

react-celo

The easiest way to access [ContractKit](#) in your React applications.

react-celo

The easiest way to access [Celo](#) in your React applications [ReactHook](#) for managing access to Celo with a built-in headless modal system for connecting to your users wallet of choice.

Now your DApp can be made available to everyone in the Celo ecosystem, from Valora users to self custodied Ledger users.

By default react-celo is styled so that you can drop it into your application and go, however it's fully customisable so you can maintain a consistent UX throughout your application.

- [Github repo](#)
- [Demo page](#)

Table of Contents

- [Installation](#)
- [Supported Wallets](#)
- [Usage](#)
- [Notes](#)
- [Support](#)
- [Guides](#)

Install

```
yarn add @celo/react-celo @celo/contractkit
```

Supported wallets

- Celo Dance
- Celo Extension Wallet (Metamask fork)
- Celo Terminal
- Celo Wallet
- Ledger
- MetaMask
- Plaintext private key
- [Omni](#)
- Valora
- WalletConnect

Basic Usage

Wrap your application with CeloProvider

react-celo uses [React's Context.Provider](#) under the hood to inject state throughout your application. You need to make sure your application is wrapped with the provider in order to be able to access all the goodies react-celo provides.

```
import
{
  CeloProvider
}
from
"@celo/react-celo" ; import
"@celo/react-celo/lib/styles.css" ;
function
```

```

WrappedApp ( )

{ return

( < CeloProvider dapp = { { name :

"My awesome dApp" , description :

"My awesome description" , url :

"https://example.com" , } }

    < App

/

    < / CeloProvider

) ; }

function

App ( )

{ // your application code }

```

Default wallets and customization

react-celo provides a list of default wallets (CeloExtensionWallet, Injected, Ledger, MetaMask, PrivateKey (dev only) and WalletConnect). It can be configured as shown below.

```

< CeloProvider dapp = { { name :

"My awesome dApp" , description :

"My awesome description" , url :

"https://example.com" , } } connectModal = { { // This options changes the title of the modal and can be either a string or a
react element title :

< span

    Connect your Wallet < / span

, providersOptions :

{ // This option hides specific wallets from the default list hideFromDefaults :

[ SupportedProvider . MetaMask , SupportedProvider . PrivateKey , SupportedProvider . CeloExtensionWallet ,
SupportedProvider . Valora , ] ,

// This option hides all default wallets hideFromDefaults :

true ,

// This option toggles on and off the searchbar searchable :

true , } , } }

    < App

/

    < / CeloProvider

```

You can also add new custom wallets that don't exist in the registry or aren't in our defaults. For now, we only support custom wallets that implement the walletconnect protocol, but more may come in the future. In the example below, we're hiding all wallets except a new custom wallet.

```

< CeloProvider dapp = { { name :

"My awesome dApp" , description :

```

"My awesome description" , url :

"https://example.com" , } } // Use the theme to customize the colors. // If you provide a theme, you must provide all values below! theme = { { primary :

"#6366f1" , secondary :

"#eef2ff" , text :

"#000000" , textSecondary :

"#1f2937" , textTertiary :

"#64748b" , muted :

"#e2e8f0" , background :

"#ffffff" , error :

"#ef4444" , } } connectModal = { { title :

< span

Connect your ExampleWallet < / span

, providersOptions :

{ hideFromDefaults :

true , additionalWCWallets :

[// see <https://github.com/WalletConnect/walletconnect-registry/#schema> for a schema example { id :

"example-wallet" , name :

"Example Wallet" , description :

"Lorem ipsum" , homepage :

"https://example.com" , chains :

["eip:4220"] , // IMPORTANT // This is the version of WC. We only support version 1 at the moment. versions :

["1"] , logos :

{ sm :

"https://via.placeholder.com/40/000000/FFFFFF" , md :

"https://via.placeholder.com/80/000000/FFFFFF" , lg :

"https://via.placeholder.com/160/000000/FFFFFF" , } , app :

{ browser :

"..." , ios :

"..." , android :

"..." , mac :

"..." , windows :

"..." , linux :

"..." , } , mobile :

{ native :

"..." , universal :

"..." , } , desktop :

{ native :

```

"...", universal :
"...", }, }, metadata :
{ shortName :
"...", colors :
{ primary :
"...", secondary :
"...", }, }, }, responsive :
{ mobileFriendly :
true, browserFriendly :
true, mobileOnly :
false, browserOnly :
false, }, }, ], }, } }
      < App
/
      < / CeloProvider

```

Prompt users to connect their wallet

react-celo provides aconnect function that will open a modal with a list of wallets your user can connect to.

```

import
{ useCelo }
from
"@celo/react-celo" ;
function
App ( )
{ const
{ connect, address }
=
useCelo ( ) ;
return
( <
      { address ?
( < div
      Connected to { address } < / div
      )
:
( < button onClick = { connect }
      Connect wallet < / button
      ) } < /

```

); } After connecting to an account the `address` property will be set.

Use ContractKit to read chain data

Now that we've connected to an account and have the user's address, we can use the kit to query on-chain data:

```
import
{ useCelo }
from
'@celo/react-celo' ;

function
App ( )
{ const
{ kit , address }
=
useCelo ( ) ;
async
function
getAccountSummary ( )
{ const accounts =
await kit . contracts . getAccounts ( ) ; await accounts . getAccountSummary ( address ) ; }
return
( ... ) }
```

Accessing user accounts

The biggest problem when developing DApps is ensuring a Web2 level experience while managing the flaky and often slow nature of blockchains. To that end we've designed react-celo in a way to abstract away most of that pain.

Initially connecting to a user's account is one thing, handled via the `connect` function we just mentioned. However once a user has connected to your DApp we can make the experience nicer for them on repeat visits.

Last connected account

react-celo will remember a user's last connected address when they navigate back to or refresh your DApp. Ensure that when developing your DApp nothing changes in the UI whether or not the user has `akit.defaultAccount` property set.

```
import
{ useCelo }
from
"@celo/react-celo" ;
const
{ address }
=
useCelo ( ) ;
```

Get a connected account

When a user refreshes or navigates back to your page, they may not necessarily have a connected account any longer,

however we shouldn't need to prompt them to login again just to view the page, that can be done only when doing an action.

For that functionality we have the `performActions` and `getConnectedKit` methods. Usage looks a little like this

```
forgetConnectedKit :  
  
import  
  
{ useCelo }  
  
from  
  
"@celo/react-celo" ;  
  
function  
  
App ( )  
  
{ const  
  
{ getConnectedKit }  
  
=  
  
useCelo ( ) ;  
  
async  
  
function  
  
transfer ( )  
  
{ const kit =  
  
await  
  
getConnectedKit ( ) ; const cUSD =  
  
await kit . contracts . getStableToken ( ) ; await cUSD . transfer ( "0x..." ,  
  
10000 ) . sendAndWaitForReceipt ( ) ; }  
  
return  
  
< button onClick = { transfer }  
  
    Transfer < / button  
  
; } and this for performActions :
```

```
import  
  
{ useCelo }  
  
from  
  
"@celo/react-celo" ;  
  
function  
  
App ( )  
  
{ const  
  
{ performActions }  
  
=  
  
useCelo ( ) ;  
  
async  
  
function  
  
transfer ( )
```

```

{ await
performActions ( async
( kit )
=>
{ const cUSD =
await kit . contracts . getStableToken ( ) ; await cUSD . transfer ( "0x..." ,
10000 ) . sendAndWaitForReceipt ( ) ; } ) ; }
return
< button onClick = { transfer }

    Transfer < / button

; } TheperformActions method will also take care of displaying a modal to the user telling them to confirm any
actions on their connected wallet.

```

Network management

react-celo provides a `network` variable and an `updateNetwork` function you can use to display the currently connected network as well as switch to a different one (ie. Alfajores, Baklava or Mainnet).

If you'd prefer your DApp to only access a specific network (maybe you're deploying your testnet website at `https://test-app.dapp.name` and your mainnet version at `https://app.dapp.name`) you can pass the network you want to use as a variable into the provider you wrap your application with:

You can also pass in a `network` prop to the `CeloProvider` as the default starting network

walletChainId vs network

`useCelo` returns two properties. `network`, an object with a name, `chainId` and `rpc` url. it is which chain/network the dapp is communicating with. `walletChainId` is the chainID the wallet/signer is connected to (assuming it this is a concept that it has one) or will be null. unless Manual Networking mode is enabled these will eventually be consistent or `walletChainId` will be null.

```

import
{
CeloProvider ,
Alfajores ,
NetworkNames
}
from
'@celo/react-celo' ;
function
WrappedApp ( {
Component , pageProps } )
{ return
( < CeloProvider ... networks = { [ Alfajores ] } network = { { name :
NetworkNames . Alfajores , rpcUrl :
'https://alfajores-forno.celo-testnet.org' , graphql :
'https://alfajores-blockscout.celo-testnet.org/graphiql' , explorer :

```

'https://alfajores-blockscout.celo-testnet.org' , chainId :

```
44787 , } }
```

```
< App
```

```
/
```

```
< / CeloProvider
```

```
) ; }
```

```
function
```

```
App
```

```
( )
```

{ ... } Be sure to check the react-celo example application for a showcase of how network management works in more depth. Usually you'll want to show a dropdown to your users allowing them to select the network to connect to.

```
import
```

```
{ useCelo }
```

```
from
```

```
"@celo/react-celo" ;
```

```
function
```

```
App ( )
```

```
{ const
```

```
{ network , updateNetwork }
```

```
=
```

```
useCelo ( ) ;
```

```
return
```

```
< div
```

```
Currently connected to { network } < / div
```

```
; }
```

Extending Supported Networks

By default Use-Contractkit only supports Celo Blockchain Networks. You can however extend this to include other chains you choose such as Ethereum, Polygon, Avalanche etc by Passing your array ofNetwork s intoCeloProvider . Note this feature is considered experimental and works better with wallets like Metamask.

Manual Networking Mode

Opt out of the React-Celo automatically switching which network the dapp / wallet are on.walletChainId might be different to dapp'snetwork.chainId . You MUST useupdateNetwork to set the desired chain in this case.

Adjust FeeCurrency

react-celo provides afeeCurrency variable and anupdateFeeCurrency function you can use to display the currently selected feeCurrency (cUSD, CELO, cEUR). The feeCurrency can also be passed to the provider component. Valid values areCeloContract.GoldToken ,CeloContract.StableToken ,CeloContract.StableTokenEUR . CeloContract can be imported like so:

```
import { CeloTokenContract } from '@celo/contractkit'
```

Themes and dark-mode

Currently react-celo supports dark mode and light (aka default) mode via tailwind out of the box, to use the modal in dark

mode simply add the `class=dark` to the root tag of the web page.

If you default styles aren't to your taste, you can provide a theme object defined as such. You can do it during the setup of your dapp, at the `Provider` level. Or on the fly (let's say, if your users can change the theme of your dapp), via `updateTheme`.

```
interface
```

```
Theme
```

```
{ primary :
```

```
string ; secondary :
```

```
string ; text :
```

```
string ; textSecondary :
```

```
string ; textTertiary :
```

```
string ; muted :
```

```
string ; background :
```

```
string ; error :
```

```
string ; }
```

Logging and debugging

We log by default `debug` or above in development mode. It is determined your environment variables: by either setting `DEBUG` to `true` or setting `NODE_ENV` to something else than `production`). In production mode, we log only `error`.

But you are welcome to provide your own logger at the provider level. It should implement our `ILogger` interface which looks like that:

```
interface
```

```
ILogger
```

```
{ debug ( ... args :
```

```
unknown [ ] ) :
```

```
void ; log ( ... args :
```

```
unknown [ ] ) :
```

```
void ; warn ( ... args :
```

```
unknown [ ] ) :
```

```
void ; error ( ... args :
```

```
unknown [ ] ) :
```

```
void ; }
```

Development

To run all the packages locally at once, simply clone this repository and run:

```
yarn ; yarn build ;
```

only needs to be run the first time

`yarn dev` ; A hot reloading server should come up on `localhost:3000`, it's the exact same as what's at `react-celo-c-labs.vercel.app`.

Alternatively, you can individually run `react-celo` and the `example` app in parallel.

For that, you still need to have `run yarn` in the root.

Then, you can run react-celo in one tab:

cd packages/react-celo yarn dev and run the example app in another:

cd packages/example yarn dev

Support

Struggling with anything react-celo related? Jump into the [GitHub Discussions](#) or [celo-org discord channel](#) and ask for help any time.

Guides

More specialized use case info can be found in our [Guides](#) [Edit this page](#) [Previous Querying on-chain identifiers with ODIS](#)
[Next Web3Modal SDK](#)