Asset Tokenization

Asset tokenization refers to the process of representing real-world assets, such as real estate, art, commodities, stocks, or other valuable assets, as digital tokens on the blockchain network. This involves converting the ownership or rights of an asset into digital tokens, which are then recorded and managed on the blockchain.

The concept is to divide high-value assets into smaller, more affordable units, representing ownership or a fraction of the asset.

This strategy enables wider participation from investors who might want to mitigate risk by investing in a portion of a digital asset rather than being the sole owner, thereby expanding accessibility to a broader range of investors.

This pattern is similar to the ERC1155 multi-token standard with additional functionality. This makes it a suitable choice for Solidity based use cases that one might want to implement on Sui.

Asset creation

Each asset is fractionalized into a total supply, with each fraction represented as either a non-fungible token (NFT) or fungible token (FT) type collectible. This ensures that each individual fraction maintains a balance equal to or greater than one, and when combined, all fractions collectively reach the total supply of the asset.

Besides the total supply, each asset is defined by various other fields such as name, description, and more. These fields collectively form the metadata for the asset, and they remain consistent across all fractions of the asset.

NFTs vs FTs distinction

Each time a tokenized asset is minted, there's a possibility for it to be created with new metadata. If new metadata is incorporated, the tokenized asset is deemed unique, transforming it into an NFT. In this case, its balance is limited to one, signifying that only a single instance of this asset exists.

If there's no additional metadata, the tokenized asset is categorized as an FT, allowing its balance to exceed one, enabling multiple identical instances of the asset to exist.

FTs possess the capability to merge (join) among themselves or be split when the balance is greater than one. This functionality allows for the aggregation or division of units of the token, offering flexibility in handling varying quantities as needed.

As previously mentioned, all the collectibles of tokenized assets, whether NFTs or FTs, when combined, can amount to the maximum total supply of the asset.

Burnability

When you create the asset, you can define whether the fractions of the asset are eligible for removal or destruction from circulation. The process of removing or destroying assets is called burning.

If a tokenized asset is burnable, then burning a fraction causes the circulating supply to decrease by the balance of the burnt item. The total supply, however, remains constant, allowing you to mint the burned fractions again if needed, thus maintaining the predetermined total supply of the asset.

As with all smart contracts on Sui, Move provides the logic that powers asset tokenization.

This reference implementation uses the Kiosk standard to ensure that tokenized assets operate within their defined policy. Use the implementation as presented to have marketable tokenized assets that support rules like royalties, commissions, and so on.

If using Kiosk is not a requirement, then you can exclude the unlock module and some of the proxy's methods related to transfer policies.

Select a module to view its details:

The tokenized_asset module operates in a manner similar to the coin library.

When it receives a new one-time witness type, it creates a unique representation of a fractional asset. This module employs similar implementations to some methods found in the Coin module. It encompasses functionalities pertinent to asset tokenization, including new asset creation, minting, splitting, joining, and burning. See One Time Witness in The Move Book for more information.

Structs

AssetCap

Generate an AssetCap for each new asset represented as a fractional NFT. In most scenarios, you should create it as an owned object, which you can then transfer to the platform's administrator for access-restricted method invocation.

AssetMetadata

The AssetMetadata struct defines the metadata representing the entire asset to fractionalize. This should be a shared object.

TokenizedAsset

The TokenizedAsset is minted with a specified balance that is less than or equal to the remaining supply. If the VecMap of an asset is populated with values, indicating multiple unique entries, it is considered an NFT. Conversely, if the VecMap of an asset is not populated, indicating an absence of individual entries, it is considered an FT.

PlatformCap

The PlatformCap refers to the capability issued to the individual who deploys the contract. This capability grants specific permissions or authority related to the platform's functionalities, allowing the deployer certain controlled actions or access rights within the deployed contract.

Functions

init

This function creates a PlatformCap and sends it to the sender.

new_asset

This function holds the responsibility of creating a fresh representation of an asset, defining its crucial attributes. Upon execution, it returns two distinct objects: the AssetCap and AssetMetadata . These objects encapsulate the necessary information and characteristics defining the asset within the system.

mint

The function performs the minting of a tokenized asset. If new metadata is introduced during this process, the resulting tokenized asset is considered unique, resulting in the creation of an NFT with a balance set to 1. Alternatively, if no new metadata is added, the tokenized asset is classified as an FT, permitting its balance to surpass 1, as specified by a provided argument. Upon execution, the function returns the tokenized asset object.

split

This function is provided with a tokenized asset of the FT type and a balance greater than 1, along with a value less than the object's balance, and performs a split operation on the tokenized asset. The operation divides the existing tokenized asset into two separate tokenized assets. The newly created tokenized asset has a balance equal to the given value, while the balance of the provided object is reduced by the specified value. Upon completion, the function returns the newly created tokenized asset. This function does not accept or operate on tokenized assets of the NFT type.

join

This function is given two tokenized assets of the FT type and executes a merge operation on the tokenized assets. The operation involves increasing the balance of the first tokenized asset by the balance of the second one. Subsequently, the second tokenized asset is burned or removed from circulation. After the process concludes, the function returns the ID of the burned tokenized asset.

This function does not accept or operate on tokenized assets of the NFT type.

burn

This function requires the assetCap as a parameter, thereby restricting its invocation solely to the platform admin. Additionally, it accepts a tokenized asset that is burned as part of its operation. Upon burning the provided tokenized asset, the circulating supply decreases by the balance of the burnt item. It necessitates a tokenized asset that is burnable.

total_supply

This function retrieves and returns the value representing the total supply of the asset.

supply

This function retrieves and returns the value representing the current circulating supply of the asset.

value

This function takes a tokenized asset as input and retrieves its associated balance value.

create_vec_map_from_arrays

This internal helper function populates a VecMap . It assists in the process of filling or setting key-value pairs within the VecMap data structure.

The proxy module comprises methods that the type owner utilizes to execute publisher-related operations.

Structs

Proxy

The PROXY struct represents the one-time witness (OTW) to claim the publisher.

Registry

This shared object serves as a repository for the Publisher object, specifically intended to control and restrict access to the creation and management of transfer policies for tokenized assets. Mutable access to this object is exclusively granted to the actual publisher.

ProtectedTP

This is a shared object that stores an empty transfer policy. It is required to create one per type generated by a user. Its involvement is apparent in the unlock module.

Functions

init

This function is responsible for creating the Publisher object, encapsulating it within the registry, and subsequently sharing the Registry object.

setup_tp

This function leverages the publisher nested within the registry and the sender's publisher. It generates and returns a transfer policy and the associated transfer policy cap specific to the TokenizedAsset . This type 'T' is derived from the Publisher object.

It also generates an empty transfer policy wrapped in a ProtectedTP object, which is shared. You can use this functionality under specific conditions to override the kiosk lock rule.

new_display

This function utilizes the publisher nested within the registry and the sender's publisher to generate and return an empty Display for the type TokenizedAsset , where T is encapsulated within the Publisher object.

transfer_policy

This function, provided with the protectedTP , returns the transfer policy specifically designed for the type TokenizedAsset

publisher_mut

This function can only be accessed by the owner of the platform cap. It requires the registry as an argument to obtain a mutable reference to the publisher.

The unlock module facilitates the unlocking of a tokenized asset specifically for authorized burning and joining.

It allows tokenized asset type creators to enable these operations for kiosk assets without necessitating adherence to the default set of requirements, such as rules or policies.

Structs

JoinPromise

A promise object is established to prevent attempts of permanently unlocking an object beyond the intended scope of joining.

BurnPromise

A promise object created to ensure the permanent burning of a specified object.

Functions

asset_from_kiosk_to_join

This helper function is intended to facilitate the joining of tokenized assets locked in kiosk. It aids in unlocking the tokenized asset that is set for burning and ensures that another tokenized asset of the same type will eventually contain its balance by returning a JoinPromise.

prove_join

A function utilized to demonstrate that the unlocked tokenized asset is successfully burned and its balance is incorporated into an existing tokenized asset.

asset_from_kiosk_to_burn

Helper function that facilitates the burning of tokenized assets locked in a kiosk. It assists in their unlocking while ensuring a promise that the circulating supply will be reduced, achieved by returning a BurnPromise .

prove_burn

Ensures that the circulating supply of the asset cap is reduced by the balance of the burned tokenized asset.

An example use case package that enables utilization of Rust WASM functionality to support seamless asset creation on the browser.

This is similar to the launchpad approach and serves as the template package whenever a new asset requires representation as a tokenized asset.

Effectively allowing users to edit fields of this template contract on the fly and publish it with the edits included. This package implements two essential modules, each catering to distinct functionalities required for asset tokenization. More details regarding how Rust WASM was implemented can be found in the Web Assembly section.

Modules

template

This is the module that supports defining a new asset.

When you need to represent a new asset as a fractional asset, modify this module to :: , with the (in capitals) being the OTW of this new asset.

This module calls the asset_tokenization::tokenized_asset::new_asset(...) method, which facilitates the declaration of new fields for the asset:

genesis

A genesis type of module that includes a OTW so that the sender can claim the publisher.

The following sequence diagram presenting how the join flow would take place. The following flow assumes that:

The following sequence diagram shows the burn flow and assumes that:

The packages and modules provided demonstrate how you could implement asset tokenization for your project. Your particular use case probably necessitates altering the contract for convenience or to introduce new features.

Instead of implementing the unlock functionality in multiple steps inside of a PTB, it would also be possible to create a method that performs the purchase, borrowing, unlocking and joining of an asset all on one function. This is how that would look like for the joining operation:

The following example splits (effectively replacing) the AssetCap into two new objects: the Treasury and the AdminCap . The access to methods defined in the original package, should now be carefully re-designed as this change can introduce unwanted effects. This required re-design is not entirely contained in this example and only some methods are changed for demonstration purposes (or as a thorough exercise).

