universal-login Documentation

Release 0.0.1

Marek Kirejczyk, Justyna Broniszewska

Apr 24, 2019

Contents:

Universal Login is a design pattern for storing funds and connecting to Ethereum applications, aiming to simplify new users on-boarding.

This documentation covers all components of the project including sdk, relayer, smart contracts and example.

CHAPTER 1

Disclaimer

Universal Login is a work in progress and is at the experimental stage. Expect breaking changes. The code is not stable and has not been audited and therefore should not be used in a production environment

1.1 Getting started

1.1.1 Overview

Technical concepts

Technically Universal Login utilizes four major concepts:

- **Personal multi-sig wallet** - a smart contract used to store personal funds. A user gets his wallet created in a barely noticeable manner. The user then gets engaged incrementally to add authorization factors and recovery options.
- **Meta-transactions** - that gives user ability to interact with the smart contract from multiple devices easily, without a need to store ether on each of those devices. Meta-transactions enable payments for execution with tokens.
- **ENS names** - naming your wallet with easy-to-remember human-readable name
- **Universal login** - wallet name can be used to log in to dapps, web, and native applications

Components

Universal Login has three components. All components are stored in one monorepo available here. Components are listed below:

- Contracts - smart contracts used by Universal Login, along with some helper functions
- Relayer - HTTP REST server that relays meta-transactions to Universal Login smart contracts

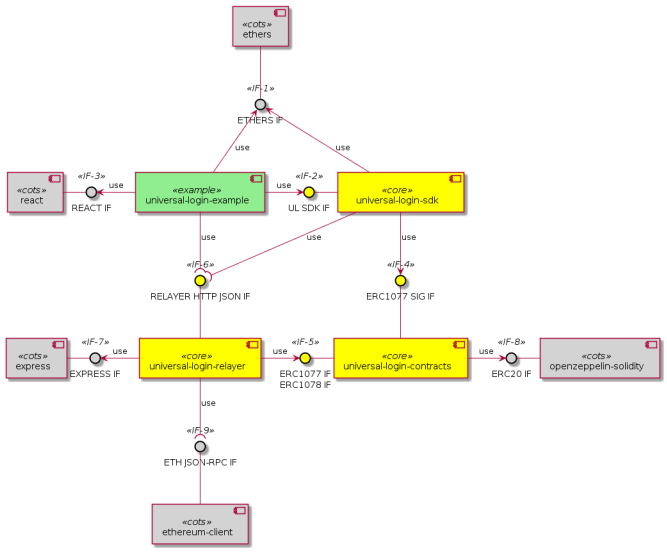
- SDK - javascript API, a thin communication layer that interacts with the Universal Login ecosystem, via both relayer and Ethereum node.

Additionally, there is one more package in the repository:

- Example - an example app, that demonstrates, used for testing and experimentation.

Dependencies

The diagram below shows dependencies between components.



The external interfaces present in the Universal Login system are identified by the lollipop use symbol:

<<IF-6>> RELAYER HTTP JSON IF this interface defines an off-chain remote API for ERC #1077 and #1078

<<IF-9>> ETH JSON-RPC IF this interface is the Ethereum JSON-RPC API for the on-chain execution

The internal interfaces defined within the Universal Login system are identified by the arrow use symbol. The main ones are:

<<IF-2>> UL SDK IF the JS applications using Universal Login shall be based on this library interface to conveniently attach to the Relayer subsystem and route their meta transactions

<<IF-4>> ERC1077 SIG IF this interface is a message hash and signature JS facility API for ERC #1077

<<IF-5>> ERC1077 IF / ERC1078 IF this interface is made up of ERC #1077 and #1078 smart contracts ABI

1.1.2 Quickstart

New project

Installation To add SDK to your project using npm type following:

```
npm i @universal-login/sdk
```

If you are using yarn than type:

```
yarn add @universal-login/sdk
```

Development environment

Summary Development environment helps quickly develop and test applications using universal login. The script that starts development environment can be run from `@universal-login/ops` project. The script does a bunch of helpful things:

- creates a mock blockchain (ganache)
- deploys mock ENS
- registers three testing ENS domains: `mylogin.eth`, `universal-id.eth`, `popularapp.eth`
- deploys example ERC20 Token that can be used to pay for transactions
- creates a database for a relayer
- starts local relayer

Prerequisites Before running the development environment, make sure you have **PostgreSQL** installed, up and running. You might want to check database configuration in file knexfile.js and make sure your database is configured correctly.

Installation To use development environment, you need to install `@universal-login/ops` as dev dependency to your project.

