Blackjack

The following documentation goes through an example implementation of the popular casino game blackjack on Sui. This guide walks through its components, providing a detailed look at the module's functions, structures, constants, and their significance in the overall gameplay mechanism.

A deployed version of the blackjack game is online at Mysten Blackjack .

Building an on-chain blackjack game shares a lot of similarities with a coin flip game. This example covers the smart contracts (Move modules), backend logic (using serverless functions), and frontend logic.

For more details on building a backend and deploying to Sui, check out the Coin Flip app example .

You can also find the full repository for this example here .

In this single-player version of blackjack, the player competes against a dealer, which is automated by the system. The dealer is equipped with a public BLS key that plays a central role in the game's mechanics. The dealer's actions are triggered by HTTP requests to serverless functions. Players generate randomness for the game by interacting with their mouse on the screen, after which they place their bet to start the game. Upon initiating the game, a request is made to the backend (dealer), which processes it by signing and subsequently dealing two cards to the player and one to themselves.

The player has the option to 'Hit' or 'Stand.' Selecting 'Stand' triggers the dealer to draw cards until the total reaches 17 or higher. After the dealer stops, the smart contract steps in to compare the totals and declare the winner. On the other hand, choosing 'Hit' prompts the dealer to draw an additional card for the player.

Note that more complex blackjack rules, such as splitting, are considered out of scope for this example, and are therefore not implemented.

The single_player_blackjack.move module includes several constants that define game statuses and help track the game's progress:

There are also constants for error handling, such as EInvalidBlsSig , EInsufficientBalance , and others, ensuring robust game mechanics.

Structs like GameCreatedEvent , GameOutcomeEvent , and HitDoneEvent capture the various events and actions within a game. The HitRequest and StandRequest structs ensure that a move (hit/stand) can be performed by the house only if the player has already requested it. HouseAdminCap and HouseData are crucial for maintaining the house's data, including balance and public key, while the Game struct contains all the necessary information about each game, such as player data, cards, and the current status.

The module's functions can be broadly categorized into initialization, game management, and utility functions. The init function sets up the house admin capability, while initialize_house_data prepares the house for the game by setting up the balance and public key. place_bet_and_create_game is the entry point for players to start a new game, involving a bet and random input. The functions first_deal , hit , and stand govern the core gameplay, handling the dealing of cards and player choices.

Utility functions like get_next_random_card and get_card_sum are essential for the game's mechanics, generating random cards and calculating hand values. The module also includes accessors for retrieving various pieces of game and house data.

For testing purposes, the module provides special functions like get_house_admin_cap_for_testing , player_won_post_handling_for_test and house_won_post_handling_for_test , ensuring that developers can thoroughly test the game mechanics and house data handling.

The next module, counter_nft.move , introduces the Counter NFT, a key component in the game's mechanics. It serves as the verifiable random function (VRF) input, ensuring uniqueness in each game. The Counter NFT's value increases after each use, maintaining its distinctiveness for every new request. For first-time players, the creation of a Counter NFT is mandatory. To enhance user experience, the user interface can automate this process by integrating the Counter NFT's creation with the game initiation in a single transaction block. This seamless integration simplifies the process for the player, allowing them to focus on the gameplay. This counter serves the same purpose as the one in the Coin Flip example .

The backend is used for all the transactions that are executed by the dealer. The backend can be completely stateless, and for that reason serverless functions are utilized. As a result, the corresponding code lies under the app/src/app/api/ directory .

The backend code is split in the following sub directories:

An interesting aspect of developing a dApp on Sui, that is coupled to using a full node with/without a load balancer, and requires

attention by the developer, is the occurrence of read-after-write and write-after-write cases.

As an example, in the blackjack game, just after the create-game transaction that the user executes, the dealer executes the initial-deal transaction. This one accepts an argument and modifies the game object, meaning that you are using an object that was just created.

To ensure that the game object is available in the Full node that the dealer is using, we need to call waitForTransaction after the create-game transaction.

In the same way, every time you re-fetch the game object in the frontend, make sure that the previous transaction that modified the game object is already available in the Full node.