Assume you want to allow the users to also burn assets, not only admins. This still needs to be an authorized operation but it would allow the flexibility of consuming tokenized assets for a use case specific purpose (for example, burning all of the collectibles you've gathered to combine them). To achieve this, the admin can mint tickets that contain the ID of the asset they are allowed to burn. To support this functionality you must redesign the smart contract and separate the admin from the asset's treasury of each asset, which now holds only supply related information. Sample changes that need to happen follow:

Structs

Create a ticket that has only the key ability so that the receiver cannot trade it.

The struct that now only holds treasury related information (results from splitting the AssetCap , meaning it's no longer part of this design) is created as a shared object. Change functions like mint to also take as input both the Treasury object and the AdminCap object.

The other half of the AssetCap functionality which retains the admin capability and the configuration of burnability is an owned object sent to the creator of type .

Method Signatures

The AdminCap here acts both as an admin capability and a type insurance. Encoding the information of both the asset type that is allowed to be deleted with this ticket. This function should assert that the asset T is burnable and return a BurnTicket .

Burning on the user side requires for them to access the shared Treasury object. This function burns the tokenized asset and decreases the supply.

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.

At this stage, you can choose to manually deploy the contracts or utilize the publish bash script that automatically deploys the contracts and sets up most of the .env Asset Tokenization related fields for you. The .env.template file denotes variables that the script automatically fills in. You can see a reference here:

For more details on publishing, please check the setup folder's README .

Select a package for specific instructions.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

In a terminal or console at the move/asset_tokenization directory of the project enter:

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

The package should successfully deploy, and you then see:

You can also view a multitude of information and transactional effects.

You should choose and store the package ID and the registry ID from the created objects in the respective fields within your .env file.

Afterward, it's necessary to modify the Move.toml file. Under the [addresses] section, replace 0x0 with the same package ID . Optionally, under the [package] section, add published-at = (this step is not needed if you see a Move.lock file after running sui

client publish).

The fields that are automatically filled are: SUI_NETWORK , ASSET_TOKENIZATION_PACKAGE_ID and REGISTRY .

To publish with the bash script run:

After publishing, you can now edit the Move.toml file like described in the Manual flow.

For more details regarding this process, please consult the setup folder's README .

In a terminal or console at the move/template directory of the project enter:

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

The package should successfully deploy, and you then see:

You can also view a multitude of information and transactional effects.

You should choose and store the package ID , asset metadata ID , asset cap ID and the Publisher ID from the created objects in the respective fields within your .env file.

The process of automatic deployment for the template package refers to publishing a new asset via the WASM library. Quick start steps:

For more details regarding this process, please consult the setup folder's README .

You can find a public facing reference to the WASM library in the move-binary-format-wasm Sui repo subfolder.

This feature was developed with the intent to enable Move bytecode serialization and deserialization on the web. In essence, this feature allows you to edit existing contracts in a web environment.

In the case of asset tokenization, these edits allow you to create and publish new types that represent physical or digital assets that we want to tokenize.

On modifications that are made to the template package this process needs to be repeated. Note that some alterations, like changing a constant name, do not affect the produced bytecode.

Before proceeding to how to make these edits, it's important to understand how the library exposes the template module bytecode. The process is currently manual. This requires that you build and retrieve the compiled bytecode. To do this, navigate inside the template folder and run the following command:

Console response

The response you should receive looks similar to the following:

Copy the output you receive and paste it in the return instruction of the getBytecode method, which is located inside the bytecode-template.ts file.

Additionally, because the template package contains two modules, and therefore has another dependency, you also need to retrieve the bytecode of the genesis module in a similar fashion. This module bytecode, however, is not edited and isn't used as is. This operation is not directly relevant to the WASM library, but is necessary to successfully deploy the edited template module. To acquire the bytecode for genesis, navigate to the template folder and run:

The output format is similar to the template module but smaller in length. Similarly to what you did with the template module, you need to copy this output but this time paste it in the bytecode constant variable located in the genesis_bytecode.ts file.

With the above setup, the library can now manipulate the bytecode by deserializing it, editing it, and serializing it again so that you can publish it.

Taking a look at the template module, you should see that a few constants have been defined:

These constants act as a reference point that the WASM library is able to modify. If you take a look at the TypeScript code that performs the edit and deploys, you can see in action how these fields are identified and updated:

Examine the updateConstant method, which is used to update constants. This method takes four arguments:

The last two arguments are required to minimize the risk of accidentally updating the wrong constant since this library is directly manipulating complied bytecode, which is quite dangerous.

Additionally, the changeIdentifiers method updates identifiers, which in our case are the module name and the struct name. This method takes a JSON object as an argument with keys of the current identifier names in the module and values being the desired names you want to change them into.

Lastly, to deploy the changed template module, build and publish:

As mentioned in the Bytecode manipulation section, the modules that you need to publish are the template and the genesis, hence the reason you have two elements in the modules array. It's also important to include any dependencies defined in the Move.toml file of the involved packages. The packageId used previously is the address the asset_tokenization package has been deployed to.

Now, you can begin interacting with the deployed smart contract and your tokenized asset.

In a terminal or console within the project's setup directory, utilize the following commands:

Create Transfer Policy

First, create a TransferPolicy and a ProtectedTP with the following command:

After executing the command, the console displays the effects of the transaction.

By searching the transaction digest on a Sui network explorer, you can locate the created objects. Subsequently, select and save the TransferPolicy ID and the ProtectedTP ID from these objects into the respective fields within your .env file.

Add Rules

In the project's file transferPolicyRules.ts located in the directory setup/src/functions , you can modify the code to include the desired rules for your transfer policy.

Code snippet to be modified:

By running the command npm run call tp-rules , the rules will be added to your transfer policy.

Now, investors can trade the fractions of your asset according to the rules you've set.

Select Kiosk

You must place the tokenized assets within a kiosk if marketable assets are desired. Subsequently, you can list and sell them to other users. It's imperative to lock the objects in the kiosk to prevent any future unauthorized usage outside the defined policy that you set.

Best practices recommend a single, comprehensive kiosk for all operations. However, this might not always be the case. Therefore, this project requires the use of only one personal kiosk to ensure consistency and better management, even if you own multiple kiosks.

To enforce this rule, execute the command npm run call select-kiosk . This provides you with the specific kiosk ID to use for this project.

Then, store the provided Kiosk ID in the appropriate field within your .env file.

Mint

In the project's file mint.ts , found in the directory setup/src/functions , you can edit the code to mint the desired type (NFT/FT) and balance for your asset.

As previously mentioned, if additional metadata is provided, the tokenized asset is treated as an NFT with a value of one. However, if there is no extra metadata, the tokenized asset is regarded as an FT, and you have the flexibility to select its balance, which can exceed one.

Here is an example from the code that needs modification:

or

Upon executing the command npm run call mint , a new tokenized asset is minted. You can save the object's ID in the .env file for future reference.

Lock

Locking the objects within the kiosk is crucial to prevent any unauthorized usage beyond the established policy.

Upon executing the command npm run call lock , your newly minted tokenized asset is secured within your kiosk.

Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to lock.

Mint and Lock

Executing the command npm run call mint-lock performs both the mint and lock functions sequentially, ensuring the minted asset is created and immediately locked within the kiosk.

List

Now that your tokenized asset is placed and locked within your kiosk, you can proceed to list it for sale.

In the project's file listItem.ts , found in the directory setup/src/functions , you can adjust the code to specify the desired asset for listing.

Code snippet to be modified:

By running the command npm run call list , your tokenized asset is listed and made available for sale.

Purchase

When a user intends to purchase an item, it needs to be listed for sale. After the user selects the item to buy, they are required to modify the following snippet of code found in the file purchaseItem.ts , located in the setup/src/functions directory.

Apart from specifying the item and its type, the buyer must set the specific price and the seller's kiosk ID to execute the purchase transaction successfully, accomplished by running npm run call purchase .

Join

When you execute the command npm run call join , two specified tokenized assets of the FT type are merged together. Before running the command, make sure that the fields FT1 and FT2 within your .env file are populated with the objects you intend to merge.

Burn

When you intend to burn a tokenized asset, execute the command npm run call burn . Following this action, the specified asset is destroyed. Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to burn.

Get Balance

By executing the command npm run call get-balance , you can retrieve the balance value associated with the specified tokenized asset.

Get Supply

By executing the command npm run call get-supply , you can retrieve the value representing the current circulating supply of the asset.

Get Total Supply

By executing the command npm run call get-total-supply , you can retrieve the value representing the current circulating supply of the asset.

High-level overview

The concept is to divide high-value assets into smaller, more affordable units, representing ownership or a fraction of the asset.

This strategy enables wider participation from investors who might want to mitigate risk by investing in a portion of a digital asset

rather than being the sole owner, thereby expanding accessibility to a broader range of investors.

This pattern is similar to the ERC1155 multi-token standard with additional functionality. This makes it a suitable choice for Solidity based use cases that one might want to implement on Sui.

Asset creation

Each asset is fractionalized into a total supply, with each fraction represented as either a non-fungible token (NFT) or fungible token (FT) type collectible. This ensures that each individual fraction maintains a balance equal to or greater than one, and when combined, all fractions collectively reach the total supply of the asset.

Besides the total supply, each asset is defined by various other fields such as name, description, and more. These fields collectively form the metadata for the asset, and they remain consistent across all fractions of the asset.

NFTs vs FTs distinction

Each time a tokenized asset is minted, there's a possibility for it to be created with new metadata. If new metadata is incorporated, the tokenized asset is deemed unique, transforming it into an NFT. In this case, its balance is limited to one, signifying that only a single instance of this asset exists.

If there's no additional metadata, the tokenized asset is categorized as an FT, allowing its balance to exceed one, enabling multiple identical instances of the asset to exist.

FTs possess the capability to merge (join) among themselves or be split when the balance is greater than one. This functionality allows for the aggregation or division of units of the token, offering flexibility in handling varying quantities as needed.