With npm:

```
npm install @universal-login/ops --save-dev
```

With yarn:

```
yarn add --dev @universal-login/ops -D
```

Adding a script The simplest way to use development environment is to add a script to `package.json` file:

```
...
"scripts": {
  ...
  "start:dev": "universal-login start:dev"
}
```

Running development environment To start development environment type in your console:

```
yarn start:dev
```

Which will start the development environment. The output should look somewhat like this:

```
Wallets:
  0x17ec8597ff92C3F44523bDc65BF0f1bE632917ff -␣
↪0x29f3edee0ad3abf8e2699402e0e28cd6492c9be7eaab00d732a791c33552f797
  0x63FC2aD3d021a4D7e64323529a55a9442C444dA0 -␣
↪0x5c8b9227cd5065c7e3f6b73826b8b42e198c4497f6688e3085d5ab3a6d520e74
  0xD1D84F0e28D6fedF03c73151f98dF95139700aa7 -␣
↪0x50c8b3fc81e908501c8cd0a60911633acaca1a567d1be8e769c5ae7007b34b23
  0xd59ca627Af68D29C547B91066297a7c469a7bF72 -␣
↪0x706618637b8ca922f6290ce1ecd4c31247e9ab75cf0530a0ac95c0332173d7c5
  0xc2FCc7Bcf743153C58Efd44E6E723E9819E9A10A -␣
↪0xe217d63f0be63e8d127815c7f26531e649204ab9486b134ec1a0ae9b0fee6bcf
  0x2ad611e02E4F7063F515C8f190E5728719937205 -␣
↪0x8101cca52cd2a6d8def002ffa2c606f05e109716522ca2440b2cc84e4d49700b
```

(continues on next page)

(continued from previous page)

```
  0x5e8b3a7e6241CeE1f375924985F9c08706f41d34 -␣
↪0x837fd366bc7402b65311de9940de0d6c0ba3125629b8509aebbfb057ebeaaa25
  0xFC6F167a5AB77Fe53C4308a44d6893e8F2E54131 -␣
↪0xba35c32f7cbda6a6cedeea5f73ff928d1e41557eddfd457123f6426a43adb1e4
  0xDe41151d0762CB537921c99208c916f1cC7dA04D -␣
↪0x71f7818582e55456cb575eea3d0ce408dcf4cbbc3d845e86a7936d2f48f74035
  0x121199e18C70ac458958E8eB0BC97c0Ba0A36979 -␣
↪0x03c909455dcef4e1e981a21ffb14c1c51214906ce19e8e7541921b758221b5ae

Node url (ganache): http://localhost:18545...
      ENS address: 0x67AC97e1088C332cBc7a7a9bAd8a4f7196D5a1Ce
Registered domains: mylogin.eth, universal-id.eth, popularapp.eth
    Token address: 0x0E2365e86A50377c567E1a62CA473656f0029F1e
      Relayer url: http://localhost:3311
```

1.1.3 Using SDK

Creating a wallet contract

To start using SDK you will need to create SDK instance and deploy a wallet contract. Below is a snippet doing precisely that for the development environment.

```
import UniversalLoginSDK from '@universal-login/sdk';

const universalLoginSDK = new UniversalLoginSDK('http://localhost:3311', 'http://
↪localhost:18545');
const [privateKey, contractAddress] = await sdk.create('myname.mylogin.eth');
```

The first argument of `UniversalLoginSDK` constructor is relayer address, second is Ethereum node address.

Sending transaction

Once you have contract wallet deployed you can execute a transaction:

```
const message = {
  from: '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
  to: '0xbA03ea3517ddcD75e38a65EDEB4dD4ae17D52A1A',
  data: '0x0',
  value: '500000000000000000',
  gasToken: '0x9f2990f93694B496F5EAc5822a45f9c642aaDB73',
  gasPrice: 1000000000,
  gasLimit: 1000000
};

await sdk.execute(message, privateKey);
```

Note: `from` field in this case is contract address.

Most fields of a message are analogous to normal Ethereum transaction, except for `gasToken`, which allows specifying token in which transaction cost will be refunded.

The token need to be supported by relayer. Wallet needs to have enough token balance to refund transaction.

A detailed explanation of each method can be found in subsections of *SDK documentation*: *creating SDK*, *creating wallet contract* and *execute*.

1.1.4 Connecting SDK to testnet

To connect SDK to the Rinkeby testnet and the test relayer:

```
import UniversalLoginSDK from '@universal-login/sdk';
import ethers from 'ethers';


const relayerUrl = 'https://relayer.universallogin.io';
const jsonRpcUrl = 'https://rinkeby.infura.io';

const universalLoginSDK = new UniversalLoginSDK(relayerUrl, jsonRpcUrl);
```

You can find example usage of SDK *here*

1.1.5 What's next?

Go to:

- *SDK documentation* - if you would like to build an application using Universal Login
- *Relayer documentation* - if you would like to set up your own relayer
- *Example documentation* - if you would like to play with the example application

1.2 SDK

SDK is a JS library, that helps to communicate with relayer. SDK makes easy to manage contract, by creating basic contract-calling messages. It uses private key to sign that messages and sends it to relayer, which propagates it to network.

