How to create a blockchain application which offers users a feature to recover the balance of their accounts for the scenario whereby their passphrase has been lost or forgotten.

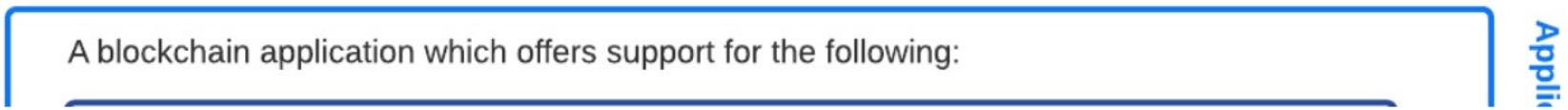
The goal is to build an account recovery tool where a user asks friends to provide access to the funds of a lost account. The user defines a recovery configuration initially by setting a list of friends and other specific parameters for the recovery.

Tip

For the **full code example** please see the SRS app on GitHub.

Table of Contents

SRS application overview



- Social Recovery System (SRS)

Account State

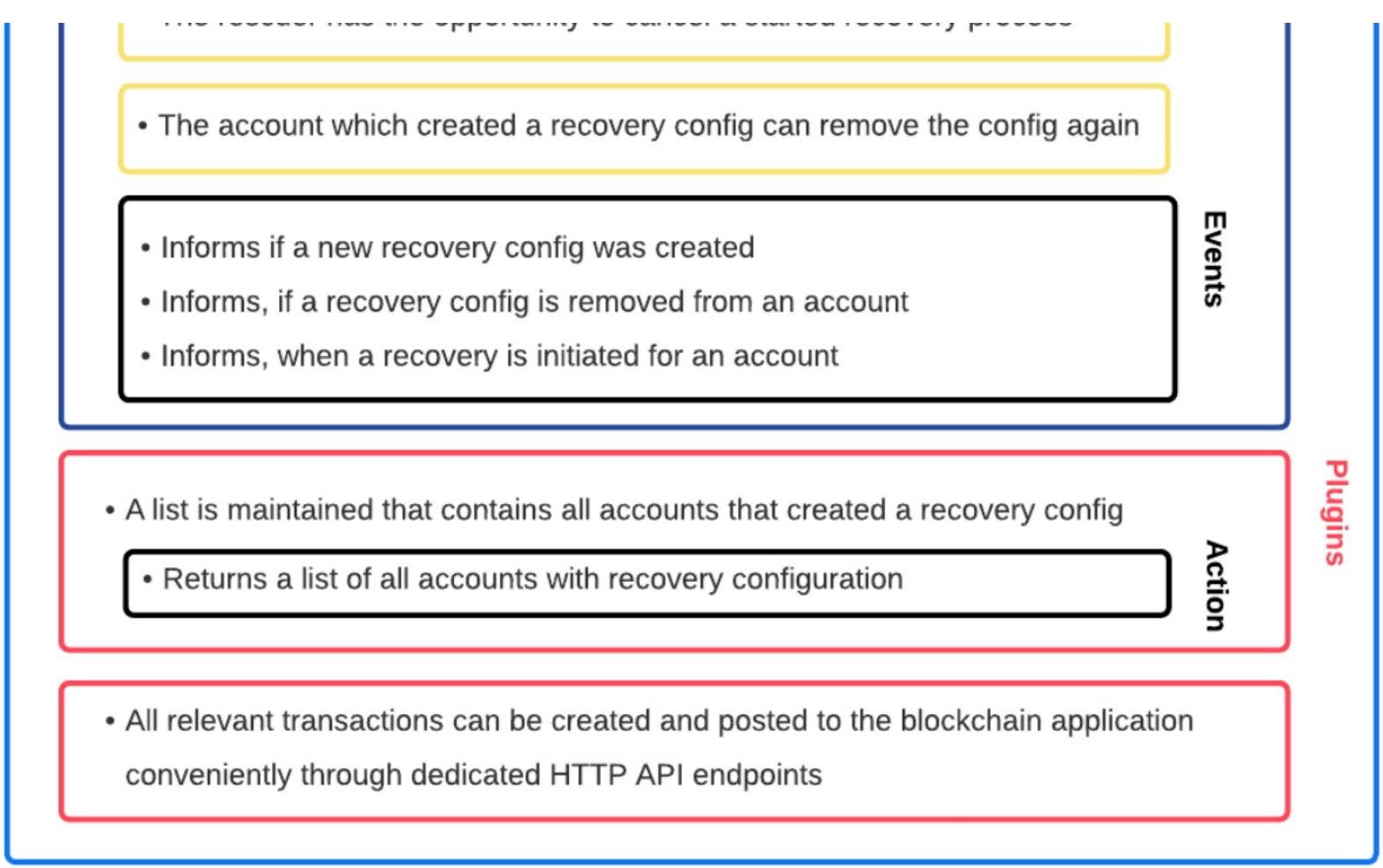
- Configuration:
  - It lets the user define a list of trusted friends for an account rescue
  - It lets the user define a minimal threshold of friends, which need to vouch for the rescuer for a successful recovery
  - It lets the user define a minimum delay of blocks to wait until the recovery can be claimed.
  - It defines an amount of tokens that users needs to deposit for initiating an account recovery.
- Status:
  - Shows if the recovery is initiated
  - Shows the rescuer account address
  - Shows when a recovery was initiated
  - Shows the deposit that was paid to initiate the account recovery
  - Shows the list of vouchers

Transactions

- The account owner can create a recovery configuration
- Anyone can initiate a recovery for an account with a recovery config (the account which initiates recovery is called "rescuer")
- The friends which are listed in the recovery configuration of an account can vouch for a specific rescuer account
- The rescuer can claim a recovery, once a minimum threshold of friends has vouched for the rescuer account and the delay period has passed.
- The rescuer has the opportunity to cancel a started recovery process

Module

cation



As explained in the illustration above, it is necessary to create the following components:

1. The SRS module (on-chain)
    a. 6 new transaction assets for the SRS module
    b. 3 SRS-specific events
2. Plugins (off-chain)
    a. The SRS data plugin
        i. 1 action
    b. The SRS API plugin

In addition to the blockchain application, a **frontend application** will also be implemented which will allow the user to interact with the blockchain application through a UI (User Interface), in the browser.

1. Project setup

Create a new folder which will contain all the files for the SRS app. The Social Recovery System consists of two applications: The blockchain application and the frontend application. First, create the folder for the blockchain application, initialize a new Node.js project and install the Lisk SDK as a dependency.

```
mkdir srs
mkdir srs/blockchain_app
cd srs/blockchain_app
npm init --yes
npm i lisk-sdk
```

Next, create a new file `index.js` and paste the following:

srs/blockchain_app/index.js

```
const { Application, genesisBlockDevnet, configDevnet } = require('lisk-sdk');

const app = Application.defaultApplication(genesisBlockDevnet, configDevnet);

app
	.run()
	.then(() => app.logger.info('App started...'))
	.catch(error => {
		console.error('Faced error in application', error);
		process.exit(1);
	});
```

This code snippet creates a default blockchain application, which is configured for development purposes. Use this app as the basis for the SRS app and extend it with a module and a plugin in the next steps to suit the desired use case.

At this point, it is already possible to start the blockchain application by running `node index.js` in the terminal. To verify the successful start of the application, observe the logs in the terminal for possible errors. If everything is correct, the

application will start to add new blocks to the chain every 10 seconds after the initial start.

2. Transaction assets

Users shall have the ability to perform the following:

0. Create recovery configs for their account.
1. Initiate a recovery for an account which has a recovery config.
2. Vouch for a rescuer account, if certain conditions are met.
3. Claim a recovery, if certain conditions are met.
4. Cancel a recovery, for example, if a lost passphrase from the old account was found again.
5. Remove a recovery config from the account.

2.1. Constants

Inside of the transaction assets, the following constants will be utilized:

Name	Value	Description
BASE_RECOVERY_DEPOSIT	'1000000000'	The base recovery deposit, used to calculate the deposit for an account recovery.
FRIEND_FACTOR_FEE	2	The friend factor fee, used to calculate the deposit for an account recovery.
CREATE_RECOVERY_ASSET_ID	0	Asset ID for the `CreateRecoveryAsset`
INITIATE_RECOVERY_ASSET_ID	1	Asset ID for the `InitiateRecoveryAsset`
VOUCH_RECOVERY_ASSET_ID	2	Asset ID for the `VouchRecoveryAsset`
CLAIM_RECOVERY_ASSET_ID	3	Asset ID for the `ClaimRecoveryAsset`
CLOSE_RECOVERY_ASSET_ID	4	Asset ID for the `CloseRecoveryAsset`
REMOVE_RECOVERY_ASSET_ID	5	Asset ID for the `RemoveRecoveryAsset`

Create a new file `constants.js` and paste the following contents:

```
const BASE_RECOVERY_DEPOSIT = '1000000000';
const FRIEND_FACTOR_FEE = 2;
const CREATE_RECOVERY_ASSET_ID = 0;
const INITIATE_RECOVERY_ASSET_ID = 1;
const VOUCH_RECOVERY_ASSET_ID = 2;
const CLAIM_RECOVERY_ASSET_ID = 3;
const CLOSE_RECOVERY_ASSET_ID = 4;
const REMOVE_RECOVERY_ASSET_ID = 5;

module.exports = {
  BASE_RECOVERY_DEPOSIT,
  FRIEND_FACTOR_FEE,
  CREATE_RECOVERY_ASSET_ID,
  VOUCH_RECOVERY_ASSET_ID,
  CLAIM_RECOVERY_ASSET_ID,
  CLOSE_RECOVERY_ASSET_ID,
  INITIATE_RECOVERY_ASSET_ID,
  REMOVE_RECOVERY_ASSET_ID,
};
```

2.2. Setting up module and transaction folders

Now, create the corresponding transaction assets for the SRS module. These transaction assets each define both, the asset schema for the transaction data, and the logic how this data is applied and stored in the database.

Create a new folder `srs_module/` and inside another new folder `transactions`:

srs/blockchain_app/

```
mkdir srs_module
mkdir srs_module/transactions ①
cd srs_module/transactions/
```

① Create a new folder `transactions/` which will contain the files for the transaction assets.

2.3. createRecovery asset

Create a new file create_recovery.js inside the newly created `transactions/` folder.

Now open the file and paste the code below:

srs/blockchain_app/srs_module/transactions/create_recovery.js

```
const { BaseAsset } = require("lisk-sdk");

// extend base asset to implement the custom asset
class CreateRecoveryAsset extends BaseAsset { ①

}

module.exports = CreateRecoveryAsset; ②
```

① Extend from the base asset to implement a custom asset.

② Export the asset, so it can be imported later into the custom module.

Now all required properties for the transaction asset are defined in sequential order.

2.3.1. Asset ID and name

srs/blockchain_app/srs_module/transactions/create_recovery.js

```
const { BaseAsset } = require("lisk-sdk");
const { CREATE_RECOVERY_ASSET_ID } = require('../constants');

// extend base asset to implement your custom asset
class CreateRecoveryAsset extends BaseAsset {
  // define unique asset name and id
  name = "createRecovery"; ①
  id = CREATE_RECOVERY_ASSET_ID; ②
}

module.exports = CreateRecoveryAsset;
```

① Set the asset name to `"createRecovery"`.

② Set the asset ID to `CREATE_RECOVERY_ASSET_ID` (=0) from the `constants.js` file.

2.3.2. Asset schema

The asset schema describes the required datatypes and the structure of the data in the respective transaction asset.

> **Tip**
>
> For more information how schemas are used in the application, check out the Codec & schema reference.

For creating a recovery configuration, the following information is required:

- `friends`: A list of trusted addresses.
- `recoveryThreshold`: Minimum amount of friends that need to vouch for a rescuer, before the rescuer can claim the recovery.
- `delayPeriod`: The minimum number of blocks required from the height at which the account recovery was initiated to successfully recover the account.

