NFT Rental Example

NFT renting is a mechanism that allows individuals without ownership or possession of a specific NFT to temporarily utilize or experience it. The implementation of this process leverages the Kiosk Apps standard to establish an infrastructure for rental transactions. This approach closely aligns with the Ethereum ERC-4907 renting standard, making it a suitable choice for Solidity-based use cases intended for implementation on Sui.

The NFT Rental example satisfies the following project requirements:

Some use cases for real-world NFT rental example include:

There are multiple cases in gaming where renting NFTs can be beneficial to user experience:

In the realm of ticketing, NFTs play a pivotal role in enhancing transferability. These digital assets facilitate a secure and traceable transfer, resale, or rental of tickets, mitigating the risk of counterfeit tickets within the secondary market. The blockchain-based nature of NFTs ensures transparency and authenticity in each transaction, providing users with a reliable and fraud-resistant means to engage in ticket-related activities. This innovation not only simplifies the process for ticket holders but also contributes to a more trustworthy and efficient secondary ticket market.

Renting virtual lands and offices in the metaverse provides businesses with flexible solutions, enabling event companies to host gatherings without the commitment of permanent acquisitions and facilitating remote work through virtual offices. This approach not only offers cost-effective alternatives but also aligns with the evolving dynamics of digital business operations.

Temporary assets and subscriptions are notable applications of rental NFTs, offering accessibility to virtual experiences like high-end virtual casinos or curated digital fashion. These NFTs cater to diverse budgets, broadening audience reach. Subscription rentals extend to pools of digital assets, allowing users to pay monthly for a set number of items, fostering accessibility, user retention, and acquisition. Holders can rent out unused subscriptions, ensuring no loss for them, potential customer gains for the protocol, and a commitment-free trial for temporary holders. This showcases the adaptability and user-centric appeal of rental NFTs in diverse scenarios.

Transferring kiosks might result in unexpected behaviors while an asset is being rented. If you want to disallow kiosk transferring all together, consider using personal kiosks.

The rental smart contract uses the Kiosk Apps standard. Both the lender and borrower must install a Kiosk extension to take part, and the creator of the borrowed asset type must create a rental policy and ProtectedTP object to allow the extension to manage rentals while enforcing royalties.

This implementation is charging a rental fee based on days. You can re-purpose and update the logic to support charging per hour, or even seconds.

The NFT Rental example uses a single module, nft_rental.move . You can find the source for this file hosted in the sui repository in the examples directory. The source code includes extensive comments to help you follow the example's logic and structure.

The nft_rental module provides an API that facilitates lending or borrowing through the following operations:

The object model of the nft_rental module provides the structure of the app, beginning with the Rentables object. The struct has only the drop ability and acts as the extension key for the kiosk Rentables extension.

The Rented struct represents a rented item. The only field the struct includes is the ID of the object. It is used as the dynamic field key in the borrower's Bag entry when someone is actively borrowing an item. The struct has store , copy , and drop abilities because they are necessary for all dynamic field keys.

The Listed struct represents a listed item. The only field the struct includes is the ID of the object. It is used as the dynamic field key in the renter's Bag entry after an item is listed for renting. Like Rented , this struct has store , copy , and drop abilities because they are necessary for all dynamic field keys.

The Promise struct is created for borrowing by value. The Promise operates as the hot potato (a struct that has no capabilities that you can only pack and unpack in its module) that can only be resolved by returning the item back to the extension's Bag .

The Promise field lacks the store ability as it shouldn't be wrapped inside other objects. It also lacks the drop ability because only the return_val function can consume it.

The Rentable struct is as a wrapper object that holds an asset that is being rented. Contains information relevant to the rental period,

cost, and renter. This struct requires the store ability because it stores a value T that definitely also has store .

The RentalPolicy struct is a shared object that every creator mints. The struct defines the royalties the creator receives from each rent invocation.

The ProtectedTP object is a shared object that creators mint to enable renting. The object provides authorized access to an empty TransferPolicy . This is in part required because of the restrictions that Kiosk imposes around royalty enforced items and their tradability. Additionally it allows the rental module to operate within the Extension framework while maintaining the guarantee that the assets handled will always be tradable.

A protected empty transfer policy is required to facilitate the rental process so that the extension can transfer the asset without any additional rules to resolve (like lock rule, loyalty rule, and so on). If creators want to enforce royalties on rentals, they can use the RentalPolicy detailed previously.

The NFT Rental example includes the following functions that define the project's logic.

The install function enables installation of the Rentables extension in a kiosk. The party facilitating the rental process is responsible for making sure that the user installs the extension in their kiosk.

