universal-login Documentation

Release 0.0.1

Marek Kirejczyk, Justyna Broniszewska

Aug 24, 2019

Contents:

1 Disclaimer **3**
- 1.1 Getting started 3
- 1.2 Overview 4
- 1.3 Tutorial 14
- 1.4 SDK 20
- 1.5 Relayer 28

2 Indices and tables **31**

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 and smart contracts.

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

Universal Login helps you build user-friendly Ethereum based applications.

To quickly start building an application go to one of *tutorial* sections below:

- *Quickstart*
- *Using SDK*
- *Connecting to existing app on testnet*
- *Helpers*
- *ENS registration*

To learn Universal Login concepts and architecture go to one of *overview* sections:

- *Introduction*
- *Main concepts*
- *Development environment*

To get API reference go to one of the following sections:

- *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

1.2 Overview

1.2.1 Introduction

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** - a wallet name can be used to log in to dapps, web, and native applications

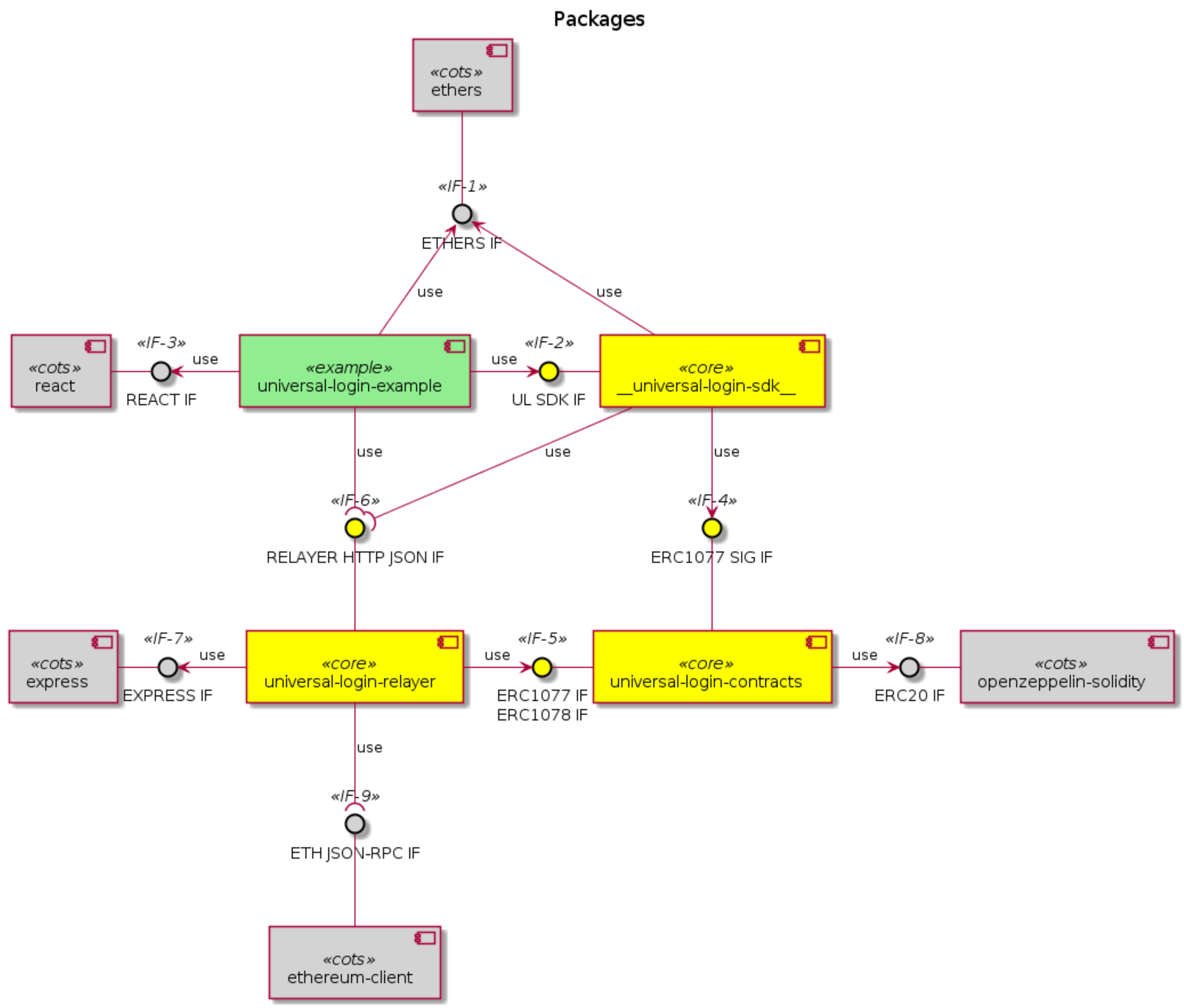
Components

Universal Login has four 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.
- React - typescript library, that contains Universal Login main components to use in react applications.

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

1.2.2 Main concepts

Deployment

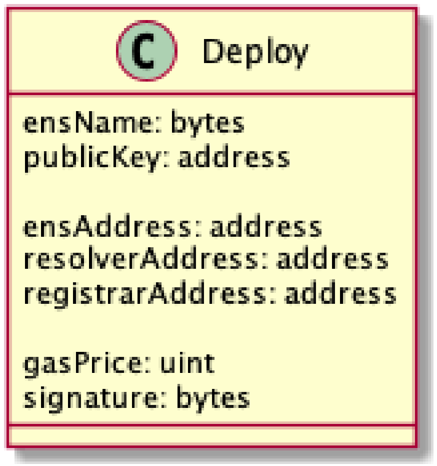
Deployment is designed in a way that makes the user pay for himself. To do that, we use counterfactual deployment (`create2` function to deploy contract).

The process looks as follows:

- **computing contract address** - The SDK computes a deterministic contract address. A contract address is unique and is obtained from the user's public key and a factory contract address connected to a particular relayer.
- **waiting for balance** - The user sends funds to this address. He can transfer ether on his own or use an on-ramp provider. The SDK waits for the future contract address balance to change. If the SDK discovers that required funds appear at this address, it sends a deploy request to the relayer.
- **deploy** - The relayer deploys the contract and gets a refund from it immediately.
- **refund** - During deployment the contract will refund the cost of the transaction to the relayer address.

