A Review of Iteration 1 of the betting smart contract

Overview

This is the first iteration of the betting smart contract for the VEX project. It is programmed in Rust and can be uploaded to the NEAR network. All development was done using the NEAR testnet. This first iteration just includes the smart contract with no DAO, FT or frontend connected. The contract allows users to make a bet, view matches and view bets. It also allows the contract owner to create a new match and finish a match.

How it works

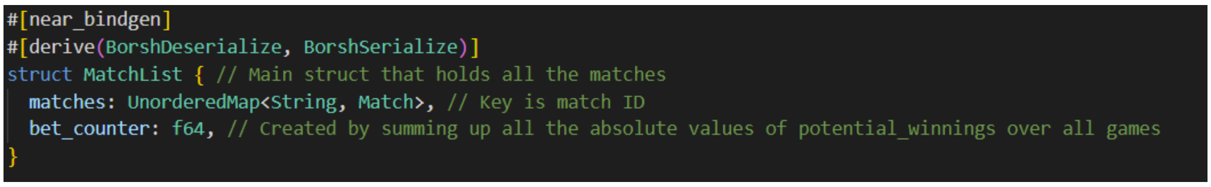
In this section, I will explain how the smart contract works.

Match List Struct

The MatchList Struct has two fields. The first field is matches which is an UnorderedMap that takes the match ID as a string as the key and an instance of the Match struct as its values. The second is bet_counter, which stores the total amount of potential pay out as an f64 number (this is a floating-point number meaning that it can store rational numbers). bet_counter is calculated by taking the sum of the absolute values of promised_winnings for each instance of Match that has not been finished. It is used to make sure that the contract will always be able to pay out all users in the case that the worst scenario happens. We want to get this number to as close to zero as possible.

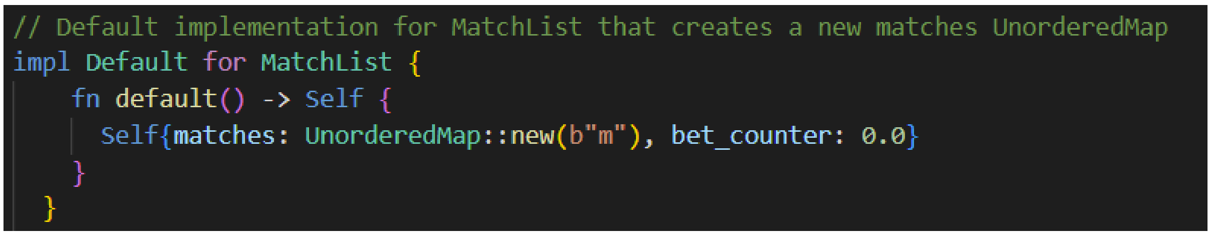
$$\text{bet\_counter} = \sum^{n} |\text{promised\_winnings}_n|$$

where n is ~~the~~ a valid match (not finished)



```rust
#[near_bindgen]
#[derive(BorshDeserialize, BorshSerialize)]
struct MatchList { // Main struct that holds all the matches
    matches: UnorderedMap<String, Match>, // Key is match ID
    bet_counter: f64, // Created by summing up all the absolute values of potential_winnings over all games
}
```

A default for MatchList is implemented so that when the contract is first used then a new empty UnorderedMap is created and the bet_counter is set to zero.



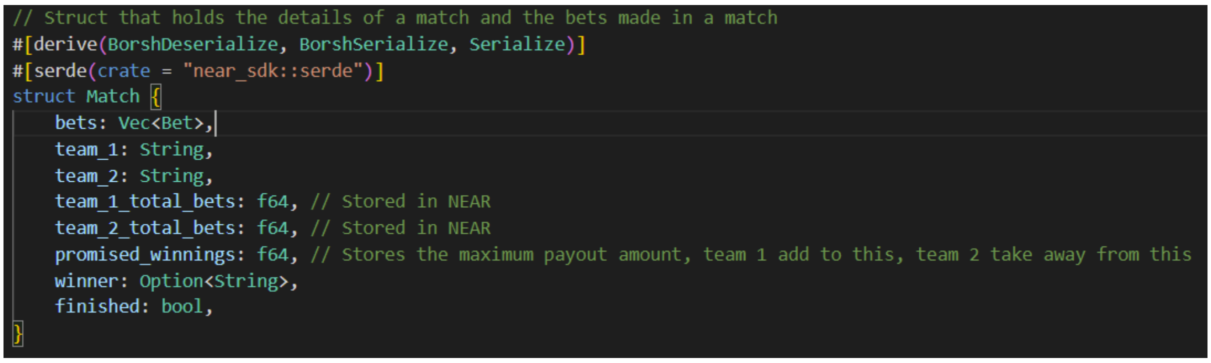
```rust
// Default implementation for MatchList that creates a new matches UnorderedMap
impl Default for MatchList {
    fn default() -> Self {
        Self{matches: UnorderedMap::new(b"m"), bet_counter: 0.0}
    }
}
```

Match struct

The Match struct has 8 fields. bets stores all the instances of the struct Bet for that match as a vector (a list). team_1 and team_2 stores the names of the teams as a String. team_1_total_bets and team_2_total_bets store the total amount of NEAR betted on each time respectively but this also includes the 'fake' bets added by the system at the start based on the initial odds as f64. promised_winnings stores the maxium pay out for the match, it is calculated by adding up all the potential_winnings for team 1 take away all the potential_winnings for team 2 so it can be positive or negative, this needs to be to as close to zero as possible. winner stores None when the match is valid and then the name of the winning team when the match is finished, stores this as Option<String>. finished stores whether the game has finished or not as a bool (true or false).

