

---

# **universal-login Documentation**

*Release 0.0.1*

**Marek Kirejczyk, Justyna Broniszewska**

**Feb 28, 2020**



---

## Contents:

---

<b>1</b>	<b>Disclaimer</b>	<b>3</b>
1.1	Getting started . . . . .	3
1.2	Overview . . . . .	4
1.3	Tutorial . . . . .	14
1.4	SDK . . . . .	21
1.5	Relayer . . . . .	30
<b>2</b>	<b>Indices and tables</b>	<b>33</b>





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.



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

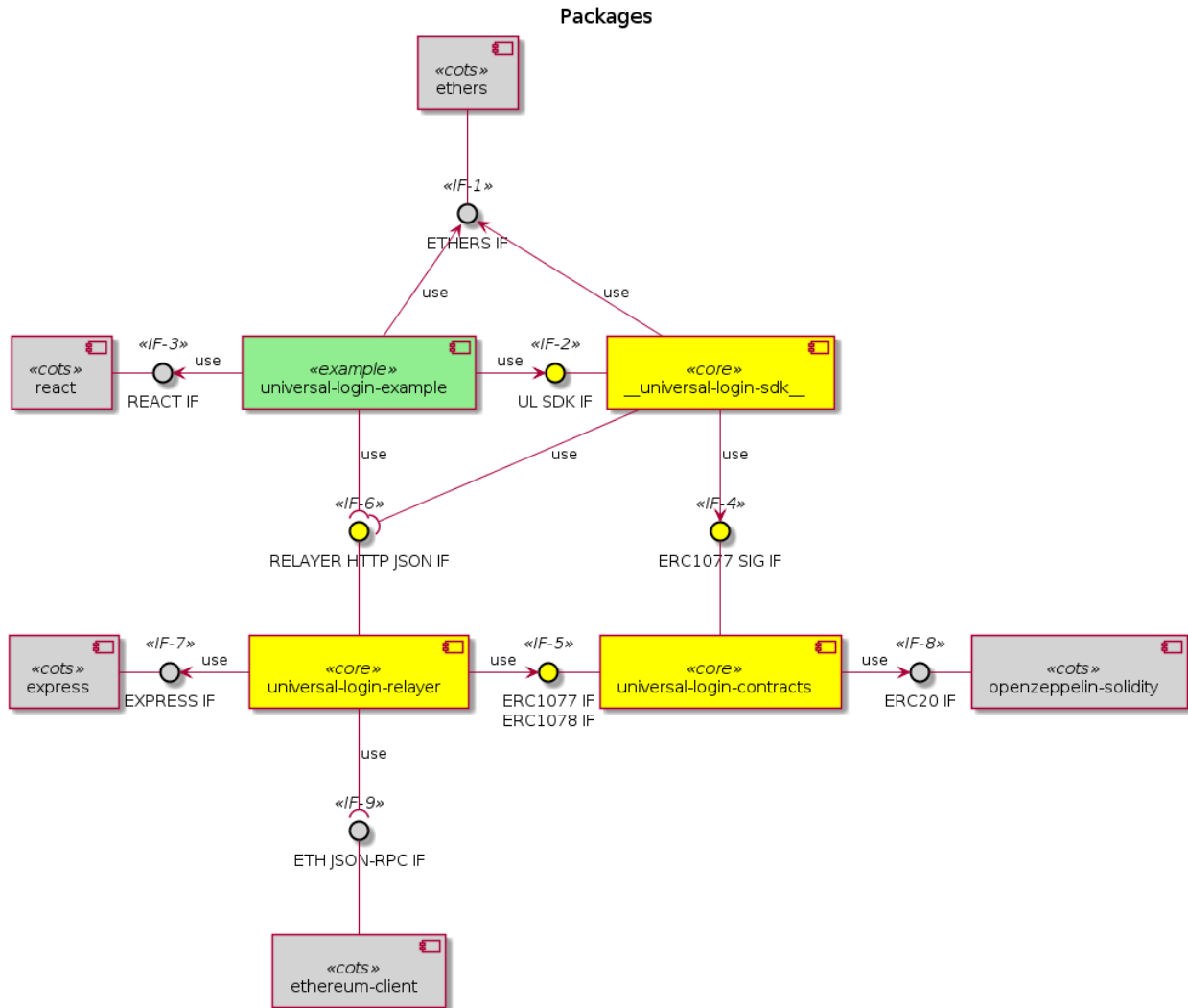
Universal Login has four components. All components are stored in one monorepo [available here](#). Components are listed below:

- **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 Relay 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

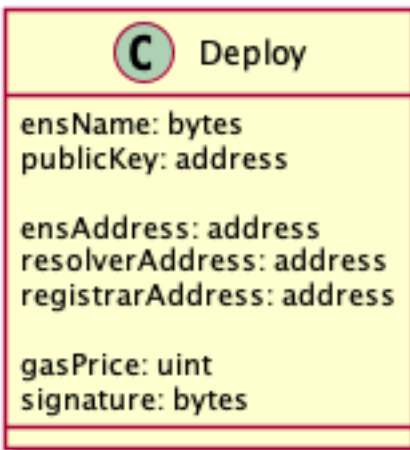
Deployment is designed in a way that makes the user pay for himself. To do that, we use counterfactual deployment (`create2` function to deploy contract).

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.

