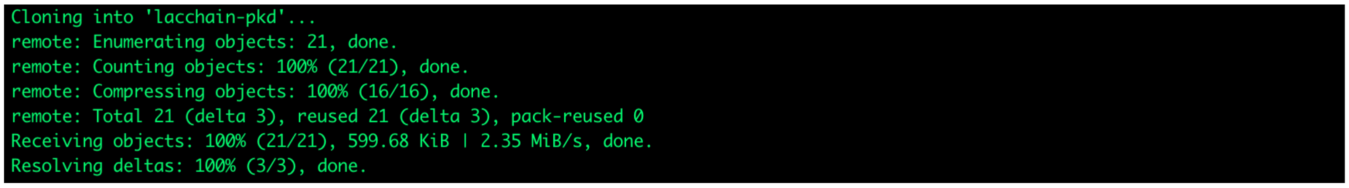
Deploy Public Key Directory

In order to use the Public Key Directory (PKD) and Trusted List (TL), it is necessary to clone the repository that contains the smart contracts and install some additional tools. Below are the steps to follow to have an environment to make the correct deployment.

1. To clone the repository, execute the following command:

```
$ git clone https://github.com/lacchain/lacchain-pkd
```

The output of the command should download the contracts folder from the repository, as shown in the following image:

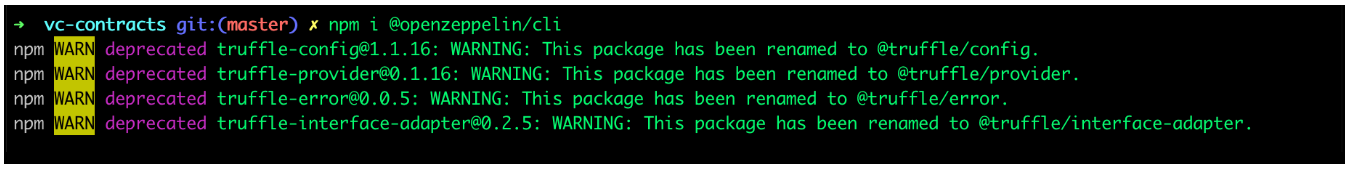


```
Cloning into 'lacchain-pkd'...
remote: Enumerating objects: 21, done.
remote: Counting objects: 100% (21/21), done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 21 (delta 3), reused 21 (delta 3), pack-reused 0
Receiving objects: 100% (21/21), 599.68 KiB | 2.35 MiB/s, done.
Resolving deltas: 100% (3/3), done.
```

2. Once the repository is cloned, we proceed to deploy the smart contracts using the OpenZeppelin Command Line Interface (CLI). To do this, the following command must be executed:

```
$ npm i @openzeppelin/cli
```

The above will install the openzeppelin CLI in the folder where the command is run:



```
➜  vc-contracts git:(master) ✗ npm i @openzeppelin/cli
npm WARN deprecated truffle-config@1.1.16: WARNING: This package has been renamed to @truffle/config.
npm WARN deprecated truffle-provider@0.1.16: WARNING: This package has been renamed to @truffle/provider.
npm WARN deprecated truffle-error@0.0.5: WARNING: This package has been renamed to @truffle/error.
npm WARN deprecated truffle-interface-adapter@0.2.5: WARNING: This package has been renamed to @truffle/interface-adapter.
```

Note: If desired, it is possible to install the tool globally by adding the -g flag to the previous command, which will allow executing the openzeppelin command line from any other project.

3. Once the OpenZeppelin CLI is installed, it is necessary to edit the network configuration to be used for the deployment. Inside the repository we rename the example configuration file called truffle-config.default to truffle-config.js

```
$ mv truffle-config.default truffle-config.js
```

And we edit the truffle-config.js file to include the LACChain network configuration. Consider the following code:

```
const HDWalletProvider = require('@truffle/hdwallet-provider');

module.exports = {
        networks: {
                lacchain: {
                        provider: () => new HDWalletProvider(
                                '862d035b908235f1d0af51713a9e6fc8a1108ac98a77a0be91131d226aa8f4d0',
                                'https://writer.lacchain.net'
                        ),
                        gas: 5000000,
                        gasPrice: 0,
                        network_id: '*'
                }
        },

        mocha: {
                timeout: 100000
        },

        compilers: {
                solc: {
                        version: "0.6.12"
                }
        }
};
```