Deployment in-depth

An SDK creates deployment. The deployment contains the following parameters:



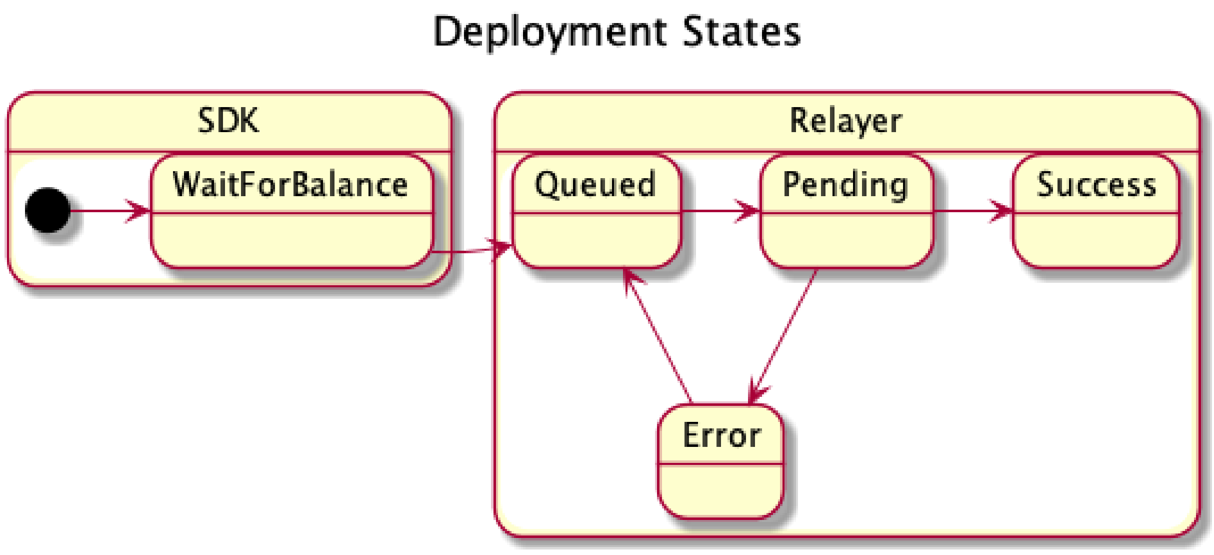
- **ensName** - ENS name chosen by a user. It is the only parameter provided by the user.
- **publicKey** - the public key of a newly generated key pair on the user's device.
- **ensAddress** - the address of ENS contract. It is required to properly register the ENS name.
- **resolverAddress** - the address of Resolver contract. It is required to properly register the ENS name.
- **registrarAddress** - the address of Registrar contract. It is required to properly register the ENS name.
- **gasPrice** - gas price used in the refund process.
- **signature** - the signature of all of the arguments above. The signature ensures parameters come from the owner of the private key (paired to the public key). In particular, it prevents against malicious ENS name registration and gas price replacement.

ENS name collision

Deployment contains ENS name so it could fail (for example when ENS name is taken). That's why we require success on register ENS name. If it fails, the contract won't be deployed, so the user can choose ENS name once again and register it on the same contract address.

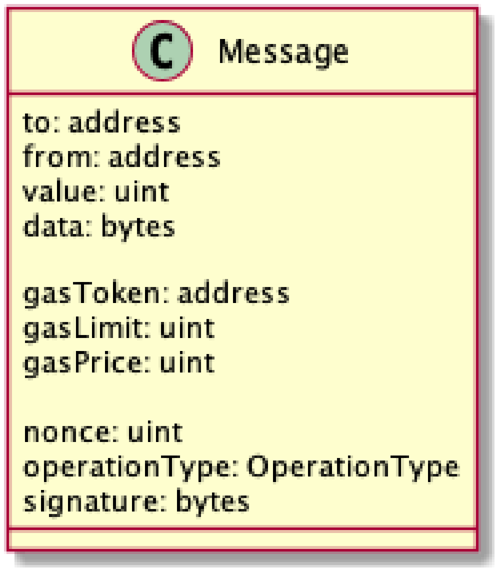
Deployment lifecycle

It starts when the user generates a contract address assigned to him. The first half of deployment is waiting for the user to send funds to the computed contract address.



Meta-transactions

Message (also known as meta-transaction or signed messages) is a way to trigger ethereum transaction from an application or device that does not possess any ether. The message states the intention of the user. It requests a wallet contract to execute a transaction. (eg: funds transfer, an external function call or an internal function call - i.e. an operation in the wallet contract itself). An application sends a message signed with one or more of the keys whitelisted in the contract to the relayer server. The relayer than wraps the message into an ethereum transaction. The message is then processed by the contract as a function call. The relayer wallet is paying for transaction gas. The wallet contract refunds the cost of execution back to the relayer in ether or ERC20 token. The message contains the following parameters:

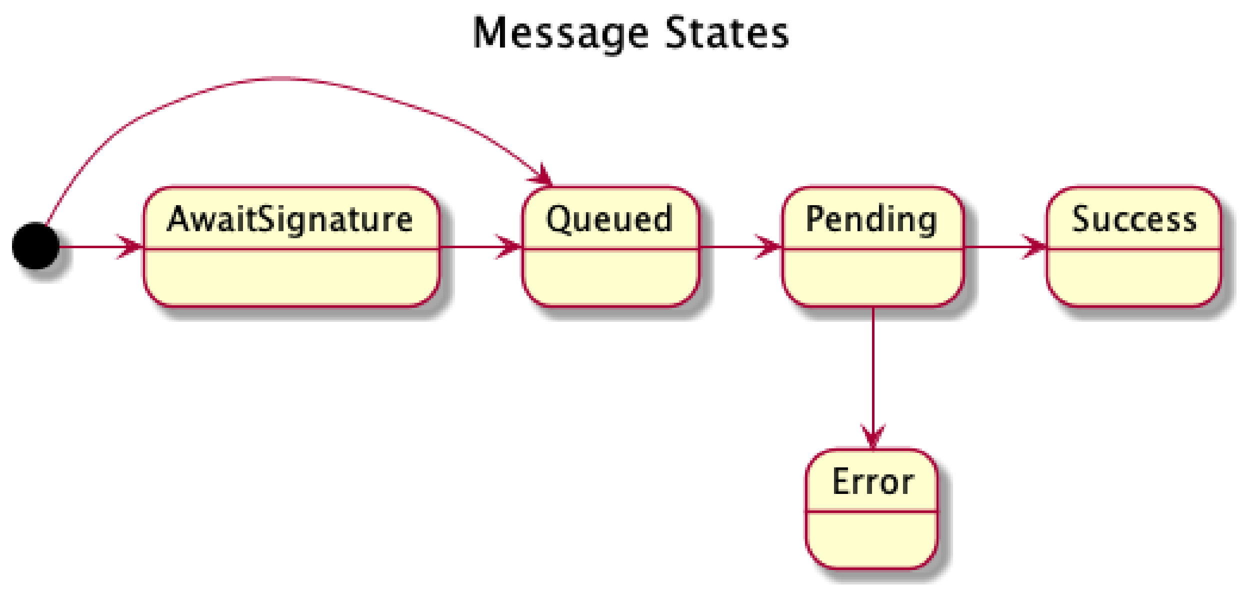


- **to** - the recipient of the message call
- **from** - the address of contract that executes the message
- **value** - number of Wei to send
- **data** - data for the transaction (i.e. an encoded function call)

