Trustless Swap

This guide is rated as advanced .

You can expect advanced guides to take 2 hours or more of dedicated time. The length of time necessary to fully understand some of the concepts raised in this guide might increase this estimate.

You can view the complete source code for this app example in the Sui repository.

This guide demonstrates how to make an app that performs atomic swaps on Sui. Atomic swaps are similar to escrows but without requiring a trusted third party.

There are three main sections to this guide:

The guide also shows how to build an app that:

Before getting started, make sure you have:

Installed the latest version of Sui .

Configured a valid network environment , as the guide has you deploy the module on Testnet.

Acquired Devnet or Testnet tokens for development purposes.

https://faucet.sui.io/ : Visit the online faucet to request SUI tokens. You can refresh your browser to perform multiple requests, but the requests are rate-limited per IP address.

Read the basics of shared versus owned objects .

To begin, create a new folder on your system titled trading that holds all your files. Inside that folder, create three more folders: api , contracts , and frontend . It's important to keep this directory structure as some helper scripts in this example target these directories by name. Different projects have their own directory structure, but it's common to split code into functional groups to help with maintenance.

In this part of the guide, you write the Move contracts that perform the trustless swaps. The guide describes how to create the package from scratch, but you can use a fork or copy of the example code in the Sui repo to follow along instead. See Write a Move Package to learn more about package structure and how to use the Sui CLI to scaffold a new project.

To begin writing your smart contracts, create an escrow folder in your contracts folder (if using recommended directory names). Create a file inside the folder named Move.toml and copy the following code into it. This is the package manifest file. If you want to learn more about the structure of the file, see Package Manifest in The Move Book.

If you are targeting a network other than Testnet, be sure to update the rev value for the Sui dependency.

With your manifest file in place, it's time to start creating the Move assets for this project. In your escrow folder, at the same level as your Move.toml file, create a sources folder. This is the common file structure of a package in Move. Create a new file inside sources titled lock.move . This file contains the logic that locks the object involved in a trade. The complete source code for this file follows and the sections that come after detail its components.

Click the titles at the top of codeblocks to open the relevant source file in GitHub.

lock.move

After a trade is initiated, you don't want the trading party to modify the object they agreed to trade. Imagine you're trading in-game items and you agree to trade a weapon with all its attachments, and its owner strips all its attachments just before the trade.

In a traditional trade, a third party typically holds the items in escrow to make sure they are not tampered with before the trade completes. This requires either trusting that the third party won't tamper with it themselves, paying the third party to ensure that doesn't happen, or both.

In a trustless swap, however, you can use the safety properties of Move to force an item's owner to prove that they have not tampered with the version of the object that you agreed to trade, without involving anyone else.

This is done by requiring that an object that is available for trading is locked with a single-use key , and asking the owner to supply the key when finalizing the trade.

To tamper with the object would require unlocking it, which consumes the key. Consequently, there would no longer be a key to finish the trade.

The lock and key are made single-use by the signatures of the lock and unlock functions. lock accepts any object of type T: store (the store ability is necessary for storing it inside a Locked), and creates both the Locked and its corresponding Key :

lock function in lock.move

The unlock function accepts the Locked and Key by value (which consumes them), and returns the underlying T as long as the correct key has been supplied for the lock:

unlock function in lock.move

Together, they ensure that a lock and key cannot have existed before the lock operation, and will not exist after a successful unlock – it is single use.

Move's type system guarantees that a given Key cannot be re-used (because unlock accepts it by value), but there are some properties that need to be confirmed with tests:

The test starts with a helper function for creating an object, it doesn't matter what kind of object it is, as long as it has the store ability. The test uses Coin , because it comes with a #[test_only] function for minting:

The first test works by creating an object to test, locking it and unlocking it – this should finish executing without aborting. The last two lines exist to keep the Move compiler happy by cleaning up the test coin and test scenario objects, because values in Move are not implicitly cleaned up unless they have the drop ability.

The other test is testing a failure scenario – that an abort happens. It creates two locked objects (this time the values are just u64 s), and use the key from one to try and unlock the other, which should fail (specified using the expected_failure attribute).

Unlike the previous test, the same clean up is not needed, because the code is expected to terminate. Instead, add another abort after the code that you expect to abort (making sure to use a different code for this second abort).

At this point, you have

From your escrow folder, run sui move test in your terminal or console. If successful, you get a response similar to the following that confirms the package builds and your tests pass:

You might notice that the Move compiler creates a build subfolder inside escrow upon a successful build. This folder contains your package's compiled bytecode, code from your package's dependencies, and various other files necessary for the build. At this point, it's enough to just be aware of these files. You don't need to fully understand the contents in build .

Create a new file in your escrow folder titled shared.move . The code in this file creates the shared Escrow object and completes the trading logic. The complete source code for this file follows and the sections that come after detail its components.

shared.move

Trading proceeds in three steps:

You can start by implementing steps two and three, by defining a new type to hold the escrowed object. It holds the escrowed object and an id: UID (because it's an object in its own right), but it also records the sender and intended recipient (to confirm they match when the trade happens), and it registers interest in the first party's object by recording the ID of the key that unlocks the Locked that contains the object.

You also need to create a function for creating the Escrow object. The object is shared because it needs to be accessed by the address that created it (in case the object needs to be returned) and by the intended recipient (to complete the swap).

create function in shared.move

If the second party stops responding, the first party can unlock their object. You need to create a function so the second party can recover their object in the symmetric case as well.

return_to_sender function in shared.move

Finally, you need to add a function to allow the first party to complete the trade.

swap function in shared.move

Tests for the escrow module are more involved than for lock – as they take advantage of test_scenario 's ability to simulate multiple transactions from different senders, and interact with shared objects.

The guide focuses on the test for a successful swap, but you can find a link to all the tests later on.

As with the lock test, start by creating a function to mint a test coin. You also create some constants to represent our transaction senders, ALICE , BOB , and DIANE .

The test body starts with a call to test_scenario::begin and ends with a call to test_scenario::end . It doesn't matter which address you pass to begin , because you pick one of ALICE or BOB at the start of each new transaction you write, so set it to @0x0 :

The first transaction is from BOB who creates a coin and locks it. You must remember the ID of the coin and the ID of the key, which you will need later, and then you transfer the locked object and the key itself to BOB , because this is what would happen in a real transaction: When simulating transactions in a test, you should only keep around primitive values, not whole objects, which would need to be written to chain between transactions.

Write these transactions inside the test_successful_swap function, between the call to begin and end .

Next, ALICE comes along and sets up the Escrow , which locks their coin. They register their interest for BOB' s coin by referencing BOB 's key's ID (ik2):

Finally, BOB completes the trade by calling swap . The take_shared function is used to simulate accepting a shared input. It uses type inference to know that the object must be an Escrow , and finds the last object of this type that was shared (by ALICE in the previous transaction). Similarly, use take_from_sender to simulate accepting owned inputs (in this case, BOB 's lock and key). The coin returned by swap is transferred back to BOB , as if it was called as part of a PTB, followed by a transfer command.

The rest of the test is designed to check that ALICE has BOB 's coin and vice versa. It starts by calling next_tx to make sure the effects of the previous transaction have been committed, before running the necessary checks.

The escrow Move package is now functional: You could publish it on chain and perform trustless swaps by creating transactions. Creating those transactions requires knowing the IDs of Locked , Key , and Escrow objects.

Locked and Key objects are typically owned by the transaction sender, and so can be queried through the Sui RPC, but Escrow objects are shared, and it is useful to be able to query them by their sender and recipient (so that users can see the trades they have offered and received).

Querying Escrow objects by their sender or recipient requires custom indexing, and to make it easy for the indexer to spot relevant transactions, add the following events to escrow.move :

Functions responsible for various aspects of the escrow's lifecycle emit these events. The custom indexer can then subscribe to transactions that emit these events and process only those, rather than the entire chain state:

emit events included in functions from shared.move

You now have shared.move and locked.move files in your sources folder. From the parent escrow folder, run sui move test in your terminal or console. If successful, you get a response similar to the following that confirms the package builds and your tests pass:

Well done. You have written the Move package! □

To turn this into a complete dApp, you need to create a frontend. However, for the frontend to be updated, it has to listen to the blockchain as escrows are made and swaps are fulfilled.

To achieve this, in the next step you create an indexing service.

With the contract adapted to emit events, you can now write an indexer that keeps track of all active Escrow objects and exposes an API for querying objects by sender or recipient.

The indexer is backed by a Prisma DB with the following schema:

schema.prisma

The core of the indexer is an event loop, initialized in a function called setupListeners .

The indexer queries events related to the escrow module, using a queryEvent filter, and keeps track of a cursor representing the latest event it has processed so it can resume indexing from the right place even if it is restarted. The filter is looking for any events whose type is from the escrow module of the Move package (see the event-indexer.ts code that follows).

The core event job works by polling: It queries RPC for events following its latest cursor and sends them to a callback for processing. If it detects more than one page of new events, it immediately requests the next page. Otherwise, the job waits for the next polling interval before checking again.

event-indexer.ts

The callback is responsible for reading the event and updating the database accordingly. For demo purposes, SQLite is being used, and so you need to issue a separate UPSERT to the database for each escrowed object. In a production setting, however, you would want to batch requests to the database to optimize data flow.

escrow-handler.ts

The data that the indexer captures can then be served over an API, so that a frontend can read it. Follow the next section to implement the API in TypeScript, to run on Node, using Express.

You want your API to accept the query string in the URL as the parameters for database WHERE query. Hence, you want a utility that can extract and parse the URL query string into valid query parameters for Prisma. With the parseWhereStatement() function, the callers filter the set of keys from the URL query string and transforms those corresponding key-value pairs into the correct format for Prisma.

parseWhereStatement in api-queries.ts

Pagination is another crucial part to ensure your API returns sufficient and/or ordered chunk of information instead of all the data that might be the vector for a DDOS attack. Similar to WHERE parameters , define a set of keys in the URL query string to be accepted as valid pagination parameters. The parsePaginationForQuery() utility function helps to achieve this by filtering the pre-determined keys sort , limit , cursor and parsing corresponding key-value pairs into ApiPagination that Prisma can consume.

In this example, the id field of the model in the database as the cursor that allows clients to continue subsequent queries with the next page.

parsePaginationForQuery in api-queries.ts

All the endpoints are defined in server.ts , particularly, there are two endpoints:

You define a list of valid query keys, such as deleted , creator , keyId , and objectId for Locked data and cancelled , swapped , recipient , and sender for Escrow data. Pass the URL query string into the pre-defined utilities to output the correct parameters that Prisma can use.

server.ts

Now that you have an indexer and an API service, you can deploy your move package and start the indexer and API service.

Install dependencies by running pnpm install --ignore-workspace or yarn install --ignore-workspace .

Setup the database by running pnpm db:setup:dev or yarn db:setup:dev .

Deploy the Sui package

Deployment instructions

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.

Use the following command to list your available environments:

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, run the following command:

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.

There are some helper functions to publish the smart contracts so you can create some demo data (for Testnet). The helper function to publish the smart contrqcts expects built smart contracts in both the escrow and demo directories. Run sui move build in both directories, if necessary. Be sure to update the Sui dependency in the manifest to point to the correct source based on your environment.

To publish the smart contracts and produce demo data:

Publish the smart contracts by running the following command from your api folder:

If successful, demo-contract.json and escrow-contract.json are created in the backend root directory. These files contain the contract addresses and are used by the backend and frontend to interact with the contracts.

Produce demo non-locked and locked objects

Produce demo escrows

If you want to reset the database (for a clean demo, for example), run pnpm db:reset:dev && pnpm db:setup:dev or yarn db:reset:dev && yarn db:setup:dev .

Run both the API and the indexer by running pnpm dev or yarn dev .

Visit http://localhost:3000/escrows or http://localhost:3000/locked

You should now have an indexer running.

With the code successfully deployed on Testnet, you can now create a frontend to display the trading data and to allow users to interact with the Move modules.

In this final part of the app example, you build a frontend (UI) that allows end users to discover trades and interact with listed escrows.

You can view the complete source code for this app example in the Sui repository.

Before getting started, make sure you have:

The UI design consists of three parts:

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

When asked for a name for your dApp, provide one of your liking. The dApp scaffold gets created in a new directory with the name you provide. This is convenient to keep your working code separate from the example source code that might already populate this folder. The codeblocks that follow point to the code in the default example location. Be aware the path to your own code includes the dApp name you provide.

First, set up import aliases to make the code more readable and maintainable. This allows you to import files using @/ instead of relative paths.

Replace the content of tsconfig.json with the following:

The paths option under compilerOptions is what defines the aliasing for TypeScript. Here, the alias @/ *is mapped to the ./src/*

directory, meaning that any time you use @/ , TypeScript resolves it as a reference to the src folder. This setup reduces the need for lengthy relative paths when importing files in your project.

Replace the content of vite.config.ts with the following:

Vite also needs to be aware of the aliasing to resolve imports correctly during the build process. In the resolve.alias configuration of vite.config.ts , we map the alias @ to the /src directory.

To streamline the styling process and keep the codebase clean and maintainable, this guide uses Tailwind CSS, which provides utility-first CSS classes to rapidly build custom designs. Run the following command from the base of your dApp project to add Tailwind CSS and its dependencies:

Next, generate the Tailwind CSS configuration file by running the following:

Replace the content of tailwind.config.js with the following:

Add the src/styles/ directory and add base.css :

First, deploy your package via the scripts in the api directory .

Then, create a src/constants.ts file and fill it with the following:

If you create a dApp using a project name so that your src files are in a subfolder of frontend , be sure to add another nesting level (../) to the import statements.

Create a src/utils/ directory and add the following file:

Create a src/components/ directory and add the following components:

ExplorerLink.tsx

InfiniteScrollArea.tsx

Loading.tsx

SuiObjectDisplay.tsx

Install the necessary dependencies:

The imported template only has a single page. To add more pages, you need to set up routing.

First, install the necessary dependencies:

Then, create a src/routes/ directory and add index.tsx . This file contains the routing configuration:

Add the following respective files to the src/routes/ directory:

root.tsx . This file contains the root component that is rendered on every page:

LockedDashboard.tsx . This file contains the component for the Manage Objects page.

EscrowDashboard.tsx . This file contains the component for the Escrows page.

Update src/main.tsx by replacing the App component with the RouterProvider and replace "dark" with "light" in the Theme component:

Note that dApp Kit provides a set of hooks for making query and mutation calls to the Sui blockchain. These hooks are thin wrappers around query and mutation hooks from @tanstack/react-query .

Create src/components/Header.tsx . This file contains the navigation links and the connect wallet button:

The dApp Kit comes with a pre-built React.js component called ConnectButton displaying a button to connect and disconnect a wallet. The connecting and disconnecting wallet logic is handled seamlessly so you don't need to worry about repeating yourself doing the same logic all over again.

At this point, you have a basic routing setup. Run your app and ensure you can:

Note, the styles should be applied. The Header component should look like this:

All the type definitions are in src/types/types.ts . Create this file and add the following:

ApiLockedObject and ApiEscrowObject represent the Locked and Escrow indexed data model the indexing and API service return.

EscrowListingQuery and LockedListingQuery are the query parameters model to provide to the API service to fetch from the endpoints /escrow and /locked accordingly.

Now, display the objects owned by the connected wallet address. This is the Manage Objects page.

First add this file src/components/locked/LockOwnedObjects.tsx :

Fetch the owned objects directly from Sui blockchain using the useSuiClientInfiniteQuery() hook from dApp Kit . This hook is a thin wrapper around Sui blockchain RPC calls, reference the documentation to learn more about these RPC hooks . Basically, supply the RPC endpoint you want to execute, in this case it's the getOwnedObjects endpoint . Supply the connected wallet account as the owner . The returned data is stored inside the cache at query key getOwnedObjects . In a future step you invalidate this cache after a mutation succeeds, so the data will be re-fetched automatically.

Next, update src/routes/LockedDashboard.tsx to include the LockOwnedObjects component:

Run your app and ensure you can:

If you don't see any objects, you might need to create some demo data or connect your wallet. You can mint objects after completing the next steps.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's create and examine the execute transaction hook.

Create src/hooks/useTransactionExecution.ts :

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useSuiClient() hook from dApp Kit to retrieve the Sui client instance configured in src/main.tsx . The useSignTransaction() function is another hook from dApp kit that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, use the executeTransaction() on the Sui client instance of the Sui TypeScript SDK.

The full source code of the demo bear smart contract is available at Trading Contracts Demo directory

You need a utility function to create a dummy object representing a real world asset so you can use it to test and demonstrate escrow users flow on the UI directly.

Create src/mutations/demo.ts :

As previously mentioned, this example uses @tanstack/react-query to query, cache, and mutate server state. Server state is data only available on remote servers, and the only way to retrieve or update this data is by interacting with these remote servers. In this case, it could be from an API or directly from Sui blockchain RPC.

When you execute a transaction call to mutate data on the Sui blockchain, use the useMutation() hook. The useMutation() hook accepts several inputs, however, you only need two of them for this example. The first parameter, mutationFn , accepts the function to execute the main mutating logic, while the second parameter, onSuccess , is a callback that runs when the mutating logic succeeds.

The main mutating logic includes executing a Move call of a package named demo_bear::new to create a dummy bear object and transferring it to the connected wallet account, all within the same Transaction . This example reuses the executeTransaction() hook from the Execute Transaction Hook step to execute the transaction.

Another benefit of wrapping the main mutating logic inside useMutation() is that you can access and manipulate the cache storing server state. This example fetches the cache from remote servers by using query call in an appropriate callback. In this case, it is the onSuccess callback. When the transaction succeeds, invalidate the cache data at the cache key called getOwnedObjects , then @tanstack/react-query handles the re-fetching mechanism for the invalidated data automatically. Do this by using invalidateQueries() on the @tanstack/react-query configured client instance retrieved by useQueryClient() hook in the Set up Routing step.

