Burner Wallet Documentation

David Mihal

Jun 11, 2020

Developer Documentation:

CHAPTER 1

Getting started

Simple, customized wallet

Do you want to customize your own version of the wallet? Check out the simple application in the *basic-wallet* directory.

Alternatively, visit https://burnerfactory.com to create your own wallet without writing any code!

1.1 Setting Infura key

By default, *basic-wallet* uses the InfuraGateway for connecting to commonly used Ethereum chains.

The entry point takes an Infura key from the *REACT_APP_INFURA_KEY* environment variable. For your wallet to function correctly, you must create a file named *.env* in the *basic-wallet* folder with the following value:

```
REACT_APP_INFURA_KEY=<your infura key>
```

You can generate an Infura key at https://infura.io/

1.2 Add a custom token

You can add any ERC20 token to your wallet by constructing a new ERC20Asset and adding it to the asset list.

The *id* parameter is the internal ID used by the wallet, while the *name* parameter is the display name that will be displayed to the user. *network* is the chain ID of the chain the token is deployed to (*'1'* for mainnet, *'100'* xDai, etc). *address* is the address where the token contract is deployed.

```
import { xdai, dai, eth, ERC20Asset } from '@burner-wallet/assets';

const bos = new ERC20Asset({
  id: 'bos',
  name: 'Boston Token',
```

(continues on next page)

(continued from previous page)

```
  network: '100',
  address: '0x52ad726d80dbb4A9D4430d03657467B99843406b',
});

const core = new BurnerCore({
  assets: [bos, xdai, dai, eth],
});
```

1.3 Local developer wallet

Are you a developer, hoping to test changes to other modules in this project (modern-ui, ui-core or various plugins)?

Run *yarn start-local* in the project root. This will start a wallet on localhost:3000 that is connected to your local Ganache instance (connecting to node http://localhost:8545 by default).

Before the wallet server launches, a script create a pre-filled account. This account will hold 1 Ganache ETH and 100 test tokens.

Note that Metamask will override the local account, disable it or open in incognito mode for local development.

CHAPTER 2

Wallet Architecture

A Burner Wallet is created by composing a number of independent modules into a single web application.

The following will outline the various modules from the lowest-level up to the higest level.

2.1 Burner Core

The foundation of a Burner Wallet is the Burner Core modules. These modules are UI-agnostic and only handle blockchain calls.

2.1.1 BurnerCore

The BurnerCore class (from the `@burner-wallet/core` package) routes all blockchain communications between various modules. Burner Wallets are inheritly multi-chain applications, so this module must route messages and transactions to the correct chain.

The BurnerCore class is also responsible for storing a log of historical on-chain events.

2.1.2 Gateways

Gateways can be thought of as "wrapped RPC providers", as they transmit data from the wallet to endpoints such as Infura.

The `@burner-wallet/core` module contains some standard gateways: InfuraGateway, XDaiGateway and InjectedGateway (which transmits messages through an injected Web3 provider such as Metamask).

Theoretically, a gateway could also be used to connect to non-standard networks, such as a state channel or centralized service.

2.1.3 Signers

Signers maintain the "accounts" available to the wallet, and are responsible for cryptographically signing transactions and messages.

The `@burner-wallet/core` module contains two standard signers: LocalSigner (which stores private keys in the browser's LocalStorage) and InjectedSigner (which uses the account from an injected Web3 provider such as Metamask).

Signers can also provide more advanced functionality, such as the FortmaticSigner which uses the Fortmatic service for signing, or the ContractWalletSigner, which generates counterfactual contract wallets from other available accounts.

2.1.4 Assets

A standard interface is used for defining all fungible digital assets that are handled by the wallet. Assets are responsible for sending assets, checking account balances and watching for incoming transactions.

In addition to defining the abstract `Asset` class, the `@burner-wallet/assets` package contains standard assets *eth*, *dai*, and *xdai*. Developers can easily define their own assets using the classes `NativeAsset` (for assets such as ETH, xDai, testnet ETH) as well as `ERC20Asset` and `ERC777Asset` for tokens.

2.2 Burner Wallet UI

The Burner Wallet interface is defined using React Components.

2.2.1 UI Core

Note: developers only need to farmiliarize themselves with the UI Core module if they want to completely change the wallet interface. Most developers only need to use the Modern UI package.

The UI Core module is the root module of the Burner application. This module handles all URL routing, all plugins, and routes data between the Burner Core instance and React components.

While the UI Core consists of a set of React components, it is design agnotstic.

2.2.2 UI Implementations (ModernUI)

Visual components are defined in a separate module. The ModernUI (`@burner-wallet/modern-ui`) module should be sufficent for most applications, however developers may also use the ClassicUI module (which resembes the original Austin Griffith Burner Wallet) or create their own UI implementation.