The remove function enables the owner (and only the owner) of the kiosk to remove the extension. The extension storage must be empty for the transaction to succeed. The extension storage empties after the user is no longer borrowing or renting any items. The kiosk_extension::remove function performs the ownership check before executing.

The setup_renting function mints and shares a ProtectedTP and a RentalPolicy object for type T . The publisher of type T is the only entity that can perform the action.

The list function enables listing of an asset within the Rentables extension's bag, creating a bag entry with the asset's ID as the key and a Rentable wrapper object as the value. Requires the existence of a ProtectedTP transfer policy that only the creator of type T can create. The function assumes an item is already placed (and optionally locked) in a kiosk.

The delist function allows the renter to delist an item, as long as it's not currently being rented. The function also places (or locks, if a lock rule is present) the object back to owner's kiosk. You should mint an empty TransferPolicy even if you don't want to apply any royalties. If at some point you do want to enforce royalties, you can always update the existing TransferPolicy .

The rent function enables renting a listed Rentable . It permits anyone to borrow an item on behalf of another user, provided they have the Rentables extension installed. The rental_policy defines the portion of the coin that is retained as fees and added to the rental policy's balance.

The borrow function enables the borrower to acquire the Rentable by reference from their bag.

The borrow_val function enables the borrower to temporarily acquire the Rentable with an agreement or promise to return it. The Promise stores all the information about the Rentable , facilitating the reconstruction of the Rentable upon object return.

The return_val function enables the borrower to return the borrowed item.

The reclaim functionality is manually invoked and the rental service provider is responsible for ensuring that the renter is reminded to reclaim . As such, this can cause the borrower to hold the asset for longer than the rental period. This can be mitigated through modification of the current contract by adding an assertion in the borrow and borrow_val functions to check if the rental period has expired.

The reclaim function enables an owner to claim back their asset after the rental period is over and place it inside their kiosk. If a lock rule is present, the example also locks the item inside the owner kiosk.

This implementation assumes that each creator, as an enabling action, creates a TransferPolicy even if empty, so that the Rentables extension can operate. This is a requirement in addition to invoking the setup_renting method.

The initialization process is part of the flow but only happens once for each entity:

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.

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

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

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

Use cases

Some use cases for real-world NFT rental example include:

There are multiple cases in gaming where renting NFTs can be beneficial to user experience:

In the realm of ticketing, NFTs play a pivotal role in enhancing transferability. These digital assets facilitate a secure and traceable transfer, resale, or rental of tickets, mitigating the risk of counterfeit tickets within the secondary market. The blockchain-based nature of NFTs ensures transparency and authenticity in each transaction, providing users with a reliable and fraud-resistant means to engage in ticket-related activities. This innovation not only simplifies the process for ticket holders but also contributes to a more trustworthy and efficient secondary ticket market.

Renting virtual lands and offices in the metaverse provides businesses with flexible solutions, enabling event companies to host gatherings without the commitment of permanent acquisitions and facilitating remote work through virtual offices. This approach not only offers cost-effective alternatives but also aligns with the evolving dynamics of digital business operations.

Temporary assets and subscriptions are notable applications of rental NFTs, offering accessibility to virtual experiences like high-end virtual casinos or curated digital fashion. These NFTs cater to diverse budgets, broadening audience reach. Subscription rentals extend to pools of digital assets, allowing users to pay monthly for a set number of items, fostering accessibility, user retention, and acquisition. Holders can rent out unused subscriptions, ensuring no loss for them, potential customer gains for the protocol, and a commitment-free trial for temporary holders. This showcases the adaptability and user-centric appeal of rental NFTs in diverse scenarios.

Transferring kiosks might result in unexpected behaviors while an asset is being rented. If you want to disallow kiosk transferring all together, consider using personal kiosks.

The rental smart contract uses the Kiosk Apps standard. Both the lender and borrower must install a Kiosk extension to take part, and the creator of the borrowed asset type must create a rental policy and ProtectedTP object to allow the extension to manage rentals while enforcing royalties.

This implementation is charging a rental fee based on days. You can re-purpose and update the logic to support charging per hour, or even seconds.

The NFT Rental example uses a single module, nft_rental.move . You can find the source for this file hosted in the sui repository in the examples directory. The source code includes extensive comments to help you follow the example's logic and structure.

The nft_rental module provides an API that facilitates lending or borrowing through the following operations:

The object model of the nft_rental module provides the structure of the app, beginning with the Rentables object. The struct has only the drop ability and acts as the extension key for the kiosk Rentables extension.

