Coin Flip

This example walks you through building a coin flip dApp, covering the full end-to-end flow of building your Sui Move module and connecting it to your React Sui dApp. This coin flip dApp utilizes verifiable random functions (VRFs) to create a fair coin game on the Sui blockchain. The user (human) plays against the house (module) and places a bet on either heads or tails. The user then either receives double their bet, or gets nothing, depending on the outcome of the game.

The guide is split into two parts:

Source code locations for the smart contracts and frontend:

Before getting started, make sure you have:

In this part of the guide, you write the Move contracts that manage the house and set up the coin-flip logic. The first step is to set up a Move package for storing your Move modules.

To follow along with this guide, set your new Move package to satoshi_flip .

This example uses several modules to create a package for the Satoshi Coin Flip game. The first module is house_data.move . You need to store the game's data somewhere, and in this module you create a shared object for all house data.

Create a new file in the sources directory with the name house_data.move and populate the file with the following code:

There are few details to take note of in this code:

Next, add some more code to this module:

So far, you've set up the data structures within the module. Now, create a function that initializes the house data and shares the HouseData object:

With the house data initialized, you also need to add some functions that enable some important administrative tasks for the house to perform:

All of these functions contain an assert! call that ensures only the house can call them:

You have established the data structure of this module, but without the appropriate functions this data is not accessible. Now add helper functions that return mutable references, read-only references, and test-only functions:

And with that, your house_data.move code is complete.

In the same sources directory, now create a file named counter_nft.move . A Counter object is used as the VRF input for every game that a player plays. First, populate the file with the following:

This might look familiar from the house module. You set the module name, import functions from the standard library, and initialize the Counter object. The Counter object has the key ability, but does not have store - this prevents the object from being transferable.

In addition, you create mint and transfer_to_sender functions used when the game is set up to create the Counter object (with an initial count of 0) and transfer the object to the sender of the transaction. And finally a burn function to allow deletion of the Counter .

You have a Counter object, as well as functions that initialize and burn the object, but you need a way to increment the counter. Add the following code to the module:

The get_vrf_input_and_increment function is the core of this module. The function takes a mutable reference to the Counter object that the mint function creates, then appends the Counter object's current count to its ID and returns the result as a vector . The function then calls the internal increment function to increment the count by one.

This code also adds a count function that returns the current count, and a test-only function that calls the burn function.

Lastly, you need a game module and object that can create a new game, distribute funds after the game, and potentially cancel games. Because this is a one-player game, create an address-owned object rather than a shared object .

Create the game module. In the sources directory, create a new file called single_player_satoshi.move and populate with the following:

This code follows the same pattern as the others. First, you include the respective imports, although this time the imports are not only from the standard library but also include modules created previously in this example. You also create several constants (in upper case), as well as constants used for errors (Pascal case prefixed with E).

Lastly in this section, you also create structs for two events to emit. Indexers consume emitted events, which enables you to track these events through API services, or your own indexer. In this case, the events are for when a new game begins (NewGame) and for the outcome of a game when it has finished (Outcome).

Add a struct to the module:

The Game struct represents a single game and all its information, including the epoch the player placed the bet (guess_placed_epoch), bet (total_stake), guess , address of the player , vrf_input , and the fee the house collects (fee_bp).

Now take a look at the main function in this game, finish_game :

Now add a function that handles game disputes:

This function, dispute_and_win , ensures that no bet can live in "purgatory". After a certain amount of time passes, the player can call this function and get all of their funds back.

The rest of the functions are accessors and helper functions used to retrieve values, check if values exist, initialize the game, and so on:

This represents a basic example of a coin flip backend in Move. The game module, single_player_satoshi , is prone to MEV attacks, but the user experience for the player is streamlined. Another example game module, mev_attack_resistant_single_player_satoshi , exists that is MEV-resistant, but has a slightly downgraded user experience (two player-transactions per game).

You can read more about both versions of the game, and view the full source code for all the modules in the Satoshi Coin Flip repository .

Now that you have written our contracts, it's time to deploy them.

See Publish a Package for a more detailed guide on publishing packages or Sui Client CLI for a complete reference of client commands in the Sui CLI.

Before publishing your code, you must first initialize the Sui Client CLI, if you haven't already. To do so, in a terminal or console at the root directory of the project enter sui client . If you receive the following response, complete the remaining instructions:

Enter y to proceed. You receive the following response:

Leave this blank (press Enter). You receive the following response:

Select 0 . Now you should have a Sui address set up.

Next, configure the Sui CLI to use testnet as the active environment, as well. If you haven't already set up a testnet environment, do so by running the following command in a terminal or console:

Run the following command to activate the testnet environment:

Before being able to publish your package to Testnet, you need Testnet SUI tokens. To get some, visit the online faucet at https://faucet.sui.io/ . For other ways to get SUI in your Testnet account, see Get SUI Tokens .

Now that you have an account with some Testnet SUI, you can deploy your contracts. To publish your package, use the following command in the same terminal or console:

For the gas budget, use a standard value such as 20000000 .

The output of this command contains a packageID value that you need to save to use the package.

Partial snippet of CLI deployment output.

Save the PackageID and the ObjectID of the HouseCap object you receive in your own response to connect to your frontend .

In this case, the PackageID is 0x4120b39e5d94845aa539d4b830743a7433fd8511bdcf3841f98080080f327ca8 and the

HouseCap ID is 0xfa1f6edad697afca055749fedbdee420b6cdba3edc2f7fd4927ed42f98a7e63a .

Well done. You have written and deployed the Move package! □

To turn this into a complete dApp, you need to create a frontend .

In this final part of the dApp example, you build a frontend (UI) that allows end users to place bets and take profits, and lets the admin manage the house.

To skip building the frontend and test out your newly deployed package, use the provided Satoshi Coin Flip Frontend Example repository and follow the instructions in the example's README.md file

Before getting started, make sure you have:

The UI of this example demonstrates how to use the dApp Kit instead of serving as a production-grade product, so the Player and the House features are in the same UI to simplify the process. In a production solution, your frontend would only contain functionality dedicated to the Player, with a backend service carrying out the interactions with House functions in the smart contracts.

The UI has two columns:

The first step is to set up the client app. Run the following command to scaffold a new app.

or

Structure the project folder according to the UI layout, meaning that all Player-related React components reside in the containers/Player folder, while all House-related React components reside in the containers/House folder.

Add the packageId value you saved from deploying your package to a new src/constants.ts file in your project:

The UI interacts with the Single Player smart contract variant of the game. This section walks you through each step in the smart contract flow and the corresponding frontend code.

The following frontend code snippets include only the most relevant sections. Refer to the Satoshi Coin Flip Frontend Example repository for complete source code.

As is common in other React projects, App.tsx is where you implement the outer layout:

Like other dApps, you need a "connect wallet" button to enable connecting users' wallets. dApp Kit contains a pre-made ConnectButton React component that you can reuse to help users onboard.

useCurrentAccount() is a React hook the dApp Kit also provides to query the current connected wallet; returning null if there isn't a wallet connection. Leverage this behavior to prevent a user from proceeding further if they haven't connected their wallet yet.