As previously mentioned, all the collectibles of tokenized assets, whether NFTs or FTs, when combined, can amount to the maximum total supply of the asset.

Burnability

When you create the asset, you can define whether the fractions of the asset are eligible for removal or destruction from circulation. The process of removing or destroying assets is called burning.

If a tokenized asset is burnable, then burning a fraction causes the circulating supply to decrease by the balance of the burnt item. The total supply, however, remains constant, allowing you to mint the burned fractions again if needed, thus maintaining the predetermined total supply of the asset.

As with all smart contracts on Sui, Move provides the logic that powers asset tokenization.

This reference implementation uses the Kiosk standard to ensure that tokenized assets operate within their defined policy. Use the implementation as presented to have marketable tokenized assets that support rules like royalties, commissions, and so on.

If using Kiosk is not a requirement, then you can exclude the unlock module and some of the proxy's methods related to transfer policies.

Select a module to view its details:

The tokenized_asset module operates in a manner similar to the coin library.

When it receives a new one-time witness type, it creates a unique representation of a fractional asset. This module employs similar implementations to some methods found in the Coin module. It encompasses functionalities pertinent to asset tokenization, including new asset creation, minting, splitting, joining, and burning. See One Time Witness in The Move Book for more information.

Structs

AssetCap

Generate an AssetCap for each new asset represented as a fractional NFT. In most scenarios, you should create it as an owned object, which you can then transfer to the platform's administrator for access-restricted method invocation.

AssetMetadata

The AssetMetadata struct defines the metadata representing the entire asset to fractionalize. This should be a shared object.

TokenizedAsset

The TokenizedAsset is minted with a specified balance that is less than or equal to the remaining supply. If the VecMap of an asset is populated with values, indicating multiple unique entries, it is considered an NFT. Conversely, if the VecMap of an asset is not populated, indicating an absence of individual entries, it is considered an FT.

PlatformCap

The PlatformCap refers to the capability issued to the individual who deploys the contract. This capability grants specific permissions or authority related to the platform's functionalities, allowing the deployer certain controlled actions or access rights within the deployed contract.

Functions

init

This function creates a PlatformCap and sends it to the sender.

new_asset

This function holds the responsibility of creating a fresh representation of an asset, defining its crucial attributes. Upon execution, it returns two distinct objects: the AssetCap and AssetMetadata . These objects encapsulate the necessary information and characteristics defining the asset within the system.

mint

The function performs the minting of a tokenized asset. If new metadata is introduced during this process, the resulting tokenized asset is considered unique, resulting in the creation of an NFT with a balance set to 1. Alternatively, if no new metadata is added, the tokenized asset is classified as an FT, permitting its balance to surpass 1, as specified by a provided argument. Upon execution, the function returns the tokenized asset object.

split

This function is provided with a tokenized asset of the FT type and a balance greater than 1, along with a value less than the object's balance, and performs a split operation on the tokenized asset. The operation divides the existing tokenized asset into two separate tokenized assets. The newly created tokenized asset has a balance equal to the given value, while the balance of the provided object is reduced by the specified value. Upon completion, the function returns the newly created tokenized asset. This function does not accept or operate on tokenized assets of the NFT type.

join

This function is given two tokenized assets of the FT type and executes a merge operation on the tokenized assets. The operation involves increasing the balance of the first tokenized asset by the balance of the second one. Subsequently, the second tokenized asset is burned or removed from circulation. After the process concludes, the function returns the ID of the burned tokenized asset.

This function does not accept or operate on tokenized assets of the NFT type.

burn

This function requires the assetCap as a parameter, thereby restricting its invocation solely to the platform admin. Additionally, it accepts a tokenized asset that is burned as part of its operation. Upon burning the provided tokenized asset, the circulating supply decreases by the balance of the burnt item. It necessitates a tokenized asset that is burnable.

total_supply

This function retrieves and returns the value representing the total supply of the asset.

supply

This function retrieves and returns the value representing the current circulating supply of the asset.

value

This function takes a tokenized asset as input and retrieves its associated balance value.

create_vec_map_from_arrays

This internal helper function populates a VecMap . It assists in the process of filling or setting key-value pairs within the VecMap data structure.

The proxy module comprises methods that the type owner utilizes to execute publisher-related operations.

Structs

Proxy

The PROXY struct represents the one-time witness (OTW) to claim the publisher.

Registry

This shared object serves as a repository for the Publisher object, specifically intended to control and restrict access to the creation and management of transfer policies for tokenized assets. Mutable access to this object is exclusively granted to the actual publisher.

ProtectedTP

This is a shared object that stores an empty transfer policy. It is required to create one per type generated by a user. Its involvement is apparent in the unlock module.

Functions

init

This function is responsible for creating the Publisher object, encapsulating it within the registry, and subsequently sharing the Registry object.

setup_tp

This function leverages the publisher nested within the registry and the sender's publisher. It generates and returns a transfer policy and the associated transfer policy cap specific to the TokenizedAsset . This type 'T' is derived from the Publisher object.

It also generates an empty transfer policy wrapped in a ProtectedTP object, which is shared. You can use this functionality under specific conditions to override the kiosk lock rule.

new_display

This function utilizes the publisher nested within the registry and the sender's publisher to generate and return an empty Display for the type TokenizedAsset , where T is encapsulated within the Publisher object.

transfer_policy

This function, provided with the protectedTP , returns the transfer policy specifically designed for the type TokenizedAsset

publisher_mut

This function can only be accessed by the owner of the platform cap. It requires the registry as an argument to obtain a mutable reference to the publisher.

The unlock module facilitates the unlocking of a tokenized asset specifically for authorized burning and joining.

It allows tokenized asset type creators to enable these operations for kiosk assets without necessitating adherence to the default set of requirements, such as rules or policies.

Structs

JoinPromise

A promise object is established to prevent attempts of permanently unlocking an object beyond the intended scope of joining.

BurnPromise

A promise object created to ensure the permanent burning of a specified object.

Functions

asset_from_kiosk_to_join

This helper function is intended to facilitate the joining of tokenized assets locked in kiosk. It aids in unlocking the tokenized asset that is set for burning and ensures that another tokenized asset of the same type will eventually contain its balance by returning a JoinPromise.

prove_join

A function utilized to demonstrate that the unlocked tokenized asset is successfully burned and its balance is incorporated into an existing tokenized asset.

asset_from_kiosk_to_burn

Helper function that facilitates the burning of tokenized assets locked in a kiosk. It assists in their unlocking while ensuring a promise that the circulating supply will be reduced, achieved by returning a BurnPromise .

prove_burn

Ensures that the circulating supply of the asset cap is reduced by the balance of the burned tokenized asset.

An example use case package that enables utilization of Rust WASM functionality to support seamless asset creation on the browser.

This is similar to the launchpad approach and serves as the template package whenever a new asset requires representation as a tokenized asset.

Effectively allowing users to edit fields of this template contract on the fly and publish it with the edits included. This package implements two essential modules, each catering to distinct functionalities required for asset tokenization. More details regarding how Rust WASM was implemented can be found in the Web Assembly section.

Modules

template

This is the module that supports defining a new asset.

When you need to represent a new asset as a fractional asset, modify this module to :: , with the (in capitals) being the OTW of this new asset.

This module calls the asset_tokenization::tokenized_asset::new_asset(...) method, which facilitates the declaration of new fields for the asset:

genesis

A genesis type of module that includes a OTW so that the sender can claim the publisher.

The following sequence diagram presenting how the join flow would take place. The following flow assumes that:

The following sequence diagram shows the burn flow and assumes that:

The packages and modules provided demonstrate how you could implement asset tokenization for your project. Your particular use case probably necessitates altering the contract for convenience or to introduce new features.

Instead of implementing the unlock functionality in multiple steps inside of a PTB, it would also be possible to create a method that performs the purchase, borrowing, unlocking and joining of an asset all on one function. This is how that would look like for the joining operation:

The following example splits (effectively replacing) the AssetCap into two new objects: the Treasury and the AdminCap . The access to methods defined in the original package, should now be carefully re-designed as this change can introduce unwanted effects. This required re-design is not entirely contained in this example and only some methods are changed for demonstration purposes (or as a thorough exercise).

Assume you want to allow the users to also burn assets, not only admins. This still needs to be an authorized operation but it would allow the flexibility of consuming tokenized assets for a use case specific purpose (for example, burning all of the collectibles you've gathered to combine them). To achieve this, the admin can mint tickets that contain the ID of the asset they are allowed to burn. To

support this functionality you must redesign the smart contract and separate the admin from the asset's treasury of each asset, which now holds only supply related information. Sample changes that need to happen follow:

Structs

Create a ticket that has only the key ability so that the receiver cannot trade it.

The struct that now only holds treasury related information (results from splitting the AssetCap , meaning it's no longer part of this design) is created as a shared object. Change functions like mint to also take as input both the Treasury object and the AdminCap object.

The other half of the AssetCap functionality which retains the admin capability and the configuration of burnability is an owned object sent to the creator of type .

Method Signatures

The AdminCap here acts both as an admin capability and a type insurance. Encoding the information of both the asset type that is allowed to be deleted with this ticket. This function should assert that the asset T is burnable and return a BurnTicket .

Burning on the user side requires for them to access the shared Treasury object. This function burns the tokenized asset and decreases the supply.

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.

At this stage, you can choose to manually deploy the contracts or utilize the publish bash script that automatically deploys the contracts and sets up most of the .env Asset Tokenization related fields for you. The .env.template file denotes variables that the script automatically fills in. You can see a reference here:

For more details on publishing, please check the setup folder's README .

Select a package for specific instructions.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

In a terminal or console at the move/asset_tokenization directory of the project enter:

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

The package should successfully deploy, and you then see:

You can also view a multitude of information and transactional effects.