2.3 Plugins

Wallet functionality can be extended by defining plugins, which are Javascript packages that conform to the Burner Wallet Package API.

At a simple level, plugins can access Web3 instances and insert React components into the wallet interface. This allows plugins to provide any functionality that is possible in a standalone dapp.

Plugins also have the ability to extend other parts of the wallet, such as using the QR code scanner, or providing human-readable names for addresses.

For more information, see the Plugin API section.

2.3.1 Exchange Plugin

The "burner-wallet-2" repository contains a number of offically-supported plugins, such as the ENS Plugin or Recent Accounts Plugin. However, the Exchange Plugin plays an important role, as it allows uses to convert between different asset types.

The Exchange Plugin itself is extendable by providing it a set of "Exchange Pairs", which define mechanisms for converting from one asset to another. Two Exchange Pair classes are provided by default: the Uniswap pair which allows converstion between any asset supported by Uniswap, and the xDai Bridge which facilitates transfers between Dai and xDai.

CHAPTER 3

Plugin Development Guide

The Burner Wallet 2 is a robust, extendable implementation of Austin Griffith's famous Burner Wallet. A simple wallet can be built and customized in only a single Javascript file. The wallet can be extended using Plugins, which require only basic knowledge of React and the Web3 library.

For assistance with plugin development, feel free to contact me, David Mihal (@dmihal on Telegram and Twitter)

3.1 What can a Burner Wallet Plugin do?

A Burner Wallet plugin can do anything that any other DApp can do! At a basic level, Burner Wallet plugins are provided with a Web3 instance and the ability to insert React components into the wallet interface.

3.2 Sample Plugins

There are many plugins that can be viewed for reference:

- The Burner Wallet 2 repo contains a number of general-purpose plugins, such as the exchange, ENS plugin, and recent accounts plugin.
- The Burner Factory Plugins repo contains some more advanced plugins such as the collectables plugin and push notification plugin.

3.3 Getting Started

3.3.1 Option 1: Clone the sample project

The easiest way to get started is to start from the sample plugin repository:

Zip file: https://github.com/burner-wallet/sample-plugin/archive/master.zip

Fork the repo: https://github.com/burner-wallet/sample-plugin

1. Download the Zip file or fork the repo.
2. `cd` into the project directory, and run `yarn install`
3. In order to use your app on Mainnet or testnets, you'll need an Infura API key. See the "envionment variables" section
4. Start your wallet! Run yarn start-local to run a wallet that connects to a local Ganache instance, or run yarn start-basic to run a wallet that connects to the Mainnet and xDai chains.
5. Navigate your browser to http://localhost:3000!

3.3.2 Option 2: Start from scratch

Creating a wallet with the Burner Wallet 2 does not require cloning any repository, as all components are NPM packages. Simply create a new react project, install the packages, and create a plugin entry point.

1. Run `create-react-app my-wallet` and `cd` to `my-wallet`
2. Run `yarn add @burner-wallet/core @burner-wallet/assets @burner-wallet/modern-ui @burner-wallet/types`
3. Paste the following code in index.tsx to create a simple wallet:

1. Add an Infura key (see Option 1)
2. Start your wallet! Run yarn start-local to run a wallet that connects to a local Ganache instance, or run yarn start-basic to run a wallet that connects to the Mainnet
3. Navigate your browser to http://localhost:3000!

3.3.3 Environment Variables

3.3.4 Typescript Support

The Burner Wallet 2 is natively built with Typescript, but you can write your plugins in TypeScript or plain Javascript.

If you would like to write a plugin using Typescript, the only dependency you need is @burner-wallet/types, which includes all the typings you'll need.

If you want to write in plain javascript, then you don't need any dependencies!

3.4 Create plugin entry point

If you have started from the sample project, you may skip this section.

Plugins are created by defining a class that implements the following Plugin interface.

```
export interface Plugin {
  initializePlugin(context: BurnerPluginContext): void;
}
```

Create a new file with the following content:

```
import { Plugin, BurnerPluginContext } from '@burner-wallet/types'

export default class MyPlugin implements Plugin {
  initializePlugin(pluginContext: BurnerPluginContext) {
    pluginContext.addElement('home-top', () => "Hello, World");
  }
}
```

Import your class in your wallet entry point, and add it to the "plugins" array.

```
import MyPlugin from "../my-plugin/MyPlugin";

const BurnerWallet = () =>
  <BurnerUI
    core={core}
    plugins={[new MyPlugin()]}
  />
```

Now, start your wallet development server by running `yarn start`

Navigate to `http://localhost:3000`. You should see a normal wallet with "Hello, World" written at the top!