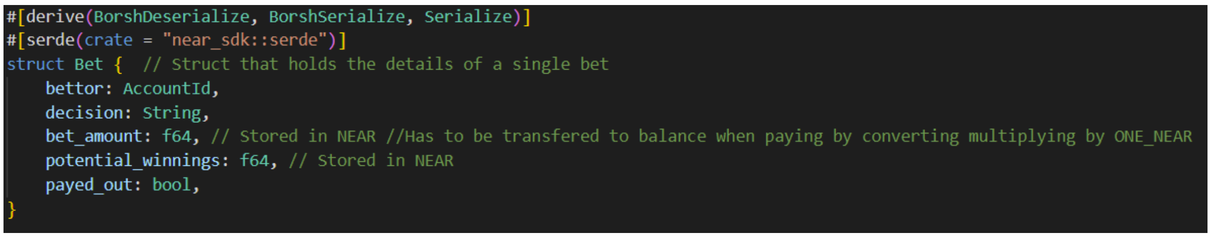
$$\text{promised\_winnings} = \sum^{n} \text{potential\_winnings}_n - \sum^{m} \text{potential\_winnings}_m$$

where n is all potential bets on team 1 and m for team 2.


```rust
// Struct that holds the details of a match and the bets made in a match
#[derive(BorshDeserialize, BorshSerialize, Serialize)]
#[serde(crate = "near_sdk::serde")]
struct Match {
    bets: Vec<Bet>,
    team_1: String,
    team_2: String,
    team_1_total_bets: f64, // Stored in NEAR
    team_2_total_bets: f64, // Stored in NEAR
    promised_winnings: f64, // Stores the maximum payout amount, team 1 add to this, team 2 take away from this
    winner: Option<String>,
    finished: bool,
}
```

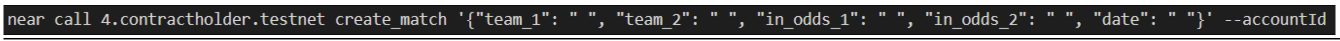
Bet Struct

The Bet struct has 5 fields. bettor stores the account id of the user making the bet as AccountId. decision stores which team they bet on as String. bet_amount stores how much the user bet in NEAR as f64. potential_winnings stores how much the user will get back if they choose the correct decision in NEAR as f64. payed_out stores whether the user has been payed or not as bool.


```rust
#[derive(BorshDeserialize, BorshSerialize, Serialize)]
#[serde(crate = "near_sdk::serde")]
struct Bet {  // Struct that holds the details of a single bet
    bettor: AccountId,
    decision: String,
    bet_amount: f64, // Stored in NEAR //Has to be transfered to balance when paying by converting multiplying by ONE_NEAR
    potential_winnings: f64, // Stored in NEAR
    payed_out: bool,
}
```

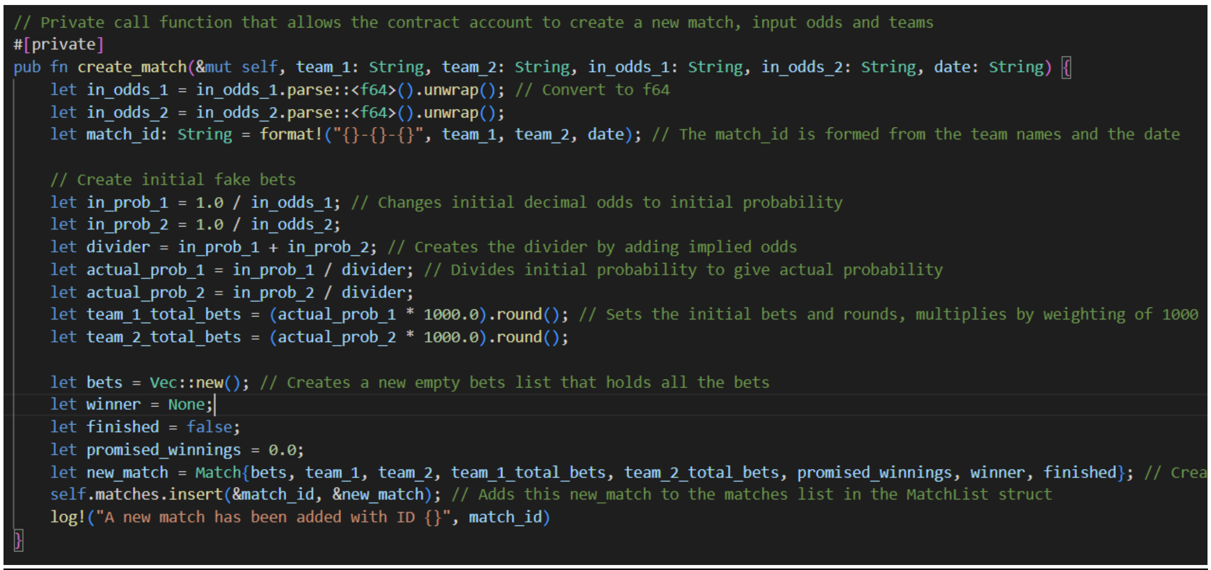
create_match private call function

create_match is a private call function meaning that when called the user calling the function has to sign it with the account ID of the account where the contract is stored and have the key to this account, this prevents just anyone creating a match. Being a call function gas is used and there has to be a signer. To create a match the user inputs the team names, the initial odds and the date of when the match will take place.



```
near call 4.contractholder.testnet create_match '{"team_1": " ", "team_2": " ", "in_odds_1": " ", "in_odds_2": " ", "date": " "}' --accountId
```