1.2.1 Creating SDK

new UniversalLoginSDK(relayerURL, providerURL, messageOptions)

Parameters:

- **relayerURL** : string - URL address of relayer
- **providerURL** : string - JSON-RPC URL of an Ethereum node
- **messageOptions** (optional) : object - specific message options as `gasPrice` or `gasLimit`

Returns: UniversalLoginSDK instance

Example:

```
import UniversalLoginSDK from '@universal-login/sdk';

const messageOptions = {
  gasPrice: 1500000000,
  gasLimit: 2000000,
  operationType: OPERATION_CALL
};
const universalLoginSDK = new UniversalLoginSDK(
  'http://myrelayer.ethworks.io',
```

(continues on next page)

(continued from previous page)

```
  'http://localhost:18545',
  messageOptions
);
```

1.2.2 Creating wallet contract

create

sdk.create(ensName)

creates new wallet contract.

Parameters:

- **ensName** : string - choosen ENS name with existing domain (ENS domain supported by relayer)

Returns: *promise*, that resolves to a pair `[privateKey, contractAddress]`, where:

- *privateKey* - private key assigend to the wallet contract for signing future transactions
- *contract address* - address of newly deployed wallet contract, with choosen ENS name assigned

Example:

```
const [privateKey, contractAddress] = await sdk.create('myname.example-
↪domain.eth');
```

connect

sdk.connect(contractAddress)

requests of adding a new key to contract.

Parameters:

- **contractAddress** : string - address of contract to manage a connect

Returns: *promise*, that resolves to `privateKey`, where:

- *privateKey* - private key that is requested to add to manage contract

Example:

```
const privateKey = sdk.connect(
↪'0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA');
```

denyRequest

sdk.denyRequest(contractAddress, publicKey)

removes request of adding new key from pending authorisations.

Parameters:

- **contractAddress** : string - address of contract to remove request
- **publicKey** : string - address to remove from add requests

Returns: *promise*, that resolves to `publicKey`, where:

- *publicKey* - address removed from pending authorisations

Example:

```
const publicKey = await sdk.denyRequest(
↪'0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
↪'0xb19Ec9bdC6733Bf0c825FCB6E6Da95518DB80D13');
```

1.2.3 Transaction execution

execute

sdk.execute(message, privateKey)

executes any message.

Parameters:

- **message** : object - message that is sent to contract, includes:
  - from : string - address of contract that requests execution
  - to : string - beneficient of this execution
  - data : string - data of execution
  - value : string - value of transaction
  - gasToken : string - token address to refund
  - gasPrice : number - price of gas to refund
  - gasLimit : number - limit of gas to refund
- **privateKey** : string - a private key to be used to sign the transaction and has permission to execute message

Returns: *promise*, that resolves to the hash of the on-chain transaction

Example:

```
const message = {
  from: '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
  to: '0xbA03ea3517ddcD75e38a65EDEB4dD4ae17D52A1A',
  data: '0x0',
  value: '500000000000000000',
  gasToken: '0x9f2990f93694B496F5EAc5822a45f9c642aaDB73',
  gasPrice: 1000000000,
  gasLimit: 1000000
};

await sdk.execute(
  message,
  '0x5c8b9227cd5065c7e3f6b73826b8b42e198c4497f6688e3085d5ab3a6d520e74'
);
```

In this case contract `0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA` sends 0.5 eth to `0xbA03ea3517ddcD75e38a65EDEB4dD4ae17D52A1A`.

SdkSigner

```
// gasToken should be configured when creating SDK instance in order to use the signer
const signer = new SdkSigner(sdk, contractAddress, privateKey);

const token = new Contract(contractAddress, contractInterface, signer)
await contract.transfer(someOtherAddress, utils.parseEther('123'))
```

Note: This is an experimental feature, expect breaking changes.

1.2.4 Managing wallet contract

addKey

sdk.addKey(contractAddress, publicKey, privateKey, transactionDetails, keysPurpose)

adds key to manage wallet contract.

