installed as a dependency. File patterns follow a similar syntax to **.gitignore**, but reversed: including a file, directory, or glob pattern (*****, ****/***, and such) will make it so that file is included in the tarball when it's packed. Omitting the field will make it default to **["*"]**, which means it will include all files.
Some special files and directories are also included or excluded regardless of whether they exist in the **files** array (see below).
You can also provide a **.npmignore** file in the root of your package or in subdirectories, which will keep files from being included. At the root of your package it will not override the "files" field, but in subdirectories it will. The **.npmignore** file works just like a **.gitignore**. If there is a **.gitignore**file, and **.npmignore** is missing, **.gitignore**'s contents will be used instead.
Files included with the "package.json#files" field *cannot* be excluded through **.npmignore** or **.gitignore**.

Certain files are always included, regardless of settings:

- **package.json**
- **README**
- **CHANGES / CHANGELOG / HISTORY**
- **LICENSE / LICENCE**
- **NOTICE**
- The file in the "main" field

README, **CHANGES**, **LICENSE** & **NOTICE** can have any case and extension.

Conversely, some files are always ignored:

- **.git**
- **CVS**
- **.svn**
- **.hg**
- **.lock-wscript**
- **.wafpickle-N**
- **.*.swp**
- **.DS_Store**
- **._***
- **npm-debug.log**
- **.npmrc**
- **node_modules**
- **config.gypi**
- ***.orig**
- **package-lock.json** (use shrinkwrap instead)

10) main
The main field is a module ID that is the primary entry point to your program. That is, if your package is named **foo**, and a user installs it, and then does **require("foo")**, then your main module's exports object will be returned.

This should be a module ID relative to the root of your package folder.

For most modules, it makes the most sense to have a main script and often not much else.

11) browser

If your module is meant to be used client-side the browser field should be used instead of the main field. This is helpful to hint users that it might rely on primitives that aren't available in Node.js modules. (e.g. **window**)

12) bin

A lot of packages have one or more executable files that they'd like to install into the PATH. npm makes this pretty easy (in fact, it uses this feature to install the "npm" executable.)

To use this, supply a **bin** field in your package.json which is a map of command name to local file name. On install, npm will symlink that file into **prefix/bin** for global installs, or **./node_modules/.bin/** for local installs.

For example, myapp could have this:

```
{ "bin" : { "myapp" : "./cli.js" } }
```

So, when you install myapp, it'll create a symlink from the **cli.js** script to**/usr/local/bin/myapp**.

If you have a single executable, and its name should be the name of the package, then you can just supply it as a string. For example:

```
{ "name": "my-program"

, "version": "1.2.5"

, "bin": "./path/to/program" }
```

would be the same as this:

```
{ "name": "my-program"

, "version": "1.2.5"

, "bin" : { "my-program" : "./path/to/program" } }
```

Please make sure that your file(s) referenced in **bin** starts with **#!/usr/bin/env node**, otherwise the scripts are started without the node executable!

13) man

Specify either a single file or an array of filenames to put in place for the **man** program to find.

If only a single file is provided, then it's installed such that it is the result from **man <pkgname>**, regardless of its actual filename. For example:

```
{ "name" : "foo"

, "version" : "1.2.3"

, "description" : "A packaged foo fooer for fooing foos"

, "main" : "foo.js"

, "man" : "./man/doc.1"

}
```

would link the **./man/doc.1** file in such that it is the target for **man foo**

If the filename doesn't start with the package name, then it's prefixed. So, this:

```
{ "name" : "foo"

, "version" : "1.2.3"

, "description" : "A packaged foo fooer for fooing foos"

, "main" : "foo.js"

, "man" : [ "./man/foo.1", "./man/bar.1" ]

}
```

will create files to do **man foo** and **man foo-bar**.
Man files must end with a number, and optionally a **.gz** suffix if they are compressed. The number dictates which man section the file is installed into.

```
{ "name" : "foo"

, "version" : "1.2.3"

, "description" : "A packaged foo fooer for fooing foos"

, "main" : "foo.js"

, "man" : [ "./man/foo.1", "./man/foo.2" ]

}
```

will create entries for **man foo** and **man 2 foo.**

14) directories

The CommonJS Packages spec details a few ways that you can indicate the structure of your package using a **directories** object. If you look at npm's package.json, you'll see that it has directories for doc, lib, and man.