Create a new file `schemas.js`. The schemas which are reused later in different places of the module and assets are stored here.

srs/blockchain_app/srs_module/schemas.js

```
const createRecoverySchema = {
  $id: 'srs/recovery/create',
  type: 'object',
  required: ['friends', 'recoveryThreshold', 'delayPeriod'],
  properties: {
    friends: {
      type: 'array',
      fieldNumber: 1,
      items: {
        dataType: 'bytes',
      },
    },
    recoveryThreshold: {
      dataType: 'uint32',
      fieldNumber: 2,
    },
```

```
    delayPeriod: {
      dataType: 'uint32',
      fieldNumber: 3,
    },
  },
};

module.exports = { createRecoverySchema };
```

Now import the asset schema into `create_recovery.js`.

srs/blockchain_app/srs_module/transactions/create_recovery.js

```
const { BaseAsset } = require('lisk-sdk');
const { CREATE_RECOVERY_ASSET_ID } = require('../constants');
const { createRecoverySchema } = require('../schemas');

class CreateRecoveryAsset extends BaseAsset {
        name = 'createRecovery';
        id = CREATE_RECOVERY_ASSET_ID;
        schema = createRecoverySchema;
}

module.exports = CreateRecoveryAsset;
```

2.3.3. The apply function

The `apply()` function has access to:

- `asset`: the posted transaction asset.
- `stateStore`: The state store is a data structure that maintains a temporary state while processing a block. It is used here to get and set certain data from and to the database.
- `reducerHandler`: This allows the usage of reducer functions of other modules inside the `apply()` function.
- `transaction`: the complete transaction object.

srs/blockchain_app/srs_module/transactions/create_recovery.js

```
const { BaseAsset, transactions } = require('lisk-sdk');
const { createRecoverySchema } = require('../schemas');
const { BASE_RECOVERY_DEPOSIT, FRIEND_FACTOR_FEE, CREATE_RECOVERY_ASSET_ID } = require('../constants');

class CreateRecoveryAsset extends BaseAsset {
        name = 'createRecovery';
        id = CREATE_RECOVERY_ASSET_ID;
        schema = createRecoverySchema;

    async apply({
                asset,
                transaction,
                stateStore,
        }) {
        const sender = await stateStore.account.get(transaction.senderAddress);
        if (sender.srs.config && sender.srs.config.friends.length !== 0) {
            throw Error('Account already has a recovery configuration.')
        }
        const sameAccount = asset.friends.find(f => f === sender.address);
        if (sameAccount) {
            throw new Error('You cannot add yourself to your own friend list.');
        }
        // Add friends to the list
        sender.srs.config.friends = [...asset.friends.sort()];
        // Minimum number of friends required to vouch
        sender.srs.config.recoveryThreshold = asset.recoveryThreshold;
        // Minimum number of blocks after recovery process when account will be recoverable
        sender.srs.config.delayPeriod = asset.delayPeriod;
        // Set the deposit based on number of friends, 10 + friends.length * 2
        const deposit = BigInt(BASE_RECOVERY_DEPOSIT) +
BigInt(transactions.convertLSKToBeddows((sender.srs.config.friends.length *
FRIEND_FACTOR_FEE).toString()));
        sender.srs.config.deposit = deposit;
        // Save the value in stateStore
        await stateStore.account.set(sender.address, sender);
    }
}

module.exports = CreateRecoveryAsset;
```

The other transaction assets are created analog to the `CreateRecoveryAsset`. Examples and descriptions for every asset are displayed below.

2.4. initiateRecovery asset

Create a new file initiate_recovery.js inside the `transactions/` folder. This will contain the logic for the transaction asset which initiates a recovery for a given account address.

The following logic is implemented:

- The asset contains the property `lostAccount`, which is the address of the account that is intended to be recovered.
- You cannot initiate a recovery for your own account.
- The account in `lostAccount` needs to have a recovery configuration for a successful initialization. If no recovery configuration is found, it should throw an error.
- The rescuer needs to pay the deposit defined in the recovery configuration of the lost account. If the rescuer doesn't have an adequate balance, it should throw an error.
- If no error is thrown, the recovery status of the lost account is updated in the following manner:
  - The recovery status is set to `true`.
  - The rescuer is set to the address of the sender of the `initiateRecovery` transaction.
  - The current blockchain height is saved to log when the recovery was initiated.

srs/blockchain_app/srs_module/transactions/initiate_recovery.js

```
const { BaseAsset } = require('lisk-sdk');
const { initiateRecoverySchema } = require('../schemas');
const { INITIATE_RECOVERY_ASSET_ID } = require('../constants');


class InitiateRecoveryAsset extends BaseAsset {
        name = 'initiateRecovery';
        id = INITIATE_RECOVERY_ASSET_ID;
        schema = initiateRecoverySchema;

    async apply({
                asset,
                transaction
```

```
        transaction,
        stateStore,
    reducerHandler,
    }) {
    const rescuer = await stateStore.account.get(transaction.senderAddress);
    const lostAccount = await stateStore.account.get(asset.lostAccount);

    const sameAccount = lostAccount.srs.config.friends.find(f => f === rescuer.address);
    if (sameAccount) {
        throw new Error('You cannot recover your own account.');
    }

    // Check if recovery configuration is present for the lost account or not
    if (lostAccount.srs.config && lostAccount.srs.config.friends.length === 0) {
        throw Error('Lost account has no recovery configuration.')
    }

    const currentHeight = stateStore.chain.lastBlockHeaders[0].height;
    const deposit = lostAccount.srs.config.deposit;

    // Check if rescuer account has enough balance
    const rescuerBalance = await reducerHandler.invoke('token:getBalance', {
        address: rescuer.address,
    });

    if (deposit > rescuerBalance) {
        throw new Error('Rescuer doesnt have enough balance to deposit for recovery process.');
    }
    // Deduct the balance from rescuer and update rescuer account
    await reducerHandler.invoke('token:debit', {
        address: rescuer.address,
        amount: deposit,
      });

    // Update lost account address to active recovery
    lostAccount.srs.status.active = true;
    lostAccount.srs.status.rescuer = rescuer.address;
    lostAccount.srs.status.created = currentHeight;
    lostAccount.srs.status.deposit = deposit;
    lostAccount.srs.status.vouchList = [];
```

```
        // Save lost account values to stateStore
        await stateStore.account.set(lostAccount.address, lostAccount);
    }
}

module.exports = InitiateRecoveryAsset;
```

2.5. vouchRecovery asset

Create a new file vouch_recovery.js inside the `transactions/` folder. This will contain the logic for the transaction asset which allows friends to vouch for an account that initiated a recovery.

The following logic is implemented:

- The asset contains:
  - the property `lostAccount`, which is the address of the account that is intended to be recovered.
  - the property `rescuer`, which is the address of the account that wants to recover the `lostAccount`.
- If the `rescuer` is not the account that has initiated a recovery for the `lostAccount` then an error will be thrown.
- If the account that vouches for the rescuer is not listed under `friends` in the recovery config of the `lostAccount`, then an error will be thrown.
- If the friend has vouched for the rescuer before, then an error will be thrown.
- If all conditions are met, add the friend's address to the `vouchList` in the `lostAccount` details.

srs/blockchain_app/srs_module/transactions/vouch_recovery.js

```
const { BaseAsset } = require('lisk-sdk');
const { vouchRecoverySchema } = require('../schemas');
const { VOUCH_RECOVERY_ASSET_ID } = require('../constants');

class VouchRecoveryAsset extends BaseAsset {
        name = 'vouchRecovery';
        id = VOUCH_RECOVERY_ASSET_ID;
        schema = vouchRecoverySchema;

    async apply({
```

```
                asset,

                transaction,
                stateStore,
        }) {
        const sender = await stateStore.account.get(transaction.senderAddress);
        const lostAccount = await stateStore.account.get(asset.lostAccount);
        const rescuer = await stateStore.account.get(asset.rescuer);

        // Make sure rescuer and lost account match according to config settings
        if (!lostAccount.srs.status.rescuer.equals(rescuer.address)) {
            throw new Error(`Rescuer address is incorrect for the recovery of
${lostAccount.address.toString('hex')}`)
        }

        const found = lostAccount.srs.config.friends.find(f => f.equals(sender.address));
        // Make sure friend is present in the configuration
        if (!found) {
            throw new Error('The sender is not part of friends who can vouch for rescuer for recovery
process.')
        }

        const foundSignature = lostAccount.srs.status.vouchList.find(f => f.equals(sender.address));
        // Make sure the friend has not already voted
        if (foundSignature) {
            throw new Error('The sender has already vouched for the rescuer for recovery process.')
        }

        // Push signature to vouch list
        lostAccount.srs.status.vouchList.push(sender.address);
        await stateStore.account.set(lostAccount.address, lostAccount);
    }
}

module.exports = VouchRecoveryAsset;
```

2.6. claimRecovery asset

Create a new file claim_recovery.js inside the `transactions/` folder. This will contain the logic for the transaction asset which allows the rescuer account to claim the tokens of the lost account after all conditions are met.

The following logic is implemented:

- The asset contains the property `lostAccount`, which is the address of the account that is intended to be recovered.
- If not enough blocks have passed since initialization of the recovery, then an error will be thrown. The amount of blocks that need to have passed since initialization is defined in the `delayPeriod` property of the recovery configuration.
- If not enough friends have vouched for the rescuer account, then an error will be thrown. The amount of friends in the `vouchList` needs to be equal or greater than the `recoveryThreshold` property of the recovery configuration.
- If all conditions are met then perform the following:
  - Transfer the balance from the `lostAccount` to the rescuer account.
  - Reset the recovery configuration and status of the `lostAccount`.

Note

It is always required to leave a minimum account balance inside of the `lostAccount`, so it is not possible to empty it completely.

srs/blockchain_app/srs_module/transactions/claim_recovery.js

```
const { BaseAsset } = require('lisk-sdk');
const { claimRecoverySchema } = require('../schemas');
const { CLAIM_RECOVERY_ASSET_ID } = require('../constants');

class ClaimRecoveryAsset extends BaseAsset {
        name = 'claimRecovery';
        id = CLAIM_RECOVERY_ASSET_ID;
        schema = claimRecoverySchema;

    async apply({
                asset,
                transaction,
                stateStore,
        reducerHandler,
```

```
        }) {
        const rescuer = await stateStore.account.get(transaction.senderAddress);
        const lostAccount = await stateStore.account.get(asset.lostAccount);

        const currentHeight = stateStore.chain.lastBlockHeaders[0].height;
        const delayPeriod = lostAccount.srs.config.delayPeriod;
        const recoveryThreshold = lostAccount.srs.config.recoveryThreshold;
        const deposit = lostAccount.srs.config.deposit;

        // Check if the delay period is passed to claim the recovery
        if ((currentHeight - rescuer.srs.status.created) < delayPeriod) {
            throw new Error(`Cannot claim account before delay period of ${delayPeriod}.`);
        }

        // Check if the recovery has received minimum number of vouch from friends
        if (lostAccount.srs.status.vouchList.length < recoveryThreshold) {
            throw new Error(`Cannot claim account until minimum threshold of
${lostAccount.srs.config.friends.length} friends have vouched.`);
        }

        const minBalance = await reducerHandler.invoke('token:getMinRemainingBalance');
        // Get the account balance of lost account
        const lostAccountBalance = await reducerHandler.invoke('token:getBalance', {
            address: lostAccount.address,
        });

        await reducerHandler.invoke('token:debit', {
            address: lostAccount.address,
            // Get the deposit back from the lost account as well as your own deposit that was locked
            amount: lostAccountBalance - minBalance,
        });

        await reducerHandler.invoke('token:credit', {
            address: rescuer.address,
            // Get the deposit back from the lost account as well as your own deposit that was locked
            amount: BigInt(2) * deposit + lostAccountBalance - minBalance,
        });

        // Reset recovery status
        await stateStore.account.set(rescuer.address, rescuer);
        // Reset all recovery values in the lost account
```

```
        lostAccount.srs.config.friends = [];
        lostAccount.srs.config.delayPeriod = 0;
        lostAccount.srs.config.recoveryThreshold = 0;
        lostAccount.srs.config.deposit = BigInt('0');
        lostAccount.srs.status.active = false;
        lostAccount.srs.status.rescuer = Buffer.from('');
        lostAccount.srs.status.created = 0;
        lostAccount.srs.status.deposit = BigInt('0');
        lostAccount.srs.status.vouchList = [];
        await stateStore.account.set(lostAccount.address, lostAccount);
        }
}

module.exports = ClaimRecoveryAsset;
```

2.7. closeRecovery asset

Create a new file close_recovery.js inside the `transactions/` folder. This will contain the logic for the transaction asset which allows closing an active recovery, for example, in the case whereby the lost credentials were found.

