RabbitLeHole: Dive Deep Into Human Truth

What's This All About?

RabbitLeHole is a betting market that plunges into the depths of what people truly value. is a betting market to explore what people think matters most in their lives. Instead of just asking people to fill out boring surveys, we let them put their money where their mouth is! Using Solana's blockchain technology, people can bet on questions like:

- "Which would you choose - living in spacious beautiful house oceanside alone or in a city surrounded by your social circle?"
- "What's preferable - being middle class in a free society or wealthy in an authoritarian one?" (Minimum 10k wallet value to vote)
- "Which leads to more fulfillment - having five 2-year deep relationships or two 5-year ones?"
- "Would you rather have perfect self-control or complete spontaneity?"
- "Which brings more satisfaction - being the smartest person in the room or the most beloved?"
- "What's better - having an extraordinary talent that few appreciate or being moderately good at something everyone values?"
- "Which would you choose - knowing exactly when you'll die or living twice as long with uncertainty?" (Minimum 50k wallet value to vote)
- "What's preferable - being able to relive your past memories perfectly or forget all painful ones?"
- "Would you rather be the author of your life story or have it written by someone who understands you completely?"

Why This is Cool

When people bet real money on their answers, we can:

- See what people *think* influences their lives most
- Watch how groups of people agree (or disagree!) about what matters
- Learn how confident people are in their beliefs (based on how much they bet)
- Test whether common self-help advice matches what people actually believe

What We Can Learn

About Money & Choices

- How much are people willing to bet on their beliefs?
- Do early voters think differently than later ones?
- Does seeing others' bets change people's minds?

About Groups

- Do friends tend to bet the same way?
- How do communities form around shared beliefs?
- What makes people change their minds?

About Decision Making

- How sure are people about their own happiness factors?
- Does expertise in an area make people bet differently?
- Can we spot patterns in successful bets?

How It Works

We're using Solana (a super-fast blockchain) because it:

- Makes betting quick and easy
- Keeps costs low
- Records everything transparently
- Lets us collect data reliably

What Data We Collect

We keep track of:

- What people bet on
- When they make their bets
- How much they're willing to bet
- How groups of people tend to bet together

Keeping Things Safe & Fair

We promise to:

- Be clear about what data we're collecting
- Keep everyone's identity private
- Make sure everyone knows this is for research
- Help prevent problematic betting behavior

Technical details

Introduction

We will develop a decentralized application (dApp) on the Solana blockchain that allows users to participate in betting markets centered around psychologically significant questions. For example, users can vote on questions like "Which has more impact on my daily happiness - sleep or my commute?". Users place bets on their chosen option, and the winning side receives the losing side's stakes, distributed either evenly or following an exponential decay model where early voters receive a larger share of the winnings.

Table of Contents

1. Project Overview
2. Technical Stack
3. Architecture

4. Smart Contract Design
5. Frontend Application
6. Backend Services
7. File Structure and Mock Code
8. Security Considerations
9. Testing Plan
10. Deployment Strategy
11. Conclusion

Project Overview

Objectives

- **Create a user-friendly betting platform** that revolves around psychologically engaging questions.
- **Leverage Solana's high throughput and low transaction fees** to provide a seamless user experience.
- **Implement fair distribution mechanisms** for winnings, including both even distribution and exponential decay models.
- **Ensure transparency and security** by utilizing blockchain technology for all transactions and record-keeping.

Key Features

- Users can **submit new questions** for betting markets.
- Users can **place bets** on their preferred options.
- The platform **calculates winnings** based on the outcome and distribution model.
- **Early voters** can receive larger shares if the exponential decay model is selected.
- A **transparent and tamper-proof** system for managing bets and payouts.