4. Once the truffle-config.js file has been saved, we proceed to initialize the OpenZeppelin project using the following command:

```
$ npx oz init
```

The command will request a name to the project: which is usually the same name as the repository and the version: which is normally 1.0.0.

```
? Welcome to the OpenZeppelin SDK! Choose a name for your project lacchain-pkd
? Initial project version 1.0.0
Project initialized. Write a new contract in the contracts folder and run 'openzeppelin deploy' to deploy it.
```

5. After initializing the OpenZeppelin project it is possible to deploy the PKD contract with the following command:

```
$ npx oz deploy
```

The above command will invoke the OpenZeppelin CLI that has been installed above the directory. If the tool was installed globally, the npx prefix can be omitted. When deploying the contract, OpenZeppelin will ask us to select the type of deployment: regular, the network: lacchain, and the contract to deploy: PKD, as shown below:

```
? Choose the kind of deployment regular
? Pick a network lacchain
? Pick a contract to deploy PKD
✓ Deployed instance of PKD
0x52663a5679b5126FA23c6f48666BFebd8466c936
```

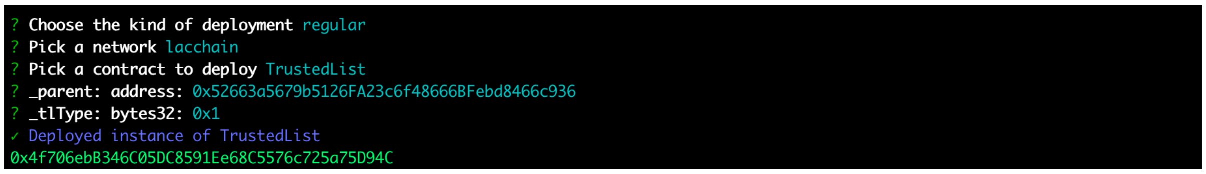
Deploy a Trusted List

Unlike a PKD, a TL can be linked hierarchically by assigning a parent contract which can correspond to another TL or a PKD (usually the root node), therefore the deployment requires specifying that parameter in the constructor from TL:

```
$ npx oz deploy
```

In this case, the following parameters must be specified:

- **network**: lacchain
- **contract**: TrustedList
- **parent**: the address of the parent node of the TL (in this case the address of the PKD)
- **tlType**: the type of Trusted List (see EIDAS framework)



```
? Choose the kind of deployment regular
? Pick a network lacchain
? Pick a contract to deploy TrustedList
? _parent: address: 0x52663a5679b5126FA23c6f48666BFebd8466c936
? _tlType: bytes32: 0x1
✓ Deployed instance of TrustedList
0x4f706ebB346C05DC8591Ee68C5576c725a75D94C
```

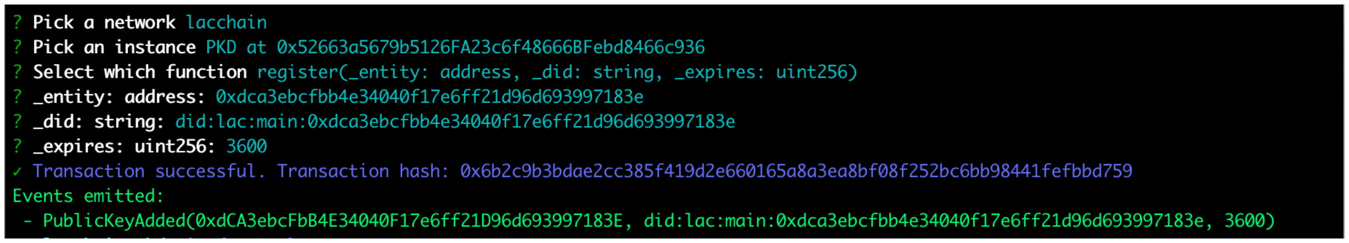
Register Public Key in PKD

Within the PKD contract, the public keys can be registered associating them to a DID and point to the DID address by invoking the register function using the OpenZeppelin CLI tool:

```
$ npx oz send-tx
```