Now the logic to create a dummy bear object exists. You just need to attach it into the button in the header.

Header.tsx

Run your app and ensure you can:

To lock the object, execute the lock Move function identified by {PACKAGE_ID}::lock::lock . The implementation is similar to what's in previous mutation functions, use useMutation() from @tanstack/react-query to wrap the main logic inside it. The lock function requires an object to be locked and its type because our smart contract lock function is generic and requires type parameters. After creating a Locked object and its Key object, transfer them to the connected wallet account within the same transaction block.

It's beneficial to extract logic of locking owned objects into a separated mutating function to enhance discoverability and encapsulation.

Create src/mutations/locked.ts :

Update src/components/locked/LockOwnedObjects.tsx to include the useLockObjectMutation hook:

LockOwnedObjects.tsx

Run your app and ensure you can:

The object should disappear from the list of owned objects. You view and unlock locked objects in later steps.

Let's take a look at the My Locked Objects tab by examining src/components/locked/OwnedLockedList.tsx . Focus on the logic on how to retrieve this list.

OwnedLockedList.tsx

This instance of useSuiClientInfiniteQuery() is similar to the one in the LockOwnedObjects component. The difference is that it fetches the locked objects instead of the owned objects. The Locked object is a struct type in the smart contract, so you need to supply the struct type to the query call as a filter . The struct type is usually identified by the format of {PACKAGE_ID}::{MODULE_NAME}::{STRUCT_TYPE} .

The (src/components/locked/LockedObject.tsx) component is mainly responsible for mapping an on-chain SuiObjectData Locked object to its corresponding ApiLockedObject , which is finally delegated to the component for rendering. The fetches the locked item object ID if the prop itemId is not supplied by using dApp Kit useSuiClientQuery() hook to call the getDynamicFieldObject RPC endpoint. Recalling that in this smart contract, the locked item is put into a dynamic object field.

LockedObject.tsx

The (src/components/locked/partials/Locked.tsx) component is mainly responsible for rendering the ApiLockedObject . Later on, it will also consist of several on-chain interactions: unlock the locked objects and create an escrow out of the locked object.

Locked.tsx

Update src/routes/LockedDashboard.tsx to include the OwnedLockedList component:

LockedDashboard.tsx

Run your app and ensure you can:

To unlock the object, execute the unlock Move function identified by {PACKAGE_ID}::lock::unlock . Call the unlock function supplying the Locked object, its corresponding Key , the struct type of the original object, and transfer the unlocked object to the current connected wallet account. Also, implement the onSuccess callback to invalidate the cache data at query key locked after one second to force react-query to re-fetch the data at corresponding query key automatically.

Unlocking owned objects is another crucial and complex on-chain action in this application. Hence, it's beneficial to extract its logic into separated mutating functions to enhance discoverability and encapsulation.

src/mutations/locked.ts

Update src/components/locked/partials/Locked.tsx to include the useUnlockObjectMutation hook:

Locked.tsx

Run your app and ensure you can:

Update src/routes/EscrowDashboard.tsx to include the LockedList component:

EscrowDashboard.tsx

Add src/components/locked/ApiLockedList.tsx :

ApiLockedList.tsx

This hook fetches all the non-deleted system Locked objects from the API in a paginated fashion. Then, it proceeds into fetching the on-chain state, to ensure the latest state of the object is displayed.

This component uses tanstack's useInfiniteQuery instead of useSuiClientInfiniteQuery since the data is being fetched from the example's API rather than Sui.

Add src/hooks/useGetLockedObject.ts

Run your app and ensure you can:

To create escrows, include a mutating function through the useCreateEscrowMutation hook in src/mutations/escrow.ts . It accepts the escrowed item to be traded and the ApiLockedObject from another party as parameters. Then, call the {PACKAGE_ID}::shared::create Move function and provide the escrowed item, the key id of the locked object to exchange, and the recipient of the escrow (locked object's owner).

escrow.ts

Update src/components/locked/partials/Locked.tsx to include the useCreateEscrowMutation hook

Add src/components/escrows/CreateEscrow.tsx

Run your app and ensure you can:

The object should disappear from the list of locked objects in the Browse Locked Objects tab in the Escrows page. You view and accept or cancel escrows in later steps.

To cancel the escrow, create a mutation through the useCancelEscrowMutation hook in src/mutations/escrow.ts . The cancel function accepts the escrow ApiEscrowObject and its on-chain data. The {PACKAGE_ID}::shared::return_to_sender Move call is generic, thus it requires the type parameters of the escrowed object. Next, execute {PACKAGE_ID}::shared::return_to_sender and transfer the returned escrowed object to the creator of the escrow.

escrow.ts

Add src/components/escrows/Escrow.tsx

Add src/components/escrows/EscrowList.tsx

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

To accept the escrow, create a mutation through the useAcceptEscrowMutation hook in src/mutations/escrow.ts . The implementation should be fairly familiar to you now. The accept function accepts the escrow ApiEscrowObject and the locked object ApiLockedObject . The {PACKAGE_ID}::shared::swap Move call is generic, thus it requires the type parameters of the escrowed and locked objects. Query the objects details by using multiGetObjects on Sui client instance. Lastly, execute the {PACKAGE_ID}::shared::swap Move call and transfer the returned escrowed item to the connected wallet account. When the mutation succeeds, invalidate the cache to allow automatic re-fetch of the data.

escrow.ts

Update src/components/escrows/Escrow.tsx to include the useAcceptEscrowMutation hook

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

At this point, you have a fully functional frontend that allows users to discover trades and interact with listed escrows. The UI is designed to be user-friendly and intuitive, allowing users to easily navigate and interact with the application. Have fun exploring the app and testing out the different features!

What the guide teaches

The guide also shows how to build an app that:

Before getting started, make sure you have:

Installed the latest version of Sui .

Configured a valid network environment , as the guide has you deploy the module on Testnet.

Acquired Devnet or Testnet tokens for development purposes.

https://faucet.sui.io/ : Visit the online faucet to request SUI tokens. You can refresh your browser to perform multiple requests, but the requests are rate-limited per IP address.

Read the basics of shared versus owned objects .

To begin, create a new folder on your system titled trading that holds all your files. Inside that folder, create three more folders: api , contracts , and frontend . It's important to keep this directory structure as some helper scripts in this example target these directories by name. Different projects have their own directory structure, but it's common to split code into functional groups to help with maintenance.

In this part of the guide, you write the Move contracts that perform the trustless swaps. The guide describes how to create the package from scratch, but you can use a fork or copy of the example code in the Sui repo to follow along instead. See Write a Move Package to learn more about package structure and how to use the Sui CLI to scaffold a new project.

To begin writing your smart contracts, create an escrow folder in your contracts folder (if using recommended directory names). Create a file inside the folder named Move.toml and copy the following code into it. This is the package manifest file. If you want to learn more about the structure of the file, see Package Manifest in The Move Book.

If you are targeting a network other than Testnet, be sure to update the rev value for the Sui dependency.

With your manifest file in place, it's time to start creating the Move assets for this project. In your escrow folder, at the same level as your Move.toml file, create a sources folder. This is the common file structure of a package in Move. Create a new file inside sources titled lock.move . This file contains the logic that locks the object involved in a trade. The complete source code for this file follows and the sections that come after detail its components.

Click the titles at the top of codeblocks to open the relevant source file in GitHub.

lock.move

After a trade is initiated, you don't want the trading party to modify the object they agreed to trade. Imagine you're trading in-game items and you agree to trade a weapon with all its attachments, and its owner strips all its attachments just before the trade.

In a traditional trade, a third party typically holds the items in escrow to make sure they are not tampered with before the trade completes. This requires either trusting that the third party won't tamper with it themselves, paying the third party to ensure that doesn't happen, or both.

In a trustless swap, however, you can use the safety properties of Move to force an item's owner to prove that they have not tampered with the version of the object that you agreed to trade, without involving anyone else.

This is done by requiring that an object that is available for trading is locked with a single-use key , and asking the owner to supply the key when finalizing the trade.

To tamper with the object would require unlocking it, which consumes the key. Consequently, there would no longer be a key to finish the trade.

The lock and key are made single-use by the signatures of the lock and unlock functions. lock accepts any object of type T: store (the store ability is necessary for storing it inside a Locked), and creates both the Locked and its corresponding Key :

lock function in lock.move

The unlock function accepts the Locked and Key by value (which consumes them), and returns the underlying T as long as the correct key has been supplied for the lock:

unlock function in lock.move

Together, they ensure that a lock and key cannot have existed before the lock operation, and will not exist after a successful unlock – it is single use.

Move's type system guarantees that a given Key cannot be re-used (because unlock accepts it by value), but there are some properties that need to be confirmed with tests:

The test starts with a helper function for creating an object, it doesn't matter what kind of object it is, as long as it has the store ability. The test uses Coin , because it comes with a #[test_only] function for minting:

The first test works by creating an object to test, locking it and unlocking it – this should finish executing without aborting. The last two lines exist to keep the Move compiler happy by cleaning up the test coin and test scenario objects, because values in Move are not implicitly cleaned up unless they have the drop ability.

The other test is testing a failure scenario – that an abort happens. It creates two locked objects (this time the values are just u64 s), and use the key from one to try and unlock the other, which should fail (specified using the expected_failure attribute).

Unlike the previous test, the same clean up is not needed, because the code is expected to terminate. Instead, add another abort after the code that you expect to abort (making sure to use a different code for this second abort).

At this point, you have

From your escrow folder, run sui move test in your terminal or console. If successful, you get a response similar to the following that confirms the package builds and your tests pass:

You might notice that the Move compiler creates a build subfolder inside escrow upon a successful build. This folder contains your package's compiled bytecode, code from your package's dependencies, and various other files necessary for the build. At this point, it's enough to just be aware of these files. You don't need to fully understand the contents in build .

Create a new file in your escrow folder titled shared.move . The code in this file creates the shared Escrow object and completes the trading logic. The complete source code for this file follows and the sections that come after detail its components.

shared.move

Trading proceeds in three steps:

You can start by implementing steps two and three, by defining a new type to hold the escrowed object. It holds the escrowed object and an id: UID (because it's an object in its own right), but it also records the sender and intended recipient (to confirm they match when the trade happens), and it registers interest in the first party's object by recording the ID of the key that unlocks the Locked that contains the object.

You also need to create a function for creating the Escrow object. The object is shared because it needs to be accessed by the address that created it (in case the object needs to be returned) and by the intended recipient (to complete the swap).

create function in shared.move

If the second party stops responding, the first party can unlock their object. You need to create a function so the second party can recover their object in the symmetric case as well.

return_to_sender function in shared.move

Finally, you need to add a function to allow the first party to complete the trade.

swap function in shared.move

Tests for the escrow module are more involved than for lock – as they take advantage of test_scenario 's ability to simulate multiple transactions from different senders, and interact with shared objects.

The guide focuses on the test for a successful swap, but you can find a link to all the tests later on.

As with the lock test, start by creating a function to mint a test coin. You also create some constants to represent our transaction senders, ALICE , BOB , and DIANE .

The test body starts with a call to test_scenario::begin and ends with a call to test_scenario::end . It doesn't matter which address you pass to begin , because you pick one of ALICE or BOB at the start of each new transaction you write, so set it to @0x0 :

The first transaction is from BOB who creates a coin and locks it. You must remember the ID of the coin and the ID of the key, which you will need later, and then you transfer the locked object and the key itself to BOB , because this is what would happen in a real transaction: When simulating transactions in a test, you should only keep around primitive values, not whole objects, which would need to be written to chain between transactions.

Write these transactions inside the test_successful_swap function, between the call to begin and end .

Next, ALICE comes along and sets up the Escrow , which locks their coin. They register their interest for BOB' s coin by referencing BOB 's key's ID (ik2):

Finally, BOB completes the trade by calling swap . The take_shared function is used to simulate accepting a shared input. It uses type inference to know that the object must be an Escrow , and finds the last object of this type that was shared (by ALICE in the previous transaction). Similarly, use take_from_sender to simulate accepting owned inputs (in this case, BOB 's lock and key). The coin returned by swap is transferred back to BOB , as if it was called as part of a PTB, followed by a transfer command.

The rest of the test is designed to check that ALICE has BOB 's coin and vice versa. It starts by calling next_tx to make sure the effects of the previous transaction have been committed, before running the necessary checks.

The escrow Move package is now functional: You could publish it on chain and perform trustless swaps by creating transactions. Creating those transactions requires knowing the IDs of Locked , Key , and Escrow objects.

Locked and Key objects are typically owned by the transaction sender, and so can be queried through the Sui RPC, but Escrow objects are shared, and it is useful to be able to query them by their sender and recipient (so that users can see the trades they have offered and received).

Querying Escrow objects by their sender or recipient requires custom indexing, and to make it easy for the indexer to spot relevant transactions, add the following events to escrow.move :

Functions responsible for various aspects of the escrow's lifecycle emit these events. The custom indexer can then subscribe to transactions that emit these events and process only those, rather than the entire chain state:

emit events included in functions from shared.move

You now have shared.move and locked.move files in your sources folder. From the parent escrow folder, run sui move test in your terminal or console. If successful, you get a response similar to the following that confirms the package builds and your tests pass:

Well done. You have written the Move package!

To turn this into a complete dApp, you need to create a frontend. However, for the frontend to be updated, it has to listen to the blockchain as escrows are made and swaps are fulfilled.

To achieve this, in the next step you create an indexing service.

With the contract adapted to emit events, you can now write an indexer that keeps track of all active Escrow objects and exposes an API for querying objects by sender or recipient.

The indexer is backed by a Prisma DB with the following schema:

schema.prisma

The core of the indexer is an event loop, initialized in a function called setupListeners .

The indexer queries events related to the escrow module, using a queryEvent filter, and keeps track of a cursor representing the latest event it has processed so it can resume indexing from the right place even if it is restarted. The filter is looking for any events whose type is from the escrow module of the Move package (see the event-indexer.ts code that follows).

The core event job works by polling: It queries RPC for events following its latest cursor and sends them to a callback for processing. If it detects more than one page of new events, it immediately requests the next page. Otherwise, the job waits for the next polling interval before checking again.

event-indexer.ts

The callback is responsible for reading the event and updating the database accordingly. For demo purposes, SQLite is being used, and so you need to issue a separate UPSERT to the database for each escrowed object. In a production setting, however, you would want to batch requests to the database to optimize data flow.

escrow-handler.ts

The data that the indexer captures can then be served over an API, so that a frontend can read it. Follow the next section to implement the API in TypeScript, to run on Node, using Express.

You want your API to accept the query string in the URL as the parameters for database WHERE query. Hence, you want a utility that can extract and parse the URL query string into valid query parameters for Prisma. With the parseWhereStatement() function, the callers filter the set of keys from the URL query string and transforms those corresponding key-value pairs into the correct format for Prisma.

parseWhereStatement in api-queries.ts

Pagination is another crucial part to ensure your API returns sufficient and/or ordered chunk of information instead of all the data that might be the vector for a DDOS attack. Similar to WHERE parameters , define a set of keys in the URL query string to be accepted as valid pagination parameters. The parsePaginationForQuery() utility function helps to achieve this by filtering the pre-determined keys sort , limit , cursor and parsing corresponding key-value pairs into ApiPagination that Prisma can consume.

In this example, the id field of the model in the database as the cursor that allows clients to continue subsequent queries with the next page.

parsePaginationForQuery in api-queries.ts

All the endpoints are defined in server.ts , particularly, there are two endpoints:

You define a list of valid query keys, such as deleted , creator , keyId , and objectId for Locked data and cancelled , swapped , recipient , and sender for Escrow data. Pass the URL query string into the pre-defined utilities to output the correct parameters that Prisma can use.

server.ts

Now that you have an indexer and an API service, you can deploy your move package and start the indexer and API service.

Install dependencies by running pnpm install --ignore-workspace or yarn install --ignore-workspace .

Setup the database by running pnpm db:setup:dev or yarn db:setup:dev .

Deploy the Sui package

Deployment instructions

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.

Use the following command to list your available environments:

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, run the following command:

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.

There are some helper functions to publish the smart contracts so you can create some demo data (for Testnet). The helper function to publish the smart contrqcts expects built smart contracts in both the escrow and demo directories. Run sui move build in both directories, if necessary. Be sure to update the Sui dependency in the manifest to point to the correct source based on your environment.

To publish the smart contracts and produce demo data:

Publish the smart contracts by running the following command from your api folder:

If successful, demo-contract.json and escrow-contract.json are created in the backend root directory. These files contain the contract addresses and are used by the backend and frontend to interact with the contracts.

Produce demo non-locked and locked objects

Produce demo escrows

If you want to reset the database (for a clean demo, for example), run pnpm db:reset:dev && pnpm db:setup:dev or yarn db:reset:dev && yarn db:setup:dev .

Run both the API and the indexer by running pnpm dev or yarn dev .

Visit http://localhost:3000/escrows or http://localhost:3000/locked

You should now have an indexer running.

With the code successfully deployed on Testnet, you can now create a frontend to display the trading data and to allow users to interact with the Move modules.

In this final part of the app example, you build a frontend (UI) that allows end users to discover trades and interact with listed escrows.

You can view the complete source code for this app example in the Sui repository.

Before getting started, make sure you have:

The UI design consists of three parts:

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

When asked for a name for your dApp, provide one of your liking. The dApp scaffold gets created in a new directory with the name you provide. This is convenient to keep your working code separate from the example source code that might already populate this folder. The codeblocks that follow point to the code in the default example location. Be aware the path to your own code includes the dApp name you provide.

First, set up import aliases to make the code more readable and maintainable. This allows you to import files using @/ instead of relative paths.