The following logic is implemented:

- The asset contains the property `rescuer`, which is the address of the account that initialized the account recovery.
- An account recovery needs to be initialized for the account to be able to close it.
- An active recovery can only be closed by the original account.
- The deposit which was debited from the rescuer account during the recovery initialization is credited to the original account.
- The recovery status in the original account is reset.

srs/blockchain_app/srs_module/transactions/close_recovery.js

```
const { BaseAsset } = require('lisk-sdk');
const { closeRecoverySchema } = require('../schemas');
const { CLOSE_RECOVERY_ASSET_ID } = require('../constants');

class CloseRecoveryAsset extends BaseAsset {
```

```
        name = 'closeRecovery';
        id = CLOSE_RECOVERY_ASSET_ID;
        schema = closeRecoverySchema;

    async apply({
                asset,
                transaction,
                stateStore,
        reducerHandler,
        }) {
        const lostAccount = await stateStore.account.get(transaction.senderAddress);
        if (!lostAccount.srs.status.active) {
            throw new Error(`No active recovery found for address
${lostAccount.address.toString('hex')}.`);
        }
        if (!lostAccount.srs.status.rescuer.equals(asset.rescuer)) {
            throw new Error(`Incorrect rescuer address`);
        }

        await reducerHandler.invoke('token:credit', {
            address: lostAccount.address,
            amount: lostAccount.srs.config.deposit,
          });

        // Reset recovery status
        lostAccount.srs.status.active = false;
        lostAccount.srs.status.rescuer = Buffer.from('');
        lostAccount.srs.status.created = 0;
        lostAccount.srs.status.deposit = BigInt('0');
        lostAccount.srs.status.vouchList = [];
        await stateStore.account.set(lostAccount.address, lostAccount);
    }
}

module.exports = CloseRecoveryAsset;
```

2.8. removeRecovery asset

Create a new file remove_recovery.js inside the `transactions/` folder. This will contain the logic for the transaction asset which allows the owner of an account to remove a previously created recovery configuration from their account.

The following logic is implemented:

- The asset contains the property `lostAccount`.
- Only accounts with created recovery configurations can remove the configurations again (obviously).
- The recovery status has to be inactive to be able to remove a recovery configuration.
- If the conditions are met, the recovery config and status inside the original account are reset completely.

srs/blockchain_app/srs_module/transactions/remove_recovery.js

```
const {
        BaseAsset
} = require('lisk-sdk');
const { removeRecoverySchema } = require('../schemas');
const { REMOVE_RECOVERY_ASSET_ID } = require('../constants');

class RemoveRecoveryAsset extends BaseAsset {
        name = 'removeRecovery';
        id = REMOVE_RECOVERY_ASSET_ID;
        schema = removeRecoverySchema;

        async apply({
                transaction,
                stateStore,
                reducerHandler,
        }) {
                const lostAccount = await stateStore.account.get(transaction.senderAddress);

                if (lostAccount.srs.config.friends.length === 0) {
                        throw Error('Account does not have a recovery configuration.')
                }

                if (lostAccount.srs.status.active) {
                        throw Error('There is active recovery in process. Please close the recovery to
remove recovery configuration.')
                }
```

```
                // Reset all the default values
                lostAccount.srs.config.friends = [];
                lostAccount.srs.config.recoveryThreshold = 0;
                lostAccount.srs.config.delayPeriod = 0;
                lostAccount.srs.config.deposit = BigInt('0');
                lostAccount.srs.status.rescuer = Buffer.from('');
                lostAccount.srs.status.deposit = BigInt('0');
                lostAccount.srs.status.vouchList = [];
                lostAccount.srs.status.created = 0;
                lostAccount.srs.status.active = false;
                await stateStore.account.set(lostAccount.address, lostAccount);
        }
}

module.exports = RemoveRecoveryAsset;
```

Note

Don't forget to update schemas.js to include the schemas of the new transaction assets.

Now that all the transaction assets are implemented, the SRS module can be created which will contain the newly created assets and some additional logic.

3. The SRS module

Inside the srs_module/ folder, create a new file index.js.

Open `index.js` and create the skeleton which will contain all parts of the SRS module:

3.1. Module ID and name

Set the unique identifier for the SRS module to `srs`, and the module ID to `1026`.

Note

Module ID until 1000 is reserved by Lisk SDK for future default modules.

```
const { BaseModule } = require('lisk-sdk');

// Extend from the base module to implement a custom module
class SRSModule extends BaseModule {
  name = 'srs';
  id = 1026;
}

module.exports = { SRSModule };
```

3.2. The account schema

Open the `schemas.js` file again which was created in section Asset schema, and add the account schema for the SRS module:

srs/blockchain_app/srs_module/schemas.js

```
//...

const SRSAccountSchema = {
    type: 'object',
    required: ['config'],
    properties: {
      config: {
        fieldNumber: 1,
        type: 'object',
        required: ['friends'],
        properties: {
          friends: {
              type: 'array',
              fieldNumber: 1,
              items: {
                  dataType: 'bytes',
              },
```

```
      },
      recoveryThreshold: {
          dataType: 'uint32',
          fieldNumber: 2,
      },
      delayPeriod: {
          dataType: 'uint32',
          fieldNumber: 3,
      },
      deposit: {
        dataType: 'uint64',
        fieldNumber: 4,
      }
    },
    default: {
      friends: [],
      recoveryThreshold: 0,
              delayPeriod: 0,
    },
  },
  status: {
    fieldNumber: 2,
    type: 'object',
    properties: {
      rescuer: {
        dataType: 'bytes',
        fieldNumber: 1,
      },
      created: {
        dataType: 'uint32',
        fieldNumber: 2,
      },
      deposit: {
        dataType: 'uint64',
        fieldNumber: 3,
      },
      vouchList: {
        type: 'array',
        fieldNumber: 4,
        items: {
```

```
            dataType: 'bytes',
          },
        },
        active: {
          dataType: 'boolean',
          fieldNumber: 5,
        },
      },
    }
  },
  default: {
  config: {
    friends: [],
    recoveryThreshold: 0,
    delayPeriod: 0,
  },
  status: {
    active: false,
    vouchList: [],
    created: 0,
    deposit: BigInt(0),
    rescuer: Buffer.from(''),
  },
 },
};

module.exports = { SRSAccountSchema, createRecoverySchema, initiateRecoverySchema, removeRecoverySchema
};
```

Now use the `SRSAccountSchema` inside of the module:

srs/blockchain_app/srs_module/index.js

```
const { BaseModule } = require('lisk-sdk');
const { SRSAccountSchema } = require('./schemas');

// Extend from the base module to implement a custom module
class SRSModule extends BaseModule {
  name = 'srs';
```

```
  id = 1026;

  accountSchema = SRSAccountSchema;

}

module.exports = { SRSModule };
```

3.3. Importing the transaction assets into the module

Now let's import the transactions which were created in section 2: Transaction assets into the module.

Add them to the `tansactionAssets` property as shown in the snippet below:

Tip

Best practice

It is good practice to name the imported transaction assets after their corresponding classname.

In this example: `CreateRecoveryAsset`, `InitiateRecoveryAsset`, `VouchRecoveryAsset`, `ClaimRecoveryAsset`, `CloseRecoveryAsset`, and `RemoveRecoveryAsset`.

srs/blockchain_app/srs_module/index.js

```
const { BaseModule } = require('lisk-sdk');
const CreateRecoveryAsset = require('./assets/create_recovery');
const ClaimRecoveryAsset = require('./assets/claim_recovery');
const InitiateRecoveryAsset = require('./assets/initiate_recovery');
const VouchRecoveryAsset = require('./assets/vouch_recovery');
const CloseRecoveryAsset = require('./assets/close_recovery');
const RemoveRecoveryAsset = require('./assets/remove_recovery');
const { SRSAccountSchema } = require('./schemas');

// Extend from the base module to implement a custom module
class SRSModule extends BaseModule {
  name = 'srs';
  id = 1026;
  accountSchema = SRSAccountSchema;
```

```
  accountSchema = SRSAccountSchema;

  transactionAssets = [
    new CreateRecoveryAsset(),
    new InitiateRecoveryAsset(),
    new VouchRecoveryAsset(),
    new ClaimRecoveryAsset(),
    new CloseRecoveryAsset(),
    new RemoveRecoveryAsset(),
  ];
}

module.exports = { SRSModule };
```

3.4. Events

As described in section SRS application overview, define three different events:

- `configCreated`
- `configRemoved`
- `recoveryInitiated`

srs/blockchain_app/srs_module/index.js

```
const { BaseModule } = require('lisk-sdk');
const CreateRecoveryAsset = require('./assets/create_recovery');
const ClaimRecoveryAsset = require('./assets/claim_recovery');
const InitiateRecoveryAsset = require('./assets/initiate_recovery');
const VouchRecoveryAsset = require('./assets/vouch_recovery');
const CloseRecoveryAsset = require('./assets/close_recovery');
const RemoveRecoveryAsset = require('./assets/remove_recovery');
const { SRSAccountSchema } = require('./schemas');

// Extend from the base module to implement a custom module
class SRSModule extends BaseModule {
  name = 'srs';
  id = 1026;
  accountSchema = SRSAccountSchema;
```

```
  transactionAssets = [
    new CreateRecoveryAsset(),
    new InitiateRecoveryAsset(),
    new VouchRecoveryAsset(),
    new ClaimRecoveryAsset(),
    new CloseRecoveryAsset(),
    new RemoveRecoveryAsset(),
  ];

  events = ['configCreated','configRemoved','recoveryInitiated'];
}

module.exports = { SRSModule };
```

3.5. Lifecycle hooks

Use the life cycle hooks of the module to publish the events that were just created in the Events section.

The hook `afterTransactionApply()` is used here. It is executed each time after a transaction is applied on the blockchain.

The following events are fired:

- `srs:configCreated` when a **create recovery** transaction is applied.
- `srs:configRemoved` when a **claim recovery** or a **remove recovery** transaction is applied.
- `srs:recoveryInitiated` when a **initiateRecovery** transaction is applied.

srs/blockchain_app/srs_module/index.js

```
const { BaseModule, codec } = require('lisk-sdk');
const CreateRecoveryAsset = require('./assets/create_recovery');
const ClaimRecoveryAsset = require('./assets/claim_recovery');
const InitiateRecoveryAsset = require('./assets/initiate_recovery');
const VouchRecoveryAsset = require('./assets/vouch_recovery');
const CloseRecoveryAsset = require('./assets/close_recovery');
const RemoveRecoveryAsset = require('./assets/remove_recovery');
const {
  SRSAccountSchema
```

```
  createRecoverySchema,
  initiateRecoverySchema,
  claimRecoverySchema
} = require('./schemas');
const {
  CREATE_RECOVERY_ASSET_ID,
  CLAIM_RECOVERY_ASSET_ID,
  INITIATE_RECOVERY_ASSET_ID,
  REMOVE_RECOVERY_ASSET_ID
} = require('./constants');

class SRSModule extends BaseModule {
  name = 'srs';
  id = 1026;
  accountSchema = SRSAccountSchema;

  transactionAssets = [
    new CreateRecoveryAsset(),
    new InitiateRecoveryAsset(),
    new VouchRecoveryAsset(),
    new ClaimRecoveryAsset(),
    new CloseRecoveryAsset(),
    new RemoveRecoveryAsset(),
  ];

  events = ['configCreated','configRemoved','recoveryInitiated'];

  async afterTransactionApply({transaction, stateStore, reducerHandler}) {
    if (transaction.moduleID === this.id && transaction.assetID === CREATE_RECOVERY_ASSET_ID) {
      let createRecoveryAsset = codec.decode(
        createRecoverySchema,
        transaction.asset
      );
      const friends = createRecoveryAsset.friends.map(bufferFriend => bufferFriend.toString('hex'));
      this._channel.publish('srs:configCreated', {
         address: transaction._senderAddress.toString('hex'),
         friends: friends,
         recoveryThreshold: createRecoveryAsset.recoveryThreshold,
         delayPeriod: createRecoveryAsset.delayPeriod
      });
```

```
    } else if (transaction.moduleID === this.id && transaction.assetID === REMOVE_RECOVERY_ASSET_ID) {
      this._channel.publish('srs:configRemoved', {
        address: transaction._senderAddress.toString('hex')
      });
    } else if (transaction.moduleID === this.id && transaction.assetID === CLAIM_RECOVERY_ASSET_ID) {
      let claimRecoveryAsset = codec.decode(
        claimRecoverySchema,
        transaction.asset
      );
      this._channel.publish('srs:configRemoved', {
        address: claimRecoveryAsset.lostAccount.toString('hex')
      });
    } else if (transaction.moduleID === this.id && transaction.assetID === INITIATE_RECOVERY_ASSET_ID) {
      const initiateRecoveryAsset = codec.decode(
        initiateRecoverySchema,
        transaction.asset
      );
      this._channel.publish('srs:recoveryInitiated', {
        address: transaction._senderAddress.toString('hex'),
        asset: initiateRecoveryAsset
      });
    }
  };

}

module.exports = { SRSModule };
```

The implementation of the SRS module is now complete.

