Wallet Standard

Browser extension wallets built for Sui use the Wallet standard . This is a cross-chain standard that defines how dApps can automatically discover and interact with wallets.

If you are building a wallet, the helper library @mysten/wallet-standard provides types and utilities to help get started.

The Wallet standard includes features to help build wallets.

Create a class that represents your wallet. Use the Wallet interface from @mysten/wallet-standard to help ensure your class adheres to the standard.

Features are standard methods consumers can use to interact with a wallet. To be listed in the Sui wallet adapter, you must implement the following features in your wallet:

Implement these features in your wallet class under the features property:

The last requirement of the wallet interface is to expose an accounts interface. This should expose all of the accounts that a connected dApp has access to. It can be empty prior to initiating a connection through the standard:connect feature.

The accounts use the ReadonlyWalletAccount class to construct an account matching the required interface.

After you have a compatible interface for your wallet, use the registerWallet function to register it.

The Wallet standard has been updated from its original design to better support changes in the Sui ecosystem. For example, the GraphQL service was introduced after Mainnet launched. The sui:signAndExecuteTransactionBlock feature is closely tied to the JSON RPC options and data structures, so its continued maintenance becomes increasingly difficult as the GraphQL service becomes more ubiquitous.

Consequently, the Wallet standard introduced the sui:signAndExecuteTransaction feature. The features of this method are more useful, regardless of which API you use to execute transactions. This usefulness comes at the expense of flexibility in what sui:signAndExecuteTransaction returns.

To solve this problem, use the sui:signTransaction feature to sign transactions, and leave transaction execution to the dApp. The dApp can query for additional data during execution using whichever API it chooses. This is consistent with the default @mysten/dapp-kit uses for the useSignAndExecuteTransaction hook, and enables dApps to take advantage of read-after-write consistency when interacting with the Full-node based JSON RPC.

The downside of this strategy is that wallets often use different RPC nodes than the dApp, and might not have indexed the previous transaction when executing multiple transactions in rapid succession. This leads to building transactions using stale data that fail upon execution.

To mitigate this, wallets can use the sui:reportTransactionEffects feature so that dApps can report the effects of transactions to the wallet. Transaction effects contain the updated versions and digests of any objects that a transaction uses or creates. By caching these values, wallets can build transactions without needing to resolve the most recent versions through an API call.

The @mysten/sui/transaction SDK exports the SerialTransactionExecutor class, which you can use to build transactions using an object cache. The class has a method to update its internal cache using the effects of a transaction.

Using the combination of sui:signTransaction and sui:reportTransactionEffects , dApps can use either API to execute transactions and query for any data the API exposes. The dApp can then report the effects of the transaction to the wallet, and the wallet can then execute transactions without running into issues caused by a lagging indexer.

The Wallet standard includes features to help your apps interact with wallets.

To query the installed wallets in a user's browser, use the get function of getWallets .

The return from this call (availableWallets in the previous code) is an array of Wallet types.

Use the Wallet.icon and Wallet.name attributes to display the wallet details on your web page.

The Wallet.accounts is an array of WalletAccount s. Each WalletAccount type has address and publicKey properties, which are most useful during development. This data fills and caches after connection.

Both the Wallet type and the WalletAccount type have a property called features . The main wallet functionality is found here. The mandatory features that wallets must implement are listed in the previous code.

Many wallets choose to omit some non-mandatory features or add some custom features, so be sure to check the relevant wallet documentation if you intend to integrate a specific wallet.

Connecting in the context of a wallet refers to a user that joins the web site for the first time and has to choose the wallet and addresses to use.

The feature that provides this functionality is called standard:connect . To connect using this feature, make the following call:

This call results in the wallet opening a pop-up dialog for the user to continue the connection process.

Similar to the connecting feature, the Wallet standard also includes standard:disconnect . The following example calls this feature:

Upon wallet connection, your app has the necessary information to execute transactions, such as address and method.

Construct the transaction separately with the @mysten/sui library and then sign it with the private key of the user. Use the sui:signTransaction feature to achieve this:

Similar to connections, this process opens a pop-up dialog for the user to either accept or decline the transaction. Upon accepting, the function returns an object in the form {bytes: String, signature: Uint8Array} . The bytes value is the b64 encoding of the transaction and the signature value is the transaction signature.