- **gasToken** - address of token used for refund
- **gasLimit** - maximum gas to use in for a specific transaction
- **gasPrice** - gas price to use in the refund process
- **nonce** - an internal nonce of the transaction relative to the contract wallet
- **operationType** - the type of execution (call, delegatecall, create)
- **signature** - the signature of all of the arguments above, which ensures parameters come from the owner of the allowed public-private key pair

Message lifecycle

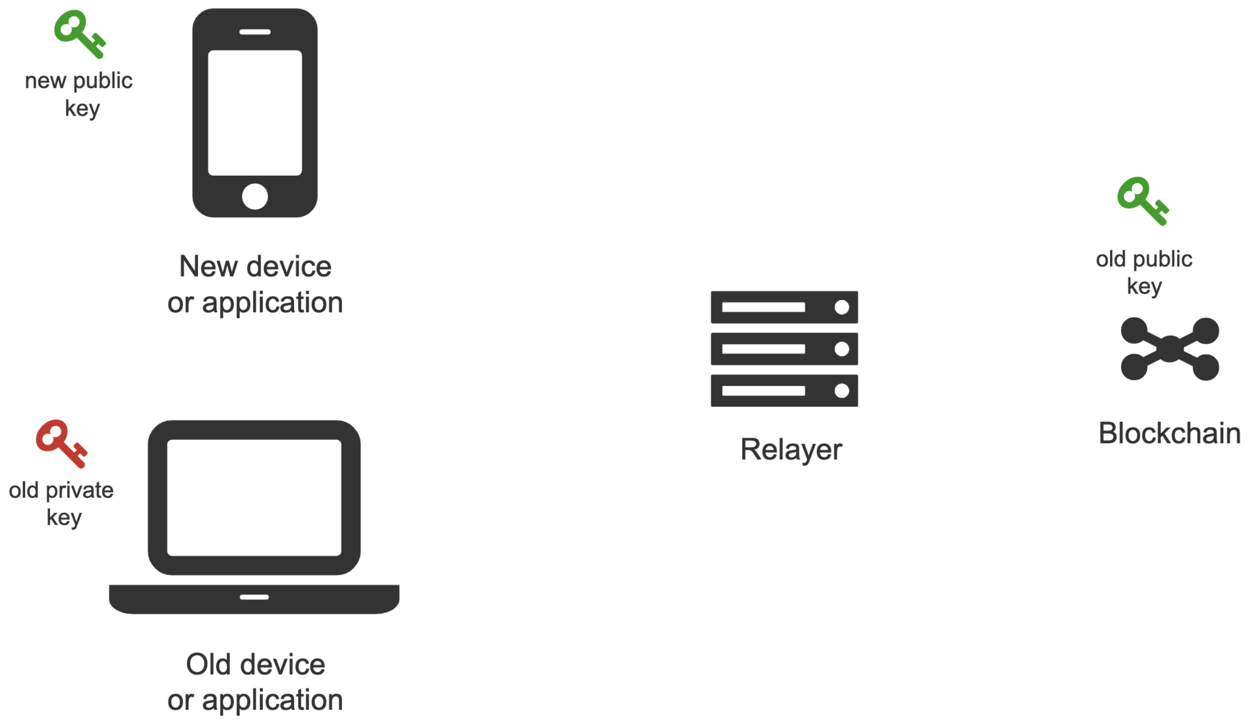
A message starts its journey when it is created and signed by a user (i.e. an application or an SDK) and then sent to a relayer. In the relayer it goes through the following states:



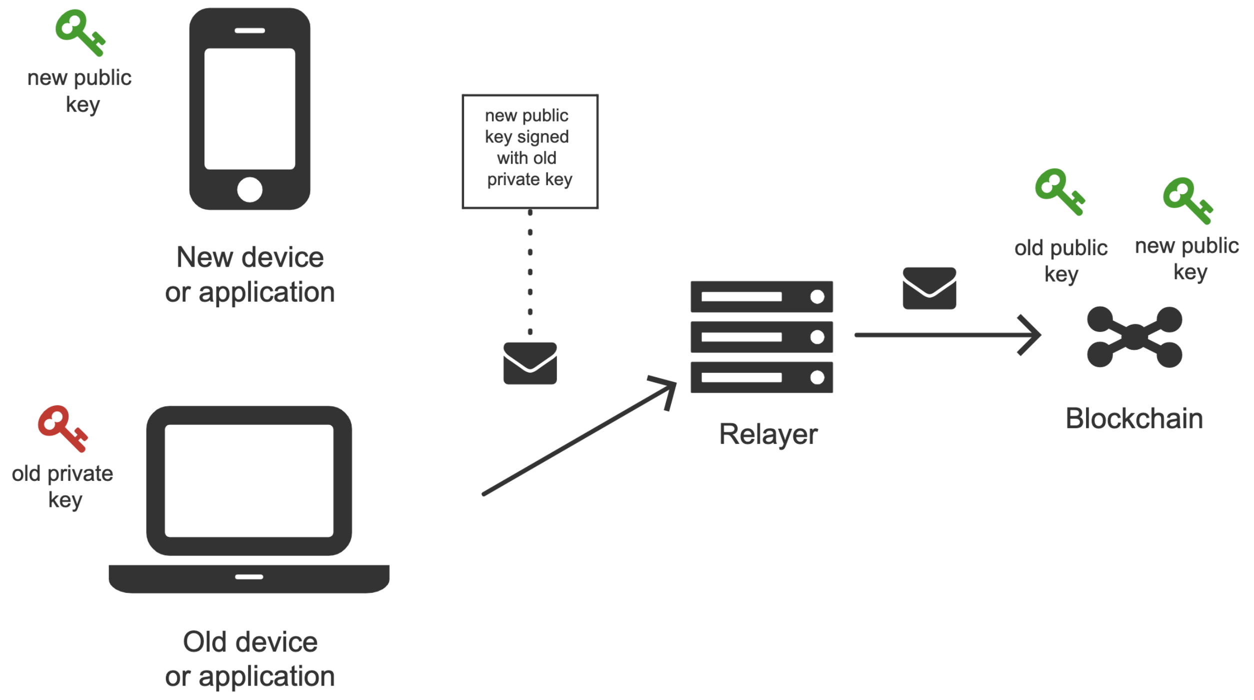
- **await signature** `optional`- The relayer waits to collect all the required signatures if the message requires more than one.
- **queued** - The message is queued to be sent.
- **pending** - The message is propagated to the network and waits to be mined. In a pending state, the message has a transaction hash.
- **sucess / error** - A mined transaction is a success or an error. In a success state, the content of the message status is not changed. In an error state, the message has an error message.