After ensuring that the user has connected their wallet, you can display the two columns described in the previous section: PlayerSesh and HouseSesh components.

Okay, that's a good start to have an overview of the project. Time to move to initializing the HouseData object. All the frontend logic for calling this lives in the HouseInitialize.tsx component. The component includes UI code, but the logic that executes the transaction follows:

To use a programmable transaction block (PTB) in Sui, create a Transaction . To initiate a Move call, you must know the global identifier of a public function in your smart contract. The global identifier usually takes the following form:

In this example, it is:

There are a few parameters that you need to pass into initialize_house_data() Move function: the HouseCap ID, the House stake, and the House BLS public key:

Now sign and execute the transaction block. dApp Kit provides a React hook useSignAndExecuteTransaction() to streamline this process. This hook, when executed, prompts the UI for you to approve, sign, and execute the transaction block. You can configure the hook with the showObjectChanges option to return the newly-created HouseData shared object as the result of the transaction block. This HouseData object is important as you use it as input for later Move calls, so save its ID somewhere.

Great, now you know how to initialize the HouseData shared object. Move to the next function call.

In this game, the users must create a Counter object to start the game. So there should be a place in the Player column UI to list the existing Counter object information for the player to choose. It seems likely that you will reuse the fetching logic for the Counter object in several places in your UI, so it's good practice to isolate this logic into a React hook, which you call useFetchCounterNft() in useFetchCounterNft.ts :

This hook logic is very basic: if there is no current connected wallet, return empty data; otherwise, fetch the Counter object and return it. dApp Kit provides a React hook, useSuiClientQuery() , that enables interaction with Sui RPC methods. Different RPC methods require different parameters. To fetch the object owned by a known address, use the getOwnedObjects query .

Now, pass the address of the connected wallet, as well as the global identifier for the Counter . This is in similar format to the global identifier type for function calls:

${PACKAGE_ID}::counter_nft::Counter

That's it, now put the hook into the UI component PlayerListCounterNft.tsx and display the data:

For the case when there is no existing Counter object, mint a new Counter for the connected wallet. Also add the minting logic into PlayerListCounterNft.tsx when the user clicks the button. You already know how to build and execute a Move call with TransactionBlock and initialize_house_data() , you can implement a similar call here.

As you might recall with Transaction , outputs from the transaction can be inputs for the next transaction. Call counter_nft::mint() , which returns the newly created Counter object, and use it as input for counter_nft::transfer_to_sender() to transfer the Counter object to the caller wallet:

Great, now you can create the game with the created Counter object. Isolate the game creation logic into PlayerCreateGame.tsx . There is one more thing to keep in mind - to flag an input as an on-chain object, you should use txb.object() with the corresponding object ID.

One final step remains: settle the game. There are a couple of ways you can use the UI to settle the game:

Event subscriptions are deprecated. To learn future-safe methods to work with events, see Using Events .

All of this logic is in HouseFinishGame.tsx :

To get the underlying SuiClient instance from the SDK, use useSuiClient() . You want to subscribe to events whenever the HouseFinishGame component loads. To do this, use the React hook useEffect() from the core React library.

SuiClient exposes a method called subscribeEvent() that enables you to subscribe to a variety of event types. SuiClient::subscribeEvent() is actually a thin wrapper around the RPC method suix_subscribeEvent .

The logic is that whenever a new game starts, you want to settle the game immediately. The necessary event to achieve this is the Move event type called single_player_satoshi::NewGame . If you inspect the parsed payload of the event through event.parsedJson , you can see the corresponding event fields declared in the smart contract. In this case, you just need to use two fields, the Game ID and the VRF input.

The next steps are similar to the previous Move calls, but you have to use the BLS private key to sign the VRF input and then pass the Game ID, signed VRF input and HouseData ID to the single_player_satoshi::finish_game() Move call.

Last but not least, remember to unsubscribe from the event whenever the HouseFinishGame component dismounts. This is important as you might not want to subscribe to the same event multiple times.

Congratulations, you completed the frontend. You can carry the lessons learned here forward when using the dApp Kit to build your next Sui project.

What the guide teaches

Before getting started, make sure you have:

In this part of the guide, you write the Move contracts that manage the house and set up the coin-flip logic. The first step is to set up a Move package for storing your Move modules.

To follow along with this guide, set your new Move package to satoshi_flip .

This example uses several modules to create a package for the Satoshi Coin Flip game. The first module is house_data.move . You need to store the game's data somewhere, and in this module you create a shared object for all house data.

Create a new file in the sources directory with the name house_data.move and populate the file with the following code:

There are few details to take note of in this code:

Next, add some more code to this module:

So far, you've set up the data structures within the module. Now, create a function that initializes the house data and shares the HouseData object:

With the house data initialized, you also need to add some functions that enable some important administrative tasks for the house to perform:

All of these functions contain an assert! call that ensures only the house can call them:

You have established the data structure of this module, but without the appropriate functions this data is not accessible. Now add helper functions that return mutable references, read-only references, and test-only functions:

And with that, your house_data.move code is complete.

In the same sources directory, now create a file named counter_nft.move . A Counter object is used as the VRF input for every game that a player plays. First, populate the file with the following:

This might look familiar from the house module. You set the module name, import functions from the standard library, and initialize the Counter object. The Counter object has the key ability, but does not have store - this prevents the object from being transferable.

In addition, you create mint and transfer_to_sender functions used when the game is set up to create the Counter object (with an initial count of 0) and transfer the object to the sender of the transaction. And finally a burn function to allow deletion of the Counter .

You have a Counter object, as well as functions that initialize and burn the object, but you need a way to increment the counter. Add the following code to the module:

The get_vrf_input_and_increment function is the core of this module. The function takes a mutable reference to the Counter object that the mint function creates, then appends the Counter object's current count to its ID and returns the result as a vector . The function then calls the internal increment function to increment the count by one.

This code also adds a count function that returns the current count, and a test-only function that calls the burn function.

Lastly, you need a game module and object that can create a new game, distribute funds after the game, and potentially cancel games. Because this is a one-player game, create an address-owned object rather than a shared object .

Create the game module. In the sources directory, create a new file called single_player_satoshi.move and populate with the following:

This code follows the same pattern as the others. First, you include the respective imports, although this time the imports are not only from the standard library but also include modules created previously in this example. You also create several constants (in upper case), as well as constants used for errors (Pascal case prefixed with E).

Lastly in this section, you also create structs for two events to emit. Indexers consume emitted events, which enables you to track these events through API services, or your own indexer. In this case, the events are for when a new game begins (NewGame) and for the outcome of a game when it has finished (Outcome).

Add a struct to the module:

The Game struct represents a single game and all its information, including the epoch the player placed the bet (guess_placed_epoch), bet (total_stake), guess , address of the player , vrf_input , and the fee the house collects (fee_bp).

Now take a look at the main function in this game, finish_game :

Now add a function that handles game disputes:

This function, dispute_and_win , ensures that no bet can live in “purgatory”. After a certain amount of time passes, the player can call this function and get all of their funds back.