To execute the command, the following parameters are used:

- **network**: lacchain
- **instance**: PKD
- **function**: register
- **address**: the address of the entity to be registered (it can be the ZERO address of Ethereum)
- **did**: the DID associated with the entity (it can be any DID method) where the public keys are
- **expires**: the expiration time of the public key in seconds



```
? Pick a network lacchain
? Pick an instance PKD at 0x52663a5679b5126FA23c6f48666BFebd8466c936
? Select which function register(_entity: address, _did: string, _expires: uint256)
? _entity: address: 0xdca3ebcfbb4e34040f17e6ff21d96d693997183e
? _did: string: did:lac:main:0xdca3ebcfbb4e34040f17e6ff21d96d693997183e
? _expires: uint256: 3600
✓ Transaction successful. Transaction hash: 0x6b2c9b3bdae2cc385f419d2e660165a8a3ea8bf08f252bc6bb98441fefbbd759
Events emitted:
 - PublicKeyAdded(0xdCA3ebcFbB4E34040F17e6ff21D96d693997183E, did:lac:main:0xdca3ebcfbb4e34040f17e6ff21d96d693997183e, 3600)
```

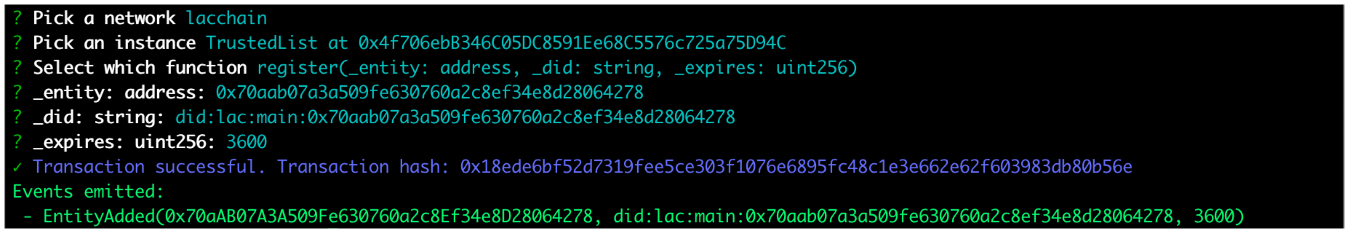
Register an Entity in TL

Within the PKD contract, the public keys can be registered associating them to a DID and point to the DID address by invoking the register function using the OpenZeppelin CLI tool.:

```
$ npx oz send-tx
```

To execute the command, the following parameters are used:

- **network**: lacchain
- **instance**: TrustedList
- **function**: register
- **address**: the address of the entity to be registered (It can be the address of a sub TL)
- **did**: the DID associated with the entity (it can be any DID method) where the public keys of the entity are
- **expires**: The entity's expiration time in seconds



```
? Pick a network lacchain
? Pick an instance TrustedList at 0x4f706ebB346C05DC8591Ee68C5576c725a75D94C
? Select which function register(_entity: address, _did: string, _expires: uint256)
? _entity: address: 0x70aab07a3a509fe630760a2c8ef34e8d28064278
? _did: string: did:lac:main:0x70aab07a3a509fe630760a2c8ef34e8d28064278
? _expires: uint256: 3600
✓ Transaction successful. Transaction hash: 0x18ede6bf52d7319fee5ce303f1076e6895fc48c1e3e662e62f603983db80b56e
Events emitted:
 - EntityAdded(0x70aAB07A3A509Fe630760a2c8Ef34e8D28064278, did:lac:main:0x70aab07a3a509fe630760a2c8ef34e8d28064278, 3600)
```