New device connection

One of the key activities is connecting a newly created public key to the existing smart contract wallet. The new public key is created on a new device or application that never interacted with the smart contract wallet before. See below.



The new public key is added using a meta-transaction. The meta-transaction needs to be signed with the private key from a device that is already authorized in the wallet smart contract. After signing, the meta-transaction is sent to the relayer, which propagates it to the blockchain. The picture below shows this process.


new public key
New device or application
new public key signed with old private key
old private key
Old device or application
Relayer
old public key
new public key
Blockchain

There are four key actors in the process:

- **Old device** or an application that is already authorized. Authorized means that there exists a public-private key pair, where the private key is kept on the device and the public key is stored in the wallet smart contract on the blockchain.

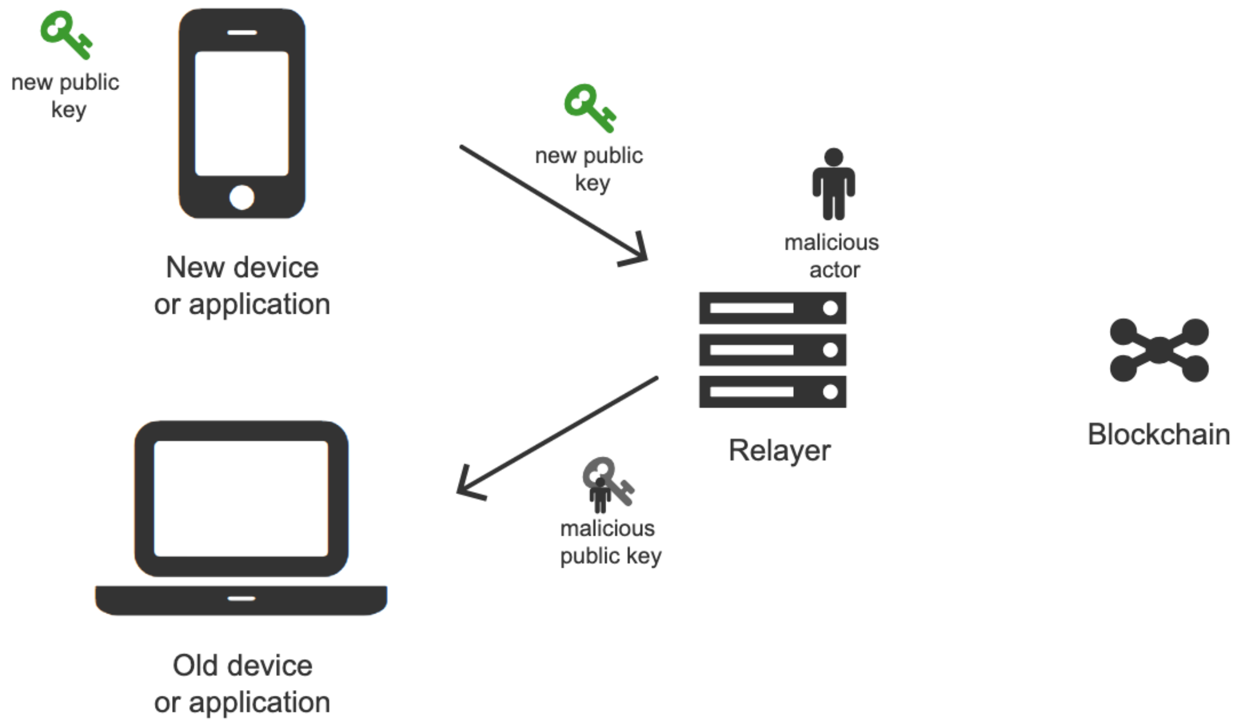
- **New device** (or a new application) that we want to authorize to use the wallet smart contract. To do that we need to generate a **new public-private key pair** and add the new public key to the wallet contract as a management or action key. The public key is added by creating a meta-transaction signed by the old device (old private key) and sending it to the relayer.
- **Relayer** - relays meta-transaction sent from an old device to the blockchain
- **Smart Contract Wallet** - a smart contract that stores keys and executes meta-transactions.

Possible attacks

The problem might seem pretty straightforward, but there are some complexities to consider. In particular, we should avoid introducing the possibility of the following attacks:

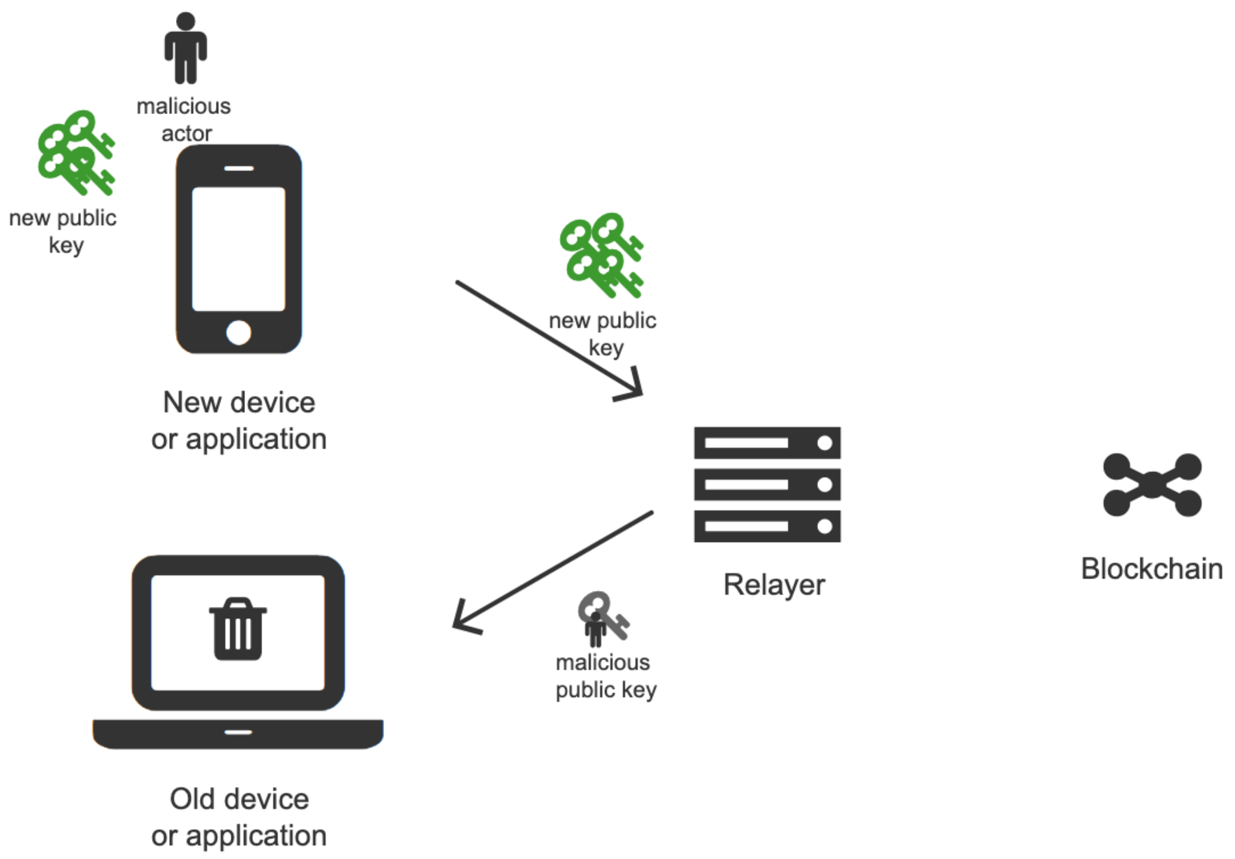
- Man in the middle

A man-in-the-middle attack can happen when the new device sends the new public key to the old device. A malicious actor that intercepts communication (e.g. a relayer) can replace the new public key with a public key that belongs to him and, as a result, take over control of the wallet contract.



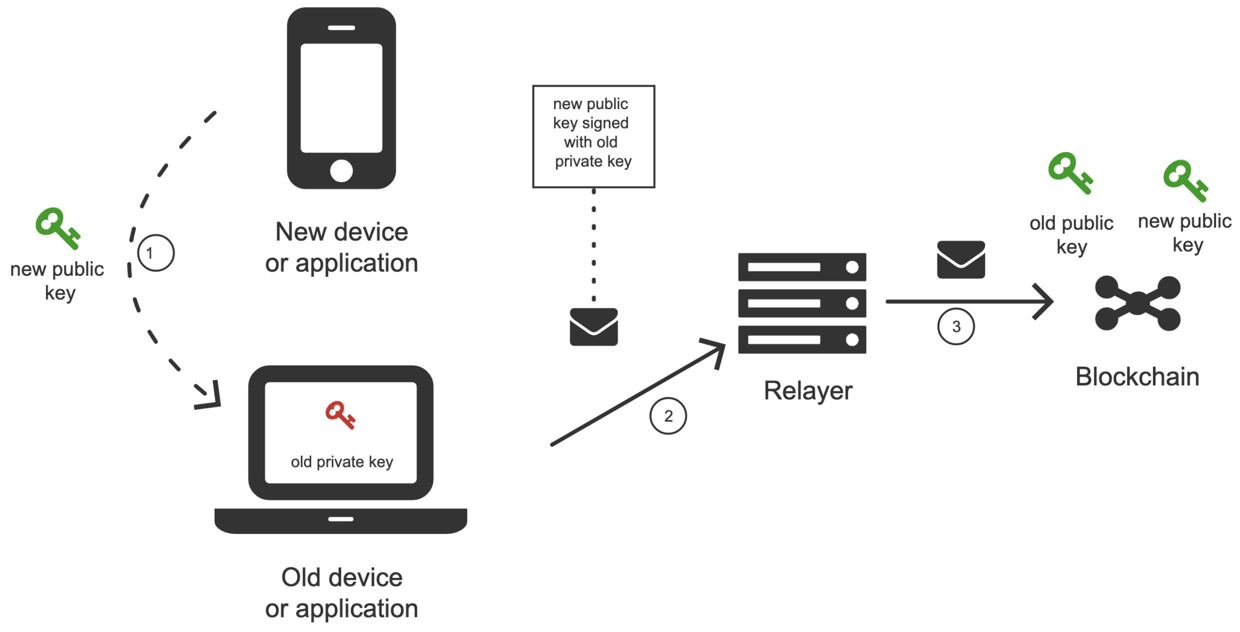
- Spamming

A spam attack can happen when a lot of new devices request to connect to an old device, therefore the old device is spammed with many notifications.



Solution 1

The first solution is pretty straightforward. A new device transfers its public key to the old device.


new public key
1
New device or application
new public key signed with old private key
2
Relayer
3
old public key
new public key
Blockchain
old private key
Old device or application

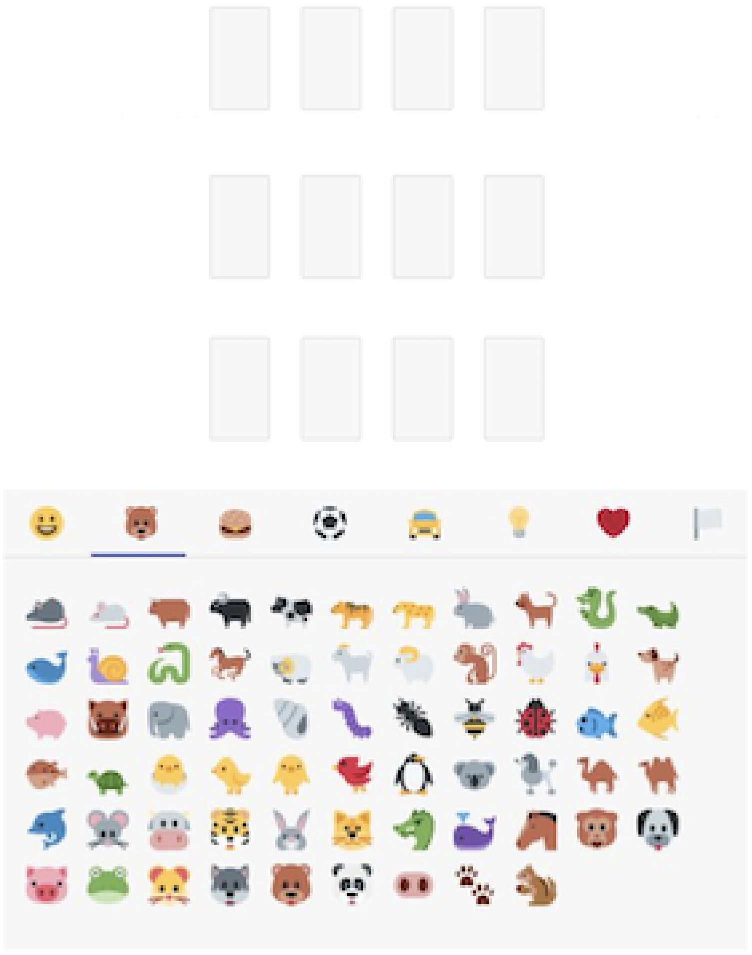
Transfer means

There are two possible ways of transferring the public key.