4. The SRS data plugin

To be able to conveniently acquire a list of all accounts which created a recovery configuration, create a custom plugin.

First, navigate out of the `srs_module` folder, and create a new folder `plugins` for the two plugins we will create for the SRS application. Inside the `plugins` folder, create a new folder `srs_data_plugin`, which will be used to store the files for the new plugin.

srs/blockchain_app/

```
mkdir plugins
mkdir plugins/srs_data_plugin
cd plugins/srs_data_plugin
```

Now create a new file `index.js` inside the newly created `srs_data_plugin/` folder.

Open `index.js` and create the skeleton, which will contain all parts of the SRS data plugin:

srs/blockchain_app/plugins/srs_data_plugin/index.js

```
const { BasePlugin } = require('lisk-sdk');
const pJSON = require('../../package.json');

class SRSDataPlugin extends BasePlugin { ①

  static get alias() { ②
    return 'SRSData';
  }

  static get info() { ③
    return {
      author: pJSON.author,
      version: pJSON.version,
      name: pJSON.name,
    };
  }

  get defaults() {
    return {};
  }

  get events() {
    return [];
  }
}

module.exports = { SRSDataPlugin }; ④
```

① Extend from the base plugin to implement a custom plugin.

② Set the alias for the plugin to `SRSData`.

③ Set the meta information for the plugin. Here, the data is reused from the `package.json` file.

④ Export the plugin, so it can be imported later into the application.

4.1. The `load()` function

The following helper functions and constants are defined:

`getDBInstance()`

Returns a key-value store for the plugin data, which is stored under the path `~/.lisk/srs-app/plugins/data/srs_data_plugin.db`.

`encodedConfigAccountsSchema`

The schema describes how the accounts and their recovery configurations are saved in the database. We define the following schema for the plugin data:

```
{
  "accounts": [
    {
      "address": bytes,
      "friends": array[bytes],
      "recoveryThreshold": number,
      "delayPeriod": number
    }
  ]
}
```

`getConfigAccounts(database)`

A helper function that returns the list of all accounts with recovery configurations from the database. Inside the `getConfigAccounts()` function, use the database key `srs:configAccounts` to get the accounts from the database. If there are no accounts saved in the database yet, then an empty list is returned.

`saveConfigAccounts(database,accounts)`

A helper function that saves the list of all accounts with recovery configurations to the database. Inside the `saveConfigAccounts()` function, encode the accounts list for the database. Use the above defined `encodedConfigAccountsSchema` for this purpose.

srs/blockchain_app/plugins/srs_data_plugin/index.js

```
const { BasePlugin, db, codec } = require('lisk-sdk');
const pJSON = require('../../package.json');
const fs_extra = require("fs-extra");
const os = require("os");
const path = require("path");

const DB_KEY_CONFIGACCOUNTS = "srs:configAccounts";

const getDBInstance = async (dataPath = '~/.lisk/srs-app/', dbName = 'srs_data_plugin.db') => {
  const dirPath = path.join(dataPath.replace('~', os.homedir()), 'plugins/data', dbName);
  await fs_extra.ensureDir(dirPath);
  return new db.KVStore(dirPath);
};

const encodedConfigAccountsSchema = {
  $id: 'srs:configAccounts',
  type: 'object',
  required: ['accounts'],
  properties: {
    accounts: {
      type: 'array',
      fieldNumber: 1,
      items: {
        type: 'object',
        properties: {
          address: {
            dataType: 'bytes',
            fieldNumber: 1,
          },
          friends: {
            type: 'array',
            fieldNumber: 2,
            items: {
              dataType: 'bytes',
            }
          },
          recoveryThreshold: {
            dataType: 'uint32',
```

```
            fieldNumber: 3

          },
          delayPeriod: {
            dataType: 'uint32',
            fieldNumber: 4
          }
        }
      },
    },
  },
};

const getConfigAccounts = async (database) => {
  try {
    const encodedConfigAccounts = await database.get(DB_KEY_CONFIGACCOUNTS);
    const { accounts } = codec.decode(encodedConfigAccountsSchema, encodedConfigAccounts);
    return accounts;
  }
  catch (error) {
    return [];
  }
};

const saveConfigAccounts = async (database, accounts) => {
  const encodedConfigs = codec.encode(encodedConfigAccountsSchema, { accounts });

  await database.put(DB_KEY_CONFIGACCOUNTS, encodedConfigs);
};

class SRSDataPlugin extends BasePlugin {
  _accountsWithConfig = undefined;
  _db = undefined;

  static get alias() {
    return 'SRSData';
  }

  static get info() {
    return {
      author: pJSON.author
```

```
    version: pJSON.version,
    name: pJSON.name,
  };
}

get defaults() {
  return {};
}

get events() {
  return [];
}

async load(channel) {
  this._db = await getDBInstance(); ①
  this._accountsWithConfig =  await getConfigAccounts(this._db); ②
  channel.subscribe('srs:createdConfig', async (info) => { ③

    let duplicate = false;
    for (let i = 0; i < this._accountsWithConfig.length; i++) {
      if (this._accountsWithConfig[i].address.toString('hex') === info.address) { ④
        duplicate = true;
        return;
      }
    }
    if (!duplicate){
      info.address = Buffer.from(info.address, 'hex'); ⑤
      info.friends = info.friends.map(friend => Buffer.from(friend, 'hex'));
      this._accountsWithConfig.push(info);
    }
    await saveConfigAccounts(this._db, this._accountsWithConfig); ⑥
  });
  channel.subscribe('srs:removedConfig', async (info) => { ⑦
    for (let i = 0; i < this._accountsWithConfig.length; i++) {
      if (this._accountsWithConfig[i].address.toString('hex') === info.address) { ⑧
        this._accountsWithConfig.splice(i, 1);
        return;
      }
    }
    await saveConfigAccounts(this._db, this._accountsWithConfig); ⑨
```

```
    });
  }

  async unload() {
  }
}

module.exports = { SRSDataPlugin };
```

① The database instance for the SRS data plugin is stored in the variable `this._db`.

② The accounts with recovery config are retrieved from the database by calling the above defined function `getConfigAccounts()` and then stored in the variable `this._accountsWithConfig`.

③ Next subscribe to the event `srs:createdConfig`, which was previously implemented in the section The SRS module.

④ If a new config was created, check if `this._accountsWithConfig` already contains this account.

⑤ If it doesn't contain the account yet, prepare the account addresses for the database by converting them into Buffers. Then push the data into the array `this._accountsWithConfig`.

⑥ The above defined function `saveConfigAccounts()` is called to save the updated accounts list in the database.

⑦ Now listen to a second event `srs:removedConfig`, which was previously implemented in the section The SRS module.

⑧ If the event `srs:removedConfig` is received, check if the respective account is part of the list `this._accountsWithConfig`. If it is found, the account will be removed from the list.

⑨ The above defined function `saveConfigAccounts()` is called to save the updated accounts list in the database.

4.2. Create an action to get all accounts with recovery configs

To make the plugin data available to the public, create a new action that returns a list of all accounts with recovery configuration.

Add the following code snippet to the existing properties in the `SRSDataPlugin` class:

```
get actions() {
  return {
    getAllRecoveryConfigs: () => { ①
      let stringAccounts = this._accountsWithConfig.map((account) => { ②
        account.address = account.address.toString('hex');
        account.friends = account.friends.map(friend => friend.toString('hex'));
        return account;
```

```
      });
      return stringAccounts; ③
    },
  };
}
```

① We give the action the name `getAllRecoveryConfigs`. The action can be invoked by external services by referring to it as shown here: `SRSDataPlugin:getAllRecoveryConfigs`.

② The account addresses are converted from Buffer to String format for all accounts in `this._accountsWithConfig`.

③ The list of all accounts with recovery configs with addresses in hex string format is returned.

The SRS data plugin is now complete. This will allow the frontend application to receive the list of all accounts with a recovery configuration by connecting to the blockchain application via WebSockets, and invoking the respective action.

5. The SRS API plugin

The SRS API plugin provides HTTP API endpoints to create and post the different transactions of the SRS module, and also for the token transfer transaction.

> **Note**
>
> This could be realized alternatively without an HTTP API, by using a WebSocket connection to the blockchain application, as realized in the SRS data plugin. Instead of providing new HTTP API endpoints, the same could be achieved by adding multiple new actions to the plugin, that can be invoked later by the frontend application.
>
> As an exercise, you could optionally try to implement all the HTTP endpoints of the SRS API plugin as plugin actions.

First, navigate out of the `srs_data_plugin` folder, and create a new folder which will be used to store the files for the SRS API plugin.

srs/blockchain_app/plugins/

```
mkdir srs_api_plugin
cd srs_api_plugin
```

Then create a new file `index.js` inside the newly created `srs_api_plugin/` folder.

Open `index.js` and paste the following snippet:

srs/blockchain_app/plugins/srs_api_plugin/index.js

```
const { BasePlugin } = require('lisk-sdk');
const pJSON = require('../../package.json');

// 1.plugin can be a daemon/HTTP/Websocket service for off-chain processing
class SRSAPIPlugin extends BasePlugin {
  _server = undefined;
  _app = undefined;
  _channel = undefined;
  _db = undefined;
  _nodeInfo = undefined;

  static get alias() {
    return 'SRSHttpApi';
  }

  static get info() {
    return {
      author: pJSON.author,
      version: pJSON.version,
      name: pJSON.name,
    };
  }

  get defaults() {
    return {};
  }

  get events() {
    return [];
  }

  get actions() {
    return {};
  }
```

```
}


module.exports = { SRSAPIPlugin };
```

5.1. Controllers

Next we will define multiple controllers for creating and posting the different transaction objects.

Create a new folder to store the different controllers for the API endpoints.

srs/blockchain_app/plugins/srs_api_plugin/

```
mkdir controllers
cd controllers
```

Inside the folder, it is necessary to define the following controllers:

- transferToken: Creates a transfer transaction and sends it to the blockchain application.
- createRecoveryConfigTrs: Creates a createRecovery transaction and sends it to the blockchain application.
- initiateRecovery: Creates a initiateRecovery transaction and sends it to the blockchain application.
- vouchRecovery: Creates a vouchRecovery transaction and sends it to the blockchain application.
- claimRecovery: Creates a claimRecovery transaction and sends it to the blockchain application.
- closeRecovery: Creates a closeRecovery transaction and sends it to the blockchain application.
- removeRecovery: Creates a removeRecovery transaction and sends it to the blockchain application.

Only the `createRecoveryConfigTrs` controller is explained here in detail, as the implementation of the different controllers is very similar.