In the future, this information may be used in other creative ways.

15) directories.lib

Tell people where the bulk of your library is. Nothing special is done with the lib folder in any way, but it's useful meta info.

16) directories.bin

If you specify a **bin** directory in **directories.bin**, all the files in that folder will be added.
Because of the way the **bin** directive works, specifying both a **bin** path and setting **directories.bin** is an error. If you want to specify individual files, use **bin**, and for all the files in an existing **bin** directory, use **directories.bin**.

17) directories.man

A folder that is full of man pages. Sugar to generate a "man" array by walking the folder.

18) directories.doc

Put markdown files in here. Eventually, these will be displayed nicely, maybe, someday.

19) directories.example

Put example scripts in here. Someday, it might be exposed in some clever way.

20) directories.test

Put your tests in here. It is currently not exposed, but it might be in the future.

21) repository

Specify the place where your code lives. This is helpful for people who want to contribute. If the git repo is on GitHub, then the **npm docs** command will be able to find you.

Do it like this:

```
"repository": {

  "type" : "git",
  "url" : "https://github.com/npm/cli.git"

}
```

```
"repository": {

  "type" : "svn",

  "url" : "https://v8.googlecode.com/svn/trunk/"

}
```

The URL should be a publicly available (perhaps read-only) url that can be handed directly to a VCS program without any modification. It should not be a url to an html project page that you put in your browser. It's for computers.

For GitHub, GitHub gist, Bitbucket, or GitLab repositories you can use the same shortcut syntax you use for **npm install**:

```
"repository": "npm/npm"

"repository": "github:user/repo"

"repository": "gist:11081aaa281"

"repository": "bitbucket:user/repo"

"repository": "gitlab:user/repo"
```

22) scripts

The "scripts" property is a dictionary containing script commands that are run at various times in the lifecycle of your package. The key is the lifecycle event, and the value is the command to run at that point.

See **npm-scripts** to find out more about writing package scripts.

23) config

A "config" object can be used to set configuration parameters used in package scripts that persist across upgrades. For instance, if a package had the following:

```
{ "name" : "foo"
, "config" : { "port" : "8080" } }
```

and then had a "start" command that then referenced the **npm_package_config_port**environment variable, then the user could override that by doing **npm config set foo:port 8001**.
See **npm-config** and **npm-scripts** for more on package configs.

24) dependecies

Dependencies are specified in a simple object that maps a package name to a version range. The version range is a string which has one or more space-separated descriptors. Dependencies can also be identified with a tarball or git URL.

Please do not put test harnesses or transpilers in your dependencies object. See **devDependencies**, below.

- **version** Must match **version** exactly
- **>version** Must be greater than **version**
- **>=version** etc
- **<version**
- **<=version**
- **~version** "Approximately equivalent to version"
- **^version** "Compatible with version"
- **1.2.x** 1.2.0, 1.2.1, etc., but not 1.3.0
- **http://...** See 'URLs as Dependencies' below
- ***** Matches any version
- "" (just an empty string) Same as *****
- **version1 - version2** Same as **>=version1 <=version2**.
- **range1 || range2** Passes if either range1 or range2 are satisfied.
- **git...** See 'Git URLs as Dependencies' below
- **user/repo** See 'GitHub URLs' below
- **tag** A specific version tagged and published as **tag**
- **path/path/path** See Local Paths below

For example, these are all valid:

```
{ "dependencies" :

  { "foo" : "1.0.0 - 2.9999.9999"

  , "bar" : ">=1.0.2 <2.1.2"

  , "baz" : ">1.0.2 <=2.3.4"

  , "boo" : "2.0.1"

  , "qux" : "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0"

  , "asd" : "http://asdf.com/asdf.tar.gz"
  , "til" : "~1.2"

  , "elf" : "~1.2.3"

  , "two" : "2.x"

  , "thr" : "3.3.x"

  , "lat" : "latest"

  , "dyl" : "file:../dyl"

  }

}
```

URLs as Dependencies

You may specify a tarball URL in place of a version range.