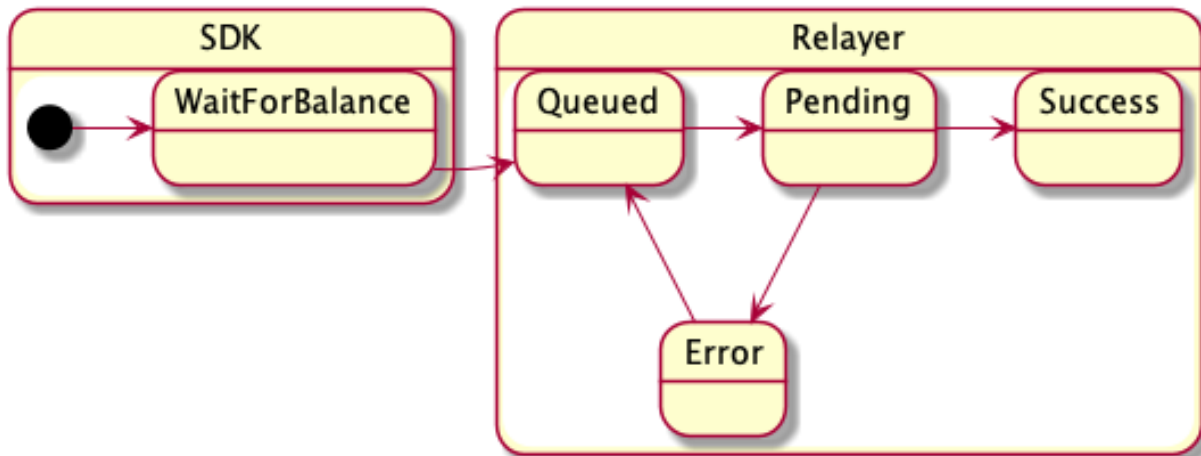
### ENS name collision

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

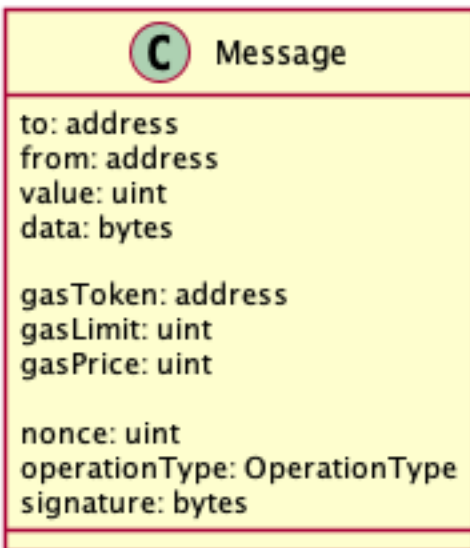
It starts when the user generates a contract address assigned to him. The first half of deployment is waiting for the user to send funds to the computed contract address.

## Deployment States



### 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 then 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:

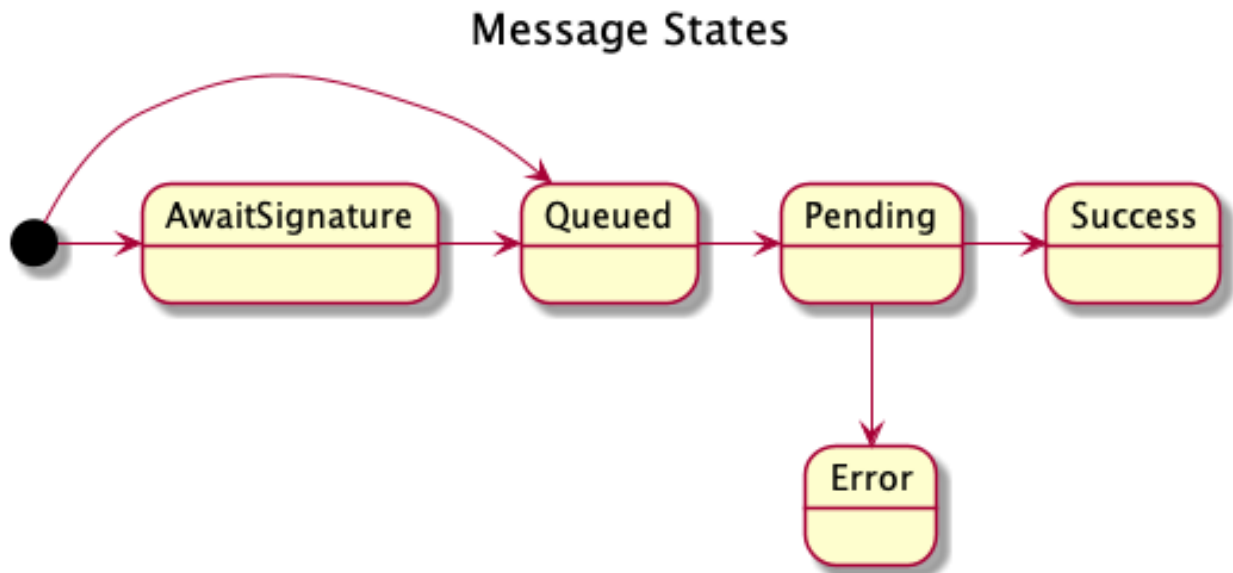


- **to** - the recipient of the message call
- **from** - the address of contract that executes the message
- **value** - number of Wei to send
- **data** - data for the transaction (i.e. an encoded function call)

- **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

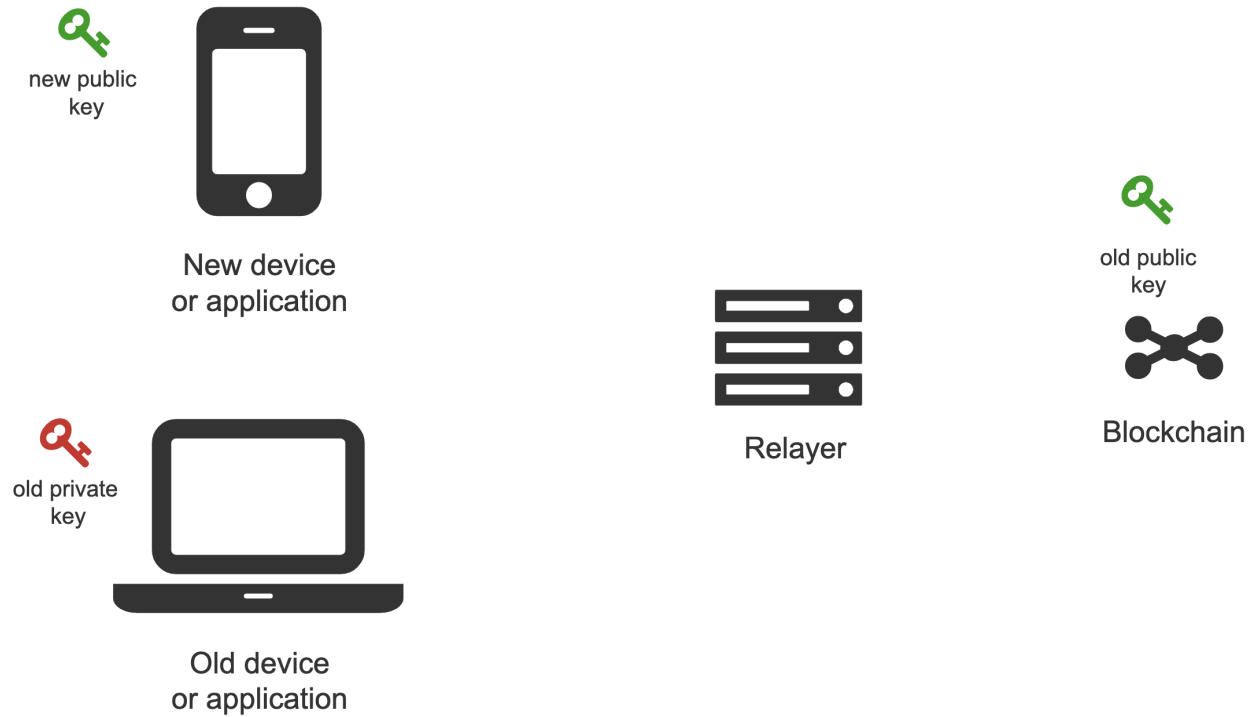
A message starts its journey when it is created and signed by a user (i.e. an application or an SDK) and then sent to a relayer. In the relayer it goes through the following states:



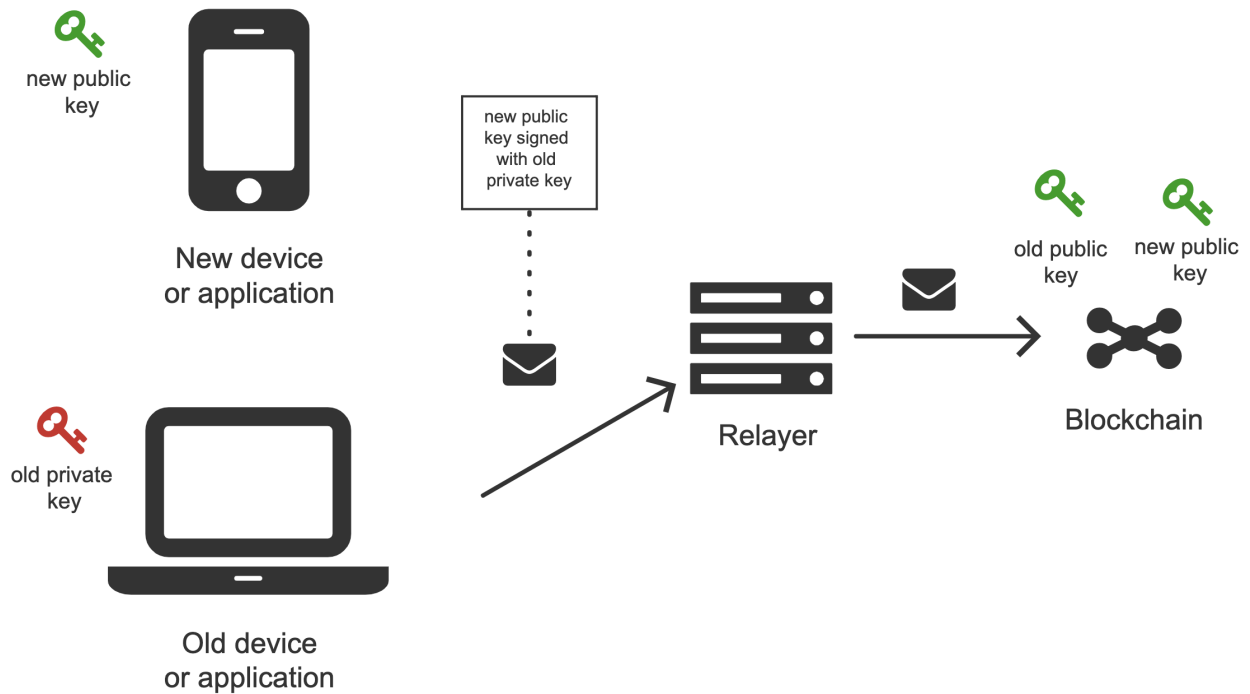
- **await signature** *optional*- The relayer waits to collect all the required signatures if the message requires more than one.
- **queued** - The message is queued to be sent.
- **pending** - The message is propagated to the network and waits to be mined. In a pending state, the message has a transaction hash.
- **success / 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.



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.