Feel free to copy and paste the files above from GitHub into your own project.

srs/blockchain_app/plugins/srs_api_plugin/controllers

```
const { transactions, cryptography } = require('@liskhq/lisk-client');
const { createRecoverySchema } = require('../schemas');
const { SRS_CREATE_ASSET_ID, SRS_MODULE_ID, DEFAULT_FEE } = require('../constants');
```

```
const createRecoveryConfigTrs = (
    codec,
    channel,
    nodeInfo,
) => async (req, res) => {
    try {
        const { passphrase, friends, delayPeriod, recoveryThreshold, fee } = req.body;
        const asset = { ①
            friends: friends.map(f => Buffer.from(f, 'hex')),
            delayPeriod: +delayPeriod,
            recoveryThreshold: +recoveryThreshold,
        };

        const { publicKey } = cryptography.getPrivateAndPublicKeyFromPassphrase(
            passphrase
        );
        const address = cryptography.getAddressFromPassphrase(passphrase);
        const account = await channel.invoke('app:getAccount', {
            address,
        });
        const { sequence: { nonce } } = codec.decodeAccount(account);

        const { id, ...tx } = transactions.signTransaction( ②
            createRecoverySchema,
            {
                moduleID: SRS_MODULE_ID,
                assetID: SRS_CREATE_ASSET_ID,
                nonce: BigInt(nonce),
                fee: fee || DEFAULT_FEE,
                senderPublicKey: publicKey,
                asset,
            },
            Buffer.from(nodeInfo.networkIdentifier, 'hex'),
            passphrase,
        );

        const encodedTransaction = codec.encodeTransaction(tx); ③
        const result = await channel.invoke('app:postTransaction', { ④
            transaction: encodedTransaction
```

```
        });

        res.status(200).json({ data: result, meta: {} }); ④
      } catch (err) {
        res.status(409).json({
          errors: [{ message: err.message }],
        });
      }
};

module.exports = {
    createRecoveryConfigTrs,
};
```

① Create the transaction asset for the `createRecovery` transaction, based on the request data.

② Create and sign the transaction object by utilizing the `signTransaction()` method of the `transactions` library.

③ Prepare the transaction by encoding it with the `encodeTransaction()` method of the `codec` library.

④ Sets the HTTP status for the API response.

5.2. The `load()` and `unload()` functions

Go back to the `index.js` file and define what logic shall be executed by the plugin. Now that all of the controllers are prepared, they can be reused in the `SRSAPIPlugin` to provide the corresponding API endpoints for each controller.

Add the new function `load()` into the `SRSAPIPlugin` with the following contents:

srs/blockchain_app/plugins/srs_api_plugin/index.js

```
const express = require('express');
const cors = require('cors');
const { BasePlugin } = require('lisk-sdk');
const pJSON = require('../../package.json');
const controllers = require('./controllers');

// 1.plugin can be a daemon/HTTP/Websocket service for off-chain processing
class SRSAPIPlugin extends BasePlugin {
  _server = undefined;
  _app = undefined;
```

```
  _channel = undefined;
  _db = undefined;
  _nodeInfo = undefined;

  static get alias() {
    return 'SRSHttpApi';
  }

  static get info() {
    return {
      author: pJSON.author,
      version: pJSON.version,
      name: pJSON.name,
    };
  }

  get defaults() {
    return {};
  }

  get events() {
    return [];
  }

  get actions() {
    return {};
  }

  async load(channel) {
    this._app = express(); ①
    this._channel = channel;
    this._nodeInfo = await this._channel.invoke('app:getNodeInfo');

    this._app.use(cors({ origin: '*', methods: ['GET', 'POST', 'PUT'] })); ②
    this._app.use(express.json()); ③

    this._app.post('/api/token/transfer', controllers.transferToken(this.codec, this._channel,
this._nodeInfo)); ④
    this._app.post('/api/recovery/create', controllers.createRecoveryConfigTrs(this.codec, this._channel,
this._nodeInfo));
    this._app.post('/api/recovery/initiate', controllers.initiateRecovery(this.codec, this._channel,
```

```
this._nodeInfo));
    this._app.post('/api/recovery/vouch', controllers.vouchRecovery(this.codec, this._channel,
this._nodeInfo));
    this._app.post('/api/recovery/claim', controllers.claimRecovery(this.codec, this._channel,
this._nodeInfo));
    this._app.post('/api/recovery/close', controllers.closeRecovery(this.codec, this._channel,
this._nodeInfo));
    this._app.post('/api/recovery/remove', controllers.removeRecovery(this.codec, this._channel,
this._nodeInfo));

    this._server = this._app.listen(8080, '0.0.0.0'); ⑤
  }

  async unload() { ⑥
    // close http server
    await new Promise((resolve, reject) => {
      this._server.close((err) => {
        if (err) {
          reject(err);
          return;
        }
        resolve();
      });
    });
  }
}

module.exports = { SRSAPIPlugin };
```

① Start the Express server and save it under the variable `this._app`.

② Enable cross-origin resource sharing.

③ Recognize the incoming request object as a JSON Object.

④ Define 7 different endpoints:

- POST `/api/token/transfer`
- POST `/api/recovery/create`
- POST `/api/recovery/initiate`
- POST `/api/recovery/vouch`
- POST `/api/recovery/claim`
- POST `/api/recovery/close`
- POST `/api/recovery/remove`

⑤ The API listens on port `8080` at localhost.

⑥ This will close the Express server when the plugin is unloaded again, for example on application shutdown.

The SRS data plugin is now complete. It will allow the frontend application to conveniently create and post the different transaction types to the blockchain application via dedicated HTTP API endpoints.

6. Registering module & plugins

Now that the SRS module and the two plugins have been implemented, it is necessary to inform the blockchain application about them.

This is performed by registering them with the blockchain application as shown below.

Open the `srs/blockchain_app/index.js` file again and paste the following code:

srs/blockchain_app/index.js

```
// 1.Import lisk sdk to create the blockchain application
const {
        Application,
        configDevnet,
        genesisBlockDevnet,
        HTTPAPIPlugin,
        utils,
} = require('lisk-sdk');
// 2.Import SRS module & plugins
const { SRSModule } = require('./srs_module');
```

```
const { SRSAPIPlugin } = require('./srs_api_plugin/');
const { SRSDataPlugin } = require('./srs_data_plugin/');

// 3.Update the genesis block accounts to include SRS module attributes
genesisBlockDevnet.header.timestamp = 1605699440;
genesisBlockDevnet.header.asset.accounts = genesisBlockDevnet.header.asset.accounts.map(
        (account) =>
                utils.objects.mergeDeep({}, account, {
                        srs: {
                                config: {
                                        friends: [],
                                        recoveryThreshold: 0,
                                        delayPeriod: 0,
                                },
                                status: {
                                        active: false,
                                        vouchList: [],
                                        created: 0,
                                        deposit: BigInt(0),
                                        rescuer: Buffer.from(''),
                                },
                        },
                }),
);

// 4.Update application config to include unique label
// and communityIdentifier to mitigate transaction replay
const appConfig = utils.objects.mergeDeep({}, configDevnet, {
        label: 'srs-app',
        genesisConfig: { communityIdentifier: 'SRS' }, //In order to have a unique networkIdentifier
        logger: {
                consoleLogLevel: 'info',
        },
    rpc: {
        enable: true,
        mode: 'ws',
        port: 8888,
    },
});
```

```
// 5.Initialize the application with genesis block and application config
const app = Application.defaultApplication(genesisBlockDevnet, appConfig);

// 6.Register custom SRS Module and Plugins
app.registerModule(SRSModule);
app.registerPlugin(HTTPAPIPlugin);
app.registerPlugin(SRSAPIPlugin);
app.registerPlugin(SRSDataPlugin);

// 7.Run the application
app
        .run()
        .then(() => console.info('SRS Blockchain running....'))
        .catch(console.error);
```

Save and close `index.js`.

Now when the application is started again by running `node index.js`, the blockchain application will load the newly created SRS module and the plugins, and the new features will become available to the blockchain application.

In the next step, we will build a simple React frontend, which allows us to interact with the blockchain application through the browser.

7. Frontend application

The final part of the social recovery system is the frontend application.

Note

The development of the frontend application is completely flexible, and you can use any technology stack that you feel comfortable with.

In this example, we use React to build the client application.

This tutorial predominantly covers and explains how to build with the Lisk SDK, therefore other parts of the frontend app are not explained in great detail. More information about how to build a React application can be found in the official React documentation.

For convenience, clone the development branch from the `lisk-sdk-examples` GitHub repository and use the prepared SRS `frontend_app` from the Lisk SDK examples.

srs/

```
git clone https://github.com/LiskHQ/lisk-sdk-examples.git
mv lisk-sdk-examples/tutorials/social-recovery/frontend_app frontend_app
rm -r ./lisk-sdk-examples
cd frontend_app
npm i
```

At this point it is now possible to try out the frontend and verify that the SRS blockchain application works as expected:

First open a second terminal window, navigate to `srs/blockchain_app` and start the blockchain application with `node index.js`, if it is not already running.

In the first terminal window, start the frontend application as shown below:

srs/frontend_app

```
npm start
```

This should open the React app in the browser under http://localhost:3000/.

Tip

To reset the database of the application, remove the `./lisk/` folder, which can normally be found in the home directory of the user:

```
rm -r ~/.lisk/srs-app
```

7.1. Frontend walkabout

Before we explore the code of the frontend app, lets first take a tour through the frontend in the browser, to see how it all works together.

The following scenario is performed:

1. Five different accounts are created and prepared with an initial amount of tokens in their balance. The accounts have the following roles:
    a. Original account
    b. Rescuer
    c. Friend 1
    d. Friend 2
    e. Friend 3
2. The original account creates a recovery configuration.
3. We assume the original account loses their credentials. The owner of the original account uses a new account to rescue the funds in the original account.
4. The rescuer account initiates a recovery process.
5. We assume the owner of the original account informs their friends about the lost credentials and then informs them about the new account address of the rescuer account.
6. The friends which are listed in the recovery configuration of the original account vouch for the rescuer account.
7. The rescuer account waits until the number of friends vouching reaches a minimum threshold and the delay period is over. The rescuer can then claim the recovery.

7.1.1. Prepare example accounts

In the application example we use the following account credentials:

Example account credentials

```
======================
Original Account
----------------------
```

```
passphrase:
peanut hundred pen hawk invite exclude brain chunk gadget wait wrong ready
address:
lskdxc4ta5j43jp9ro3f8zqbxta9fn6jwzjucw7yt
address (hex string):
d04699e57c4a3846c988f3c15306796f8eae5c1c
========================
Rescuer
------------------------
passphrase:
endless focus guilt bronze hold economy bulk parent soon tower cement venue
address:
lsktrqfj84n34tn97vraaq2ztmrgwgwakmqyskqw4
address (hex string):
9cabee3d27426676b852ce6b804cb2fdff7cd0b5
========================
Friend 1
------------------------
passphrase:
mushroom edit regular pencil ten casino wine north vague bachelor swim piece
address:
lsk32gnhxjs887bqmgoz6y6ozh6c4c6ztpz7wjfa9
address (hex string):
463e7e879b7bdc6a97ec02a2a603aa1a46a04c80
========================
Friend 2
------------------------
passphrase:
thought talk cherry write armed valve salute fabric auction maid join rebuild
address:
lskb6bufqcbrwvgkzuu5wqu6wnruz7awvhxwfkonb
address (hex string):
328d0f546695c5fa02105deb055cf2801d9b8ba1
========================
Friend 3
------------------------
passphrase:
exist night more net diesel exact will purse orbit vacuum birth wide
address:
lskomdmvwhb9r3sgj3ryp4fsqnzfn8c8twzkecugt
```

```
address (hex string):
6174515fa66c91bff1128913edd4e0f1de37cee0
```

To be able to execute all desired tasks with the different accounts, send some tokens to them from the Devnet genesis account.

In this example, we use the genesis account as the original account, so it is only necessary to send tokens to the rescuer account and to the accounts belonging to the 3 friends.