This tarball will be downloaded and installed locally to your package at install time.

Git URLs as Dependencies

Git urls are of the form:

```
<protocol>://[<user>[:<password>]@]<hostname>[:<port>]
[:][/]<path>[#<commit-ish> | #semver:<semver>]
```

<protocol> is one of **git**, **git+ssh**, **git+http**, **git+https**, or **git+file**.
If **#<commit-ish>** is provided, it will be used to clone exactly that commit. If the commit-ish has the format **#semver:<semver>**, **<semver>** can be any valid semver range or exact version, and npm will look for any tags or refs matching that range in the remote repository, much as it would for a registry dependency. If neither **#<commit-ish>** or **#semver:<semver>** is specified, then **master** is used.

Examples:

```
git+ssh://git@github.com:npm/cli.git#v1.0.27

git+ssh://git@github.com:npm/cli#semver:^5.0

git+https://isaacs@github.com/npm/cli.git

git://github.com/npm/cli.git#v1.0.27
```

GitHub URLs

As of version 1.1.65, you can refer to GitHub urls as just “foo”: “user/foo-project”. Just as with git URLs, a **commit-ish** suffix can be included. For example:

```
{

  "name": "foo",

  "version": "0.0.0",

  "dependencies": {

    "express": "expressjs/express",

    "mocha": "mochajs/mocha#4727d357ea",

    "module": "user/repo#feature\/branch"

  }

}
```

Local Paths

As of version 2.0.0 you can provide a path to a local directory that contains a package. Local paths can be saved using **npm install -S** or **npm install --save**, using any of these forms:

```
../foo/bar

~/foo/bar

./foo/bar

/foo/bar
```

in which case they will be normalized to a relative path and added to your **package.json**. For example:

```
{

  "name": "baz",

  "dependencies": {

    "bar": "file:../foo/bar"
```

```
 }

}
```

This feature is helpful for local offline development and creating tests that require npm installing where you don't want to hit an external server, but should not be used when publishing packages to the public registry.

25) devDependecies

If someone is planning on downloading and using your module in their program, then they probably don't want or need to download and build the external test or documentation framework that you use.

In this case, it's best to map these additional items in a **devDependencies** object.
These things will be installed when doing **npm link** or **npm install** from the root of a package, and can be managed like any other npm configuration param. See **npm-config** for more on the topic.
For build steps that are not platform-specific, such as compiling CoffeeScript or other languages to JavaScript, use the **prepare** script to do this, and make the required package a devDependency.

For example:

```
{ "name": "ethopia-waza",

  "description": "a delightfully fruity coffee varietal",

  "version": "1.2.3",

  "devDependencies": {

    "coffee-script": "~1.6.3"

  },

  "scripts": {

    "prepare": "coffee -o lib/ -c src/waza.coffee"

  },

  "main": "lib/waza.js"

}
```

The **prepare** script will be run before publishing, so that users can consume the functionality without requiring them to compile it themselves. In dev mode (ie, locally running **npm install**), it'll run this script as well, so that you can test it easily.

26) peerDependencies
In some cases, you want to express the compatibility of your package with a host tool or library, while not necessarily doing a **require** of this host. This is usually referred to as a *plugin*. Notably, your module may be exposing a specific interface, expected and specified by the host documentation.