3.5 Adding a page

Note: this step is completed by default in the sample project:

Create a new file with the following content:

```
import React from 'react';
import { PluginPageContext, Asset } from '@burner-wallet/types';

const MyPage: React.FC<PluginPageContext> = ({ BurnerComponents, assets,␣
↪defaultAccount }) => {
  const { Page } = BurnerComponents;
  return (
    <Page title="My Page">
      <div>Account: {defaultAccount}</div>
      <div>Assets: {assets.map((asset: Asset) => asset.name).join(', ')}</div>
    </Page>
  );
};

export default MyPage;
```

Now import and add that page to the entry-point class:

```
import { Plugin, BurnerPluginContext } from '@burner-wallet/types';
import MyPage from './MyPage';

export default class MyPlugin implements Plugin {
  initializePlugin(pluginContext: BurnerPluginContext) {
    pluginContext.addPage('/mypage', MyPage);
  }
}
```

We also want to add a button to the home page so that users can access the page:

```
import { Plugin, BurnerPluginContext } from '@burner-wallet/types';
import MyPage from './MyPage';

export default class MyPlugin implements Plugin {
  initializePlugin(pluginContext: BurnerPluginContext) {
    pluginContext.addPage('/mypage', MyPage);
    pluginContext.addButton('apps', 'My Page', '/mypage');
  }
}
```

3.6 Interacting with a contract

This section will allow our plugin to interact with a smart contract using Web3.

First, we need to save the `pluginContext` to an instance variable so that it can be accessed outside of the initializePlugin function.

```
import { Plugin, BurnerPluginContext } from '@burner-wallet/types';
import MyPage from './MyPage';

export default class MyPlugin implements Plugin {
  private pluginContext?: BurnerPluginContext;

  initializePlugin(pluginContext: BurnerPluginContext) {
    this.pluginContext = pluginContext;

    pluginContext.addPage('/mypage', MyPage);
    ...
  }
}
```

Now, let's import our contract ABI from a JSON file and add a getContract function:

```
import { Plugin, BurnerPluginContext } from '@burner-wallet/types';
import MyPage from './MyPage';
import gameAbi from './game-abi.json';

const GAME_CONTRACT_ADDRESS = '0x0123456789012345678901234567890123456789';

export default class MyPlugin implements Plugin {
  private pluginContext?: BurnerPluginContext;

  initializePlugin(pluginContext: BurnerPluginContext) {
    this.pluginContext = pluginContext;

    pluginContext.addPage('/mypage', MyPage);
    ...
  }

  getContract() {
    const web3 = this.pluginContext!.getWeb3('100' /* xDai */);
    return new web3.eth.Contract(gameAbi as any, GAME_CONTRACT_ADDRESS);
  }
}
```

Note the exclamation point after `this.pluginContext`. Technically, it's possible that pluginContext is `undefined`, however we can assume that no code will run until after initializePlugin has been called.

In that example, we have hardcoded the contract address and network ID. If you want the plugin to be more flexible and reusable, you can also make those constructor arguments like this:

```
import { Plugin, BurnerPluginContext } from '@burner-wallet/types';
import MyPage from './MyPage';
import gameAbi from './game-abi.json';

const GAME_CONTRACT_ADDRESS = '0x0123456789012345678901234567890123456789';

export default class MyPlugin implements Plugin {
  private pluginContext?: BurnerPluginContext;
  private contractAddress: string;
  private chainId: string;

  constructor(contractAddress: string, chainId: string) {
    this.contractAddress = contractAddress;
    this.chainId = chainId;
  }

  initializePlugin(pluginContext: BurnerPluginContext) {
    this.pluginContext = pluginContext;

    pluginContext.addPage('/mypage', MyPage);
    ...
  }

  getContract() {
    const web3 = this.pluginContext!.getWeb3(this.chainId);
    return new web3.eth.Contract(gameAbi as any, this.contractAddress);
  }
}
```

Now, let's add a some contract calls.

```
import { Plugin, BurnerPluginContext } from '@burner-wallet/types';
import MyPage from './MyPage';
import gameAbi from './game-abi.json';

const GAME_CONTRACT_ADDRESS = '0x0123456789012345678901234567890123456789';

export default class MyPlugin implements Plugin {
  private pluginContext?: BurnerPluginContext;
  private contractAddress: string;
  private chainId: string;

  constructor(contractAddress: string, chainId: string) {
    this.contractAddress = contractAddress;
    this.chainId = chainId;
  }

  initializePlugin(pluginContext: BurnerPluginContext) {
    this.pluginContext = pluginContext;

    pluginContext.addPage('/mypage', MyPage);
    ...
```

(continues on next page)

(continued from previous page)

```
  }

  getContract() {
    const web3 = this.pluginContext!.getWeb3(this.chainId);
    return new web3.eth.Contract(gameAbi as any, this.contractAddress);
  }

  async getScore(address: string) {
    const contract = this.getContract();
    const score = await contract.methods.getScore(address).call();
    return score;
  }

  async buyTokens(address: string, numTokens: string) {
    const contract = this.getContract();
    await contract.methods.buyTokens(numTokens).send({ from: address });
  }
}
```

Our plugin can now fetch data by calling `getScore` on this contract, as well as send a transaction to the contract's `buyTokens` function.