The Rented struct represents a rented item. The only field the struct includes is the ID of the object. It is used as the dynamic field key in the borrower's Bag entry when someone is actively borrowing an item. The struct has store , copy , and drop abilities because they are necessary for all dynamic field keys.

The Listed struct represents a listed item. The only field the struct includes is the ID of the object. It is used as the dynamic field key in the renter's Bag entry after an item is listed for renting. Like Rented , this struct has store , copy , and drop abilities because they are necessary for all dynamic field keys.

The Promise struct is created for borrowing by value. The Promise operates as the hot potato (a struct that has no capabilities that you can only pack and unpack in its module) that can only be resolved by returning the item back to the extension's Bag .

The Promise field lacks the store ability as it shouldn't be wrapped inside other objects. It also lacks the drop ability because only the return_val function can consume it.

The Rentable struct is as a wrapper object that holds an asset that is being rented. Contains information relevant to the rental period, cost, and renter. This struct requires the store ability because it stores a value T that definitely also has store .

The RentalPolicy struct is a shared object that every creator mints. The struct defines the royalties the creator receives from each rent invocation.

The ProtectedTP object is a shared object that creators mint to enable renting. The object provides authorized access to an empty TransferPolicy . This is in part required because of the restrictions that Kiosk imposes around royalty enforced items and their tradability. Additionally it allows the rental module to operate within the Extension framework while maintaining the guarantee that the assets handled will always be tradable.

A protected empty transfer policy is required to facilitate the rental process so that the extension can transfer the asset without any additional rules to resolve (like lock rule, loyalty rule, and so on). If creators want to enforce royalties on rentals, they can use the RentalPolicy detailed previously.

The NFT Rental example includes the following functions that define the project's logic.

The install function enables installation of the Rentables extension in a kiosk. The party facilitating the rental process is responsible for making sure that the user installs the extension in their kiosk.

The remove function enables the owner (and only the owner) of the kiosk to remove the extension. The extension storage must be empty for the transaction to succeed. The extension storage empties after the user is no longer borrowing or renting any items. The kiosk_extension::remove function performs the ownership check before executing.

The setup_renting function mints and shares a ProtectedTP and a RentalPolicy object for type T . The publisher of type T is the only entity that can perform the action.

The list function enables listing of an asset within the Rentables extension's bag, creating a bag entry with the asset's ID as the key and a Rentable wrapper object as the value. Requires the existence of a ProtectedTP transfer policy that only the creator of type T can create. The function assumes an item is already placed (and optionally locked) in a kiosk.

The delist function allows the renter to delist an item, as long as it's not currently being rented. The function also places (or locks, if a lock rule is present) the object back to owner's kiosk. You should mint an empty TransferPolicy even if you don't want to apply any royalties. If at some point you do want to enforce royalties, you can always update the existing TransferPolicy .

The rent function enables renting a listed Rentable . It permits anyone to borrow an item on behalf of another user, provided they have the Rentables extension installed. The rental_policy defines the portion of the coin that is retained as fees and added to the rental policy's balance.

The borrow function enables the borrower to acquire the Rentable by reference from their bag.

The borrow_val function enables the borrower to temporarily acquire the Rentable with an agreement or promise to return it. The Promise stores all the information about the Rentable , facilitating the reconstruction of the Rentable upon object return.

The return_val function enables the borrower to return the borrowed item.

The reclaim functionality is manually invoked and the rental service provider is responsible for ensuring that the renter is reminded to reclaim . As such, this can cause the borrower to hold the asset for longer than the rental period. This can be mitigated through modification of the current contract by adding an assertion in the borrow and borrow_val functions to check if the rental period has expired.

The reclaim function enables an owner to claim back their asset after the rental period is over and place it inside their kiosk. If a lock rule is present, the example also locks the item inside the owner kiosk.

This implementation assumes that each creator, as an enabling action, creates a TransferPolicy even if empty, so that the Rentables extension can operate. This is a requirement in addition to invoking the setup_renting method.

The initialization process is part of the flow but only happens once for each entity:

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.

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

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

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

Smart contract design

Transferring kiosks might result in unexpected behaviors while an asset is being rented. If you want to disallow kiosk transferring all together, consider using personal kiosks.

The rental smart contract uses the Kiosk Apps standard. Both the lender and borrower must install a Kiosk extension to take part, and the creator of the borrowed asset type must create a rental policy and ProtectedTP object to allow the extension to manage rentals while enforcing royalties.

This implementation is charging a rental fee based on days. You can re-purpose and update the logic to support charging per hour, or even seconds.