Verify an Entity in a TL

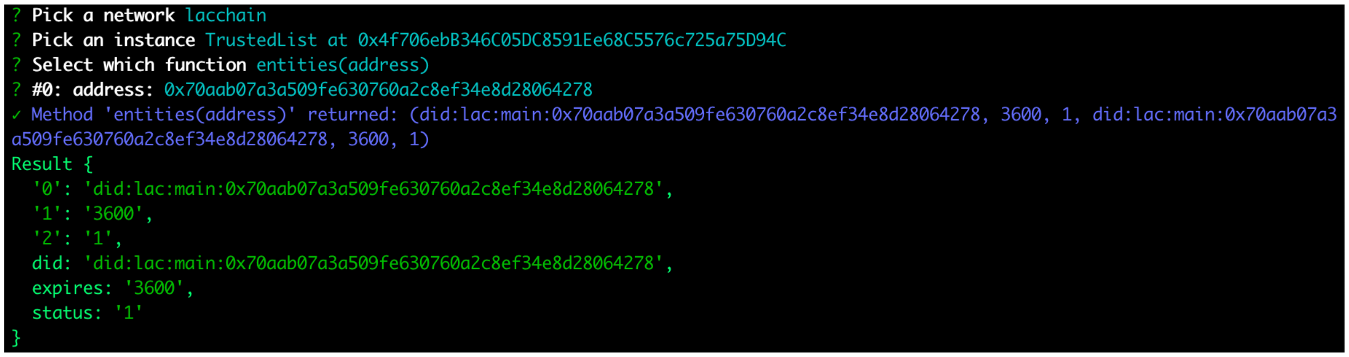
To check if an entity is registered within a Trusted List is very simple, you just have to make a call to the entities (address) method of the Smart Contract, as shown below:

```
$ npx oz call
```

The call to the contract is made with the following parameters:

- **network**: lacchain
- **instance**: TrustedList
- **function**: entities(address)
- **address**: the address of the entity to be verified

If the entity exists and is registered within the TrustedList, the result is as follows:



```
? Pick a network lacchain
? Pick an instance TrustedList at 0x4f706ebB346C05DC8591Ee68C5576c725a75D94C
? Select which function entities(address)
? #0: address: 0x70aab07a3a509fe630760a2c8ef34e8d28064278
✓ Method 'entities(address)' returned: (did:lac:main:0x70aab07a3a509fe630760a2c8ef34e8d28064278, 3600, 1, did:lac:main:0x70aab07a3
a509fe630760a2c8ef34e8d28064278, 3600, 1)
Result {
  '0': 'did:lac:main:0x70aab07a3a509fe630760a2c8ef34e8d28064278',
  '1': '3600',
  '2': '1',
  did: 'did:lac:main:0x70aab07a3a509fe630760a2c8ef34e8d28064278',
  expires: '3600',
  status: '1'
}
```

Verify the Root-of-Trust of an Entity

The Root-of-Trust (RoT) consists of verifying the entity on a TL, but following hierarchically the path until reaching a root node (usually a PKD) which is trusted and guarantees the identity of the entire chain. To verify the RoT it is necessary to execute the previous procedure recursively obtaining the parent node through the following call to the TL contract:

```
$ npx oz call
```

The call to the contract is made with the following parameters:

- **network**: lacchain
- **instance**: TrustedList
- **function**: parent()

Execution of the command will return the address of the parent TL or PKD which is used to repeat the same verification process. However, in this case, the address corresponds to a PKD.

```
? Pick a network lacchain
? Pick an instance TrustedList at 0x4f706ebB346C05DC8591Ee68C5576c725a75D94C
? Select which function parent()
✓ Method 'parent()' returned: 0x52663a5679b5126FA23c6f48666BFebd8466c936
0x52663a5679b5126FA23c6f48666BFebd8466c936
```

Once the address of the PKD is obtained, we can verify if the TL is registered as an entity within the PKD. Remember that the TL address is: 0x4f706ebB346C05DC8591Ee68C5576c725a75D94C. So we will proceed to find that address by calling the isActive function of the PKD contract, as shown below.

The call to the contract is made with the following parameters:

- **network**: lacchain
- **instance**: TrustedList
- **function**: isActive
- **entity**: the entity's address (in this case, the TL's address)

If the entity is registered and has not been revoked, the result will be a boolean value (true).