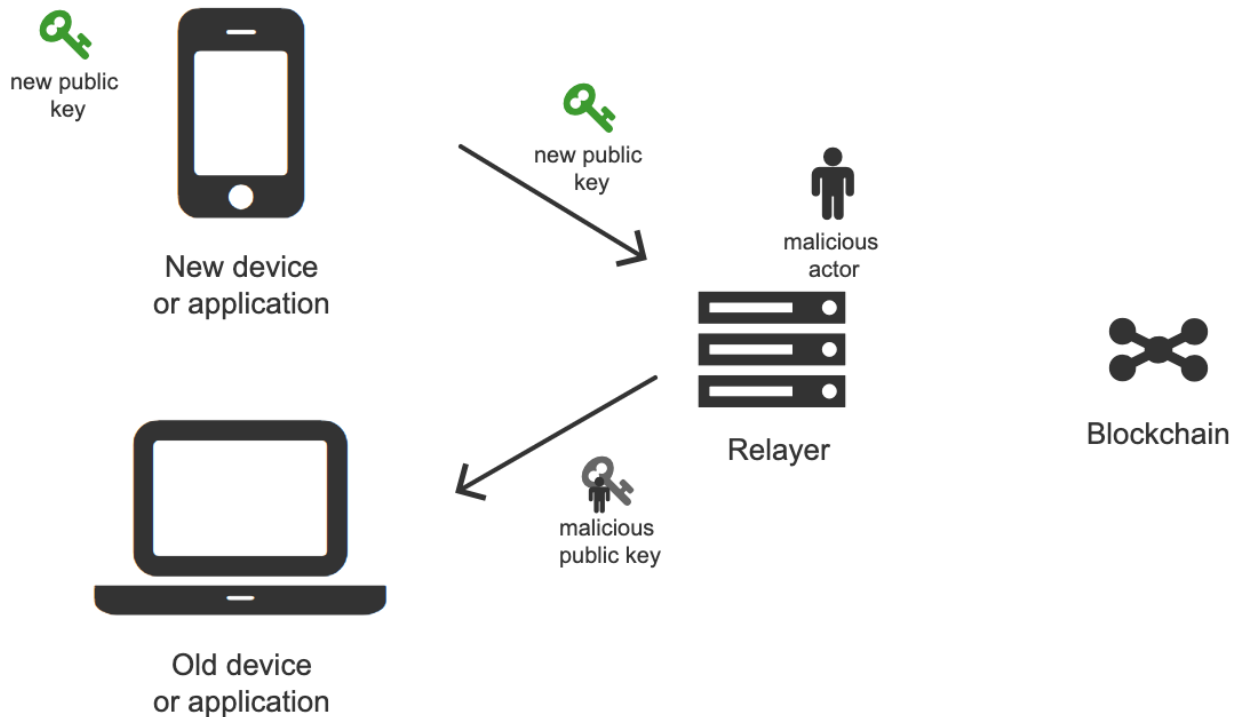
- **New device** (or a new application) that we want to authorize to use the wallet smart contract. To do that we need to generate a **new public-private key pair** and add the new public key to the wallet contract as a management or action key. The public key is added by creating a meta-transaction signed by the old device (old private key) and sending it to the relay.
- **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:

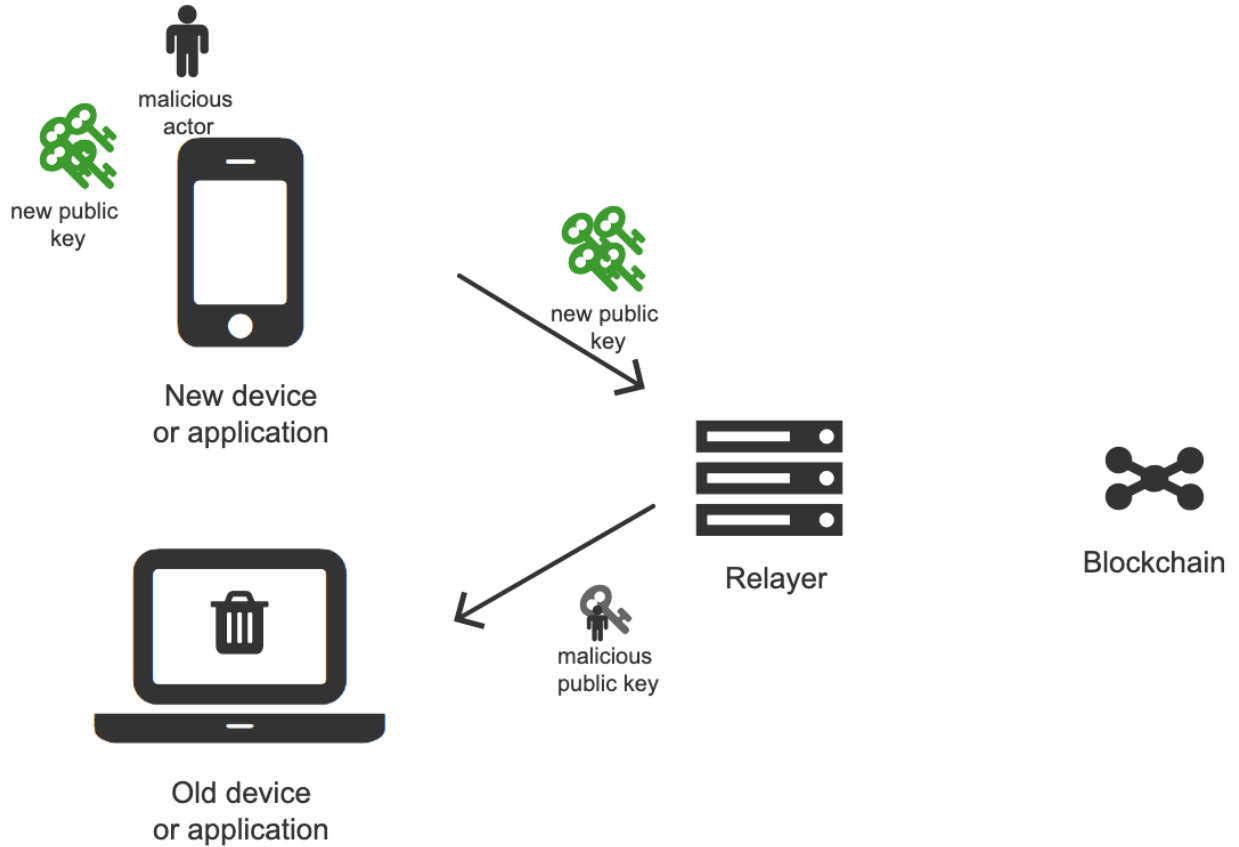
- Man in the middle

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 relay) 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.



