Distributed Counter

This guide is rated as basic .

You can expect basic guides to take 30-45 minutes of dedicated time. The length of time necessary to fully understand some of the concepts raised in this guide might increase this estimate.

This example walks you through building a basic distributed counter app, covering the full end-to-end flow of building your Sui Move module and connecting it to your React Sui dApp. The app allows users to create counters that anyone can increment, but only the owner can reset.

The guide is split into two parts:

Example source code

Before getting started, make sure you have:

To begin, create a new folder on your system titled react-e2e-counter to hold all your project files. You can name this directory differently, but the rest of the guide references this file structure. Inside that folder, create two more folders: move and src . Inside the move folder, create a counter directory. Finally, create a sources folder inside counter . Different projects have their own directory structure, but it's common to split code into functional groups to help with maintenance. See Write a Move Package to learn more about package structure and how to use the Sui CLI to scaffold a new project.

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

In this part of the guide, you write the Move contracts that create, increment, and reset counters.

To begin writing your smart contracts, create a file inside react-e2e-counter/move/counter named Move.toml and copy the following code into it. This is the package manifest file. If you want to learn more about the structure of the file, see Package Manifest in The Move Book.

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

To begin creating the smart contract that defines the on-chain counter, create a counter.move file inside your react-e2e-counter/move/counter folder. Define the module that holds your smart contract logic.

Add the Counter struct and elements described in the following sections to the module.

In the create function, a new Counter object is created and shared .

The increment function accepts a mutable reference to any shared Counter object and increments its value field.

The set_value function accepts a mutable reference to any shared Counter object, the value to set its value field, and the ctx which contains the sender of the transaction. The Counter owner is the only one that can run this function.

Learn more about taking object references as input

The final module should look like this

Your smart contract is complete. You should be able to run the sui move build command from react-e2e-counter/move/counter and receive a response similar to the following:

You always run sui move build at the same level as your Move.toml file. After a successful build, you now have a build folder inside react-e2e-counter/move/counter .

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

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

Enter y to proceed. You receive the following response:

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

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

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

Run the following command to activate the testnet environment:

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

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

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

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

Partial snippet of CLI deployment output.

Store the PackageID value you receive in your own response to connect to your frontend .

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

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

In this final part of the app example, you build a frontend (UI) that allows end users to create, increment, and reset Counter objects.

To skip building the frontend and test out your newly deployed package, create this example using the following template and follow the instructions in the template's README.md file:

Before getting started, make sure you have:

The UI design consists of two parts:

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

This app uses the react-spinners package for icons. Install it by running the following command:

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

Update the src/networkConfig.ts file to include the packageID constants.

You need a way to create a new Counter object.

Create src/CreateCounter.tsx and add the following code:

This component renders a button that enables the user to create a counter. Now, update your create function so that it calls the create function from your Move module.

Update the create function in the src/CreateCounter.tsx file:

The create function now creates a new Sui Transaction and calls the create function from your Move module. The PTB is then signed and executed via the useSignAndExecuteTransaction hook. The onCreated callback is called with the new counter's ID when the transaction is successful.

Now that your users can create counters, you need a way to route to them. Routing in a React app can be complex, but this example keeps it basic.

Set up your src/App.tsx file so that you render the CreateCounter component by default, and if you want to display a specific counter you can put its ID into the hash portion of the URL.

This sets up your app to read the hash from the URL, and get the counter's ID if the hash is a valid object ID. Then, if you have a counter ID, it renders a Counter (which you define in the next step). If you don't have a counter ID, then it renders the CreateCounter button from the previous step. When a counter is created, you update the URL, and set the counter ID.

Currently, the Counter component doesn't exist, so the app displays an empty page if you navigate to a counter ID.

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

The create counter button should look like this:

Create a new file: src/Counter.tsx .

For your counter, you want to display three elements:

Add the following code to your src/Counter.tsx file:

This snippet has a few new concepts to examine. It uses the useSuiClientQuery hook to make the getObject RPC call. This returns a data object representing your counter. dApp Kit doesn't know which fields your counter object has, so define a getCounterFields helper that gets the counter fields, and adds a type-cast so that you can access the expected value and owner fields in your component.

The code also adds an executeMoveCall function that still needs implementing. This works just like the create function you used to create the counter. Instead of using a callback prop like you did for CreateCounter , you can use the refetch provided by useSuiClientQuery to reload your Counter object after you've executed your PTB.