You should choose and store the package ID and the registry ID from the created objects in the respective fields within your .env file.

Afterward, it's necessary to modify the Move.toml file. Under the [addresses] section, replace 0x0 with the same package ID . Optionally, under the [package] section, add published-at = (this step is not needed if you see a Move.lock file after running sui client publish).

The fields that are automatically filled are: SUI_NETWORK , ASSET_TOKENIZATION_PACKAGE_ID and REGISTRY .

To publish with the bash script run:

After publishing, you can now edit the Move.toml file like described in the Manual flow.

For more details regarding this process, please consult the setup folder's README .

In a terminal or console at the move/template directory of the project enter:

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

The package should successfully deploy, and you then see:

You can also view a multitude of information and transactional effects.

You should choose and store the package ID , asset metadata ID , asset cap ID and the Publisher ID from the created objects in the respective fields within your .env file.

The process of automatic deployment for the template package refers to publishing a new asset via the WASM library. Quick start steps:

For more details regarding this process, please consult the setup folder's README .

You can find a public facing reference to the WASM library in the move-binary-format-wasm Sui repo subfolder.

This feature was developed with the intent to enable Move bytecode serialization and deserialization on the web. In essence, this feature allows you to edit existing contracts in a web environment.

In the case of asset tokenization, these edits allow you to create and publish new types that represent physical or digital assets that we want to tokenize.

On modifications that are made to the template package this process needs to be repeated. Note that some alterations, like changing a constant name, do not affect the produced bytecode.

Before proceeding to how to make these edits, it's important to understand how the library exposes the template module bytecode. The process is currently manual. This requires that you build and retrieve the compiled bytecode. To do this, navigate inside the template folder and run the following command:

Console response

The response you should receive looks similar to the following:

Copy the output you receive and paste it in the return instruction of the getBytecode method, which is located inside the bytecode-template.ts file.

Additionally, because the template package contains two modules, and therefore has another dependency, you also need to retrieve the bytecode of the genesis module in a similar fashion. This module bytecode, however, is not edited and isn't used as is. This operation is not directly relevant to the WASM library, but is necessary to successfully deploy the edited template module. To acquire the bytecode for genesis, navigate to the template folder and run:

The output format is similar to the template module but smaller in length. Similarly to what you did with the template module, you need to copy this output but this time paste it in the bytecode constant variable located in the genesis_bytecode.ts file.

With the above setup, the library can now manipulate the bytecode by deserializing it, editing it, and serializing it again so that you can publish it.

Taking a look at the template module, you should see that a few constants have been defined:

These constants act as a reference point that the WASM library is able to modify. If you take a look at the TypeScript code that performs the edit and deploys, you can see in action how these fields are identified and updated:

Examine the updateConstant method, which is used to update constants. This method takes four arguments:

The last two arguments are required to minimize the risk of accidentally updating the wrong constant since this library is directly manipulating complied bytecode, which is quite dangerous.

Additionally, the changeIdentifiers method updates identifiers, which in our case are the module name and the struct name. This method takes a JSON object as an argument with keys of the current identifier names in the module and values being the desired names you want to change them into.

Lastly, to deploy the changed template module, build and publish:

As mentioned in the Bytecode manipulation section, the modules that you need to publish are the template and the genesis, hence the

reason you have two elements in the modules array. It's also important to include any dependencies defined in the Move.toml file of the involved packages. The packageId used previously is the address the asset_tokenization package has been deployed to.

Now, you can begin interacting with the deployed smart contract and your tokenized asset.

In a terminal or console within the project's setup directory, utilize the following commands:

Create Transfer Policy

First, create a TransferPolicy and a ProtectedTP with the following command:

After executing the command, the console displays the effects of the transaction.

By searching the transaction digest on a Sui network explorer, you can locate the created objects. Subsequently, select and save the TransferPolicy ID and the ProtectedTP ID from these objects into the respective fields within your .env file.

Add Rules

In the project's file transferPolicyRules.ts located in the directory setup/src/functions , you can modify the code to include the desired rules for your transfer policy.

Code snippet to be modified:

By running the command npm run call tp-rules , the rules will be added to your transfer policy.

Now, investors can trade the fractions of your asset according to the rules you've set.

Select Kiosk

You must place the tokenized assets within a kiosk if marketable assets are desired. Subsequently, you can list and sell them to other users. It's imperative to lock the objects in the kiosk to prevent any future unauthorized usage outside the defined policy that you set.

Best practices recommend a single, comprehensive kiosk for all operations. However, this might not always be the case. Therefore, this project requires the use of only one personal kiosk to ensure consistency and better management, even if you own multiple kiosks.

To enforce this rule, execute the command npm run call select-kiosk . This provides you with the specific kiosk ID to use for this project.

Then, store the provided Kiosk ID in the appropriate field within your .env file.

Mint

In the project's file mint.ts , found in the directory setup/src/functions , you can edit the code to mint the desired type (NFT/FT) and balance for your asset.

As previously mentioned, if additional metadata is provided, the tokenized asset is treated as an NFT with a value of one. However, if there is no extra metadata, the tokenized asset is regarded as an FT, and you have the flexibility to select its balance, which can exceed one.

Here is an example from the code that needs modification:

or

Upon executing the command npm run call mint , a new tokenized asset is minted. You can save the object's ID in the .env file for future reference.

Lock

Locking the objects within the kiosk is crucial to prevent any unauthorized usage beyond the established policy.

Upon executing the command npm run call lock , your newly minted tokenized asset is secured within your kiosk.

Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to lock.

Mint and Lock

Executing the command npm run call mint-lock performs both the mint and lock functions sequentially, ensuring the minted asset is created and immediately locked within the kiosk.

List

Now that your tokenized asset is placed and locked within your kiosk, you can proceed to list it for sale.

In the project's file listItem.ts , found in the directory setup/src/functions , you can adjust the code to specify the desired asset for listing.

Code snippet to be modified:

By running the command npm run call list , your tokenized asset is listed and made available for sale.

Purchase

When a user intends to purchase an item, it needs to be listed for sale. After the user selects the item to buy, they are required to modify the following snippet of code found in the file purchaseItem.ts , located in the setup/src/functions directory.

Apart from specifying the item and its type, the buyer must set the specific price and the seller's kiosk ID to execute the purchase transaction successfully, accomplished by running npm run call purchase .

Join

When you execute the command npm run call join , two specified tokenized assets of the FT type are merged together. Before running the command, make sure that the fields FT1 and FT2 within your .env file are populated with the objects you intend to merge.

Burn

When you intend to burn a tokenized asset, execute the command npm run call burn . Following this action, the specified asset is destroyed. Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to burn.

Get Balance

By executing the command npm run call get-balance , you can retrieve the balance value associated with the specified tokenized asset.

Get Supply

By executing the command npm run call get-supply , you can retrieve the value representing the current circulating supply of the asset.

Get Total Supply

By executing the command npm run call get-total-supply , you can retrieve the value representing the current circulating supply of the asset.

Move packages

As with all smart contracts on Sui, Move provides the logic that powers asset tokenization.

This reference implementation uses the Kiosk standard to ensure that tokenized assets operate within their defined policy. Use the implementation as presented to have marketable tokenized assets that support rules like royalties, commissions, and so on.

If using Kiosk is not a requirement, then you can exclude the unlock module and some of the proxy's methods related to transfer policies.

Select a module to view its details:

The tokenized_asset module operates in a manner similar to the coin library.

When it receives a new one-time witness type, it creates a unique representation of a fractional asset. This module employs similar implementations to some methods found in the Coin module. It encompasses functionalities pertinent to asset tokenization, including new asset creation, minting, splitting, joining, and burning. See One Time Witness in The Move Book for more information.

Structs

AssetCap

Generate an AssetCap for each new asset represented as a fractional NFT. In most scenarios, you should create it as an owned object, which you can then transfer to the platform's administrator for access-restricted method invocation.

AssetMetadata

The AssetMetadata struct defines the metadata representing the entire asset to fractionalize. This should be a shared object.

TokenizedAsset

The TokenizedAsset is minted with a specified balance that is less than or equal to the remaining supply. If the VecMap of an asset is populated with values, indicating multiple unique entries, it is considered an NFT. Conversely, if the VecMap of an asset is not populated, indicating an absence of individual entries, it is considered an FT.

PlatformCap

The PlatformCap refers to the capability issued to the individual who deploys the contract. This capability grants specific permissions or authority related to the platform's functionalities, allowing the deployer certain controlled actions or access rights within the deployed contract.

Functions

init

This function creates a PlatformCap and sends it to the sender.

new_asset

This function holds the responsibility of creating a fresh representation of an asset, defining its crucial attributes. Upon execution, it returns two distinct objects: the AssetCap and AssetMetadata . These objects encapsulate the necessary information and characteristics defining the asset within the system.

mint

The function performs the minting of a tokenized asset. If new metadata is introduced during this process, the resulting tokenized asset is considered unique, resulting in the creation of an NFT with a balance set to 1. Alternatively, if no new metadata is added, the tokenized asset is classified as an FT, permitting its balance to surpass 1, as specified by a provided argument. Upon execution, the function returns the tokenized asset object.

split

This function is provided with a tokenized asset of the FT type and a balance greater than 1, along with a value less than the object's balance, and performs a split operation on the tokenized asset. The operation divides the existing tokenized asset into two separate tokenized assets. The newly created tokenized asset has a balance equal to the given value, while the balance of the provided object is reduced by the specified value. Upon completion, the function returns the newly created tokenized asset. This function does not accept or operate on tokenized assets of the NFT type.

join

This function is given two tokenized assets of the FT type and executes a merge operation on the tokenized assets. The operation involves increasing the balance of the first tokenized asset by the balance of the second one. Subsequently, the second tokenized asset is burned or removed from circulation. After the process concludes, the function returns the ID of the burned tokenized asset.

