usedapp

Ethworks team

Oct 10, 2021

CONTENTS:

1 Getting started **3**
- 1.1 Installation . . . 3
- 1.2 Example . . . 3
- 1.3 Setup . . . 4
- 1.4 Connecting to a network . . . 4
- 1.5 Ether balance . . . 5
- 1.6 Token balance . . . 5
- 1.7 Troubleshooting . . . 5

2 Guides **7**
- 2.1 Connecting to network . . . 7
- 2.2 Reading from blockchain . . . 9
- 2.3 Transactions . . . 12
- 2.4 Handling wallet activation errrors . . . 14

3 Core API **17**
- 3.1 Providers . . . 17
- 3.2 Hooks . . . 18
- 3.3 Models . . . 25
- 3.4 Constants . . . 28
- 3.5 Helpers . . . 29

4 Developer tools **33**
- 4.1 Installation . . . 33
- 4.2 List of events . . . 34
- 4.3 Adding custom ABIs . . . 35
- 4.4 Adding name tags . . . 36

5 CoinGecko API **37**
- 5.1 Hooks . . . 37

useDapp

Ethereum React

Framework for rapid Dapp development.
Simple. Robust. Extendable. Testable.

CHAPTER ONE

GETTING STARTED

1.1 Installation

To start working with useDapp you need to have working React environment.

To get started, add following npm package `@usedapp/core` to your project:

Yarn

NPM

```
yarn add @usedapp/core
```

```
npm install @usedapp/core
```

1.2 Example

Below is a simple example:

```
import { ChainId, DAppProvider, useEtherBalance, useEthers } from '@usedapp/core'
import { formatEther } from '@ethersproject/units'

const config: Config = {
  readOnlyChainId: ChainId.Mainnet,
  readOnlyUrls: {
    [ChainId.Mainnet]: 'https://mainnet.infura.io/v3/62687d1a985d4508b2b7a24827551934
↪',
  },
}

ReactDOM.render(
  <React.StrictMode>
    <DAppProvider config={config}>
      <App />
    </DAppProvider>
  </React.StrictMode>,
  document.getElementById('root')
)

export function App() {
  const { activateBrowserWallet, account } = useEthers()
  const etherBalance = useEtherBalance(account)
```

(continues on next page)

(continued from previous page)

```
  return (
    <div>
      <div>
        <button onClick={() => activateBrowserWallet()}>Connect</button>
      </div>
      {account && <p>Account: {account}</p>}
      {etherBalance && <p>Balance: {formatEther(etherBalance)}</p>}
    </div>
  )
}
```

Example is available here and full example code is available here.

Let's go over it step by step.

1.3 Setup

The first thing you need to do is set up **DAppProvider** with optional config and wrap your whole App in it. You can read about config *here*.

```
<DAppProvider>
  <App /> {/* Wrap your app with the Provider */}
</DAppProvider>
```

1.4 Connecting to a network

Then you need to activate the provider using **activateBrowserWallet**. It's best to do when the user clicks "Connect" button.

```
export function App() {
  const { activateBrowserWallet, account } = useEthers()
  return (
    <div>
      <div>
        <button onClick={() => activateBrowserWallet()}>Connect</button>
      </div>
      {account && <p>Account: {account}</p>}
    </div>
  )
}
```

After the activation (i.e. user connects to a wallet like MetaMask) the component will show the user's address.

If you need to use another connector than a browser wallet, use the *activate* method from *useEthers*. See the *web3-react <https://github.com/NoahZinsmeister/web3-react/tree/v6/docs#overview>* doc for that one.

1.5 Ether balance

useEtherBalance(address: string)

Provides a way to fetch the account balance. Takes the account address as an argument and returns `BigNumber` or `undefined` when data is not available (i.e. not connected). To obtain currently connected `account` employ `useEthers()`.

```
import { formatEther } from '@ethersproject/units'

export function EtherBalance() {
  const { account } = useEthers()
  const etherBalance = useEtherBalance(account)

  return (
    <div>
      {etherBalance && <p>Balance: {formatEther(etherBalance)}</p>}
    </div>
  )
}
```

1.6 Token balance

useTokenBalance(address: string, tokenAddress: string)

Provides a way to fetch balance of ERC20 token specified by `tokenAddress` for provided `address`. Returns `BigNumber` or `undefined` when data is not available.

```
import { formatUnits } from '@ethersproject/units'

const DAI = '0x6b175474e89094c44da98b954eedeac495271d0f'

export function TokenBalance() {
  const { account } = useEthers()
  const tokenBalance = useTokenBalance(DAI, account)

  return (
    <div>
      {tokenBalance && <p>Balance: {formatUnits(tokenBalance, 18)}</p>}
    </div>
  )
}
```

1.7 Troubleshooting

1.7.1 Type mismatch when building

If when building an app you see errors about type mismatch in `@ethersproject`.

For example:

```
$ yarn build
yarn run v1.22.10
$ tsc --noEmit && rimraf build && webpack --mode production --progress
src/components/Transactions/Forms.tsx:12:52 - error TS2345: Argument of type
↪'Interface' is not assignable to parameter of type 'ContractInterface'.
  Property 'getError' is missing in type 'import("github.com/ethworks/usedapp/
↪packages/example/node_modules/@ethersproject/abi/lib/interface").Interface' but␣
↪required in type 'import("github.com/ethworks/usedapp/packages/example/node_modules/
↪@ethersproject/contracts/node_modules/@ethersproject/abi/lib/interface").Interface'.

12 const contract = new Contract(wethContractAddress, wethInterface)
                                                      ~~~~~~~~~~~~~

  node_modules/@ethersproject/contracts/node_modules/@ethersproject/abi/lib/interface.
↪d.ts:53:5
    53     getError(nameOrSignatureOrSighash: string): ErrorFragment;
          ~~~~~~~~
    'getError' is declared here.


Found 1 error.

error Command failed with exit code 2.
info Visit https://yarnpkg.com/en/docs/cli/run for documentation about this command.
```

It may be an error of yarn getting internal versions of `@ethersproject` that are higher then specified in useDApp. To fix this you need to add resolutions to your `package.json` with etherspoject packages that cause an error, with correct version. Resolutions force yarn to install specified versions of packages.

For example:

```
"resolutions": {
  "@ethersproject/abi": "5.2.0",
  "@ethersproject/contracts": "5.2.0"
}
```

CHAPTER

TWO

GUIDES

2.1 Connecting to network

2.1.1 Read-only

To connect to the network in read-only mode, provide `readOnlyChainId` and `readOnlyUrls` fields in application configuration.

See example configuration below:

```
const config = {
  readOnlyChainId: ChainId.Mainnet,
  readOnlyUrls: {
    [ChainId.Mainnet]: 'https://mainnet.infura.io/v3/62687d1a985d4508b2b7a24827551934
↪',
  },
}
```

2.1.2 Browser wallet

To connect to a wallet in a web3-enabled browser, use `activateBrowserWallet` returned by `useEthers()`. Once connected `account` variable will be available.

See example below:

```
export function App() {
  const { activateBrowserWallet, account } = useEthers()
  return (
    <div>
      {!account && <button onClick={activateBrowserWallet}> Connect </button>}
      {account && <p>Account: {account}</p>}
    </div>
  )
}
```

2.1.3 useEthers

`useEthers` hook returns a number of useful functions and variables, see below:

- `account` - current user account (or `null` if not connected or connected in read-only mode)
- `chainId` - current chainId (or `undefined` if not connected)
- `library` - an instance of ethers Web3Provider (or `undefined` if not connected). Read more about ethers providers here.
- `active` - boolean that indicates if provider is connected (read or write mode)
- `activate` - function that allows to connect to a wallet. This is a web3-react function that can take various connectors.
- `deactivate` - function that disconnects wallet
- `error` - an error that occurred during connecting (e.g. connection is broken, unsupported network)

2.1.4 Example

Example below demonstrates how to manage and use connection.

Application allow to see the balance of Ethereum 2.0 staking contracts in read-only mode. When wallet is connected additionally it shows user's account along with it's balance.

Example is available here.

```
const config = {
  readOnlyChainId: ChainId.Mainnet,
  readOnlyUrls: {
    [ChainId.Mainnet]: 'https://mainnet.infura.io/v3/62687d1a985d4508b2b7a24827551934
↪',
  },
}

ReactDOM.render(
  <DAppProvider config={config}>
    <App />
  </DAppProvider>
  document.getElementById('root')
)

const STAKING_CONTRACT = '0x00000000219ab540356cBB839Cbe05303d7705Fa'

export function App() {
  const { activateBrowserWallet, deactivate, account } = useEthers()
  const userBalance = useEtherBalance(account)
  const stakingBalance = useEtherBalance(STAKING_CONTRACT)

  return (
    <div>
      {!account && <button onClick={activateBrowserWallet}> Connect </button>}
      {account && <button onClick={deactivate}> Disconnect </button>}

      {stakingBalance && <p>ETH2 staking balance: {formatEther(stakingBalance)} ETH </
↪p>}
      {account && <p>Account: {account}</p>}
```

(continues on next page)

(continued from previous page)

```
      {userBalance && <p>Ether balance: {formatEther(userBalance)} ETH </p>}
    </div>
  )
}
```

2.2 Reading from blockchain

There is a number of useful hooks that you can use to read blockchain state:

- `useBlockMeta()` - return meta information (`timestamp` and `difficulty`) about most recent block mined
- `useEtherBalance(address)` - returns ether balance as BigNumber for given address (or `undefined`)
- `useTokenBalance(tokenAddress, address)` - returns balance of a given token as BigNumber for given address (or undefined)
- `useTokenAllowance(tokenAddress, ownerAddress, spenderAddress)` - returns allowance of a given token as BigNumber for given owner and spender address pair (or undefined)

Sooner or later you will want to make a custom call to a smart contract. Use `useContractCall` and `useContractCalls` for that purpose. See section below on creating custom hooks.

2.2.1 Custom hooks

Creating a custom hook with the use of our core hooks is straightforward, for example let's examine the *useTokenBalance* hook.

The hook will retrieve a balance of an ERC20 token of the provided address.

```
function useTokenBalance(tokenAddress: string | Falsy, address: string | Falsy) {
  const [tokenBalance] = useContractCall(
    ERC20Interface, // ABI interface of the called contract
    tokenAddress, // On-chain address of the deployed contract
    'balanceOf', // Method to be called
    address && [address] // Method arguments - address to be checked for balance
  ) ?? []
  return tokenBalance
}
```

Another example is useTokenAllowance hook. Instead of balanceOf, we use allowance on ERC20 interface.

```
function useTokenAllowance(
  tokenAddress: string | Falsy,
  ownerAddress: string | Falsy,
  spenderAddress: string | Falsy
) {
  const [allowance] =
    useContractCall(
      ownerAddress &&
        spenderAddress &&
        tokenAddress && {
          abi: ERC20Interface,
          address: tokenAddress,
```

(continues on next page)

(continued from previous page)

```
          method: 'allowance',
          args: [ownerAddress, spenderAddress],
        }
    ) ?? []
  return allowance
}
```

The *useContractCall* hook will take care of updating the balance of new blocks. The results are deferred so that the hook does not update too frequently.

In our custom hooks we can use any standard react hooks, custom react hooks and useDapp hooks. Rules of hooks apply.

Documentation for hooks is available *here*.

2.2.2 Using hooks considerations

There are some important considerations when using hooks based on *useChainCall*, *useChainCalls* and *useContractCalls*.

Avoid using the result of one hook in another. This will break single multicall into multiple multicalls. It will reduce performance, generate delays, and flickering for the user. Instead, try to retrieve needed information in a single call or multiple parallel calls. That might require modification of smart contracts. If that is too complex consider using a custom backend or The Graph.

2.2.3 Testing hooks

Let's take `useTokenAllowance` as an example.

To write a test, start with a setup code that will create a mock provider and test wallets.

```
const mockProvider = new MockProvider()
const [deployer, spender] = mockProvider.getWallets()
```

Before each test, deploy an ERC20 contract. It's important as otherwise the result of one test could break the other one.

```
let token: Contract

beforeEach(async () => {
  const args = ['MOCKToken', 'MOCK', deployer.address, utils.parseEther("10")]
  token = await deployContract(deployer, ERC20Mock, args)
})
```

After setup, we have to test the hook.

```
await token.approve(spender.address, utils.parseEther('1'))

const { result, waitForCurrent } = await renderWeb3Hook(
  () => useTokenAllowance(token.address, deployer.address, spender.address),
  {
    mockProvider,
  }
)
await waitForCurrent((val) => val !== undefined)
```

(continues on next page)

(continued from previous page)

```
expect(result.error).to.be.undefined
expect(result.current).to.eq(utils.parseEther('1'))
```

To check if the hook reads data correctly, we need to prepare it first. We approve the spender so that we can check if our hook returns the correct value.

To test the hook we need to render it using `renderWeb3Hook`. It works like `renderHook` from the react-testing-library, but it wraps the hook into additional providers.

React components update asynchronically. Reading data from the blockchain is also an async operation. To get the return value from the hook, wait for the result to be set. You can do it with `waitForCurrent`.

Then we can check if our result is correct. `result.current` is a value returned from our hook. It should be equal to 1 Ether.

Full example

```
import { MockProvider } from '@ethereum-waffle/provider'
import { Contract } from '@ethersproject/contracts'
import { useTokenAllowance, ERC20Mock } from '@usedapp/core'
import { renderWeb3Hook } from '@usedapp/testing'
import chai, { expect } from 'chai'
import { solidity, deployContract } from 'ethereum-waffle'
import { utils } from 'ethers'

chai.use(solidity)

describe('useTokenAllowance', () => {
  const mockProvider = new MockProvider()
  const [deployer, spender] = mockProvider.getWallets()
  let token: Contract

  beforeEach(async () => {
    const args = ['MOCKToken', 'MOCK', deployer.address, utils.parseEther("10")]
    token = await deployContract(deployer, ERC20Mock, args)
  })

  it('returns current allowance', async () => {
    await token.approve(spender.address, utils.parseEther('1'))

    const { result, waitForCurrent } = await renderWeb3Hook(
      () => useTokenAllowance(token.address, deployer.address, spender.address),
      {
        mockProvider,
      }
    )
    await waitForCurrent((val) => val !== undefined)

    expect(result.error).to.be.undefined
    expect(result.current).to.eq(utils.parseEther('1'))
  })
})
```

2.3 Transactions

2.3.1 Sending transaction

Example is available here.

Sending transactions is really simple with useDApp. All we need to send a simple transaction, is to use *useSendTransaction* hook, which returns a `sendTransaction` function and `state` object.

Example

Simply call a hook in a component.

```
const { sendTransaction, state } = useSendTransaction()
```

Then when you want to send a transaction, call `sendTransaction` for example in a button callback. Function accepts a Transaction Request object as a parameter. In example below `setDisabled(true)` sets input components to disabled while transaction is being processed (It is a good practice to disable component when transaction is mining).

```
const handleClick = () => {
  setDisabled(true)
  sendTransaction({ to: address, value: utils.parseEther(amount) })
}
```

After that you can use state to check the state of your transtaction. State is of type *TransactionStatus*. Example below clears inputs and enables all disabled components back:

```
useEffect(() => {
  if (state.status != 'Mining') {
    setDisabled(false)
    setAmount('0')
    setAddress('')
  }
}, [state])
```

2.3.2 Executing contract function

To send a transaction that executes a function of a contract on a blockchain, you can use a *useContractFunction* hook, it works similarly to *useSendTransaction*. It returns a `send` function that we can use to call a contract function and `state` object.

To use `useContractFunction` we need to supply it with a Contract of type Contract. And a string `functionName`.

`send` function maps arguments 1 to 1 with functions of a contract and also accepts one additional argument of type TransactionOverrides

Example

Start by declaring a contract variable with address of contract you want to call and ABI interface of a contract.

```
import { utils } from 'ethers'
import { Contract } from '@ethersproject/contracts'

...
```

(continues on next page)

(continued from previous page)

```
const wethInterface = new utils.Interface(WethAbi)
const wethContractAddress = '0xA243FEB70BaCF6cD77431269e68135cf470051b4'
const contract = new Contract(wethContractAddress, wethInterface)
```

After that you can use the hook to create `send` function and `state` object.

```
const { state, send } = useContractFunction(contract, 'deposit', { transactionName:
↪'Wrap' })

const depositEther = (etherAmount: string) => {
  send({ value: utils.parseEther(etherAmount) })
}
```

```
const { state, send } = useContractFunction(contract, 'withdraw', { transactionName:
↪'Unwrap' })

const withdrawEther = (wethAmount: string) => {
  send(utils.parseEther(wethAmount))
}
```

The code snippets above will wrap and unwrap Ether into WETH using Wrapped Ether contract respectively. Deposit function of a contract has no input arguments and instead wraps amount of ether sent to it. To send given amount of ether simply use a `TransactionOverrides` object. Withdraw function needs amount of ether to withdraw as a input argument.

2.3.3 History

See *useTransactions*

To access history of transactions, use `useTransactions` hook.

```
const { transactions } = useTransactions()
```

`transactions` is an array so you can use `transactions.map(...)` to display all of transactions.

For example:

```
{transactions.map((transaction) => (
        <ListElement
          transaction={transaction.transaction}
          title={transaction.transactionName}
          icon={TransactionIcon(transaction)}
          key={transaction.transaction.hash}
          date={transaction.submittedAt}
        />
      ))}
```

`ListElement` is a react function that displays information about single transaction.

2.3.4 Notifications

See *useNotifications*.

To use notifications in your app simply call:

```
const { notifications } = useNotifications()
```

After that you can use `notifications` as an array. Notifications are automatically removed from array after time declared in config.notifications.expirationPeriod.

In react you can simply use `notifications.map(...)` to display them.

For example :

```
{notifications.map((notification) => {
  if ('transaction' in notification)
    return (
      <NotificationElement
        key={notification.id}
        icon={notificationContent[notification.type].icon}
        title={notificationContent[notification.type].title}
        transaction={notification.transaction}
        date={Date.now()}
      />
    )
  else
    return (
      <NotificationElement
        key={notification.id}
        icon={notificationContent[notification.type].icon}
        title={notificationContent[notification.type].title}
        date={Date.now()}
      />
    )
})}
```

`NotificationElement` is a react function that renders a single notification. `notificationContent` is an object that holds information about what title and icon to show. You have to remember that object in `notifications` array may not contain transaction field

(that's why there is if statement).

2.4 Handling wallet activation errrors

Because `activateBrowserWallet()` from *useEthers* is using activate from web3-react. It is made so that it can handle errors the same way that `activate()` handles them, for more info see here.

As such the error can be handled in 3 ways:

- By passing a callback as first parameter of :

```
const onError = (error: Error) => {
  console.log(error.message)
}
activateBrowserWallet(onError)
```

- By passing a true as second argument will make activateBrowserWallet throw on errors :

```
try{
  await activateBrowserWallet(undefined,true)
} catch(error) {
  console.log(error)
}
```

- By checking if *const {error} = useEthers()* changes :

```
const [activateError, setActivateError] = useState('')
const { error } = useEthers()
useEffect(() => {
  if (error) {
    setActivateError(error.message)
  }
}, [error])

const activate = async () => {
  setActivateError('')
  activateBrowserWallet()
}
```

Becouse useDApp defaults to read only connector `error` from `useEthers()` is only shown for few frames as such if you want to handle it you need to store error in a state

CHAPTER THREE

CORE API

3.1 Providers

3.1.1 <DAppProvider>

Provides basic services for a DApp. It combines the following components: `<ConfigProvider>`, `<EthersProvider>`, `<BlockNumberProvider>`, `<ChainStateProvider>` and `<ReadOnlyProviderActivator>`

Properties:

- `config: Partial<Config>`: configuration of the DApp, see *Config*

Example:

```
const config = {
  readOnlyChainId: ChainId.Mainnet,
  readOnlyUrls: {
    [ChainId.Mainnet]: `https://mainnet.infura.io/v3/${INFURA_ID}`,
  },
}

return (
  <DAppProvider config={config}>
    <App />
  </DAppProvider>
)
```

3.1.2 <ConfigProvider>

Stores configurations and makes them available via *useConfig* hook.

3.1.3 <EthersProvider>

Requires: `ConfigProvider`

3.1.4 <BlockNumberProvider>

3.1.5 <LocalMulticallProvider>

Ensures that a multicall contract address is available when developing on a local chain. A multicall contract will be deployed when a multicall address on a local chainID is not defined in the *Config*.

While the contract is being deployed, a temporary "Deploying multicall…" message will be rendered instead of the user's child components.

3.1.6 <ChainStateProvider>

3.1.7 <ReadOnlyProviderActivator>

3.2 Hooks

3.2.1 useBlock

3.2.2 useBlockMeta

3.2.3 useBlockNumber

Get the current block number. Will update automatically when the new block is mined.

3.2.4 useChainCall

Makes a call to a specific contract and returns the value. The hook will cause the component to refresh whenever a new block is mined and the value is changed.

Calls will be combined into a single multicall across all uses of *useChainCall* and *useChainCalls*.

It is recommended to use *useContractCall* where applicable instead of this method.

Parameters

- `call: ChainCall | Falsy` - a single call, also see *ChainCall*. A call can be *Falsy*, as it is important to keep the same ordering of hooks even if in a given render cycle there might be not enough information to perform a call.

3.2.5 useChainCalls

Makes multiple calls to specific contracts and returns values. The hook will cause the component to refresh when values change.

Calls will be combined into a single multicall across all uses of *useChainCall* and *useChainCalls*. It is recommended to use *useContractCall* where applicable instead of this method.

Parameters

- `calls: ChainCall[]` - list of calls, also see *ChainCall*. Calls need to be in the same order across component renders.

3.2.6 useContractCall

Makes a call to a specific contract and returns the value. The hook will cause the component to refresh when a new block is mined and the return value changes. A syntax sugar for *useChainCall* that uses ABI, function name, and arguments instead of raw data.

Parameters

- `calls: ContractCall | Falsy` - a single call to a contract , also see *ContractCall*

Returns

- `any[] | undefined` - the result of a call or undefined if call didn't return yet

3.2.7 useContractCalls

Makes calls to specific contracts and returns values. The hook will cause the component to refresh when a new block is mined and the return values change. A syntax sugar for *useChainCalls* that uses ABI, function name, and arguments instead of raw data.

Parameters

- `calls: ContractCall[]` - a list of contract calls , also see *ContractCall*

Returns

- `any[] | undefined` - array of results. Undefined if call didn't return yet

3.2.8 useContractFunction

Hook returns an object with three variables: `state` , `send` and `events`.

The `state` represents the status of transaction. See *TransactionStatus*.

The `events` is a array of parsed transaction events of type LogDescription.

To send a transaction use `send` function returned by `useContractFunction`. The function forwards arguments to ethers.js contract object, so that arguments map 1 to 1 with Solidity function arguments. Additionally, there can be one extra argument - TransactionOverrides, which can be used to manipulate transaction parameters like gasPrice, nonce, etc

Parameters

- `contract: Contract` - contract which function is to be called , also see Contract
- `functionName: string` - name of function to call

- `options?: Options` - additional options of type *TransactionOptions*.

Returns

- `{ send: (...args: any[]) => void, state: TransactionStatus, events: LogDescription[] }` - object with variables: `send` , `state` , `events`

Example

```
const { state, send } = useContractFunction(contract, 'deposit', { transactionName:
↪'Wrap' })

const depositEther = (etherAmount: string) => {
  send({ value: utils.parseEther(etherAmount) })
}
```

```
const { state, send } = useContractFunction(contract, 'withdraw', { transactionName:
↪'Unwrap' })

const withdrawEther = (wethAmount: string) => {
  send(utils.parseEther(wethAmount))
}
```

3.2.9 useSendTransaction

Hook returns an object with two variables: `state` and `sendTransaction`.

The former represents the status of transaction. See *TransactionStatus*.

To send a transaction use `sendTransaction` function returned by `useSendTransaction`.

Function accepts a Transaction Request object as a parameter.

Parameters

- `options?: Options` - additional options of type *TransactionOptions*.

Returns

- `{ sendTransaction: (...args: any[]) => void, state: TransactionStatus }` - object with two variables: `sendTransaction` and `state`

Example

```
const { sendTransaction, state } = useSendTransaction({ transactionName: 'Send␣
↪Ethereum' })

const handleClick = () => {

  ...

  sendTransaction({ to: address, value: utils.parseEther(amount) })
}
```

3.2.10 useConfig

Returns singleton instance of *Config*.

Function takes no parameters.

3.2.11 useDebounce

Debounce a value of type T. It stores a single value but returns after debounced time unless a new value is assigned before the debounce time elapses, in which case the process restarts.

Generic parameters

- `T` - type of stored value

Parameters

- `value: T` - variable to be debounced
- `delay: number` - debounce time - amount of time in ms

Returns

- `T` - debounced value

Example

```
const [someValue, setValue] = useState(...)
const debouncedValue = useDebounce(value, 1000)
```

3.2.12 useDebouncePair

Debounce a pair of values of types T and U. It stores a single value but returns after debounced time unless a new value is assigned before the debounce time elapses, in which case the process restarts.

This function is used for debouncing multicall until enough calls are aggregated.

Generic parameters

- `T` - type of first stored value
- `U` - type of second stored value

Parameters

- `first: T` - first variable to be debounced
- `second: U` - second variable to be debounced
- `delay: number` - debounce time - amount of time in ms

Returns

- `[T, U]` - debounced values

3.2.13 useEtherBalance

Returns ether balance of a given account.

Parameters

- `address: string | Falsy` - address of an account

Returns

- `balance: BigNumber | undefined` - a balance of the account which is BigNumber or *undefined* if not connected to network or address is a falsy value

Example

```
const { account } = useEthers()
const etherBalance = useEtherBalance(account)

return (
  {etherBalance && <p>Ether balance: {formatEther(etherBalance)} ETH </p>}
)
```

3.2.14 useEthers

Returns connection state and functions that allow to manipulate the state.

Returns:

- `account: null | string` - current user account (or *null* if not connected or connected in read-only mode)
- `chainId: ChainId` - current chainId (or *undefined* if not connected)
- `library: Web3Provider` - an instance of ethers Web3Provider (or *undefined* if not connected)
- `active: boolean` - returns if provider is connected (read or write mode)
- `activateBrowserWallet(onError?: (error: Error) => void, throwErrors?: boolean)` - function that will initiate connection to browser web3 extension (e.g. Metamask)
- `async activate(connector: AbstractConnector, onError?: (error: Error) => void, throwErrors?: boolean)` - function that allows to connect to a wallet
- `async deactivate()` - function that disconnects wallet
- `error?: Error` - an error that occurred during connecting (e.g. connection is broken, unsupported network)

Requires: `<ConfigProvider>`

3.2.15 useGasPrice

Returns gas price of current network.

Returns

- `gasPrice: BigNumber | undefined` - gas price of current network. Undefined if not initialised

3.2.16 useMulticallAddress

3.2.17 useNotifications

`useNotifications` is a hook that is used to access notifications. Notifications include information about: new transactions, transaction success or failure, as well as connection to a new wallet.

To use this hook call:

```
const { notifications } = useNotifications()
```

`notifications` is an array of `NotificationPayload`.

Each notification is removed from `notifications` after time declared in config.notifications.expirationPeriod

Each can be one of the following:

```
{
  type: 'walletConnected';
  address: string
}
```

```
{
  type: 'transactionStarted';
  submittedAt: number
  transaction: TransactionResponse;
  transactionName?: string
}
```

```
{
  type: 'transactionSucceed'
  transaction: TransactionResponse
  receipt: TransactionReceipt
  transactionName?: string
}
```

```
{
  type: 'transactionFailed'
  transaction: TransactionResponse
  receipt: TransactionReceipt
  transactionName?: string
}
```

Link to: Transaction Response.

Link to: Transaction Receipt.

3.2.18 useTokenBalance

Returns a balance of a given token for a given address.

Parameters

- `tokenAddress: string | Falsy` - address of a token contract
- `address: string | Falsy` - address of an account

Returns

- `balance: BigNumber | undefined` - a balance which is BigNumber or undefined if address or token is *Falsy* or not connected

Example

```
const DAI_ADDRESS = '0x6b175474e89094c44da98b954eedeac495271d0f'
const { account } = useEthers()
const daiBalance = useTokenBalance(DAI_ADDRESS, account)

return (
  {daiBalance && <p>Dai balance: {formatUnits(daiBalance, 18)} DAI</p>}
)
```

3.2.19 useTokenAllowance

Returns allowance (tokens left to use by spender) for given tokenOwner - spender relationship.

Parameters

- `tokenAddress: string | Falsy` - address of a token contract
- `ownerAddress: string | Falsy` - address of an account to which tokens are linked
- `spenderAddress: string | Falsy` - address of an account allowed to spend tokens

Returns

- `remainingAllowance: BigNumber | undefined` - an allowance which is BigNumber or undefined if any address or token is *Falsy* or not connected

Example

```
const TOKEN_ADDRESS = '0x6b175474e89094c44da98b954eedeac495271d0f'
const SPENDER_ADDRESS = '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA'
const { account, chainId } = useEthers()
const allowance = useTokenAllowance(TOKEN_ADDRESS, account, SPENDER_ADDRESS)

return (
  {allowance && <p>Remaining allowance: {formatUnits(allowance, 18)} tokens</p>}
)
```

3.2.20 useTransactions

`useTransactions` hook returns a list `transactions`. This list contains all transactions that were sent using `useContractFunction` and `useSendTransaction`. Transactions are stored in local storage and the status is rechecked on every new block.

Each transaction has following type:

```
export interface StoredTransaction {
  transaction: TransactionResponse
  submittedAt: number
  receipt?: TransactionReceipt
  lastCheckedBlockNumber?: number
  transactionName?: string
}
```

Link to: Transaction Response.

Link to: Transaction Receipt.

3.2.21 useLookupAddress

`useLookupAddress` is a hook that is used to retrieve the ENS (e.g. *name.eth*) for the connected wallet.

Returns

- `address: String | undefined` - a string if the connected account has an ENS attached.

Example

```
const { account } = useEthers()
const ens = useDisplayName()

return (
  <p>Account: {ens ?? account}</p>
)
```

3.3 Models

3.3.1 Config

readOnlyChainId

`ChainId` of a chain you want to connect to by default in a read-only mode

readOnlyUrls

Mapping of `ChainId`'s to node URLs to use in read-only mode.

Example

```
{
  ...
  readOnlyUrls: {
    [ChainId.Mainnet]: 'https://mainnet.infura.io/v3/62687d1a985d4508b2b7a24827551934'
  }
}
```

multicallAddresses

supportedChains List of intended supported chains. If a user tries to connect to an unsupported chain an error value will be returned by *useEthers*.

Default value: `[ChainId.Mainnet, ChainId.Goerli, ChainId.Kovan, ChainId.Rinkeby, ChainId.Ropsten, ChainId.xDai]`

pollingInterval Polling interval for a new block.

localStorage Paths to locations in local storage

Default value:

```
{
  transactionPath: 'transactions'
}
```

3.3.2 ChainCall

Represents a single call on the blockchain that can be included in multicall.

Fields:

- `address: string` - address of a contract to call
- `data: string` - calldata of the call that encodes function call

3.3.3 ContractCall

Represents a single call to a contract that can be included in multicall.

Fields:

- `abi: Interface` - ABI of a contract, see Interface
- `address: string` - address of a contract to call
- `method: string` - function name
- `args: any[]` - arguments for the function

3.3.4 Currency

The `Currency` class is tasked with representing the individual currencies as well as handling formatting.

The base `Currency` class is constructed with the following parameters: - `name` - name of the currency - `ticker` - e.g. USD, EUR, BTC - `decimals` - number of decimal places (e.g. 2 for USD, 18 for ETH) - `formattingOptions` - define how the currency values are formatted

The following formatting options are supported:

- `decimals` - Defaults to the decimals of the currency.
- `thousandSeparator` - Defaults to `','`. Used for separating thousands.
- `decimalSeparator` - Defaults to `'.'`. Used for separating the integer part from the decimal part.
- `significantDigits` - Defaults to Infinity. Can limit the number of digits on the decimal part, such that either the total number of displayed digits is equal to this parameter or more digits are displayed, but the decimal part is missing.

- `useFixedPrecision` - Defaults to false. Switches from using significant digits to fixed precision digits.
- `fixedPrecisionDigits` - Defaults to 0. Can specify the number of digits on the decimal part.
- `prefix` - Defaults to `''`. Prepended to the result.
- `suffix` - Defaults to `''`. Appended to the result.

Other variants of `Currency` include `FiatCurrency`, `NativeCurrency` and `Token`.

`FiatCurrency` takes the same parameters as `Currency` but uses fixed precision digits by default.

`NativeCurrency` additionally takes a `chainId` parameter. The format function is configured with the ticker prefix and 6 significant digits by default.

`Token` additionally takes a `chainId` parameter as well as an `address` parameter. The format function is configured with the ticker prefix and 6 significant digits by default.

3.3.5 CurrencyValue

The `CurrencyValue` class represents a value tied to a currency. The methods include:

- `static fromString(currency, value)` - creates a new CurrencyValue from string.
- `static zero(currency)` - creates a new CurrencyValue equal to 0.
- `toString()` - returns the value of the CurrencyValue as a decimal string with no formatting.
- `format(overrideOptions?)` - formats the value according to the currency. The caller can override the formatting options.
- `map(fn)` - returns a new CurrencyValue with value transformed by the callback.
- `add(other)` - returns a new CurrencyValue with value being the sum of this value and other value. The argument must be a CurrencyValue with the same Currency.
- `sub(other)` - returns a new CurrencyValue with value being the difference of this value and other value. The argument must be a CurrencyValue with the same Currency.
- `mul(value)` - returns a new CurrencyValue with value multiplied by the argument.
- `div(value)` - returns a new CurrencyValue with value divided by the argument.
- `mod(value)` - returns a new CurrencyValue with value modulo the argument.
- `equals(other)` - performs an equality check on the currencies and the values of both objects.
- `lt(other)` - checks if this value is less than the other value. The argument must be a CurrencyValue with the same Currency.
- `lte(other)` - checks if this value is less than or equal to the other value. The argument must be a CurrencyValue with the same Currency.
- `gt(other)` - checks if this value is greater than the other value. The argument must be a CurrencyValue with the same Currency.
- `gte(other)` - checks if this value is greater than or equal to the other value. The argument must be a CurrencyValue with the same Currency.
- `isZero()` - returns true if the value is zero.

3.3.6 TransactionOptions

Represents a options for sending transactions. All fields are optional.

Fields:

- `signer?: Signer` - specifies signer for a transaction.
- `transactionName?: string` - specifies a transaction name. Used by notifications and history hooks.

3.3.7 TransactionStatus

Represents a state of a single transaction.

Fields:

- `status: TransactionState` - string that can contain one of `None Mining Success Fail Exception`
- `transaction?: TransactionResponse` - optional field. See Transaction Response.
- `receipt?: TransactionReceipt` - optional field. See Transaction Receipt.
- `chainId?: ChainId` - optional field. See *chainId*.
- `errorMessage?: string` - optional field that contains error message when transaction fails or throws.

`status` can be one of the following:

- **None** - before a transaction is created.
- **Mining** - when a transaction is sent to the network, but not yet mined. In this state `transaction: TransactionResponse` is available.
- **Success** - when a transaction has been mined successfully. In this state `transaction: TransactionResponse` and `receipt: TransactionReceipt` are available.
- **Failed** - when a transaction has been mined, but ended up reverted. Again `transaction: TransactionResponse` and `receipt: TransactionReceipt` are available.
- **Exception** - when a transaction hasn't started, due to the exception that was thrown before the transaction was propagated to the network. The exception can come from application/library code (e.g. unexpected exception like malformed arguments) or externally (e.g user discarded transaction in Metamask). In this state the `errorMessage: string` is available (as well as exception object).

Additionally all states except `None`, contain `chainId: ChainId`.

Change in `state` will update the component so you can use it in useEffect.

3.4 Constants

3.4.1 ChainId

Enum that represents chain ids.

Values:

`Mainnet, Goerli, Kovan, Rinkeby, Ropsten, BSC, xDai, Polygon, Moonriver, Mumbai, Harmony, Theta, Palm, Fantom`

3.5 Helpers

3.5.1 getExplorerAddressLink

Returns URL to blockchain explorer for an address on a given chain.

Parameters

- `address: string` - account address
- `chainId: ChainId` - id of a chain

Example

```
getExplorerAddressLink('0xC7095A52C403ee3625Ce8B9ae8e2e46083b81987', ChainId.Mainnet)
// https://etherscan.io/address/0xC7095A52C403ee3625Ce8B9ae8e2e46083b81987

getExplorerAddressLink('0xC7095A52C403ee3625Ce8B9ae8e2e46083b81987', ChainId.Ropsten)
// https://ropsten.etherscan.io/address/0xC7095A52C403ee3625Ce8B9ae8e2e46083b81987

getExplorerAddressLink('0xC7095A52C403ee3625Ce8B9ae8e2e46083b81987', ChainId.xDai)
// https://blockscout.com/poa/xdai/address/0xC7095A52C403ee3625Ce8B9ae8e2e46083b81987/
↪transactions

  getExplorerAddressLink('0xC7095A52C403ee3625Ce8B9ae8e2e46083b81987', ChainId.
↪Harmony)
// https://explorer.harmony.one/address/0xc7095a52c403ee3625ce8b9ae8e2e46083b81987
```

3.5.2 getExplorerTransactionLink

Returns URL to blockchain explorer for a transaction hash on a given chain.

Parameters

- `transactionHash: string` - hash of a transaction
- `chainId: ChainId` - id of a chain

Example

```
getExplorerTransactionLink('0xC7095A52C403ee3625Ce8B9ae8e2e46083b81987', ChainId.
↪Mainnet)
// https://etherscan.io/tx/
↪0x5d53558791c9346d644d077354420f9a93600acf54eb6a279f12b43025392c3a

getExplorerTransactionLink('0xC7095A52C403ee3625Ce8B9ae8e2e46083b81987', ChainId.
↪Ropsten)
// https://ropsten.etherscan.io/tx/
↪0x5d53558791c9346d644d077354420f9a93600acf54eb6a279f12b43025392c3a

getExplorerTransactionLink('0xC7095A52C403ee3625Ce8B9ae8e2e46083b81987', ChainId.xDai)
// https://blockscout.com/poa/xdai/tx/
↪0x5d53558791c9346d644d077354420f9a93600acf54eb6a279f12b43025392c3a/internal-
↪transactions

 getExplorerTransactionLink('0xC7095A52C403ee3625Ce8B9ae8e2e46083b81987', ChainId.
↪Harmony)
// https://explorer.harmony.one/tx/
↪0x5d53558791c9346d644d077354420f9a93600acf54eb6a279f12b43025392c3a
```

(continues on next page)

(continued from previous page)

3.5.3 getChainName

Returns name of a chain for a given *chainId*.

Parameters

- `chainId: ChainId` - id of a chain

Example

```
getChainName(ChainId.Mainnet) // Mainnet
getChainName(ChainId.Ropsten) // Ropsten
getChainName(ChainId.xDai)    // xDai
getChainName(ChainId.Theta) // Theta
getChainName(ChainId.Harmony) // Harmony
getChainName(ChainId.Moonriver) // Moonriver
getChainName(ChainId.Fantom) // Fantom
```

3.5.4 isTestChain

Returns if a given chain is a testnet.

Parameters

- `chainId: ChainId` - id of a chain

Example

```
isTestChain(ChainId.Mainnet) // false
isTestChain(ChainId.Ropsten) // true
isTestChain(ChainId.xDai)    // false
```

3.5.5 shortenAddress

Returns short representation of address or throws an error if address is incorrect.

Parameters

- `address: string` - address to shorten

Example

```
shortenAddress('0x6E9e7A8Fb61b0e1Bc3cB30e6c8E335046267D3A0')
// 0x6E9e...D3A0

shortenAddress('6E9e7A8Fb61b0e1Bc3cB30e6c8E335046267D3A0')
// 0x6E9e...D3A0

shortenAddress("i'm not an address")
// TypeError("Invalid input, address can't be parsed")
```

3.5.6 shortenIfAddress

Returns short representation of address or throws an error if address is incorrect. Returns empty string if no address is provided.

Parameters

- `address: string | 0 | null | undefined | false` - address to shorten

Example

```
shortenIfAddress('0x6E9e7A8Fb61b0e1Bc3cB30e6c8E335046267D3A0')
// 0x6E9e...D3A0

shortenIfAddress('')
// ''

shortenIfAddress(undefined)
// ''

shortenIfAddress("i'm not an address")
// TypeError("Invalid input, address can't be parsed")
```

3.5.7 transactionErrored

Returns true if transaction failed or had an exception

Parameters

- `transaction: TransactionStatus` - transaction to check.

3.5.8 compareAddress

Returns 1 if first address is bigger than second address. Returns 0 if both addresses are equal. Returns -1 if first address is smaller than second address. If any address can't be parsed throws an error.

Parameters

- `firstAddress` - first address to compare
- `secondAddress` - second address to compare

Example

```
address1 = '0x24d53843ce280bbae7d47635039a94b471547fd5'
address2 = '0x24d53843ce280bbae7d47635039a94b471000000'
compareAddress(address1, address2)
// 1

address1 = '0x000000440ad484f55997750cfae3e13ca1751283'
address2 = '0xe24212440ad484f55997750cfae3e13ca1751283'
compareAddress(address1, address2)
// -1

address1 = 'im not an address'
address2 = '0xb293c3b2b4596824c57ad642ea2da4e146cca4cf'
compareAddress(address1, address2)
// TypeError("Invalid input, address can't be parsed")
```

3.5.9 addressEqual

Returns true if both addresses are them same. Returns false if addresses are different. Throws an error if address can't be parsed.

Parameters

- `firstAddress` - first address to compare
- `secondAddress` - second address to compare

Example

```
address1 = '0x24d53843ce280bbae7d47635039a94b471547fd5'
address2 = '0x24d53843ce280bbae7d47635039a94b471547fd5'
addressEqual(address1, address2)
// true

address1 = '0x24d53843ce280bbae7d47635039a94b471547fd5'
address2 = '0xe24212440ad484f55997750cfae3e13ca1751283'
addressEqual(address1, address2)
// false

address1 = 'im not an address'
address2 = '0xb293c3b2b4596824c57ad642ea2da4e146cca4cf'
compareAddress(address1, address2)
// TypeError("Invalid input, address can't be parsed")
```

CHAPTER FOUR

DEVELOPER TOOLS

4.1 Installation

In order to use useDapp developer tools you can install it for your browser.

1. Chrome Web Store (not available yet)
2. Firefox Add-ons (not available yet)

Alternatively you can built it from source and manually install it.

1. Clone this repository and build the source

```
git clone https://github.com/EthWorks/useDApp.git
cd useDApp
yarn && yarn build
```

2. If you are using the chrome browser:

1. Go to chrome://extensions/
2. Toggle developer mode
3. Click load unpacked
4. Open `useDapp/packages/extension/build`

3. If you are using firefox:

1. Go to about:debugging
2. Click This Firefox
3. Click Load Temporary Add-on
4. Open `useDapp/packages/extension/build/manifest.json`

4.2 List of events

4.2.1 Init

This is always the first event being displayed. It is fired when the useDapp library loads and contains a timestamp of that. All other events are timed relative to Init.

4.2.2 Network connected / disconnected

When the network changes on the provider this event is fired. Each network has its own name and color and other events are color coded according to the network they happened on. Those colors are also the exact same that Metamask uses, so you will feel right at home.

4.2.3 Account connected / disconnected

When the user connects an account or changes it this event gets fired. You will also see the first four digits of the address for easier visual grepping.

4.2.4 Block found

useDapp is constantly listening for new blocks. It does this so that it can maintain up to date blockchain state.

4.2.5 Calls updated

To understand the information that the extension presents it is necessary to understand how useDapp manages blockchain data. Whenever the app makes a call to a high level hook like `useTokenBalance` a low level `useContractCalls` is invoked. useDapp maintains a list of blockchain calls that will be made whenever a new block is found.

`useContractCalls` adds or removes calls from that list. Because making lots of blockchain calls at once is problematic the calls are actually aggregated together in one giant call to the Multicall contract. The resulting data is later decoded and returned back to higher level hooks.

The developer tools extension decodes and displays all the calls so that it is trivial to tell what is going on. Whenever the call list is modified the Calls updated event is emitted and a detailed breakdown of changes is made available to the user. The calls displayed can be Added, Removed or Persisted.

4.2.6 State updated

Every change to the call list and every new block being mined trigger a blockchain call. Once the call is resolved the state is updated with the new data. This event outlines which state entries have been updated or removed as well as what data was fetched from the blockchain.

4.2.7 Fetch error

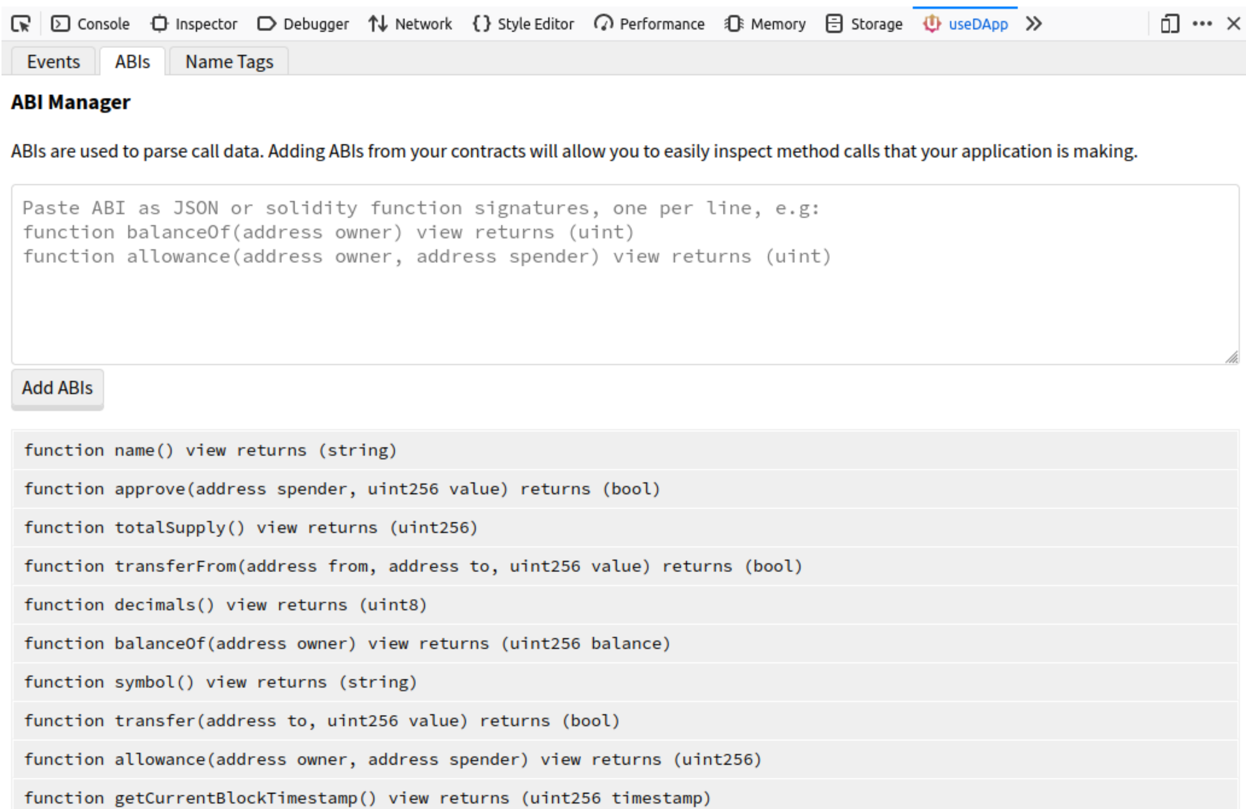
If the call to the blockchain results in an error this event will be emitted alongside the details of the error.

4.2.8 Error

Best explained in the web3-react documentation.

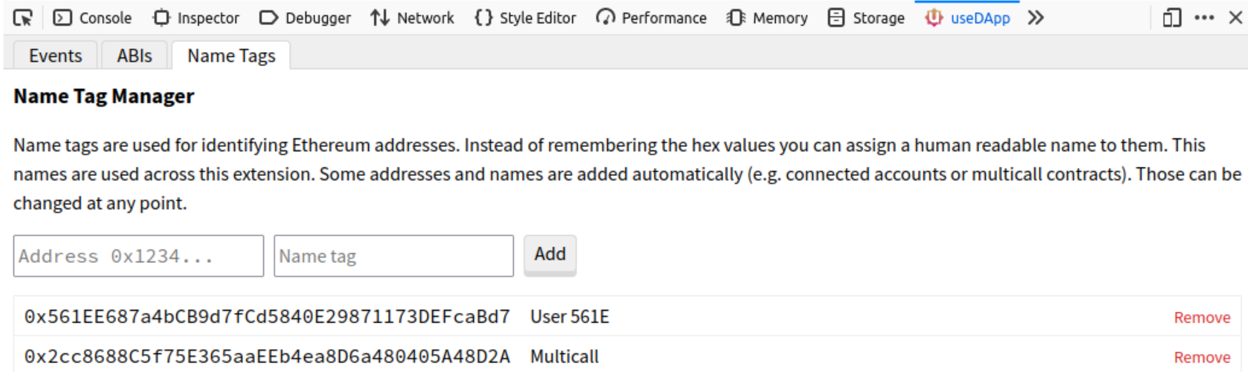
4.3 Adding custom ABIs

ABIs are used to parse call data. Adding ABIs from your contracts will allow you to easily inspect method calls that your application is making. You can add ABIs as Solidity function signatures or as JSON.



4.4 Adding name tags

Name tags are used for identifying Ethereum addresses. Instead of remembering the hex values you can assign a human readable name to them. This names are used across this extension. Some addresses and names are added automatically (e.g. connected accounts or multicall contracts). Those can be changed at any point.



CHAPTER

FIVE

COINGECKO API

5.1 Hooks

5.1.1 useCoingeckoPrice

Given base token name and the supported currencies to get token price from CoinGecko.

Parameters

- `base: string` - the token name that you can get at URL while search in CoinGecko. Or find the token from https://api.coingecko.com/api/v3/coins/list
- `quote: string` - (optional) the supported currencies in CoinGecko. Default quote is *usd*. See https://api.coingecko.com/api/v3/simple/supported_vs_currencies

Returns

- `string | undefined` - token price

Example

```
import { useCoingeckoPrice } from '@usedapp/coingecko'

const etherPrice = useCoingeckoPrice('ethereum', 'usd')

return etherPrice && (<p>$ {etherPrice}</p>)
```

5.1.2 useCoingeckoTokenPrice

Given token contract and the supported currencies to get token price from CoinGecko.

Parameters

- `contract: string` - the token contract
- `quote: string` - (optional) the supported currencies in CoinGecko. Default quote is *usd*. See https://api.coingecko.com/api/v3/simple/supported_vs_currencies
- `platform: string` - (optional) the platform issuing tokens. Default platform id is *ethereum*. See https://api.coingecko.com/api/v3/asset_platforms

Returns

- `string | undefined` - token price

Example

```
import { useCoingeckoTokenPrice } from '@usedapp/coingecko'

const WETH_CONTRACT = '0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2'
const wethPrice = useCoingeckoTokenPrice(WETH_CONTRACT, 'usd')

return wethPrice && (<p>$ {wethPrice}</p>)
```