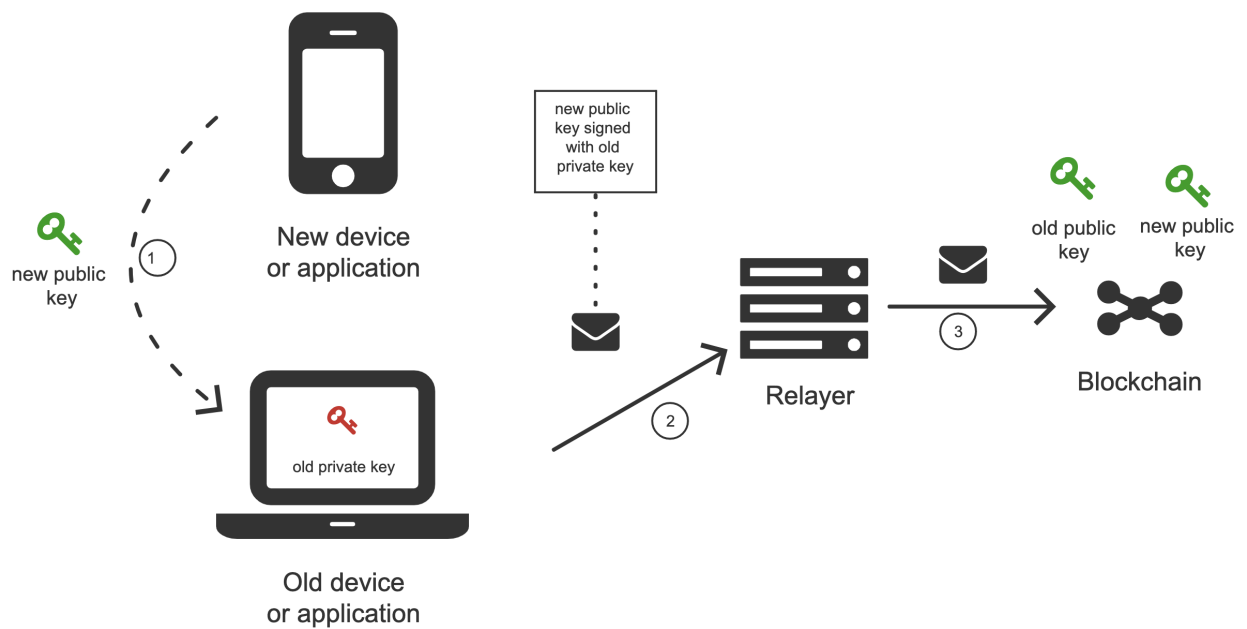
- Spamming

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.



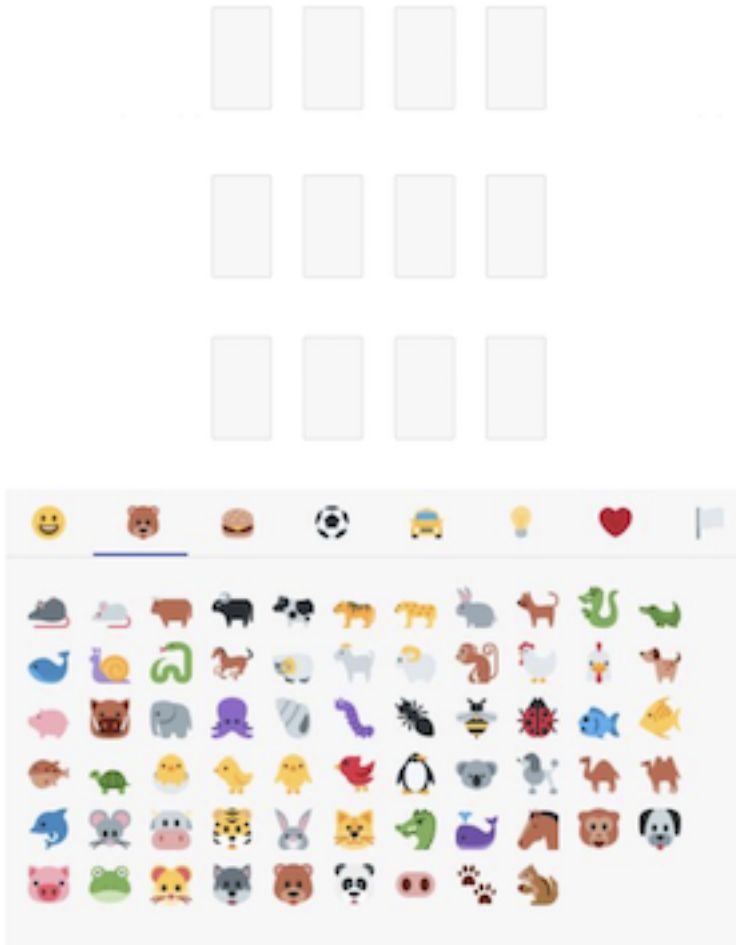
**Transfer means**

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 elliptic 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 same one device -> copy paste. (or in some cases even send by e-mail)

### Solution 2

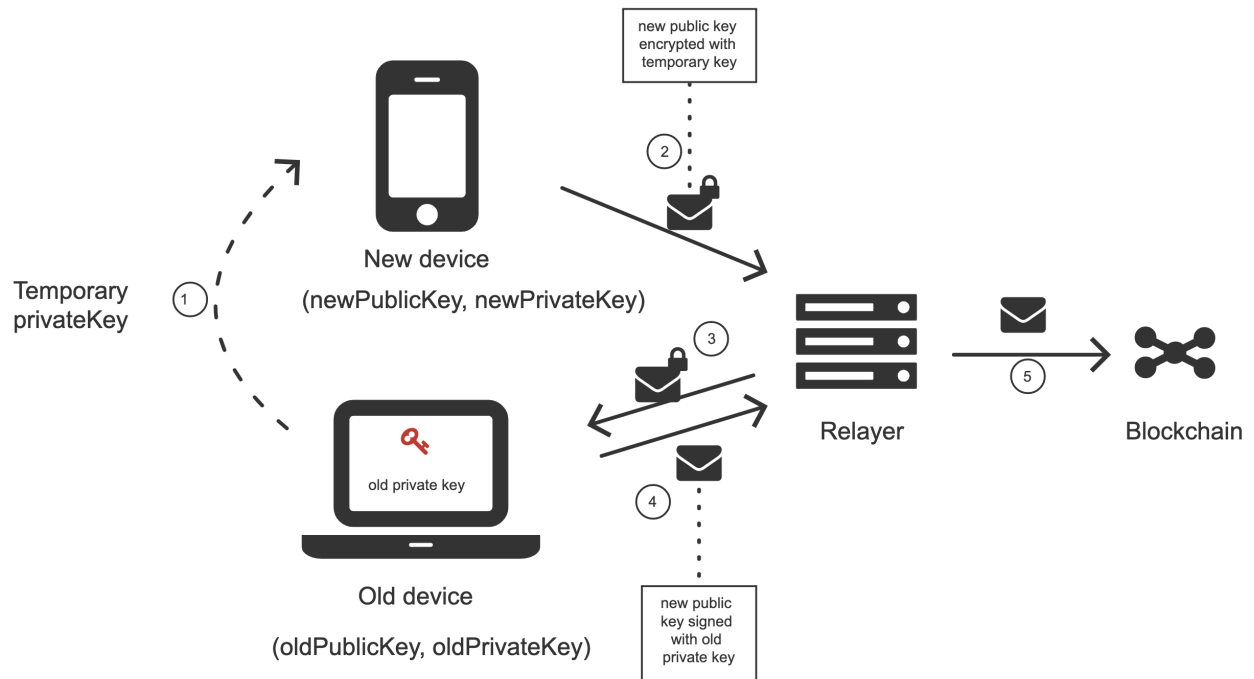
The second solution might be useful if, for some reason, we want to transfer information from an old device to a new device. That might make a difference in the case of using QR codes and the old device does not possess a camera.