Now, we can integrate these functions into our page:

```
import React, { useState, useEffect } from 'react';
import { PluginPageContext } from '@burner-wallet/types';
import MyPlugin from './MyPlugin';

const MyPage: React.FC<PluginPageContext> = ({ BurnerComponents, defaultAccount }) =>
↪{
  const [score, setScore] = useState('');
  const [numTokens, setNumTokens] = useState('0');

  const _plugin = plugin as MyPlugin;

  useEffect(() => {
    _plugin.getScore(defaultAccount).then(score => setScore(score));
  }, []);

  const buyTokens = async () => {
    await _plugin.buyTokens(defaultAccount, numTokens);
    setNumTokens('0');

    const score = await _plugin.getScore(defaultAccount);
    setScore(score);
  };

  const { Page, Button } = BurnerComponents;
  return (
    <Page title="My Page">
      <div>Score: {score}</div>
      <div>
        <input type="number" value={numTokens} onChange={e => setNumTokens(e.target.
↪value)} />
        <Button onClick={buyTokens}>Buy Tokens</Button>
      </div>
    </Page>
```

(continues on next page)

(continued from previous page)

```
  );
};
```

Users can now interact with the deployed contract!

CHAPTER 4

API Reference

4.1 Plugin Class

All plugins are defined by creating a simple class that implements the following `initializePlugin` method:

```
interface Plugin {
  initializePlugin(context: BurnerPluginContext): void;
}
```

Inside this method, plugins declare all wallet integrations using the Plugin Context API.

The Plugin Class is also an ideal location to define blockchain logic, since React components will have access to the plugin instance.

```
import { Plugin, BurnerPluginContext } from '@burner-wallet/types'
import Game from './ui/Game';
import gameAbi from './gameAbi.json';

const GAME_ADDRESS = '0x0123456789012345678901234567890123456789';

export default class GamePlugin implements Plugin {
  private pluginContext?: BurnerPluginContext;

  initializePlugin(pluginContext: BurnerPluginContext) {
    this.pluginContext = pluginContext;

    pluginContext.addPage('/game', Game);
    pluginContext.addButton('apps', '/game', {
      'description': 'Play this fun game!',
    });
  }

  getContract() {
    const web3 = this.pluginContext!.getWeb3('1');
```

(continues on next page)

(continued from previous page)

```
    const contract = new web3.eth.Contract(gameAbi, GAME_ADDRESS);
    return contract;
  }

  async getScore(userAddress) {
    const contract = this.getContract();
    const score = await contract.methods.getScore(userAddress).call();
    return score;
  }
}
```

4.2 Plugin Context

When the wallet is loaded, the wallet will call the `initializePlugin(pluginContext)` function for each plugin. This function is provided an object with the following interface:

```
interface BurnerPluginContext {
  addElement: (position: string, Component: PluginElement, options?: any) => void;
  addButton: (position: string, title: string, path: string, options?: any) => any;
  addPage: (path: string, Component: PluginPage) => any;
  getAssets: () => Asset[];
  getWeb3: (network: string, options?: any) => Web3;
  addAddressToNameResolver: (callback: AddressToNameResolver) => void;
  onAccountSearch: (callback: AccountSearchFn) => void;
  onQRScanned: (callback: QRScannedFn) => void;
  onSent: (callback: TXSentFn) => void;
}
```

4.2.1 addElement

```
pluginContext.addElement(position: string, Component: React.ComponentType, [options?:␣
↪any])
```

Adds a React component to a defined position in an existing wallet page.

Paramaters

- `position`: The defined position in the application to insert the component at. The ModernUI defines the following positions:
  - `home-top`
  - `home-middle`
  - `home-bottom`
  - `home-tab`: Adds component as a tab on the home page. Accepts an option with the value `title`
  - `advanced`
- `Component`: The React component to be used. The component will receive the Burner Plugin Component Props
- `options`: Some positions may expect additional options to be provided

Example

```
import { Plugin, BurnerPluginContext } from '@burner-wallet/types';
import BalanceTab from './Username';
import BalanceTab from './BalanceTab';

export default class BalancePlugin implements Plugin {
  initializePlugin(context: BurnerPluginContext) {
    context.addElement('home-middle', Username);
    context.addElement('home-tab', BalanceTab, { title: 'Cash' });
  }
}
```

4.2.2 addPage

```
pluginContext.addPage(path: string, Component: React.ComponentType)
```

Creates a new page in the wallet with it's own URL route.