Parameters:

- **contractAddress** : string - address of contract that requests to add new key
- **publicKey** : string - public key to manage contract
- **privateKey** : string - private key that has permission to add new keys
- **transactionDetails** : object - refund options
- **keysPurpose** (optional) : number - key purpose: MANAGEMENT_KEY - `1`, ACTION_KEY - `2`, set to MANAGAMENT_KEY by default

Returns: *promise*, that resolves to the hash of the on-chain transaction

Example:

```
const transactionDetails = {
  gasToken: '0x850437540FE07d02045f88cAe122Bc66B1BdE957',
  gasPrice: 1000000,
  gasLimit: 150000
};
await sdk.addKey(
  '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
  '0x96E8B90685AFD981453803f1aE2f05f8Ebc3cfD0',
  '0x5c8b9227cd5065c7e3f6b73826b8b42e198c4497f6688e3085d5ab3a6d520e74',
  transactionDetails,
  ACTION_KEY
);
```

addKeys

sdk.addKeys(contractAddress, publicKeys, privateKey, transactionDetails, keysPurpose)

adds multiple keys to manage contract.

Parameters:

- **contractAddress** : string - address of contract that requests to add keys
- **publicKeys** : array of strings - public keys to add

- **privateKey** : string - private key that has permission to add new keys
- **transactionDetails** : object - refund options
- **keysPurpose** (optional) : number - key purpose: MANAGEMENT - `1`, ACTION - `2`, set to MANAGAMENT_KEY by default

Returns: *promise*, that resolves to the hash of the on-chain transaction

Example:

```
const publicKeys = [
  '0x96E8B90685AFD981453803f1aE2f05f8Ebc3cfD0',
  '0xb19Ec9bdC6733Bf0c825FCB6E6Da95518DB80D13'
];
const transactionDetails = {
  gasToken: '0x850437540FE07d02045f88cAe122Bc66B1BdE957',
  gasPrice: 1000000,
  gasLimit: 150000
};
await sdk.addKeys(
  '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
  publicKeys,
  '0x5c8b9227cd5065c7e3f6b73826b8b42e198c4497f6688e3085d5ab3a6d520e74',
  transactionDetails,
  ACTION_KEY
);
```

removeKey

sdk.removeKey(contractAddress, publicKey, privateKey, transactionDetails)

removes key from contract.

Parameters:

- **contractAddress** : string - address of contract, that we want remove key from
- **publicKey** : string - public key to remove
- **privateKey** : string - private key with permission of removing key
- **transactionDetails** : object - optional parameter, that includes details of transactions for example gasLimit or gasPrice

Returns: *promise*, that resolves to the hash of the on-chain transaction

Example

```
const transactionDetails = {
  gasToken: '0x9f2990f93694B496F5EAc5822a45f9c642aaDB73',
  gasPrice: 1000000,
  gasLimit: 150000
};
await sdk.removeKey(
  '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
  '0xbA03ea3517ddcD75e38a65EDEB4dD4ae17D52A1A',
  '0x5c8b9227cd5065c7e3f6b73826b8b42e198c4497f6688e3085d5ab3a6d520e74',
  transactionDetails
);
```

getWalletContractAddress(ensName)

gets wallet contract address by ENS name

Parameters:

- **ensName** : string - ENS name

Returns: *promise*, that resolves to `address` if ENS name is registered or `null` if ENS name is available

Example:

```
const contractAddress = await sdk.getWalletContractAddress('justyna.my-
↪super-domain.test');
```

walletContractExist(ensName)

checks if ENS name is registered.

Parameters:

- **ensName** : string - ENS name

Returns: *promise*, that resolves to `true` if ENS name is registered or `false` if ENS name is available

Example:

```
const walletContractExist = await sdk.walletContractExist('justyna.my-
↪super-domain.test');
```

1.2.5 Events

sdk.start()

Starts to listen relayer and blockchain events.

sdk.stop()

Stops to listen relayer and blockchain events.

Subscribe

sdk.subscribe(eventType, filter, callback)

subscribes an event.

Parameters:

- **eventType** : string - type of event, possible event types: `KeyAdded`, `KeyRemoved` and `AuthorisationsChanged`
- **filter** : object - filter for events, includes:
  - contractAddress : string - address of contract to observe
  - key (optional) : string - public key, using when subscribe to events with specific key (only for `KeyAdded` and `KeyRemoved`)
- **callback**

Returns: event listener

Example:

```
const filter = {
  contractAddress: '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
  key: '0xbA03ea3517ddcD75e38a65EDEB4dD4ae17D52A1A'
};
const subscription = sdk.subscribe(
  'KeyAdded',
  filter,
  (keyInfo) => {
    console.log(`${keyInfo.key} was added.`);
  }
);
```

Result

```
0xbA03ea3517ddcD75e38a65EDEB4dD4ae17D52A1A was added
```

Example:

```
const filter = {
  contractAddress: '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA'
};
const subscription = sdk.subscribe(
  'AuthorisationsChanged',
  filter,
  (authorisations) => {
    console.log(`${authorisations}`);
  }
);
```