The rest of the functions are accessors and helper functions used to retrieve values, check if values exist, initialize the game, and so on:

This represents a basic example of a coin flip backend in Move. The game module, single_player_satoshi , is prone to MEV attacks, but the user experience for the player is streamlined. Another example game module, mev_attack_resistant_single_player_satoshi , exists that is MEV-resistant, but has a slightly downgraded user experience (two player-transactions per game).

You can read more about both versions of the game, and view the full source code for all the modules in the Satoshi Coin Flip repository .

Now that you have written our contracts, it's time to deploy them.

See Publish a Package for a more detailed guide on publishing packages or Sui Client CLI for a complete reference of client commands in the Sui CLI.

Before publishing your code, you must first initialize the Sui Client CLI, if you haven't already. To do so, in a terminal or console at the root directory of the project enter sui client . If you receive the following response, complete the remaining instructions:

Enter y to proceed. You receive the following response:

Leave this blank (press Enter). You receive the following response:

Select 0 . Now you should have a Sui address set up.

Next, configure the Sui CLI to use testnet as the active environment, as well. If you haven't already set up a testnet environment, do so by running the following command in a terminal or console:

Run the following command to activate the testnet environment:

Before being able to publish your package to Testnet, you need Testnet SUI tokens. To get some, visit the online faucet at https://faucet.sui.io/ . For other ways to get SUI in your Testnet account, see Get SUI Tokens .

Now that you have an account with some Testnet SUI, you can deploy your contracts. To publish your package, use the following command in the same terminal or console:

For the gas budget, use a standard value such as 20000000 .

The output of this command contains a packageID value that you need to save to use the package.

Partial snippet of CLI deployment output.

Save the PackageID and the ObjectID of the HouseCap object you receive in your own response to connect to your frontend .

In this case, the PackageID is 0x4120b39e5d94845aa539d4b830743a7433fd8511bdcf3841f98080080f327ca8 and the HouseCap ID is 0xfa1f6edad697afca055749fedbdee420b6cdba3edc2f7fd4927ed42f98a7e63a .

Well done. You have written and deployed the Move package! □

To turn this into a complete dApp, you need to create a frontend .

In this final part of the dApp example, you build a frontend (UI) that allows end users to place bets and take profits, and lets the admin manage the house.

To skip building the frontend and test out your newly deployed package, use the provided Satoshi Coin Flip Frontend Example repository and follow the instructions in the example's README.md file

Before getting started, make sure you have:

The UI of this example demonstrates how to use the dApp Kit instead of serving as a production-grade product, so the Player and the House features are in the same UI to simplify the process. In a production solution, your frontend would only contain functionality dedicated to the Player, with a backend service carrying out the interactions with House functions in the smart contracts.

The UI has two columns:

The first step is to set up the client app. Run the following command to scaffold a new app.

or

Structure the project folder according to the UI layout, meaning that all Player-related React components reside in the containers/Player folder, while all House-related React components reside in the containers/House folder.

Add the packageId value you saved from deploying your package to a new src/constants.ts file in your project:

The UI interacts with the Single Player smart contract variant of the game. This section walks you through each step in the smart contract flow and the corresponding frontend code.

The following frontend code snippets include only the most relevant sections. Refer to the Satoshi Coin Flip Frontend Example repository for complete source code.

As is common in other React projects, App.tsx is where you implement the outer layout:

Like other dApps, you need a "connect wallet" button to enable connecting users' wallets. dApp Kit contains a pre-made ConnectButton React component that you can reuse to help users onboard.

useCurrentAccount() is a React hook the dApp Kit also provides to query the current connected wallet; returning null if there isn't a wallet connection. Leverage this behavior to prevent a user from proceeding further if they haven't connected their wallet yet.

After ensuring that the user has connected their wallet, you can display the two columns described in the previous section: PlayerSesh and HouseSesh components.

Okay, that's a good start to have an overview of the project. Time to move to initializing the HouseData object. All the frontend logic for calling this lives in the HouseInitialize.tsx component. The component includes UI code, but the logic that executes the transaction follows:

To use a programmable transaction block (PTB) in Sui, create a Transaction . To initiate a Move call, you must know the global identifier of a public function in your smart contract. The global identifier usually takes the following form:

In this example, it is:

There are a few parameters that you need to pass into initialize_house_data() Move function: the HouseCap ID, the House stake, and the House BLS public key:

Now sign and execute the transaction block. dApp Kit provides a React hook useSignAndExecuteTransaction() to streamline this process. This hook, when executed, prompts the UI for you to approve, sign, and execute the transaction block. You can configure the hook with the showObjectChanges option to return the newly-created HouseData shared object as the result of the transaction block. This HouseData object is important as you use it as input for later Move calls, so save its ID somewhere.

Great, now you know how to initialize the HouseData shared object. Move to the next function call.

In this game, the users must create a Counter object to start the game. So there should be a place in the Player column UI to list the existing Counter object information for the player to choose. It seems likely that you will reuse the fetching logic for the Counter object in several places in your UI, so it's good practice to isolate this logic into a React hook, which you call useFetchCounterNft() in useFetchCounterNft.ts :

This hook logic is very basic: if there is no current connected wallet, return empty data; otherwise, fetch the Counter object and return it. dApp Kit provides a React hook, useSuiClientQuery() , that enables interaction with Sui RPC methods. Different RPC methods require different parameters. To fetch the object owned by a known address, use the getOwnedObjects query .

Now, pass the address of the connected wallet, as well as the global identifier for the Counter . This is in similar format to the global identifier type for function calls:

${PACKAGE_ID}::counter_nft::Counter

That's it, now put the hook into the UI component PlayerListCounterNft.tsx and display the data:

For the case when there is no existing Counter object, mint a new Counter for the connected wallet. Also add the minting logic into PlayerListCounterNft.tsx when the user clicks the button. You already know how to build and execute a Move call with TransactionBlock and initialize_house_data() , you can implement a similar call here.

As you might recall with Transaction , outputs from the transaction can be inputs for the next transaction. Call counter_nft::mint() , which returns the newly created Counter object, and use it as input for counter_nft::transfer_to_sender() to transfer the Counter object to the caller wallet:

Great, now you can create the game with the created Counter object. Isolate the game creation logic into PlayerCreateGame.tsx . There is one more thing to keep in mind - to flag an input as an on-chain object, you should use txb.object() with the corresponding object ID.

One final step remains: settle the game. There are a couple of ways you can use the UI to settle the game:

Event subscriptions are deprecated. To learn future-safe methods to work with events, see Using Events .

All of this logic is in HouseFinishGame.tsx :

To get the underlying SuiClient instance from the SDK, use useSuiClient() . You want to subscribe to events whenever the HouseFinishGame component loads. To do this, use the React hook useEffect() from the core React library.

SuiClient exposes a method called subscribeEvent() that enables you to subscribe to a variety of event types. SuiClient::subscribeEvent() is actually a thin wrapper around the RPC method suix_subscribeEvent .