This leads to the need of exchanging the txDigest between the frontend and the backend, and use waitForTransaction on each write-after-write or read-after-write case.

The page component , a central element of the blackjack game's frontend module, is structured to create an interactive and responsive gaming experience. Written in React, it integrates several features and functions to handle the game's logic and user interactions effectively.

The frontend is a NextJS project, that follows the NextJS App Router project structure . The main code of the frontend lies under the app/src/ directory. The main sub-directories are:

Custom Hooks: To keep the code as structured as possible, multiple custom hooks are utilized to manage the complex state of the game board at each step. The useBlackjackGame custom hook encapsulates the game state and logic, exposing all the required information (with fields such as game , isInitialDealLoading), and the required functionality (with methods such as handleCreateGame and handleHit) to display and play the game. Multiple additional custom hooks, such as useCreateBlackjackGame , and useMakeMoveInBlackjackGame are encapsulating their own piece of state and logic to make the code readable and maintainable.

Component for Game Initialization: The StartGame component is implemented to facilitate the creation of a new game. It renders the CollectMouseRandomness to capture the randomness and uses the handleCreateGame function of the useBlackjackGame hook to execute the create-game transaction.

Randomness Generation: Fair game outcomes are also ensured by the CollectMouseRandomness component. This component is using the useMouseRandomness custom hook, and is in charge of capturing some user's mouse movements and generating a random bytes array. This array is converted to a hexadecimal string (randomness) and used in the create-game transaction.

Card Displaying and Management: The DealerCards and the PlayerCards components are used to display the total points and the cards owned by the dealer and the player respectively.

Game Actions: The GameActions component is used to display the Hit and Stand buttons, and trigger the corresponding actions, as they are exported by the useBlackjackGame hook to execute the corresponding transactions.

BlackjackBanner : The BlackjackBanner component is used as a custom view to display when the player wins with a blackjack.

Blockchain-Based Logic: Both games are built on Sui, leveraging its capabilities for decentralized applications. The core game logic for each resides in Move modules, ensuring secure and verifiable gameplay.

State Management: In both games, state management is crucial. For blackjack, this involves managing the player and dealer's hands and scores using React state hooks. In Satoshi Coin Flip, the state is managed through Move structs like HouseData , which track the house's balance and other game-related details.

Randomness and Fair Play: Both games emphasize randomness for fairness. Blackjack uses a Counter NFT and player mouse movements to generate randomness, while Satoshi Coin Flip only uses a Counter NFT as a unique input for the Verifiable Random Function (VRF) in each game.

Smart Contract Interactions: Each game involves smart contract interactions for game actions like placing bets, dealing cards (Blackjack), or making guesses (Coin Flip). These interactions are crucial for executing the game's logic on the blockchain.

Game Mechanics and Complexity: Blackjack is a more complex game with multiple actions (hit, stand, deal) and state updates, requiring a more dynamic frontend. In contrast, Satoshi Coin Flip has a simpler mechanic centered around a single bet and guess outcome.

User Interface (UI) Complexity: The Blackjack game involves a more intricate UI to display cards, manage game states, and player

interactions. Satoshi Coin Flip, being simpler in gameplay, requires a less complex UI.

Backend Processing: In Blackjack, the dealer is automated (the machine), and the player's actions directly influence game outcomes. In the Coin Flip game, the house (smart contract) plays a more passive role, primarily in initializing and finalizing the game based on the player's guess.

Module Structure and Focus: The Blackjack game focuses more on frontend interactions and real-time updates. The Satoshi Coin Flip game, delves into backend logic, with structures like HouseCap and house_data for initializing and managing game data securely on the blockchain.

Multi-Version Implementation: The Satoshi Coin Flip game mentions two versions – one susceptible to MEV attacks and another that is resistant, indicating a focus on security and user experience variations. Such variations aren't implemented in Blackjack.

The complete app example can be found in the blackjack-sui repo .