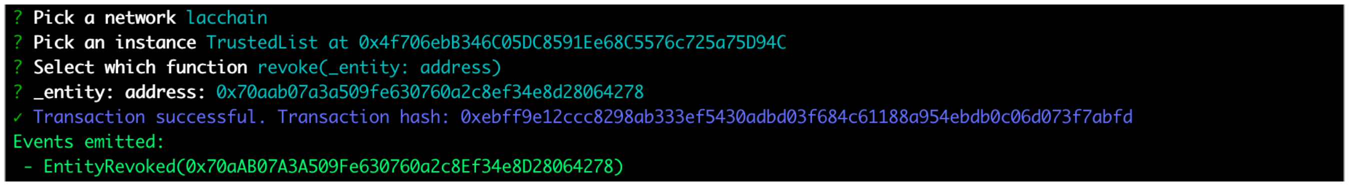
```
? Pick a network lacchain
? Pick an instance PKD at 0x52663a5679b5126FA23c6f48666BFebd8466c936
? Select which function isActive(entity: address)
? entity: address: 0x4f706ebB346C05DC8591Ee68C5576c725a75D94C
✓ Method 'isActive(address)' returned: true
true
```

In this way, although an entity can register as the parent node the address of a valid PKD, the RoT verification process consists of asking the PKD if the entity or TL is really registered and is valid..

Revoke an entity in a TL

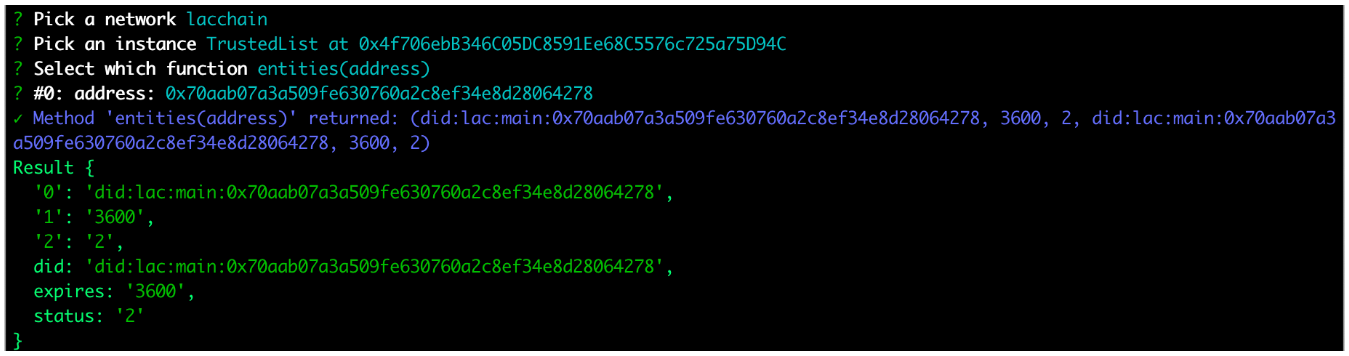
The revocation process of an entity registered in a TL must be done by invoking the revoke function of the Smart Contract associated with the TL from the owner account that deployed the contract..

The invocation of the contract is done with the following parameters:
- **network**: lacchain
- **instance**: TrustedList
- **function**: revoke
- **entity**: the address of the entity (in this case: 0x70aab07a3a509fe630760a2c8ef34e8d28064278)



```
? Pick a network lacchain
? Pick an instance TrustedList at 0x4f706ebB346C05DC8591Ee68C5576c725a75D94C
? Select which function revoke(_entity: address)
? _entity: address: 0x70aab07a3a509fe630760a2c8ef34e8d28064278
✓ Transaction successful. Transaction hash: 0xebff9e12ccc8298ab333ef5430adbd03f684c61188a954ebdb0c06d073f7abfd
Events emitted:
 - EntityRevoked(0x70aAB07A3A509Fe630760a2c8Ef34e8D28064278)
```