To execute the transaction, use SuiClient from @mysten/sui :

Your app then sends the transaction effects back to the wallet, which reports results to the user. The wallet expects the effects to be b64 encoded.

Many wallets abstract the above flow into one feature: sui:signAndExecuteTransaction . The required arguments for this feature are the raw transaction and the options with the desired information to be included in the response:

The wallet emits events on certain user actions that apps can listen to. These events allow your app to be responsive to user actions on their wallets.

The wallet standard only defines the change event that can apply to chains, features, or accounts.

To subscribe your apps to events with the following call:

This call returns a function that can be called to unsubscribe from listening to the events.

The callback is the handler that contains the logic to perform when the event fires. The input to the callback function is an object with the following type:

These values are all arrays containing the new or changed items. Consequently, every event populates only one array in most cases, the rest are empty.

Mysten Labs offers a bare bones scaffold for React-based applications called @mysten/dapp-kit . See the dApp Kit documentation for more informqtion.

Working with wallets

The Wallet standard includes features to help build wallets.

Create a class that represents your wallet. Use the Wallet interface from @mysten/wallet-standard to help ensure your class adheres to the standard.

Features are standard methods consumers can use to interact with a wallet. To be listed in the Sui wallet adapter, you must implement the following features in your wallet:

Implement these features in your wallet class under the features property:

The last requirement of the wallet interface is to expose an accounts interface. This should expose all of the accounts that a connected dApp has access to. It can be empty prior to initiating a connection through the standard:connect feature.

The accounts use the ReadonlyWalletAccount class to construct an account matching the required interface.

After you have a compatible interface for your wallet, use the registerWallet function to register it.

The Wallet standard has been updated from its original design to better support changes in the Sui ecosystem. For example, the GraphQL service was introduced after Mainnet launched. The sui:signAndExecuteTransactionBlock feature is closely tied to the JSON RPC options and data structures, so its continued maintenance becomes increasingly difficult as the GraphQL service becomes more ubiquitous.

Consequently, the Wallet standard introduced the sui:signAndExecuteTransaction feature. The features of this method are more useful, regardless of which API you use to execute transactions. This usefulness comes at the expense of flexibility in what sui:signAndExecuteTransaction returns.

To solve this problem, use the sui:signTransaction feature to sign transactions, and leave transaction execution to the dApp. The dApp can query for additional data during execution using whichever API it chooses. This is consistent with the default @mysten/dapp-kit uses for the useSignAndExecuteTransaction hook, and enables dApps to take advantage of read-after-write consistency when interacting with the Full-node based JSON RPC.

The downside of this strategy is that wallets often use different RPC nodes than the dApp, and might not have indexed the previous transaction when executing multiple transactions in rapid succession. This leads to building transactions using stale data that fail upon execution.

To mitigate this, wallets can use the sui:reportTransactionEffects feature so that dApps can report the effects of transactions to the wallet. Transaction effects contain the updated versions and digests of any objects that a transaction uses or creates. By caching these values, wallets can build transactions without needing to resolve the most recent versions through an API call.

The @mysten/sui/transaction SDK exports the SerialTransactionExecutor class, which you can use to build transactions using an object cache. The class has a method to update its internal cache using the effects of a transaction.

Using the combination of sui:signTransaction and sui:reportTransactionEffects , dApps can use either API to execute transactions and query for any data the API exposes. The dApp can then report the effects of the transaction to the wallet, and the wallet can then execute transactions without running into issues caused by a lagging indexer.

The Wallet standard includes features to help your apps interact with wallets.

To query the installed wallets in a user's browser, use the get function of getWallets .

The return from this call (availableWallets in the previous code) is an array of Wallet types.

Use the Wallet.icon and Wallet.name attributes to display the wallet details on your web page.

The Wallet.accounts is an array of WalletAccount s. Each WalletAccount type has address and publicKey properties, which are most useful during development. This data fills and caches after connection.

Both the Wallet type and the WalletAccount type have a property called features . The main wallet functionality is found here. The mandatory features that wallets must implement are listed in the previous code.

Many wallets choose to omit some non-mandatory features or add some custom features, so be sure to check the relevant wallet documentation if you intend to integrate a specific wallet.