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.

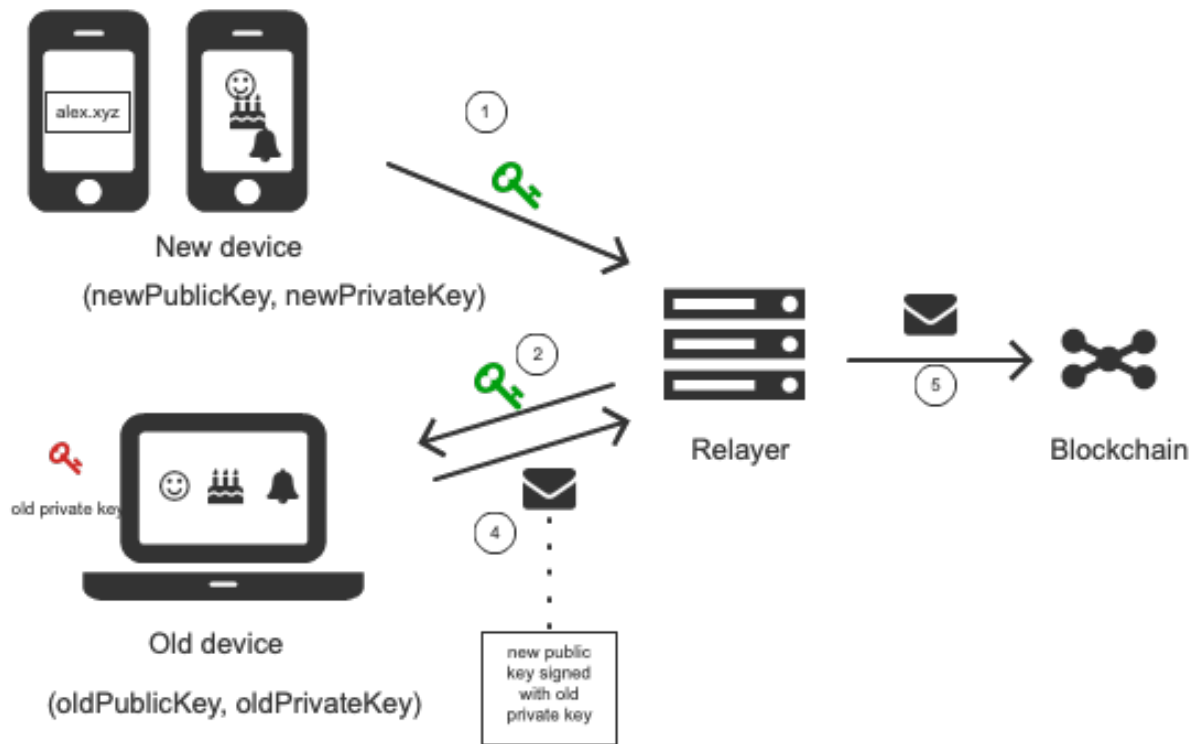


- The old device decrypts the new device's public key and sends a meta-transaction to the relay adding it to the wallet smart contract.



### 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 relay 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.



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 @unilogin/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

#### Starting with Web3 provider

If you are developing a web application the easiest way to get started with Universal-Login is by using our Web3 provider.

## Installation

To add the provider to your project using npm type the following:

```
npm i web3 @unilogin/web3 react react-dom
```

If you are using yarn than type:

```
yarn add @unilogin/web3 react react-dom
```

## Creating the provider

```
const ulProvider = ULWeb3Provider.getDefaultProvider('mainnet');  
const web3 = new Web3(ulProvider);
```

Now you can continue writing your application as a normal Web3 DApp. To read more how to write web3 app go to [Web3 docs](#).

**User onboarding** First time you send a transaction the modal will appear asking users to create an account. After user finishes the onboarding flow the transaction will be executed as normal

You can also trigger an early onboarding (for example if you have a “login” button on your website). To do so call the `create` on the provider:

```
await ulProvider.create();
```

The promise will be resolved once the user completes the flow.

**Detecting Universal-Login** To detect whether the web3 is running with UniversalLogin use the following code:

```
const usingUniversalLogin = web3.currentProvider && web3.currentProvider.  
  ↳isUniLogin;
```

## Starting with SDK

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

```
npm i @unilogin/sdk
```

If you are using yarn than type:

```
yarn add @unilogin/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 `@unilogin/ops` as dev dependency to your project.

With npm:

```
npm install @unilogin/ops --save-dev
```

With yarn:

```
yarn add --dev @unilogin/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 -_
↪0x837fd366bc7402b65311de9940de0d6c0ba3125629b8509aebbf057ebeaaa25
  0xFC6F167a5AB77Fe53C4308a44d6893e8F2E54131 -_
↪0xba35c32f7cbda6a6cedeea5f73ff928d1e41557eddf457123f6426a43adb1e4
  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
  Relay 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 '@unilogin/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 relay address, second is an Ethereum node address.