4.2.3 addButton

```
pluginContext.addButton(position: string, title: string, path: string, [options?:␣
↪any])
```

Add a button do a pre-defined location in the wallet.

Paramaters

- `position`: A button position defined by the Wallet UI. Currently, ModernUI only supports "app", while ClassicUI only supports "home"
- `title`: The text to display in the button
- `path`: The URL path to navigate to when clicked
- `options`: Additional data to provide the button. For example, ModernUI accepts `description` and `icon` values.

Example

```
import { Plugin, BurnerPluginContext } from '@burner-wallet/types';

export default class MenuPlugin implements Plugin {
  initializePlugin(context: BurnerPluginContext) {
    pluginContext.addButton('apps', 'Drink Menu', '/menu', {
      description: 'Order drinks from the bar',
      icon: '/beericon.png',
    });
  }
}
```

4.2.4 getAssets

```
pluginContext.getAssets(): Asset[]
```

Returns an array of all Asset objects used by the wallet.

4.2.5 getWeb3

```
pluginContext.getWeb3(chain: string): Web3
```

Returns a Web3 instance for the requested chain. This allows lower-level blockchain calls (querying transactions & blocks) as well as constructing Web3 Contract instances.

Note: Burner Wallet uses Web3 v1.2.x

Paramaters

- `chain`: The chain ID for the requested chain (ex: '1' for mainnet, '42' for Kovan testnet, '100' for xDai)

Example

```
import { Plugin, BurnerPluginContext } from '@burner-wallet/types'

const GAME_ADDRESS = '0x0123456789012345678901234567890123456789';

export default class GamePlugin implements Plugin {
  private pluginContext?: BurnerPluginContext;
  import gameAbi from './gameAbi.json';

  initializePlugin(pluginContext: BurnerPluginContext) {
    this.pluginContext = pluginContext;
  }

  async getBlockNumber() {
    const web3 = this.pluginContext!.getWeb3('1');
    return await web3.eth.getBlockNumber();
  }

  getContract() {
    const web3 = this.pluginContext!.getWeb3('1');
    const contract = new web3.eth.Contract(gameAbi, GAME_ADDRESS);
    return contract;
  }

  async getScore(userAddress) {
    const contract = this.getContract();
    const score = await contract.methods.getScore(userAddress).call();
    return score;
  }
}
```

4.2.6 addAddressToNameResolver

```
type AddressToNameResolver = (address: string) => Promise<string | null>;

pluginContext.addAddressToNameResolver(callback: AddressToNameResolver);
```

This API allows plugins to provide human-readable names for addresses displayed in the wallet UI. For example, the ENS plugin uses this to replace addresses with ENS names.

Paramaters

- `callback`: A function that can resolve addresses to human readable names. Callbacks are

passed an Ethereum address as a paramater, and should return a string or `null` if the address can not be resolved.

Example

```
import { BurnerPluginContext, Plugin, Account } from '@burner-wallet/types';

export default class ENSPlugin implements Plugin {

  initializePlugin(pluginContext: BurnerPluginContext) {
    pluginContext.addAddressToNameResolver(async (address: string) => {
      const name = await ens.reverseLookup(address);
      return name;
    });
  }
}
```

4.2.7 onAccountSearch

```
type AccountSearchFn = (query: string) => Promise<Account[]>;

pluginContext.onAccountSearch(callback: AccountSearchFn)
```

This API allows plugins to suggest accounts to user when they are typing in the "address" field for a new transaction. For example, the ENS Plugin uses this API to resolve ENS names, while the Recent Accounts Plugin uses this API to suggest accounts that the user has recently interacted with.

Paramaters

- `callback`: A function that will receive a search query as a paramater, and should return an array of "Account" objects (or an empty array). "Accounts" are objects that contain an "address" and "name" property.

Example

```
import { BurnerPluginContext, Plugin } from '@burner-wallet/types';

export default class ENSPlugin implements Plugin {
```

(continues on next page)

(continued from previous page)

```
  initializePlugin(pluginContext: BurnerPluginContext) {
    pluginContext.onAccountSearch(async (search: string) => {
      if (search.length < 3) {
        return [];
      }
      const address = await ens.getAddress(search);
      return address ? [{ address: address, name: search }] : [];
    });
  }
}
```

4.2.8 onQRScanned

```
type QRScannedFn = (qr: string, context: { actions: Actions }) => boolean | undefined;

pluginContext.onQRScanned(callback: QRScannedFn)
```

Provide a function to be called when the user scans a QR code using the default QR code scanner. The function is passed the text of the QR code and the "actions" object (see below).