Connecting in the context of a wallet refers to a user that joins the web site for the first time and has to choose the wallet and addresses to use.

The feature that provides this functionality is called standard:connect . To connect using this feature, make the following call:

This call results in the wallet opening a pop-up dialog for the user to continue the connection process.

Similar to the connecting feature, the Wallet standard also includes standard:disconnect . The following example calls this feature:

Upon wallet connection, your app has the necessary information to execute transactions, such as address and method.

Construct the transaction separately with the @mysten/sui library and then sign it with the private key of the user. Use the sui:signTransaction feature to achieve this:

Similar to connections, this process opens a pop-up dialog for the user to either accept or decline the transaction. Upon accepting,

the function returns an object in the form {bytes: String, signature: Uint8Array} . The bytes value is the b64 encoding of the transaction and the signature value is the transaction signature.

To execute the transaction, use SuiClient from @mysten/sui :

Your app then sends the transaction effects back to the wallet, which reports results to the user. The wallet expects the effects to be b64 encoded.

Many wallets abstract the above flow into one feature: sui:signAndExecuteTransaction . The required arguments for this feature are the raw transaction and the options with the desired information to be included in the response:

The wallet emits events on certain user actions that apps can listen to. These events allow your app to be responsive to user actions on their wallets.

The wallet standard only defines the change event that can apply to chains, features, or accounts.

To subscribe your apps to events with the following call:

This call returns a function that can be called to unsubscribe from listening to the events.

The callback is the handler that contains the logic to perform when the event fires. The input to the callback function is an object with the following type:

These values are all arrays containing the new or changed items. Consequently, every event populates only one array in most cases, the rest are empty.

Mysten Labs offers a bare bones scaffold for React-based applications called @mysten/dapp-kit . See the dApp Kit documentation for more informqtion.

Managing wallets

The Wallet standard includes features to help your apps interact with wallets.

To query the installed wallets in a user's browser, use the get function of getWallets .

The return from this call (availableWallets in the previous code) is an array of Wallet types.

Use the Wallet.icon and Wallet.name attributes to display the wallet details on your web page.

The Wallet.accounts is an array of WalletAccount s. Each WalletAccount type has address and publicKey properties, which are most useful during development. This data fills and caches after connection.

Both the Wallet type and the WalletAccount type have a property called features . The main wallet functionality is found here. The mandatory features that wallets must implement are listed in the previous code.

Many wallets choose to omit some non-mandatory features or add some custom features, so be sure to check the relevant wallet documentation if you intend to integrate a specific wallet.

Connecting in the context of a wallet refers to a user that joins the web site for the first time and has to choose the wallet and addresses to use.

The feature that provides this functionality is called standard:connect . To connect using this feature, make the following call:

This call results in the wallet opening a pop-up dialog for the user to continue the connection process.

Similar to the connecting feature, the Wallet standard also includes standard:disconnect . The following example calls this feature:

Upon wallet connection, your app has the necessary information to execute transactions, such as address and method.

Construct the transaction separately with the @mysten/sui library and then sign it with the private key of the user. Use the sui:signTransaction feature to achieve this:

Similar to connections, this process opens a pop-up dialog for the user to either accept or decline the transaction. Upon accepting, the function returns an object in the form {bytes: String, signature: Uint8Array} . The bytes value is the b64 encoding of the transaction and the signature value is the transaction signature.

To execute the transaction, use SuiClient from @mysten/sui :

Your app then sends the transaction effects back to the wallet, which reports results to the user. The wallet expects the effects to be b64 encoded.

Many wallets abstract the above flow into one feature: sui:signAndExecuteTransaction . The required arguments for this feature are the raw transaction and the options with the desired information to be included in the response:

The wallet emits events on certain user actions that apps can listen to. These events allow your app to be responsive to user actions on their wallets.

The wallet standard only defines the change event that can apply to chains, features, or accounts.

To subscribe your apps to events with the following call:

This call returns a function that can be called to unsubscribe from listening to the events.

The callback is the handler that contains the logic to perform when the event fires. The input to the callback function is an object with the following type:

These values are all arrays containing the new or changed items. Consequently, every event populates only one array in most cases, the rest are empty.

Mysten Labs offers a bare bones scaffold for React-based applications called @mysten/dapp-kit . See the dApp Kit documentation for more informqtion.