Replace the content of tsconfig.json with the following:

The paths option under compilerOptions is what defines the aliasing for TypeScript. Here, the alias @/ *is mapped to the ./src/* directory, meaning that any time you use @/ , TypeScript resolves it as a reference to the src folder. This setup reduces the need for lengthy relative paths when importing files in your project.

Replace the content of vite.config.ts with the following:

Vite also needs to be aware of the aliasing to resolve imports correctly during the build process. In the resolve.alias configuration of vite.config.ts , we map the alias @ to the /src directory.

To streamline the styling process and keep the codebase clean and maintainable, this guide uses Tailwind CSS, which provides utility-first CSS classes to rapidly build custom designs. Run the following command from the base of your dApp project to add

Tailwind CSS and its dependencies:

Next, generate the Tailwind CSS configuration file by running the following:

Replace the content of tailwind.config.js with the following:

Add the src/styles/ directory and add base.css :

First, deploy your package via the scripts in the api directory .

Then, create a src/constants.ts file and fill it with the following:

If you create a dApp using a project name so that your src files are in a subfolder of frontend , be sure to add another nesting level (../) to the import statements.

Create a src/utils/ directory and add the following file:

Create a src/components/ directory and add the following components:

ExplorerLink.tsx

InfiniteScrollArea.tsx

Loading.tsx

SuiObjectDisplay.tsx

Install the necessary dependencies:

The imported template only has a single page. To add more pages, you need to set up routing.

First, install the necessary dependencies:

Then, create a src/routes/ directory and add index.tsx . This file contains the routing configuration:

Add the following respective files to the src/routes/ directory:

root.tsx . This file contains the root component that is rendered on every page:

LockedDashboard.tsx . This file contains the component for the Manage Objects page.

EscrowDashboard.tsx . This file contains the component for the Escrows page.

Update src/main.tsx by replacing the App component with the RouterProvider and replace "dark" with "light" in the Theme component:

Note that dApp Kit provides a set of hooks for making query and mutation calls to the Sui blockchain. These hooks are thin wrappers around query and mutation hooks from @tanstack/react-query .

Create src/components/Header.tsx . This file contains the navigation links and the connect wallet button:

The dApp Kit comes with a pre-built React.js component called ConnectButton displaying a button to connect and disconnect a wallet. The connecting and disconnecting wallet logic is handled seamlessly so you don't need to worry about repeating yourself doing the same logic all over again.

At this point, you have a basic routing setup. Run your app and ensure you can:

Note, the styles should be applied. The Header component should look like this:

All the type definitions are in src/types/types.ts . Create this file and add the following:

ApiLockedObject and ApiEscrowObject represent the Locked and Escrow indexed data model the indexing and API service return.

EscrowListingQuery and LockedListingQuery are the query parameters model to provide to the API service to fetch from the endpoints /escrow and /locked accordingly.

Now, display the objects owned by the connected wallet address. This is the Manage Objects page.

First add this file src/components/locked/LockOwnedObjects.tsx :

Fetch the owned objects directly from Sui blockchain using the useSuiClientInfiniteQuery() hook from dApp Kit . This hook is a thin wrapper around Sui blockchain RPC calls, reference the documentation to learn more about these RPC hooks . Basically, supply the RPC endpoint you want to execute, in this case it's the getOwnedObjects endpoint . Supply the connected wallet account as the owner . The returned data is stored inside the cache at query key getOwnedObjects . In a future step you invalidate this cache after a mutation succeeds, so the data will be re-fetched automatically.

Next, update src/routes/LockedDashboard.tsx to include the LockOwnedObjects component:

Run your app and ensure you can:

If you don't see any objects, you might need to create some demo data or connect your wallet. You can mint objects after completing the next steps.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's create and examine the execute transaction hook.

Create src/hooks/useTransactionExecution.ts :

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useSuiClient() hook from dApp Kit to retrieve the Sui client instance configured in src/main.tsx . The useSignTransaction() function is another hook from dApp kit that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, use the executeTransaction() on the Sui client instance of the Sui TypeScript SDK.

The full source code of the demo bear smart contract is available at Trading Contracts Demo directory

You need a utility function to create a dummy object representing a real world asset so you can use it to test and demonstrate escrow users flow on the UI directly.

Create src/mutations/demo.ts :

As previously mentioned, this example uses @tanstack/react-query to query, cache, and mutate server state. Server state is data only available on remote servers, and the only way to retrieve or update this data is by interacting with these remote servers. In this case, it could be from an API or directly from Sui blockchain RPC.

When you execute a transaction call to mutate data on the Sui blockchain, use the useMutation() hook. The useMutation() hook accepts several inputs, however, you only need two of them for this example. The first parameter, mutationFn , accepts the function to execute the main mutating logic, while the second parameter, onSuccess , is a callback that runs when the mutating logic succeeds.

The main mutating logic includes executing a Move call of a package named demo_bear::new to create a dummy bear object and transferring it to the connected wallet account, all within the same Transaction . This example reuses the executeTransaction() hook from the Execute Transaction Hook step to execute the transaction.

Another benefit of wrapping the main mutating logic inside useMutation() is that you can access and manipulate the cache storing server state. This example fetches the cache from remote servers by using query call in an appropriate callback. In this case, it is the onSuccess callback. When the transaction succeeds, invalidate the cache data at the cache key called getOwnedObjects , then @tanstack/react-query handles the re-fetching mechanism for the invalidated data automatically. Do this by using invalidateQueries() on the @tanstack/react-query configured client instance retrieved by useQueryClient() hook in the Set up Routing step.

Now the logic to create a dummy bear object exists. You just need to attach it into the button in the header.

Header.tsx

Run your app and ensure you can:

To lock the object, execute the lock Move function identified by {PACKAGE_ID}::lock::lock . The implementation is similar to what's in previous mutation functions, use useMutation() from @tanstack/react-query to wrap the main logic inside it. The lock function requires an object to be locked and its type because our smart contract lock function is generic and requires type parameters. After creating a Locked object and its Key object, transfer them to the connected wallet account within the same transaction block.

It's beneficial to extract logic of locking owned objects into a separated mutating function to enhance discoverability and encapsulation.

Create src/mutations/locked.ts :

Update src/components/locked/LockOwnedObjects.tsx to include the useLockObjectMutation hook:

LockOwnedObjects.tsx

Run your app and ensure you can:

The object should disappear from the list of owned objects. You view and unlock locked objects in later steps.

Let's take a look at the My Locked Objects tab by examining src/components/locked/OwnedLockedList.tsx . Focus on the logic on how to retrieve this list.

OwnedLockedList.tsx

This instance of useSuiClientInfiniteQuery() is similar to the one in the LockOwnedObjects component. The difference is that it fetches the locked objects instead of the owned objects. The Locked object is a struct type in the smart contract, so you need to supply the struct type to the query call as a filter . The struct type is usually identified by the format of {PACKAGE_ID}::{MODULE_NAME}::{STRUCT_TYPE} .

The (src/components/locked/LockedObject.tsx) component is mainly responsible for mapping an on-chain SuiObjectData Locked object to its corresponding ApiLockedObject , which is finally delegated to the component for rendering. The fetches the locked item object ID if the prop itemId is not supplied by using dApp Kit useSuiClientQuery() hook to call the getDynamicFieldObject RPC endpoint. Recalling that in this smart contract, the locked item is put into a dynamic object field.

LockedObject.tsx

The (src/components/locked/partials/Locked.tsx) component is mainly responsible for rendering the ApiLockedObject . Later on, it will also consist of several on-chain interactions: unlock the locked objects and create an escrow out of the locked object.

Locked.tsx

Update src/routes/LockedDashboard.tsx to include the OwnedLockedList component:

LockedDashboard.tsx

Run your app and ensure you can:

To unlock the object, execute the unlock Move function identified by {PACKAGE_ID}::lock::unlock . Call the unlock function supplying the Locked object, its corresponding Key , the struct type of the original object, and transfer the unlocked object to the current connected wallet account. Also, implement the onSuccess callback to invalidate the cache data at query key locked after one second to force react-query to re-fetch the data at corresponding query key automatically.

Unlocking owned objects is another crucial and complex on-chain action in this application. Hence, it's beneficial to extract its logic into separated mutating functions to enhance discoverability and encapsulation.

src/mutations/locked.ts

Update src/components/locked/partials/Locked.tsx to include the useUnlockObjectMutation hook:

Locked.tsx

Run your app and ensure you can:

Update src/routes/EscrowDashboard.tsx to include the LockedList component:

EscrowDashboard.tsx

Add src/components/locked/ApiLockedList.tsx :

ApiLockedList.tsx

This hook fetches all the non-deleted system Locked objects from the API in a paginated fashion. Then, it proceeds into fetching the on-chain state, to ensure the latest state of the object is displayed.

This component uses tanstack's useInfiniteQuery instead of useSuiClientInfiniteQuery since the data is being fetched from the example's API rather than Sui.

Add src/hooks/useGetLockedObject.ts

Run your app and ensure you can:

To create escrows, include a mutating function through the useCreateEscrowMutation hook in src/mutations/escrow.ts . It accepts the escrowed item to be traded and the ApiLockedObject from another party as parameters. Then, call the {PACKAGE_ID}::shared::create Move function and provide the escrowed item, the key id of the locked object to exchange, and the recipient of the escrow (locked object's owner).

escrow.ts

Update src/components/locked/partials/Locked.tsx to include the useCreateEscrowMutation hook

Add src/components/escrows/CreateEscrow.tsx

Run your app and ensure you can:

The object should disappear from the list of locked objects in the Browse Locked Objects tab in the Escrows page. You view and accept or cancel escrows in later steps.

To cancel the escrow, create a mutation through the useCancelEscrowMutation hook in src/mutations/escrow.ts . The cancel function accepts the escrow ApiEscrowObject and its on-chain data. The {PACKAGE_ID}::shared::return_to_sender Move call is generic, thus it requires the type parameters of the escrowed object. Next, execute {PACKAGE_ID}::shared::return_to_sender and transfer the returned escrowed object to the creator of the escrow.

escrow.ts

Add src/components/escrows/Escrow.tsx

Add src/components/escrows/EscrowList.tsx

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

To accept the escrow, create a mutation through the useAcceptEscrowMutation hook in src/mutations/escrow.ts . The implementation should be fairly familiar to you now. The accept function accepts the escrow ApiEscrowObject and the locked object ApiLockedObject . The {PACKAGE_ID}::shared::swap Move call is generic, thus it requires the type parameters of the escrowed and locked objects. Query the objects details by using multiGetObjects on Sui client instance. Lastly, execute the {PACKAGE_ID}::shared::swap Move call and transfer the returned escrowed item to the connected wallet account. When the mutation succeeds, invalidate the cache to allow automatic re-fetch of the data.

escrow.ts

Update src/components/escrows/Escrow.tsx to include the useAcceptEscrowMutation hook

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

At this point, you have a fully functional frontend that allows users to discover trades and interact with listed escrows. The UI is designed to be user-friendly and intuitive, allowing users to easily navigate and interact with the application. Have fun exploring the app and testing out the different features!

What you need

Before getting started, make sure you have:

Installed the latest version of Sui .

Configured a valid network environment , as the guide has you deploy the module on Testnet.

Acquired Devnet or Testnet tokens for development purposes.

https://faucet.sui.io/ : Visit the online faucet to request SUI tokens. You can refresh your browser to perform multiple requests, but the requests are rate-limited per IP address.

Read the basics of shared versus owned objects .

To begin, create a new folder on your system titled trading that holds all your files. Inside that folder, create three more folders: api , contracts , and frontend . It's important to keep this directory structure as some helper scripts in this example target these directories by name. Different projects have their own directory structure, but it's common to split code into functional groups to help with maintenance.

In this part of the guide, you write the Move contracts that perform the trustless swaps. The guide describes how to create the package from scratch, but you can use a fork or copy of the example code in the Sui repo to follow along instead. See Write a Move Package to learn more about package structure and how to use the Sui CLI to scaffold a new project.

To begin writing your smart contracts, create an escrow folder in your contracts folder (if using recommended directory names). Create a file inside the folder named Move.toml and copy the following code into it. This is the package manifest file. If you want to learn more about the structure of the file, see Package Manifest in The Move Book.

If you are targeting a network other than Testnet, be sure to update the rev value for the Sui dependency.

With your manifest file in place, it's time to start creating the Move assets for this project. In your escrow folder, at the same level as your Move.toml file, create a sources folder. This is the common file structure of a package in Move. Create a new file inside sources titled lock.move . This file contains the logic that locks the object involved in a trade. The complete source code for this file follows and the sections that come after detail its components.

Click the titles at the top of codeblocks to open the relevant source file in GitHub.

lock.move

After a trade is initiated, you don't want the trading party to modify the object they agreed to trade. Imagine you're trading in-game items and you agree to trade a weapon with all its attachments, and its owner strips all its attachments just before the trade.

In a traditional trade, a third party typically holds the items in escrow to make sure they are not tampered with before the trade completes. This requires either trusting that the third party won't tamper with it themselves, paying the third party to ensure that doesn't happen, or both.

In a trustless swap, however, you can use the safety properties of Move to force an item's owner to prove that they have not tampered with the version of the object that you agreed to trade, without involving anyone else.

This is done by requiring that an object that is available for trading is locked with a single-use key , and asking the owner to supply the key when finalizing the trade.

To tamper with the object would require unlocking it, which consumes the key. Consequently, there would no longer be a key to finish the trade.

The lock and key are made single-use by the signatures of the lock and unlock functions. lock accepts any object of type T: store (the store ability is necessary for storing it inside a Locked), and creates both the Locked and its corresponding Key :

lock function in lock.move

The unlock function accepts the Locked and Key by value (which consumes them), and returns the underlying T as long as the correct key has been supplied for the lock:

unlock function in lock.move

Together, they ensure that a lock and key cannot have existed before the lock operation, and will not exist after a successful unlock – it is single use.

Move's type system guarantees that a given Key cannot be re-used (because unlock accepts it by value), but there are some properties that need to be confirmed with tests:

The test starts with a helper function for creating an object, it doesn't matter what kind of object it is, as long as it has the store ability. The test uses Coin , because it comes with a #[test_only] function for minting:

The first test works by creating an object to test, locking it and unlocking it – this should finish executing without aborting. The last two lines exist to keep the Move compiler happy by cleaning up the test coin and test scenario objects, because values in Move are not implicitly cleaned up unless they have the drop ability.

The other test is testing a failure scenario – that an abort happens. It creates two locked objects (this time the values are just u64 s), and use the key from one to try and unlock the other, which should fail (specified using the expected_failure attribute).

Unlike the previous test, the same clean up is not needed, because the code is expected to terminate. Instead, add another abort after the code that you expect to abort (making sure to use a different code for this second abort).

At this point, you have

From your escrow folder, run sui move test in your terminal or console. If successful, you get a response similar to the following that confirms the package builds and your tests pass:

You might notice that the Move compiler creates a build subfolder inside escrow upon a successful build. This folder contains your package's compiled bytecode, code from your package's dependencies, and various other files necessary for the build. At this point, it's enough to just be aware of these files. You don't need to fully understand the contents in build .

Create a new file in your escrow folder titled shared.move . The code in this file creates the shared Escrow object and completes the trading logic. The complete source code for this file follows and the sections that come after detail its components.

shared.move

Trading proceeds in three steps:

You can start by implementing steps two and three, by defining a new type to hold the escrowed object. It holds the escrowed object and an id: UID (because it's an object in its own right), but it also records the sender and intended recipient (to confirm they match when the trade happens), and it registers interest in the first party's object by recording the ID of the key that unlocks the Locked that contains the object.

You also need to create a function for creating the Escrow object. The object is shared because it needs to be accessed by the address that created it (in case the object needs to be returned) and by the intended recipient (to complete the swap).

create function in shared.move

If the second party stops responding, the first party can unlock their object. You need to create a function so the second party can recover their object in the symmetric case as well.

return_to_sender function in shared.move

Finally, you need to add a function to allow the first party to complete the trade.

swap function in shared.move

Tests for the escrow module are more involved than for lock – as they take advantage of test_scenario 's ability to simulate multiple transactions from different senders, and interact with shared objects.

The guide focuses on the test for a successful swap, but you can find a link to all the tests later on.

As with the lock test, start by creating a function to mint a test coin. You also create some constants to represent our transaction senders, ALICE , BOB , and DIANE .

The test body starts with a call to test_scenario::begin and ends with a call to test_scenario::end . It doesn't matter which address you pass to begin , because you pick one of ALICE or BOB at the start of each new transaction you write, so set it to @0x0 :

The first transaction is from BOB who creates a coin and locks it. You must remember the ID of the coin and the ID of the key, which you will need later, and then you transfer the locked object and the key itself to BOB , because this is what would happen in a real transaction: When simulating transactions in a test, you should only keep around primitive values, not whole objects, which would need to be written to chain between transactions.

Write these transactions inside the test_successful_swap function, between the call to begin and end .

Next, ALICE comes along and sets up the Escrow , which locks their coin. They register their interest for BOB' s coin by referencing BOB 's key's ID (ik2):

Finally, BOB completes the trade by calling swap . The take_shared function is used to simulate accepting a shared input. It uses type inference to know that the object must be an Escrow , and finds the last object of this type that was shared (by ALICE in the previous transaction). Similarly, use take_from_sender to simulate accepting owned inputs (in this case, BOB 's lock and key). The coin returned by swap is transferred back to BOB , as if it was called as part of a PTB, followed by a transfer command.

The rest of the test is designed to check that ALICE has BOB 's coin and vice versa. It starts by calling next_tx to make sure the effects of the previous transaction have been committed, before running the necessary checks.