For example:

```
{

  "name": "tea-latte",

  "version": "1.3.5",

  "peerDependencies": {

    "tea": "2.x"

  }

}
```

This ensures your package **tea-latte** can be installed *along* with the second major version of the host package **tea** only. **npm install tea-latte** could possibly yield the following dependency graph:

```
├── tea-latte@1.3.5

└── tea@2.2.0
```

NOTE: npm versions 1 and 2 will automatically install peerDependencies if they are not explicitly depended upon higher in the dependency tree. In the next major version of npm (npm@3), this will no longer be the case. You will receive a warning that the peerDependency is not installed instead. The behavior in npms 1 & 2 was frequently confusing and could easily put you into dependency hell, a situation that npm is designed to avoid as much as possible.

Trying to install another plugin with a conflicting requirement will cause an error. For this reason, make sure your plugin requirement is as broad as possible, and not to lock it down to specific patch versions.

Assuming the host complies with semver, only changes in the host package's major version will break your plugin. Thus, if you've worked with every 1.x version of the host package, use **"^1.0"**or **"1.x"** to express this. If you depend on features introduced in 1.5.2, use **">= 1.5.2 < 2"**.

27) bundledDependencies

This defines an array of package names that will be bundled when publishing the package.

In cases where you need to preserve npm packages locally or have them available through a single file download, you can bundle the packages in a tarball file by specifying the package names in the **bundledDependencies** array and executing **npm pack**.

For example:

If we define a package.json like this:

```
{

  "name": "awesome-web-framework",

  "version": "1.0.0",

  "bundledDependencies": [

    "renderized", "super-streams"

  ]

}
```

we can obtain **awesome-web-framework-1.0.0.tgz** file by running **npm pack**. This file contains the dependencies **renderized** and **super-streams** which can be installed in a new project by executing **npm install awesome-web-framework-1.0.0.tgz**.
If this is spelled **"bundleDependencies"**, then that is also honored.

28) optionalDependencies
If a dependency can be used, but you would like npm to proceed if it cannot be found or fails to install, then you may put it in the **optionalDependencies** object. This is a map of package name to version or url, just like the **dependencies** object. The difference is that build failures do not cause installation to fail.

It is still your program's responsibility to handle the lack of the dependency. For example, something like this:

```
try {

  var foo = require('foo')

  var fooVersion = require('foo/package.json').version

} catch (er) {

  foo = null

}

if ( notGoodFooVersion(fooVersion) ) {

  foo = null

}

// .. then later in your program ..

if (foo) {

  foo.doFooThings()

}
```

Entries in **optionalDependencies** will override entries of the same name in **dependencies**, so it's usually best to only put in one place.

29) engines

You can specify the version of node that your stuff works on:

```
{ "engines" : { "node" : ">=0.10.3 <0.12" } }
```

And, like with dependencies, if you don't specify the version (or if you specify "*" as the version), then any version of node will do.

If you specify an "engines" field, then npm will require that "node" be somewhere on that list. If "engines" is omitted, then npm will just assume that it works on node.

You can also use the "engines" field to specify which versions of npm are capable of properly installing your program. For example:

```
{ "engines" : { "npm" : "~1.0.20" } }
```

Unless the user has set the **engine-strict** config flag, this field is advisory only and will only produce warnings when your package is installed as a dependency.

30) engineStrict

This feature was removed in npm 3.0.0

Prior to npm 3.0.0, this feature was used to treat this package as if the user had set **engine-strict**. It is no longer used.

31) os

You can specify which operating systems your module will run on:

```
"os" : [ "darwin", "linux" ]
```

You can also blacklist instead of whitelist operating systems, just prepend the blacklisted os with a '!':

```
"os" : [ "!win32" ]
```

The host operating system is determined by **process.platform**

It is allowed to both blacklist, and whitelist, although there isn't any good reason to do this.