Result

```
[{deviceInfo:
    {
      ipAddress: '89.67.68.130',
      browser: 'Safari',
      city: 'Warsaw'
    },
  id: 1,
  walletContractAddress: '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
  key: ''}]
```

Unsubscribe

subscription.remove()

removes subscription

Example:

```
const subscription = sdk.subscribe(
  'KeyAdded',
  filter,
  (keyInfo) => {
    subscription.remove();
  }
);
```

Example

```
import {Wallet} from 'ethers';

const privateKey = await sdk.connect(
↪'0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA');
const wallet = new Wallet(privateKey);
const filter = {
  contractAddress: '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
  key: wallet.address
};
const subscription = sdk.subscribe(
  'KeyAdded',
  filter,
  (keyInfo) => {
    this.myWallet = wallet;
    subscription.remove();
  }
);
```

1.3 Relayer

Relayer is a RESTful JSON API server written in node.js and express.js, that allows interacting with wallet contract using meta-transactions. Relayer gets signed message and propagates it to the network. It pays for transactions and gets the refund from contracts.

Below are the instructions how to run relayer.

If you would like to use your own domain, jump to the section *ENS registration*. To learn how to build custom relayer jump to *Custom relayer*.

1.3.1 Starting relayer

Prerequisites

To run relayer in development mode and to run tests you need to have Postgres installed and running. You also need to have *universal_login_relayer_development* database created.

You can do it in your favorite database UI, or from *psql*:

```
psql
> create database universal_login_relayer_development;
> \q
```

There are two ways to setup relayer *from command line* and *programmatically*.

From command line

To start relayer from command line, clone UniversalLoginSDK github repository and follow steps:

1. Setup environment

Create `.env` file in `/universal-login-relayer` directory and fill up .env file with parameters:

- **JSON_RPC_URL** : string - JSON-RPC URL of an Ethereum node

- **PORT** : number - relayer endpoint
- **PRIVATE_KEY** : string - private key of relayer wallet
- **ENS_ADDRESS** : string - address of ENS
- **ENS_DOMAIN** : string - name of domain

example .env file

```
JSON_RPC_URL='https://rinkeby.infura.io'
PORT=3311
PRIVATE_KEY='YOUR_PRIVATE_KEY'
ENS_ADDRESS='0xe7410170f87102DF0055eB195163A03B7F2Bff4A'
ENS_DOMAIN_1='poppularapp.test'
ENS_DOMAIN_2='my-id.test'
ENS_DOMAIN_3='my-super-domain.test'
```

2. Run relayer

Run following command from `universal-login-relayer` directory

```
yarn relayer:start
```

Programmatically

To run relayer from your application you will need to create a relayer instance. Relayer constructor documentation below.

new Relayer(config, provider)

Parameters:

- **config** : object - specific config parameters, includes:
  - **legacyENS** : boolean - ENS version deployed on network, for the Rinkeby testnet is `true`, for the Ropsten testnet is `false`
  - **jsonRpcUrl** : string - JSON-RPC URL of an Ethereum node
  - **port** : number - relayer endpoint
  - **privateKey** : string - private key of relayer wallet
  - **ensAddress** : string - address of ENS
  - **ensRegistrars** : array of strings - possible domains
- **provider** : object (optional) - instance of provider of an Ethereum node

Returns: Relayer instance

Example

```
import Relayer from 'unviersal-login-relayer';

const config = {
  legacyENS: true,
  jsonRpcUrl: 'https://rinkeby.infura.io',
  port: 3311,
  privateKey: 'YOUR_PRIVATE_KEY',
  chainSpec: {
```

(continues on next page)

(continued from previous page)

```
    ensAddress: '0xe7410170f87102DF0055eB195163A03B7F2Bff4A',
    chainId: 0
  },
  ensRegistrars: [
    'poppularapp.test',
    'my-id.test',
    'my-super-domain.test'
  ]
};

const relayer = new Relayer(config);
relayer.start();
```

Example: connecting to testnet

config.js file

```
const config = {
  legacyENS: true,
  jsonRpcUrl: process.env.JSON_RPC_URL,
  port: process.env.PORT,
  privateKey: process.env.PRIVATE_KEY,
  chainSpec: {
    ensAddress: process.env.ENS_ADDRESS,
    chainId: 0
  },
  ensRegistrars: [
    process.env.ENS_DOMAIN_1,
    process.env.ENS_DOMAIN_2,
    process.env.ENS_DOMAIN_3
  ]
}
```

.env file

```
JSON_RPC_URL='https://rinkeby.infura.io'
PORT=3311
PRIVATE_KEY='YOUR_PRIVATE_KEY'
ENS_ADDRESS='0xe7410170f87102DF0055eB195163A03B7F2Bff4A'
ENS_DOMAIN_1='poppularapp.test'
ENS_DOMAIN_2='my-id.test'
ENS_DOMAIN_3='my-super-domain.test'
```

1.3.2 Custom relayer

You can subclass relayer to create custom behaviot, e.g. a relayer that grants ether or tokens to a newly created wallet contract.