This function does not accept or operate on tokenized assets of the NFT type.

burn

This function requires the assetCap as a parameter, thereby restricting its invocation solely to the platform admin. Additionally, it

accepts a tokenized asset that is burned as part of its operation. Upon burning the provided tokenized asset, the circulating supply decreases by the balance of the burnt item. It necessitates a tokenized asset that is burnable.

total_supply

This function retrieves and returns the value representing the total supply of the asset.

supply

This function retrieves and returns the value representing the current circulating supply of the asset.

value

This function takes a tokenized asset as input and retrieves its associated balance value.

create_vec_map_from_arrays

This internal helper function populates a VecMap . It assists in the process of filling or setting key-value pairs within the VecMap data structure.

The proxy module comprises methods that the type owner utilizes to execute publisher-related operations.

Structs

Proxy

The PROXY struct represents the one-time witness (OTW) to claim the publisher.

Registry

This shared object serves as a repository for the Publisher object, specifically intended to control and restrict access to the creation and management of transfer policies for tokenized assets. Mutable access to this object is exclusively granted to the actual publisher.

ProtectedTP

This is a shared object that stores an empty transfer policy. It is required to create one per type generated by a user. Its involvement is apparent in the unlock module.

Functions

init

This function is responsible for creating the Publisher object, encapsulating it within the registry, and subsequently sharing the Registry object.

setup_tp

This function leverages the publisher nested within the registry and the sender's publisher. It generates and returns a transfer policy and the associated transfer policy cap specific to the TokenizedAsset . This type 'T' is derived from the Publisher object.

It also generates an empty transfer policy wrapped in a ProtectedTP object, which is shared. You can use this functionality under specific conditions to override the kiosk lock rule.

new_display

This function utilizes the publisher nested within the registry and the sender's publisher to generate and return an empty Display for the type TokenizedAsset , where T is encapsulated within the Publisher object.

transfer_policy

This function, provided with the protectedTP , returns the transfer policy specifically designed for the type TokenizedAsset

publisher_mut

This function can only be accessed by the owner of the platform cap. It requires the registry as an argument to obtain a mutable reference to the publisher.

The unlock module facilitates the unlocking of a tokenized asset specifically for authorized burning and joining.

It allows tokenized asset type creators to enable these operations for kiosk assets without necessitating adherence to the default set of requirements, such as rules or policies.

Structs

JoinPromise

A promise object is established to prevent attempts of permanently unlocking an object beyond the intended scope of joining.

BurnPromise

A promise object created to ensure the permanent burning of a specified object.

Functions

asset_from_kiosk_to_join

This helper function is intended to facilitate the joining of tokenized assets locked in kiosk. It aids in unlocking the tokenized asset that is set for burning and ensures that another tokenized asset of the same type will eventually contain its balance by returning a JoinPromise.

prove_join

A function utilized to demonstrate that the unlocked tokenized asset is successfully burned and its balance is incorporated into an existing tokenized asset.

asset_from_kiosk_to_burn

Helper function that facilitates the burning of tokenized assets locked in a kiosk. It assists in their unlocking while ensuring a promise that the circulating supply will be reduced, achieved by returning a BurnPromise .

prove_burn

Ensures that the circulating supply of the asset cap is reduced by the balance of the burned tokenized asset.

An example use case package that enables utilization of Rust WASM functionality to support seamless asset creation on the browser.

This is similar to the launchpad approach and serves as the template package whenever a new asset requires representation as a tokenized asset.

Effectively allowing users to edit fields of this template contract on the fly and publish it with the edits included. This package implements two essential modules, each catering to distinct functionalities required for asset tokenization. More details regarding how Rust WASM was implemented can be found in the Web Assembly section.

Modules

template

This is the module that supports defining a new asset.

When you need to represent a new asset as a fractional asset, modify this module to :: , with the (in capitals) being the OTW of this new asset.

This module calls the asset_tokenization::tokenized_asset::new_asset(...) method, which facilitates the declaration of new fields for the asset:

genesis

A genesis type of module that includes a OTW so that the sender can claim the publisher.

The following sequence diagram presenting how the join flow would take place. The following flow assumes that:

The following sequence diagram shows the burn flow and assumes that:

The packages and modules provided demonstrate how you could implement asset tokenization for your project. Your particular use case probably necessitates altering the contract for convenience or to introduce new features.

Instead of implementing the unlock functionality in multiple steps inside of a PTB, it would also be possible to create a method that performs the purchase, borrowing, unlocking and joining of an asset all on one function. This is how that would look like for the joining operation:

The following example splits (effectively replacing) the AssetCap into two new objects: the Treasury and the AdminCap . The access to methods defined in the original package, should now be carefully re-designed as this change can introduce unwanted effects. This required re-design is not entirely contained in this example and only some methods are changed for demonstration purposes (or as a thorough exercise).

Assume you want to allow the users to also burn assets, not only admins. This still needs to be an authorized operation but it would allow the flexibility of consuming tokenized assets for a use case specific purpose (for example, burning all of the collectibles you've gathered to combine them). To achieve this, the admin can mint tickets that contain the ID of the asset they are allowed to burn. To support this functionality you must redesign the smart contract and separate the admin from the asset's treasury of each asset, which now holds only supply related information. Sample changes that need to happen follow:

Structs

Create a ticket that has only the key ability so that the receiver cannot trade it.

The struct that now only holds treasury related information (results from splitting the AssetCap , meaning it's no longer part of this design) is created as a shared object. Change functions like mint to also take as input both the Treasury object and the AdminCap object.

The other half of the AssetCap functionality which retains the admin capability and the configuration of burnability is an owned object sent to the creator of type .

Method Signatures

The AdminCap here acts both as an admin capability and a type insurance. Encoding the information of both the asset type that is allowed to be deleted with this ticket. This function should assert that the asset T is burnable and return a BurnTicket .

Burning on the user side requires for them to access the shared Treasury object. This function burns the tokenized asset and decreases the supply.

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.

At this stage, you can choose to manually deploy the contracts or utilize the publish bash script that automatically deploys the contracts and sets up most of the .env Asset Tokenization related fields for you. The .env.template file denotes variables that the script automatically fills in. You can see a reference here:

For more details on publishing, please check the setup folder's README .

Select a package for specific instructions.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

In a terminal or console at the move/asset_tokenization directory of the project enter:

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

The package should successfully deploy, and you then see:

You can also view a multitude of information and transactional effects.

You should choose and store the package ID and the registry ID from the created objects in the respective fields within your .env file.

Afterward, it's necessary to modify the Move.toml file. Under the [addresses] section, replace 0x0 with the same package ID . Optionally, under the [package] section, add published-at = (this step is not needed if you see a Move.lock file after running sui client publish).

The fields that are automatically filled are: SUI_NETWORK , ASSET_TOKENIZATION_PACKAGE_ID and REGISTRY .

To publish with the bash script run:

After publishing, you can now edit the Move.toml file like described in the Manual flow.

For more details regarding this process, please consult the setup folder's README .

In a terminal or console at the move/template directory of the project enter:

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

The package should successfully deploy, and you then see:

You can also view a multitude of information and transactional effects.

You should choose and store the package ID , asset metadata ID , asset cap ID and the Publisher ID from the created objects in the respective fields within your .env file.

The process of automatic deployment for the template package refers to publishing a new asset via the WASM library. Quick start steps:

For more details regarding this process, please consult the setup folder's README .

You can find a public facing reference to the WASM library in the move-binary-format-wasm Sui repo subfolder.

This feature was developed with the intent to enable Move bytecode serialization and deserialization on the web. In essence, this feature allows you to edit existing contracts in a web environment.

In the case of asset tokenization, these edits allow you to create and publish new types that represent physical or digital assets that we want to tokenize.

On modifications that are made to the template package this process needs to be repeated. Note that some alterations, like changing a constant name, do not affect the produced bytecode.

Before proceeding to how to make these edits, it's important to understand how the library exposes the template module bytecode. The process is currently manual. This requires that you build and retrieve the compiled bytecode. To do this, navigate inside the template folder and run the following command:

Console response

The response you should receive looks similar to the following:

Copy the output you receive and paste it in the return instruction of the getBytecode method, which is located inside the bytecode-template.ts file.

Additionally, because the template package contains two modules, and therefore has another dependency, you also need to retrieve the bytecode of the genesis module in a similar fashion. This module bytecode, however, is not edited and isn't used as is. This operation is not directly relevant to the WASM library, but is necessary to successfully deploy the edited template module. To acquire the bytecode for genesis, navigate to the template folder and run:

The output format is similar to the template module but smaller in length. Similarly to what you did with the template module, you need to copy this output but this time paste it in the bytecode constant variable located in the genesis_bytecode.ts file.

With the above setup, the library can now manipulate the bytecode by deserializing it, editing it, and serializing it again so that you can publish it.

Taking a look at the template module, you should see that a few constants have been defined:

These constants act as a reference point that the WASM library is able to modify. If you take a look at the TypeScript code that performs the edit and deploys, you can see in action how these fields are identified and updated:

Examine the updateConstant method, which is used to update constants. This method takes four arguments:

The last two arguments are required to minimize the risk of accidentally updating the wrong constant since this library is directly manipulating complied bytecode, which is quite dangerous.

Additionally, the changeIdentifiers method updates identifiers, which in our case are the module name and the struct name. This method takes a JSON object as an argument with keys of the current identifier names in the module and values being the desired names you want to change them into.

Lastly, to deploy the changed template module, build and publish:

As mentioned in the Bytecode manipulation section, the modules that you need to publish are the template and the genesis, hence the reason you have two elements in the modules array. It's also important to include any dependencies defined in the Move.toml file of the involved packages. The packageId used previously is the address the asset_tokenization package has been deployed to.

Now, you can begin interacting with the deployed smart contract and your tokenized asset.

In a terminal or console within the project's setup directory, utilize the following commands:

Create Transfer Policy

First, create a TransferPolicy and a ProtectedTP with the following command:

After executing the command, the console displays the effects of the transaction.