Update the executeMoveCall function in the src/Counter.tsx file:

Now that you have a Counter component, you need to update your App component to render it when you have a counter ID.

Update the src/App.tsx file to render the Counter component when you have a counter ID:

At this point, you have the complete app! Run it and ensure you can:

The Counter component should look like this:

What the guide teaches

Before getting started, make sure you have:

To begin, create a new folder on your system titled react-e2e-counter to hold all your project files. You can name this directory differently, but the rest of the guide references this file structure. Inside that folder, create two more folders: move and src . Inside the move folder, create a counter directory. Finally, create a sources folder inside counter . Different projects have their own directory structure, but it's common to split code into functional groups to help with maintenance. See Write a Move Package to learn more about package structure and how to use the Sui CLI to scaffold a new project.

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

In this part of the guide, you write the Move contracts that create, increment, and reset counters.

To begin writing your smart contracts, create a file inside react-e2e-counter/move/counter named Move.toml and copy the following code into it. This is the package manifest file. If you want to learn more about the structure of the file, see Package Manifest in The Move Book.

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

To begin creating the smart contract that defines the on-chain counter, create a counter.move file inside your react-e2e-counter/move/counter folder. Define the module that holds your smart contract logic.

Add the Counter struct and elements described in the following sections to the module.

In the create function, a new Counter object is created and shared .

The increment function accepts a mutable reference to any shared Counter object and increments its value field.

The set_value function accepts a mutable reference to any shared Counter object, the value to set its value field, and the ctx which contains the sender of the transaction. The Counter owner is the only one that can run this function.

Learn more about taking object references as input

The final module should look like this

Your smart contract is complete. You should be able to run the sui move build command from react-e2e-counter/move/counter and receive a response similar to the following:

You always run sui move build at the same level as your Move.toml file. After a successful build, you now have a build folder inside react-e2e-counter/move/counter .

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

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

Enter y to proceed. You receive the following response:

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

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

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

Run the following command to activate the testnet environment:

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

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

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

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

Partial snippet of CLI deployment output.

Store the PackageID value you receive in your own response to connect to your frontend .

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

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

In this final part of the app example, you build a frontend (UI) that allows end users to create, increment, and reset Counter objects.

To skip building the frontend and test out your newly deployed package, create this example using the following template and follow the instructions in the template's README.md file:

Before getting started, make sure you have:

The UI design consists of two parts:

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

This app uses the react-spinners package for icons. Install it by running the following command:

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

Update the src/networkConfig.ts file to include the packageID constants.

You need a way to create a new Counter object.

Create src/CreateCounter.tsx and add the following code:

This component renders a button that enables the user to create a counter. Now, update your create function so that it calls the create function from your Move module.

Update the create function in the src/CreateCounter.tsx file:

The create function now creates a new Sui Transaction and calls the create function from your Move module. The PTB is then signed and executed via the useSignAndExecuteTransaction hook. The onCreated callback is called with the new counter's ID when the transaction is successful.

Now that your users can create counters, you need a way to route to them. Routing in a React app can be complex, but this example keeps it basic.

Set up your src/App.tsx file so that you render the CreateCounter component by default, and if you want to display a specific counter you can put its ID into the hash portion of the URL.

This sets up your app to read the hash from the URL, and get the counter's ID if the hash is a valid object ID. Then, if you have a counter ID, it renders a Counter (which you define in the next step). If you don't have a counter ID, then it renders the CreateCounter button from the previous step. When a counter is created, you update the URL, and set the counter ID.

Currently, the Counter component doesn't exist, so the app displays an empty page if you navigate to a counter ID.

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

The create counter button should look like this:

Create a new file: src/Counter.tsx .

For your counter, you want to display three elements:

Add the following code to your src/Counter.tsx file:

This snippet has a few new concepts to examine. It uses the useSuiClientQuery hook to make the getObject RPC call. This returns a data object representing your counter. dApp Kit doesn't know which fields your counter object has, so define a getCounterFields helper that gets the counter fields, and adds a type-cast so that you can access the expected value and owner fields in your component.

The code also adds an executeMoveCall function that still needs implementing. This works just like the create function you used to create the counter. Instead of using a callback prop like you did for CreateCounter , you can use the refetch provided by useSuiClientQuery to reload your Counter object after you've executed your PTB.

Update the executeMoveCall function in the src/Counter.tsx file:

Now that you have a Counter component, you need to update your App component to render it when you have a counter ID.