Note: This is a public key, so we don't worry about intercepting.

Note: The seed for the ecliptic curve key that we use has 128 bits or 16 bytes.

- Scan the QR code
- Manually copy the public key by typing. That might have different shades.
  - Retype the letters (32 chars if hex or 26 with just mix cased letters + digits).
  - Use emojis (12 emojis with 1000 emoji base), see the example interface below.



  - If both applications are on the some one device -> copy paste. (or in some cases even send by e-mail)

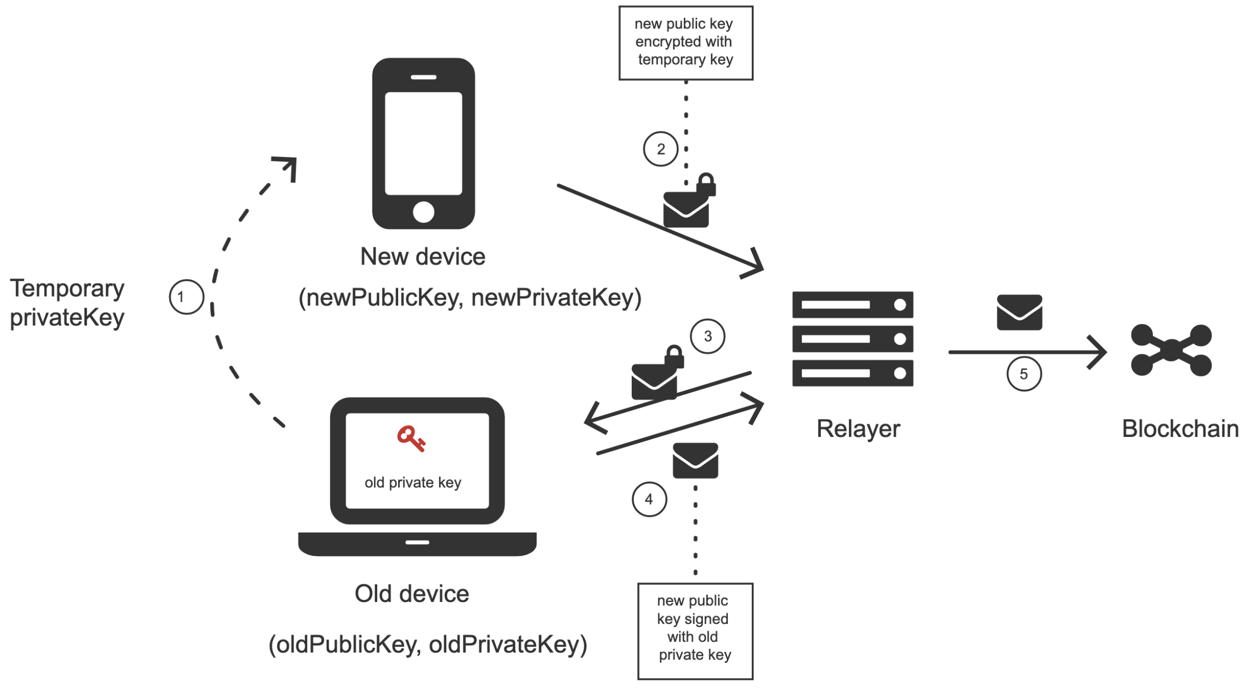
Solution 2

The second solution might be useful if, for some reason, we want to transfer information from an old device to a new device. That might make a difference in the case of using QR codes and the old device does not possess a camera.

The process goes as follows:

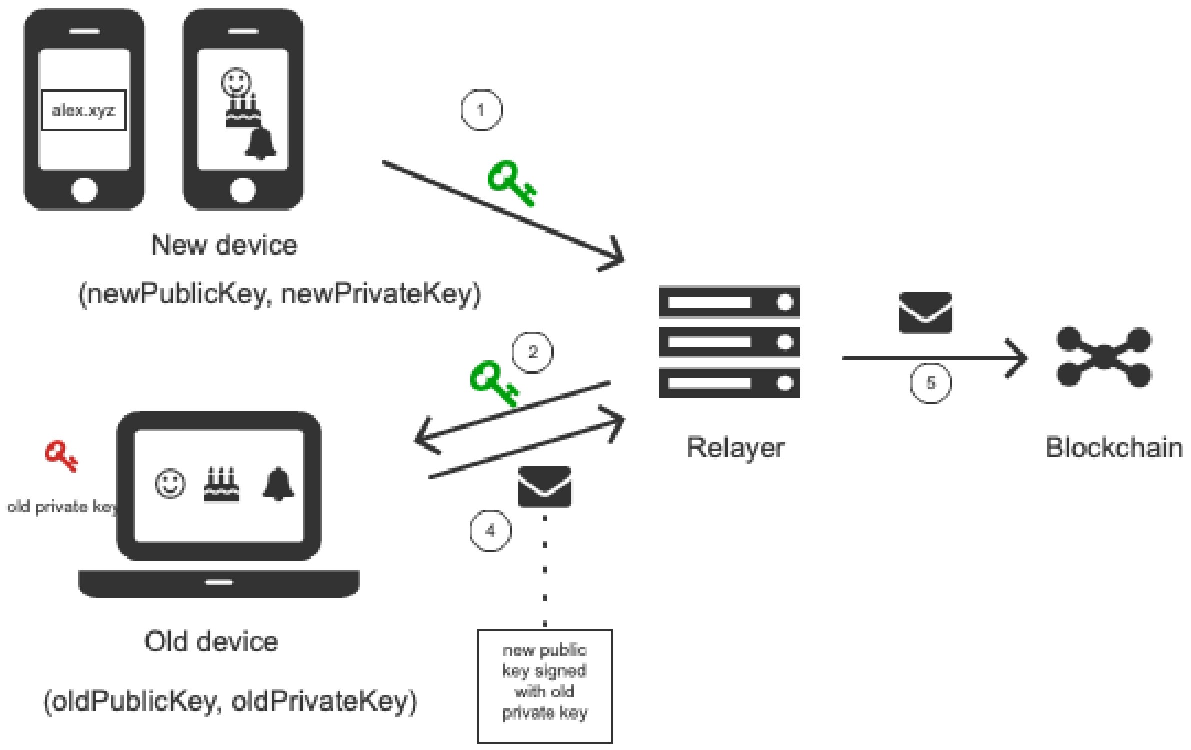
1. The old device generates a temporary key pair.
2. The temporary private key gets transferred to the new device.
3. The new device encrypts a new public key using the temporary private key and transfers it to the old device.