By searching the transaction digest on a Sui network explorer, you can locate the created objects. Subsequently, select and save the TransferPolicy ID and the ProtectedTP ID from these objects into the respective fields within your .env file.

Add Rules

In the project's file transferPolicyRules.ts located in the directory setup/src/functions , you can modify the code to include the desired rules for your transfer policy.

Code snippet to be modified:

By running the command npm run call tp-rules , the rules will be added to your transfer policy.

Now, investors can trade the fractions of your asset according to the rules you've set.

Select Kiosk

You must place the tokenized assets within a kiosk if marketable assets are desired. Subsequently, you can list and sell them to other users. It's imperative to lock the objects in the kiosk to prevent any future unauthorized usage outside the defined policy that you set.

Best practices recommend a single, comprehensive kiosk for all operations. However, this might not always be the case. Therefore, this project requires the use of only one personal kiosk to ensure consistency and better management, even if you own multiple kiosks.

To enforce this rule, execute the command npm run call select-kiosk . This provides you with the specific kiosk ID to use for this project.

Then, store the provided Kiosk ID in the appropriate field within your .env file.

Mint

In the project's file mint.ts , found in the directory setup/src/functions , you can edit the code to mint the desired type (NFT/FT) and balance for your asset.

As previously mentioned, if additional metadata is provided, the tokenized asset is treated as an NFT with a value of one. However, if there is no extra metadata, the tokenized asset is regarded as an FT, and you have the flexibility to select its balance, which can exceed one.

Here is an example from the code that needs modification:

or

Upon executing the command npm run call mint , a new tokenized asset is minted. You can save the object's ID in the .env file for future reference.

Lock

Locking the objects within the kiosk is crucial to prevent any unauthorized usage beyond the established policy.

Upon executing the command npm run call lock , your newly minted tokenized asset is secured within your kiosk.

Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to lock.

Mint and Lock

Executing the command npm run call mint-lock performs both the mint and lock functions sequentially, ensuring the minted asset is created and immediately locked within the kiosk.

List

Now that your tokenized asset is placed and locked within your kiosk, you can proceed to list it for sale.

In the project's file listItem.ts , found in the directory setup/src/functions , you can adjust the code to specify the desired asset for listing.

Code snippet to be modified:

By running the command npm run call list , your tokenized asset is listed and made available for sale.

Purchase

When a user intends to purchase an item, it needs to be listed for sale. After the user selects the item to buy, they are required to modify the following snippet of code found in the file purchaseItem.ts , located in the setup/src/functions directory.

Apart from specifying the item and its type, the buyer must set the specific price and the seller's kiosk ID to execute the purchase transaction successfully, accomplished by running npm run call purchase .

Join

When you execute the command npm run call join , two specified tokenized assets of the FT type are merged together. Before running the command, make sure that the fields FT1 and FT2 within your .env file are populated with the objects you intend to merge.

Burn

When you intend to burn a tokenized asset, execute the command npm run call burn . Following this action, the specified asset is destroyed. Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to burn.

Get Balance

By executing the command npm run call get-balance , you can retrieve the balance value associated with the specified tokenized asset.

Get Supply

By executing the command npm run call get-supply , you can retrieve the value representing the current circulating supply of the asset.

Get Total Supply

By executing the command npm run call get-total-supply , you can retrieve the value representing the current circulating supply of the asset.

Variations

The packages and modules provided demonstrate how you could implement asset tokenization for your project. Your particular use case probably necessitates altering the contract for convenience or to introduce new features.

Instead of implementing the unlock functionality in multiple steps inside of a PTB, it would also be possible to create a method that performs the purchase, borrowing, unlocking and joining of an asset all on one function. This is how that would look like for the joining operation:

The following example splits (effectively replacing) the AssetCap into two new objects: the Treasury and the AdminCap . The access to methods defined in the original package, should now be carefully re-designed as this change can introduce unwanted effects. This required re-design is not entirely contained in this example and only some methods are changed for demonstration purposes (or as a thorough exercise).

Assume you want to allow the users to also burn assets, not only admins. This still needs to be an authorized operation but it would allow the flexibility of consuming tokenized assets for a use case specific purpose (for example, burning all of the collectibles you've gathered to combine them). To achieve this, the admin can mint tickets that contain the ID of the asset they are allowed to burn. To support this functionality you must redesign the smart contract and separate the admin from the asset's treasury of each asset, which now holds only supply related information. Sample changes that need to happen follow:

Structs

Create a ticket that has only the key ability so that the receiver cannot trade it.

The struct that now only holds treasury related information (results from splitting the AssetCap , meaning it's no longer part of this design) is created as a shared object. Change functions like mint to also take as input both the Treasury object and the AdminCap object.

The other half of the AssetCap functionality which retains the admin capability and the configuration of burnability is an owned object sent to the creator of type .

Method Signatures

The AdminCap here acts both as an admin capability and a type insurance. Encoding the information of both the asset type that is allowed to be deleted with this ticket. This function should assert that the asset T is burnable and return a BurnTicket .

Burning on the user side requires for them to access the shared Treasury object. This function burns the tokenized asset and decreases the supply.

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.

At this stage, you can choose to manually deploy the contracts or utilize the publish bash script that automatically deploys the contracts and sets up most of the .env Asset Tokenization related fields for you. The .env.template file denotes variables that the script automatically fills in. You can see a reference here:

For more details on publishing, please check the setup folder's README .

Select a package for specific instructions.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

In a terminal or console at the move/asset_tokenization directory of the project enter:

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

The package should successfully deploy, and you then see:

You can also view a multitude of information and transactional effects.

You should choose and store the package ID and the registry ID from the created objects in the respective fields within your .env file.

Afterward, it's necessary to modify the Move.toml file. Under the [addresses] section, replace 0x0 with the same package ID . Optionally, under the [package] section, add published-at = (this step is not needed if you see a Move.lock file after running sui client publish).

The fields that are automatically filled are: SUI_NETWORK , ASSET_TOKENIZATION_PACKAGE_ID and REGISTRY .

To publish with the bash script run:

After publishing, you can now edit the Move.toml file like described in the Manual flow.

For more details regarding this process, please consult the setup folder's README .

In a terminal or console at the move/template directory of the project enter:

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

The package should successfully deploy, and you then see:

You can also view a multitude of information and transactional effects.

You should choose and store the package ID , asset metadata ID , asset cap ID and the Publisher ID from the created objects in the respective fields within your .env file.

The process of automatic deployment for the template package refers to publishing a new asset via the WASM library. Quick start steps:

For more details regarding this process, please consult the setup folder's README .

You can find a public facing reference to the WASM library in the move-binary-format-wasm Sui repo subfolder.

This feature was developed with the intent to enable Move bytecode serialization and deserialization on the web. In essence, this feature allows you to edit existing contracts in a web environment.

In the case of asset tokenization, these edits allow you to create and publish new types that represent physical or digital assets that we want to tokenize.

On modifications that are made to the template package this process needs to be repeated. Note that some alterations, like changing a constant name, do not affect the produced bytecode.

Before proceeding to how to make these edits, it's important to understand how the library exposes the template module bytecode. The process is currently manual. This requires that you build and retrieve the compiled bytecode. To do this, navigate inside the template folder and run the following command:

Console response

The response you should receive looks similar to the following:

Copy the output you receive and paste it in the return instruction of the getBytecode method, which is located inside the bytecode-template.ts file.

Additionally, because the template package contains two modules, and therefore has another dependency, you also need to retrieve the bytecode of the genesis module in a similar fashion. This module bytecode, however, is not edited and isn't used as is. This operation is not directly relevant to the WASM library, but is necessary to successfully deploy the edited template module. To acquire the bytecode for genesis, navigate to the template folder and run:

The output format is similar to the template module but smaller in length. Similarly to what you did with the template module, you need to copy this output but this time paste it in the bytecode constant variable located in the genesis_bytecode.ts file.

With the above setup, the library can now manipulate the bytecode by deserializing it, editing it, and serializing it again so that you

can publish it.

Taking a look at the template module, you should see that a few constants have been defined:

These constants act as a reference point that the WASM library is able to modify. If you take a look at the TypeScript code that performs the edit and deploys, you can see in action how these fields are identified and updated:

Examine the updateConstant method, which is used to update constants. This method takes four arguments:

The last two arguments are required to minimize the risk of accidentally updating the wrong constant since this library is directly manipulating complied bytecode, which is quite dangerous.

Additionally, the changeIdentifiers method updates identifiers, which in our case are the module name and the struct name. This method takes a JSON object as an argument with keys of the current identifier names in the module and values being the desired names you want to change them into.

Lastly, to deploy the changed template module, build and publish:

As mentioned in the Bytecode manipulation section, the modules that you need to publish are the template and the genesis, hence the reason you have two elements in the modules array. It's also important to include any dependencies defined in the Move.toml file of the involved packages. The packageId used previously is the address the asset_tokenization package has been deployed to.

Now, you can begin interacting with the deployed smart contract and your tokenized asset.

In a terminal or console within the project's setup directory, utilize the following commands:

Create Transfer Policy

First, create a TransferPolicy and a ProtectedTP with the following command:

After executing the command, the console displays the effects of the transaction.

By searching the transaction digest on a Sui network explorer, you can locate the created objects. Subsequently, select and save the TransferPolicy ID and the ProtectedTP ID from these objects into the respective fields within your .env file.

Add Rules

In the project's file transferPolicyRules.ts located in the directory setup/src/functions , you can modify the code to include the desired rules for your transfer policy.

Code snippet to be modified:

By running the command npm run call tp-rules , the rules will be added to your transfer policy.

Now, investors can trade the fractions of your asset according to the rules you've set.

Select Kiosk

You must place the tokenized assets within a kiosk if marketable assets are desired. Subsequently, you can list and sell them to other users. It's imperative to lock the objects in the kiosk to prevent any future unauthorized usage outside the defined policy that you set.