The NFT Rental example uses a single module, nft_rental.move . You can find the source for this file hosted in the sui repository in the examples directory. The source code includes extensive comments to help you follow the example's logic and structure.

The nft_rental module provides an API that facilitates lending or borrowing through the following operations:

The object model of the nft_rental module provides the structure of the app, beginning with the Rentables object. The struct has only the drop ability and acts as the extension key for the kiosk Rentables extension.

The Rented struct represents a rented item. The only field the struct includes is the ID of the object. It is used as the dynamic field key in the borrower's Bag entry when someone is actively borrowing an item. The struct has store , copy , and drop abilities because they are necessary for all dynamic field keys.

The Listed struct represents a listed item. The only field the struct includes is the ID of the object. It is used as the dynamic field key in the renter's Bag entry after an item is listed for renting. Like Rented , this struct has store , copy , and drop abilities because they are necessary for all dynamic field keys.

The Promise struct is created for borrowing by value. The Promise operates as the hot potato (a struct that has no capabilities that you can only pack and unpack in its module) that can only be resolved by returning the item back to the extension's Bag .

The Promise field lacks the store ability as it shouldn't be wrapped inside other objects. It also lacks the drop ability because only the return_val function can consume it.

The Rentable struct is as a wrapper object that holds an asset that is being rented. Contains information relevant to the rental period, cost, and renter. This struct requires the store ability because it stores a value T that definitely also has store .

The RentalPolicy struct is a shared object that every creator mints. The struct defines the royalties the creator receives from each rent invocation.

The ProtectedTP object is a shared object that creators mint to enable renting. The object provides authorized access to an empty TransferPolicy . This is in part required because of the restrictions that Kiosk imposes around royalty enforced items and their tradability. Additionally it allows the rental module to operate within the Extension framework while maintaining the guarantee that the assets handled will always be tradable.

A protected empty transfer policy is required to facilitate the rental process so that the extension can transfer the asset without any additional rules to resolve (like lock rule, loyalty rule, and so on). If creators want to enforce royalties on rentals, they can use the

RentalPolicy detailed previously.

The NFT Rental example includes the following functions that define the project's logic.

The install function enables installation of the Rentables extension in a kiosk. The party facilitating the rental process is responsible for making sure that the user installs the extension in their kiosk.

The remove function enables the owner (and only the owner) of the kiosk to remove the extension. The extension storage must be empty for the transaction to succeed. The extension storage empties after the user is no longer borrowing or renting any items. The kiosk_extension::remove function performs the ownership check before executing.

The setup_renting function mints and shares a ProtectedTP and a RentalPolicy object for type T . The publisher of type T is the only entity that can perform the action.

The list function enables listing of an asset within the Rentables extension's bag, creating a bag entry with the asset's ID as the key and a Rentable wrapper object as the value. Requires the existence of a ProtectedTP transfer policy that only the creator of type T can create. The function assumes an item is already placed (and optionally locked) in a kiosk.

The delist function allows the renter to delist an item, as long as it's not currently being rented. The function also places (or locks, if a lock rule is present) the object back to owner's kiosk. You should mint an empty TransferPolicy even if you don't want to apply any royalties. If at some point you do want to enforce royalties, you can always update the existing TransferPolicy .

The rent function enables renting a listed Rentable . It permits anyone to borrow an item on behalf of another user, provided they have the Rentables extension installed. The rental_policy defines the portion of the coin that is retained as fees and added to the rental policy's balance.

The borrow function enables the borrower to acquire the Rentable by reference from their bag.

The borrow_val function enables the borrower to temporarily acquire the Rentable with an agreement or promise to return it. The Promise stores all the information about the Rentable , facilitating the reconstruction of the Rentable upon object return.

The return_val function enables the borrower to return the borrowed item.

The reclaim functionality is manually invoked and the rental service provider is responsible for ensuring that the renter is reminded to reclaim . As such, this can cause the borrower to hold the asset for longer than the rental period. This can be mitigated through modification of the current contract by adding an assertion in the borrow and borrow_val functions to check if the rental period has expired.

The reclaim function enables an owner to claim back their asset after the rental period is over and place it inside their kiosk. If a lock rule is present, the example also locks the item inside the owner kiosk.

This implementation assumes that each creator, as an enabling action, creates a TransferPolicy even if empty, so that the Rentables extension can operate. This is a requirement in addition to invoking the setup_renting method.

The initialization process is part of the flow but only happens once for each entity:

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.

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

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

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

Move modules

The NFT Rental example uses a single module, nft_rental.move . You can find the source for this file hosted in the sui repository in the examples directory. The source code includes extensive comments to help you follow the example's logic and structure.