4. The old device decrypts the new device's public key and sends a meta-transaction to the relayer adding it to the wallet smart contract.


new public key encrypted with temporary key
2
New device
(newPublicKey, newPrivateKey)
Temporary privateKey
1
3
Relayer
5
Blockchain
old private key
4
Old device
(oldPublicKey, oldPrivateKey)
new public key signed with old private key

Solution 3

The third solution is an alternative to the previous solutions. The new device generates a new key pair and shows the user emojis based on a hash of the new public key to the later use on an old device. The newly generated public key is sent to a relayer and forwarded to the old device. To finalize connection of a new device, the user has to arrange emojis on the old device in the same order he has seen on the new device. See below.


alex.xyz
New device
(newPublicKey, newPrivateKey)
1
2
4
5
Relayer
Blockchain
old private key
Old device
(oldPublicKey, oldPrivateKey)
new public key signed with old private key

In case of spamming attack in place, the user has to type the emojis manually.

1.2.3 Development environment

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

Read more in *tutorial*

1.3 Tutorial

1.3.1 Quickstart

New project

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

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

If you are using yarn than type:

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

Development environment

Prerequisites Before running the development environment, make sure you have **PostgreSQL** installed, up and running.

Installation To use the 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 the development environment is to add a script to `package.json` file:

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

Running development environment To start the 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
  0x5e8b3a7e6241CeE1f375924985F9c08706f41d34 -␣
↪0x837fd366bc7402b65311de9940de0d6c0ba3125629b8509aebbfb057ebeaaa25
  0xFC6F167a5AB77Fe53C4308a44d6893e8F2E54131 -␣
↪0xba35c32f7cbda6a6cedeea5f73ff928d1e41557eddfd457123f6426a43adb1e4
  0xDe41151d0762CB537921c99208c916f1cC7dA04D -␣
↪0x71f7818582e55456cb575eea3d0ce408dcf4cbbc3d845e86a7936d2f48f74035
  0x121199e18C70ac458958E8eB0BC97c0Ba0A36979 -␣
↪0x03c909455dcef4e1e981a21ffb14c1c51214906ce19e8e7541921b758221b5ae
```

(continues on next page)

(continued from previous page)

```
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
```

1.3.2 Using the SDK

Creating a wallet contract

To start using the SDK you will need to create an 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 a relayer address, second is an Ethereum node address.

Sending a meta-transaction

Once you have the contract wallet deployed you can execute a transaction via relayer:

```
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 the contract address.

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

The token need to be supported by a relayer. The wallet contact needs to have enough token balance to refund the transaction.

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

1.3.3 Connecting to an existing app on testnet

Create a wallet contract

Create your own wallet contract using Universal Login Wallet 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://ropsten.infura.io';

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

Start listening for events

Then make UniversalLoginSDK start listening for relayer and blockchain events

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

Request a connection

Now, you can request a connection to the created wallet contract

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

Subscribe to KeyAdded

Subscribe to `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 a connection request

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

Stop listening for events

Remember to stop listening for relayer and blockchain events

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

1.3.4 Helpers

Prerequisites

Install the universal-login toolkit:

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

Test token

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

Example:

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

Sending funds

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

Parameters:

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

Example:

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

1.3.5 ENS registration

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

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

You can register the domain in two ways: from command line and programmatically. To use a 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 - the address of an ENS contract
- **PUBLIC_RESOLVER_ADDRESS** : string - the address of a public resolver. For the Ropsten test network a working public resolver address is `0x4C641FB9BAd9b60EF180c31F56051cE826d21A9A` and for the Rinkeby test network a 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 an ENS domain, in universal-login-ops directory type in the console:

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

Parameters:

- **my-domain** - a domain to register
- **tld** - a 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 your 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 - a private key to execute registrations
  - **ensAddress** : string - the address of an ENS contract
  - **publicResolverAddress** : string - the address of a 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 a new domain and saves all information about newly registered domain to a new file (a registrar address or resolver address)

Parameters:

- **domain** : string - a domain to register
- **tld** : string - a 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'
```

1.4 SDK

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

1.4.1 Creating an SDK

new UniversalLoginSDK(relayerURL, providerURL, messageOptions)

Parameters:

- **relayerURL** : string - a URL address of a 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',
  'http://localhost:18545',
  messageOptions
);
```

1.4.2 Creating a wallet contract

createFutureWallet

sdk.createFutureWallet()

Creates a FutureWallet, which contains all information required to deploy and use a Wallet in the future.

Returns: *promise* that resolves to `FutureWallet`.

FutureWallet contains:

- *privateKey* - that will be connected to ContractWallet. The key will be used to sign transactions once the wallet is deployed.
- *contract address* - an address under which the wallet will be deployed in the future.
- *waitForBalance* - a function that waits for a contract address balance change in a way that will allow the wallet contract to be deployed.

  Returns: *promise*, that resolves (only when the wallet contract balance is changed to satisfy relayer requirements) to `{tokenAddress, contractAddress}`
- *deploy* - a function that requests wallet contract deployment.

  Parameters:

  - **ensName** : string - a chosen ENS name
  - **gasPrice** : string - gas price of a deployment transaction

  Returns: *promise* that resolves to the deployed wallet contract address

Example:

```
const {privateKey, contractAddress, waitForBalance, deploy} = await sdk.
↪createFutureWallet();
await waitForBalance();
await deploy('myname.example-domain.eth');
```

connect

sdk.connect(contractAddress)

requests adding a new key to a contract.

Parameters:

- **contractAddress** : string - an address of the contract to manage a connection

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

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

Example:

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

denyRequest

sdk.denyRequest(contractAddress, publicKey, privateKey)

removes the request for adding a new key from pending authorizations.

Parameters:

- **contractAddress** : string - an address of a contract to remove a request
- **publicKey** : string - an address to remove from add requests
- **privateKey** : string - a private key to sign a request

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

- *publicKey* - an address removed from pending authorisations

Example:

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

1.4.3 Transaction execution

execute

sdk.execute(message, privateKey)

executes any message.