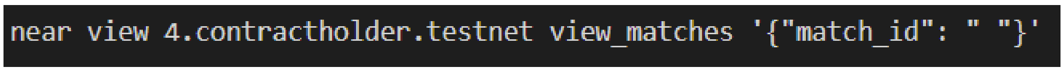
The odds are inputted as strings so first they are converted to f64. Next the match_id is created in the format team_1-team_2-date this is used as a unique identifier for the match and will be used as a key. Next the function takes the initial odds and finds the probability of each team winning (adding to 100%) and multiplies by a weighting constant which is set a 1000, this determines how elastic the odds will be to user bets (this is more prevalent when betting commences and fades out with more bets) this is then rounded to a whole number. A new vector of bets is created, finished is set to false, promised_winnings is set to 0 and a new match is created with the fields filled in and then inserted to the matches UnordedMap with the match_id as the key.



```rust
// Private call function that allows the contract account to create a new match, input odds and teams
#[private]
pub fn create_match(&mut self, team_1: String, team_2: String, in_odds_1: String, in_odds_2: String, date: String) {
    let in_odds_1 = in_odds_1.parse::<f64>().unwrap(); // Convert to f64
    let in_odds_2 = in_odds_2.parse::<f64>().unwrap();
    let match_id: String = format!("{}-{}-{}", team_1, team_2, date); // The match_id is formed from the team names and the date

    // Create initial fake bets
    let in_prob_1 = 1.0 / in_odds_1; // Changes initial decimal odds to initial probability
    let in_prob_2 = 1.0 / in_odds_2;
    let divider = in_prob_1 + in_prob_2; // Creates the divider by adding implied odds
    let actual_prob_1 = in_prob_1 / divider; // Divides initial probability to give actual probability
    let actual_prob_2 = in_prob_2 / divider;
    let team_1_total_bets = (actual_prob_1 * 1000.0).round(); // Sets the initial bets and rounds, multiplies by weighting of 1000
    let team_2_total_bets = (actual_prob_2 * 1000.0).round();

    let bets = Vec::new(); // Creates a new empty bets list that holds all the bets
    let winner = None;
    let finished = false;
    let promised_winnings = 0.0;
    let new_match = Match{bets, team_1, team_2, team_1_total_bets, team_2_total_bets, promised_winnings, winner, finished}; // Crea
    self.matches.insert(&match_id, &new_match); // Adds this new_match to the matches list in the MatchList struct
    log!("A new match has been added with ID {}", match_id)
}
```

view_matches public view function

view_matches is a public view function meaning that anyone can use this view function and no gas is needed. To use this function the user inputs the match_id of the match they want to see, or they can input "all" to view all the matches.

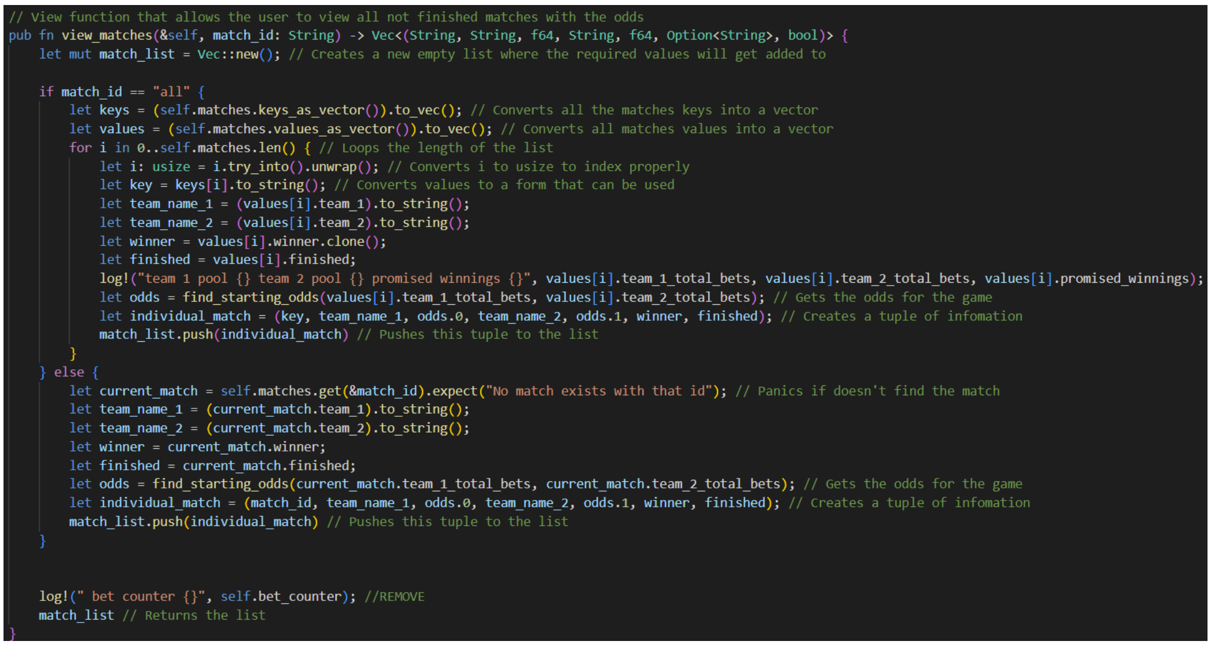


```
near view 4.contractholder.testnet view_matches '{"match_id": " "}'
```

If "all" is inputted then the keys and values are each separately converted to vectors. Then all the matches are looped through. The current odds for the game are found to 2sf by using the find_starting_odds function. For each match the key, team names, current odds, winner and finished fields are pushed onto a vector match_list.

If a match_id is inputted then just the information for that match is found and pushed to the match_list vector.