After every operations on contract, there is emitted an event. You can add listeners to this events and transfer funds for every operation.

Possible events:

- **created** - emitted on new contract creation

- **added** - emitted on add new key to manage contract
- **keysAdded** - emitted on add multiple keys to manage contract

Note: Events are emitted right after send transaction, not when transaction is mined. You need to wait until it is mined (e.g. use waitToBeMined function).

Event returns transaction detalis as transaction hash or gasPrice.

this.hooks.addListener(eventType, callback)

subscribes an event.

Parameters:

- **eventType** : string - type of event, possible event types: `created`, `added` and `keysAdded`
- **callback**

Returns: event listener

In this example, we create ether granting relayer, that gives tokens to wallet contract for creation, adding key and adding keys.

```
import ethers from 'ethers';
import {waitToBeMined} from '@universal-login/commons';

class EtherGrantingRelayer extends Relayer {
  constructor(config, provider = '') {
    super(config, provider);
    this.addHooks();
  }

  addHooks() {
    this.hooks.addListener('created', async (transaction) => {
      const receipt = await waitToBeMined(this.provider, transaction.hash);
      if (receipt.status) {
        this.wallet.sendTransaction({
          to: receipt.contractAddress,
          value: ethers.utils.parseEther('0.01')
        });
      }
    });

    this.addKeySubscription = this.hooks.addListener('added', async␣
↪(transaction) => {
      const receipt = await waitToBeMined(this.provider, transaction.hash);
      if (receipt.status) {
        this.wallet.sendTransaction({
          to: receipt.contractAddress,
          value: ethers.utils.parseEther('0.001')
        });
      }
    });

    this.addKeysSubscription = this.hooks.addListener('keysAdded', async␣
↪(transaction) => {
      const recepit = await waitToBeMined(this.provider, transaction.hash);
      if (recepit.status) {
        this.wallet.sendTransaction({
          to: receipt.contractAddress,
```

(continues on next page)

(continued from previous page)

```
          value: ethers.utils.parseEther('0.005')
        });
      }
    });
  }
}
```

Note: Relayer will issue a new transaction after contract is deployed. Therefore ether/tokens will not appear instantly, but after a while.

You can also take a look at TokenGrantingRelayer used in dev environment.

1.4 Example App

Example App is a dapp using Universal Login SDK written using React, written for demonstration and testing purposes. With the example, you can create a wallet contract, execute a simple operation on `Clicker` contract and manage keys.

Prerequisites Before running the example, make sure you have PostgreSQL installed, up and running. You might want to check database configuration in file knexfile.js and make sure your database is configured correctly.

Installing dependencies To install dependencies and build projects run following commands from the main project directory:

```
yarn
```

1.4.1 Running example on dev environment

Running example To run example in development mode use following commands:

```
cd universal-login-example
yarn start:dev [hostAddress]
```

Parameters:

- *hostAddress* (optional) - is host address where the Universal Login relayer will be accessible via HTTP (default is localhost, only works in the local browser).

This command will start a local development environment, including:

- start mock development blockchain (ganache)
- deploy mock ens system and testing domains (`mylogin.eth`, `universal-id.eth`, `popularapp.eth`)
- start pre-configured relayer
- deploy test token that can be used for transaction refund
- deploy `Clicker` contract used by example application
- setup HTTP server for example app at `localhost:3000`