32) cpu

If your code only runs on certain cpu architectures, you can specify which ones.

```
"cpu" : [ "x64", "ia32" ]
```

Like the **os** option, you can also blacklist architectures:

```
"cpu" : [ "!arm", "!mips" ]
```

The host architecture is determined by **process.arch**

33) private

If you set **"private": true** in your package.json, then npm will refuse to publish it.
This is a way to prevent accidental publication of private repositories. If you would like to ensure that a given package is only ever published to a specific registry (for example, an internal registry), then use the **publishConfig** dictionary described below to override the **registry** config param at publish-time.

34) publishConfig

This is a set of config values that will be used at publish-time. It's especially handy if you want to set the tag, registry or access, so that you can ensure that a given package is not tagged with "latest", published to the global public registry or that a scoped module is private by default.

Any config values can be overridden, but only "tag", "registry" and "access" probably matter for the purposes of publishing.

35) Default Values

npm will default some values based on package contents.

- **"scripts": {"start": "node server.js"}**
  If there is a **server.js** file in the root of your package, then npm will default the **start**command to **node server.js**.
- **"scripts":{"install": "node-gyp rebuild"}**
  If there is a **binding.gyp** file in the root of your package and you have not defined an **install** or **preinstall** script, npm will default the **install** command to compile using node-gyp.
- **"contributors": [...]**
  If there is an **AUTHORS** file in the root of your package, npm will treat each line as a **Name <email> (url)** format, where email and url are optional. Lines which start with a **#** or are blank, will be ignored.

B. What Socket IO is

Socket.IO is a library that enables real-time, bidirectional and event-based communication between the browser and the server. It consists of:

- a Node.js server: Source | API
- a Javascript client library for the browser (which can be also run from Node.js): Source | API

Its main features are:

1) Reliability

Connections are established even in the presence of:

- proxies and load balancers.
- personal firewall and antivirus software.

For this purpose, it relies on Engine.IO, which first establishes a long-polling connection, then tries to upgrade to better transports that are "tested" on the side, like WebSocket. Please see the Goals section for more information.

2) Auto-reconnection support

Unless instructed otherwise a disconnected client will try to reconnect forever, until the server is available again. Please see the available reconnection options here.

3) Disconnection detection

A heartbeat mechanism is implemented at the Engine.IO level, allowing both the server and the client to know when the other one is not responding anymore.

That functionality is achieved with timers set on both the server and the client, with timeout values (the pingInterval and pingTimeout parameters) shared during the connection handshake. Those timers require any subsequent client calls to be directed to the same server, hence the sticky-session requirement when using multiples nodes.

4) *Binary support*

Any serializable data structures can be emitted, including:

- ArrayBuffer and Blob in the browser
- ArrayBuffer and Buffer in Node.js

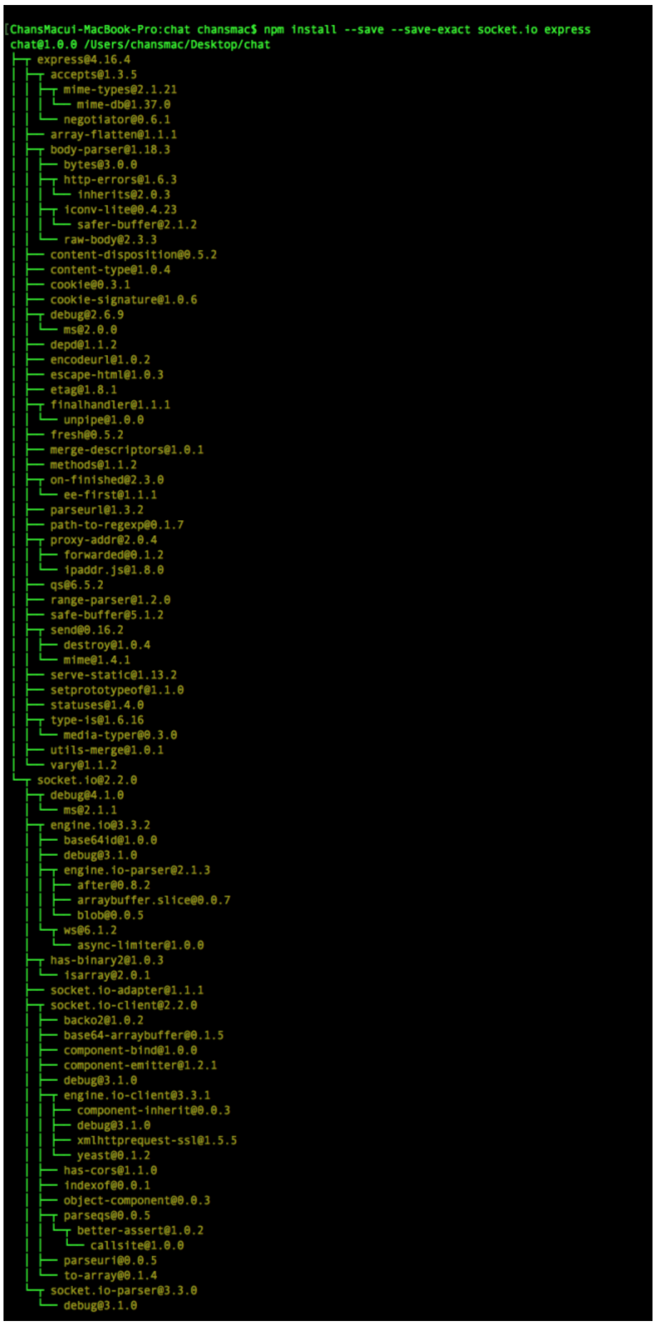
5) *Multiplexing support*

In order to create separation of concerns within your application (for example per module, or based on permissions), Socket.IO allows you to create several Namespaces, which will act as separate communication channels but will share the same underlying connection.

6) *Room support*

Within each Namespace, you can define arbitrary channels, called Rooms, that sockets can join and leave. You can then broadcast to any given room, reaching every socket that has joined it.