Gameplay

In this single-player version of blackjack, the player competes against a dealer, which is automated by the system. The dealer is equipped with a public BLS key that plays a central role in the game's mechanics. The dealer's actions are triggered by HTTP requests to serverless functions. Players generate randomness for the game by interacting with their mouse on the screen, after which they place their bet to start the game. Upon initiating the game, a request is made to the backend (dealer), which processes it by signing and subsequently dealing two cards to the player and one to themselves.

The player has the option to 'Hit' or 'Stand.' Selecting 'Stand' triggers the dealer to draw cards until the total reaches 17 or higher. After the dealer stops, the smart contract steps in to compare the totals and declare the winner. On the other hand, choosing 'Hit' prompts the dealer to draw an additional card for the player.

Note that more complex blackjack rules, such as splitting, are considered out of scope for this example, and are therefore not implemented.

The single_player_blackjack.move module includes several constants that define game statuses and help track the game's progress:

There are also constants for error handling, such as EInvalidBlsSig , EInsufficientBalance , and others, ensuring robust game mechanics.

Structs like GameCreatedEvent , GameOutcomeEvent , and HitDoneEvent capture the various events and actions within a game. The HitRequest and StandRequest structs ensure that a move (hit/stand) can be performed by the house only if the player has already requested it. HouseAdminCap and HouseData are crucial for maintaining the house's data, including balance and public key, while the Game struct contains all the necessary information about each game, such as player data, cards, and the current status.

The module's functions can be broadly categorized into initialization, game management, and utility functions. The init function sets up the house admin capability, while initialize_house_data prepares the house for the game by setting up the balance and public key. place_bet_and_create_game is the entry point for players to start a new game, involving a bet and random input. The functions first_deal , hit , and stand govern the core gameplay, handling the dealing of cards and player choices.

Utility functions like get_next_random_card and get_card_sum are essential for the game's mechanics, generating random cards and calculating hand values. The module also includes accessors for retrieving various pieces of game and house data.

For testing purposes, the module provides special functions like get_house_admin_cap_for_testing , player_won_post_handling_for_test and house_won_post_handling_for_test , ensuring that developers can thoroughly test the game mechanics and house data handling.

The next module, counter_nft.move , introduces the Counter NFT, a key component in the game's mechanics. It serves as the verifiable random function (VRF) input, ensuring uniqueness in each game. The Counter NFT's value increases after each use, maintaining its distinctiveness for every new request. For first-time players, the creation of a Counter NFT is mandatory. To enhance user experience, the user interface can automate this process by integrating the Counter NFT's creation with the game initiation in a single transaction block. This seamless integration simplifies the process for the player, allowing them to focus on the gameplay. This counter serves the same purpose as the one in the Coin Flip example .

The backend is used for all the transactions that are executed by the dealer. The backend can be completely stateless, and for that reason serverless functions are utilized. As a result, the corresponding code lies under the app/src/app/api/ directory .

The backend code is split in the following sub directories:

An interesting aspect of developing a dApp on Sui, that is coupled to using a full node with/without a load balancer, and requires attention by the developer, is the occurrence of read-after-write and write-after-write cases.

As an example, in the blackjack game, just after the create-game transaction that the user executes, the dealer executes the initial-deal transaction. This one accepts an argument and modifies the game object, meaning that you are using an object that was just created.

To ensure that the game object is available in the Full node that the dealer is using, we need to call waitForTransaction after the create-game transaction.

In the same way, every time you re-fetch the game object in the frontend, make sure that the previous transaction that modified the game object is already available in the Full node.

This leads to the need of exchanging the txDigest between the frontend and the backend, and use waitForTransaction on each write-after-write or read-after-write case.

The page component , a central element of the blackjack game's frontend module, is structured to create an interactive and responsive gaming experience. Written in React, it integrates several features and functions to handle the game's logic and user interactions effectively.

The frontend is a NextJS project, that follows the NextJS App Router project structure . The main code of the frontend lies under the app/src/ directory. The main sub-directories are:

Custom Hooks: To keep the code as structured as possible, multiple custom hooks are utilized to manage the complex state of the game board at each step. The useBlackjackGame custom hook encapsulates the game state and logic, exposing all the required information (with fields such as game , isInitialDealLoading), and the required functionality (with methods such as handleCreateGame and handleHit) to display and play the game. Multiple additional custom hooks, such as useCreateBlackjackGame , and useMakeMoveInBlackjackGame are encapsulating their own piece of state and logic to make the code readable and maintainable.