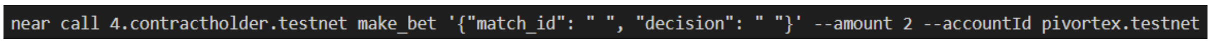
The match_list vector is then outputted.



```
// View function that allows the user to view all not finished matches with the odds
pub fn view_matches(&self, match_id: String) -> Vec<(String, String, f64, String, f64, Option<String>, bool)> {
    let mut match_list = Vec::new(); // Creates a new empty list where the required values will get added to

    if match_id == "all" {
        let keys = (self.matches.keys_as_vector()).to_vec(); // Converts all the matches keys into a vector
        let values = (self.matches.values_as_vector()).to_vec(); // Converts all matches values into a vector
        for i in 0..self.matches.len() { // Loops the length of the list
            let i: usize = i.try_into().unwrap(); // Converts i to usize to index properly
            let key = keys[i].to_string(); // Converts values to a form that can be used
            let team_name_1 = (values[i].team_1).to_string();
            let team_name_2 = (values[i].team_2).to_string();
            let winner = values[i].winner.clone();
            let finished = values[i].finished;
            log!("team 1 pool {} team 2 pool {} promised winnings {}", values[i].team_1_total_bets, values[i].team_2_total_bets, values[i].promised_winnings);
            let odds = find_starting_odds(values[i].team_1_total_bets, values[i].team_2_total_bets); // Gets the odds for the game
            let individual_match = (key, team_name_1, odds.0, team_name_2, odds.1, winner, finished); // Creates a tuple of infomation
            match_list.push(individual_match) // Pushes this tuple to the list
        }
    } else {
        let current_match = self.matches.get(&match_id).expect("No match exists with that id"); // Panics if doesn't find the match
        let team_name_1 = (current_match.team_1).to_string();
        let team_name_2 = (current_match.team_2).to_string();
        let winner = current_match.winner;
        let finished = current_match.finished;
        let odds = find_starting_odds(current_match.team_1_total_bets, current_match.team_2_total_bets); // Gets the odds for the game
        let individual_match = (match_id, team_name_1, odds.0, team_name_2, odds.1, winner, finished); // Creates a tuple of infomation
        match_list.push(individual_match) // Pushes this tuple to the list
    }


    log!(" bet counter {}", self.bet_counter); //REMOVE
    match_list // Returns the list
}
```

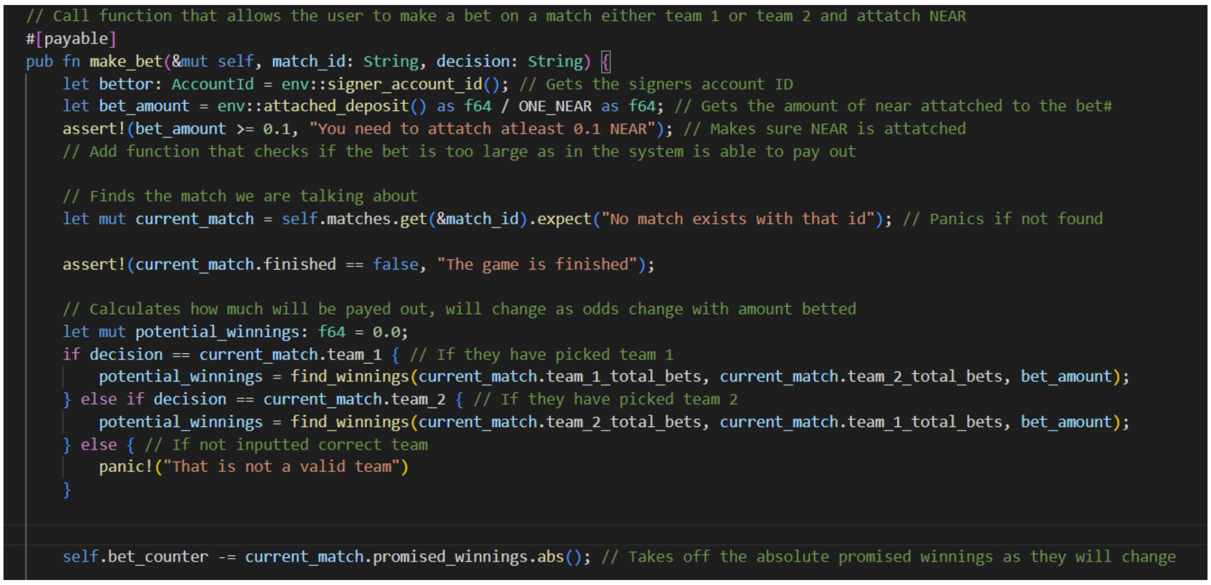
<u>make_bet public function</u>

To use this function the user inputs the match_id of the match they want to bet on, the team they want to bet on and they also need to attach an amount of NEAR.

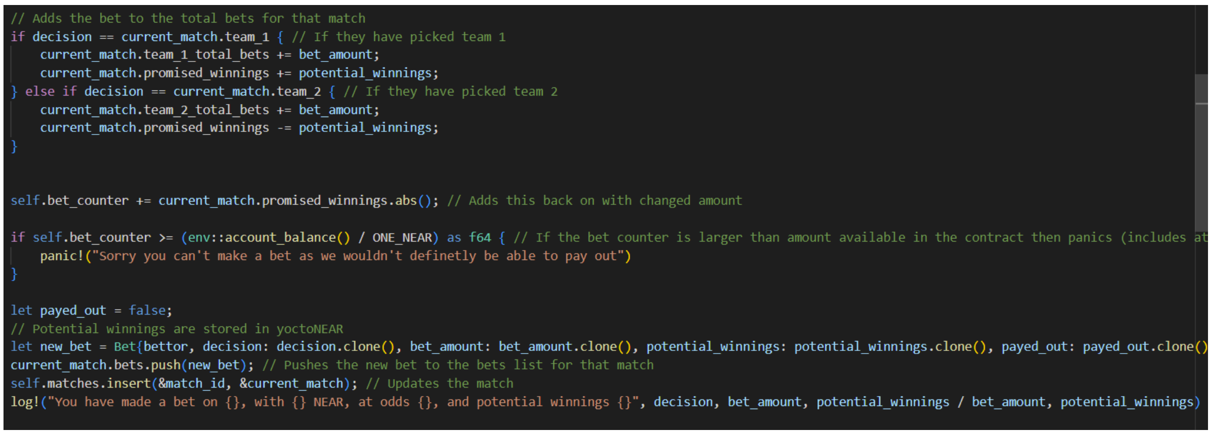


```
near call 4.contractholder.testnet make_bet '{"match_id": " ", "decision": " "}' --amount 2 --accountId pivortex.testnet
```