The logic is that whenever a new game starts, you want to settle the game immediately. The necessary event to achieve this is the Move event type called single_player_satoshi::NewGame . If you inspect the parsed payload of the event through event.parsedJson , you can see the corresponding event fields declared in the smart contract. In this case, you just need to use two fields, the Game ID and the VRF input.

The next steps are similar to the previous Move calls, but you have to use the BLS private key to sign the VRF input and then pass the Game ID, signed VRF input and HouseData ID to the single_player_satoshi::finish_game() Move call.

Last but not least, remember to unsubscribe from the event whenever the HouseFinishGame component dismounts. This is important as you might not want to subscribe to the same event multiple times.

Congratulations, you completed the frontend. You can carry the lessons learned here forward when using the dApp Kit to build your next Sui project.

What you need

Before getting started, make sure you have:

In this part of the guide, you write the Move contracts that manage the house and set up the coin-flip logic. The first step is to set up a Move package for storing your Move modules.

To follow along with this guide, set your new Move package to satoshi_flip .

This example uses several modules to create a package for the Satoshi Coin Flip game. The first module is house_data.move . You need to store the game's data somewhere, and in this module you create a shared object for all house data.

Create a new file in the sources directory with the name house_data.move and populate the file with the following code:

There are few details to take note of in this code:

Next, add some more code to this module:

So far, you've set up the data structures within the module. Now, create a function that initializes the house data and shares the HouseData object:

With the house data initialized, you also need to add some functions that enable some important administrative tasks for the house to perform:

All of these functions contain an assert! call that ensures only the house can call them:

You have established the data structure of this module, but without the appropriate functions this data is not accessible. Now add helper functions that return mutable references, read-only references, and test-only functions:

And with that, your house_data.move code is complete.

In the same sources directory, now create a file named counter_nft.move . A Counter object is used as the VRF input for every game that a player plays. First, populate the file with the following:

This might look familiar from the house module. You set the module name, import functions from the standard library, and initialize the Counter object. The Counter object has the key ability, but does not have store - this prevents the object from being transferable.

In addition, you create mint and transfer_to_sender functions used when the game is set up to create the Counter object (with an initial count of 0) and transfer the object to the sender of the transaction. And finally a burn function to allow deletion of the Counter .

You have a Counter object, as well as functions that initialize and burn the object, but you need a way to increment the counter. Add the following code to the module:

The get_vrf_input_and_increment function is the core of this module. The function takes a mutable reference to the Counter object that the mint function creates, then appends the Counter object's current count to its ID and returns the result as a vector . The function then calls the internal increment function to increment the count by one.

This code also adds a count function that returns the current count, and a test-only function that calls the burn function.

Lastly, you need a game module and object that can create a new game, distribute funds after the game, and potentially cancel games. Because this is a one-player game, create an address-owned object rather than a shared object .

Create the game module. In the sources directory, create a new file called single_player_satoshi.move and populate with the following:

This code follows the same pattern as the others. First, you include the respective imports, although this time the imports are not only from the standard library but also include modules created previously in this example. You also create several constants (in upper case), as well as constants used for errors (Pascal case prefixed with E).

Lastly in this section, you also create structs for two events to emit. Indexers consume emitted events, which enables you to track these events through API services, or your own indexer. In this case, the events are for when a new game begins (NewGame) and for the outcome of a game when it has finished (Outcome).

Add a struct to the module:

The Game struct represents a single game and all its information, including the epoch the player placed the bet (guess_placed_epoch), bet (total_stake), guess , address of the player , vrf_input , and the fee the house collects (fee_bp).

Now take a look at the main function in this game, finish_game :

Now add a function that handles game disputes:

This function, dispute_and_win , ensures that no bet can live in "purgatory". After a certain amount of time passes, the player can call this function and get all of their funds back.

The rest of the functions are accessors and helper functions used to retrieve values, check if values exist, initialize the game, and so on:

This represents a basic example of a coin flip backend in Move. The game module, single_player_satoshi , is prone to MEV attacks, but the user experience for the player is streamlined. Another example game module, mev_attack_resistant_single_player_satoshi , exists that is MEV-resistant, but has a slightly downgraded user experience (two player-transactions per game).

You can read more about both versions of the game, and view the full source code for all the modules in the Satoshi Coin Flip repository .

Now that you have written our contracts, it's time to deploy them.

See Publish a Package for a more detailed guide on publishing packages or Sui Client CLI for a complete reference of client commands in the Sui CLI.

Before publishing your code, you must first initialize the Sui Client CLI, if you haven't already. To do so, in a terminal or console at the root directory of the project enter sui client . If you receive the following response, complete the remaining instructions:

Enter y to proceed. You receive the following response:

Leave this blank (press Enter). You receive the following response:

Select 0 . Now you should have a Sui address set up.

Next, configure the Sui CLI to use testnet as the active environment, as well. If you haven't already set up a testnet environment, do so by running the following command in a terminal or console:

Run the following command to activate the testnet environment:

Before being able to publish your package to Testnet, you need Testnet SUI tokens. To get some, visit the online faucet at https://faucet.sui.io/ . For other ways to get SUI in your Testnet account, see Get SUI Tokens .

Now that you have an account with some Testnet SUI, you can deploy your contracts. To publish your package, use the following command in the same terminal or console:

For the gas budget, use a standard value such as 20000000 .

The output of this command contains a packageID value that you need to save to use the package.

Partial snippet of CLI deployment output.

Save the PackageID and the ObjectID of the HouseCap object you receive in your own response to connect to your frontend .

In this case, the PackageID is 0x4120b39e5d94845aa539d4b830743a7433fd8511bdcf3841f98080080f327ca8 and the HouseCap ID is 0xfa1f6edad697afca055749fedbdee420b6cdba3edc2f7fd4927ed42f98a7e63a .

Well done. You have written and deployed the Move package! □

To turn this into a complete dApp, you need to create a frontend .

In this final part of the dApp example, you build a frontend (UI) that allows end users to place bets and take profits, and lets the admin manage the house.

To skip building the frontend and test out your newly deployed package, use the provided Satoshi Coin Flip Frontend Example repository and follow the instructions in the example's README.md file

Before getting started, make sure you have:

The UI of this example demonstrates how to use the dApp Kit instead of serving as a production-grade product, so the Player and the House features are in the same UI to simplify the process. In a production solution, your frontend would only contain functionality dedicated to the Player, with a backend service carrying out the interactions with House functions in the smart contracts.

The UI has two columns:

The first step is to set up the client app. Run the following command to scaffold a new app.

or

Structure the project folder according to the UI layout, meaning that all Player-related React components reside in the containers/Player folder, while all House-related React components reside in the containers/House folder.

Add the packageId value you saved from deploying your package to a new src/constants.ts file in your project:

The UI interacts with the Single Player smart contract variant of the game. This section walks you through each step in the smart contract flow and the corresponding frontend code.

The following frontend code snippets include only the most relevant sections. Refer to the Satoshi Coin Flip Frontend Example repository for complete source code.