The escrow Move package is now functional: You could publish it on chain and perform trustless swaps by creating transactions. Creating those transactions requires knowing the IDs of Locked , Key , and Escrow objects.

Locked and Key objects are typically owned by the transaction sender, and so can be queried through the Sui RPC, but Escrow objects are shared, and it is useful to be able to query them by their sender and recipient (so that users can see the trades they have offered and received).

Querying Escrow objects by their sender or recipient requires custom indexing, and to make it easy for the indexer to spot relevant transactions, add the following events to escrow.move :

Functions responsible for various aspects of the escrow's lifecycle emit these events. The custom indexer can then subscribe to transactions that emit these events and process only those, rather than the entire chain state:

emit events included in functions from shared.move

You now have shared.move and locked.move files in your sources folder. From the parent escrow folder, run sui move test in your terminal or console. If successful, you get a response similar to the following that confirms the package builds and your tests pass:

Well done. You have written the Move package! □

To turn this into a complete dApp, you need to create a frontend. However, for the frontend to be updated, it has to listen to the blockchain as escrows are made and swaps are fulfilled.

To achieve this, in the next step you create an indexing service.

With the contract adapted to emit events, you can now write an indexer that keeps track of all active Escrow objects and exposes an API for querying objects by sender or recipient.

The indexer is backed by a Prisma DB with the following schema:

schema.prisma

The core of the indexer is an event loop, initialized in a function called setupListeners .

The indexer queries events related to the escrow module, using a queryEvent filter, and keeps track of a cursor representing the latest event it has processed so it can resume indexing from the right place even if it is restarted. The filter is looking for any events whose type is from the escrow module of the Move package (see the event-indexer.ts code that follows).

The core event job works by polling: It queries RPC for events following its latest cursor and sends them to a callback for processing. If it detects more than one page of new events, it immediately requests the next page. Otherwise, the job waits for the next polling interval before checking again.

event-indexer.ts

The callback is responsible for reading the event and updating the database accordingly. For demo purposes, SQLite is being used, and so you need to issue a separate UPSERT to the database for each escrowed object. In a production setting, however, you would want to batch requests to the database to optimize data flow.

escrow-handler.ts

The data that the indexer captures can then be served over an API, so that a frontend can read it. Follow the next section to implement the API in TypeScript, to run on Node, using Express.

You want your API to accept the query string in the URL as the parameters for database WHERE query. Hence, you want a utility that can extract and parse the URL query string into valid query parameters for Prisma. With the parseWhereStatement() function, the callers filter the set of keys from the URL query string and transforms those corresponding key-value pairs into the correct format for Prisma.

parseWhereStatement in api-queries.ts

Pagination is another crucial part to ensure your API returns sufficient and/or ordered chunk of information instead of all the data that might be the vector for a DDOS attack. Similar to WHERE parameters , define a set of keys in the URL query string to be accepted as valid pagination parameters. The parsePaginationForQuery() utility function helps to achieve this by filtering the pre-determined keys sort , limit , cursor and parsing corresponding key-value pairs into ApiPagination that Prisma can consume.

In this example, the id field of the model in the database as the cursor that allows clients to continue subsequent queries with the next page.

parsePaginationForQuery in api-queries.ts

All the endpoints are defined in server.ts , particularly, there are two endpoints:

You define a list of valid query keys, such as deleted , creator , keyId , and objectId for Locked data and cancelled , swapped , recipient , and sender for Escrow data. Pass the URL query string into the pre-defined utilities to output the correct parameters that Prisma can use.

server.ts

Now that you have an indexer and an API service, you can deploy your move package and start the indexer and API service.

Install dependencies by running pnpm install --ignore-workspace or yarn install --ignore-workspace .

Setup the database by running pnpm db:setup:dev or yarn db:setup:dev .

Deploy the Sui package

Deployment instructions

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.

Use the following command to list your available environments:

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, run the following command:

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.

There are some helper functions to publish the smart contracts so you can create some demo data (for Testnet). The helper function to publish the smart contrqcts expects built smart contracts in both the escrow and demo directories. Run sui move build in both directories, if necessary. Be sure to update the Sui dependency in the manifest to point to the correct source based on your environment.

To publish the smart contracts and produce demo data:

Publish the smart contracts by running the following command from your api folder:

If successful, demo-contract.json and escrow-contract.json are created in the backend root directory. These files contain the contract addresses and are used by the backend and frontend to interact with the contracts.

Produce demo non-locked and locked objects

Produce demo escrows

If you want to reset the database (for a clean demo, for example), run pnpm db:reset:dev && pnpm db:setup:dev or yarn db:reset:dev && yarn db:setup:dev .

Run both the API and the indexer by running pnpm dev or yarn dev .

Visit http://localhost:3000/escrows or http://localhost:3000/locked

You should now have an indexer running.

With the code successfully deployed on Testnet, you can now create a frontend to display the trading data and to allow users to interact with the Move modules.

In this final part of the app example, you build a frontend (UI) that allows end users to discover trades and interact with listed escrows.

You can view the complete source code for this app example in the Sui repository.

Before getting started, make sure you have:

The UI design consists of three parts:

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

When asked for a name for your dApp, provide one of your liking. The dApp scaffold gets created in a new directory with the name you provide. This is convenient to keep your working code separate from the example source code that might already populate this folder. The codeblocks that follow point to the code in the default example location. Be aware the path to your own code includes the dApp name you provide.

First, set up import aliases to make the code more readable and maintainable. This allows you to import files using @/ instead of relative paths.

Replace the content of tsconfig.json with the following:

The paths option under compilerOptions is what defines the aliasing for TypeScript. Here, the alias *@/ is mapped to the ./src/* directory, meaning that any time you use @/ , TypeScript resolves it as a reference to the src folder. This setup reduces the need for lengthy relative paths when importing files in your project.

Replace the content of vite.config.ts with the following:

Vite also needs to be aware of the aliasing to resolve imports correctly during the build process. In the resolve.alias configuration of vite.config.ts , we map the alias @ to the /src directory.

To streamline the styling process and keep the codebase clean and maintainable, this guide uses Tailwind CSS, which provides utility-first CSS classes to rapidly build custom designs. Run the following command from the base of your dApp project to add Tailwind CSS and its dependencies:

Next, generate the Tailwind CSS configuration file by running the following:

Replace the content of tailwind.config.js with the following:

Add the src/styles/ directory and add base.css :

First, deploy your package via the scripts in the api directory .

Then, create a src/constants.ts file and fill it with the following:

If you create a dApp using a project name so that your src files are in a subfolder of frontend , be sure to add another nesting level (

../) to the import statements.

Create a src/utils/ directory and add the following file:

Create a src/components/ directory and add the following components:

ExplorerLink.tsx

InfiniteScrollArea.tsx

Loading.tsx

SuiObjectDisplay.tsx

Install the necessary dependencies:

The imported template only has a single page. To add more pages, you need to set up routing.

First, install the necessary dependencies:

Then, create a src/routes/ directory and add index.tsx . This file contains the routing configuration:

Add the following respective files to the src/routes/ directory:

root.tsx . This file contains the root component that is rendered on every page:

LockedDashboard.tsx . This file contains the component for the Manage Objects page.

EscrowDashboard.tsx . This file contains the component for the Escrows page.

Update src/main.tsx by replacing the App component with the RouterProvider and replace "dark" with "light" in the Theme component:

Note that dApp Kit provides a set of hooks for making query and mutation calls to the Sui blockchain. These hooks are thin wrappers around query and mutation hooks from @tanstack/react-query .

Create src/components/Header.tsx . This file contains the navigation links and the connect wallet button:

The dApp Kit comes with a pre-built React.js component called ConnectButton displaying a button to connect and disconnect a wallet. The connecting and disconnecting wallet logic is handled seamlessly so you don't need to worry about repeating yourself doing the same logic all over again.

At this point, you have a basic routing setup. Run your app and ensure you can:

Note, the styles should be applied. The Header component should look like this:

All the type definitions are in src/types/types.ts . Create this file and add the following:

ApiLockedObject and ApiEscrowObject represent the Locked and Escrow indexed data model the indexing and API service return.

EscrowListingQuery and LockedListingQuery are the query parameters model to provide to the API service to fetch from the endpoints /escrow and /locked accordingly.

Now, display the objects owned by the connected wallet address. This is the Manage Objects page.

First add this file src/components/locked/LockOwnedObjects.tsx :

Fetch the owned objects directly from Sui blockchain using the useSuiClientInfiniteQuery() hook from dApp Kit . This hook is a thin wrapper around Sui blockchain RPC calls, reference the documentation to learn more about these RPC hooks . Basically, supply the RPC endpoint you want to execute, in this case it's the getOwnedObjects endpoint . Supply the connected wallet account as the owner . The returned data is stored inside the cache at query key getOwnedObjects . In a future step you invalidate this cache after a mutation succeeds, so the data will be re-fetched automatically.

Next, update src/routes/LockedDashboard.tsx to include the LockOwnedObjects component:

Run your app and ensure you can:

If you don't see any objects, you might need to create some demo data or connect your wallet. You can mint objects after completing the next steps.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's create and examine the execute transaction hook.

Create src/hooks/useTransactionExecution.ts :

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useSuiClient() hook from dApp Kit to retrieve the Sui client instance configured in src/main.tsx . The useSignTransaction() function is another hook from dApp kit that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, use the executeTransaction() on the Sui client instance of the Sui TypeScript SDK.

The full source code of the demo bear smart contract is available at Trading Contracts Demo directory

You need a utility function to create a dummy object representing a real world asset so you can use it to test and demonstrate escrow users flow on the UI directly.

Create src/mutations/demo.ts :

As previously mentioned, this example uses @tanstack/react-query to query, cache, and mutate server state. Server state is data only available on remote servers, and the only way to retrieve or update this data is by interacting with these remote servers. In this case, it could be from an API or directly from Sui blockchain RPC.

When you execute a transaction call to mutate data on the Sui blockchain, use the useMutation() hook. The useMutation() hook accepts several inputs, however, you only need two of them for this example. The first parameter, mutationFn , accepts the function to execute the main mutating logic, while the second parameter, onSuccess , is a callback that runs when the mutating logic succeeds.

The main mutating logic includes executing a Move call of a package named demo_bear::new to create a dummy bear object and transferring it to the connected wallet account, all within the same Transaction . This example reuses the executeTransaction() hook from the Execute Transaction Hook step to execute the transaction.

Another benefit of wrapping the main mutating logic inside useMutation() is that you can access and manipulate the cache storing server state. This example fetches the cache from remote servers by using query call in an appropriate callback. In this case, it is the onSuccess callback. When the transaction succeeds, invalidate the cache data at the cache key called getOwnedObjects , then @tanstack/react-query handles the re-fetching mechanism for the invalidated data automatically. Do this by using invalidateQueries() on the @tanstack/react-query configured client instance retrieved by useQueryClient() hook in the Set up Routing step.

Now the logic to create a dummy bear object exists. You just need to attach it into the button in the header.

Header.tsx

Run your app and ensure you can:

To lock the object, execute the lock Move function identified by {PACKAGE_ID}::lock::lock . The implementation is similar to what's in previous mutation functions, use useMutation() from @tanstack/react-query to wrap the main logic inside it. The lock function requires an object to be locked and its type because our smart contract lock function is generic and requires type parameters. After creating a Locked object and its Key object, transfer them to the connected wallet account within the same transaction block.

It's beneficial to extract logic of locking owned objects into a separated mutating function to enhance discoverability and encapsulation.

Create src/mutations/locked.ts :

Update src/components/locked/LockOwnedObjects.tsx to include the useLockObjectMutation hook:

LockOwnedObjects.tsx

Run your app and ensure you can:

The object should disappear from the list of owned objects. You view and unlock locked objects in later steps.

Let's take a look at the My Locked Objects tab by examining src/components/locked/OwnedLockedList.tsx . Focus on the logic on how to retrieve this list.

OwnedLockedList.tsx

This instance of useSuiClientInfiniteQuery() is similar to the one in the LockOwnedObjects component. The difference is that it fetches the locked objects instead of the owned objects. The Locked object is a struct type in the smart contract, so you need to supply the struct type to the query call as a filter . The struct type is usually identified by the format of {PACKAGE_ID}::{MODULE_NAME}::{STRUCT_TYPE} .

The (src/components/locked/LockedObject.tsx) component is mainly responsible for mapping an on-chain SuiObjectData Locked object to its corresponding ApiLockedObject , which is finally delegated to the component for rendering. The fetches the locked item object ID if the prop itemId is not supplied by using dApp Kit useSuiClientQuery() hook to call the getDynamicFieldObject RPC endpoint. Recalling that in this smart contract, the locked item is put into a dynamic object field.

LockedObject.tsx

The (src/components/locked/partials/Locked.tsx) component is mainly responsible for rendering the ApiLockedObject . Later on, it will also consist of several on-chain interactions: unlock the locked objects and create an escrow out of the locked object.

Locked.tsx

Update src/routes/LockedDashboard.tsx to include the OwnedLockedList component:

LockedDashboard.tsx

Run your app and ensure you can:

To unlock the object, execute the unlock Move function identified by {PACKAGE_ID}::lock::unlock . Call the unlock function supplying the Locked object, its corresponding Key , the struct type of the original object, and transfer the unlocked object to the current connected wallet account. Also, implement the onSuccess callback to invalidate the cache data at query key locked after one second to force react-query to re-fetch the data at corresponding query key automatically.

Unlocking owned objects is another crucial and complex on-chain action in this application. Hence, it's beneficial to extract its logic into separated mutating functions to enhance discoverability and encapsulation.

src/mutations/locked.ts

Update src/components/locked/partials/Locked.tsx to include the useUnlockObjectMutation hook:

Locked.tsx

Run your app and ensure you can:

Update src/routes/EscrowDashboard.tsx to include the LockedList component:

EscrowDashboard.tsx

Add src/components/locked/ApiLockedList.tsx :

ApiLockedList.tsx

This hook fetches all the non-deleted system Locked objects from the API in a paginated fashion. Then, it proceeds into fetching the on-chain state, to ensure the latest state of the object is displayed.

This component uses tanstack's useInfiniteQuery instead of useSuiClientInfiniteQuery since the data is being fetched from the example's API rather than Sui.

Add src/hooks/useGetLockedObject.ts

Run your app and ensure you can:

To create escrows, include a mutating function through the useCreateEscrowMutation hook in src/mutations/escrow.ts . It accepts the escrowed item to be traded and the ApiLockedObject from another party as parameters. Then, call the

{PACKAGE_ID}::shared::create Move function and provide the escrowed item, the key id of the locked object to exchange, and the recipient of the escrow (locked object's owner).

escrow.ts

Update src/components/locked/partials/Locked.tsx to include the useCreateEscrowMutation hook

Add src/components/escrows/CreateEscrow.tsx

Run your app and ensure you can:

The object should disappear from the list of locked objects in the Browse Locked Objects tab in the Escrows page. You view and accept or cancel escrows in later steps.

To cancel the escrow, create a mutation through the useCancelEscrowMutation hook in src/mutations/escrow.ts . The cancel function accepts the escrow ApiEscrowObject and its on-chain data. The {PACKAGE_ID}::shared::return_to_sender Move call is generic, thus it requires the type parameters of the escrowed object. Next, execute {PACKAGE_ID}::shared::return_to_sender and transfer the returned escrowed object to the creator of the escrow.

escrow.ts

Add src/components/escrows/Escrow.tsx

Add src/components/escrows/EscrowList.tsx

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

To accept the escrow, create a mutation through the useAcceptEscrowMutation hook in src/mutations/escrow.ts . The implementation should be fairly familiar to you now. The accept function accepts the escrow ApiEscrowObject and the locked object ApiLockedObject . The {PACKAGE_ID}::shared::swap Move call is generic, thus it requires the type parameters of the escrowed and locked objects. Query the objects details by using multiGetObjects on Sui client instance. Lastly, execute the {PACKAGE_ID}::shared::swap Move call and transfer the returned escrowed item to the connected wallet account. When the mutation succeeds, invalidate the cache to allow automatic re-fetch of the data.

escrow.ts

Update src/components/escrows/Escrow.tsx to include the useAcceptEscrowMutation hook

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

At this point, you have a fully functional frontend that allows users to discover trades and interact with listed escrows. The UI is designed to be user-friendly and intuitive, allowing users to easily navigate and interact with the application. Have fun exploring the app and testing out the different features!

Directory structure

To begin, create a new folder on your system titled trading that holds all your files. Inside that folder, create three more folders: api , contracts , and frontend . It's important to keep this directory structure as some helper scripts in this example target these directories by name. Different projects have their own directory structure, but it's common to split code into functional groups to help with maintenance.

In this part of the guide, you write the Move contracts that perform the trustless swaps. The guide describes how to create the package from scratch, but you can use a fork or copy of the example code in the Sui repo to follow along instead. See Write a Move Package to learn more about package structure and how to use the Sui CLI to scaffold a new project.

To begin writing your smart contracts, create an escrow folder in your contracts folder (if using recommended directory names). Create a file inside the folder named Move.toml and copy the following code into it. This is the package manifest file. If you want to learn more about the structure of the file, see Package Manifest in The Move Book.

If you are targeting a network other than Testnet, be sure to update the rev value for the Sui dependency.

With your manifest file in place, it's time to start creating the Move assets for this project. In your escrow folder, at the same level as your Move.toml file, create a sources folder. This is the common file structure of a package in Move. Create a new file inside sources titled lock.move . This file contains the logic that locks the object involved in a trade. The complete source code for this file follows and the sections that come after detail its components.

Click the titles at the top of codeblocks to open the relevant source file in GitHub.

lock.move

After a trade is initiated, you don't want the trading party to modify the object they agreed to trade. Imagine you're trading in-game items and you agree to trade a weapon with all its attachments, and its owner strips all its attachments just before the trade.