The output of the command should look like this:

```
  Wallets:
    0x17ec8597ff92C3F44523bDc65BF0f1bE632917ff -␣
↪0x29f3edee0ad3abf8e2699402e0e28cd6492c9be7eaab00d732a791c33552f797
    0x63FC2aD3d021a4D7e64323529a55a9442C444dA0 -␣
↪0x5c8b9227cd5065c7e3f6b73826b8b42e198c4497f6688e3085d5ab3a6d520e74
    0xD1D84F0e28D6fedF03c73151f98dF95139700aa7 -␣
↪0x50c8b3fc81e908501c8cd0a60911633acaca1a567d1be8e769c5ae7007b34b23
    0xd59ca627Af68D29C547B91066297a7c469a7bF72 -␣
↪0x706618637b8ca922f6290ce1ecd4c31247e9ab75cf0530a0ac95c0332173d7c5
    0xc2FCc7Bcf743153C58Efd44E6E723E9819E9A10A -␣
↪0xe217d63f0be63e8d127815c7f26531e649204ab9486b134ec1a0ae9b0fee6bcf
    0x2ad611e02E4F7063F515C8f190E5728719937205 -␣
↪0x8101cca52cd2a6d8def002ffa2c606f05e109716522ca2440b2cc84e4d49700b
    0x5e8b3a7e6241CeE1f375924985F9c08706f41d34 -␣
↪0x837fd366bc7402b65311de9940de0d6c0ba3125629b8509aebbfb057ebeaaa25
    0xFC6F167a5AB77Fe53C4308a44d6893e8F2E54131 -␣
↪0xba35c32f7cbda6a6cedeea5f73ff928d1e41557eddfd457123f6426a43adb1e4
    0xDe41151d0762CB537921c99208c916f1cC7dA04D -␣
↪0x71f7818582e55456cb575eea3d0ce408dcf4cbbc3d845e86a7936d2f48f74035
    0x121199e18C70ac458958E8eB0BC97c0Ba0A36979 -␣
↪0x03c909455dcef4e1e981a21ffb14c1c51214906ce19e8e7541921b758221b5ae

  Node url (ganache): http://localhost:18545...
        ENS address: 0x67AC97e1088C332cBc7a7a9bAd8a4f7196D5a1Ce
  Registered domains: mylogin.eth, universal-id.eth, popularapp.eth
      Token address: 0x0E2365e86A50377c567E1a62CA473656f0029F1e
        Relayer url: http://localhost:3311
    Clicker address: 0xD3C4A8F56538e07Be4522D20A6410c2c4e4B26a6

web: $ parcel --no-cache ./src/index.html

web: Server running at http://localhost:1234
```

You can now go to `http://localhost:1234` to play with the application.

1.4.2 Running example on testnet environment

To configure the example application, you need to set a couple on environmental variables. A simple way to do that you need to create `.env` file.

To use existing testnet relayer create `.env` file in `universal-login-example` directory and fill it up with following parameters:

```
JSON_RPC_URL='https://rinkeby.infura.io'
RELAYER_URL='https://relayer.universallogin.io'
ENS_ADDRESS='0xe7410170f87102DF0055eB195163A03B7F2Bff4A'
ENS_DOMAIN_1='poppularapp.test'
ENS_DOMAIN_2='my-id.test'
TOKEN_CONTRACT_ADDRESS='0x5F81E2afde8297F90b3F9179F8F3eA172f3155A8'
CLICKER_CONTRACT_ADDRESS='0x01Ed4566E61E3a964059c692e511f441F9B3B8B2'
```

To run the application in production mode, type in the console in `universal-login-example` directory following command:

```
yarn start
```

You can use the configuration above in your own applications.

1.5 Deployment

Prerequisites Install universal-login toolkit:

```
yarn global add @universal-login/ops
```

1.5.1 Test token

To deploy test token use deploy token `universal-login deploy:token --nodeUrl [url] --privateKey [privateKey]`

Example:

```
universal-login deploy:token --nodeUrl http://localhost:18545 --privateKey␣
↪0x29f3edee0ad3abf8e2699402e0e28cd6492c9be7eaab00d732a791c33552f797
```

1.5.2 Sending funds

To send funds to an address `universal-login send [to] [amount] [currency] --nodeUrl [url] --privateKey [privateKey]`

Parameters:

- **to** - an address to send funds
- **amount** - amount to send to the address
- **currency** - currency of transfer
- **nodeUrl** (optional) - JSON-RPC URL of an Ethereum node, set to `http://localhost:18545` by default
- **privateKey** (optional) - private key of wallet with additional balance, set to `DEV_DEFAULT_PRIVATE_KEY` by default and has enoguh ethers

Example:

```
universal-login send 0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA 4 ETH
```

1.5.3 ENS registration

To use Universal Login with your own ENS domain, you will need to register it, connect to the resolver and deploy own registrar. There is a script for that.

Note: script currently works only for `.test` domains. Tested on the Rinkeby and the Ropsten test networks.

You can register domain on two ways: from command line and programmatically. To use registered domain in your relayer, type its name in relayer config.

From command line

First, prepare `.env` file in universal-login-ops directory.

Parameters:

- **JSON_RPC_URL** : string - JSON-RPC URL of an Ethereum node
- **PRIVATE_KEY** : string - private key to execute registrations. *Note:* You need to have ether on it to pay for contracts deployment.
- **ENS_ADDRESS** : string - address of ENS
- **PUBLIC_RESOLVER_ADDRESS** : string - address of public resolver. For the Ropsten test network working public resolver address is `0x4C641FB9BAd9b60EF180c31F56051cE826d21A9A` and for the Rinkeby test network public resolver address is `0x5d20cf83cb385e06d2f2a892f9322cd4933eacdc`.