As is common in other React projects, App.tsx is where you implement the outer layout:

Like other dApps, you need a "connect wallet" button to enable connecting users' wallets. dApp Kit contains a pre-made ConnectButton React component that you can reuse to help users onboard.

useCurrentAccount() is a React hook the dApp Kit also provides to query the current connected wallet; returning null if there isn't a wallet connection. Leverage this behavior to prevent a user from proceeding further if they haven't connected their wallet yet.

After ensuring that the user has connected their wallet, you can display the two columns described in the previous section:

PlayerSesh and HouseSesh components.

Okay, that's a good start to have an overview of the project. Time to move to initializing the HouseData object. All the frontend logic for calling this lives in the HouseInitialize.tsx component. The component includes UI code, but the logic that executes the transaction follows:

To use a [programmable transaction block]() (PTB) in Sui, create a Transaction . To initiate a Move call, you must know the global identifier of a public function in your smart contract. The global identifier usually takes the following form:

In this example, it is:

There are a few parameters that you need to pass into initialize_house_data() Move function: the HouseCap ID, the House stake, and the House BLS public key:

Now sign and execute the transaction block. dApp Kit provides a React hook useSignAndExecuteTransaction() to streamline this process. This hook, when executed, prompts the UI for you to approve, sign, and execute the transaction block. You can configure the hook with the showObjectChanges option to return the newly-created HouseData shared object as the result of the transaction block. This HouseData object is important as you use it as input for later Move calls, so save its ID somewhere.

Great, now you know how to initialize the HouseData shared object. Move to the next function call.

In this game, the users must create a Counter object to start the game. So there should be a place in the Player column UI to list the existing Counter object information for the player to choose. It seems likely that you will reuse the fetching logic for the Counter object in several places in your UI, so it's good practice to isolate this logic into a React hook, which you call useFetchCounterNft() in useFetchCounterNft.ts :

This hook logic is very basic: if there is no current connected wallet, return empty data; otherwise, fetch the Counter object and return it. dApp Kit provides a React hook, useSuiClientQuery() , that enables interaction with [Sui RPC]() methods. Different RPC methods require different parameters. To fetch the object owned by a known address, use the [getOwnedObjects]() query .

Now, pass the address of the connected wallet, as well as the global identifier for the Counter . This is in similar format to the global identifier type for function calls:

${PACKAGE_ID}::counter_nft::Counter

That's it, now put the hook into the UI component PlayerListCounterNft.tsx and display the data:

For the case when there is no existing Counter object, mint a new Counter for the connected wallet. Also add the minting logic into PlayerListCounterNft.tsx when the user clicks the button. You already know how to build and execute a Move call with TransactionBlock and initialize_house_data() , you can implement a similar call here.

As you might recall with Transaction , outputs from the transaction can be inputs for the next transaction. Call counter_nft::mint() , which returns the newly created Counter object, and use it as input for counter_nft::transfer_to_sender() to transfer the Counter object to the caller wallet:

Great, now you can create the game with the created Counter object. Isolate the game creation logic into PlayerCreateGame.tsx . There is one more thing to keep in mind - to flag an input as an on-chain object, you should use txb.object() with the corresponding object ID.

One final step remains: settle the game. There are a couple of ways you can use the UI to settle the game:

Event subscriptions are deprecated. To learn future-safe methods to work with events, see [Using Events]() .

All of this logic is in HouseFinishGame.tsx :

To get the underlying SuiClient instance from the SDK, use useSuiClient() . You want to subscribe to events whenever the HouseFinishGame component loads. To do this, use the React hook useEffect() from the core React library.

SuiClient exposes a method called subscribeEvent() that enables you to subscribe to a variety of event types. SuiClient::subscribeEvent() is actually a thin wrapper around the RPC method [suix_subscribeEvent]() .

The logic is that whenever a new game starts, you want to settle the game immediately. The necessary event to achieve this is the Move event type called single_player_satoshi::NewGame . If you inspect the parsed payload of the event through event.parsedJson , you can see the corresponding event fields declared in the smart contract. In this case, you just need to use two fields, the Game ID and the VRF input.

The next steps are similar to the previous Move calls, but you have to use the BLS private key to sign the VRF input and then pass the Game ID, signed VRF input and HouseData ID to the single_player_satoshi::finish_game() Move call.

Last but not least, remember to unsubscribe from the event whenever the HouseFinishGame component dismounts. This is important as you might not want to subscribe to the same event multiple times.

Congratulations, you completed the frontend. You can carry the lessons learned here forward when using the dApp Kit to build your next Sui project.

Smart contracts

In this part of the guide, you write the Move contracts that manage the house and set up the coin-flip logic. The first step is to set up a Move package for storing your Move modules.

To follow along with this guide, set your new Move package to satoshi_flip .

This example uses several modules to create a package for the Satoshi Coin Flip game. The first module is house_data.move . You need to store the game's data somewhere, and in this module you create a shared object for all house data.

Create a new file in the sources directory with the name house_data.move and populate the file with the following code:

There are few details to take note of in this code:

Next, add some more code to this module:

So far, you've set up the data structures within the module. Now, create a function that initializes the house data and shares the HouseData object:

With the house data initialized, you also need to add some functions that enable some important administrative tasks for the house to perform:

All of these functions contain an assert! call that ensures only the house can call them:

You have established the data structure of this module, but without the appropriate functions this data is not accessible. Now add helper functions that return mutable references, read-only references, and test-only functions:

And with that, your house_data.move code is complete.

In the same sources directory, now create a file named counter_nft.move . A Counter object is used as the VRF input for every game that a player plays. First, populate the file with the following:

This might look familiar from the house module. You set the module name, import functions from the standard library, and initialize the Counter object. The Counter object has the key ability, but does not have store - this prevents the object from being transferable.

In addition, you create mint and transfer_to_sender functions used when the game is set up to create the Counter object (with an initial count of 0) and transfer the object to the sender of the transaction. And finally a burn function to allow deletion of the Counter .

You have a Counter object, as well as functions that initialize and burn the object, but you need a way to increment the counter. Add the following code to the module:

The get_vrf_input_and_increment function is the core of this module. The function takes a mutable reference to the Counter object that the mint function creates, then appends the Counter object's current count to its ID and returns the result as a vector . The function then calls the internal increment function to increment the count by one.

This code also adds a count function that returns the current count, and a test-only function that calls the burn function.

Lastly, you need a game module and object that can create a new game, distribute funds after the game, and potentially cancel games. Because this is a one-player game, create an address-owned object rather than a shared object .

Create the game module. In the sources directory, create a new file called single_player_satoshi.move and populate with the following:

This code follows the same pattern as the others. First, you include the respective imports, although this time the imports are not only from the standard library but also include modules created previously in this example. You also create several constants (in upper case), as well as constants used for errors (Pascal case prefixed with E).

Lastly in this section, you also create structs for two events to emit. Indexers consume emitted events, which enables you to track these events through API services, or your own indexer. In this case, the events are for when a new game begins (NewGame) and for the outcome of a game when it has finished (Outcome).

Add a struct to the module:

The Game struct represents a single game and all its information, including the epoch the player placed the bet (guess_placed_epoch), bet (total_stake), guess , address of the player , vrf_input , and the fee the house collects (fee_bp).