In a traditional trade, a third party typically holds the items in escrow to make sure they are not tampered with before the trade completes. This requires either trusting that the third party won't tamper with it themselves, paying the third party to ensure that doesn't happen, or both.

In a trustless swap, however, you can use the safety properties of Move to force an item's owner to prove that they have not tampered with the version of the object that you agreed to trade, without involving anyone else.

This is done by requiring that an object that is available for trading is locked with a single-use key , and asking the owner to supply the key when finalizing the trade.

To tamper with the object would require unlocking it, which consumes the key. Consequently, there would no longer be a key to finish the trade.

The lock and key are made single-use by the signatures of the lock and unlock functions. lock accepts any object of type T: store (the store ability is necessary for storing it inside a Locked), and creates both the Locked and its corresponding Key :

lock function in lock.move

The unlock function accepts the Locked and Key by value (which consumes them), and returns the underlying T as long as the correct key has been supplied for the lock:

unlock function in lock.move

Together, they ensure that a lock and key cannot have existed before the lock operation, and will not exist after a successful unlock – it is single use.

Move's type system guarantees that a given Key cannot be re-used (because unlock accepts it by value), but there are some properties that need to be confirmed with tests:

The test starts with a helper function for creating an object, it doesn't matter what kind of object it is, as long as it has the store ability. The test uses Coin , because it comes with a #[test_only] function for minting:

The first test works by creating an object to test, locking it and unlocking it – this should finish executing without aborting. The last two lines exist to keep the Move compiler happy by cleaning up the test coin and test scenario objects, because values in Move are not implicitly cleaned up unless they have the drop ability.

The other test is testing a failure scenario – that an abort happens. It creates two locked objects (this time the values are just u64 s), and use the key from one to try and unlock the other, which should fail (specified using the expected_failure attribute).

Unlike the previous test, the same clean up is not needed, because the code is expected to terminate. Instead, add another abort after the code that you expect to abort (making sure to use a different code for this second abort).

At this point, you have

From your escrow folder, run sui move test in your terminal or console. If successful, you get a response similar to the following that confirms the package builds and your tests pass:

You might notice that the Move compiler creates a build subfolder inside escrow upon a successful build. This folder contains your package's compiled bytecode, code from your package's dependencies, and various other files necessary for the build. At this point, it's enough to just be aware of these files. You don't need to fully understand the contents in build .

Create a new file in your escrow folder titled shared.move . The code in this file creates the shared Escrow object and completes the trading logic. The complete source code for this file follows and the sections that come after detail its components.

shared.move

Trading proceeds in three steps:

You can start by implementing steps two and three, by defining a new type to hold the escrowed object. It holds the escrowed object and an id: UID (because it's an object in its own right), but it also records the sender and intended recipient (to confirm they match when the trade happens), and it registers interest in the first party's object by recording the ID of the key that unlocks the Locked that contains the object.

You also need to create a function for creating the Escrow object. The object is shared because it needs to be accessed by the address that created it (in case the object needs to be returned) and by the intended recipient (to complete the swap).

create function in shared.move

If the second party stops responding, the first party can unlock their object. You need to create a function so the second party can recover their object in the symmetric case as well.

return_to_sender function in shared.move

Finally, you need to add a function to allow the first party to complete the trade.

swap function in shared.move

Tests for the escrow module are more involved than for lock – as they take advantage of test_scenario 's ability to simulate multiple transactions from different senders, and interact with shared objects.

The guide focuses on the test for a successful swap, but you can find a link to all the tests later on.

As with the lock test, start by creating a function to mint a test coin. You also create some constants to represent our transaction senders, ALICE , BOB , and DIANE .

The test body starts with a call to test_scenario::begin and ends with a call to test_scenario::end . It doesn't matter which address you pass to begin , because you pick one of ALICE or BOB at the start of each new transaction you write, so set it to @0x0 :

The first transaction is from BOB who creates a coin and locks it. You must remember the ID of the coin and the ID of the key, which you will need later, and then you transfer the locked object and the key itself to BOB , because this is what would happen in a real transaction: When simulating transactions in a test, you should only keep around primitive values, not whole objects, which would need to be written to chain between transactions.

Write these transactions inside the test_successful_swap function, between the call to begin and end .

Next, ALICE comes along and sets up the Escrow , which locks their coin. They register their interest for BOB' s coin by referencing BOB 's key's ID (ik2):

Finally, BOB completes the trade by calling swap . The take_shared function is used to simulate accepting a shared input. It uses type inference to know that the object must be an Escrow , and finds the last object of this type that was shared (by ALICE in the previous transaction). Similarly, use take_from_sender to simulate accepting owned inputs (in this case, BOB 's lock and key). The coin returned by swap is transferred back to BOB , as if it was called as part of a PTB, followed by a transfer command.

The rest of the test is designed to check that ALICE has BOB 's coin and vice versa. It starts by calling next_tx to make sure the effects of the previous transaction have been committed, before running the necessary checks.

The escrow Move package is now functional: You could publish it on chain and perform trustless swaps by creating transactions. Creating those transactions requires knowing the IDs of Locked , Key , and Escrow objects.

Locked and Key objects are typically owned by the transaction sender, and so can be queried through the Sui RPC, but Escrow objects are shared, and it is useful to be able to query them by their sender and recipient (so that users can see the trades they have offered and received).

Querying Escrow objects by their sender or recipient requires custom indexing, and to make it easy for the indexer to spot relevant transactions, add the following events to escrow.move :

Functions responsible for various aspects of the escrow's lifecycle emit these events. The custom indexer can then subscribe to transactions that emit these events and process only those, rather than the entire chain state:

emit events included in functions from shared.move

You now have shared.move and locked.move files in your sources folder. From the parent escrow folder, run sui move test in your terminal or console. If successful, you get a response similar to the following that confirms the package builds and your tests pass:

Well done. You have written the Move package!

To turn this into a complete dApp, you need to create a frontend. However, for the frontend to be updated, it has to listen to the blockchain as escrows are made and swaps are fulfilled.

To achieve this, in the next step you create an indexing service.

With the contract adapted to emit events, you can now write an indexer that keeps track of all active Escrow objects and exposes an API for querying objects by sender or recipient.

The indexer is backed by a Prisma DB with the following schema:

schema.prisma

The core of the indexer is an event loop, initialized in a function called setupListeners .

The indexer queries events related to the escrow module, using a queryEvent filter, and keeps track of a cursor representing the latest event it has processed so it can resume indexing from the right place even if it is restarted. The filter is looking for any events whose type is from the escrow module of the Move package (see the event-indexer.ts code that follows).

The core event job works by polling: It queries RPC for events following its latest cursor and sends them to a callback for processing. If it detects more than one page of new events, it immediately requests the next page. Otherwise, the job waits for the next polling interval before checking again.

event-indexer.ts

The callback is responsible for reading the event and updating the database accordingly. For demo purposes, SQLite is being used, and so you need to issue a separate UPSERT to the database for each escrowed object. In a production setting, however, you would want to batch requests to the database to optimize data flow.

escrow-handler.ts

The data that the indexer captures can then be served over an API, so that a frontend can read it. Follow the next section to implement the API in TypeScript, to run on Node, using Express.

You want your API to accept the query string in the URL as the parameters for database WHERE query. Hence, you want a utility that can extract and parse the URL query string into valid query parameters for Prisma. With the parseWhereStatement() function, the callers filter the set of keys from the URL query string and transforms those corresponding key-value pairs into the correct format for Prisma.

parseWhereStatement in api-queries.ts

Pagination is another crucial part to ensure your API returns sufficient and/or ordered chunk of information instead of all the data that might be the vector for a DDOS attack. Similar to WHERE parameters , define a set of keys in the URL query string to be accepted as valid pagination parameters. The parsePaginationForQuery() utility function helps to achieve this by filtering the pre-determined keys sort , limit , cursor and parsing corresponding key-value pairs into ApiPagination that Prisma can consume.

In this example, the id field of the model in the database as the cursor that allows clients to continue subsequent queries with the next page.

parsePaginationForQuery in api-queries.ts

All the endpoints are defined in server.ts , particularly, there are two endpoints:

You define a list of valid query keys, such as deleted , creator , keyId , and objectId for Locked data and cancelled , swapped , recipient , and sender for Escrow data. Pass the URL query string into the pre-defined utilities to output the correct parameters that Prisma can use.

server.ts

Now that you have an indexer and an API service, you can deploy your move package and start the indexer and API service.

Install dependencies by running pnpm install --ignore-workspace or yarn install --ignore-workspace .

Setup the database by running pnpm db:setup:dev or yarn db:setup:dev .

Deploy the Sui package

Deployment instructions

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.

Use the following command to list your available environments:

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, run the following command:

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.

There are some helper functions to publish the smart contracts so you can create some demo data (for Testnet). The helper function to publish the smart contrqcts expects built smart contracts in both the escrow and demo directories. Run sui move build in both directories, if necessary. Be sure to update the Sui dependency in the manifest to point to the correct source based on your environment.

To publish the smart contracts and produce demo data:

Publish the smart contracts by running the following command from your api folder:

If successful, demo-contract.json and escrow-contract.json are created in the backend root directory. These files contain the contract addresses and are used by the backend and frontend to interact with the contracts.

Produce demo non-locked and locked objects

Produce demo escrows

If you want to reset the database (for a clean demo, for example), run pnpm db:reset:dev && pnpm db:setup:dev or yarn db:reset:dev && yarn db:setup:dev .

Run both the API and the indexer by running pnpm dev or yarn dev .

Visit http://localhost:3000/escrows or http://localhost:3000/locked

You should now have an indexer running.

With the code successfully deployed on Testnet, you can now create a frontend to display the trading data and to allow users to interact with the Move modules.

In this final part of the app example, you build a frontend (UI) that allows end users to discover trades and interact with listed escrows.

You can view the complete source code for this app example in the Sui repository.

Before getting started, make sure you have:

The UI design consists of three parts:

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

When asked for a name for your dApp, provide one of your liking. The dApp scaffold gets created in a new directory with the name you provide. This is convenient to keep your working code separate from the example source code that might already populate this folder. The codeblocks that follow point to the code in the default example location. Be aware the path to your own code includes the dApp name you provide.

First, set up import aliases to make the code more readable and maintainable. This allows you to import files using @/ instead of relative paths.

Replace the content of tsconfig.json with the following:

The paths option under compilerOptions is what defines the aliasing for TypeScript. Here, the alias @/ *is mapped to the ./src/* directory, meaning that any time you use @/ , TypeScript resolves it as a reference to the src folder. This setup reduces the need for lengthy relative paths when importing files in your project.

Replace the content of vite.config.ts with the following:

Vite also needs to be aware of the aliasing to resolve imports correctly during the build process. In the resolve.alias configuration of vite.config.ts , we map the alias @ to the /src directory.

To streamline the styling process and keep the codebase clean and maintainable, this guide uses Tailwind CSS, which provides utility-first CSS classes to rapidly build custom designs. Run the following command from the base of your dApp project to add Tailwind CSS and its dependencies:

Next, generate the Tailwind CSS configuration file by running the following:

Replace the content of tailwind.config.js with the following:

Add the src/styles/ directory and add base.css :

First, deploy your package via the scripts in the api directory .

Then, create a src/constants.ts file and fill it with the following:

If you create a dApp using a project name so that your src files are in a subfolder of frontend , be sure to add another nesting level (../) to the import statements.

Create a src/utils/ directory and add the following file:

Create a src/components/ directory and add the following components:

ExplorerLink.tsx

InfiniteScrollArea.tsx

Loading.tsx

SuiObjectDisplay.tsx

Install the necessary dependencies:

The imported template only has a single page. To add more pages, you need to set up routing.

First, install the necessary dependencies:

Then, create a src/routes/ directory and add index.tsx . This file contains the routing configuration:

Add the following respective files to the src/routes/ directory:

root.tsx . This file contains the root component that is rendered on every page:

LockedDashboard.tsx . This file contains the component for the Manage Objects page.

EscrowDashboard.tsx . This file contains the component for the Escrows page.

Update src/main.tsx by replacing the App component with the RouterProvider and replace "dark" with "light" in the Theme component:

Note that dApp Kit provides a set of hooks for making query and mutation calls to the Sui blockchain. These hooks are thin wrappers around query and mutation hooks from @tanstack/react-query .

Create src/components/Header.tsx . This file contains the navigation links and the connect wallet button:

The dApp Kit comes with a pre-built React.js component called ConnectButton displaying a button to connect and disconnect a wallet. The connecting and disconnecting wallet logic is handled seamlessly so you don't need to worry about repeating yourself doing the same logic all over again.

At this point, you have a basic routing setup. Run your app and ensure you can:

Note, the styles should be applied. The Header component should look like this:

All the type definitions are in src/types/types.ts . Create this file and add the following:

ApiLockedObject and ApiEscrowObject represent the Locked and Escrow indexed data model the indexing and API service return.

EscrowListingQuery and LockedListingQuery are the query parameters model to provide to the API service to fetch from the endpoints /escrow and /locked accordingly.

Now, display the objects owned by the connected wallet address. This is the Manage Objects page.

First add this file src/components/locked/LockOwnedObjects.tsx :

Fetch the owned objects directly from Sui blockchain using the useSuiClientInfiniteQuery() hook from dApp Kit . This hook is a thin wrapper around Sui blockchain RPC calls, reference the documentation to learn more about these RPC hooks . Basically, supply the RPC endpoint you want to execute, in this case it's the getOwnedObjects endpoint . Supply the connected wallet account as the owner . The returned data is stored inside the cache at query key getOwnedObjects . In a future step you invalidate this cache after a mutation succeeds, so the data will be re-fetched automatically.

Next, update src/routes/LockedDashboard.tsx to include the LockOwnedObjects component:

Run your app and ensure you can:

If you don't see any objects, you might need to create some demo data or connect your wallet. You can mint objects after completing the next steps.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's create and examine the execute transaction hook.

Create src/hooks/useTransactionExecution.ts :

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useSuiClient() hook from dApp Kit to retrieve the Sui client instance configured in src/main.tsx . The useSignTransaction() function is another hook from dApp kit that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, use the executeTransaction() on the Sui client instance of the Sui TypeScript SDK.

The full source code of the demo bear smart contract is available at Trading Contracts Demo directory

You need a utility function to create a dummy object representing a real world asset so you can use it to test and demonstrate escrow users flow on the UI directly.

Create src/mutations/demo.ts :

As previously mentioned, this example uses @tanstack/react-query to query, cache, and mutate server state. Server state is data only available on remote servers, and the only way to retrieve or update this data is by interacting with these remote servers. In this case, it could be from an API or directly from Sui blockchain RPC.

When you execute a transaction call to mutate data on the Sui blockchain, use the useMutation() hook. The useMutation() hook accepts several inputs, however, you only need two of them for this example. The first parameter, mutationFn , accepts the function to execute the main mutating logic, while the second parameter, onSuccess , is a callback that runs when the mutating logic succeeds.

The main mutating logic includes executing a Move call of a package named demo_bear::new to create a dummy bear object and transferring it to the connected wallet account, all within the same Transaction . This example reuses the executeTransaction() hook from the Execute Transaction Hook step to execute the transaction.

Another benefit of wrapping the main mutating logic inside useMutation() is that you can access and manipulate the cache storing server state. This example fetches the cache from remote servers by using query call in an appropriate callback. In this case, it is the onSuccess callback. When the transaction succeeds, invalidate the cache data at the cache key called getOwnedObjects , then @tanstack/react-query handles the re-fetching mechanism for the invalidated data automatically. Do this by using invalidateQueries() on the @tanstack/react-query configured client instance retrieved by useQueryClient() hook in the Set up Routing step.

Now the logic to create a dummy bear object exists. You just need to attach it into the button in the header.

Header.tsx

Run your app and ensure you can:

To lock the object, execute the lock Move function identified by {PACKAGE_ID}::lock::lock . The implementation is similar to what's in previous mutation functions, use useMutation() from @tanstack/react-query to wrap the main logic inside it. The lock function requires an object to be locked and its type because our smart contract lock function is generic and requires type parameters. After creating a Locked object and its Key object, transfer them to the connected wallet account within the same transaction block.

It's beneficial to extract logic of locking owned objects into a separated mutating function to enhance discoverability and encapsulation.

Create src/mutations/locked.ts :

Update src/components/locked/LockOwnedObjects.tsx to include the useLockObjectMutation hook:

LockOwnedObjects.tsx

Run your app and ensure you can:

The object should disappear from the list of owned objects. You view and unlock locked objects in later steps.

Let's take a look at the My Locked Objects tab by examining src/components/locked/OwnedLockedList.tsx . Focus on the logic on how to retrieve this list.

OwnedLockedList.tsx

This instance of useSuiClientInfiniteQuery() is similar to the one in the LockOwnedObjects component. The difference is that it fetches the locked objects instead of the owned objects. The Locked object is a struct type in the smart contract, so you need to supply the struct type to the query call as a filter . The struct type is usually identified by the format of {PACKAGE_ID}::{MODULE_NAME}::{STRUCT_TYPE} .

The (src/components/locked/LockedObject.tsx) component is mainly responsible for mapping an on-chain SuiObjectData Locked object to its corresponding ApiLockedObject , which is finally delegated to the component for rendering. The fetches the locked item object ID if the prop itemId is not supplied by using dApp Kit useSuiClientQuery() hook to call the getDynamicFieldObject RPC endpoint. Recalling that in this smart contract, the locked item is put into a dynamic object field.

LockedObject.tsx

The (src/components/locked/partials/Locked.tsx) component is mainly responsible for rendering the ApiLockedObject . Later on, it will also consist of several on-chain interactions: unlock the locked objects and create an escrow out of the locked object.