Component for Game Initialization: The StartGame component is implemented to facilitate the creation of a new game. It renders the CollectMouseRandomness to capture the randomness and uses the handleCreateGame function of the useBlackjackGame hook to execute the create-game transaction.

Randomness Generation: Fair game outcomes are also ensured by the CollectMouseRandomness component. This component is using the useMouseRandomness custom hook, and is in charge of capturing some user's mouse movements and generating a random bytes array. This array is converted to a hexadecimal string (randomness) and used in the create-game transaction.

Card Displaying and Management: The DealerCards and the PlayerCards components are used to display the total points and the cards owned by the dealer and the player respectively.

Game Actions: The GameActions component is used to display the Hit and Stand buttons, and trigger the corresponding actions, as they are exported by the useBlackjackGame hook to execute the corresponding transactions.

BlackjackBanner : The BlackjackBanner component is used as a custom view to display when the player wins with a blackjack.

Blockchain-Based Logic: Both games are built on Sui, leveraging its capabilities for decentralized applications. The core game logic for each resides in Move modules, ensuring secure and verifiable gameplay.

State Management: In both games, state management is crucial. For blackjack, this involves managing the player and dealer's hands and scores using React state hooks. In Satoshi Coin Flip, the state is managed through Move structs like HouseData , which track the house's balance and other game-related details.

Randomness and Fair Play: Both games emphasize randomness for fairness. Blackjack uses a Counter NFT and player mouse movements to generate randomness, while Satoshi Coin Flip only uses a Counter NFT as a unique input for the Verifiable Random Function (VRF) in each game.

Smart Contract Interactions: Each game involves smart contract interactions for game actions like placing bets, dealing cards (Blackjack), or making guesses (Coin Flip). These interactions are crucial for executing the game's logic on the blockchain.

Game Mechanics and Complexity: Blackjack is a more complex game with multiple actions (hit, stand, deal) and state updates, requiring a more dynamic frontend. In contrast, Satoshi Coin Flip has a simpler mechanic centered around a single bet and guess outcome.

User Interface (UI) Complexity: The Blackjack game involves a more intricate UI to display cards, manage game states, and player interactions. Satoshi Coin Flip, being simpler in gameplay, requires a less complex UI.

Backend Processing: In Blackjack, the dealer is automated (the machine), and the player's actions directly influence game outcomes. In the Coin Flip game, the house (smart contract) plays a more passive role, primarily in initializing and finalizing the game based on the player's guess.

Module Structure and Focus: The Blackjack game focuses more on frontend interactions and real-time updates. The Satoshi Coin Flip game, delves into backend logic, with structures like HouseCap and house_data for initializing and managing game data securely on the blockchain.

Multi-Version Implementation: The Satoshi Coin Flip game mentions two versions – one susceptible to MEV attacks and another that is resistant, indicating a focus on security and user experience variations. Such variations aren't implemented in Blackjack.

The complete app example can be found in the blackjack-sui repo .

Smart contracts

The single_player_blackjack.move module includes several constants that define game statuses and help track the game's progress:

There are also constants for error handling, such as EInvalidBlsSig , EInsufficientBalance , and others, ensuring robust game mechanics.

Structs like GameCreatedEvent , GameOutcomeEvent , and HitDoneEvent capture the various events and actions within a game. The HitRequest and StandRequest structs ensure that a move (hit/stand) can be performed by the house only if the player has already requested it. HouseAdminCap and HouseData are crucial for maintaining the house's data, including balance and public key, while the Game struct contains all the necessary information about each game, such as player data, cards, and the current status.

The module's functions can be broadly categorized into initialization, game management, and utility functions. The init function sets up the house admin capability, while initialize_house_data prepares the house for the game by setting up the balance and public key. place_bet_and_create_game is the entry point for players to start a new game, involving a bet and random input. The functions first_deal , hit , and stand govern the core gameplay, handling the dealing of cards and player choices.