For example, the ERC681 plugin uses this API to handle QR codes that contain the ERC681 URI format (`ethereum:0xf01acd...`).

Note: URLs of the same domain as the wallet are automatically handled. For example, if a wallet is hosted at `mywallet.com` and the user scans a QR code for `https://mywallet.com/mypage`, then the wallet will automatically route to `/mypage`.

Paramaters

- **`callback`: A function that parses the scanned QR code string and can chose to take action. This function must return `t`**
  - `qr`: The string value of the scanned QR code
  - `context`: This object currently only contains a single paramater, `actions`. However, more values may be added in the future.

Example

```
import { BurnerPluginContext, Plugin } from '@burner-wallet/types';

export default class ERC681Plugin implements Plugin {
  initializePlugin(pluginContext: BurnerPluginContext) {
    pluginContext.onQRScanned((qr: string, ctx: any) => {
      if (qr.indexOf('ethereum:') === 0) {
        const parsed = parse(qr);

        if (parsed === null) {
          return false;
        }

        ctx.actions.send({
          to: parsed.recipient,
```

(continues on next page)

(continued from previous page)

```
          value: parsed.value,
          asset: parsed.asset,
        });

        return true;
      }
      return false;
    });
  }
}
```

4.2.9 onSent

```
type TXSentFn = (data: SendData) => string | void | null;

pluginContext.onSent(callback: TXSentFn);
```

Provide a function to be called when the user sends an asset through the normal send mechanism. Callback will receive an object with the asset, sender and recipient address, amount, message, Web3 receipt, transaction hash, and an ID if specified in the send function.

Typically, a user will be redirected to the Receipt page after a transaction has been sent. However, plugins can override this behavior by returning a path string from the onSent callback.

Example

```
import { BurnerPluginContext, Plugin, SendData } from '@burner-wallet/types';
import OrderCompletePage from './OrderCompletePage';

export default class ShoppingPlugin implements Plugin {
  initializePlugin(pluginContext: BurnerPluginContext) {
    pluginContext.addPage('/order-complete/:id', OrderCompletePage);

    pluginContext.onSent((tx: SendData) => {
      if (tx.id.indexOf('order:') === 0) {
        return `/order-complete/${tx.id.substr(6)}`;
      }
    });
  }
}
```

4.2.10 sendPluginMessage

```
actions.sendPluginMessage(topic: string, ...message: any[]): any
```

Send cross-plugin messages

Paramaters

- `topic`: Topic ID that other plugins are listening for

- All other arguments will be included

Example

```
import { BurnerPluginContext, Plugin, SendData } from '@burner-wallet/types';

export class NamePlugin implements Plugin {
  private pluginContext: BurnerPluginContext;

  initializePlugin(pluginContext: BurnerPluginContext) {
    this.pluginContext = pluginContext;
  }

  changeName(newName: string) {
    this.pluginContext!.sendPluginMessage('name-changed', newName);
  }
}

export class OtherPlugin implements Plugin {
  private pluginContext: BurnerPluginContext;

  initializePlugin(pluginContext: BurnerPluginContext) {
    pluginContext.onPluginMessage('name-changed', (newName) => console.log('new name',
↪ newName));
  }
}
```

4.2.11 onPluginMessage

```
type PluginMessageListener = (...message: any[]) => any;

actions.onPluginMessage(topic: string, listener: PluginMessageListener)
```

Paramaters

- `topic`: Topic ID to listen for
- `listener`: A callback that will be passed all arguments from the message sender

Example

See example for sendPluginMessage

4.3 Plugin Component Props

Pages (added with `pluginContext.addPage`) and elements (added with `pluginContext.addElement`) will receive the following props.

4.3.1 plugin

Pages and Elements are provided with the instance of the Plugin that added them. This allows React components to access values from the plugin constructor, Web3 instances, and more.

Example

```
import { BurnerPluginContext, Plugin } from '@burner-wallet/types';

export default class MyPlugin implements Plugin {
  public id: string;
  private pluginContext?: BurnerPluginContext;

  constructor(id: string) {
    this.id = id;
  }

  initializePlugin(pluginContext: BurnerPluginContext) {
    this.pluginContext = pluginContext;
    pluginContext.addPage('/myPage', MyPage);
  }

  async send(account: string) {
    const web3 = this.pluginContext!.getWeb3('100');
    const contract = new web3.eth.Contract(abi, address);
    await contract.methods.myMethod().send({ from: account });
  }
}

const MyPage: React.FC<> = ({ plugin, defaultAccount }) => {
  const _plugin = plugin as MyPlugin;
  return (
    <div>
      <h1>ID: {_plugin.id}</h1>
      <button onClick={() => _plugin.send()}>Send Tx</button>
    </div>
  );
};
```

4.3.2 assets

`Array[]`

An array of Asset objects.