Update the src/App.tsx file to render the Counter component when you have a counter ID:

At this point, you have the complete app! Run it and ensure you can:

The Counter component should look like this:

What you need

Before getting started, make sure you have:

To begin, create a new folder on your system titled react-e2e-counter to hold all your project files. You can name this directory differently, but the rest of the guide references this file structure. Inside that folder, create two more folders: move and src . Inside the move folder, create a counter directory. Finally, create a sources folder inside counter . Different projects have their own directory structure, but it's common to split code into functional groups to help with maintenance. See Write a Move Package to learn more about package structure and how to use the Sui CLI to scaffold a new project.

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

In this part of the guide, you write the Move contracts that create, increment, and reset counters.

To begin writing your smart contracts, create a file inside react-e2e-counter/move/counter named Move.toml and copy the following code into it. This is the package manifest file. If you want to learn more about the structure of the file, see Package Manifest in The Move Book.

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

To begin creating the smart contract that defines the on-chain counter, create a counter.move file inside your react-e2e-counter/move/counter folder. Define the module that holds your smart contract logic.

Add the Counter struct and elements described in the following sections to the module.

In the create function, a new Counter object is created and shared .

The increment function accepts a mutable reference to any shared Counter object and increments its value field.

The set_value function accepts a mutable reference to any shared Counter object, the value to set its value field, and the ctx which contains the sender of the transaction. The Counter owner is the only one that can run this function.

Learn more about taking object references as input

The final module should look like this

Your smart contract is complete. You should be able to run the sui move build command from react-e2e-counter/move/counter and receive a response similar to the following:

You always run sui move build at the same level as your Move.toml file. After a successful build, you now have a build folder inside react-e2e-counter/move/counter .

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

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

Enter y to proceed. You receive the following response:

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

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

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

Run the following command to activate the testnet environment:

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

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

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

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

Partial snippet of CLI deployment output.

Store the PackageID value you receive in your own response to connect to your frontend .

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

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

In this final part of the app example, you build a frontend (UI) that allows end users to create, increment, and reset Counter objects.

To skip building the frontend and test out your newly deployed package, create this example using the following template and follow the instructions in the template's README.md file:

Before getting started, make sure you have:

The UI design consists of two parts:

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

This app uses the react-spinners package for icons. Install it by running the following command:

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

Update the src/networkConfig.ts file to include the packageID constants.

You need a way to create a new Counter object.

Create src/CreateCounter.tsx and add the following code:

This component renders a button that enables the user to create a counter. Now, update your create function so that it calls the create function from your Move module.

Update the create function in the src/CreateCounter.tsx file:

The create function now creates a new Sui Transaction and calls the create function from your Move module. The PTB is then signed and executed via the useSignAndExecuteTransaction hook. The onCreated callback is called with the new counter's ID when the transaction is successful.

Now that your users can create counters, you need a way to route to them. Routing in a React app can be complex, but this example keeps it basic.

Set up your src/App.tsx file so that you render the CreateCounter component by default, and if you want to display a specific counter you can put its ID into the hash portion of the URL.

This sets up your app to read the hash from the URL, and get the counter's ID if the hash is a valid object ID. Then, if you have a counter ID, it renders a Counter (which you define in the next step). If you don't have a counter ID, then it renders the CreateCounter button from the previous step. When a counter is created, you update the URL, and set the counter ID.

Currently, the Counter component doesn't exist, so the app displays an empty page if you navigate to a counter ID.

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

The create counter button should look like this:

Create a new file: src/Counter.tsx .

For your counter, you want to display three elements:

Add the following code to your src/Counter.tsx file:

This snippet has a few new concepts to examine. It uses the useSuiClientQuery hook to make the getObject RPC call. This returns a data object representing your counter. dApp Kit doesn't know which fields your counter object has, so define a getCounterFields helper that gets the counter fields, and adds a type-cast so that you can access the expected value and owner fields in your component.

The code also adds an executeMoveCall function that still needs implementing. This works just like the create function you used to create the counter. Instead of using a callback prop like you did for CreateCounter , you can use the refetch provided by useSuiClientQuery to reload your Counter object after you've executed your PTB.

Update the executeMoveCall function in the src/Counter.tsx file:

Now that you have a Counter component, you need to update your App component to render it when you have a counter ID.

Update the src/App.tsx file to render the Counter component when you have a counter ID:

At this point, you have the complete app! Run it and ensure you can:

The Counter component should look like this:

Directory structure

To begin, create a new folder on your system titled react-e2e-counter to hold all your project files. You can name this directory differently, but the rest of the guide references this file structure. Inside that folder, create two more folders: move and src . Inside the move folder, create a counter directory. Finally, create a sources folder inside counter . Different projects have their own directory structure, but it's common to split code into functional groups to help with maintenance. See Write a Move Package to learn more about package structure and how to use the Sui CLI to scaffold a new project.