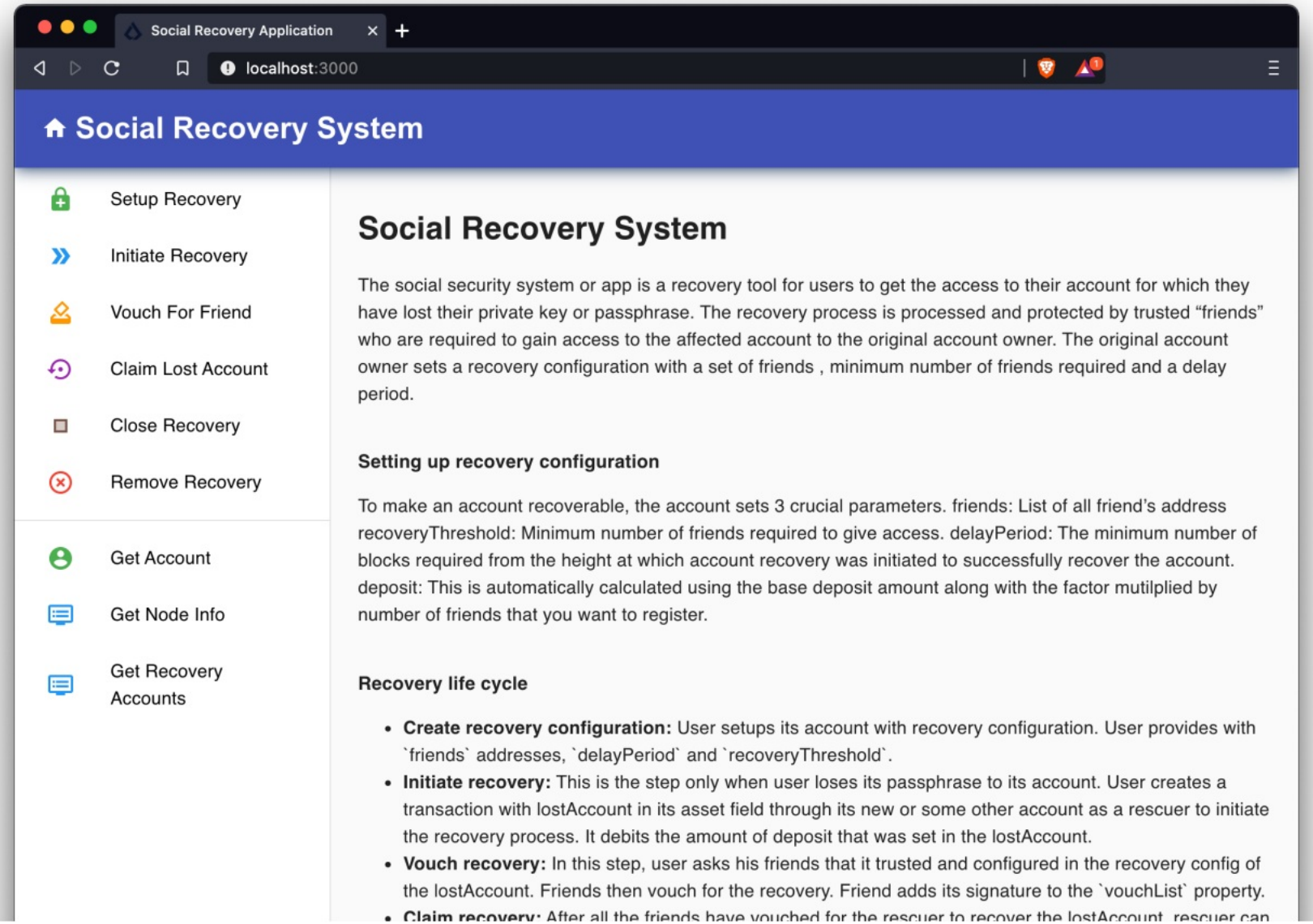
Send tokens to the example accounts.

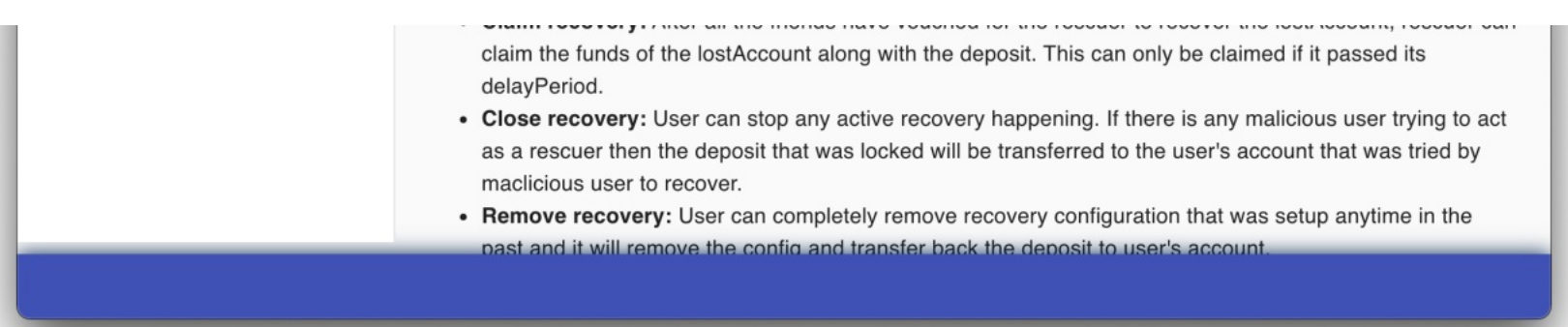
```
# Send tokens to the rescuer account
curl -X POST -H "Content-Type: application/json" \
-d
'{"amount":"108489300000000","recipientAddress":"9cabee3d27426676b852ce6b804cb2fdff7cd0b5","data":"transf
er to a friend","passphrase":"peanut hundred pen hawk invite exclude brain chunk gadget wait wrong
ready"}' \
http://localhost:8080/api/token/transfer

# Send tokens to the friend 1 account
curl -X POST -H "Content-Type: application/json" \
-d
'{"amount":"10000000000","recipientAddress":"463e7e879b7bdc6a97ec02a2a603aa1a46a04c80","data":"transfer
to a friend","passphrase":"peanut hundred pen hawk invite exclude brain chunk gadget wait wrong ready"}'
\
http://localhost:8080/api/token/transfer

# Send tokens to the friend 2 account
curl -X POST -H "Content-Type: application/json" \
-d
'{"amount":"10000000000","recipientAddress":"328d0f546695c5fa02105deb055cf2801d9b8ba1","data":"transfer
to a friend","passphrase":"peanut hundred pen hawk invite exclude brain chunk gadget wait wrong ready"}'
\
http://localhost:8080/api/token/transfer

# Send tokens to the friend 3 account
curl -X POST -H "Content-Type: application/json" \
-d
```

```
'{"amount":"10000000000","recipientAddress":"6174515fa66c91bff1128913edd4e0f1de37cee0","data":"transfer
to a friend","passphrase":"peanut hundred pen hawk invite exclude brain chunk gadget wait wrong ready"}'
\
http://localhost:8080/api/token/transfer
```

7.1.2. The home page

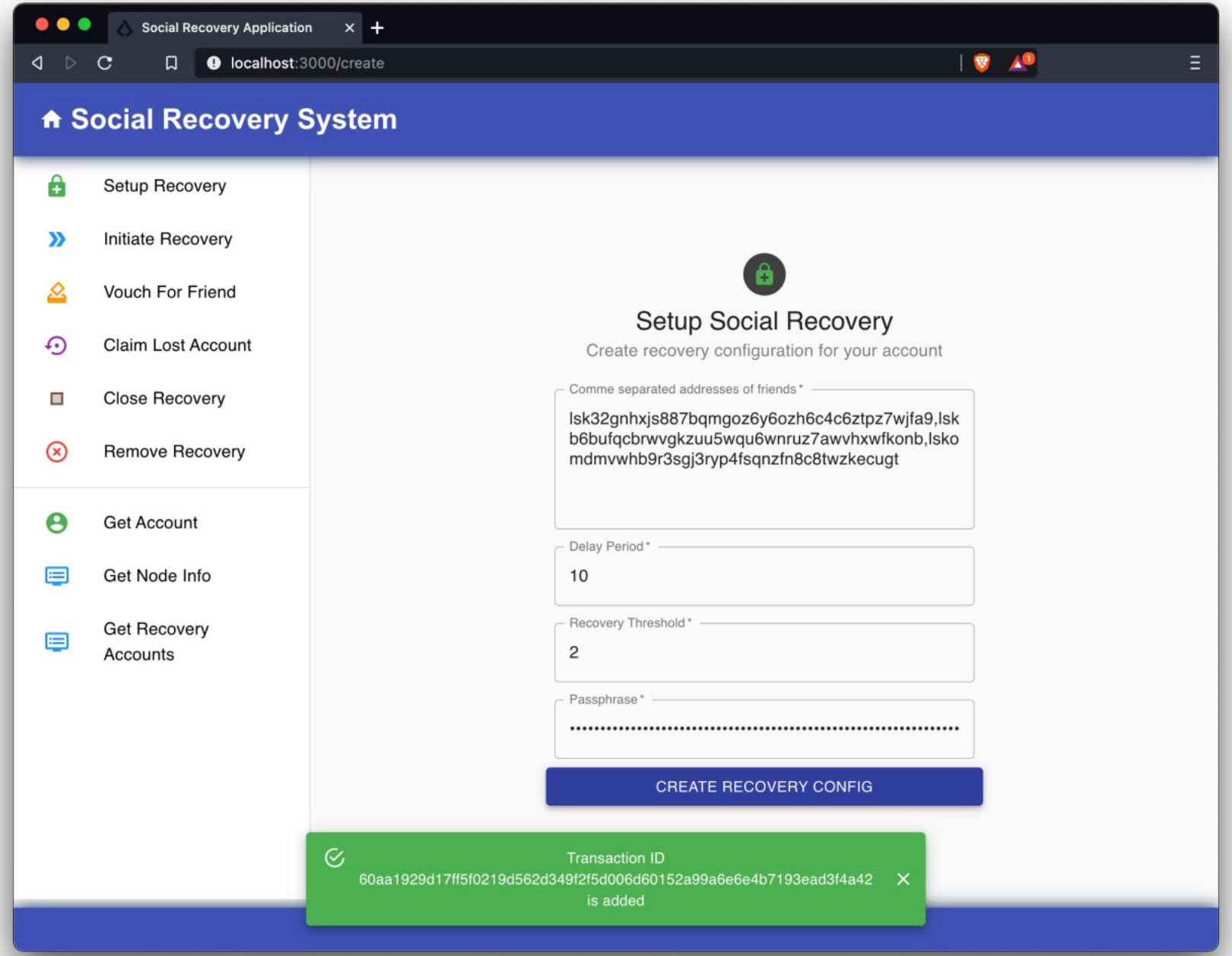
The home page is the landing page that is seen when opening the frontend app under http://localhost:3000/ in the browser.





7.1.3. Creating a recovery config

Click on `Setup Recovery` to create a recovery configuration. For convenience, the fields in the form are already pre-filled with the correct data, so you can simply press the button `CREAT RECOVERY CONFIG`.