Best practices recommend a single, comprehensive kiosk for all operations. However, this might not always be the case. Therefore, this project requires the use of only one personal kiosk to ensure consistency and better management, even if you own multiple kiosks.

To enforce this rule, execute the command npm run call select-kiosk . This provides you with the specific kiosk ID to use for this project.

Then, store the provided Kiosk ID in the appropriate field within your .env file.

Mint

In the project's file mint.ts , found in the directory setup/src/functions , you can edit the code to mint the desired type (NFT/FT) and balance for your asset.

As previously mentioned, if additional metadata is provided, the tokenized asset is treated as an NFT with a value of one. However,

if there is no extra metadata, the tokenized asset is regarded as an FT, and you have the flexibility to select its balance, which can exceed one.

Here is an example from the code that needs modification:

or

Upon executing the command npm run call mint , a new tokenized asset is minted. You can save the object's ID in the .env file for future reference.

Lock

Locking the objects within the kiosk is crucial to prevent any unauthorized usage beyond the established policy.

Upon executing the command npm run call lock , your newly minted tokenized asset is secured within your kiosk.

Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to lock.

Mint and Lock

Executing the command npm run call mint-lock performs both the mint and lock functions sequentially, ensuring the minted asset is created and immediately locked within the kiosk.

List

Now that your tokenized asset is placed and locked within your kiosk, you can proceed to list it for sale.

In the project's file listItem.ts , found in the directory setup/src/functions , you can adjust the code to specify the desired asset for listing.

Code snippet to be modified:

By running the command npm run call list , your tokenized asset is listed and made available for sale.

Purchase

When a user intends to purchase an item, it needs to be listed for sale. After the user selects the item to buy, they are required to modify the following snippet of code found in the file purchaseItem.ts , located in the setup/src/functions directory.

Apart from specifying the item and its type, the buyer must set the specific price and the seller's kiosk ID to execute the purchase transaction successfully, accomplished by running npm run call purchase .

Join

When you execute the command npm run call join , two specified tokenized assets of the FT type are merged together. Before running the command, make sure that the fields FT1 and FT2 within your .env file are populated with the objects you intend to merge.

Burn

When you intend to burn a tokenized asset, execute the command npm run call burn . Following this action, the specified asset is destroyed. Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to burn.

Get Balance

By executing the command npm run call get-balance , you can retrieve the balance value associated with the specified tokenized asset.

Get Supply

By executing the command npm run call get-supply , you can retrieve the value representing the current circulating supply of the asset.

Get Total Supply

By executing the command npm run call get-total-supply , you can retrieve the value representing the current circulating supply of the asset.

Publishing

At this stage, you can choose to manually deploy the contracts or utilize the publish bash script that automatically deploys the contracts and sets up most of the .env Asset Tokenization related fields for you. The .env.template file denotes variables that the script automatically fills in. You can see a reference here:

For more details on publishing, please check the setup folder's README .

Select a package for specific instructions.

Beginning with the Sui v1.24.1 release , the --gas-budget option is no longer required for CLI commands.

In a terminal or console at the move/asset_tokenization directory of the project enter:

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

The package should successfully deploy, and you then see:

You can also view a multitude of information and transactional effects.

You should choose and store the package ID and the registry ID from the created objects in the respective fields within your .env file.

Afterward, it's necessary to modify the Move.toml file. Under the [addresses] section, replace 0x0 with the same package ID . Optionally, under the [package] section, add published-at = (this step is not needed if you see a Move.lock file after running sui client publish).

The fields that are automatically filled are: SUI_NETWORK , ASSET_TOKENIZATION_PACKAGE_ID and REGISTRY .

To publish with the bash script run:

After publishing, you can now edit the Move.toml file like described in the Manual flow.

For more details regarding this process, please consult the setup folder's README .

In a terminal or console at the move/template directory of the project enter:

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

The package should successfully deploy, and you then see:

You can also view a multitude of information and transactional effects.

You should choose and store the package ID , asset metadata ID , asset cap ID and the Publisher ID from the created objects in the respective fields within your .env file.

The process of automatic deployment for the template package refers to publishing a new asset via the WASM library. Quick start steps:

For more details regarding this process, please consult the setup folder's README .

You can find a public facing reference to the WASM library in the move-binary-format-wasm Sui repo subfolder.

This feature was developed with the intent to enable Move bytecode serialization and deserialization on the web. In essence, this feature allows you to edit existing contracts in a web environment.

In the case of asset tokenization, these edits allow you to create and publish new types that represent physical or digital assets that we want to tokenize.

On modifications that are made to the template package this process needs to be repeated. Note that some alterations, like changing a constant name, do not affect the produced bytecode.

Before proceeding to how to make these edits, it's important to understand how the library exposes the template module bytecode. The process is currently manual. This requires that you build and retrieve the compiled bytecode. To do this, navigate inside the template folder and run the following command:

Console response

The response you should receive looks similar to the following:

Copy the output you receive and paste it in the return instruction of the getBytecode method, which is located inside the bytecode-template.ts file.

Additionally, because the template package contains two modules, and therefore has another dependency, you also need to retrieve the bytecode of the genesis module in a similar fashion. This module bytecode, however, is not edited and isn't used as is. This operation is not directly relevant to the WASM library, but is necessary to successfully deploy the edited template module. To acquire the bytecode for genesis, navigate to the template folder and run:

The output format is similar to the template module but smaller in length. Similarly to what you did with the template module, you need to copy this output but this time paste it in the bytecode constant variable located in the genesis_bytecode.ts file.

With the above setup, the library can now manipulate the bytecode by deserializing it, editing it, and serializing it again so that you can publish it.

Taking a look at the template module, you should see that a few constants have been defined:

These constants act as a reference point that the WASM library is able to modify. If you take a look at the TypeScript code that performs the edit and deploys, you can see in action how these fields are identified and updated:

Examine the updateConstant method, which is used to update constants. This method takes four arguments:

The last two arguments are required to minimize the risk of accidentally updating the wrong constant since this library is directly manipulating complied bytecode, which is quite dangerous.

Additionally, the changeIdentifiers method updates identifiers, which in our case are the module name and the struct name. This method takes a JSON object as an argument with keys of the current identifier names in the module and values being the desired names you want to change them into.

Lastly, to deploy the changed template module, build and publish:

As mentioned in the Bytecode manipulation section, the modules that you need to publish are the template and the genesis, hence the reason you have two elements in the modules array. It's also important to include any dependencies defined in the Move.toml file of the involved packages. The packageId used previously is the address the asset_tokenization package has been deployed to.

Now, you can begin interacting with the deployed smart contract and your tokenized asset.

In a terminal or console within the project's setup directory, utilize the following commands:

Create Transfer Policy

First, create a TransferPolicy and a ProtectedTP with the following command:

After executing the command, the console displays the effects of the transaction.

By searching the transaction digest on a Sui network explorer, you can locate the created objects. Subsequently, select and save the TransferPolicy ID and the ProtectedTP ID from these objects into the respective fields within your .env file.

Add Rules

In the project's file transferPolicyRules.ts located in the directory setup/src/functions , you can modify the code to include the desired rules for your transfer policy.

Code snippet to be modified:

By running the command npm run call tp-rules , the rules will be added to your transfer policy.

Now, investors can trade the fractions of your asset according to the rules you've set.

Select Kiosk

You must place the tokenized assets within a kiosk if marketable assets are desired. Subsequently, you can list and sell them to other users. It's imperative to lock the objects in the kiosk to prevent any future unauthorized usage outside the defined policy that you set.

Best practices recommend a single, comprehensive kiosk for all operations. However, this might not always be the case. Therefore, this project requires the use of only one personal kiosk to ensure consistency and better management, even if you own multiple kiosks.

To enforce this rule, execute the command npm run call select-kiosk . This provides you with the specific kiosk ID to use for this project.

Then, store the provided Kiosk ID in the appropriate field within your .env file.

Mint

In the project's file mint.ts , found in the directory setup/src/functions , you can edit the code to mint the desired type (NFT/FT) and balance for your asset.

As previously mentioned, if additional metadata is provided, the tokenized asset is treated as an NFT with a value of one. However, if there is no extra metadata, the tokenized asset is regarded as an FT, and you have the flexibility to select its balance, which can exceed one.

Here is an example from the code that needs modification:

or

Upon executing the command npm run call mint , a new tokenized asset is minted. You can save the object's ID in the .env file for future reference.

Lock

Locking the objects within the kiosk is crucial to prevent any unauthorized usage beyond the established policy.

Upon executing the command npm run call lock , your newly minted tokenized asset is secured within your kiosk.

Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to lock.

Mint and Lock

Executing the command npm run call mint-lock performs both the mint and lock functions sequentially, ensuring the minted asset is created and immediately locked within the kiosk.

List

Now that your tokenized asset is placed and locked within your kiosk, you can proceed to list it for sale.

In the project's file listItem.ts , found in the directory setup/src/functions , you can adjust the code to specify the desired asset for listing.

Code snippet to be modified:

By running the command npm run call list , your tokenized asset is listed and made available for sale.

Purchase

When a user intends to purchase an item, it needs to be listed for sale. After the user selects the item to buy, they are required to modify the following snippet of code found in the file purchaseItem.ts , located in the setup/src/functions directory.

Apart from specifying the item and its type, the buyer must set the specific price and the seller's kiosk ID to execute the purchase transaction successfully, accomplished by running npm run call purchase .

Join

When you execute the command npm run call join , two specified tokenized assets of the FT type are merged together. Before

running the command, make sure that the fields FT1 and FT2 within your .env file are populated with the objects you intend to merge.

Burn

When you intend to burn a tokenized asset, execute the command npm run call burn . Following this action, the specified asset is destroyed. Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to burn.

Get Balance

By executing the command npm run call get-balance , you can retrieve the balance value associated with the specified tokenized asset.

Get Supply

By executing the command npm run call get-supply , you can retrieve the value representing the current circulating supply of the asset.

Get Total Supply

By executing the command npm run call get-total-supply , you can retrieve the value representing the current circulating supply of the asset.

WebAssembly (WASM) and template package

You can find a public facing reference to the WASM library in the move-binary-format-wasm Sui repo subfolder.

This feature was developed with the intent to enable Move bytecode serialization and deserialization on the web. In essence, this feature allows you to edit existing contracts in a web environment.

In the case of asset tokenization, these edits allow you to create and publish new types that represent physical or digital assets that we want to tokenize.

On modifications that are made to the template package this process needs to be repeated. Note that some alterations, like changing a constant name, do not affect the produced bytecode.

Before proceeding to how to make these edits, it's important to understand how the library exposes the template module bytecode. The process is currently manual. This requires that you build and retrieve the compiled bytecode. To do this, navigate inside the template folder and run the following command:

Console response

The response you should receive looks similar to the following:

Copy the output you receive and paste it in the return instruction of the getBytecode method, which is located inside the bytecode-template.ts file.

Additionally, because the template package contains two modules, and therefore has another dependency, you also need to retrieve the bytecode of the genesis module in a similar fashion. This module bytecode, however, is not edited and isn't used as is. This operation is not directly relevant to the WASM library, but is necessary to successfully deploy the edited template module. To acquire the bytecode for genesis, navigate to the template folder and run:

The output format is similar to the template module but smaller in length. Similarly to what you did with the template module, you need to copy this output but this time paste it in the bytecode constant variable located in the genesis_bytecode.ts file.

With the above setup, the library can now manipulate the bytecode by deserializing it, editing it, and serializing it again so that you can publish it.

Taking a look at the template module, you should see that a few constants have been defined:

These constants act as a reference point that the WASM library is able to modify. If you take a look at the TypeScript code that performs the edit and deploys, you can see in action how these fields are identified and updated:

Examine the updateConstant method, which is used to update constants. This method takes four arguments:

The last two arguments are required to minimize the risk of accidentally updating the wrong constant since this library is directly manipulating complied bytecode, which is quite dangerous.

Additionally, the changeIdentifiers method updates identifiers, which in our case are the module name and the struct name. This method takes a JSON object as an argument with keys of the current identifier names in the module and values being the desired names you want to change them into.

Lastly, to deploy the changed template module, build and publish:

As mentioned in the Bytecode manipulation section, the modules that you need to publish are the template and the genesis, hence the reason you have two elements in the modules array. It's also important to include any dependencies defined in the Move.toml file of the involved packages. The packageId used previously is the address the asset_tokenization package has been deployed to.

Now, you can begin interacting with the deployed smart contract and your tokenized asset.

In a terminal or console within the project's setup directory, utilize the following commands:

Create Transfer Policy

First, create a TransferPolicy and a ProtectedTP with the following command:

After executing the command, the console displays the effects of the transaction.

By searching the transaction digest on a Sui network explorer, you can locate the created objects. Subsequently, select and save the TransferPolicy ID and the ProtectedTP ID from these objects into the respective fields within your .env file.

Add Rules

In the project's file transferPolicyRules.ts located in the directory setup/src/functions , you can modify the code to include the desired rules for your transfer policy.

Code snippet to be modified:

By running the command npm run call tp-rules , the rules will be added to your transfer policy.

Now, investors can trade the fractions of your asset according to the rules you've set.

Select Kiosk

You must place the tokenized assets within a kiosk if marketable assets are desired. Subsequently, you can list and sell them to other users. It's imperative to lock the objects in the kiosk to prevent any future unauthorized usage outside the defined policy that you set.

Best practices recommend a single, comprehensive kiosk for all operations. However, this might not always be the case. Therefore, this project requires the use of only one personal kiosk to ensure consistency and better management, even if you own multiple kiosks.

To enforce this rule, execute the command npm run call select-kiosk . This provides you with the specific kiosk ID to use for this project.

Then, store the provided Kiosk ID in the appropriate field within your .env file.

Mint

In the project's file mint.ts , found in the directory setup/src/functions , you can edit the code to mint the desired type (NFT/FT) and balance for your asset.

As previously mentioned, if additional metadata is provided, the tokenized asset is treated as an NFT with a value of one. However, if there is no extra metadata, the tokenized asset is regarded as an FT, and you have the flexibility to select its balance, which can exceed one.

Here is an example from the code that needs modification:

or

Upon executing the command npm run call mint , a new tokenized asset is minted. You can save the object's ID in the .env file for future reference.

Lock

Locking the objects within the kiosk is crucial to prevent any unauthorized usage beyond the established policy.

Upon executing the command npm run call lock , your newly minted tokenized asset is secured within your kiosk.

Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to lock.

Mint and Lock

Executing the command npm run call mint-lock performs both the mint and lock functions sequentially, ensuring the minted asset is created and immediately locked within the kiosk.

List

Now that your tokenized asset is placed and locked within your kiosk, you can proceed to list it for sale.

In the project's file listItem.ts , found in the directory setup/src/functions , you can adjust the code to specify the desired asset for listing.

Code snippet to be modified:

By running the command npm run call list , your tokenized asset is listed and made available for sale.

Purchase

When a user intends to purchase an item, it needs to be listed for sale. After the user selects the item to buy, they are required to modify the following snippet of code found in the file purchaseItem.ts , located in the setup/src/functions directory.

Apart from specifying the item and its type, the buyer must set the specific price and the seller's kiosk ID to execute the purchase transaction successfully, accomplished by running npm run call purchase .

Join

When you execute the command npm run call join , two specified tokenized assets of the FT type are merged together. Before running the command, make sure that the fields FT1 and FT2 within your .env file are populated with the objects you intend to merge.

Burn

When you intend to burn a tokenized asset, execute the command npm run call burn . Following this action, the specified asset is destroyed. Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to burn.

Get Balance

By executing the command npm run call get-balance , you can retrieve the balance value associated with the specified tokenized asset.

Get Supply

By executing the command npm run call get-supply , you can retrieve the value representing the current circulating supply of the asset.

Get Total Supply

By executing the command npm run call get-total-supply , you can retrieve the value representing the current circulating supply of the asset.

TypeScript

Now, you can begin interacting with the deployed smart contract and your tokenized asset.

In a terminal or console within the project's setup directory, utilize the following commands:

Create Transfer Policy

First, create a TransferPolicy and a ProtectedTP with the following command:

After executing the command, the console displays the effects of the transaction.

By searching the transaction digest on a Sui network explorer, you can locate the created objects. Subsequently, select and save the TransferPolicy ID and the ProtectedTP ID from these objects into the respective fields within your .env file.

Add Rules

In the project's file transferPolicyRules.ts located in the directory setup/src/functions , you can modify the code to include the desired rules for your transfer policy.

Code snippet to be modified:

By running the command npm run call tp-rules , the rules will be added to your transfer policy.

Now, investors can trade the fractions of your asset according to the rules you've set.

Select Kiosk

You must place the tokenized assets within a kiosk if marketable assets are desired. Subsequently, you can list and sell them to other users. It's imperative to lock the objects in the kiosk to prevent any future unauthorized usage outside the defined policy that you set.

Best practices recommend a single, comprehensive kiosk for all operations. However, this might not always be the case. Therefore, this project requires the use of only one personal kiosk to ensure consistency and better management, even if you own multiple kiosks.

To enforce this rule, execute the command npm run call select-kiosk . This provides you with the specific kiosk ID to use for this project.

Then, store the provided Kiosk ID in the appropriate field within your .env file.

Mint

In the project's file mint.ts , found in the directory setup/src/functions , you can edit the code to mint the desired type (NFT/FT) and balance for your asset.

As previously mentioned, if additional metadata is provided, the tokenized asset is treated as an NFT with a value of one. However, if there is no extra metadata, the tokenized asset is regarded as an FT, and you have the flexibility to select its balance, which can exceed one.

Here is an example from the code that needs modification:

or

Upon executing the command npm run call mint , a new tokenized asset is minted. You can save the object's ID in the .env file for future reference.

Lock

Locking the objects within the kiosk is crucial to prevent any unauthorized usage beyond the established policy.

Upon executing the command npm run call lock , your newly minted tokenized asset is secured within your kiosk.

Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to lock.

Mint and Lock

Executing the command npm run call mint-lock performs both the mint and lock functions sequentially, ensuring the minted asset is created and immediately locked within the kiosk.

List

Now that your tokenized asset is placed and locked within your kiosk, you can proceed to list it for sale.

In the project's file listItem.ts , found in the directory setup/src/functions , you can adjust the code to specify the desired asset for listing.

Code snippet to be modified:

By running the command npm run call list , your tokenized asset is listed and made available for sale.

Purchase

When a user intends to purchase an item, it needs to be listed for sale. After the user selects the item to buy, they are required to modify the following snippet of code found in the file purchaseItem.ts , located in the setup/src/functions directory.

Apart from specifying the item and its type, the buyer must set the specific price and the seller's kiosk ID to execute the purchase transaction successfully, accomplished by running npm run call purchase .

Join

When you execute the command npm run call join , two specified tokenized assets of the FT type are merged together. Before running the command, make sure that the fields FT1 and FT2 within your .env file are populated with the objects you intend to merge.

Burn

When you intend to burn a tokenized asset, execute the command npm run call burn . Following this action, the specified asset is destroyed. Before running the command, make sure that the field TOKENIZED_ASSET within your .env file is populated with the object you intend to burn.

Get Balance

By executing the command npm run call get-balance , you can retrieve the balance value associated with the specified tokenized asset.

Get Supply

By executing the command npm run call get-supply , you can retrieve the value representing the current circulating supply of the asset.

Get Total Supply

By executing the command npm run call get-total-supply , you can retrieve the value representing the current circulating supply of the asset.