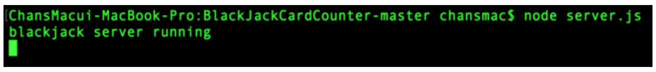
This is a useful feature to send notifications to a group of users, or to a given user connected on several devices for example.



```
[ChansMacui-MacBook-Pro:chat chansmac$ npm install --save --save-exact socket.io express
 chat@1.0.0 /Users/chansmac/Desktop/chat
 ├─┬ express@4.16.4
 │ ├─┬ accepts@1.3.5
 │ │ ├─┬ mime-types@2.1.21
 │ │ │ └── mime-db@1.37.0
 │ │ └── negotiator@0.6.1
 │ ├── array-flatten@1.1.1
 │ ├─┬ body-parser@1.18.3
 │ │ ├── bytes@3.0.0
 │ │ ├─┬ http-errors@1.6.3
 │ │ │ └── inherits@2.0.3
 │ │ ├─┬ iconv-lite@0.4.23
 │ │ │ └── safer-buffer@2.1.2
 │ │ └── raw-body@2.3.3
 │ ├── content-disposition@0.5.2
 │ ├── content-type@1.0.4
 │ ├── cookie@0.3.1
 │ ├── cookie-signature@1.0.6
 │ ├─┬ debug@2.6.9
 │ │ └── ms@2.0.0
 │ ├── depd@1.1.2
 │ ├── encodeurl@1.0.2
 │ ├── escape-html@1.0.3
 │ ├── etag@1.8.1
 │ ├─┬ finalhandler@1.1.1
 │ │ └── unpipe@1.0.0
 │ ├── fresh@0.5.2
 │ ├── merge-descriptors@1.0.1
 │ ├── methods@1.1.2
 │ ├─┬ on-finished@2.3.0
 │ │ └── ee-first@1.1.1
 │ ├── parseurl@1.3.2
 │ ├── path-to-regexp@0.1.7
 │ ├─┬ proxy-addr@2.0.4
 │ │ ├── forwarded@0.1.2
 │ │ └── ipaddr.js@1.8.0
 │ ├── qs@6.5.2
 │ ├── range-parser@1.2.0
 │ ├── safe-buffer@5.1.2
 │ ├─┬ send@0.16.2
 │ │ ├── destroy@1.0.4
 │ │ └── mime@1.4.1
 │ ├── serve-static@1.13.2
 │ ├── setprototypeof@1.1.0
 │ ├── statuses@1.4.0
 │ ├─┬ type-is@1.6.16
 │ │ └── media-typer@0.3.0
 │ ├── utils-merge@1.0.1
 │ └── vary@1.1.2
 └─┬ socket.io@2.2.0
   ├─┬ debug@4.1.0
   │ └── ms@2.1.1
   ├─┬ engine.io@3.3.2
   │ ├── base64id@1.0.0
   │ ├── debug@3.1.0
   │ ├─┬ engine.io-parser@2.1.3
   │ │ ├── after@0.8.2
   │ │ ├── arraybuffer.slice@0.0.7
   │ │ └── blob@0.0.5
   │ └─┬ ws@6.1.2
   │   └── async-limiter@1.0.0
   ├─┬ has-binary2@1.0.3
   │ └── isarray@2.0.1
   ├── socket.io-adapter@1.1.1
   ├─┬ socket.io-client@2.2.0
   │ ├── backo2@1.0.2
   │ ├── base64-arraybuffer@0.1.5
   │ ├── component-bind@1.0.0
   │ ├── component-emitter@1.2.1
   │ ├── debug@3.1.0
   │ ├─┬ engine.io-client@3.3.1
   │ │ ├── component-inherit@0.0.3
   │ │ ├── debug@3.1.0
   │ │ ├── xmlhttprequest-ssl@1.5.5
   │ │ └── yeast@0.1.2
   │ ├── has-cors@1.1.0
   │ ├── indexof@0.0.1
   │ ├── object-component@0.0.3
   │ ├─┬ parseqs@0.0.5
   │ │ └─┬ better-assert@1.0.2
   │ │   └── callsite@1.0.0
   │ ├── parseuri@0.0.5
   │ └── to-array@0.1.4
   └─┬ socket.io-parser@3.3.0
     └── debug@3.1.0
```

C. *Run the Server*

To run the server, the final command is **node server.js**



```
[ChansMacui-MacBook-Pro:BlackJackCardCounter-master chansmac$ node server.js
blackjack server running
█
```

VIII. Discussion

During the whole process of development of our program, we had many hard times. In the first time, we thought it would be a simple program to develop. However, as the time pass, we found out that it was not really simple as we had thought. The main problem was the lack of communication between developers. The miscommunication between front-end developers and back-end developers made big problems during the process, so we had to revise our program entirely. Also, the communication via Kakao and only just Github (in terms of the program) made us difficult to handle the whole process of development. So, we tried to meet in offline more often to solve the problem. Moreover, we did our communication via voice chat when the members are not in a same place. However, we did not get enough meeting in offline during the process, so the limitation of the communication existed. After the whole process of development, we realized that the offline meeting is crucial part of the development of software, even though we can handle it through online. The efficiency of the work speed and the efficiency of the communication were really low when working through online. We should have realized those facts much earlier, but if we think in positive way, maybe we can fix our problems in our own project next time because we learned from this semester. But we think that the obstacle that make us to not get meeting often was the distance among the location of each member's home. However, it might be the only excuse for ourselves, and we should have made a much more effort to break that obstacle.

In overall, we had really important semester in terms of our whole career or our life. We learned many things form this experience. Being a one team when developing software is not that easy as we thought and the communication and cooperating well among team members is the most difficult part of the whole process. From this precious experience, we will not make a same mistake in our career.