Locked.tsx

Update src/routes/LockedDashboard.tsx to include the OwnedLockedList component:

LockedDashboard.tsx

Run your app and ensure you can:

To unlock the object, execute the unlock Move function identified by {PACKAGE_ID}::lock::unlock . Call the unlock function supplying the Locked object, its corresponding Key , the struct type of the original object, and transfer the unlocked object to the current connected wallet account. Also, implement the onSuccess callback to invalidate the cache data at query key locked after one second to force react-query to re-fetch the data at corresponding query key automatically.

Unlocking owned objects is another crucial and complex on-chain action in this application. Hence, it's beneficial to extract its logic into separated mutating functions to enhance discoverability and encapsulation.

src/mutations/locked.ts

Update src/components/locked/partials/Locked.tsx to include the useUnlockObjectMutation hook:

Locked.tsx

Run your app and ensure you can:

Update src/routes/EscrowDashboard.tsx to include the LockedList component:

EscrowDashboard.tsx

Add src/components/locked/ApiLockedList.tsx :

ApiLockedList.tsx

This hook fetches all the non-deleted system Locked objects from the API in a paginated fashion. Then, it proceeds into fetching the on-chain state, to ensure the latest state of the object is displayed.

This component uses tanstack's useInfiniteQuery instead of useSuiClientInfiniteQuery since the data is being fetched from the example's API rather than Sui.

Add src/hooks/useGetLockedObject.ts

Run your app and ensure you can:

To create escrows, include a mutating function through the useCreateEscrowMutation hook in src/mutations/escrow.ts . It accepts the escrowed item to be traded and the ApiLockedObject from another party as parameters. Then, call the {PACKAGE_ID}::shared::create Move function and provide the escrowed item, the key id of the locked object to exchange, and the recipient of the escrow (locked object's owner).

escrow.ts

Update src/components/locked/partials/Locked.tsx to include the useCreateEscrowMutation hook

Add src/components/escrows/CreateEscrow.tsx

Run your app and ensure you can:

The object should disappear from the list of locked objects in the Browse Locked Objects tab in the Escrows page. You view and accept or cancel escrows in later steps.

To cancel the escrow, create a mutation through the useCancelEscrowMutation hook in src/mutations/escrow.ts . The cancel function accepts the escrow ApiEscrowObject and its on-chain data. The {PACKAGE_ID}::shared::return_to_sender Move call is generic, thus it requires the type parameters of the escrowed object. Next, execute {PACKAGE_ID}::shared::return_to_sender and transfer the returned escrowed object to the creator of the escrow.

escrow.ts

Add src/components/escrows/Escrow.tsx

Add src/components/escrows/EscrowList.tsx

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

To accept the escrow, create a mutation through the useAcceptEscrowMutation hook in src/mutations/escrow.ts . The implementation should be fairly familiar to you now. The accept function accepts the escrow ApiEscrowObject and the locked object ApiLockedObject . The {PACKAGE_ID}::shared::swap Move call is generic, thus it requires the type parameters of the escrowed and locked objects. Query the objects details by using multiGetObjects on Sui client instance. Lastly, execute the {PACKAGE_ID}::shared::swap Move call and transfer the returned escrowed item to the connected wallet account. When the mutation succeeds, invalidate the cache to allow automatic re-fetch of the data.

escrow.ts

Update src/components/escrows/Escrow.tsx to include the useAcceptEscrowMutation hook

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

At this point, you have a fully functional frontend that allows users to discover trades and interact with listed escrows. The UI is designed to be user-friendly and intuitive, allowing users to easily navigate and interact with the application. Have fun exploring the app and testing out the different features!

Smart contracts

In this part of the guide, you write the Move contracts that perform the trustless swaps. The guide describes how to create the package from scratch, but you can use a fork or copy of the example code in the Sui repo to follow along instead. See Write a Move Package to learn more about package structure and how to use the Sui CLI to scaffold a new project.

To begin writing your smart contracts, create an escrow folder in your contracts folder (if using recommended directory names). Create a file inside the folder named Move.toml and copy the following code into it. This is the package manifest file. If you want to learn more about the structure of the file, see Package Manifest in The Move Book.

If you are targeting a network other than Testnet, be sure to update the rev value for the Sui dependency.

With your manifest file in place, it's time to start creating the Move assets for this project. In your escrow folder, at the same level as your Move.toml file, create a sources folder. This is the common file structure of a package in Move. Create a new file inside sources titled lock.move . This file contains the logic that locks the object involved in a trade. The complete source code for this file follows and the sections that come after detail its components.

Click the titles at the top of codeblocks to open the relevant source file in GitHub.

lock.move

After a trade is initiated, you don't want the trading party to modify the object they agreed to trade. Imagine you're trading in-game items and you agree to trade a weapon with all its attachments, and its owner strips all its attachments just before the trade.

In a traditional trade, a third party typically holds the items in escrow to make sure they are not tampered with before the trade completes. This requires either trusting that the third party won't tamper with it themselves, paying the third party to ensure that doesn't happen, or both.

In a trustless swap, however, you can use the safety properties of Move to force an item's owner to prove that they have not tampered with the version of the object that you agreed to trade, without involving anyone else.

This is done by requiring that an object that is available for trading is locked with a single-use key , and asking the owner to supply the key when finalizing the trade.

To tamper with the object would require unlocking it, which consumes the key. Consequently, there would no longer be a key to finish the trade.

The lock and key are made single-use by the signatures of the lock and unlock functions. lock accepts any object of type T: store (the store ability is necessary for storing it inside a Locked), and creates both the Locked and its corresponding Key :

lock function in lock.move

The unlock function accepts the Locked and Key by value (which consumes them), and returns the underlying T as long as the correct key has been supplied for the lock:

unlock function in lock.move

Together, they ensure that a lock and key cannot have existed before the lock operation, and will not exist after a successful unlock – it is single use.

Move's type system guarantees that a given Key cannot be re-used (because unlock accepts it by value), but there are some properties that need to be confirmed with tests:

The test starts with a helper function for creating an object, it doesn't matter what kind of object it is, as long as it has the store ability. The test uses Coin , because it comes with a #[test_only] function for minting:

The first test works by creating an object to test, locking it and unlocking it – this should finish executing without aborting. The last two lines exist to keep the Move compiler happy by cleaning up the test coin and test scenario objects, because values in Move are not implicitly cleaned up unless they have the drop ability.

The other test is testing a failure scenario – that an abort happens. It creates two locked objects (this time the values are just u64 s), and use the key from one to try and unlock the other, which should fail (specified using the expected_failure attribute).

Unlike the previous test, the same clean up is not needed, because the code is expected to terminate. Instead, add another abort after the code that you expect to abort (making sure to use a different code for this second abort).

At this point, you have

From your escrow folder, run sui move test in your terminal or console. If successful, you get a response similar to the following that confirms the package builds and your tests pass:

You might notice that the Move compiler creates a build subfolder inside escrow upon a successful build. This folder contains your package's compiled bytecode, code from your package's dependencies, and various other files necessary for the build. At this point, it's enough to just be aware of these files. You don't need to fully understand the contents in build .

Create a new file in your escrow folder titled shared.move . The code in this file creates the shared Escrow object and completes the trading logic. The complete source code for this file follows and the sections that come after detail its components.

shared.move

Trading proceeds in three steps:

You can start by implementing steps two and three, by defining a new type to hold the escrowed object. It holds the escrowed object and an id: UID (because it's an object in its own right), but it also records the sender and intended recipient (to confirm they match when the trade happens), and it registers interest in the first party's object by recording the ID of the key that unlocks the Locked that contains the object.

You also need to create a function for creating the Escrow object. The object is shared because it needs to be accessed by the address that created it (in case the object needs to be returned) and by the intended recipient (to complete the swap).

create function in shared.move

If the second party stops responding, the first party can unlock their object. You need to create a function so the second party can recover their object in the symmetric case as well.

return_to_sender function in shared.move

Finally, you need to add a function to allow the first party to complete the trade.

swap function in shared.move

Tests for the escrow module are more involved than for lock – as they take advantage of test_scenario 's ability to simulate multiple transactions from different senders, and interact with shared objects.

The guide focuses on the test for a successful swap, but you can find a link to all the tests later on.

As with the lock test, start by creating a function to mint a test coin. You also create some constants to represent our transaction senders, ALICE , BOB , and DIANE .

The test body starts with a call to test_scenario::begin and ends with a call to test_scenario::end . It doesn't matter which address you pass to begin , because you pick one of ALICE or BOB at the start of each new transaction you write, so set it to @0x0 :

The first transaction is from BOB who creates a coin and locks it. You must remember the ID of the coin and the ID of the key, which you will need later, and then you transfer the locked object and the key itself to BOB , because this is what would happen in a real transaction: When simulating transactions in a test, you should only keep around primitive values, not whole objects, which would need to be written to chain between transactions.

Write these transactions inside the test_successful_swap function, between the call to begin and end .

Next, ALICE comes along and sets up the Escrow , which locks their coin. They register their interest for BOB' s coin by referencing BOB 's key's ID (ik2):

Finally, BOB completes the trade by calling swap . The take_shared function is used to simulate accepting a shared input. It uses type inference to know that the object must be an Escrow , and finds the last object of this type that was shared (by ALICE in the previous transaction). Similarly, use take_from_sender to simulate accepting owned inputs (in this case, BOB 's lock and key). The coin returned by swap is transferred back to BOB , as if it was called as part of a PTB, followed by a transfer command.

The rest of the test is designed to check that ALICE has BOB 's coin and vice versa. It starts by calling next_tx to make sure the effects of the previous transaction have been committed, before running the necessary checks.

The escrow Move package is now functional: You could publish it on chain and perform trustless swaps by creating transactions. Creating those transactions requires knowing the IDs of Locked , Key , and Escrow objects.

Locked and Key objects are typically owned by the transaction sender, and so can be queried through the Sui RPC, but Escrow objects are shared, and it is useful to be able to query them by their sender and recipient (so that users can see the trades they have offered and received).

Querying Escrow objects by their sender or recipient requires custom indexing, and to make it easy for the indexer to spot relevant transactions, add the following events to escrow.move :

Functions responsible for various aspects of the escrow's lifecycle emit these events. The custom indexer can then subscribe to transactions that emit these events and process only those, rather than the entire chain state:

emit events included in functions from shared.move

You now have shared.move and locked.move files in your sources folder. From the parent escrow folder, run sui move test in your terminal or console. If successful, you get a response similar to the following that confirms the package builds and your tests pass:

Well done. You have written the Move package!

To turn this into a complete dApp, you need to create a frontend. However, for the frontend to be updated, it has to listen to the blockchain as escrows are made and swaps are fulfilled.

To achieve this, in the next step you create an indexing service.

With the contract adapted to emit events, you can now write an indexer that keeps track of all active Escrow objects and exposes an API for querying objects by sender or recipient.

The indexer is backed by a Prisma DB with the following schema:

schema.prisma

The core of the indexer is an event loop, initialized in a function called setupListeners .

The indexer queries events related to the escrow module, using a queryEvent filter, and keeps track of a cursor representing the latest event it has processed so it can resume indexing from the right place even if it is restarted. The filter is looking for any events whose type is from the escrow module of the Move package (see the event-indexer.ts code that follows).

The core event job works by polling: It queries RPC for events following its latest cursor and sends them to a callback for processing. If it detects more than one page of new events, it immediately requests the next page. Otherwise, the job waits for the next polling interval before checking again.

event-indexer.ts

The callback is responsible for reading the event and updating the database accordingly. For demo purposes, SQLite is being used, and so you need to issue a separate UPSERT to the database for each escrowed object. In a production setting, however, you would want to batch requests to the database to optimize data flow.

escrow-handler.ts

The data that the indexer captures can then be served over an API, so that a frontend can read it. Follow the next section to implement the API in TypeScript, to run on Node, using Express.

You want your API to accept the query string in the URL as the parameters for database WHERE query. Hence, you want a utility that can extract and parse the URL query string into valid query parameters for Prisma. With the parseWhereStatement() function, the callers filter the set of keys from the URL query string and transforms those corresponding key-value pairs into the correct format for Prisma.

parseWhereStatement in api-queries.ts

Pagination is another crucial part to ensure your API returns sufficient and/or ordered chunk of information instead of all the data that might be the vector for a DDOS attack. Similar to WHERE parameters , define a set of keys in the URL query string to be accepted as valid pagination parameters. The parsePaginationForQuery() utility function helps to achieve this by filtering the pre-determined keys sort , limit , cursor and parsing corresponding key-value pairs into ApiPagination that Prisma can consume.

In this example, the id field of the model in the database as the cursor that allows clients to continue subsequent queries with the next page.

parsePaginationForQuery in api-queries.ts

All the endpoints are defined in server.ts , particularly, there are two endpoints:

You define a list of valid query keys, such as deleted , creator , keyId , and objectId for Locked data and cancelled , swapped , recipient , and sender for Escrow data. Pass the URL query string into the pre-defined utilities to output the correct parameters that Prisma can use.

server.ts

Now that you have an indexer and an API service, you can deploy your move package and start the indexer and API service.

Install dependencies by running pnpm install --ignore-workspace or yarn install --ignore-workspace .

Setup the database by running pnpm db:setup:dev or yarn db:setup:dev .

Deploy the Sui package

Deployment instructions

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.

Use the following command to list your available environments:

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, run the following command:

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.

There are some helper functions to publish the smart contracts so you can create some demo data (for Testnet). The helper function to publish the smart contrqcts expects built smart contracts in both the escrow and demo directories. Run sui move build in both directories, if necessary. Be sure to update the Sui dependency in the manifest to point to the correct source based on your environment.

To publish the smart contracts and produce demo data:

Publish the smart contracts by running the following command from your api folder:

If successful, demo-contract.json and escrow-contract.json are created in the backend root directory. These files contain the contract addresses and are used by the backend and frontend to interact with the contracts.

Produce demo non-locked and locked objects

Produce demo escrows

If you want to reset the database (for a clean demo, for example), run pnpm db:reset:dev && pnpm db:setup:dev or yarn db:reset:dev && yarn db:setup:dev .

Run both the API and the indexer by running pnpm dev or yarn dev .

Visit http://localhost:3000/escrows or http://localhost:3000/locked

You should now have an indexer running.

With the code successfully deployed on Testnet, you can now create a frontend to display the trading data and to allow users to interact with the Move modules.

In this final part of the app example, you build a frontend (UI) that allows end users to discover trades and interact with listed escrows.

You can view the complete source code for this app example in the Sui repository.

Before getting started, make sure you have:

The UI design consists of three parts:

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

When asked for a name for your dApp, provide one of your liking. The dApp scaffold gets created in a new directory with the name you provide. This is convenient to keep your working code separate from the example source code that might already populate this folder. The codeblocks that follow point to the code in the default example location. Be aware the path to your own code includes the dApp name you provide.

First, set up import aliases to make the code more readable and maintainable. This allows you to import files using @/ instead of relative paths.

Replace the content of tsconfig.json with the following:

The paths option under compilerOptions is what defines the aliasing for TypeScript. Here, the alias @/ *is mapped to the ./src/* directory, meaning that any time you use @/ , TypeScript resolves it as a reference to the src folder. This setup reduces the need for lengthy relative paths when importing files in your project.

Replace the content of vite.config.ts with the following:

Vite also needs to be aware of the aliasing to resolve imports correctly during the build process. In the resolve.alias configuration of vite.config.ts , we map the alias @ to the /src directory.

To streamline the styling process and keep the codebase clean and maintainable, this guide uses Tailwind CSS, which provides utility-first CSS classes to rapidly build custom designs. Run the following command from the base of your dApp project to add

Tailwind CSS and its dependencies:

Next, generate the Tailwind CSS configuration file by running the following:

Replace the content of tailwind.config.js with the following:

Add the src/styles/ directory and add base.css :

First, deploy your package via the scripts in the api directory .

Then, create a src/constants.ts file and fill it with the following:

If you create a dApp using a project name so that your src files are in a subfolder of frontend , be sure to add another nesting level (../) to the import statements.

Create a src/utils/ directory and add the following file:

Create a src/components/ directory and add the following components:

ExplorerLink.tsx

InfiniteScrollArea.tsx

Loading.tsx

SuiObjectDisplay.tsx

Install the necessary dependencies:

The imported template only has a single page. To add more pages, you need to set up routing.

First, install the necessary dependencies:

Then, create a src/routes/ directory and add index.tsx . This file contains the routing configuration:

Add the following respective files to the src/routes/ directory:

root.tsx . This file contains the root component that is rendered on every page:

LockedDashboard.tsx . This file contains the component for the Manage Objects page.

EscrowDashboard.tsx . This file contains the component for the Escrows page.

Update src/main.tsx by replacing the App component with the RouterProvider and replace "dark" with "light" in the Theme component:

Note that dApp Kit provides a set of hooks for making query and mutation calls to the Sui blockchain. These hooks are thin wrappers around query and mutation hooks from @tanstack/react-query .

Create src/components/Header.tsx . This file contains the navigation links and the connect wallet button:

The dApp Kit comes with a pre-built React.js component called ConnectButton displaying a button to connect and disconnect a wallet. The connecting and disconnecting wallet logic is handled seamlessly so you don't need to worry about repeating yourself doing the same logic all over again.

At this point, you have a basic routing setup. Run your app and ensure you can:

Note, the styles should be applied. The Header component should look like this:

All the type definitions are in src/types/types.ts . Create this file and add the following:

ApiLockedObject and ApiEscrowObject represent the Locked and Escrow indexed data model the indexing and API service return.

EscrowListingQuery and LockedListingQuery are the query parameters model to provide to the API service to fetch from the endpoints /escrow and /locked accordingly.

Now, display the objects owned by the connected wallet address. This is the Manage Objects page.