### Sending a meta-transaction

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

```
const message = {
  from: '0xA193E42526F1FEA8C99AF609dcEabf30C1c29fAA',
  to: '0xbA03ea3517ddcD75e38a65EDEB4dD4ae17D52A1A',
  data: '0x0',
  value: '5000000000000000000',
  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 relay. The wallet contract 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 '@unilogin/sdk';
import ethers from 'ethers';

const relayUrl = 'https://relay.universallogin.io';
const jsonRpcUrl = 'https://ropsten.infura.io';

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

### Start listening for events

Then make `UniversalLoginSDK` start listening for relay 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 @unilogin/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 relay, type its name in relay config.

### From command line

To register an `.eth` ENS domain type in the console:

```
universal-login register:eth:domain [my-domain] --ensAddress [ensAddress] --
↳privateKey [privateKey] --nodeUrl [url] --gasPrice [gasPrice]
```

### Parameters:

- **my-domain** - a domain to register
- **ensAddress** : string - the address of an ENS contract ([list of ENS addresses](<https://docs.ens.domains/ens-deployments>) on public networks)
- **privateKey** : string - private key to execute registrations. *Note:* You need to have ether on it to pay for contracts deployment.
- **nodeUrl** : string - JSON-RPC URL of an Ethereum node
- **gasPrice** : string *optional* - gas price of transactions, default gas price value is 9 gwei

To register `cool-domain.eth` (each transaction's `gasPrice` set to 11 gwei):

```
universal-login register:eth:domain cool-domain --ensAddress_
↳0x314159265dd8dbb310642f98f50c066173c1259b --privateKey 'YOUR_PRIVATE_KEY' --
↳nodeUrl 'https://mainnet.infura.io' --gasPrice 11000000000
```

Result:

```
Registering cool-domain.eth...
New public resolver deployed: 0x75242e98198486fe0307e5ee307f340Af5c950a4
Resolver for cool-domain.eth set to 0x75242e98198486fe0307e5ee307f340Af5c950a4_
↳(public resolver)
New registrar deployed: 0x09873b81932b7726595b86Fe8612c82e65bdB9Fe
cool-domain.eth owner set to: 0x09873b81932b7726595b86Fe8612c82e65bdB9Fe_
↳(registrar)
```

Note: You must be the owner of the domain that means you need to buy it. You can buy an ENS domain for example [here<https://ethsimple.com/>](https://ethsimple.com/)‘\_

To register an `.test` ENS domain type in the console:

```
universal-login register:test:domain [my-domain] [publicResolverAddress] --
↳ensAddress [ensAddress] --privateKey [privateKey] --nodeUrl [url]
```

Parameters:

- **my-domain** - a domain to register
- **publicResolverAddress** : 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`.
- **ensAddress** : string - the address of an ENS contract ([‘list of ENS addresses<https://docs.ens.domains/ens-deployments>](https://docs.ens.domains/ens-deployments)‘\_ on public networks)
- **privateKey** : string - private key to execute registrations. *Note:* You need to have ether on it to pay for contracts deployment.
- **nodeUrl** : string - JSON-RPC URL of an Ethereum node

To register `cool-domain.test` on a test network that supports registration of test domains (e.g. Ropsten, Rinkeby, Görli) and connect it to resolver at address `0x4C641FB9BAD9b60EF180c31F56051cE826d21A9A` type following:

```
universal-login register:test:domain cool-domain_
↳0x4C641FB9BAD9b60EF180c31F56051cE826d21A9A --ensAddress_
↳0x112234455c3a32fd11230c42e7bccd4a84e02010 --privateKey 'YOUR_PRIVATE_KEY' --
↳nodeUrl 'https://ropsten.infura.io'
```

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)
```

Note: use `.test` tld only on testnets.



## Programmatically

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

`new DomainRegistrar(config)` creates `DomainRegistrar`.

### Parameters:

- **ensInfo** : object - required informations about ENS:
  - **ensAddress** : string - the address of an ENS contract
  - **publicResolverAddress** : string - the address of a public resolver
- **wallet** : ethers.Wallet - instance of ethers Wallet connected to the specific network

**Returns:** `DomainRegistrar` instance

### Example:

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

const ensInfo = {
  ensAddress: '0x112234455c3a32fd11230c42e7bccd4a84e02010',
  publicResolverAddress: '0x4C641FB9BAd9b60EF180c31F56051cE826d21A9A'
};
const provider = new providers.JsonRpcProvider('https://ropsten.infura.io');
const wallet = new Wallet('YOUR_PRIVATE_KEY', provider);

const registrar = new DomainRegistrar(ensInfo, wallet);
```

`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 relay. 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 relay, which propagates them to the network.

### 1.4.1 Creating an SDK

`new UniversalLoginSDK(relayerURL, providerURL, sdkConfig)`

**Parameters:**

- **relayerURL** : string - a URL address of a relayer
- **providerURL** : string - JSON-RPC URL of an Ethereum node
- **sdkConfig** (optional) : object - specific sdk options listed below (each of them is optional):
  - **applicationInfo** : object - detailed info about application: applicationName, type (e.g. laptop, mobile), logo (image source)
  - **paymentOptions** : object - specific transaction options: gasLimit, gasPrice, gasToken
  - **observedTokensAddresses** : array of strings - addresses of tokens used
  - **observedCurrencies**: array with combination of USD, ETH and DAI - shortcuts of currencies used
  - **notice** : string - additional info shown on the top of React components e.g. 'Beta version'
  - **executionFactoryTick** : number - period of time in milliseconds which factory wait with retry if transaction failed
  - **authorizationsObserverTick** : number - period of time in milliseconds which last between authorization check
  - **balanceObserverTick** : number - period of time in milliseconds which last between balance update
  - **priceObserverTick** : number - period of time in milliseconds which last between price update

**Returns:** UniversalLoginSDK instance

**Example:**

```
import UniversalLoginSDK from '@unilogin/sdk';

const sdkConfig = {
  applicationInfo: {
    applicationName: 'Universal Login Application',
    type: 'laptop',
    logo: 'http://logo.universallogin.io'
  },
  paymentOptions: {
    gasPrice: 1500000000,
    gasLimit: 2000000,
    gasToken: '0x0000000000000000000000000000000000000000000000000000000000000000'
  },
  observedTokensAddresses: ['0x0000000000000000000000000000000000000000000000000000000000000000',
  ↪ '0x9f8f72aa9304c8b593d555f12ef6589cc3a579a2'],
  observedCurrencies: ['USD', 'ETH'],
  notice: 'This is beta version',
  executionFactoryTick: 1000,
  authorizationsObserverTick: 1000,
  balanceObserverTick: 1000,
  priceObserverTick: 1000
};

const universalLoginSDK = new UniversalLoginSDK(
  'http://myrelayer.ethworks.io',
  'http://localhost:18545',
```

(continues on next page)

(continued from previous page)