First the contract gets the bettors account ID and the bet amount. On the NEAR network all transaction are done in yocto NEAR where one NEAR is equal to 10^{24} yocto NEAR, so this amount is converted to f64 by dividing by this number. Next it is checked that the bettor has attached at least 0.1 NEAR. It checks that the user has inputted a valid match_id, and that the game hasn't been finished. When the code does these sort of checks, if it fails then the code will be ended, but these checks need to be done as soon as possible as to save as much gas as possible. Next the potential_winnings are decided by calling the find_winnings function, this has to be called differently depending on which team they chose. It also checks they haven't picked an invalid team. The absolute value of promised_winnings for this match is taken off the bet_counter, then if they chose team 1 the promised_winings is added to by the users potential_winnings, if its team 2 then potential_winnings is taken away from promised_winnings, the bet_counter is then added to by the absolute value of potential_winnings. The bet_counter is then checked against the account balance of the contract account, if it is larger than the account balance then the transaction will be cancelled. The users bet_amount is added to team_1 or team_2's total_bets pool so the odds will change for the next person betting. Payed out is set to false and then all the fields are filled out for a new bet and this is added to the matches bet list. This match is then put back onto the UnorderedMap of matches.



```
// Call function that allows the user to make a bet on a match either team 1 or team 2 and attatch NEAR
#[payable]
pub fn make_bet(&mut self, match_id: String, decision: String) {
    let bettor: AccountId = env::signer_account_id(); // Gets the signers account ID
    let bet_amount = env::attached_deposit() as f64 / ONE_NEAR as f64; // Gets the amount of near attatched to the bet#
    assert!(bet_amount >= 0.1, "You need to attatch atleast 0.1 NEAR"); // Makes sure NEAR is attatched
    // Add function that checks if the bet is too large as in the system is able to pay out

    // Finds the match we are talking about
    let mut current_match = self.matches.get(&match_id).expect("No match exists with that id"); // Panics if not found

    assert!(current_match.finished == false, "The game is finished");

    // Calculates how much will be payed out, will change as odds change with amount betted
    let mut potential_winnings: f64 = 0.0;
    if decision == current_match.team_1 { // If they have picked team 1
        potential_winnings = find_winnings(current_match.team_1_total_bets, current_match.team_2_total_bets, bet_amount);
    } else if decision == current_match.team_2 { // If they have picked team 2
        potential_winnings = find_winnings(current_match.team_2_total_bets, current_match.team_1_total_bets, bet_amount);
    } else { // If not inputted correct team
        panic!("That is not a valid team")
    }


    self.bet_counter -= current_match.promised_winnings.abs(); // Takes off the absolute promised winnings as they will change
```



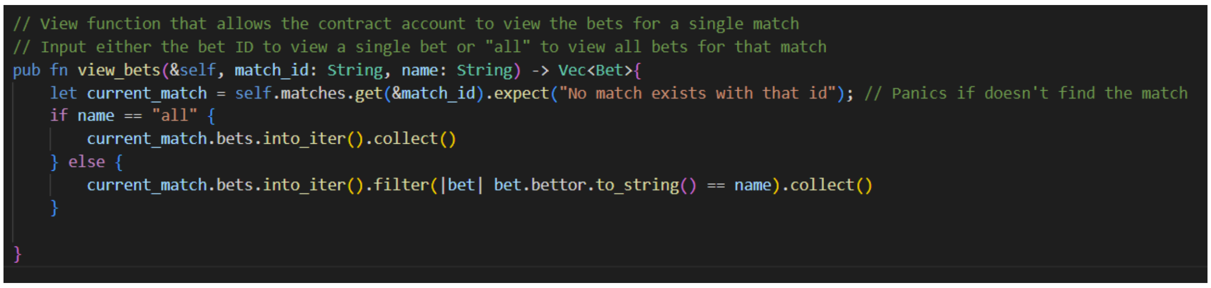
```
// Adds the bet to the total bets for that match
if decision == current_match.team_1 { // If they have picked team 1
    current_match.team_1_total_bets += bet_amount;
    current_match.promised_winnings += potential_winnings;
} else if decision == current_match.team_2 { // If they have picked team 2
    current_match.team_2_total_bets += bet_amount;
    current_match.promised_winnings -= potential_winnings;
}


self.bet_counter += current_match.promised_winnings.abs(); // Adds this back on with changed amount

if self.bet_counter >= (env::account_balance() / ONE_NEAR) as f64 { // If the bet counter is larger than amount available in the contract then panics (includes at
    panic!("Sorry you can't make a bet as we wouldn't definetly be able to pay out")
}

let payed_out = false;
// Potential winnings are stored in yoctoNEAR
let new_bet = Bet{bettor, decision: decision.clone(), bet_amount: bet_amount.clone(), potential_winnings: potential_winnings.clone(), payed_out: payed_out.clone()
current_match.bets.push(new_bet); // Pushes the new bet to the bets list for that match
self.matches.insert(&match_id, &current_match); // Updates the match
log!("You have made a bet on {}, with {} NEAR, at odds {}, and potential winnings {}", decision, bet_amount, potential_winnings / bet_amount, potential_winnings)
```

<u>view_bets public view function</u>

This should maybe be made private. The user inputs the match_id of the match they want to bet on and the name of the bettor they want to see the bets for. They can set the name to “all” to get all the bets for a match.

```
near view 4.contractholder.testnet view_bets '{"match_id": " ", "name": " "}'
```

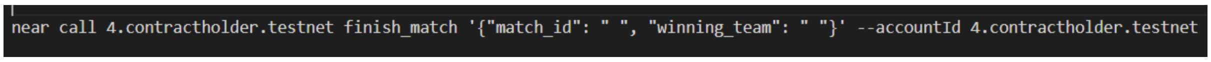
The code checks if the user has inputted a valid match_id. Then if they input an account ID then it collects all the bets where the bettor is equal to the name they inputted, if they input “all” then it will show all bets for that match.



```
// View function that allows the contract account to view the bets for a single match
// Input either the bet ID to view a single bet or "all" to view all bets for that match
pub fn view_bets(&self, match_id: String, name: String) -> Vec<Bet>{
    let current_match = self.matches.get(&match_id).expect("No match exists with that id"); // Panics if doesn't find the match
    if name == "all" {
        current_match.bets.into_iter().collect()
    } else {
        current_match.bets.into_iter().filter(|bet| bet.bettor.to_string() == name).collect()
    }

}
```

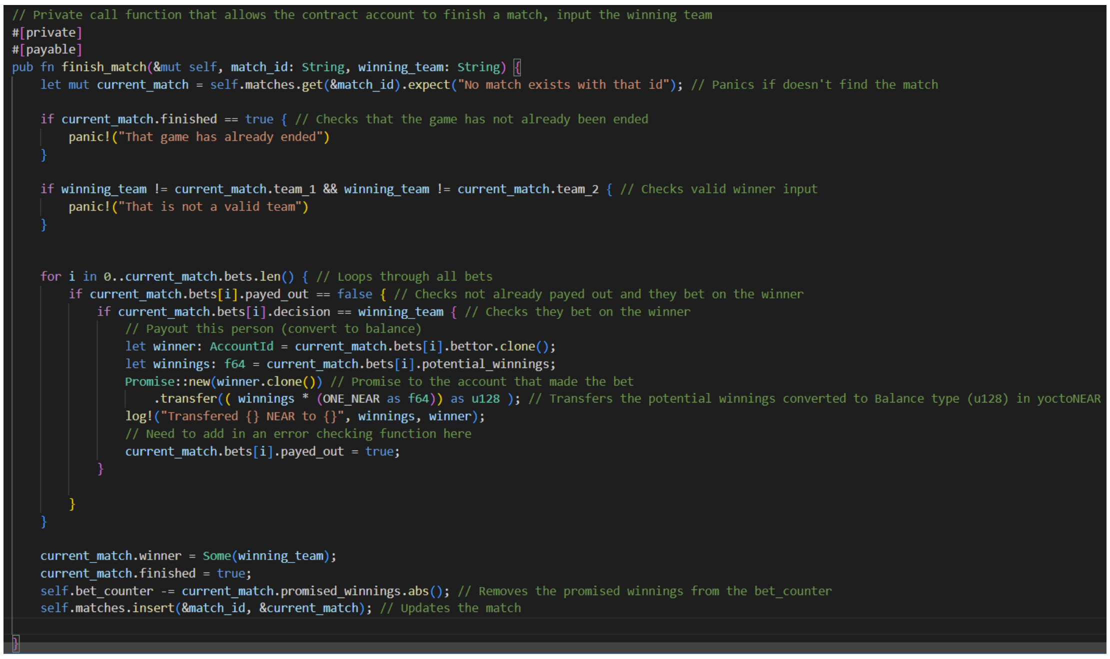
finish_match private call function

The contract holder signs this transaction and inputs the match_id and the winning_team.



```
near call 4.contractholder.testnet finish_match '{"match_id": " ", "winning_team": " "}' --accountId 4.contractholder.testnet
```

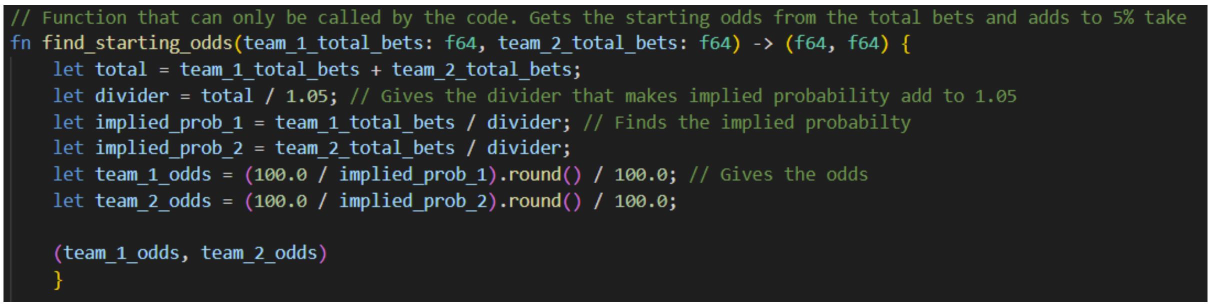
The function checks that the match exists, the match hasn't been finished and that one of the teams is equal to the inputted team for that match. The function then loops through all of the bets for that match that have payed_out equal to false and bet on the winning team. The code then transfers that bettor the potential_winnings for that bet (making sure to convert back to yocto NEAR) and changes payed_out to true. It then sets the winner of the match to the team that won and sets finished to true. It then takes the promised_winnings for this match away from the bet_counter. Then the matches UnorderedMap is updated for this match.



```
// Private call function that allows the contract account to finish a match, input the winning team
#[private]
#[payable]
pub fn finish_match(&mut self, match_id: String, winning_team: String) {
    let mut current_match = self.matches.get(&match_id).expect("No match exists with that id"); // Panics if doesn't find the match

    if current_match.finished == true { // Checks that the game has not already been ended
        panic!("That game has already ended")
    }

    if winning_team != current_match.team_1 && winning_team != current_match.team_2 { // Checks valid winner input
        panic!("That is not a valid team")
    }


    for i in 0..current_match.bets.len() { // Loops through all bets
        if current_match.bets[i].payed_out == false { // Checks not already payed out and they bet on the winner
            if current_match.bets[i].decision == winning_team { // Checks they bet on the winner
                // Payout this person (convert to balance)
                let winner: AccountId = current_match.bets[i].bettor.clone();
                let winnings: f64 = current_match.bets[i].potential_winnings;
                Promise::new(winner.clone()) // Promise to the account that made the bet
                    .transfer(( winnings * (ONE_NEAR as f64)) as u128 ); // Transfers the potential winnings converted to Balance type (u128) in yoctoNEAR
                log!("Transfered {} NEAR to {}", winnings, winner);
                // Need to add in an error checking function here
                current_match.bets[i].payed_out = true;
            }

        }
    }

    current_match.winner = Some(winning_team);
    current_match.finished = true;
    self.bet_counter -= current_match.promised_winnings.abs(); // Removes the promised winnings from the bet_counter
    self.matches.insert(&match_id, &current_match); // Updates the match

}
```