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

In this part of the guide, you write the Move contracts that create, increment, and reset counters.

To begin writing your smart contracts, create a file inside react-e2e-counter/move/counter named Move.toml and copy the following code into it. This is the package manifest file. If you want to learn more about the structure of the file, see Package Manifest in The Move Book.

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

To begin creating the smart contract that defines the on-chain counter, create a counter.move file inside your react-e2e-counter/move/counter folder. Define the module that holds your smart contract logic.

Add the Counter struct and elements described in the following sections to the module.

In the create function, a new Counter object is created and shared .

The increment function accepts a mutable reference to any shared Counter object and increments its value field.

The set_value function accepts a mutable reference to any shared Counter object, the value to set its value field, and the ctx which contains the sender of the transaction. The Counter owner is the only one that can run this function.

Learn more about taking object references as input

The final module should look like this

Your smart contract is complete. You should be able to run the sui move build command from react-e2e-counter/move/counter and receive a response similar to the following:

You always run sui move build at the same level as your Move.toml file. After a successful build, you now have a build folder inside react-e2e-counter/move/counter .

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

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

Enter y to proceed. You receive the following response:

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

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

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

Run the following command to activate the testnet environment:

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

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

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

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

Partial snippet of CLI deployment output.

Store the PackageID value you receive in your own response to connect to your frontend .

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

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

In this final part of the app example, you build a frontend (UI) that allows end users to create, increment, and reset Counter objects.

To skip building the frontend and test out your newly deployed package, create this example using the following template and follow the instructions in the template's README.md file:

Before getting started, make sure you have:

The UI design consists of two parts:

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

This app uses the react-spinners package for icons. Install it by running the following command:

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

Update the src/networkConfig.ts file to include the packageID constants.

You need a way to create a new Counter object.

Create src/CreateCounter.tsx and add the following code:

This component renders a button that enables the user to create a counter. Now, update your create function so that it calls the create function from your Move module.

Update the create function in the src/CreateCounter.tsx file:

The create function now creates a new Sui Transaction and calls the create function from your Move module. The PTB is then signed and executed via the useSignAndExecuteTransaction hook. The onCreated callback is called with the new counter's ID when the transaction is successful.

Now that your users can create counters, you need a way to route to them. Routing in a React app can be complex, but this example keeps it basic.

Set up your src/App.tsx file so that you render the CreateCounter component by default, and if you want to display a specific counter you can put its ID into the hash portion of the URL.

This sets up your app to read the hash from the URL, and get the counter's ID if the hash is a valid object ID. Then, if you have a counter ID, it renders a Counter (which you define in the next step). If you don't have a counter ID, then it renders the CreateCounter button from the previous step. When a counter is created, you update the URL, and set the counter ID.

Currently, the Counter component doesn't exist, so the app displays an empty page if you navigate to a counter ID.

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

The create counter button should look like this:

Create a new file: src/Counter.tsx .

For your counter, you want to display three elements:

Add the following code to your src/Counter.tsx file:

This snippet has a few new concepts to examine. It uses the useSuiClientQuery hook to make the getObject RPC call. This returns a data object representing your counter. dApp Kit doesn't know which fields your counter object has, so define a getCounterFields helper that gets the counter fields, and adds a type-cast so that you can access the expected value and owner fields in your component.

The code also adds an executeMoveCall function that still needs implementing. This works just like the create function you used to

create the counter. Instead of using a callback prop like you did for CreateCounter , you can use the refetch provided by useSuiClientQuery to reload your Counter object after you've executed your PTB.

Update the executeMoveCall function in the src/Counter.tsx file:

Now that you have a Counter component, you need to update your App component to render it when you have a counter ID.

Update the src/App.tsx file to render the Counter component when you have a counter ID:

At this point, you have the complete app! Run it and ensure you can:

The Counter component should look like this:

Smart contracts

In this part of the guide, you write the Move contracts that create, increment, and reset counters.

To begin writing your smart contracts, create a file inside react-e2e-counter/move/counter named Move.toml and copy the following code into it. This is the package manifest file. If you want to learn more about the structure of the file, see Package Manifest in The Move Book.