4.3.3 defaultAccount

`string`

The Ethereum address of the default account (equivelent to `accounts[0]`).

4.3.4 accounts

`string[]`

An array of addresses representing all available accounts.

4.3.5 actions

Object containing a number of functions. See Actions section.

4.3.6 BurnerComponents

An object containing a number of React components that can be used. See the Burner Components section.

4.3.7 React Router props

Page components (added with `addPage`) will also receive the match, location and history props from React Router.

Typescript users may define expected paramaters by passing a type argument to `PluginPageContext`.

Example

```
import { PluginPageContext } from '@burner-wallet/types';

interface MatchParams {
  level: string;
}

const Game: React.FC<PluginPageContext<MatchParams>> = ({ match }) => {
  return (
    <div>Welcome to level {match.params.level}</div>
  );
};
```

4.4 Actions

4.4.1 callSigner

```
actions.callSigner(action: string, target: string, ...props: any[]): string
```

Calls a method defined by the signer.

Paramaters

- `action`: The method to call. LocalSigner supports the methods "readKey", "writeKey", "burn", while InjectedSigner supports "enable".
- `target`: Either the address to call an action on (`0x9f31ca...`) or the ID of a signer (`local`, `injected`).
- `props`: Additional arguments, dependent on the actions.

Example

```
const PrivateKeyChanger = ({ actions, defaultAccount }) => {
  const [newKey, setNewKey] = useState('');
  const canChangeKey = actions.canCallSigner('writeKey', defaultAccount);

  return (
   <div>
      {canChangeKey ? (
         <div>
           <input value={newKey} onChange={e => setNewKey(e.target.value)} />
           <button onClick={() => actions.callSigner('writeKey', defaultAccount,␣
↪newKey)}>
             Change Key
           </button>
         <div>
      ) : "Can not update private key"}
   </div>
  );
};
```

4.4.2 canCallSigner

```
actions.canCallSigner(action: string, target: string, ...props: any[]): boolean
```

Paramaters

- `action`: The method to call. LocalSigner supports the methods "readKey", "writeKey", "burn", while InjectedSigner supports "enable".
- `target`: Either the address to call an action on (`0x9f31ca...`) or the ID of a signer (`local`, `injected`).
- `props`: Additional arguments, dependent on the actions.

Example

See example for `callSigner`

4.4.3 openDefaultQRScanner

```
actions.openDefaultQRScanner(): Promise
```

Open the full-screen QR code scanner. If a QR code is scanned, it will be handled with the default logic.

The default logic is as follows:

1. Plugins may handle QR codes by returning true from their `onQRScanned` callback.

2. If an address was scanned, the user will be redirected to the send page 2. If a private key was scanned, it will be handled with `safeSetPK` 2. URLs that contain the same domain as the wallet will be automatically routed

4.4.4 scanQRCode

```
actions.scanQRCode: () => Promise<string>
```

Opens the full-screen QR code scanner. Unlike `openDefaultQRScanner`, there is no default logic for handling the scanned QR code. The promise will resolve once a QR code is scanned, or will reject if the user cancels.

4.4.5 safeSetPK

```
actions.safeSetPK(newPK: string)
```

Attempts to update the user's private key, without losing any funds.

1. If there is no balance in the existing account, the new private key will be automatically updated
2. If the current account has funds, the user will be prompted with the following options:
    - Move all assets from the existing account to the new account
    - Move all assets from the new account to the existing account
    - Discard funds in the existing account and switch to the new account
    - Cancel, maintaining the current account

4.4.6 send

```
actions.send(params: SendData)
```

Prompt the user to send an asset, redirecting them to the send confirmation page.

Paramaters

TODO

4.4.7 navigateTo

```
actions.navigateTo(location: string | number, [state?: any])
```

Navigate the app's internal router to the path described (react-router is used internally).

Alternatively, pass a number to navigate forward or backwards through the history (pass `-1` to go back).