First add this file src/components/locked/LockOwnedObjects.tsx :

Fetch the owned objects directly from Sui blockchain using the useSuiClientInfiniteQuery() hook from dApp Kit . This hook is a thin wrapper around Sui blockchain RPC calls, reference the documentation to learn more about these RPC hooks . Basically, supply the RPC endpoint you want to execute, in this case it's the getOwnedObjects endpoint . Supply the connected wallet account as the owner . The returned data is stored inside the cache at query key getOwnedObjects . In a future step you invalidate this cache after a mutation succeeds, so the data will be re-fetched automatically.

Next, update src/routes/LockedDashboard.tsx to include the LockOwnedObjects component:

Run your app and ensure you can:

If you don't see any objects, you might need to create some demo data or connect your wallet. You can mint objects after completing the next steps.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's create and examine the execute transaction hook.

Create src/hooks/useTransactionExecution.ts :

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useSuiClient() hook from dApp Kit to retrieve the Sui client instance configured in src/main.tsx . The useSignTransaction() function is another hook from dApp kit that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, use the executeTransaction() on the Sui client instance of the Sui TypeScript SDK.

The full source code of the demo bear smart contract is available at Trading Contracts Demo directory

You need a utility function to create a dummy object representing a real world asset so you can use it to test and demonstrate escrow users flow on the UI directly.

Create src/mutations/demo.ts :

As previously mentioned, this example uses @tanstack/react-query to query, cache, and mutate server state. Server state is data only available on remote servers, and the only way to retrieve or update this data is by interacting with these remote servers. In this case, it could be from an API or directly from Sui blockchain RPC.

When you execute a transaction call to mutate data on the Sui blockchain, use the useMutation() hook. The useMutation() hook accepts several inputs, however, you only need two of them for this example. The first parameter, mutationFn , accepts the function to execute the main mutating logic, while the second parameter, onSuccess , is a callback that runs when the mutating logic succeeds.

The main mutating logic includes executing a Move call of a package named demo_bear::new to create a dummy bear object and transferring it to the connected wallet account, all within the same Transaction . This example reuses the executeTransaction() hook from the Execute Transaction Hook step to execute the transaction.

Another benefit of wrapping the main mutating logic inside useMutation() is that you can access and manipulate the cache storing server state. This example fetches the cache from remote servers by using query call in an appropriate callback. In this case, it is the onSuccess callback. When the transaction succeeds, invalidate the cache data at the cache key called getOwnedObjects , then @tanstack/react-query handles the re-fetching mechanism for the invalidated data automatically. Do this by using invalidateQueries() on the @tanstack/react-query configured client instance retrieved by useQueryClient() hook in the Set up Routing step.

Now the logic to create a dummy bear object exists. You just need to attach it into the button in the header.

Header.tsx

Run your app and ensure you can:

To lock the object, execute the lock Move function identified by {PACKAGE_ID}::lock::lock . The implementation is similar to what's in previous mutation functions, use useMutation() from @tanstack/react-query to wrap the main logic inside it. The lock function requires an object to be locked and its type because our smart contract lock function is generic and requires type parameters. After creating a Locked object and its Key object, transfer them to the connected wallet account within the same transaction block.

It's beneficial to extract logic of locking owned objects into a separated mutating function to enhance discoverability and encapsulation.

Create src/mutations/locked.ts :

Update src/components/locked/LockOwnedObjects.tsx to include the useLockObjectMutation hook:

LockOwnedObjects.tsx

Run your app and ensure you can:

The object should disappear from the list of owned objects. You view and unlock locked objects in later steps.

Let's take a look at the My Locked Objects tab by examining src/components/locked/OwnedLockedList.tsx . Focus on the logic on how to retrieve this list.

OwnedLockedList.tsx

This instance of useSuiClientInfiniteQuery() is similar to the one in the LockOwnedObjects component. The difference is that it fetches the locked objects instead of the owned objects. The Locked object is a struct type in the smart contract, so you need to supply the struct type to the query call as a filter . The struct type is usually identified by the format of {PACKAGE_ID}::{MODULE_NAME}::{STRUCT_TYPE} .

The (src/components/locked/LockedObject.tsx) component is mainly responsible for mapping an on-chain SuiObjectData Locked object to its corresponding ApiLockedObject , which is finally delegated to the component for rendering. The fetches the locked item object ID if the prop itemId is not supplied by using dApp Kit useSuiClientQuery() hook to call the getDynamicFieldObject RPC endpoint. Recalling that in this smart contract, the locked item is put into a dynamic object field.

LockedObject.tsx

The (src/components/locked/partials/Locked.tsx) component is mainly responsible for rendering the ApiLockedObject . Later on, it will also consist of several on-chain interactions: unlock the locked objects and create an escrow out of the locked object.

Locked.tsx

Update src/routes/LockedDashboard.tsx to include the OwnedLockedList component:

LockedDashboard.tsx

Run your app and ensure you can:

To unlock the object, execute the unlock Move function identified by {PACKAGE_ID}::lock::unlock . Call the unlock function supplying the Locked object, its corresponding Key , the struct type of the original object, and transfer the unlocked object to the current connected wallet account. Also, implement the onSuccess callback to invalidate the cache data at query key locked after one second to force react-query to re-fetch the data at corresponding query key automatically.

Unlocking owned objects is another crucial and complex on-chain action in this application. Hence, it's beneficial to extract its logic into separated mutating functions to enhance discoverability and encapsulation.

src/mutations/locked.ts

Update src/components/locked/partials/Locked.tsx to include the useUnlockObjectMutation hook:

Locked.tsx

Run your app and ensure you can:

Update src/routes/EscrowDashboard.tsx to include the LockedList component:

EscrowDashboard.tsx

Add src/components/locked/ApiLockedList.tsx :

ApiLockedList.tsx

This hook fetches all the non-deleted system Locked objects from the API in a paginated fashion. Then, it proceeds into fetching the on-chain state, to ensure the latest state of the object is displayed.

This component uses tanstack's useInfiniteQuery instead of useSuiClientInfiniteQuery since the data is being fetched from the example's API rather than Sui.

Add src/hooks/useGetLockedObject.ts

Run your app and ensure you can:

To create escrows, include a mutating function through the useCreateEscrowMutation hook in src/mutations/escrow.ts . It accepts the escrowed item to be traded and the ApiLockedObject from another party as parameters. Then, call the {PACKAGE_ID}::shared::create Move function and provide the escrowed item, the key id of the locked object to exchange, and the recipient of the escrow (locked object's owner).

escrow.ts

Update src/components/locked/partials/Locked.tsx to include the useCreateEscrowMutation hook

Add src/components/escrows/CreateEscrow.tsx

Run your app and ensure you can:

The object should disappear from the list of locked objects in the Browse Locked Objects tab in the Escrows page. You view and accept or cancel escrows in later steps.

To cancel the escrow, create a mutation through the useCancelEscrowMutation hook in src/mutations/escrow.ts . The cancel function accepts the escrow ApiEscrowObject and its on-chain data. The {PACKAGE_ID}::shared::return_to_sender Move call is generic, thus it requires the type parameters of the escrowed object. Next, execute {PACKAGE_ID}::shared::return_to_sender and transfer the returned escrowed object to the creator of the escrow.

escrow.ts

Add src/components/escrows/Escrow.tsx

Add src/components/escrows/EscrowList.tsx

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

To accept the escrow, create a mutation through the useAcceptEscrowMutation hook in src/mutations/escrow.ts . The implementation should be fairly familiar to you now. The accept function accepts the escrow ApiEscrowObject and the locked object ApiLockedObject . The {PACKAGE_ID}::shared::swap Move call is generic, thus it requires the type parameters of the escrowed and locked objects. Query the objects details by using multiGetObjects on Sui client instance. Lastly, execute the {PACKAGE_ID}::shared::swap Move call and transfer the returned escrowed item to the connected wallet account. When the mutation succeeds, invalidate the cache to allow automatic re-fetch of the data.

escrow.ts

Update src/components/escrows/Escrow.tsx to include the useAcceptEscrowMutation hook

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

At this point, you have a fully functional frontend that allows users to discover trades and interact with listed escrows. The UI is designed to be user-friendly and intuitive, allowing users to easily navigate and interact with the application. Have fun exploring the app and testing out the different features!

Backend indexer

With the contract adapted to emit events, you can now write an indexer that keeps track of all active Escrow objects and exposes an API for querying objects by sender or recipient.

The indexer is backed by a Prisma DB with the following schema:

schema.prisma

The core of the indexer is an event loop, initialized in a function called setupListeners .

The indexer queries events related to the escrow module, using a queryEvent filter, and keeps track of a cursor representing the latest event it has processed so it can resume indexing from the right place even if it is restarted. The filter is looking for any events whose type is from the escrow module of the Move package (see the event-indexer.ts code that follows).

The core event job works by polling: It queries RPC for events following its latest cursor and sends them to a callback for processing. If it detects more than one page of new events, it immediately requests the next page. Otherwise, the job waits for the next polling interval before checking again.

event-indexer.ts

The callback is responsible for reading the event and updating the database accordingly. For demo purposes, SQLite is being used, and so you need to issue a separate UPSERT to the database for each escrowed object. In a production setting, however, you would want to batch requests to the database to optimize data flow.

escrow-handler.ts

The data that the indexer captures can then be served over an API, so that a frontend can read it. Follow the next section to implement the API in TypeScript, to run on Node, using Express.

You want your API to accept the query string in the URL as the parameters for database WHERE query. Hence, you want a utility that can extract and parse the URL query string into valid query parameters for Prisma. With the parseWhereStatement() function, the callers filter the set of keys from the URL query string and transforms those corresponding key-value pairs into the correct format for Prisma.

parseWhereStatement in api-queries.ts

Pagination is another crucial part to ensure your API returns sufficient and/or ordered chunk of information instead of all the data that might be the vector for a DDOS attack. Similar to WHERE parameters , define a set of keys in the URL query string to be accepted as valid pagination parameters. The parsePaginationForQuery() utility function helps to achieve this by filtering the pre-determined keys sort , limit , cursor and parsing corresponding key-value pairs into ApiPagination that Prisma can consume.

In this example, the id field of the model in the database as the cursor that allows clients to continue subsequent queries with the next page.

parsePaginationForQuery in api-queries.ts

All the endpoints are defined in server.ts , particularly, there are two endpoints:

You define a list of valid query keys, such as deleted , creator , keyId , and objectId for Locked data and cancelled , swapped , recipient , and sender for Escrow data. Pass the URL query string into the pre-defined utilities to output the correct parameters that Prisma can use.

server.ts

Now that you have an indexer and an API service, you can deploy your move package and start the indexer and API service.

Install dependencies by running pnpm install --ignore-workspace or yarn install --ignore-workspace .

Setup the database by running pnpm db:setup:dev or yarn db:setup:dev .

Deploy the Sui package

Deployment instructions

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.

Use the following command to list your available environments:

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, run the following command:

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.

There are some helper functions to publish the smart contracts so you can create some demo data (for Testnet). The helper function to publish the smart contrqcts expects built smart contracts in both the escrow and demo directories. Run sui move build in both directories, if necessary. Be sure to update the Sui dependency in the manifest to point to the correct source based on your environment.

To publish the smart contracts and produce demo data:

Publish the smart contracts by running the following command from your api folder:

If successful, demo-contract.json and escrow-contract.json are created in the backend root directory. These files contain the contract addresses and are used by the backend and frontend to interact with the contracts.

Produce demo non-locked and locked objects

Produce demo escrows

If you want to reset the database (for a clean demo, for example), run pnpm db:reset:dev && pnpm db:setup:dev or yarn db:reset:dev && yarn db:setup:dev .

Run both the API and the indexer by running pnpm dev or yarn dev .

Visit http://localhost:3000/escrows or http://localhost:3000/locked

You should now have an indexer running.

With the code successfully deployed on Testnet, you can now create a frontend to display the trading data and to allow users to interact with the Move modules.

In this final part of the app example, you build a frontend (UI) that allows end users to discover trades and interact with listed escrows.

You can view the complete source code for this app example in the Sui repository.

Before getting started, make sure you have:

The UI design consists of three parts:

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

When asked for a name for your dApp, provide one of your liking. The dApp scaffold gets created in a new directory with the name you provide. This is convenient to keep your working code separate from the example source code that might already populate this folder. The codeblocks that follow point to the code in the default example location. Be aware the path to your own code includes the dApp name you provide.

First, set up import aliases to make the code more readable and maintainable. This allows you to import files using @/ instead of relative paths.

Replace the content of tsconfig.json with the following:

The paths option under compilerOptions is what defines the aliasing for TypeScript. Here, the alias @/ *is mapped to the ./src/* directory, meaning that any time you use @/ , TypeScript resolves it as a reference to the src folder. This setup reduces the need for lengthy relative paths when importing files in your project.

Replace the content of vite.config.ts with the following:

Vite also needs to be aware of the aliasing to resolve imports correctly during the build process. In the resolve.alias configuration of vite.config.ts , we map the alias @ to the /src directory.

To streamline the styling process and keep the codebase clean and maintainable, this guide uses Tailwind CSS, which provides utility-first CSS classes to rapidly build custom designs. Run the following command from the base of your dApp project to add Tailwind CSS and its dependencies:

Next, generate the Tailwind CSS configuration file by running the following:

Replace the content of tailwind.config.js with the following:

Add the src/styles/ directory and add base.css :

First, deploy your package via the scripts in the api directory .

Then, create a src/constants.ts file and fill it with the following:

If you create a dApp using a project name so that your src files are in a subfolder of frontend , be sure to add another nesting level (../) to the import statements.

Create a src/utils/ directory and add the following file:

Create a src/components/ directory and add the following components:

ExplorerLink.tsx

InfiniteScrollArea.tsx

Loading.tsx

SuiObjectDisplay.tsx

Install the necessary dependencies:

The imported template only has a single page. To add more pages, you need to set up routing.

First, install the necessary dependencies:

Then, create a src/routes/ directory and add index.tsx . This file contains the routing configuration:

Add the following respective files to the src/routes/ directory:

root.tsx . This file contains the root component that is rendered on every page:

LockedDashboard.tsx . This file contains the component for the Manage Objects page.

EscrowDashboard.tsx . This file contains the component for the Escrows page.

Update src/main.tsx by replacing the App component with the RouterProvider and replace "dark" with "light" in the Theme component:

Note that dApp Kit provides a set of hooks for making query and mutation calls to the Sui blockchain. These hooks are thin wrappers around query and mutation hooks from @tanstack/react-query .

Create src/components/Header.tsx . This file contains the navigation links and the connect wallet button:

The dApp Kit comes with a pre-built React.js component called ConnectButton displaying a button to connect and disconnect a wallet. The connecting and disconnecting wallet logic is handled seamlessly so you don't need to worry about repeating yourself doing the same logic all over again.

At this point, you have a basic routing setup. Run your app and ensure you can:

Note, the styles should be applied. The Header component should look like this:

All the type definitions are in src/types/types.ts . Create this file and add the following:

ApiLockedObject and ApiEscrowObject represent the Locked and Escrow indexed data model the indexing and API service return.

EscrowListingQuery and LockedListingQuery are the query parameters model to provide to the API service to fetch from the endpoints /escrow and /locked accordingly.

Now, display the objects owned by the connected wallet address. This is the Manage Objects page.

First add this file src/components/locked/LockOwnedObjects.tsx :

Fetch the owned objects directly from Sui blockchain using the useSuiClientInfiniteQuery() hook from dApp Kit . This hook is a thin wrapper around Sui blockchain RPC calls, reference the documentation to learn more about these RPC hooks . Basically, supply the RPC endpoint you want to execute, in this case it's the getOwnedObjects endpoint . Supply the connected wallet account as the owner . The returned data is stored inside the cache at query key getOwnedObjects . In a future step you invalidate this cache after a mutation succeeds, so the data will be re-fetched automatically.

Next, update src/routes/LockedDashboard.tsx to include the LockOwnedObjects component:

Run your app and ensure you can:

If you don't see any objects, you might need to create some demo data or connect your wallet. You can mint objects after completing the next steps.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's create and examine the execute transaction hook.

Create src/hooks/useTransactionExecution.ts :

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useSuiClient() hook from dApp Kit to retrieve the Sui client instance configured in src/main.tsx . The useSignTransaction() function is another hook from dApp kit that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, use the executeTransaction() on the Sui client instance of the Sui TypeScript SDK.

The full source code of the demo bear smart contract is available at Trading Contracts Demo directory

You need a utility function to create a dummy object representing a real world asset so you can use it to test and demonstrate escrow users flow on the UI directly.

Create src/mutations/demo.ts :

As previously mentioned, this example uses @tanstack/react-query to query, cache, and mutate server state. Server state is data only available on remote servers, and the only way to retrieve or update this data is by interacting with these remote servers. In this case, it could be from an API or directly from Sui blockchain RPC.

When you execute a transaction call to mutate data on the Sui blockchain, use the useMutation() hook. The useMutation() hook accepts several inputs, however, you only need two of them for this example. The first parameter, mutationFn , accepts the function to execute the main mutating logic, while the second parameter, onSuccess , is a callback that runs when the mutating logic succeeds.

The main mutating logic includes executing a Move call of a package named demo_bear::new to create a dummy bear object and transferring it to the connected wallet account, all within the same Transaction . This example reuses the executeTransaction() hook from the Execute Transaction Hook step to execute the transaction.

Another benefit of wrapping the main mutating logic inside useMutation() is that you can access and manipulate the cache storing server state. This example fetches the cache from remote servers by using query call in an appropriate callback. In this case, it is the onSuccess callback. When the transaction succeeds, invalidate the cache data at the cache key called getOwnedObjects , then @tanstack/react-query handles the re-fetching mechanism for the invalidated data automatically. Do this by using invalidateQueries() on the @tanstack/react-query configured client instance retrieved by useQueryClient() hook in the Set up Routing step.