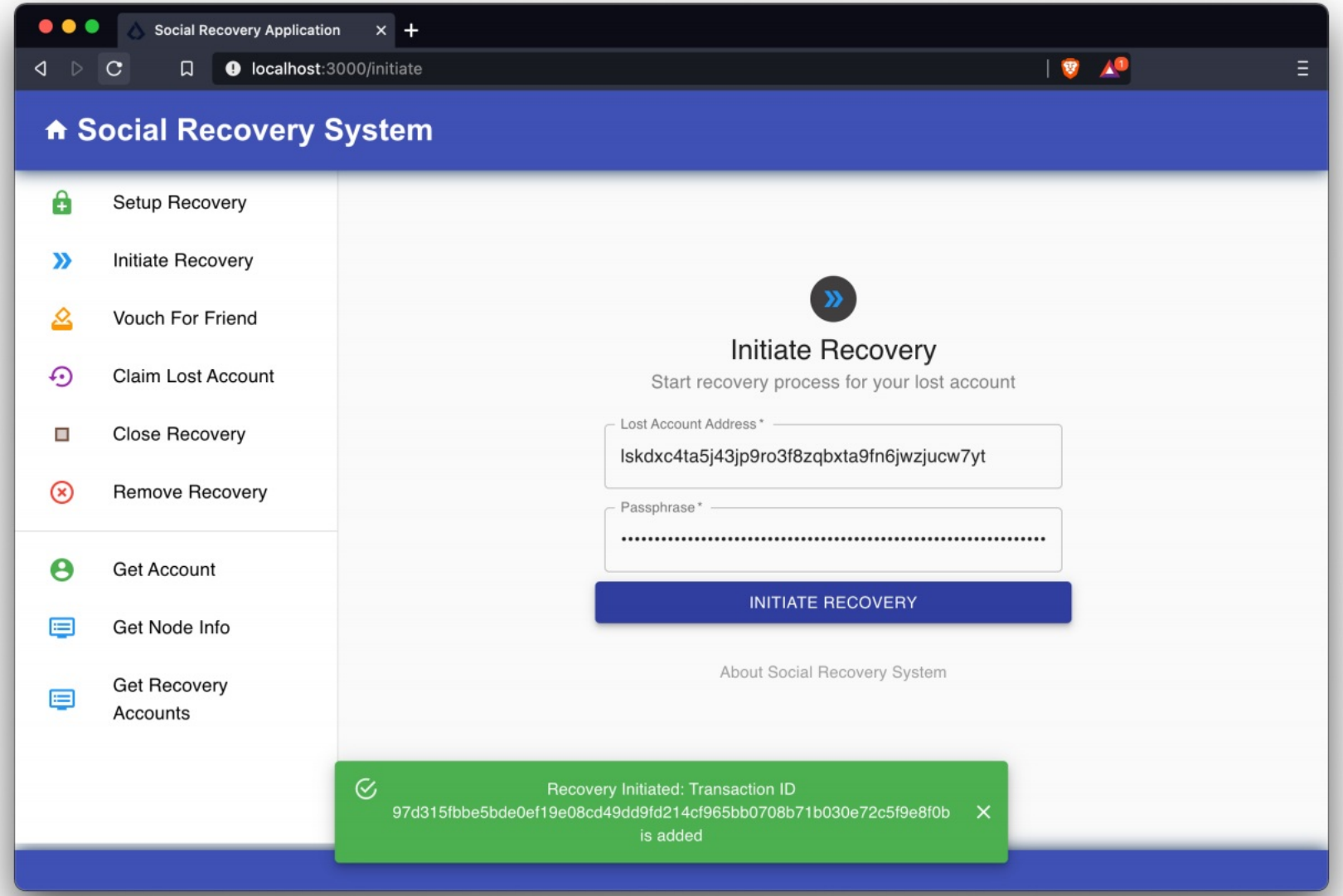


Logs of the blockchain app when a valid transaction is posted to the node

```
14:27:10 INFO lisk-framework: Forged new block (module=lisk:app)
{
 "id": "ce543f0fb39372f5f93d080cc09aa6ea6d67b3b51d959fb2132589d4644a05fc",
 "generatorAddress": "290abc4a2244bf0ecf5aa1ccee8ac8f60f8bce48",
 "seedReveal": "8172a6e21812be17290411930088c490",
 "height": 32,
 "slot": 985619,
 "reward": "0"
}
14:27:13 INFO lisk-framework: Added transaction to pool (module=lisk:app)
{
 "id": "60aa1929d17ff5f0219d562d349f2f5d006d60152a99a6e6e4b7193ead3f4a42",
 "nonce": "5",
 "senderPublicKey": "0fe9a3f1a21b5530f27f87a414b549e79a940bf24fdf2b2f05e7f22aeeecc86a"
}
14:27:20 INFO lisk-framework: New block added to the chain (module=lisk:app)
{
 "id": "d957a437e0165f8ef727f8ff530c1fc2ad3c3cc5e6cb3db0634e0d9fc299ca93",
 "height": 33,
 "numberOfTransactions": 1
}
14:27:20 INFO lisk-framework: Forged new block (module=lisk:app)
{
 "id": "d957a437e0165f8ef727f8ff530c1fc2ad3c3cc5e6cb3db0634e0d9fc299ca93",
 "generatorAddress": "2cf52c08cc76091d884e800c1c697b13f69907d4",
 "seedReveal": "6f20a1c7471a99e0c3a835559ad47e39",
 "height": 33,
 "slot": 985620,
 "reward": "0"
}
```

7.1.4. Initiating an account recovery

We assume that the original account owner lost their credentials and that they will use a new account to rescue the funds of the old account. Click on `Initiate Recovery` to initiate the recovery process for the original account. We use the passphrase of the rescuer account to sign the transaction. The data is pre-filled again with the correct values, so you can simply press the button `INITIATE RECOVERY` to initiate the recovery process.

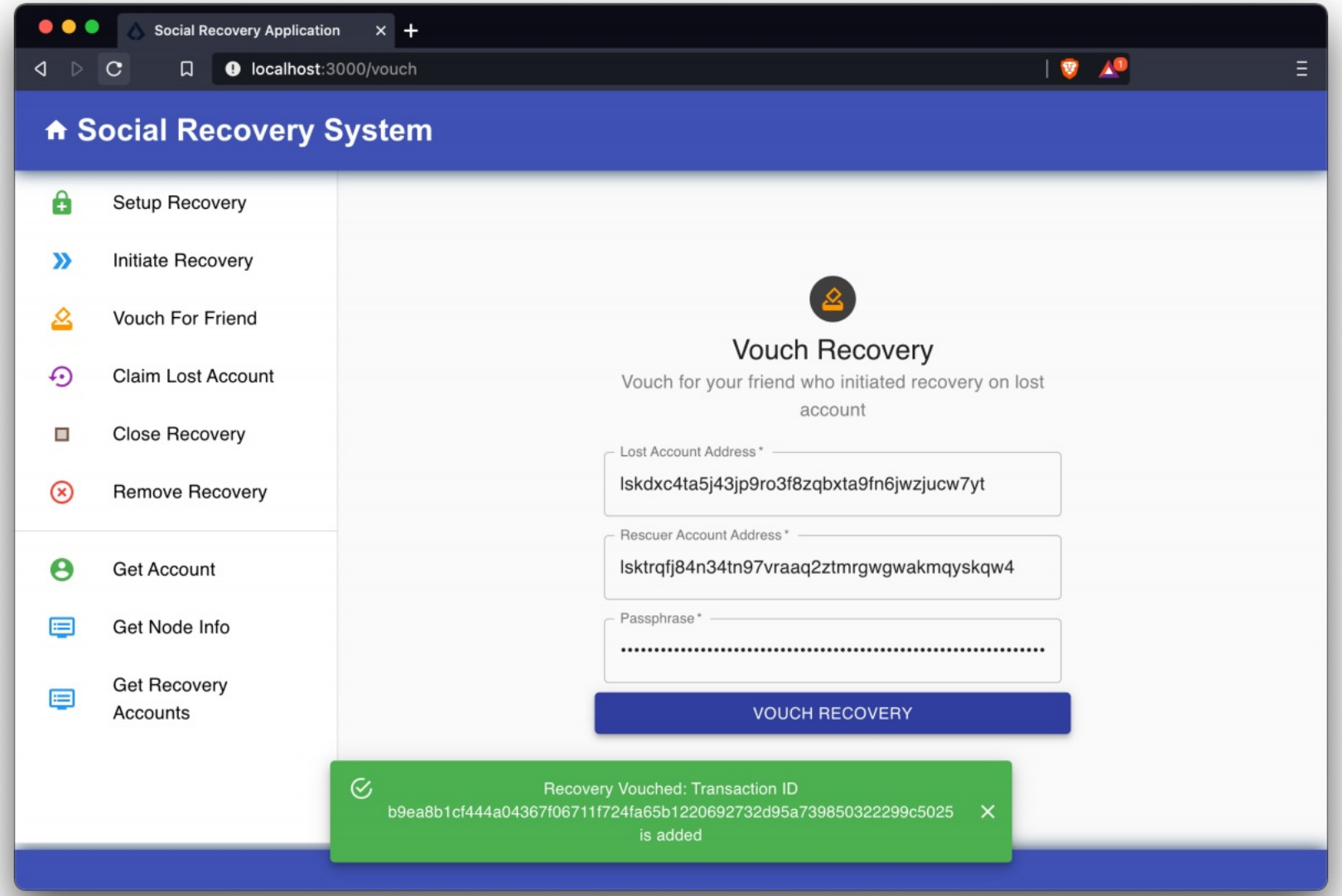


7.1.5. Vouching for a friend

We assume the owner of the rescuer account (who is also the owner of the original account), asks their 3 friends who are listed in the recovery configuration to vouch for their new account.

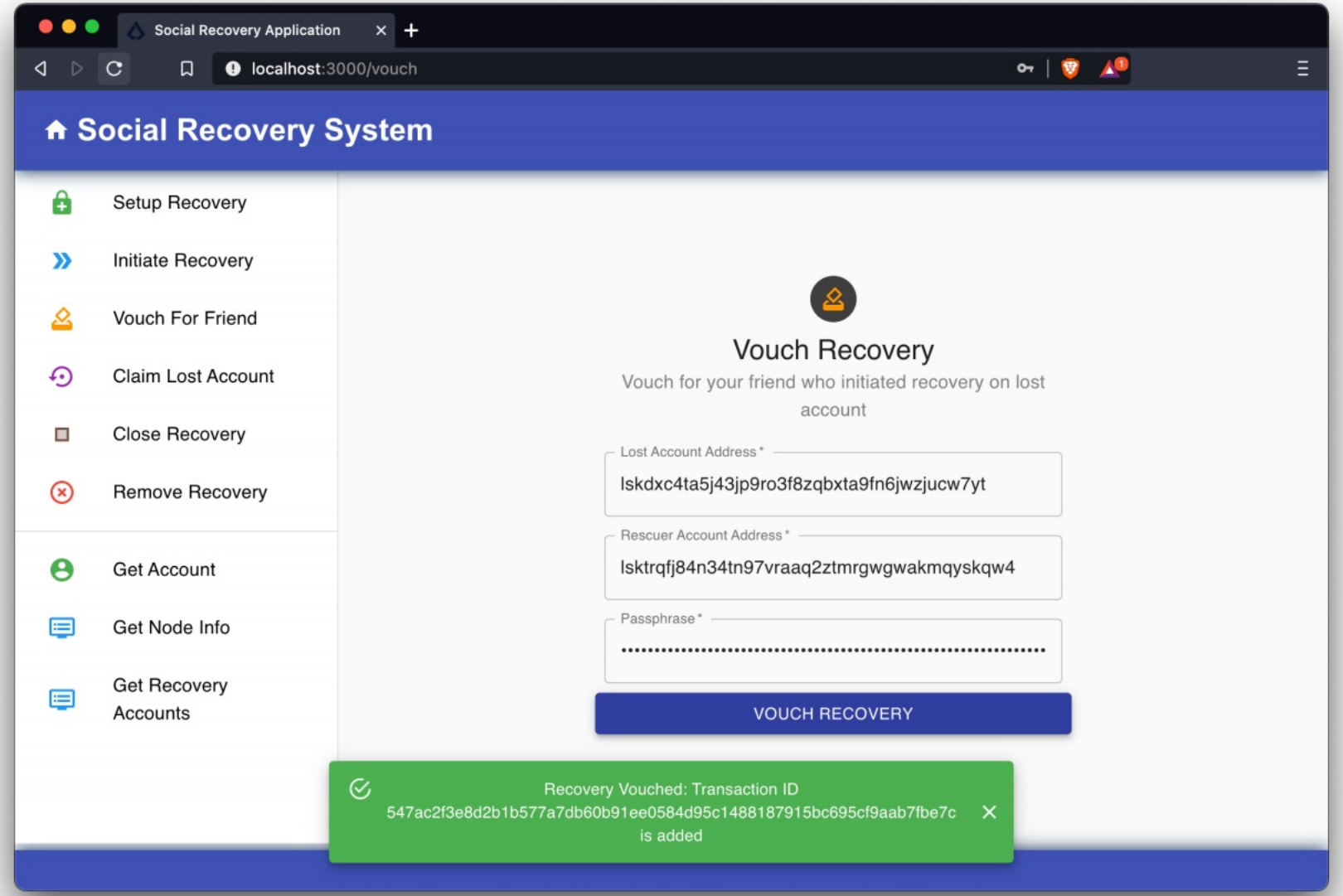
The first friend would then click on `Vouch For Friend` and has to vouch that the owner of the original account is the same as for the rescuer account. The data is pre-filled again with the correct values, so you can simply press the button

`VOUCH RECOVERY` to vouch for the rescuer account.