Example

```
const MyPage = ({ actions }) => (
  <div>
    <Button onClick={() => actions.navigateTo('/game')}>Game</Button>
    <Button onClick={() => actions.navigateTo('/game', { level: 2 })}>Level 2</Button>
    <Button onClick={() => actions.navigateTo(-1)}>Back</Button>
  </div>
);
```

4.4.8 setLoading

```
actions.setLoading(status: string | null)
```

4.4.9 getHistoryEvents

```
actions.getHistoryEvents([options?: any]): HistoryEvent[]
```

4.4.10 onHistoryEvent

```
actions.onHistoryEvent(callback: HistoryEventCallback)
```

4.4.11 removeHistoryEventListener

```
actions.removeHistoryEventListener(callback: HistoryEventCallback)
```

4.5 Burner Components: UI Components

4.5.1 Page

Props

- `title`: (string) Page title to display at top of page
- `children`: (React node) Page content

Example

```
const MyPage = ({ BurnerCompents }) => {
  const { Page } = BurnerComponents;
  return (
    <Page title="My Page">
      Content
    </Page>
  );
};
```

4.5.2 AssetSelector

TODO

4.5.3 AmountInput

TODO

4.5.4 Button

TODO

4.5.5 QRCode

TODO

4.6 Burner Components: Data Providers

A number of non-visual components are available. Many of these components simplify the process of accessing blockchain data using render props.

4.6.1 AccountBalance

Props

- `asset`: (string or Asset)
- [`account`]: (string) Account to look up data from. Optional, will use the default account if omitted
- `render`: (callback)

Callback data

The callback will be called with `null` while data is loading or unavailable. Once loaded, the callback will be called with an object with the following properties:

- `balance`: (string) The account balance, in wei-equivelent units (the balance divided by 10^18)
- `displayBalance`: (string) The balance in decimal format
- `maximumSendableBalance`: (string) The maximum that can be sent. For native assets like ETH, this will be the total balance minus the gas fee for a simple transaction
- `displayMaximumSendableBalance`: (string) `maximumSendableBalance` in decimal format
- `usdBalance`: (string | `null`) If price data is available, it will be the balance multiplied by the balance

Example

TODO

4.6.2 AccountKeys

TODO

4.6.3 AddressName

Retrieves the human-readable name for an address, if available

Props

- `address`
- `render`

Callback data

The render method will be called with two arguments:

- `name` (string or `null`) The human readable name, if available
- `address` (string)

Example

TODO

4.6.4 History

Render a list of history events

Props

- `account`: (string) The Ethereum account to fetch history for
- `render`: (callback) Render function that will be called once for each history element

Callback data

Example

TODO

4.6.5 PluginButtons

Define a region where other plugins may insert elements

Props

- `position`: (string) The name of the position
- `Component`: (React Component) Optional, the component to render each button. If omitted, `Button` will be used
- Other props will be passed through to inserted buttons

4.6.6 PluginElements

Define a region where other plugins may insert elements

Props

- `position`: (string) The name of the position
- Other props will be passed through to inserted elements

4.6.7 TransactionDetails

Props

- `asset`: (string) Asset ID
- `txHash`: (string) Transaction hash
- `render`: (callback) Render function

Callback data

The render function will provide a single SendData object with the following properties;

- `asset`: (string);
- `value`: (string) The amount transfered, in wei-equivelent units
- `ether`: (string) The amount transfered, in ether-equivelent units (typically `value * (10 ** 18)`)
- `from`: (string) The transaction sender
- `to`: (string) The transaction recipient (note: if this is a token transfer, it will be the transfer recipient, not the transaction recipient)
- `message`?: (string or `null`) The transaction message or `null`
- `hash`?: (string) Transaction hash
- `timestamp`: (number) Unix timestamp of the transaction (block time)

Example

TODO

CHAPTER 5

Packages

5.1 Main Packages

5.1.1 @burner-wallet/modern-ui

5.1.2 @burner-wallet/ui-core

5.1.3 @burner-wallet/classic-ui

5.1.4 @burner-wallet/types

5.2 Core Packages

5.2.1 @burner-wallet/core

5.2.2 @burner-wallet/assets

5.3 Standard Plugins

5.3.1 @burner-wallet/ens-plugin

5.3.2 @burner-wallet/erc681-plugin

5.3.3 @burner-wallet/exchange

5.3.4 @burner-wallet/legacy-plugin

5.3.5 @burner-wallet/link-plugin

5.3.6 @burner-wallet/metamask-plugin

5.3.7 @burner-wallet/onboarding-plugin

5.3.8 @burner-wallet/recent-accounts-plugin

CHAPTER 6

Theming

The Burner Wallet (specifically the ModernUI package) has minimal support for themes, although we plan allow further customization in the future.

Theme values can be passed to the optional `theme` prop of ModernUI:

Logo

Passing the URL to an image will display that image as the logo in the wallet header.

Accent Color

The `accentColor` value will set the color for elements such as buttons.

The values `accentLight` and `accentDark` are automatically calculated based on `accentColor`, however you may override these values.