Example `.env` file:

```
JSON_RPC_URL='https://ropsten.infura.io'
PRIVATE_KEY='YOUR_PRIVATE_KEY'
ENS_ADDRESS='0x112234455c3a32fd11230c42e7bccd4a84e02010'
PUBLIC_RESOLVER_ADDRESS='0x4C641FB9BAd9b60EF180c31F56051cE826d21A9A'
```

To register ENS domain, in universal-login-ops directory type in the console:

```
yarn register:domain my-domain tld
```

Parameters:

- **my-domain** - domain to register
- **tld** - top level domain, for example: `eth` or on testnets: `test`

Example:

```
yarn register:domain cool-domain test
```

Result:

```
Registering cool-domain.test...
Registrar address for test: 0x21397c1A1F4aCD9132fE36Df011610564b87E24b
Registered cool-domain.test with owner: 0xf4C1A210B6436eEe17fDEe880206E9d3Ab178c18
Resolver for cool-domain.test set to 0x4C641FB9BAd9b60EF180c31F56051cE826d21A9A␣
↪(public resolver)
New registrar deployed: 0xf1Af1CCEEC4464212Fc7b790c205ca3b8E74ba67
cool-domain.test owner set to: 0xf1Af1CCEEC4464212Fc7b790c205ca3b8E74ba67␣
↪(registrar)
```

Programmatically

To register own ENS domain programmatically, you should use DomainRegistrar.

new DomainRegistrar(config) creates DomainRegistrar.

Parameters:

- **config** : object - specific config parameters, includes:
  - **jsonRpcUrl** : string - JSON-RPC URL of an Ethereum node
  - **privateKey** : string - private key to execute registrations
  - **ensAddress** : string - address of ENS
  - **publicResolverAddress** : string - address of public resolver

Returns: DomainRegistrar instance

Example:

```
const ensRegistrationConfig = {
  jsonRpcUrl: 'https://ropsten.infura.io',
  privateKey: 'YOUR_PRIVATE_KEY',
  chainSpec: {
    ensAddress: '0x112234455c3a32fd11230c42e7bccd4a84e02010',
    publicResolverAddress: '0x4C641FB9BAd9b60EF180c31F56051cE826d21A9A',
    chainId: 0
  }
}
const registrar = new DomainRegistrar(ensRegistrationConfig);
```

registrar.registerAndSave(domain, tld) registers new domain and saves to new file all informations about newly registered domain (registrar address or resolver address)

Parameters:

- **domain** : string - domain to register
- **tld** : string - top level domain, for example: `eth` or on testnets: `test`

Example:

```
registrar.registerAndSave('new-domain', 'test');
```

Result: file named `extra-domain.test_info` that includes:

```
DOMAIN='extra-domain.test'
PUBLIC_RESOLVER_ADDRESS='0x4C641FB9BAd9b60EF180c31F56051cE826d21A9A'
REGISTRAR_ADDRESS='0xEe0b357352C7Ba455EFD0E20d192bC44F1Bf8d22'
```

1.6 Tutorials

1.6.1 Connecting to existing app on testnet

Create wallet contract

Create your own wallet contract using Universal Login Example App and get your contract address.

Create UniversalLoginSDK

In your project, create the UniversalLoginSDK

```
import UniversalLoginSDK from '@universal-login/sdk';
import ethers from 'ethers';


const relayerUrl = 'https://relayer.universallogin.io';
const jsonRpcUrl = 'https://rinkeby.infura.io';

const universalLoginSDK = new UniversalLoginSDK(relayerUrl, jsonRpcUrl);
```

Start listen events

Then make UniversalLoginSDK start listening relayer and blockchain events

```
sdk.start();
```

Request connection

Now, you can request connection to created wallet contract

```
const privateKey = await sdk.connect('YOUR_CONTRACT_ADDRESS');
```

Subscribe KeyAdded

Subscribe `KeyAdded` event with your new key filter

```
const key = new ethers.Wallet(privateKey).address;
const filter =
  {
    contractAddress: 'YOUR_CONTRACT_ADDRESS',
    key
  };

const subscription = sdk.subscribe(
  'KeyAdded',
  filter,
  (keyInfo) =>
    {
      console.log(`${keyInfo.key} now has permission to manage wallet contract`);
    });
```

Accept connection request

Accept connection request in Universal Login Example App. After that your newly created key has permission to manage your wallet contract.

Stop listen events

Remember about stop listening relayer and blockchain events

```
sdk.stop();
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search