```

    sdkConfig
  );

```

**Default sdkConfig:**

```

sdkConfig = {
  applicationInfo: {
    applicationName: 'Unknown application',
    logo: 'none',
    type: 'unknown'
  },
  paymentOptions: {
    gasPrice: 10000000000,
    gasLimit: 200000,
    gasToken: '0x0000000000000000000000000000000000000000000000000000000000000000'
  },
  observedTokensAddresses: ['0x0000000000000000000000000000000000000000000000000000000000000000'
↪],
  observedCurrencies: ['USD', 'DAI', 'ETH'],
  notice: '',
  executionFactoryTick: 1000,
  authorizationsObserverTick: 3000,
  balanceObserverTick: 3000,
  priceObserverTick: 300000
};

```

**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');

```

## 1.4.2 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 Creating a deployed wallet

### new DeployedWallet(contractAddress, name, privateKey, sdk)

creates the DeployedWallet object

#### Parameters:

- **contractAddress** : string - an address of a contract to remove a request
- **name** : string - a name for deployed wallet
- **privateKey** : string - a private key to sign a request
- **sdk** : object - a UniversalLoginSDK object

**Returns:** DeployedWallet instance

Example:

```
import {DeployedWallet} from '@unilogin/sdk';

const deployedWallet = new DeployedWallet('0x282828282215356332', 'name.
  ↪mylogin.eth', '0x1183823828282356343143', sdk);
```

## 1.4.4 Transaction execution

### execute

#### deployedWallet.execute(message)

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 beneficiary 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

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

Example:

```
const message = {...transferMessage, from: contractAddress, gasToken: ↵  
↳ETHER_NATIVE_TOKEN.address, data: '0x'};  
const {waitToBeSuccess} = await deployedWallet.execute(message);
```

**Execution** contains:

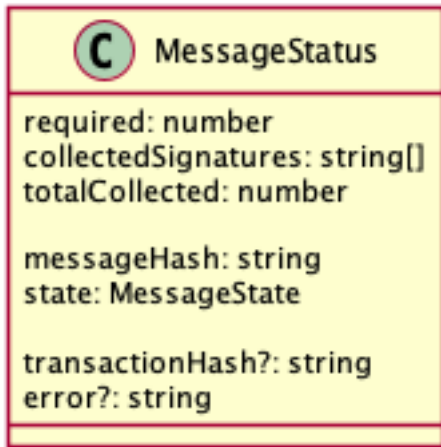
- **messageStatus** - a current status of the sent message ([learn more](#))
- **waitToBeMined** - a function that returns a promise that resolves to `MessageStatus` once the transaction enclosed with `Message` is mined

Example:

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

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`

## 1.4.5 Managing a wallet contract

### addKey

#### deployedWallet.addKey(publicKey, executionOptions)

adds a key to manage a wallet contract.

#### Parameters:

- **publicKey** : string - a public key to manage the contract
- **executionOptions** : object - an optional parameter that includes details of transactions for example `gasLimit` or `gasPrice`

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

**Example:**

```
const executionOptions = {
  gasToken: '0x850437540FE07d02045f88cAe122Bc66B1BdE957',
  gasPrice: 1000000,
  gasLimit: 150000
};
await deployedWallet.addKey(
  '0x96E8B90685AFD981453803f1aE2f05f8Ebc3cfD0',
  executionOptions,
);
```

## addKeys

### deployedWallet.addKey(publicKeys, executionOptions)

adds multiple keys to manage a contract.

**Parameters:**

- **publicKeys** : array of strings - public keys to add
- **executionOptions** : object - an optional parameter that includes details of transactions for example gasLimit or gasPrice

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

**Example:**

```
const publicKeys = [
  '0x96E8B90685AFD981453803f1aE2f05f8Ebc3cfD0',
  '0xb19Ec9bdC6733Bf0c825FCB6E6Da95518DB80D13'
];
const executionOptions = {
  gasToken: '0x850437540FE07d02045f88cAe122Bc66B1BdE957',
  gasPrice: 1000000,
  gasLimit: 150000
};
await deployedWallet.addKeys(
  publicKeys,
  executionOptions,
);
```

## removeKey

### deployedWallet.removeKey(publicKey, executionOptions)

removes a key from a contract.

**Parameters:**

- **publicKey** : string - a public key to remove
- **executionOptions** : object - an optional parameter that includes details of transactions for example gasLimit or gasPrice

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

**Example**



```
const executionOptions = {
  gasToken: '0x9f2990f93694B496F5EAc5822a45f9c642aaDB73',
  gasPrice: 1000000,
  gasLimit: 150000
};
await deployedWallet.removeKey(
  '0xbA03ea3517ddcD75e38a65EDEB4dD4ae17D52A1A',
  executionOptions
);
```

## 1.4.6 Events

### Key added and key removed

#### **sdk.start()**

Starts listening to blockchain events and fetches supported tokens details.

#### **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 Relay

A relay is a RESTful JSON API server written in node.js and express.js that allows interacting with a wallet contract using meta-transactions. The relay 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 relay.

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 using `psql`:

```
psql -c "create database universal_login_relayer_development;"
```

#### 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='popularapp.test'
ENS_DOMAIN_2='my-login.test'
ENS_DOMAIN_3='universal-login.test'
WALLET_MASTER_ADDRESS='0xfb152D3b3bB7330aA52b2504BF5ed1f376B1C189'
FACTORY_ADDRESS='0xE316A2134F6c2BE3eeFdAde5518ce3F685af27E7'
```

## 2. Run relay

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

```
yarn relay:start
```

## CHAPTER 2

---

### Indices and tables

---

- [genindex](#)
- [modindex](#)
- [search](#)