Parameters:

- **message** : object - a message that is sent to a contract, includes:
  - from : string - an address of the contract that requests execution
  - to : string - a beneficient of this execution
  - data : string - the 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 a permission to execute the message

Returns: *promise* that resolves to the `Execution`

Execution contains:

- **messageStatus** - a current status of the sent message (*learn more*)
- **waitToBeMined** - a function that returns a promise that resolves to MessageStatus once the transaction enclosed with Message is mined

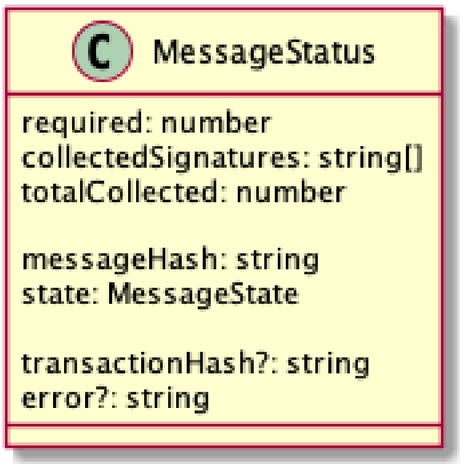
Example:

```
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`.

messageStatus



- **required** : number - the amount of required signatures to execute the message
- **collectedSignatures** : string[] - signatures collected by a relayer

- **totalCollected** : number - the amount of collected signatures
- **messageHash** : string - hash of the message
- **state** : MessageState - one of the message states: `AwaitSignatures`, `Queued`, `Pending`, `Error`, `Success`
- **transactionHash** (optional) : string - a transaction hash is only possible when the message state is `Pending`, `Success` or `Error`
- **error** (optional) : string - only when the message state is `Error`

sdk.getMessageStatus(messageHash)

requests a message status of a specific message

Parameters:

- **messageHash** - a hash of a message

Returns: *promise* that resolves to `MessageStatus`

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.

1.4.4 Managing a wallet contract

addKey

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

adds a key to manage a wallet contract.

Parameters:

- **contractAddress** : string - an address of a contract that requests to add a new key
- **publicKey** : string - a public key to manage the contract
- **privateKey** : string - a private key that has a 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 *Execution*

Example:

```
const transactionDetails = {
  gasToken: '0x850437540FE07d02045f88cAe122Bc66B1BdE957',
  gasPrice: 1000000,
  gasLimit: 150000
```

(continues on next page)

(continued from previous page)

```
};
await sdk.addKey(
  '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
  '0x96E8B90685AFD981453803f1aE2f05f8Ebc3cfD0',
  '0x5c8b9227cd5065c7e3f6b73826b8b42e198c4497f6688e3085d5ab3a6d520e74',
  transactionDetails,
  ACTION_KEY
);
```

addKeys

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

adds multiple keys to manage a contract.

Parameters:

- **contractAddress** : string - an address of a contract that requests to add keys
- **publicKeys** : array of strings - public keys to add
- **privateKey** : string - a private key that has a 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 *Execution*

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 a key from a contract.

Parameters:

- **contractAddress** : string - an address of a contract that we want to remove a key from the contract

- **publicKey** : string - a public key to remove
- **privateKey** : string - a private key with a permission of removing keys
- **transactionDetails** : object - an optional parameter that includes details of transactions for example gasLimit or gasPrice

Returns: *promise* that resolves to the *Execution*

Example

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

getWalletContractAddress(ensName)

gets a wallet contract address by an ENS name

Parameters:

- **ensName** : string - an ENS name

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

Example:

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

walletContractExist(ensName)

checks if an ENS name is registered.

Parameters:

- **ensName** : string - an ENS name

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

Example:

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

1.4.5 Events

Key added and key removed

sdk.start()

Starts listening to blockchain events and fetches supported tokens detials.

sdk.stop()

Stops listening to blockchain events.

sdk.subscribe(eventType, filter, callback)

subscribes KeyAdded or KeyRemoved event.

Parameters:

- **eventType** : string - a type of an event, possible event types: `KeyAdded`, `KeyRemoved`
- **filter** : object - a filter for events, includes:
  - contractAddress : string - an address of a contract to observe
  - key : string - a public key used to subscribe to an event
- **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
```

subscription.remove()

removes subscription

Example:

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

Authorisations

sdk.subscribeAuthorisations(walletContractAddress, privateKey, callback)

subscribes AuthorisationChanged event

Parameters:

- **walletContractAddress** : string - an address of a contract to observe

- **privateKey** : string - a private key used to sign a get authorization request
- **callback**

Returns: unsubscribe function

Example:

```
const unsubscribe = sdk.subscribe(
  '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
  '0x5c8b9227cd5065c7e3f6b73826b8b42e198c4497f6688e3085d5ab3a6d520e74',
  (authorisations) => {
    console.log(`${authorisations}`);
    unsubscribe();
  }
);
```

Result

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

1.5 Relayer

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

Below are the instructions on how to run the relayer.

If you would like to use your own domain, jump to the section *ENS registration*.

1.5.1 Starting a relayer

Prerequisites

Database

To run a relayer in a 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
```

Factory contract

To run a relayer you also need to deploy your own factory contract, with the same wallet that the relayer will have. To do that you will need a wallet master contract address (you can deploy your own or use ours). To deploy the factory contract run:

```
universal-login deploy:factory [walletMasterAddress] --privateKey 'YOUR_
↪PRIVATE_KEY' --nodeUrl 'JSON-RPC URL'
```

Example

```
universal-login deploy:factory 0xfb152D3b3bB7330aA52b2504BF5ed1f376B1C189 --
↪privateKey 'YOUR_PRIVATE_KEY' --nodeUrl https://ropsten.infura.io
```

From command line

To start a relayer from the 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 - a relayer endpoint
- **PRIVATE_KEY** : string - a private key of a relayer wallet
- **ENS_ADDRESS** : string - an address of an ENS contract
- **ENS_DOMAIN** : string - a name of a domain
- **WALLET_MASTER_ADDRESS** : string - WalletMaster contract address
- **FACTORY_ADDRESS** : string - Factory contract address

example .env file

```
JSON_RPC_URL='https://ropsten.infura.io'
PORT=3311
PRIVATE_KEY='YOUR_PRIVATE_KEY'
ENS_ADDRESS='0x112234455c3a32fd11230c42e7bccd4a84e02010'
ENS_DOMAIN_1='poppularapp.test'
ENS_DOMAIN_2='my-login.test'
ENS_DOMAIN_3='universal-login.test'
WALLET_MASTER_ADDRESS='0xfb152D3b3bB7330aA52b2504BF5ed1f376B1C189'
FACTORY_ADDRESS='0xE316A2134F6c2BE3eeFdAde5518ce3F685af27E7'
```

2. Run relayer

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

```
yarn relayer:start
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search