Now the logic to create a dummy bear object exists. You just need to attach it into the button in the header.

Header.tsx

Run your app and ensure you can:

To lock the object, execute the lock Move function identified by {PACKAGE_ID}::lock::lock . The implementation is similar to what's in previous mutation functions, use useMutation() from @tanstack/react-query to wrap the main logic inside it. The lock function requires an object to be locked and its type because our smart contract lock function is generic and requires type parameters. After creating a Locked object and its Key object, transfer them to the connected wallet account within the same transaction block.

It's beneficial to extract logic of locking owned objects into a separated mutating function to enhance discoverability and encapsulation.

Create src/mutations/locked.ts :

Update src/components/locked/LockOwnedObjects.tsx to include the useLockObjectMutation hook:

LockOwnedObjects.tsx

Run your app and ensure you can:

The object should disappear from the list of owned objects. You view and unlock locked objects in later steps.

Let's take a look at the My Locked Objects tab by examining src/components/locked/OwnedLockedList.tsx . Focus on the logic on how to retrieve this list.

OwnedLockedList.tsx

This instance of useSuiClientInfiniteQuery() is similar to the one in the LockOwnedObjects component. The difference is that it fetches the locked objects instead of the owned objects. The Locked object is a struct type in the smart contract, so you need to supply the struct type to the query call as a filter . The struct type is usually identified by the format of {PACKAGE_ID}::{MODULE_NAME}::{STRUCT_TYPE} .

The (src/components/locked/LockedObject.tsx) component is mainly responsible for mapping an on-chain SuiObjectData Locked object to its corresponding ApiLockedObject , which is finally delegated to the component for rendering. The fetches the locked item object ID if the prop itemId is not supplied by using dApp Kit useSuiClientQuery() hook to call the getDynamicFieldObject RPC endpoint. Recalling that in this smart contract, the locked item is put into a dynamic object field.

LockedObject.tsx

The (src/components/locked/partials/Locked.tsx) component is mainly responsible for rendering the ApiLockedObject . Later on, it will also consist of several on-chain interactions: unlock the locked objects and create an escrow out of the locked object.

Locked.tsx

Update src/routes/LockedDashboard.tsx to include the OwnedLockedList component:

LockedDashboard.tsx

Run your app and ensure you can:

To unlock the object, execute the unlock Move function identified by {PACKAGE_ID}::lock::unlock . Call the unlock function supplying the Locked object, its corresponding Key , the struct type of the original object, and transfer the unlocked object to the current connected wallet account. Also, implement the onSuccess callback to invalidate the cache data at query key locked after one second to force react-query to re-fetch the data at corresponding query key automatically.

Unlocking owned objects is another crucial and complex on-chain action in this application. Hence, it's beneficial to extract its logic into separated mutating functions to enhance discoverability and encapsulation.

src/mutations/locked.ts

Update src/components/locked/partials/Locked.tsx to include the useUnlockObjectMutation hook:

Locked.tsx

Run your app and ensure you can:

Update src/routes/EscrowDashboard.tsx to include the LockedList component:

EscrowDashboard.tsx

Add src/components/locked/ApiLockedList.tsx :

ApiLockedList.tsx

This hook fetches all the non-deleted system Locked objects from the API in a paginated fashion. Then, it proceeds into fetching the on-chain state, to ensure the latest state of the object is displayed.

This component uses tanstack's useInfiniteQuery instead of useSuiClientInfiniteQuery since the data is being fetched from the example's API rather than Sui.

Add src/hooks/useGetLockedObject.ts

Run your app and ensure you can:

To create escrows, include a mutating function through the useCreateEscrowMutation hook in src/mutations/escrow.ts . It accepts the escrowed item to be traded and the ApiLockedObject from another party as parameters. Then, call the {PACKAGE_ID}::shared::create Move function and provide the escrowed item, the key id of the locked object to exchange, and the recipient of the escrow (locked object's owner).

escrow.ts

Update src/components/locked/partials/Locked.tsx to include the useCreateEscrowMutation hook

Add src/components/escrows/CreateEscrow.tsx

Run your app and ensure you can:

The object should disappear from the list of locked objects in the Browse Locked Objects tab in the Escrows page. You view and accept or cancel escrows in later steps.

To cancel the escrow, create a mutation through the useCancelEscrowMutation hook in src/mutations/escrow.ts . The cancel function accepts the escrow ApiEscrowObject and its on-chain data. The {PACKAGE_ID}::shared::return_to_sender Move call is generic, thus it requires the type parameters of the escrowed object. Next, execute {PACKAGE_ID}::shared::return_to_sender and transfer the returned escrowed object to the creator of the escrow.

escrow.ts

Add src/components/escrows/Escrow.tsx

Add src/components/escrows/EscrowList.tsx

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

To accept the escrow, create a mutation through the useAcceptEscrowMutation hook in src/mutations/escrow.ts . The implementation should be fairly familiar to you now. The accept function accepts the escrow ApiEscrowObject and the locked object ApiLockedObject . The {PACKAGE_ID}::shared::swap Move call is generic, thus it requires the type parameters of the escrowed and locked objects. Query the objects details by using multiGetObjects on Sui client instance. Lastly, execute the {PACKAGE_ID}::shared::swap Move call and transfer the returned escrowed item to the connected wallet account. When the mutation succeeds, invalidate the cache to allow automatic re-fetch of the data.

escrow.ts

Update src/components/escrows/Escrow.tsx to include the useAcceptEscrowMutation hook

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

At this point, you have a fully functional frontend that allows users to discover trades and interact with listed escrows. The UI is designed to be user-friendly and intuitive, allowing users to easily navigate and interact with the application. Have fun exploring the app and testing out the different features!

Frontend

In this final part of the app example, you build a frontend (UI) that allows end users to discover trades and interact with listed escrows.

You can view the complete source code for this app example in the Sui repository.

Before getting started, make sure you have:

The UI design consists of three parts:

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

When asked for a name for your dApp, provide one of your liking. The dApp scaffold gets created in a new directory with the name you provide. This is convenient to keep your working code separate from the example source code that might already populate this folder. The codeblocks that follow point to the code in the default example location. Be aware the path to your own code includes the dApp name you provide.

First, set up import aliases to make the code more readable and maintainable. This allows you to import files using @/ instead of relative paths.

Replace the content of tsconfig.json with the following:

The paths option under compilerOptions is what defines the aliasing for TypeScript. Here, the alias @/ *is mapped to the ./src/* directory, meaning that any time you use @/ , TypeScript resolves it as a reference to the src folder. This setup reduces the need for lengthy relative paths when importing files in your project.

Replace the content of vite.config.ts with the following:

Vite also needs to be aware of the aliasing to resolve imports correctly during the build process. In the resolve.alias configuration of vite.config.ts , we map the alias @ to the /src directory.

To streamline the styling process and keep the codebase clean and maintainable, this guide uses Tailwind CSS, which provides utility-first CSS classes to rapidly build custom designs. Run the following command from the base of your dApp project to add Tailwind CSS and its dependencies:

Next, generate the Tailwind CSS configuration file by running the following:

Replace the content of tailwind.config.js with the following:

Add the src/styles/ directory and add base.css :

First, deploy your package via the scripts in the api directory .

Then, create a src/constants.ts file and fill it with the following:

If you create a dApp using a project name so that your src files are in a subfolder of frontend , be sure to add another nesting level (../) to the import statements.

Create a src/utils/ directory and add the following file:

Create a src/components/ directory and add the following components:

ExplorerLink.tsx

InfiniteScrollArea.tsx

Loading.tsx

SuiObjectDisplay.tsx

Install the necessary dependencies:

The imported template only has a single page. To add more pages, you need to set up routing.

First, install the necessary dependencies:

Then, create a src/routes/ directory and add index.tsx . This file contains the routing configuration:

Add the following respective files to the src/routes/ directory:

root.tsx . This file contains the root component that is rendered on every page:

LockedDashboard.tsx . This file contains the component for the Manage Objects page.

EscrowDashboard.tsx . This file contains the component for the Escrows page.

Update src/main.tsx by replacing the App component with the RouterProvider and replace "dark" with "light" in the Theme component:

Note that dApp Kit provides a set of hooks for making query and mutation calls to the Sui blockchain. These hooks are thin wrappers around query and mutation hooks from @tanstack/react-query .

Create src/components/Header.tsx . This file contains the navigation links and the connect wallet button:

The dApp Kit comes with a pre-built React.js component called ConnectButton displaying a button to connect and disconnect a wallet. The connecting and disconnecting wallet logic is handled seamlessly so you don't need to worry about repeating yourself doing the same logic all over again.

At this point, you have a basic routing setup. Run your app and ensure you can:

Note, the styles should be applied. The Header component should look like this:

All the type definitions are in src/types/types.ts . Create this file and add the following:

ApiLockedObject and ApiEscrowObject represent the Locked and Escrow indexed data model the indexing and API service return.

EscrowListingQuery and LockedListingQuery are the query parameters model to provide to the API service to fetch from the endpoints /escrow and /locked accordingly.

Now, display the objects owned by the connected wallet address. This is the Manage Objects page.

First add this file src/components/locked/LockOwnedObjects.tsx :

Fetch the owned objects directly from Sui blockchain using the useSuiClientInfiniteQuery() hook from dApp Kit . This hook is a thin wrapper around Sui blockchain RPC calls, reference the documentation to learn more about these RPC hooks . Basically, supply the RPC endpoint you want to execute, in this case it's the getOwnedObjects endpoint . Supply the connected wallet account as the owner . The returned data is stored inside the cache at query key getOwnedObjects . In a future step you invalidate this cache after a mutation succeeds, so the data will be re-fetched automatically.

Next, update src/routes/LockedDashboard.tsx to include the LockOwnedObjects component:

Run your app and ensure you can:

If you don't see any objects, you might need to create some demo data or connect your wallet. You can mint objects after completing the next steps.

In the frontend, you might need to execute a transaction block in multiple places, hence it's better to extract the transaction execution logic and reuse it everywhere. Let's create and examine the execute transaction hook.

Create src/hooks/useTransactionExecution.ts :

A Transaction is the input, sign it with the current connected wallet account, execute the transaction block, return the execution result, and finally display a basic toast message to indicate whether the transaction is successful or not.

Use the useSuiClient() hook from dApp Kit to retrieve the Sui client instance configured in src/main.tsx . The useSignTransaction() function is another hook from dApp kit that helps to sign the transaction block using the currently connected wallet. It displays the UI for users to review and sign their transactions with their selected wallet. To execute a transaction block, use the executeTransaction() on the Sui client instance of the Sui TypeScript SDK.

The full source code of the demo bear smart contract is available at Trading Contracts Demo directory

You need a utility function to create a dummy object representing a real world asset so you can use it to test and demonstrate escrow users flow on the UI directly.

Create src/mutations/demo.ts :

As previously mentioned, this example uses @tanstack/react-query to query, cache, and mutate server state. Server state is data only available on remote servers, and the only way to retrieve or update this data is by interacting with these remote servers. In this case, it could be from an API or directly from Sui blockchain RPC.

When you execute a transaction call to mutate data on the Sui blockchain, use the useMutation() hook. The useMutation() hook accepts several inputs, however, you only need two of them for this example. The first parameter, mutationFn , accepts the function to execute the main mutating logic, while the second parameter, onSuccess , is a callback that runs when the mutating logic succeeds.

The main mutating logic includes executing a Move call of a package named demo_bear::new to create a dummy bear object and transferring it to the connected wallet account, all within the same Transaction . This example reuses the executeTransaction() hook from the Execute Transaction Hook step to execute the transaction.

Another benefit of wrapping the main mutating logic inside useMutation() is that you can access and manipulate the cache storing server state. This example fetches the cache from remote servers by using query call in an appropriate callback. In this case, it is the onSuccess callback. When the transaction succeeds, invalidate the cache data at the cache key called getOwnedObjects , then @tanstack/react-query handles the re-fetching mechanism for the invalidated data automatically. Do this by using invalidateQueries() on the @tanstack/react-query configured client instance retrieved by useQueryClient() hook in the Set up Routing step.

Now the logic to create a dummy bear object exists. You just need to attach it into the button in the header.

Header.tsx

Run your app and ensure you can:

To lock the object, execute the lock Move function identified by {PACKAGE_ID}::lock::lock . The implementation is similar to what's in previous mutation functions, use useMutation() from @tanstack/react-query to wrap the main logic inside it. The lock function requires an object to be locked and its type because our smart contract lock function is generic and requires type parameters. After creating a Locked object and its Key object, transfer them to the connected wallet account within the same transaction block.

It's beneficial to extract logic of locking owned objects into a separated mutating function to enhance discoverability and encapsulation.

Create src/mutations/locked.ts :

Update src/components/locked/LockOwnedObjects.tsx to include the useLockObjectMutation hook:

LockOwnedObjects.tsx

Run your app and ensure you can:

The object should disappear from the list of owned objects. You view and unlock locked objects in later steps.

Let's take a look at the My Locked Objects tab by examining src/components/locked/OwnedLockedList.tsx . Focus on the logic on how to retrieve this list.

OwnedLockedList.tsx

This instance of useSuiClientInfiniteQuery() is similar to the one in the LockOwnedObjects component. The difference is that it fetches the locked objects instead of the owned objects. The Locked object is a struct type in the smart contract, so you need to supply the struct type to the query call as a filter . The struct type is usually identified by the format of {PACKAGE_ID}::{MODULE_NAME}::{STRUCT_TYPE} .

The (src/components/locked/LockedObject.tsx) component is mainly responsible for mapping an on-chain SuiObjectData Locked object to its corresponding ApiLockedObject , which is finally delegated to the component for rendering. The fetches the locked item object ID if the prop itemId is not supplied by using dApp Kit useSuiClientQuery() hook to call the getDynamicFieldObject RPC endpoint. Recalling that in this smart contract, the locked item is put into a dynamic object field.

LockedObject.tsx

The (src/components/locked/partials/Locked.tsx) component is mainly responsible for rendering the ApiLockedObject . Later on, it will also consist of several on-chain interactions: unlock the locked objects and create an escrow out of the locked object.

Locked.tsx

Update src/routes/LockedDashboard.tsx to include the OwnedLockedList component:

LockedDashboard.tsx

Run your app and ensure you can:

To unlock the object, execute the unlock Move function identified by {PACKAGE_ID}::lock::unlock . Call the unlock function supplying the Locked object, its corresponding Key , the struct type of the original object, and transfer the unlocked object to the current connected wallet account. Also, implement the onSuccess callback to invalidate the cache data at query key locked after one second to force react-query to re-fetch the data at corresponding query key automatically.

Unlocking owned objects is another crucial and complex on-chain action in this application. Hence, it's beneficial to extract its logic into separated mutating functions to enhance discoverability and encapsulation.

src/mutations/locked.ts

Update src/components/locked/partials/Locked.tsx to include the useUnlockObjectMutation hook:

Locked.tsx

Run your app and ensure you can:

Update src/routes/EscrowDashboard.tsx to include the LockedList component:

EscrowDashboard.tsx

Add src/components/locked/ApiLockedList.tsx :

ApiLockedList.tsx

This hook fetches all the non-deleted system Locked objects from the API in a paginated fashion. Then, it proceeds into fetching the on-chain state, to ensure the latest state of the object is displayed.

This component uses tanstack's useInfiniteQuery instead of useSuiClientInfiniteQuery since the data is being fetched from the example's API rather than Sui.

Add src/hooks/useGetLockedObject.ts

Run your app and ensure you can:

To create escrows, include a mutating function through the useCreateEscrowMutation hook in src/mutations/escrow.ts . It accepts the escrowed item to be traded and the ApiLockedObject from another party as parameters. Then, call the {PACKAGE_ID}::shared::create Move function and provide the escrowed item, the key id of the locked object to exchange, and the recipient of the escrow (locked object's owner).

escrow.ts

Update src/components/locked/partials/Locked.tsx to include the useCreateEscrowMutation hook

Add src/components/escrows/CreateEscrow.tsx

Run your app and ensure you can:

The object should disappear from the list of locked objects in the Browse Locked Objects tab in the Escrows page. You view and accept or cancel escrows in later steps.

To cancel the escrow, create a mutation through the useCancelEscrowMutation hook in src/mutations/escrow.ts . The cancel function accepts the escrow ApiEscrowObject and its on-chain data. The {PACKAGE_ID}::shared::return_to_sender Move call is generic, thus it requires the type parameters of the escrowed object. Next, execute {PACKAGE_ID}::shared::return_to_sender and transfer the returned escrowed object to the creator of the escrow.

escrow.ts

Add src/components/escrows/Escrow.tsx

Add src/components/escrows/EscrowList.tsx

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

To accept the escrow, create a mutation through the useAcceptEscrowMutation hook in src/mutations/escrow.ts . The implementation should be fairly familiar to you now. The accept function accepts the escrow ApiEscrowObject and the locked object ApiLockedObject . The {PACKAGE_ID}::shared::swap Move call is generic, thus it requires the type parameters of the escrowed and locked objects. Query the objects details by using multiGetObjects on Sui client instance. Lastly, execute the {PACKAGE_ID}::shared::swap Move call and transfer the returned escrowed item to the connected wallet account. When the mutation succeeds, invalidate the cache to allow automatic re-fetch of the data.

escrow.ts

Update src/components/escrows/Escrow.tsx to include the useAcceptEscrowMutation hook

Update src/routes/EscrowDashboard.tsx to include the EscrowList component

Run your app and ensure you can:

At this point, you have a fully functional frontend that allows users to discover trades and interact with listed escrows. The UI is designed to be user-friendly and intuitive, allowing users to easily navigate and interact with the application. Have fun exploring the app and testing out the different features!