Utility functions like get_next_random_card and get_card_sum are essential for the game's mechanics, generating random cards and calculating hand values. The module also includes accessors for retrieving various pieces of game and house data.

For testing purposes, the module provides special functions like get_house_admin_cap_for_testing , player_won_post_handling_for_test and house_won_post_handling_for_test , ensuring that developers can thoroughly test the game mechanics and house data handling.

The next module, counter_nft.move , introduces the Counter NFT, a key component in the game's mechanics. It serves as the verifiable random function (VRF) input, ensuring uniqueness in each game. The Counter NFT's value increases after each use, maintaining its distinctiveness for every new request. For first-time players, the creation of a Counter NFT is mandatory. To enhance user experience, the user interface can automate this process by integrating the Counter NFT's creation with the game initiation in a single transaction block. This seamless integration simplifies the process for the player, allowing them to focus on the gameplay. This counter serves the same purpose as the one in the Coin Flip example .

The backend is used for all the transactions that are executed by the dealer. The backend can be completely stateless, and for that reason serverless functions are utilized. As a result, the corresponding code lies under the app/src/app/api/ directory .

The backend code is split in the following sub directories:

An interesting aspect of developing a dApp on Sui, that is coupled to using a full node with/without a load balancer, and requires attention by the developer, is the occurrence of read-after-write and write-after-write cases.

As an example, in the blackjack game, just after the create-game transaction that the user executes, the dealer executes the initial-deal transaction. This one accepts an argument and modifies the game object, meaning that you are using an object that was just created.

To ensure that the game object is available in the Full node that the dealer is using, we need to call waitForTransaction after the create-game transaction.

In the same way, every time you re-fetch the game object in the frontend, make sure that the previous transaction that modified the game object is already available in the Full node.

This leads to the need of exchanging the txDigest between the frontend and the backend, and use waitForTransaction on each write-after-write or read-after-write case.

The page component , a central element of the blackjack game's frontend module, is structured to create an interactive and responsive gaming experience. Written in React, it integrates several features and functions to handle the game's logic and user interactions effectively.

The frontend is a NextJS project, that follows the NextJS App Router project structure . The main code of the frontend lies under the app/src/ directory. The main sub-directories are:

Custom Hooks: To keep the code as structured as possible, multiple custom hooks are utilized to manage the complex state of the game board at each step. The useBlackjackGame custom hook encapsulates the game state and logic, exposing all the required information (with fields such as game , isInitialDealLoading), and the required functionality (with methods such as handleCreateGame and handleHit) to display and play the game. Multiple additional custom hooks, such as useCreateBlackjackGame , and useMakeMoveInBlackjackGame are encapsulating their own piece of state and logic to make the code readable and maintainable.

Component for Game Initialization: The StartGame component is implemented to facilitate the creation of a new game. It renders the CollectMouseRandomness to capture the randomness and uses the handleCreateGame function of the useBlackjackGame hook to execute the create-game transaction.

Randomness Generation: Fair game outcomes are also ensured by the CollectMouseRandomness component. This component is using the useMouseRandomness custom hook, and is in charge of capturing some user's mouse movements and generating a random bytes array. This array is converted to a hexadecimal string (randomness) and used in the create-game transaction.

Card Displaying and Management: The DealerCards and the PlayerCards components are used to display the total points and the cards owned by the dealer and the player respectively.

Game Actions: The GameActions component is used to display the Hit and Stand buttons, and trigger the corresponding actions, as they are exported by the useBlackjackGame hook to execute the corresponding transactions.

BlackjackBanner : The BlackjackBanner component is used as a custom view to display when the player wins with a blackjack.

Blockchain-Based Logic: Both games are built on Sui, leveraging its capabilities for decentralized applications. The core game logic for each resides in Move modules, ensuring secure and verifiable gameplay.

State Management: In both games, state management is crucial. For blackjack, this involves managing the player and dealer's hands and scores using React state hooks. In Satoshi Coin Flip, the state is managed through Move structs like HouseData , which track the house's balance and other game-related details.