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

To begin creating the smart contract that defines the on-chain counter, create a counter.move file inside your react-e2e-counter/move/counter folder. Define the module that holds your smart contract logic.

Add the Counter struct and elements described in the following sections to the module.

In the create function, a new Counter object is created and shared .

The increment function accepts a mutable reference to any shared Counter object and increments its value field.

The set_value function accepts a mutable reference to any shared Counter object, the value to set its value field, and the ctx which contains the sender of the transaction. The Counter owner is the only one that can run this function.

Learn more about taking object references as input

The final module should look like this

Your smart contract is complete. You should be able to run the sui move build command from react-e2e-counter/move/counter and receive a response similar to the following:

You always run sui move build at the same level as your Move.toml file. After a successful build, you now have a build folder inside react-e2e-counter/move/counter .

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

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

Enter y to proceed. You receive the following response:

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

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

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

Run the following command to activate the testnet environment:

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

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

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

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

Partial snippet of CLI deployment output.

Store the PackageID value you receive in your own response to connect to your frontend .

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

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

In this final part of the app example, you build a frontend (UI) that allows end users to create, increment, and reset Counter objects.

To skip building the frontend and test out your newly deployed package, create this example using the following template and follow the instructions in the template's README.md file:

Before getting started, make sure you have:

The UI design consists of two parts:

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

This app uses the react-spinners package for icons. Install it by running the following command:

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

Update the src/networkConfig.ts file to include the packageID constants.

You need a way to create a new Counter object.

Create src/CreateCounter.tsx and add the following code:

This component renders a button that enables the user to create a counter. Now, update your create function so that it calls the create function from your Move module.

Update the create function in the src/CreateCounter.tsx file:

The create function now creates a new Sui Transaction and calls the create function from your Move module. The PTB is then signed and executed via the useSignAndExecuteTransaction hook. The onCreated callback is called with the new counter's ID when the transaction is successful.

Now that your users can create counters, you need a way to route to them. Routing in a React app can be complex, but this example keeps it basic.

Set up your src/App.tsx file so that you render the CreateCounter component by default, and if you want to display a specific counter you can put its ID into the hash portion of the URL.

This sets up your app to read the hash from the URL, and get the counter's ID if the hash is a valid object ID. Then, if you have a counter ID, it renders a Counter (which you define in the next step). If you don't have a counter ID, then it renders the CreateCounter button from the previous step. When a counter is created, you update the URL, and set the counter ID.

Currently, the Counter component doesn't exist, so the app displays an empty page if you navigate to a counter ID.

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

The create counter button should look like this:

Create a new file: src/Counter.tsx .

For your counter, you want to display three elements:

Add the following code to your src/Counter.tsx file:

This snippet has a few new concepts to examine. It uses the useSuiClientQuery hook to make the getObject RPC call. This returns a data object representing your counter. dApp Kit doesn't know which fields your counter object has, so define a getCounterFields helper that gets the counter fields, and adds a type-cast so that you can access the expected value and owner fields in your component.

The code also adds an executeMoveCall function that still needs implementing. This works just like the create function you used to create the counter. Instead of using a callback prop like you did for CreateCounter , you can use the refetch provided by useSuiClientQuery to reload your Counter object after you've executed your PTB.

Update the executeMoveCall function in the src/Counter.tsx file:

Now that you have a Counter component, you need to update your App component to render it when you have a counter ID.

Update the src/App.tsx file to render the Counter component when you have a counter ID:

At this point, you have the complete app! Run it and ensure you can:

The Counter component should look like this:

Finished package

The final module should look like this

Your smart contract is complete. You should be able to run the sui move build command from react-e2e-counter/move/counter and receive a response similar to the following:

You always run sui move build at the same level as your Move.toml file. After a successful build, you now have a build folder inside react-e2e-counter/move/counter .

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

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

Enter y to proceed. You receive the following response:

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

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

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

Run the following command to activate the testnet environment:

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

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

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

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

Partial snippet of CLI deployment output.

Store the PackageID value you receive in your own response to connect to your frontend .

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

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

In this final part of the app example, you build a frontend (UI) that allows end users to create, increment, and reset Counter objects.

To skip building the frontend and test out your newly deployed package, create this example using the following template and follow the instructions in the template's README.md file:

Before getting started, make sure you have:

The UI design consists of two parts:

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

This app uses the react-spinners package for icons. Install it by running the following command:

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

Update the src/networkConfig.ts file to include the packageID constants.

You need a way to create a new Counter object.