Technical Stack

- **Blockchain Platform**: Solana
- **Smart Contract Language**: Rust
- **Frontend**: React.js with TypeScript
- **Backend**: Node.js with Express (if necessary for off-chain services)
- **Wallet Integration**: Solana wallets such as Phantom
- **State Management**: Redux or Context API
- **Data Storage**: On-chain for essential data; off-chain (e.g., Firebase or MongoDB) for non-critical data if needed
- **Testing Framework**: Mocha, Chai for JavaScript; Rust's built-in testing for smart contracts

---

Architecture


Architecture Diagram

Components

1. **Smart Contracts (Programs)**
    - Manages betting markets, user bets, and payout calculations.
2. **Frontend Application**
    - User interface for interacting with the dApp.
3. **Backend Services**
    - May handle non-critical operations like analytics or notifications.
4. **Wallet Integration**
    - Allows users to connect their Solana wallets for transactions.

Smart Contract Design

Overview

The smart contract will be responsible for:

- **Creating and managing betting markets** (questions).
- **Accepting and recording bets** from users.
- **Closing markets and determining winners**.
- **Calculating and distributing winnings** based on the selected distribution model.

Key Data Structures

- **Market**
    - Unique ID
    - Question text
    - Options (e.g., "Sleep", "Commute")
    - Status (Open, Closed)
    - Total stakes per option
    - Distribution model (Even, Exponential Decay)
    - Creation and closing timestamps
- **Bet**
    - Market ID
    - User's wallet address
    - Selected option
    - Amount staked
    - Timestamp

Frontend Application

Features

- **Market List View**
    - Displays all open and closed markets.
- **Market Detail View**
    - Shows question details, options, and betting interface.
- **Betting Interface**
    - Allows users to place bets.

- **Wallet Integration**
  - Users can connect their Solana wallets.
- **Results and Winnings**
  - Displays outcomes and users' winnings.

Backend Services

Though the core functionality is on-chain, we may need backend services for:

- **Notifications**
  - Email or push notifications for market closures or winnings.
- **Analytics**
  - Aggregate data for user engagement metrics.

File Structure and Mock Code

Smart Contracts (Solana Programs)

1. `programs/psych_betting/src/lib.rs`

```
//! Main entry point for the betting market program.

use solana_program::{
    account_info::AccountInfo,
    entrypoint,
    entrypoint::ProgramResult,
    pubkey::Pubkey,
    msg,
};

pub mod processor;
pub mod instruction;
pub mod state;
pub mod error;

entrypoint!(process_instruction);

/// Processes instructions received by the program.
fn process_instruction(
    program_id: &Pubkey,         // Program ID
    accounts: &[AccountInfo],    // Accounts involved in the transaction
    instruction_data: &[u8],     // Serialized instruction data
) -> ProgramResult {
    msg!("Processing instruction");
    processor::process(program_id, accounts, instruction_data)
}
```

2. `programs/psych_betting/src/processor.rs`

```
//! Processor module for handling instructions.

use solana_program::{
    account_info::AccountInfo,
    pubkey::Pubkey,
    program_error::ProgramError,
};

use crate::{instruction::PsychBettingInstruction, state::Market};

/// Processes all instructions for the program.
pub struct Processor;

impl Processor {
    /// Entry point for processing instructions.
    pub fn process(
        program_id: &Pubkey,
        accounts: &[AccountInfo],
        instruction_data: &[u8],
    ) -> ProgramResult {
        // Deserialize instruction data
        let instruction =
PsychBettingInstruction::unpack(instruction_data)?;

        match instruction {
            PsychBettingInstruction::CreateMarket { question, options,
distribution_model } => {
                Self::process_create_market(accounts, question, options,
distribution_model, program_id)
            },
            PsychBettingInstruction::PlaceBet { market_id, option, amount
} => {
                Self::process_place_bet(accounts, market_id, option,
amount, program_id)
            },
            PsychBettingInstruction::CloseMarket { market_id,
winning_option } => {
                Self::process_close_market(accounts, market_id,
winning_option, program_id)
            },
        }
    }

    /// Processes the creation of a new betting market.
    fn process_create_market(
        accounts: &[AccountInfo],
        question: String,
        options: Vec<String>,
        distribution_model: u8,
        program_id: &Pubkey,
    ) -> ProgramResult {
        // Implementation goes here...
```

```
        Ok(())
    }

    /// Processes placing a bet on a market.
    fn process_place_bet(
        accounts: &[AccountInfo],
        market_id: u64,
        option: String,
        amount: u64,
        program_id: &Pubkey,
    ) -> ProgramResult {
        // Implementation goes here...
        Ok(())
    }

    /// Processes closing a market and distributing winnings.
    fn process_close_market(
        accounts: &[AccountInfo],
        market_id: u64,
        winning_option: String,
        program_id: &Pubkey,
    ) -> ProgramResult {
        // Implementation goes here...
        Ok(())
    }
}
```