find_starting_odds private function

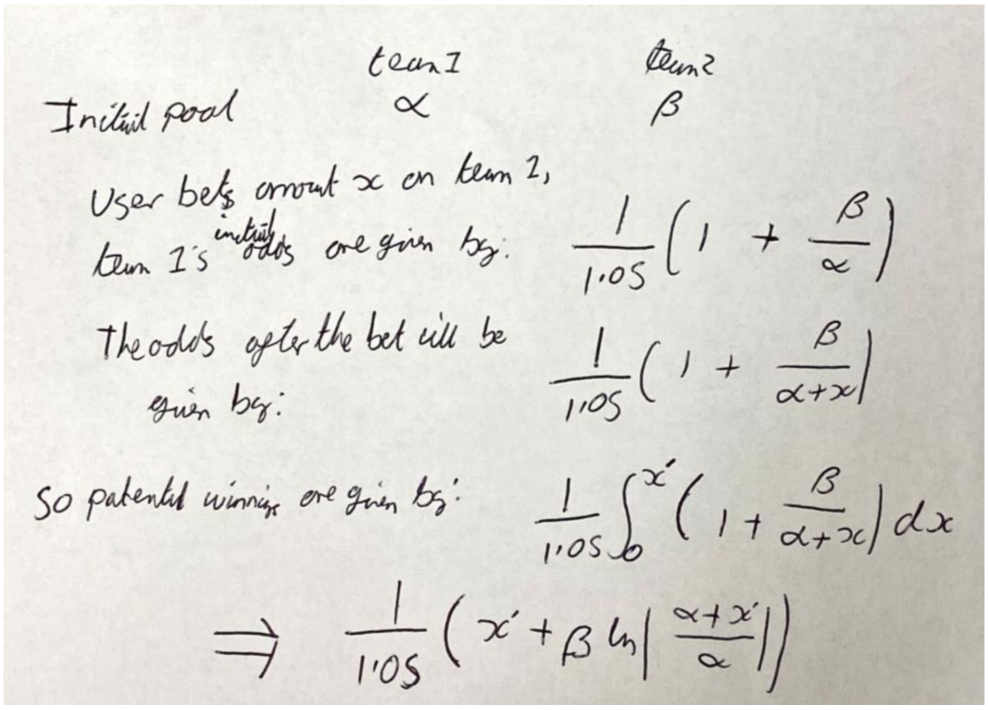
This function can only be called by the code itself, can't be accessed by users. This function takes the total bets for each team for a given match and finds the odds that add to 105% rounded to 2dp, this is for displaying odds purposes only.



```
// Function that can only be called by the code. Gets the starting odds from the total bets and adds to 5% take
fn find_starting_odds(team_1_total_bets: f64, team_2_total_bets: f64) -> (f64, f64) {
    let total = team_1_total_bets + team_2_total_bets;
    let divider = total / 1.05; // Gives the divider that makes implied probability add to 1.05
    let implied_prob_1 = team_1_total_bets / divider; // Finds the implied probabilty
    let implied_prob_2 = team_2_total_bets / divider;
    let team_1_odds = (100.0 / implied_prob_1).round() / 100.0; // Gives the odds
    let team_2_odds = (100.0 / implied_prob_2).round() / 100.0;

    (team_1_odds, team_2_odds)
    }
```

find_winnings private function

This function can only be called by the code itself. This function gives the potential_winnings for a bet. We want the odds to change as the user makes a larger bet. This function works by integrating over the odds calculating function by there bet. This can be divided by the bet amount to get the odds they are actually getting.



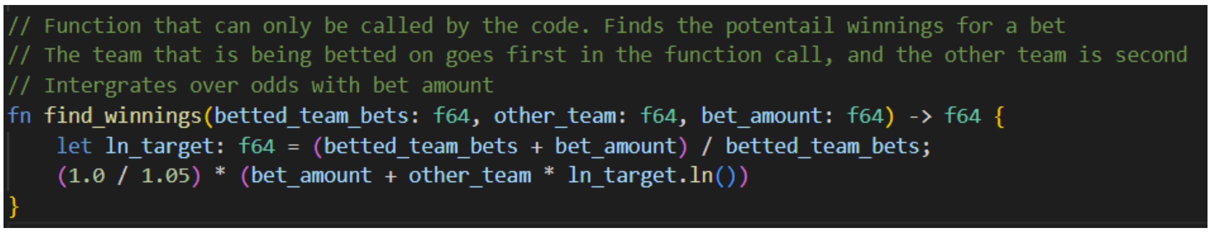
	team 1	team 2
Initial pool	α	β

User bets amount x on team 1,
team 1's initial odds are given by: $\frac{1}{1.05}\left(1 + \frac{\beta}{\alpha}\right)$