Randomness and Fair Play: Both games emphasize randomness for fairness. Blackjack uses a Counter NFT and player mouse movements to generate randomness, while Satoshi Coin Flip only uses a Counter NFT as a unique input for the Verifiable Random Function (VRF) in each game.

Smart Contract Interactions: Each game involves smart contract interactions for game actions like placing bets, dealing cards (Blackjack), or making guesses (Coin Flip). These interactions are crucial for executing the game's logic on the blockchain.

Game Mechanics and Complexity: Blackjack is a more complex game with multiple actions (hit, stand, deal) and state updates, requiring a more dynamic frontend. In contrast, Satoshi Coin Flip has a simpler mechanic centered around a single bet and guess outcome.

User Interface (UI) Complexity: The Blackjack game involves a more intricate UI to display cards, manage game states, and player interactions. Satoshi Coin Flip, being simpler in gameplay, requires a less complex UI.

Backend Processing: In Blackjack, the dealer is automated (the machine), and the player's actions directly influence game outcomes. In the Coin Flip game, the house (smart contract) plays a more passive role, primarily in initializing and finalizing the game based on the player's guess.

Module Structure and Focus: The Blackjack game focuses more on frontend interactions and real-time updates. The Satoshi Coin Flip game, delves into backend logic, with structures like HouseCap and house_data for initializing and managing game data securely on the blockchain.

Multi-Version Implementation: The Satoshi Coin Flip game mentions two versions – one susceptible to MEV attacks and another that is resistant, indicating a focus on security and user experience variations. Such variations aren't implemented in Blackjack.

The complete app example can be found in the blackjack-sui repo .

Backend

The backend is used for all the transactions that are executed by the dealer. The backend can be completely stateless, and for that reason serverless functions are utilized. As a result, the corresponding code lies under the app/src/app/api/ directory .

The backend code is split in the following sub directories:

An interesting aspect of developing a dApp on Sui, that is coupled to using a full node with/without a load balancer, and requires attention by the developer, is the occurrence of read-after-write and write-after-write cases.

As an example, in the blackjack game, just after the create-game transaction that the user executes, the dealer executes the initial-deal transaction. This one accepts an argument and modifies the game object, meaning that you are using an object that was just created.

To ensure that the game object is available in the Full node that the dealer is using, we need to call waitForTransaction after the create-game transaction.

In the same way, every time you re-fetch the game object in the frontend, make sure that the previous transaction that modified the game object is already available in the Full node.

This leads to the need of exchanging the txDigest between the frontend and the backend, and use waitForTransaction on each write-after-write or read-after-write case.

The page component , a central element of the blackjack game's frontend module, is structured to create an interactive and responsive gaming experience. Written in React, it integrates several features and functions to handle the game's logic and user interactions effectively.

The frontend is a NextJS project, that follows the NextJS App Router project structure . The main code of the frontend lies under the app/src/ directory. The main sub-directories are:

Custom Hooks: To keep the code as structured as possible, multiple custom hooks are utilized to manage the complex state of the game board at each step. The useBlackjackGame custom hook encapsulates the game state and logic, exposing all the required information (with fields such as game , isInitialDealLoading), and the required functionality (with methods such as handleCreateGame and handleHit) to display and play the game. Multiple additional custom hooks, such as useCreateBlackjackGame , and useMakeMoveInBlackjackGame are encapsulating their own piece of state and logic to make the code readable and maintainable.

Component for Game Initialization: The StartGame component is implemented to facilitate the creation of a new game. It renders the CollectMouseRandomness to capture the randomness and uses the handleCreateGame function of the useBlackjackGame hook to execute the create-game transaction.

Randomness Generation: Fair game outcomes are also ensured by the CollectMouseRandomness component. This component is using the useMouseRandomness custom hook, and is in charge of capturing some user's mouse movements and generating a random bytes array. This array is converted to a hexadecimal string (randomness) and used in the create-game transaction.

Card Displaying and Management: The DealerCards and the PlayerCards components are used to display the total points and the cards owned by the dealer and the player respectively.

Game Actions: The GameActions component is used to display the Hit and Stand buttons, and trigger the corresponding actions, as they are exported by the useBlackjackGame hook to execute the corresponding transactions.

BlackjackBanner : The BlackjackBanner component is used as a custom view to display when the player wins with a blackjack.

Blockchain-Based Logic: Both games are built on Sui, leveraging its capabilities for decentralized applications. The core game logic for each resides in Move modules, ensuring secure and verifiable gameplay.

State Management: In both games, state management is crucial. For blackjack, this involves managing the player and dealer's hands and scores using React state hooks. In Satoshi Coin Flip, the state is managed through Move structs like HouseData , which track the house's balance and other game-related details.

Randomness and Fair Play: Both games emphasize randomness for fairness. Blackjack uses a Counter NFT and player mouse movements to generate randomness, while Satoshi Coin Flip only uses a Counter NFT as a unique input for the Verifiable Random

Function (VRF) in each game.

Smart Contract Interactions: Each game involves smart contract interactions for game actions like placing bets, dealing cards (Blackjack), or making guesses (Coin Flip). These interactions are crucial for executing the game's logic on the blockchain.

Game Mechanics and Complexity: Blackjack is a more complex game with multiple actions (hit, stand, deal) and state updates, requiring a more dynamic frontend. In contrast, Satoshi Coin Flip has a simpler mechanic centered around a single bet and guess outcome.

User Interface (UI) Complexity: The Blackjack game involves a more intricate UI to display cards, manage game states, and player interactions. Satoshi Coin Flip, being simpler in gameplay, requires a less complex UI.

Backend Processing: In Blackjack, the dealer is automated (the machine), and the player's actions directly influence game outcomes. In the Coin Flip game, the house (smart contract) plays a more passive role, primarily in initializing and finalizing the game based on the player's guess.

Module Structure and Focus: The Blackjack game focuses more on frontend interactions and real-time updates. The Satoshi Coin Flip game, delves into backend logic, with structures like HouseCap and house_data for initializing and managing game data securely on the blockchain.

Multi-Version Implementation: The Satoshi Coin Flip game mentions two versions – one susceptible to MEV attacks and another that is resistant, indicating a focus on security and user experience variations. Such variations aren't implemented in Blackjack.

The complete app example can be found in the blackjack-sui repo .

Frontend

The page component , a central element of the blackjack game's frontend module, is structured to create an interactive and responsive gaming experience. Written in React, it integrates several features and functions to handle the game's logic and user interactions effectively.

The frontend is a NextJS project, that follows the NextJS App Router project structure . The main code of the frontend lies under the app/src/ directory. The main sub-directories are:

Custom Hooks: To keep the code as structured as possible, multiple custom hooks are utilized to manage the complex state of the game board at each step. The useBlackjackGame custom hook encapsulates the game state and logic, exposing all the required information (with fields such as game , isInitialDealLoading), and the required functionality (with methods such as handleCreateGame and handleHit) to display and play the game. Multiple additional custom hooks, such as useCreateBlackjackGame , and useMakeMoveInBlackjackGame are encapsulating their own piece of state and logic to make the code readable and maintainable.

Component for Game Initialization: The StartGame component is implemented to facilitate the creation of a new game. It renders the CollectMouseRandomness to capture the randomness and uses the handleCreateGame function of the useBlackjackGame hook to execute the create-game transaction.

Randomness Generation: Fair game outcomes are also ensured by the CollectMouseRandomness component. This component is using the useMouseRandomness custom hook, and is in charge of capturing some user's mouse movements and generating a random bytes array. This array is converted to a hexadecimal string (randomness) and used in the create-game transaction.

Card Displaying and Management: The DealerCards and the PlayerCards components are used to display the total points and the cards owned by the dealer and the player respectively.

Game Actions: The GameActions component is used to display the Hit and Stand buttons, and trigger the corresponding actions, as they are exported by the useBlackjackGame hook to execute the corresponding transactions.

BlackjackBanner : The BlackjackBanner component is used as a custom view to display when the player wins with a blackjack.

Blockchain-Based Logic: Both games are built on Sui, leveraging its capabilities for decentralized applications. The core game logic for each resides in Move modules, ensuring secure and verifiable gameplay.

State Management: In both games, state management is crucial. For blackjack, this involves managing the player and dealer's hands and scores using React state hooks. In Satoshi Coin Flip, the state is managed through Move structs like HouseData , which track the house's balance and other game-related details.

Randomness and Fair Play: Both games emphasize randomness for fairness. Blackjack uses a Counter NFT and player mouse movements to generate randomness, while Satoshi Coin Flip only uses a Counter NFT as a unique input for the Verifiable Random Function (VRF) in each game.

Smart Contract Interactions: Each game involves smart contract interactions for game actions like placing bets, dealing cards (Blackjack), or making guesses (Coin Flip). These interactions are crucial for executing the game's logic on the blockchain.

Game Mechanics and Complexity: Blackjack is a more complex game with multiple actions (hit, stand, deal) and state updates, requiring a more dynamic frontend. In contrast, Satoshi Coin Flip has a simpler mechanic centered around a single bet and guess outcome.

User Interface (UI) Complexity: The Blackjack game involves a more intricate UI to display cards, manage game states, and player interactions. Satoshi Coin Flip, being simpler in gameplay, requires a less complex UI.

Backend Processing: In Blackjack, the dealer is automated (the machine), and the player's actions directly influence game outcomes. In the Coin Flip game, the house (smart contract) plays a more passive role, primarily in initializing and finalizing the game based on the player's guess.

Module Structure and Focus: The Blackjack game focuses more on frontend interactions and real-time updates. The Satoshi Coin Flip game, delves into backend logic, with structures like HouseCap and house_data for initializing and managing game data securely on the blockchain.

Multi-Version Implementation: The Satoshi Coin Flip game mentions two versions – one susceptible to MEV attacks and another that is resistant, indicating a focus on security and user experience variations. Such variations aren't implemented in Blackjack.

The complete app example can be found in the blackjack-sui repo .

Comparison: Blackjack and Coin Flip

Blockchain-Based Logic: Both games are built on Sui, leveraging its capabilities for decentralized applications. The core game logic for each resides in Move modules, ensuring secure and verifiable gameplay.

State Management: In both games, state management is crucial. For blackjack, this involves managing the player and dealer's hands and scores using React state hooks. In Satoshi Coin Flip, the state is managed through Move structs like HouseData , which track the house's balance and other game-related details.

Randomness and Fair Play: Both games emphasize randomness for fairness. Blackjack uses a Counter NFT and player mouse movements to generate randomness, while Satoshi Coin Flip only uses a Counter NFT as a unique input for the Verifiable Random Function (VRF) in each game.

Smart Contract Interactions: Each game involves smart contract interactions for game actions like placing bets, dealing cards (Blackjack), or making guesses (Coin Flip). These interactions are crucial for executing the game's logic on the blockchain.

Game Mechanics and Complexity: Blackjack is a more complex game with multiple actions (hit, stand, deal) and state updates, requiring a more dynamic frontend. In contrast, Satoshi Coin Flip has a simpler mechanic centered around a single bet and guess outcome.

User Interface (UI) Complexity: The Blackjack game involves a more intricate UI to display cards, manage game states, and player interactions. Satoshi Coin Flip, being simpler in gameplay, requires a less complex UI.

Backend Processing: In Blackjack, the dealer is automated (the machine), and the player's actions directly influence game outcomes. In the Coin Flip game, the house (smart contract) plays a more passive role, primarily in initializing and finalizing the game based on the player's guess.

Module Structure and Focus: The Blackjack game focuses more on frontend interactions and real-time updates. The Satoshi Coin Flip game, delves into backend logic, with structures like HouseCap and house_data for initializing and managing game data securely on the blockchain.

Multi-Version Implementation: The Satoshi Coin Flip game mentions two versions – one susceptible to MEV attacks and another that is resistant, indicating a focus on security and user experience variations. Such variations aren't implemented in Blackjack.

The complete app example can be found in the blackjack-sui repo .