The nft_rental module provides an API that facilitates lending or borrowing through the following operations:

The object model of the nft_rental module provides the structure of the app, beginning with the Rentables object. The struct has only the drop ability and acts as the extension key for the kiosk Rentables extension.

The Rented struct represents a rented item. The only field the struct includes is the ID of the object. It is used as the dynamic field key in the borrower's Bag entry when someone is actively borrowing an item. The struct has store , copy , and drop abilities because they are necessary for all dynamic field keys.

The Listed struct represents a listed item. The only field the struct includes is the ID of the object. It is used as the dynamic field key in the renter's Bag entry after an item is listed for renting. Like Rented , this struct has store , copy , and drop abilities because they are necessary for all dynamic field keys.

The Promise struct is created for borrowing by value. The Promise operates as the hot potato (a struct that has no capabilities that you can only pack and unpack in its module) that can only be resolved by returning the item back to the extension's Bag .

The Promise field lacks the store ability as it shouldn't be wrapped inside other objects. It also lacks the drop ability because only the return_val function can consume it.

The Rentable struct is as a wrapper object that holds an asset that is being rented. Contains information relevant to the rental period, cost, and renter. This struct requires the store ability because it stores a value T that definitely also has store .

The RentalPolicy struct is a shared object that every creator mints. The struct defines the royalties the creator receives from each rent invocation.

The ProtectedTP object is a shared object that creators mint to enable renting. The object provides authorized access to an empty TransferPolicy . This is in part required because of the restrictions that Kiosk imposes around royalty enforced items and their tradability. Additionally it allows the rental module to operate within the Extension framework while maintaining the guarantee that the assets handled will always be tradable.

A protected empty transfer policy is required to facilitate the rental process so that the extension can transfer the asset without any additional rules to resolve (like lock rule, loyalty rule, and so on). If creators want to enforce royalties on rentals, they can use the RentalPolicy detailed previously.

The NFT Rental example includes the following functions that define the project's logic.

The install function enables installation of the Rentables extension in a kiosk. The party facilitating the rental process is responsible for making sure that the user installs the extension in their kiosk.

The remove function enables the owner (and only the owner) of the kiosk to remove the extension. The extension storage must be empty for the transaction to succeed. The extension storage empties after the user is no longer borrowing or renting any items. The kiosk_extension::remove function performs the ownership check before executing.

The setup_renting function mints and shares a ProtectedTP and a RentalPolicy object for type T . The publisher of type T is the only entity that can perform the action.

The list function enables listing of an asset within the Rentables extension's bag, creating a bag entry with the asset's ID as the key and a Rentable wrapper object as the value. Requires the existence of a ProtectedTP transfer policy that only the creator of type T can create. The function assumes an item is already placed (and optionally locked) in a kiosk.

The delist function allows the renter to delist an item, as long as it's not currently being rented. The function also places (or locks, if a lock rule is present) the object back to owner's kiosk. You should mint an empty TransferPolicy even if you don't want to apply any royalties. If at some point you do want to enforce royalties, you can always update the existing TransferPolicy .

The rent function enables renting a listed Rentable . It permits anyone to borrow an item on behalf of another user, provided they have the Rentables extension installed. The rental_policy defines the portion of the coin that is retained as fees and added to the rental policy's balance.

The borrow function enables the borrower to acquire the Rentable by reference from their bag.

The borrow_val function enables the borrower to temporarily acquire the Rentable with an agreement or promise to return it. The Promise stores all the information about the Rentable , facilitating the reconstruction of the Rentable upon object return.

The return_val function enables the borrower to return the borrowed item.

The reclaim functionality is manually invoked and the rental service provider is responsible for ensuring that the renter is reminded to reclaim . As such, this can cause the borrower to hold the asset for longer than the rental period. This can be mitigated through modification of the current contract by adding an assertion in the borrow and borrow_val functions to check if the rental period has expired.

The reclaim function enables an owner to claim back their asset after the rental period is over and place it inside their kiosk. If a lock rule is present, the example also locks the item inside the owner kiosk.

This implementation assumes that each creator, as an enabling action, creates a TransferPolicy even if empty, so that the Rentables extension can operate. This is a requirement in addition to invoking the setup_renting method.

The initialization process is part of the flow but only happens once for each entity:

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.

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

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

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

Sequence diagrams

This implementation assumes that each creator, as an enabling action, creates a TransferPolicy even if empty, so that the Rentables extension can operate. This is a requirement in addition to invoking the setup_renting method.

The initialization process is part of the flow but only happens once for each entity:

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.

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

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

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

Deployment

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.

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

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

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

Related links