The odds after the bet will be given by: $\frac{1}{1.05}\left(1 + \frac{\beta}{\alpha + x}\right)$

So potential winnings are given by: $\frac{1}{1.05}\int_0^{x'}\left(1 + \frac{\beta}{\alpha + x}\right)dx$

$$\Rightarrow \frac{1}{1.05}\left(x' + \beta \ln\left|\frac{\alpha + x'}{\alpha}\right|\right)$$



```
// Function that can only be called by the code. Finds the potentail winnings for a bet
// The team that is being betted on goes first in the function call, and the other team is second
// Intergrates over odds with bet amount
fn find_winnings(betted_team_bets: f64, other_team: f64, bet_amount: f64) -> f64 {
    let ln_target: f64 = (betted_team_bets + bet_amount) / betted_team_bets;
    (1.0 / 1.05) * (bet_amount + other_team * ln_target.ln())
}
```

<u>What’s next?</u>

This contract I would say is near to what the final contract will look like. Additions for this contract that need to be implemented and possibly changed are:

- Additional checks to make sure all inputs are in the correct form and can’t be malicious.
- Adding a check to make sure bets are placed to no more than 2dp and that returns / odds are given to 2dp.
- A function needs to be implemented for the finish_match function that checks whether the payment was successful or not.
- Adding an oracle, this will replace the inputs for create_match and finish_match, but we might still want the option to use these if something was to go wrong with the oracle. The contract (more specifically these mentioned functions) may need to be adapted to work with the data outputted by the oracle.
- Changing the payment system so users can place bets in USDC.
- Making it so that the contract pays for the gas of a user making a bet.
- A function needs to be added that at a certain time interval, assuming that liquidity on the contract has increased, a certain amount of funds is transferred to another account that can be used by the DAO / given to stakers.
- The initial bet weighting constant (currently set at 1000) needs to be optimised. If it is high then it will make the odds less of a function of just user bets, but if it is low then the odds will be wild at the start rapidly going up or down. Also, consider the scenario where the weighting is low and a user makes a large bet when the betting has just opened, the odds will change quite a bit so they will get poor odds (assuming the odds started in line with how people actually view them).
- The minimum bet amount needs to be optimised. This needs to be set above the maximum potential gas of a transaction.
- Should there be a function that allows users to view bets? Make this private so that if there is an error and someone hasn’t been paid out that it can be done.
- Also links to all transactions, making a bet and paying out bets need to be collected and stored.
- How much control should the admin have to the contract, more means its more safe but also means it isn’t as much decentralised (not trustless).

What is next for the technical side of the project as a whole?

- The code should be adapted to allow for betting in different scenarios like if there is a draw case or betting on certain players. I think in the MVP just either team winning is enough for just CSGO. But add a function that just returns bets if something goes wrong with a game.
- A front end needs to be added for this contract using ReactJS, this will be outsourced. I will need to communicate with the person creating this a lot to get exactly what we want. I need to be easy to use and aesthetic. The programmer will need to use the call and view functions defined by the smart contract and know the outputs of the view functions. The front end will need to be constantly updated with new odds and matches in live time.
- A fungible ‘VEX’ token needs to be created.
- A DAO needs to be created that will allow normal functions like voting on policies. A staking function will need to be created where users stake VEX token to gain a certain percentage of commission from the contract. We need to get something out of the staking such as it provides

more liquidity for the contract to accept more bets. The DAO will probably be created with sputnik DAO.
- Make onboarding easy, have guest accounts, just pay in dollars, different currencies and so on.

Potential problems

This is a list of potential problems with the contract as of right now and the technical problems later down the line.

- Need to make sure users can't be malicious with or contract to charge us gas or get funds from the contract. An example of this is as we plan to pay for gas, users could spam the system with invalid bets costing us lots of gas. It could possibly be set up so that the smart contract can only be accessed via the front end so bets are made in a formatted way via the front end so no invalid bets could be entered. But I think there would still be a way to access the contract using the NEAR CLI. These potential weaknesses can be found by talking with other developers and having smart contract audits.
- A concern is that gas fees will be high. As the contract is storing lots of information that is constantly being updated gas will likely be high. Each time a bet is made, matches are viewed, bets are viewed then the whole bet list is deserialized (pulled off the blockchain and converted into readable data) this costs a lot of gas. The contract may have to be structured differently to decrease this cost, for example creating a new account or each match. Gas prices can be figured out with testing, but lots of data will have to be uploaded to the blockchain to test the case when there is a lot of bets made.
- As data transmission on the NEAR protocol is asynchronous a possible malicious attack is that lots of people come together to agree to make a bet on a specific team for one match at the same time. To my understanding, the contract will not be able to update the odds for the match until the block is completed (which is currently running at around 1.1 seconds per block) this means that users would get better odds then they should. This can be seen if it is a problem during testing, also a function could be implemented that only allows one bet per block, but this would make the system slow and a poor experience for the user.
- A concern is that bettors would start to bet a lot on one team for a given match. This would cause the promised_winnings to become large and so the bet_counter to become large especially if this happens across multiple matches this will become even larger. The concern of this is that the contract may have to pay out large funds and so a lot of money is lost if the matches turn out unfavourable for the contract, also if bet_counter was high it would mean that no more bets can be accepted for that one team this sort of keeps the odds within certain bounds. We hope that bet_counter will be low because of arbitrage betting, and we can see if this will be an issue with mass testing, but this will have to be testing done on a very large scale and by either actual people or a computer making reasonable human bets.