Once the entity is revoked, we can recheck its status by calling the entices function of the TL, as can be seen in the following image:



```
? Pick a network lacchain
? Pick an instance TrustedList at 0x4f706ebB346C05DC8591Ee68C5576c725a75D94C
? Select which function entities(address)
? #0: address: 0x70aab07a3a509fe630760a2c8ef34e8d28064278
✓ Method 'entities(address)' returned: (did:lac:main:0x70aab07a3a509fe630760a2c8ef34e8d28064278, 3600, 2, did:lac:main:0x70aab07a3
a509fe630760a2c8ef34e8d28064278, 3600, 2)
Result {
  '0': 'did:lac:main:0x70aab07a3a509fe630760a2c8ef34e8d28064278',
  '1': '3600',
  '2': '2',
  did: 'did:lac:main:0x70aab07a3a509fe630760a2c8ef34e8d28064278',
  expires: '3600',
  status: '2'
}
```

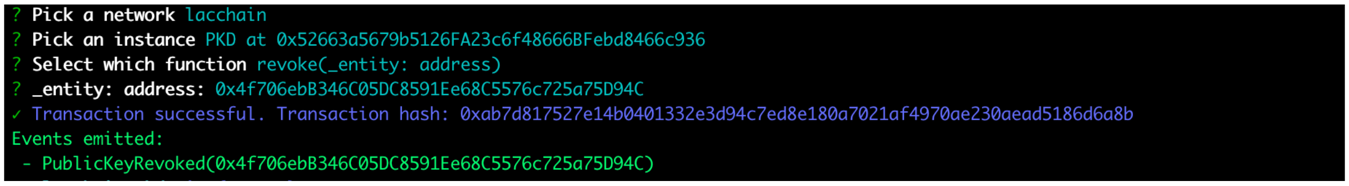
The result shows a status = 2, which indicates that the entity has been revoked. The status list that is managed within a TL is:

- 0: Unregistered
- 1: Registered
- 2: Revoked

Revoke an entity in a TL

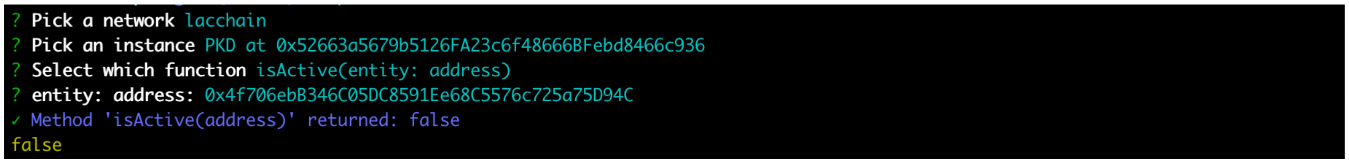
The process of revoking a public key (which can also be associated with a TL) is similar to revoking an entity within a TL.

The invocation of the contract is done with the following parameters:
- **network**: lacchain
- **instance**: PKD
- **function**: revoke
- **entity**: the address of the public key (in this case, that of the TL address 0x4f706ebB346C05DC8591Ee68C5576c725a75D94C)



```
? Pick a network lacchain
? Pick an instance PKD at 0x52663a5679b5126FA23c6f48666BFebd8466c936
? Select which function revoke(_entity: address)
? _entity: address: 0x4f706ebB346C05DC8591Ee68C5576c725a75D94C
✓ Transaction successful. Transaction hash: 0xab7d817527e14b0401332e3d94c7ed8e180a7021af4970ae230aead5186d6a8b
Events emitted:
 - PublicKeyRevoked(0x4f706ebB346C05DC8591Ee68C5576c725a75D94C)
```

Once the public key associated with a TL has been revoked, we can recheck its status by calling the isActive function of the PKD, as shown below.



```
? Pick a network lacchain
? Pick an instance PKD at 0x52663a5679b5126FA23c6f48666BFebd8466c936
? Select which function isActive(entity: address)
? entity: address: 0x4f706ebB346C05DC8591Ee68C5576c725a75D94C
✓ Method 'isActive(address)' returned: false
false
```

As we can see now the verification result is **false**, which indicates that the entity has been revoked.