Create src/CreateCounter.tsx and add the following code:

This component renders a button that enables the user to create a counter. Now, update your create function so that it calls the create function from your Move module.

Update the create function in the src/CreateCounter.tsx file:

The create function now creates a new Sui Transaction and calls the create function from your Move module. The PTB is then signed and executed via the useSignAndExecuteTransaction hook. The onCreated callback is called with the new counter's ID when the transaction is successful.

Now that your users can create counters, you need a way to route to them. Routing in a React app can be complex, but this example keeps it basic.

Set up your src/App.tsx file so that you render the CreateCounter component by default, and if you want to display a specific counter you can put its ID into the hash portion of the URL.

This sets up your app to read the hash from the URL, and get the counter's ID if the hash is a valid object ID. Then, if you have a counter ID, it renders a Counter (which you define in the next step). If you don't have a counter ID, then it renders the CreateCounter button from the previous step. When a counter is created, you update the URL, and set the counter ID.

Currently, the Counter component doesn't exist, so the app displays an empty page if you navigate to a counter ID.

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

The create counter button should look like this:

Create a new file: src/Counter.tsx .

For your counter, you want to display three elements:

Add the following code to your src/Counter.tsx file:

This snippet has a few new concepts to examine. It uses the useSuiClientQuery hook to make the getObject RPC call. This returns a data object representing your counter. dApp Kit doesn't know which fields your counter object has, so define a getCounterFields helper that gets the counter fields, and adds a type-cast so that you can access the expected value and owner fields in your component.

The code also adds an executeMoveCall function that still needs implementing. This works just like the create function you used to create the counter. Instead of using a callback prop like you did for CreateCounter , you can use the refetch provided by useSuiClientQuery to reload your Counter object after you've executed your PTB.

Update the executeMoveCall function in the src/Counter.tsx file:

Now that you have a Counter component, you need to update your App component to render it when you have a counter ID.

Update the src/App.tsx file to render the Counter component when you have a counter ID:

At this point, you have the complete app! Run it and ensure you can:

The Counter component should look like this:

Frontend

In this final part of the app example, you build a frontend (UI) that allows end users to create, increment, and reset Counter objects.

To skip building the frontend and test out your newly deployed package, create this example using the following template and follow the instructions in the template's README.md file:

Before getting started, make sure you have:

The UI design consists of two parts:

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

This app uses the react-spinners package for icons. Install it by running the following command:

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

Update the src/networkConfig.ts file to include the packageID constants.

You need a way to create a new Counter object.

Create src/CreateCounter.tsx and add the following code:

This component renders a button that enables the user to create a counter. Now, update your create function so that it calls the create function from your Move module.

Update the create function in the src/CreateCounter.tsx file:

The create function now creates a new Sui Transaction and calls the create function from your Move module. The PTB is then signed and executed via the useSignAndExecuteTransaction hook. The onCreated callback is called with the new counter's ID when the transaction is successful.

Now that your users can create counters, you need a way to route to them. Routing in a React app can be complex, but this example keeps it basic.

Set up your src/App.tsx file so that you render the CreateCounter component by default, and if you want to display a specific counter you can put its ID into the hash portion of the URL.

This sets up your app to read the hash from the URL, and get the counter's ID if the hash is a valid object ID. Then, if you have a counter ID, it renders a Counter (which you define in the next step). If you don't have a counter ID, then it renders the CreateCounter button from the previous step. When a counter is created, you update the URL, and set the counter ID.

Currently, the Counter component doesn't exist, so the app displays an empty page if you navigate to a counter ID.

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

The create counter button should look like this:

Create a new file: src/Counter.tsx .

For your counter, you want to display three elements:

Add the following code to your src/Counter.tsx file:

This snippet has a few new concepts to examine. It uses the useSuiClientQuery hook to make the getObject RPC call. This returns a data object representing your counter. dApp Kit doesn't know which fields your counter object has, so define a getCounterFields helper that gets the counter fields, and adds a type-cast so that you can access the expected value and owner fields in your component.

The code also adds an executeMoveCall function that still needs implementing. This works just like the create function you used to create the counter. Instead of using a callback prop like you did for CreateCounter , you can use the refetch provided by useSuiClientQuery to reload your Counter object after you've executed your PTB.

Update the executeMoveCall function in the src/Counter.tsx file:

Now that you have a Counter component, you need to update your App component to render it when you have a counter ID.

Update the src/App.tsx file to render the Counter component when you have a counter ID:

At this point, you have the complete app! Run it and ensure you can:

The Counter component should look like this: