OCTOBER 1, 2018 / #ETHEREUM

How to build an Ethereum Wallet web app

A review of the coolest parts of eth-hot-wallet

This article is a technical review of the interesting parts of **eth-hot-wallet**, an Ethereum wallet web app with erc20 token native support. The source code can be found on GitHub (MIT License).

Table of contents:

- Ethereum wallet as a web app
- The stack
- The containers of eth-hot-wallet
- Unified Design for Ethereum Wallet
- Redux and Redux-Saga
- *Secure* password generator
- eth-lightwallet and SignerProvider
- Encrypted offline storage
- Sending Ethereum using Web3.js
- Sending erc20 tokens using Web3.js
- Subscribing to Ethereum transaction life-cycle using Web3.js V1 and Redux-Saga channels
- Polling Ethereum blockchain and price data using Redux-Saga
- Keeping an eye on the bundle size
- Conclusion

Ethereum wallet as a web app

that can be used from any modern web-browser.

Moreover, recent improvements in PWA support significantly improve the user experience on mobile.

Pros:

- No additional software is required
- No installation of any kind is necessary
- Ability to use modern web development tools.
- Easy to deploy and to upgrade

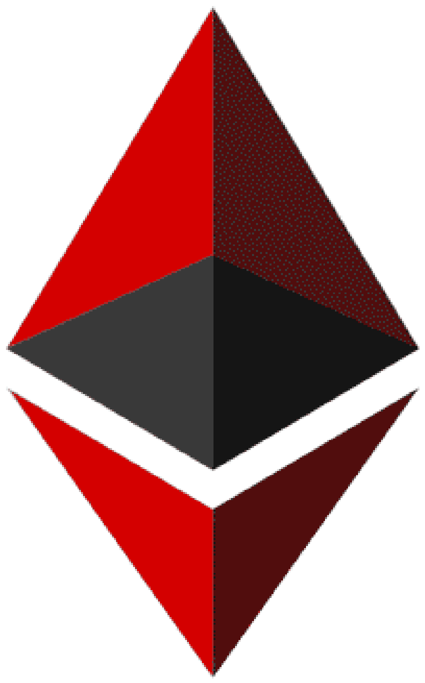
Cons:

- More prone to phishing attacks.
- Browser plugins might inject malicious code into the page.
- High loading time on slow internet connections
- Limited access to device storage

The fact that malicious browser extensions might inject JavaScript code in an attempt to extract the keys is significant. To migrate this risk, the user should be encouraged to turn off extensions (i.e. by using in incognito mode) or integrate the web with an external web3 provider such as MetaMask or Trust browser. Converting the web app into a desktop app is also a viable option.

As for phishing, the user should be encouraged to bookmark the page and access it via google search. It is highly unlikely for a phishing site to rank

minimum friction. In my opinion, the web is the best target platform for new apps.



eth-hot-wallet logo

The stack

Most of the code is dedicated to the **front-end:**

The final package consists of many packages as can be seen in package.json.

Top level components include:

- Eth-lightwallet — Lightweight JS Wallet for Node and the browser for keystore management

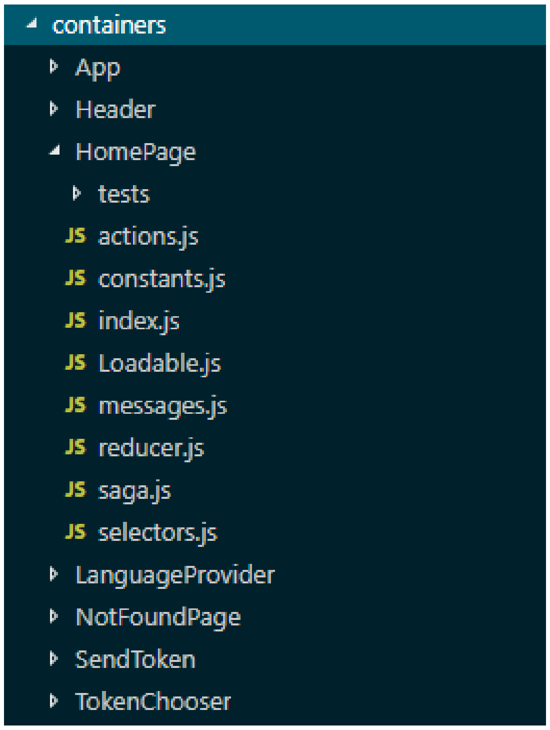
- Ant design — excellent set of UI components for react
- Webpack — A bundler for JavaScript and friends.

And for the **back-end:**

The final bundle is deployed directly to GitHub pages from a dedicated branch in the repository. There is no need for a back-end in the traditional scene.

To create the Testnet Ethereum faucet, we'll use a Serverless framework. It significantly improves the developer experience when using AWS Lambda. It is a very cost-effective solution that obliterates the need to maintain infrastructure, especially on low volume applications.

The containers of eth-hot-wallet



```
containers
  App
  Header
  HomePage
    tests
    actions.js
    constants.js
    index.js
    Loadable.js
    messages.js
    reducer.js
    saga.js
    selectors.js
  LanguageProvider
  NotFoundPage
  SendToken
  TokenChooser
```

eth-hot-wallet homepage container

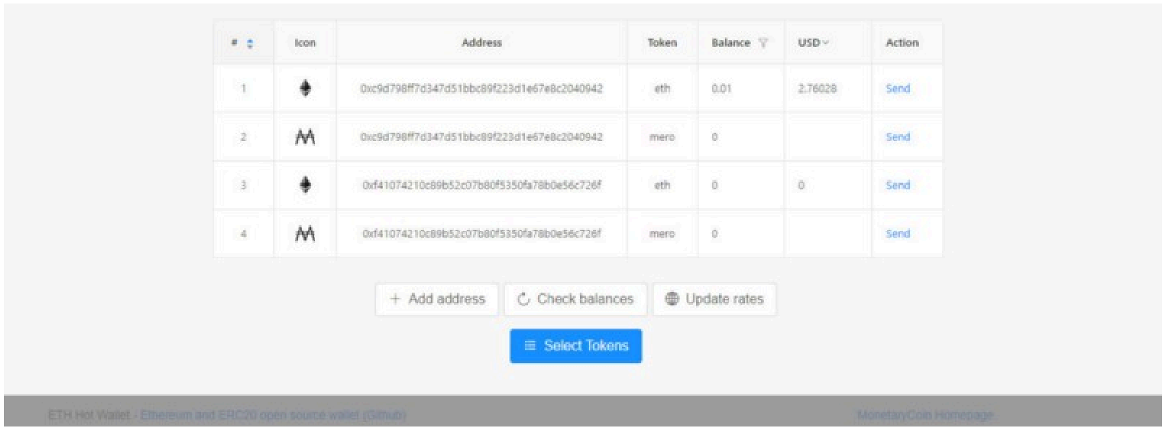
container (may) consist of the following ingredients:

- index.js — for rendering the GUI
- actions.js
- reducer.js
- saga.js
- selectors.js
- constants.js

As Dan Abramov stated, there is more than one approach for whether to use a component or a container. From my experience, if a component has more then ~8 attributes inside the application state, it should be separated into a new container. This is just a rule of thumb. The number of attributes may bloat over time. With complex components, it's better to have a unique container than to cluster the state of the main Container.

Not every container needs to have all the ingredients. In eth-hot-wallet, the `sendToken` container does not use its own Saga.js. We separated it so as not to burden the state of the homepage component.

The Homepage container



#	Icon	Address	Token	Balance	USD	Action
1		0xc9d798ff7d347d51bbc89f223d1e67e8c2040942	eth	0.01	2.76028	Send
2		0xc9d798ff7d347d51bbc89f223d1e67e8c2040942	mero	0		Send
3		0xf41074210c89b52c07b80f5350fa78b0e56c726f	eth	0	0	Send
4		0xf41074210c89b52c07b80f5350fa78b0e56c726f	mero	0		Send

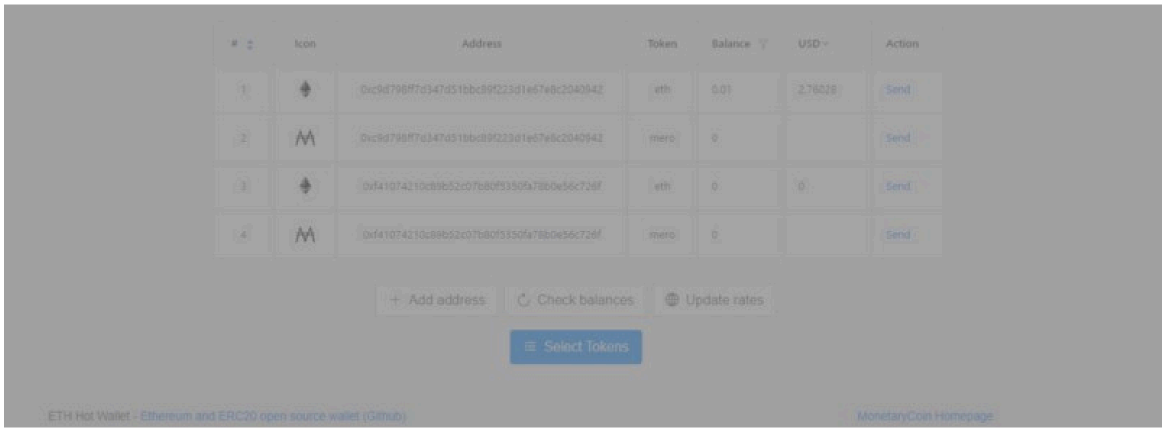
eth-hot-wallet homepage container

The primary container, where most of the action takes place, is the homepage container. In the homepage container, Saga.js is responsible for dealing with side effects. Besides GUI, its main responsibility will be dealing with **keystore operations**.

The ETH-Lightwallet package provides the keystore. All related operations including keys, seeds, encryption, importing, exporting are done in this section.

The Header container

The header demonstrates the fact that a container is much more than just a GUI component:



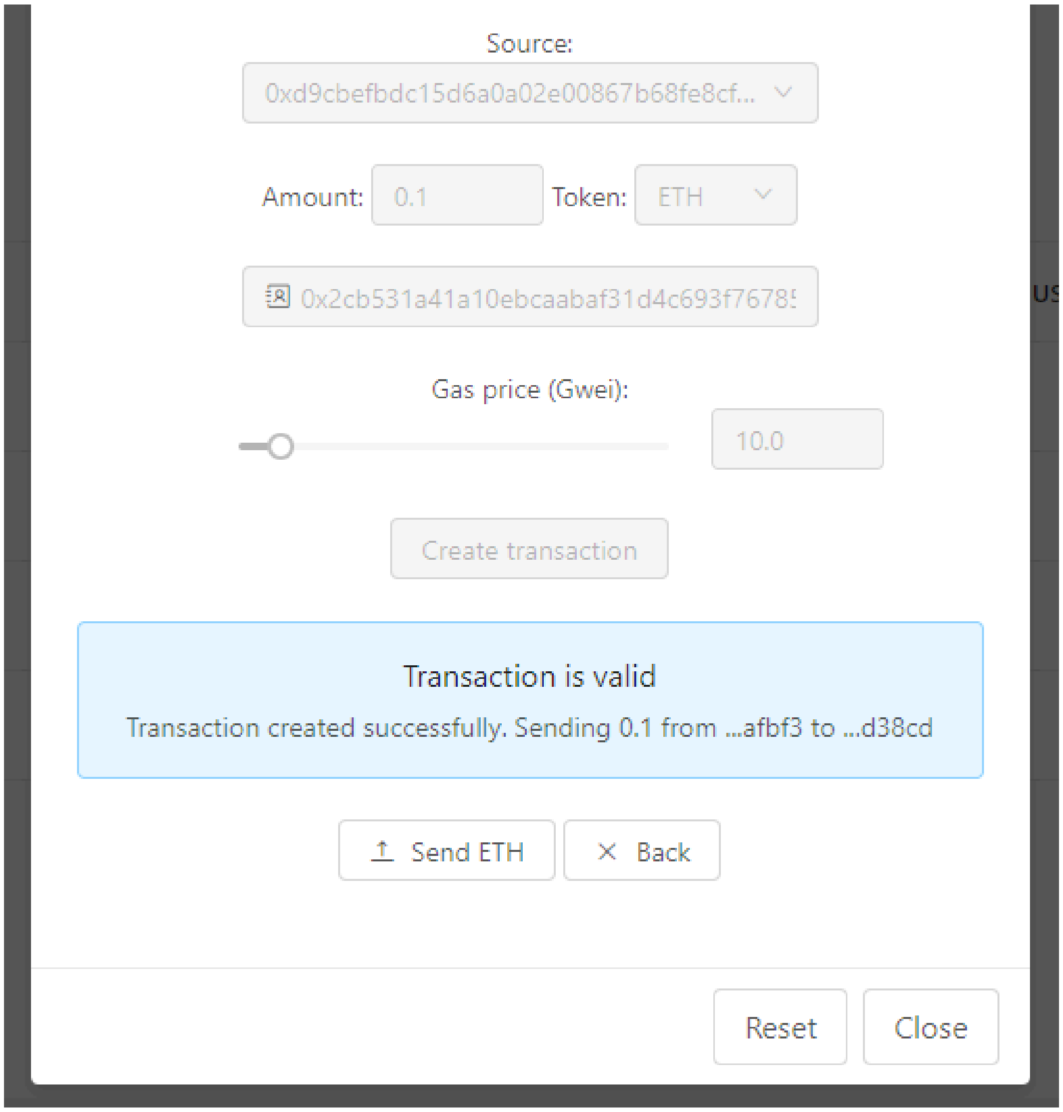
#	Icon	Address	Token	Balance	USD	Action
1		0xc9d798ff7d347d51bbc89f223d1e67e8c2040942	eth	0.01	2.76028	Send
2		0xc9d798ff7d347d51bbc89f223d1e67e8c2040942	mero	0		Send
3		0xf41074210c89b52c07b80f53350fa78b0e56c726f	eth	0	0	Send
4		0xf41074210c89b52c07b80f53350fa78b0e56c726f	mero	0		Send

eth-hot-wallet header container

This container might look simple at first with only a logo and a network selector. Does it even need to be in its own container? The answer is that in eth-hot-wallet **every network communication-related action and state management is done inside the header container**. More than enough for any container.

The SendToken container

SendToken is a modal that appears while the user selects to send Ether/ tokens.



Transaction is valid

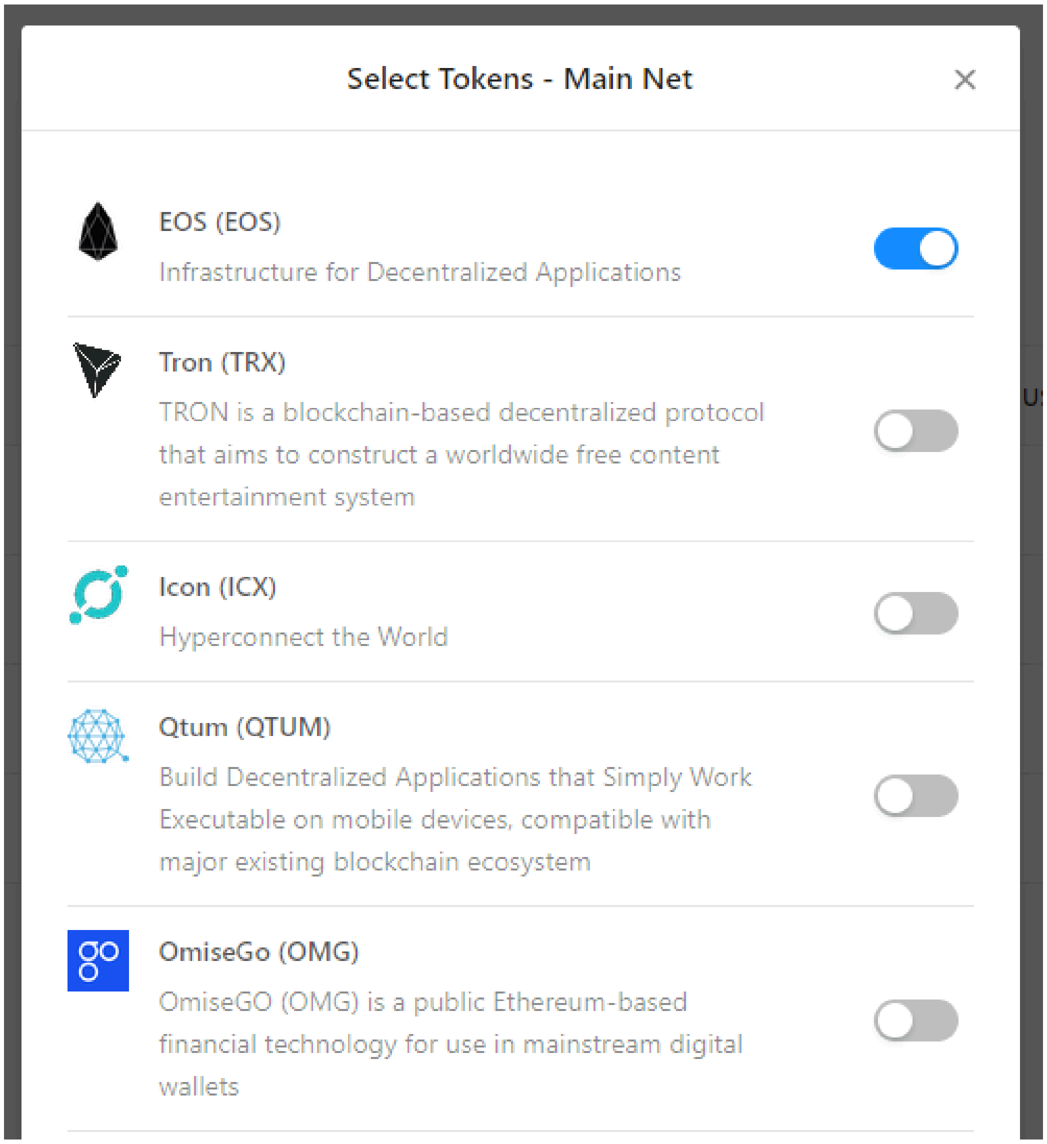
Transaction created successfully. Sending 0.1 from ...afbf3 to ...d38cd

eth-hot-wallet SendToken container

The modal includes some basic input verification, like amount and Ethereum address check. It does not use Saga.js to initiate side effects, but instead uses actions provided by the homepage and header containers.

We separated it into a new container to reduce clustering the state of the homepage container.

wallet will manage.


Select Tokens - Main Net

EOS (EOS)
Infrastructure for Decentralized Applications

Tron (TRX)
TRON is a blockchain-based decentralized protocol that aims to construct a worldwide free content entertainment system

Icon (ICX)
Hyperconnect the World

Qtum (QTUM)
Build Decentralized Applications that Simply Work Executable on mobile devices, compatible with major existing blockchain ecosystem

OmiseGo (OMG)
OmiseGO (OMG) is a public Ethereum-based financial technology for use in mainstream digital wallets

eth-hot-wallet TokenChooser container

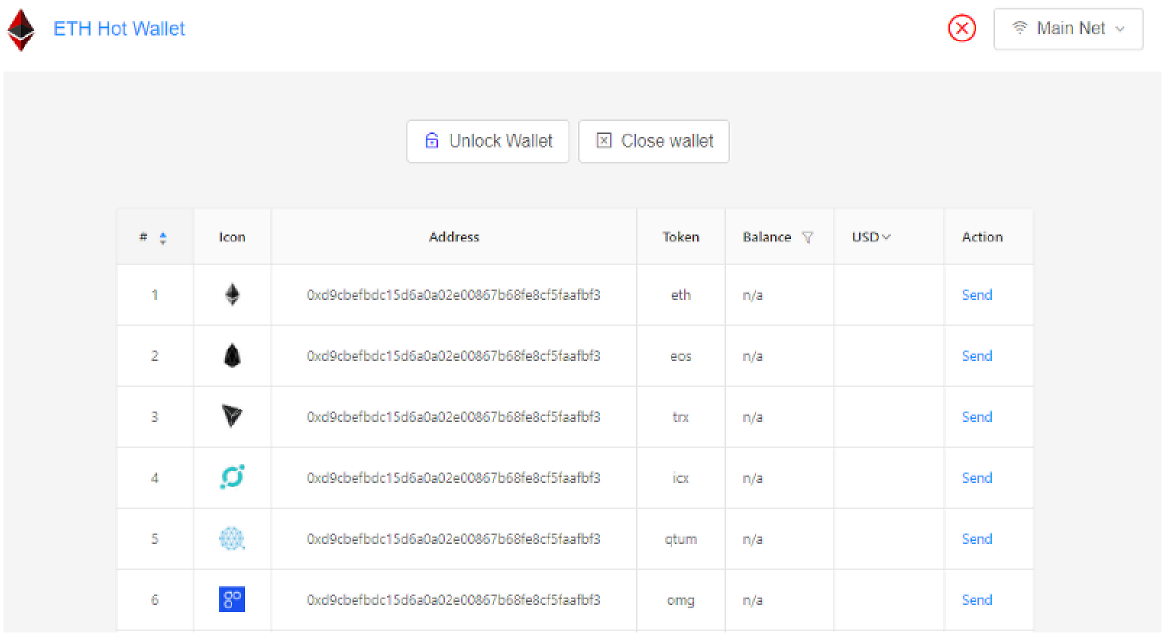
The `TokenChooser` name was selected in order not to be confused with the term "selector" which appears many times through the wallet code in a different context (reduxjs/reselect: Selector library for Redux).

Same as with the `SendToken` container, `TokenChooser` does not use its own Saga.js file but calls actions from the homepage container when

A Unified design for Ethereum Wallet

Since the appearance of the ERC20 standard (EIP20), it was obvious that tokens were going to be an important part of the Ethereum ecosystem. The Ethereum wallet was designed with a unified design approach in mind. **Ether and token should be treated equally from the user's perspective.**

Under the hood, the API for sending Ether and sending tokens is quite different. So is checking the balance, but they will appear the same on the GUI.



#	Icon	Address	Token	Balance	USD	Action
1		0xd9cbefbdc15d6a0a02e00867b68fe8cf5faafbf3	eth	n/a		Send
2		0xd9cbefbdc15d6a0a02e00867b68fe8cf5faafbf3	eos	n/a		Send
3		0xd9cbefbdc15d6a0a02e00867b68fe8cf5faafbf3	trx	n/a		Send
4		0xd9cbefbdc15d6a0a02e00867b68fe8cf5faafbf3	icx	n/a		Send
5		0xd9cbefbdc15d6a0a02e00867b68fe8cf5faafbf3	qtum	n/a		Send
6		0xd9cbefbdc15d6a0a02e00867b68fe8cf5faafbf3	omg	n/a		Send

A Unified design for Ethereum Wallet

To send Ether, we need to use native functions provided by the web3.js library, while sending tokens and checking balances involves interaction with a smart contract. More on this issue later.

GUI actions and user-triggered flows can be relatively easily managed by actions and reducers provided by Redux.

Aside from keeping the UI state, the Redux store also holds the key-store object (a partially encrypted JavaScript object supplied by eth-lightwallet). This makes the keystore accessible throughout the app by using a selector.

Redux-Saga is what makes the entire setup shine.

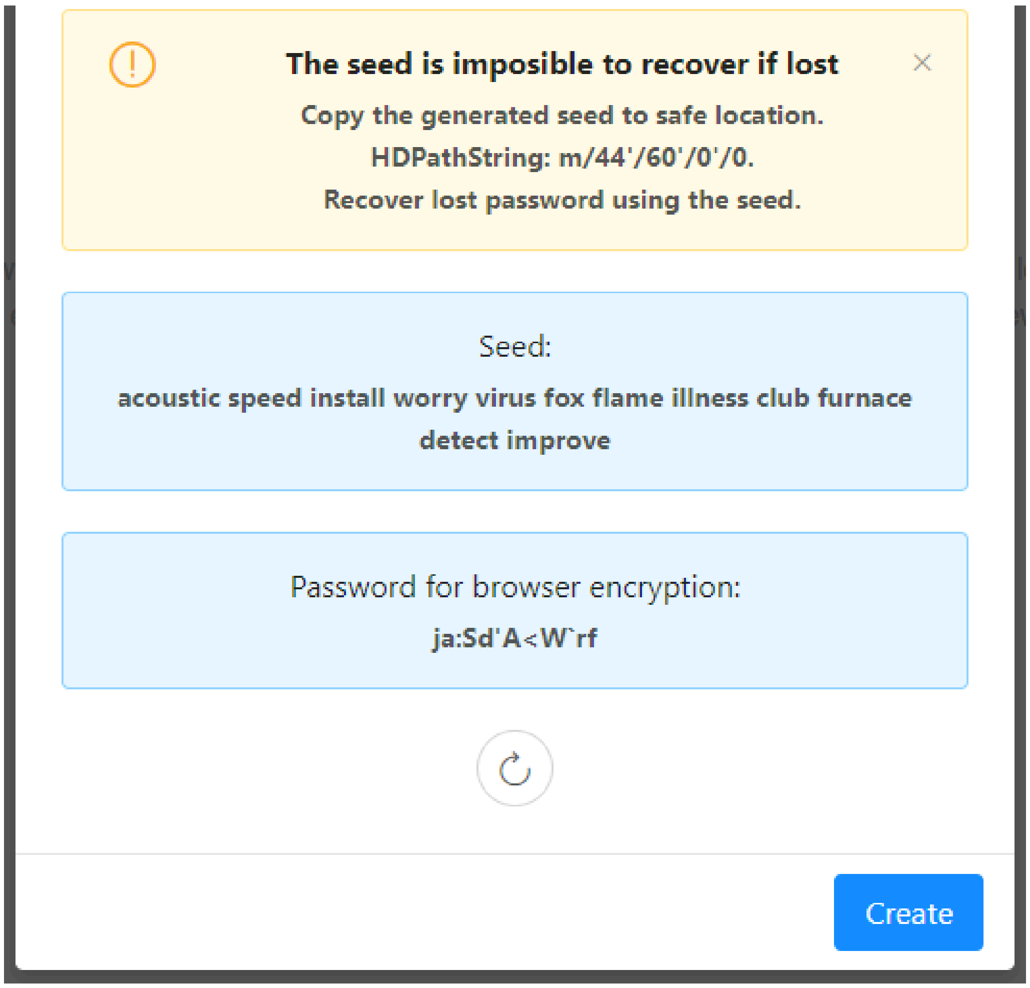
> `redux-saga` is a library that aims to make application side effects (i.e., asynchronous things like data fetching and impure things like accessing the browser cache) easier to manage, more efficient to execute, easy to test, and better at handling failures.

Saga.js uses Generators to make **asynchronous flows easy to read and write**. By doing so, these asynchronous flows look like your standard synchronous JavaScript code (kind of like `async` / `await` but with more customization options).

In the case of Ethereum wallet, by using Saga we get a comfortable way to handle asynchronous actions such as rest API calls, keystore actions, Ethereum blockchain calls via web3.js, and more. All the requests are cleanly managed in one place, no callback hell, and very intuitive API.

Example usage for redux-saga:

Secure password generator


The seed is imposible to recover if lost

Copy the generated seed to safe location.

HDPathString: m/44'/60'/0'/0.

Recover lost password using the seed.

Seed:

acoustic speed install worry virus fox flame illness club furnace detect improve

Password for browser encryption:

ja:Sd'A<W`rf

A seed and a secure password is auto-generated for the user

To adequately secure the user's keystore, we need to encrypt it with a strong password. When using eth-lightwallet, the password needs to be provided during the initiation of the hd-wallet.

Let's assume that we have a function called `generateString` , which can provide genuinely random strings at any length.

If the user wants to generate a new wallet, we will produce the following parameters:

`generateString` implementation: We will use the relatively new **window.crypto API** to get random values (currently supported by all major browsers).

Eth-hot-wallet implementation is based on the following code to generate **random but human-readable strings**:

After the user has accepted the password and the seed, we can use the values and generate the new wallet.

eth-lightwallet and SignerProvider

1. LightWallet is intended to be a signing provider for the Hooked Web3 provider.
2. Hooked Web3 provider has been deprecated, and currently the author recommends the package ethjs-provider-signer as an alternative.
3. At the moment of writing, there is a bug in ethjs-provider-signer that prevents the display of error messages. The bug was fixed but didn't merge back into the main branch. Those error messages are critical for this setup to function correctly.

Bottom line: Use eth-lightwallet with this version of ethjs-provider-signer: https://github.com/ethjs/ethjs-provider-signer/pull/3 to save time on trial and error.

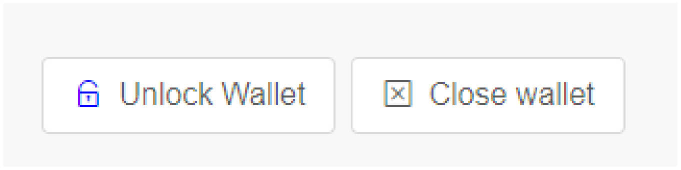
Encrypted offline storage

safekeeping the password and provides it with any action.

eth-hot-wallet uses Store.js — *Cross-browser storage for all use cases, used across the web*. Store.js allows us to store the encrypted keystore easily and extract it back from storage when the webpage is accessed.

When the wallet is loaded for the first time, it will check if there is a keystore in local storage and will auto load it to Redux state if so.

At this point, we can read the public data of the keystore but not the keys. To display public data before the user enters the encryption password, we need an additional operation mode: **loaded and locked.** In this mode, the wallet will display the addresses and fetch the balances but will not be able to perform operations such as sending transactions or even generating new addresses. Triggering any of those actions will prompt for the user's password.



locked wallet after loading from local storage

Sending Ethereum using Web3.js

When using Web3.js@0.2.x, the function `sendTransaction` is provided in the following form:

```
web3.eth.sendTransaction(transactionObject [, callback])
```

`sendTransaction` function with a promise:

This way we continue regular Saga.js execution after `sendTransaction` is called.

Sending erc20 tokens using Web3.js

The Ethereum blockchain does not provide primitives that encapsulate token functionality, nor should it. Every token deployed on Ethereum is, in fact, a program that corresponds to the **eip20 specification**. Since the Ethereum virtual machine (EVM) is Turing complete (with some restrictions), every token might have a different implementation (even for the same functionality). What unifies all those programs under the term "token" is that they provide the same API as defined by the specification.

When we are sending a token on Ethereum, we are interacting with a smart contract. To communicate with a smart contract we need to know its API, the format for sharing contract's API called Ethereum Contract ABI.

We will store the erc20 ABI as part of our JavaScript bundle and instantiate a contract during the program run-time:

```
const erc20Contract = web3.eth.contract(erc20Abi);
```

After contract setup, we can easily interact with it programmatically using the Web3.js contract API.

For each token we will need a dedicated contract instance:

functions by calling the desired function straight from JavaScript:

See Web3.js contract API for the full details.

We will promisify the `tokenContract.transfer.sendTransaction` to use it with our redux-saga flow:

It is possible to use es6-promisify or similar instead of writing the promise directly, but I prefer the direct approach in this case.

Subscribing to Ethereum transaction life-cycle using Web3.js V1 and redux-saga channels

eth-hot-wallet uses web3.js v0.2.x and does not support this feature at the moment. The example is provided by another project. It is an important feature and should be used extensively.

The new version of Web3.js (V1.0.0) is shipped with a new contract API that can inform us about transaction life-cycle changes.

Enter the `PromiEvent` : A promise combined event emitter.

```
web3.eth.sendTransaction({...}).once('transactionHash', function(hash)
```

`web3.eth.sendTransaction()` will return an object (a promise) that will resolve once the transaction is mined. The same object will allow us to subscribe to `‘transactionHash’` , `‘receipt’` , `‘confirmation’` and `‘error’` events.

web app with the help of Saga.js channels. The motivation is to update the application state (Redux store) once a change to the transaction state is detected.

In the following example, we will create a 'commit' transaction to an arbitrary smart contract and update app state when we get `'transactionHash'`, `'receipt'` and `'error'` events.

We need to initialize the new channel and fork a handler:

The handler will catch all channel events and will call the appropriate Redux action creator.

Once the channel and the handler are both ready and the user initiates the transaction, we need to register to the generated events:

In fact, we don't need a new channel for each transaction and can **use the same channel for all types of transactions.**

The full source code of this example can be found here.

Polling Ethereum blockchain and price data using redux-saga

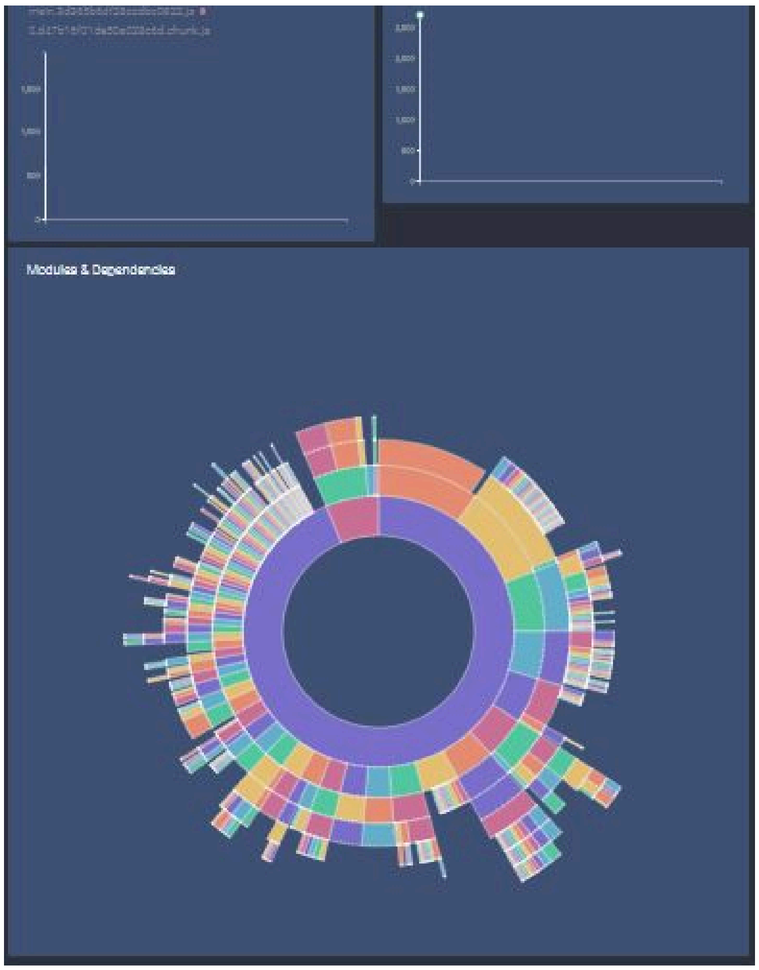
There are several ways to watch for blockchain changes. It is possible to use Web3.js to subscribe to events or we can poll the blockchain by ourselves and have more control over some aspects of polling.

In eth-hot-wallet, the wallet is polling the blockchain periodically for balance changes and Coinmarketcap API for price changes.

`checkAllBalances` function is called. It can end with one of two possible outcomes: `CHECK_BALANCES_SUCCESS` or `CHECK_BALANCES_ERROR` . Each one of them will be caught by `watchPollData()` to wait X seconds and call `checkAllBalance` again. This routine will continue until `STOP_POLL_BALANCES` is caught by `watchPollData` . After that, it is possible to resume the polling by submitting `CHECK_BALANCES` action again.

Keeping an eye on the bundle size

When building web apps using JavaScript and npm, it might be tempting to add new packages without analyzing the footprint increase. Eth-hot-wallet uses webpack-monitor to display a chart of all the dependencies and **the differences between each build**. It allows the developer to see the bundle size change clearly after each new package is added.



webpack-monitor example chart

Webpack monitor also can help in finding the most demanding dependencies and might even surprise the developer by **highlighting the dependencies that do little for the app but contribute a lot to the bundle size.**

Webpack-monitor is easy to integrate and is definitely worth including in any webpack based web app.

Conclusion

continue and create a successful wallet.

Building a wallet can also be a great introduction to the world of Ethereum since most distributed applications (DApps) require a similar set of capabilities both from the front-end and blockchain perspective.

ETH Hot Wallet - Ethereum Wallet with ERC20 support
*ETH Hot wallet is an Ethereum wallet with ERC20 Support. The keys are generated inside the browser and never sent…*eth-hot-wallet.com

In case you have any questions regarding eth-hot-wallet or any related subject, feel free to contact me via Twitter or open an issue on GitHub.