Now take a look at the main function in this game, finish_game :

Now add a function that handles game disputes:

This function, dispute_and_win , ensures that no bet can live in "purgatory". After a certain amount of time passes, the player can call this function and get all of their funds back.

The rest of the functions are accessors and helper functions used to retrieve values, check if values exist, initialize the game, and so on:

This represents a basic example of a coin flip backend in Move. The game module, single_player_satoshi , is prone to MEV attacks, but the user experience for the player is streamlined. Another example game module, mev_attack_resistant_single_player_satoshi , exists that is MEV-resistant, but has a slightly downgraded user experience (two player-transactions per game).

You can read more about both versions of the game, and view the full source code for all the modules in the Satoshi Coin Flip repository .

Now that you have written our contracts, it's time to deploy them.

See Publish a Package for a more detailed guide on publishing packages or Sui Client CLI for a complete reference of client commands in the Sui CLI.

Before publishing your code, you must first initialize the Sui Client CLI, if you haven't already. To do so, in a terminal or console at the root directory of the project enter sui client . If you receive the following response, complete the remaining instructions:

Enter y to proceed. You receive the following response:

Leave this blank (press Enter). You receive the following response:

Select 0 . Now you should have a Sui address set up.

Next, configure the Sui CLI to use testnet as the active environment, as well. If you haven't already set up a testnet environment, do so by running the following command in a terminal or console:

Run the following command to activate the testnet environment:

Before being able to publish your package to Testnet, you need Testnet SUI tokens. To get some, visit the online faucet at https://faucet.sui.io/ . For other ways to get SUI in your Testnet account, see Get SUI Tokens .

Now that you have an account with some Testnet SUI, you can deploy your contracts. To publish your package, use the following command in the same terminal or console:

For the gas budget, use a standard value such as 20000000 .

The output of this command contains a packageID value that you need to save to use the package.

Partial snippet of CLI deployment output.

Save the PackageID and the ObjectID of the HouseCap object you receive in your own response to connect to your frontend .

In this case, the PackageID is 0x4120b39e5d94845aa539d4b830743a7433fd8511bdcf3841f98080080f327ca8 and the HouseCap ID is 0xfa1f6edad697afca055749fedbdee420b6cdba3edc2f7fd4927ed42f98a7e63a .

Well done. You have written and deployed the Move package! □

To turn this into a complete dApp, you need to create a frontend .

In this final part of the dApp example, you build a frontend (UI) that allows end users to place bets and take profits, and lets the admin manage the house.

To skip building the frontend and test out your newly deployed package, use the provided Satoshi Coin Flip Frontend Example repository and follow the instructions in the example's README.md file

Before getting started, make sure you have:

The UI of this example demonstrates how to use the dApp Kit instead of serving as a production-grade product, so the Player and the House features are in the same UI to simplify the process. In a production solution, your frontend would only contain functionality dedicated to the Player, with a backend service carrying out the interactions with House functions in the smart contracts.

The UI has two columns:

The first step is to set up the client app. Run the following command to scaffold a new app.

or

Structure the project folder according to the UI layout, meaning that all Player-related React components reside in the containers/Player folder, while all House-related React components reside in the containers/House folder.

Add the packageId value you saved from deploying your package to a new src/constants.ts file in your project:

The UI interacts with the Single Player smart contract variant of the game. This section walks you through each step in the smart contract flow and the corresponding frontend code.

The following frontend code snippets include only the most relevant sections. Refer to the Satoshi Coin Flip Frontend Example repository for complete source code.

As is common in other React projects, App.tsx is where you implement the outer layout:

Like other dApps, you need a "connect wallet" button to enable connecting users' wallets. dApp Kit contains a pre-made ConnectButton React component that you can reuse to help users onboard.

useCurrentAccount() is a React hook the dApp Kit also provides to query the current connected wallet; returning null if there isn't a wallet connection. Leverage this behavior to prevent a user from proceeding further if they haven't connected their wallet yet.

After ensuring that the user has connected their wallet, you can display the two columns described in the previous section: PlayerSesh and HouseSesh components.

Okay, that's a good start to have an overview of the project. Time to move to initializing the HouseData object. All the frontend logic for calling this lives in the HouseInitialize.tsx component. The component includes UI code, but the logic that executes the transaction follows:

To use a programmable transaction block (PTB) in Sui, create a Transaction . To initiate a Move call, you must know the global identifier of a public function in your smart contract. The global identifier usually takes the following form:

In this example, it is:

There are a few parameters that you need to pass into initialize_house_data() Move function: the HouseCap ID, the House stake, and the House BLS public key:

Now sign and execute the transaction block. dApp Kit provides a React hook useSignAndExecuteTransaction() to streamline this process. This hook, when executed, prompts the UI for you to approve, sign, and execute the transaction block. You can configure the hook with the showObjectChanges option to return the newly-created HouseData shared object as the result of the transaction block. This HouseData object is important as you use it as input for later Move calls, so save its ID somewhere.

Great, now you know how to initialize the HouseData shared object. Move to the next function call.

In this game, the users must create a Counter object to start the game. So there should be a place in the Player column UI to list the existing Counter object information for the player to choose. It seems likely that you will reuse the fetching logic for the Counter object in several places in your UI, so it's good practice to isolate this logic into a React hook, which you call useFetchCounterNft() in useFetchCounterNft.ts :

This hook logic is very basic: if there is no current connected wallet, return empty data; otherwise, fetch the Counter object and return it. dApp Kit provides a React hook, useSuiClientQuery() , that enables interaction with Sui RPC methods. Different RPC methods require different parameters. To fetch the object owned by a known address, use the getOwnedObjects query .

Now, pass the address of the connected wallet, as well as the global identifier for the Counter . This is in similar format to the global identifier type for function calls:

${PACKAGE_ID}::counter_nft::Counter

That's it, now put the hook into the UI component PlayerListCounterNft.tsx and display the data:

For the case when there is no existing Counter object, mint a new Counter for the connected wallet. Also add the minting logic into PlayerListCounterNft.tsx when the user clicks the button. You already know how to build and execute a Move call with TransactionBlock and initialize_house_data() , you can implement a similar call here.

As you might recall with Transaction , outputs from the transaction can be inputs for the next transaction. Call counter_nft::mint() , which returns the newly created Counter object, and use it as input for counter_nft::transfer_to_sender() to transfer the Counter object to the caller wallet:

Great, now you can create the game with the created Counter object. Isolate the game creation logic into PlayerCreateGame.tsx . There is one more thing to keep in mind - to flag an input as an on-chain object, you should use txb.object() with the corresponding object ID.

One final step remains: settle the game. There are a couple of ways you can use the UI to settle the game:

Event subscriptions are deprecated. To learn future-safe methods to work with events, see Using Events .

All of this logic is in HouseFinishGame.tsx :

To get the underlying SuiClient instance from the SDK, use useSuiClient() . You want to subscribe to events whenever the HouseFinishGame component loads. To do this, use the React hook useEffect() from the core React library.

SuiClient exposes a method called subscribeEvent() that enables you to subscribe to a variety of event types. SuiClient::subscribeEvent() is actually a thin wrapper around the RPC method suix_subscribeEvent .

The logic is that whenever a new game starts, you want to settle the game immediately. The necessary event to achieve this is the Move event type called single_player_satoshi::NewGame . If you inspect the parsed payload of the event through event.parsedJson , you can see the corresponding event fields declared in the smart contract. In this case, you just need to use two fields, the Game ID and the VRF input.

The next steps are similar to the previous Move calls, but you have to use the BLS private key to sign the VRF input and then pass the Game ID, signed VRF input and HouseData ID to the single_player_satoshi::finish_game() Move call.

Last but not least, remember to unsubscribe from the event whenever the HouseFinishGame component dismounts. This is important as you might not want to subscribe to the same event multiple times.

Congratulations, you completed the frontend. You can carry the lessons learned here forward when using the dApp Kit to build your next Sui project.

Finished package

This represents a basic example of a coin flip backend in Move. The game module, single_player_satoshi , is prone to MEV attacks, but the user experience for the player is streamlined. Another example game module, mev_attack_resistant_single_player_satoshi , exists that is MEV-resistant, but has a slightly downgraded user experience (two player-transactions per game).

You can read more about both versions of the game, and view the full source code for all the modules in the Satoshi Coin Flip repository .

Now that you have written our contracts, it's time to deploy them.

See Publish a Package for a more detailed guide on publishing packages or Sui Client CLI for a complete reference of client commands in the Sui CLI.

Before publishing your code, you must first initialize the Sui Client CLI, if you haven't already. To do so, in a terminal or console at the root directory of the project enter sui client . If you receive the following response, complete the remaining instructions:

Enter y to proceed. You receive the following response:

Leave this blank (press Enter). You receive the following response:

Select 0 . Now you should have a Sui address set up.

Next, configure the Sui CLI to use testnet as the active environment, as well. If you haven't already set up a testnet environment, do so by running the following command in a terminal or console:

Run the following command to activate the testnet environment:

Before being able to publish your package to Testnet, you need Testnet SUI tokens. To get some, visit the online faucet at https://faucet.sui.io/ . For other ways to get SUI in your Testnet account, see Get SUI Tokens .

Now that you have an account with some Testnet SUI, you can deploy your contracts. To publish your package, use the following command in the same terminal or console:

For the gas budget, use a standard value such as 20000000 .

The output of this command contains a packageID value that you need to save to use the package.

Partial snippet of CLI deployment output.

Save the PackageID and the ObjectID of the HouseCap object you receive in your own response to connect to your frontend .

In this case, the PackageID is 0x4120b39e5d94845aa539d4b830743a7433fd8511bdcf3841f98080080f327ca8 and the HouseCap ID is 0xfa1f6edad697afca055749fedbdee420b6cdba3edc2f7fd4927ed42f98a7e63a .

Well done. You have written and deployed the Move package! □

To turn this into a complete dApp, you need to create a frontend .

In this final part of the dApp example, you build a frontend (UI) that allows end users to place bets and take profits, and lets the admin manage the house.

To skip building the frontend and test out your newly deployed package, use the provided Satoshi Coin Flip Frontend Example repository and follow the instructions in the example's README.md file

Before getting started, make sure you have:

The UI of this example demonstrates how to use the dApp Kit instead of serving as a production-grade product, so the Player and the House features are in the same UI to simplify the process. In a production solution, your frontend would only contain functionality dedicated to the Player, with a backend service carrying out the interactions with House functions in the smart contracts.

The UI has two columns:

The first step is to set up the client app. Run the following command to scaffold a new app.

or

Structure the project folder according to the UI layout, meaning that all Player-related React components reside in the containers/Player folder, while all House-related React components reside in the containers/House folder.

Add the packageId value you saved from deploying your package to a new src/constants.ts file in your project:

The UI interacts with the Single Player smart contract variant of the game. This section walks you through each step in the smart contract flow and the corresponding frontend code.

The following frontend code snippets include only the most relevant sections. Refer to the Satoshi Coin Flip Frontend Example repository for complete source code.

As is common in other React projects, App.tsx is where you implement the outer layout:

Like other dApps, you need a "connect wallet" button to enable connecting users' wallets. dApp Kit contains a pre-made

ConnectButton React component that you can reuse to help users onboard.

useCurrentAccount() is a React hook the dApp Kit also provides to query the current connected wallet; returning null if there isn't a wallet connection. Leverage this behavior to prevent a user from proceeding further if they haven't connected their wallet yet.

After ensuring that the user has connected their wallet, you can display the two columns described in the previous section: PlayerSesh and HouseSesh components.

Okay, that's a good start to have an overview of the project. Time to move to initializing the HouseData object. All the frontend logic for calling this lives in the HouseInitialize.tsx component. The component includes UI code, but the logic that executes the transaction follows:

To use a programmable transaction block (PTB) in Sui, create a Transaction . To initiate a Move call, you must know the global identifier of a public function in your smart contract. The global identifier usually takes the following form:

In this example, it is:

There are a few parameters that you need to pass into initialize_house_data() Move function: the HouseCap ID, the House stake, and the House BLS public key:

Now sign and execute the transaction block. dApp Kit provides a React hook useSignAndExecuteTransaction() to streamline this process. This hook, when executed, prompts the UI for you to approve, sign, and execute the transaction block. You can configure the hook with the showObjectChanges option to return the newly-created HouseData shared object as the result of the transaction block. This HouseData object is important as you use it as input for later Move calls, so save its ID somewhere.

Great, now you know how to initialize the HouseData shared object. Move to the next function call.

In this game, the users must create a Counter object to start the game. So there should be a place in the Player column UI to list the existing Counter object information for the player to choose. It seems likely that you will reuse the fetching logic for the Counter object in several places in your UI, so it's good practice to isolate this logic into a React hook, which you call useFetchCounterNft() in useFetchCounterNft.ts :

This hook logic is very basic: if there is no current connected wallet, return empty data; otherwise, fetch the Counter object and return it. dApp Kit provides a React hook, useSuiClientQuery() , that enables interaction with Sui RPC methods. Different RPC methods require different parameters. To fetch the object owned by a known address, use the getOwnedObjects query .

Now, pass the address of the connected wallet, as well as the global identifier for the Counter . This is in similar format to the global identifier type for function calls:

${PACKAGE_ID}::counter_nft::Counter

That's it, now put the hook into the UI component PlayerListCounterNft.tsx and display the data:

For the case when there is no existing Counter object, mint a new Counter for the connected wallet. Also add the minting logic into PlayerListCounterNft.tsx when the user clicks the button. You already know how to build and execute a Move call with TransactionBlock and initialize_house_data() , you can implement a similar call here.

As you might recall with Transaction , outputs from the transaction can be inputs for the next transaction. Call counter_nft::mint() , which returns the newly created Counter object, and use it as input for counter_nft::transfer_to_sender() to transfer the Counter object to the caller wallet:

Great, now you can create the game with the created Counter object. Isolate the game creation logic into PlayerCreateGame.tsx . There is one more thing to keep in mind - to flag an input as an on-chain object, you should use txb.object() with the corresponding object ID.

One final step remains: settle the game. There are a couple of ways you can use the UI to settle the game:

Event subscriptions are deprecated. To learn future-safe methods to work with events, see Using Events .

All of this logic is in HouseFinishGame.tsx :

To get the underlying SuiClient instance from the SDK, use useSuiClient() . You want to subscribe to events whenever the HouseFinishGame component loads. To do this, use the React hook useEffect() from the core React library.

SuiClient exposes a method called subscribeEvent() that enables you to subscribe to a variety of event types.

SuiClient::subscribeEvent() is actually a thin wrapper around the RPC method suix_subscribeEvent .

The logic is that whenever a new game starts, you want to settle the game immediately. The necessary event to achieve this is the Move event type called single_player_satoshi::NewGame . If you inspect the parsed payload of the event through event.parsedJson , you can see the corresponding event fields declared in the smart contract. In this case, you just need to use two fields, the Game ID and the VRF input.

The next steps are similar to the previous Move calls, but you have to use the BLS private key to sign the VRF input and then pass the Game ID, signed VRF input and HouseData ID to the single_player_satoshi::finish_game() Move call.

Last but not least, remember to unsubscribe from the event whenever the HouseFinishGame component dismounts. This is important as you might not want to subscribe to the same event multiple times.

Congratulations, you completed the frontend. You can carry the lessons learned here forward when using the dApp Kit to build your next Sui project.

Frontend

In this final part of the dApp example, you build a frontend (UI) that allows end users to place bets and take profits, and lets the admin manage the house.

To skip building the frontend and test out your newly deployed package, use the provided Satoshi Coin Flip Frontend Example repository and follow the instructions in the example's README.md file

Before getting started, make sure you have:

The UI of this example demonstrates how to use the dApp Kit instead of serving as a production-grade product, so the Player and the House features are in the same UI to simplify the process. In a production solution, your frontend would only contain functionality dedicated to the Player, with a backend service carrying out the interactions with House functions in the smart contracts.

The UI has two columns:

The first step is to set up the client app. Run the following command to scaffold a new app.

or

Structure the project folder according to the UI layout, meaning that all Player-related React components reside in the containers/Player folder, while all House-related React components reside in the containers/House folder.

Add the packageId value you saved from deploying your package to a new src/constants.ts file in your project:

The UI interacts with the Single Player smart contract variant of the game. This section walks you through each step in the smart contract flow and the corresponding frontend code.

The following frontend code snippets include only the most relevant sections. Refer to the Satoshi Coin Flip Frontend Example repository for complete source code.

As is common in other React projects, App.tsx is where you implement the outer layout:

Like other dApps, you need a "connect wallet" button to enable connecting users' wallets. dApp Kit contains a pre-made ConnectButton React component that you can reuse to help users onboard.

useCurrentAccount() is a React hook the dApp Kit also provides to query the current connected wallet; returning null if there isn't a wallet connection. Leverage this behavior to prevent a user from proceeding further if they haven't connected their wallet yet.

After ensuring that the user has connected their wallet, you can display the two columns described in the previous section: PlayerSesh and HouseSesh components.

Okay, that's a good start to have an overview of the project. Time to move to initializing the HouseData object. All the frontend logic for calling this lives in the HouseInitialize.tsx component. The component includes UI code, but the logic that executes the transaction follows:

To use a programmable transaction block (PTB) in Sui, create a Transaction . To initiate a Move call, you must know the global identifier of a public function in your smart contract. The global identifier usually takes the following form:

In this example, it is:

There are a few parameters that you need to pass into initialize_house_data() Move function: the HouseCap ID, the House stake, and the House BLS public key:

Now sign and execute the transaction block. dApp Kit provides a React hook useSignAndExecuteTransaction() to streamline this process. This hook, when executed, prompts the UI for you to approve, sign, and execute the transaction block. You can configure the hook with the showObjectChanges option to return the newly-created HouseData shared object as the result of the transaction block. This HouseData object is important as you use it as input for later Move calls, so save its ID somewhere.

Great, now you know how to initialize the HouseData shared object. Move to the next function call.

In this game, the users must create a Counter object to start the game. So there should be a place in the Player column UI to list the existing Counter object information for the player to choose. It seems likely that you will reuse the fetching logic for the Counter object in several places in your UI, so it's good practice to isolate this logic into a React hook, which you call useFetchCounterNft() in useFetchCounterNft.ts :

This hook logic is very basic: if there is no current connected wallet, return empty data; otherwise, fetch the Counter object and return it. dApp Kit provides a React hook, useSuiClientQuery() , that enables interaction with Sui RPC methods. Different RPC methods require different parameters. To fetch the object owned by a known address, use the getOwnedObjects query .

Now, pass the address of the connected wallet, as well as the global identifier for the Counter . This is in similar format to the global identifier type for function calls:

${PACKAGE_ID}::counter_nft::Counter

That's it, now put the hook into the UI component PlayerListCounterNft.tsx and display the data:

For the case when there is no existing Counter object, mint a new Counter for the connected wallet. Also add the minting logic into PlayerListCounterNft.tsx when the user clicks the button. You already know how to build and execute a Move call with TransactionBlock and initialize_house_data() , you can implement a similar call here.

As you might recall with Transaction , outputs from the transaction can be inputs for the next transaction. Call counter_nft::mint() , which returns the newly created Counter object, and use it as input for counter_nft::transfer_to_sender() to transfer the Counter object to the caller wallet:

Great, now you can create the game with the created Counter object. Isolate the game creation logic into PlayerCreateGame.tsx . There is one more thing to keep in mind - to flag an input as an on-chain object, you should use txb.object() with the corresponding object ID.

One final step remains: settle the game. There are a couple of ways you can use the UI to settle the game:

Event subscriptions are deprecated. To learn future-safe methods to work with events, see Using Events .

All of this logic is in HouseFinishGame.tsx :

To get the underlying SuiClient instance from the SDK, use useSuiClient() . You want to subscribe to events whenever the HouseFinishGame component loads. To do this, use the React hook useEffect() from the core React library.

SuiClient exposes a method called subscribeEvent() that enables you to subscribe to a variety of event types. SuiClient::subscribeEvent() is actually a thin wrapper around the RPC method suix_subscribeEvent .

The logic is that whenever a new game starts, you want to settle the game immediately. The necessary event to achieve this is the Move event type called single_player_satoshi::NewGame . If you inspect the parsed payload of the event through event.parsedJson , you can see the corresponding event fields declared in the smart contract. In this case, you just need to use two fields, the Game ID and the VRF input.

The next steps are similar to the previous Move calls, but you have to use the BLS private key to sign the VRF input and then pass the Game ID, signed VRF input and HouseData ID to the single_player_satoshi::finish_game() Move call.

Last but not least, remember to unsubscribe from the event whenever the HouseFinishGame component dismounts. This is important as you might not want to subscribe to the same event multiple times.

Congratulations, you completed the frontend. You can carry the lessons learned here forward when using the dApp Kit to build your next Sui project.