3. **programs/psych_betting/src/instruction.rs**

```
//! Defines the instructions supported by the program.

use solana_program::program_error::ProgramError;

/// Instruction definitions for the betting market program.
pub enum PsychBettingInstruction {
    /// Creates a new betting market.
    ///
    /// Accounts:
    /// - [Signer] Creator's account.
    CreateMarket { question: String, options: Vec<String>,
distribution_model: u8 },

    /// Places a bet on an existing market.
    ///
    /// Accounts:
    /// - [Signer] Bettor's account.
    PlaceBet { market_id: u64, option: String, amount: u64 },

    /// Closes a market and determines the winning option.
    ///
    /// Accounts:
```

```
    /// - [Signer] Market creator's account.
    CloseMarket { market_id: u64, winning_option: String },
}

impl PsychBettingInstruction {
    /// Unpacks a byte buffer into a [PsychBettingInstruction]
(enum.PsychBettingInstruction.html).
    pub fn unpack(input: &[u8]) -> Result<Self, ProgramError> {
        // Deserialize instruction data
        Ok(Self::CreateMarket {
            question: String::from("Example question"),
            options: vec![String::from("Option A"), String::from("Option
B")],
            distribution_model: 0,
        })
    }
}
```

4. `programs/psych_betting/src/state.rs`

```
//! Defines the data structures stored on-chain.

use solana_program::pubkey::Pubkey;

/// Represents a betting market.
pub struct Market {
    /// Unique identifier for the market.
    pub market_id: u64,
    /// The question posed in the market.
    pub question: String,
    /// Available options to bet on.
    pub options: Vec<String>,
    /// The creator of the market.
    pub creator: Pubkey,
    /// Current status of the market (e.g., Open, Closed).
    pub status: u8,
    /// Total stakes placed on each option.
    pub total_stakes: Vec<u64>,
    /// Distribution model (0 for Even, 1 for Exponential Decay).
    pub distribution_model: u8,
    /// Timestamps for creation and closing.
    pub timestamps: (u64, Option<u64>),
}
```

Frontend Application

1. `src/App.tsx`

```
// Main application component.

import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import { MarketsList } from './components/MarketsList';
import { MarketDetail } from './components/MarketDetail';
import { Header } from './components/Header';

const App: React.FC = () => {
  return (
    <Router>
      <Header />
      <Switch>
        <Route exact path="/" component={MarketsList} />
        <Route path="/market/:id" component={MarketDetail} />
      </Switch>
    </Router>
  );
};

export default App;
```

2. `src/components/MarketsList.tsx`

```
// Component to display a list of all betting markets.

import React, { useEffect, useState } from 'react';
import { MarketCard } from './MarketCard';

export const MarketsList: React.FC = () => {
  const [markets, setMarkets] = useState([]);

  useEffect(() => {
    // Fetch list of markets from the blockchain or an API
  }, []);

  return (
    <div>
      <h2>Open Markets</h2>
      {markets.map((market) => (
        <MarketCard key={market.marketId} market={market} />
      ))}
    </div>
  );
};
```

3. `src/components/MarketDetail.tsx`

```
// Component to display details of a single market and allow betting.

import React, { useEffect, useState } from 'react';
import { useParams } from 'react-router-dom';
import { placeBet } from '../utils/transactions';

export const MarketDetail: React.FC = () => {
  const { id } = useParams<{ id: string }>();
  const [market, setMarket] = useState(null);
  const [selectedOption, setSelectedOption] = useState('');
  const [betAmount, setBetAmount] = useState(0);

  useEffect(() => {
    // Fetch market details using the market ID
  }, [id]);

  const handleBet = () => {
    // Calls the placeBet function to place a bet on the blockchain
    placeBet(market.marketId, selectedOption, betAmount);
  };

  return (
    <div>
      {market && (
        <>
          <h2>{market.question}</h2>
          <div>
            {market.options.map((option) => (
              <button key={option} onClick={() =>
setSelectedOption(option)}>
                {option}
              </button>
            ))}
          </div>
          <input
            type="number"
            value={betAmount}
            onChange={(e) => setBetAmount(Number(e.target.value))}
          />
          <button onClick={handleBet}>Place Bet</button>
        </>
      )}
    </div>
  );
};
```

4. `src/utils/transactions.ts`

```
// Utility functions for interacting with the Solana blockchain.
```

```
import { Connection, PublicKey, Transaction } from '@solana/web3.js';

export const placeBet = async (marketId: number, option: string, amount:
number) => {
  // Connect to wallet and send transaction to place a bet
};
```

Backend Services (Optional)

1. `backend/server.js`

```
// Backend server for handling optional services like notifications.

const express = require('express');
const app = express();
const port = process.env.PORT || 3000;

// Middleware and routes would be set up here

app.listen(port, () => {
  console.log(`Backend server is running on port ${port}`);
});
```

Security Considerations

- **Input Validation**
  - Ensure all user inputs are validated and sanitized both on the client and smart contract level.
- **Access Control**
  - Restrict functions like closing a market to authorized accounts (e.g., market creator).
- **Reentrancy**
  - Although not common in Solana due to the single-threaded runtime, ensure state changes happen before external calls.
- **Overflow/Underflow**
  - Use Rust's safe arithmetic operations to prevent numeric overflows or underflows.

Testing Plan

Smart Contracts

- **Unit Tests**
  - Test each instruction handler (create market, place bet, close market).
- **Integration Tests**
  - Simulate user interactions and ensure state changes are as expected.
- **Edge Cases**
  - Test with maximum and minimum values, and invalid inputs.

Frontend

- **Component Tests**
  - Test UI components render correctly with various states.
- **Functional Tests**
  - Simulate user actions like placing bets, connecting wallets.

Deployment Strategy

Smart Contracts

- **Devnet Deployment**
  - Deploy the program to Solana's Devnet for initial testing.
- **Mainnet Deployment**
  - Once thoroughly tested, deploy to Solana Mainnet Beta.

Frontend

- **Hosting**
  - Use services like Netlify or Vercel for hosting.
- **Environment Variables**
  - Configure the app to connect to the appropriate RPC endpoints.

Conclusion

This detailed plan outlines the development of a Solana-based betting dApp focusing on psychologically significant questions. The project leverages Solana's speed and low fees to provide an engaging and fair betting platform. With careful planning of the architecture, thorough documentation, and emphasis on security and testing, this dApp aims to deliver a robust and user-friendly experience.

Next Steps

- **Set Up Development Environment**
  - Install necessary tools for Solana and React development.
- **Implement Smart Contracts**
  - Begin coding the smart contracts as per the outlined structure.
- **Develop Frontend Components**
  - Build out the React components and integrate with wallet providers.
- **Continuous Testing**
  - Write tests alongside development to ensure code quality.
- **Deploy to Devnet**
  - Regularly deploy to Devnet for integration testing.
- **Gather Feedback**
  - Share with a small group of users or testers to get early feedback.

Let me know if you'd like to explore any section in more detail or have any specific questions! """