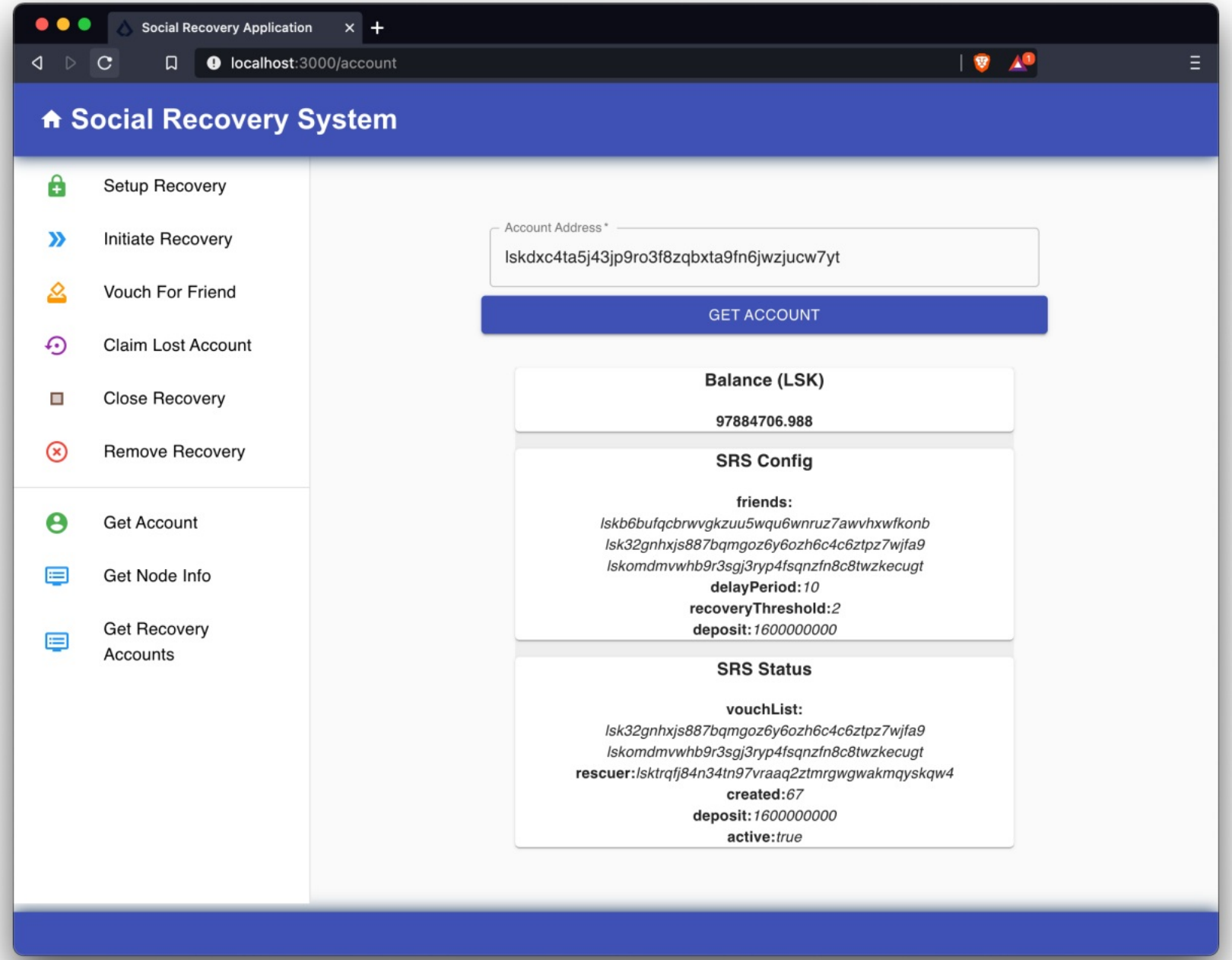
As we set the recovery threshold to 2 in the recovery config, we need one more friend that vouches for the rescuer, before the recovery can be claimed. Copy and paste the passphrase of `Friend 2` (or `Friend 3`, whichever you prefer), into the

passphrase field and vouch again by pressing the button `VOUCH RECOVERY`.



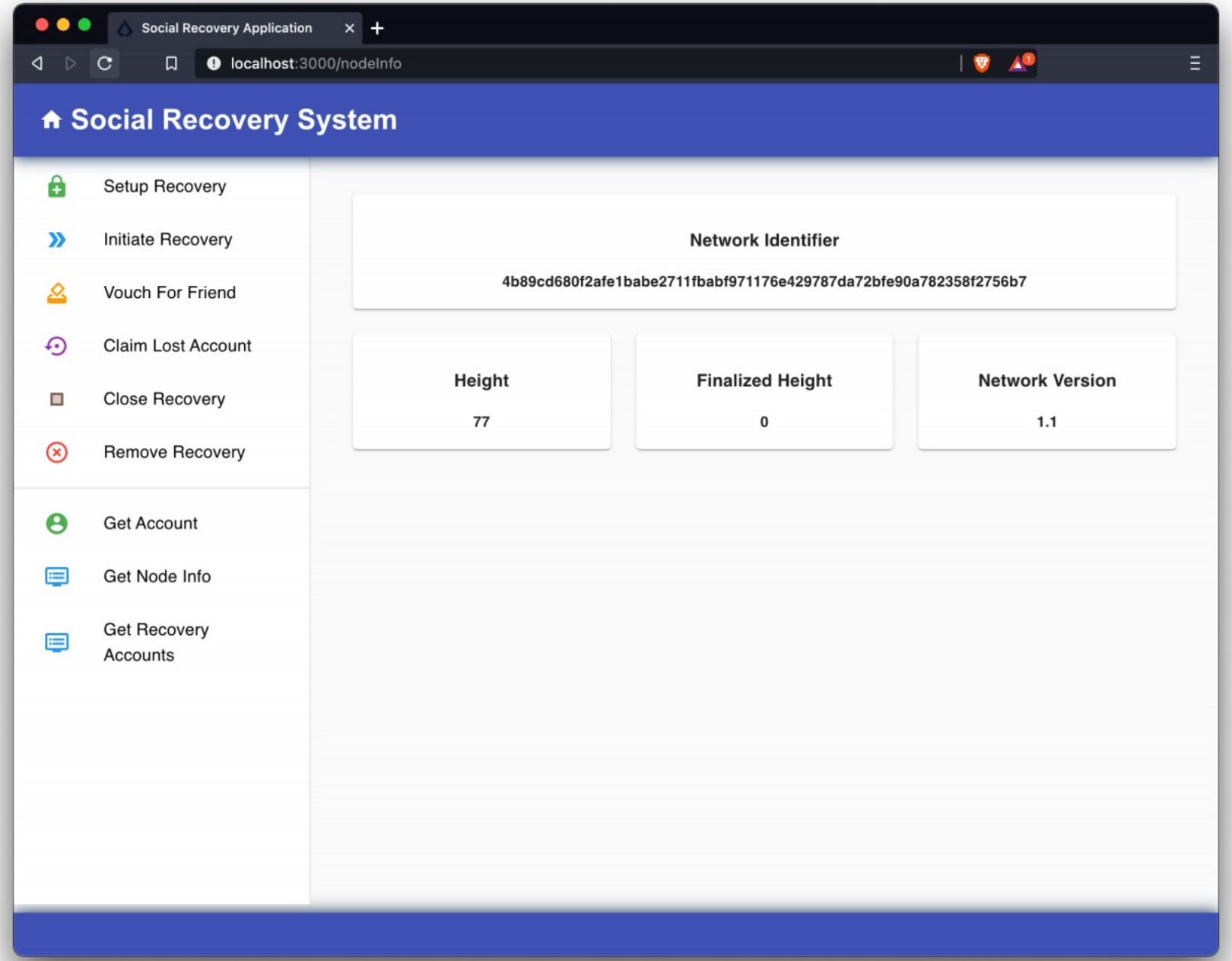
7.1.6. Checking the recovery status

After a few seconds, the transaction should be included in the blockchain. You can check the updated recovery status on the `Get Account` page. The vouch list should display the addresses of the two friend accounts that vouched for the rescuer account.



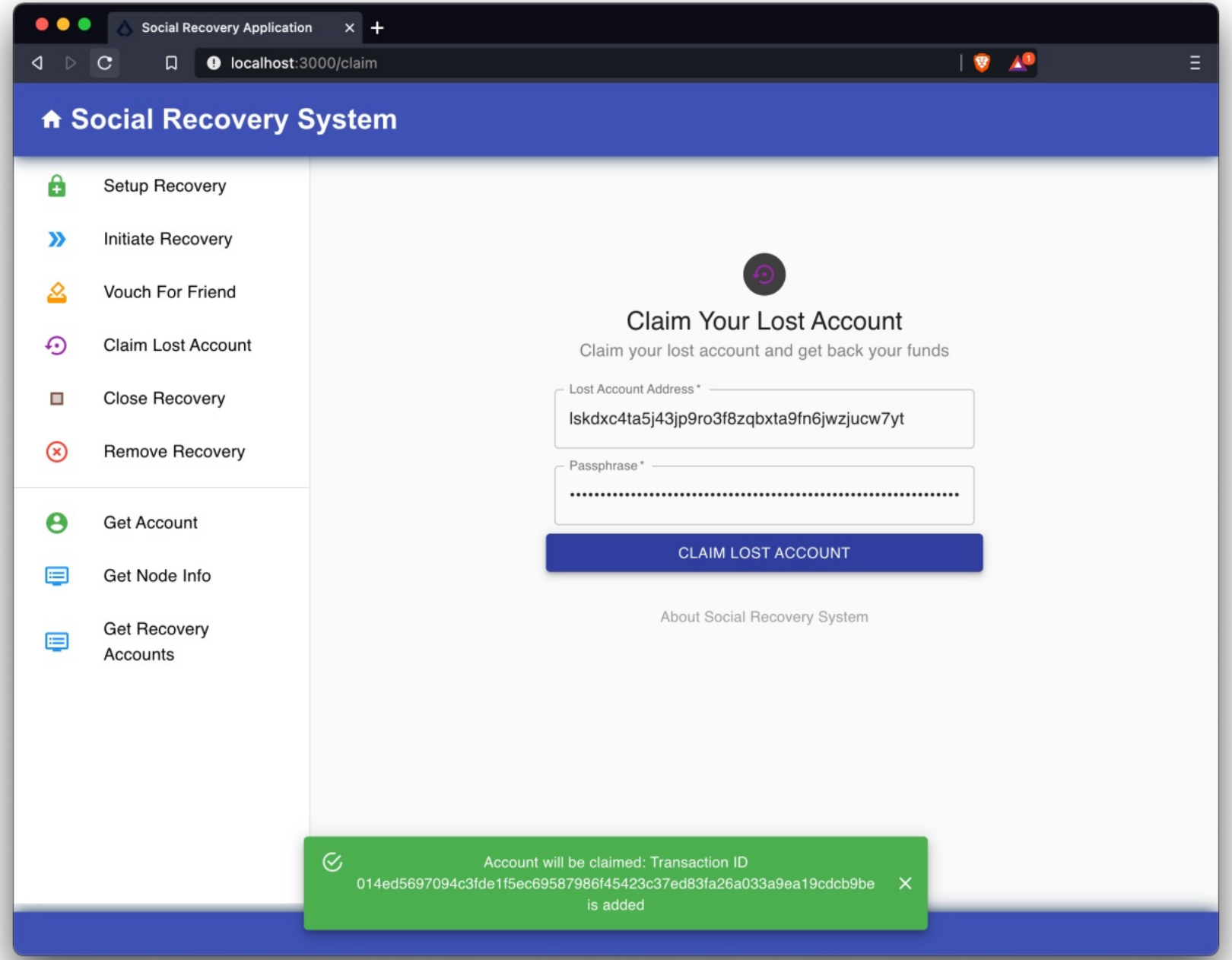
Check the current block height of the network on the `Get Node Info` page. The delay period is defined as 10 blocks in the recovery config of the account. This means that after initiating a recovery, the rescuer needs to wait for at least 10 blocks

until they can claim the recovery. The block height at which the recovery was initiated, can be seen in the recovery status of the original account.

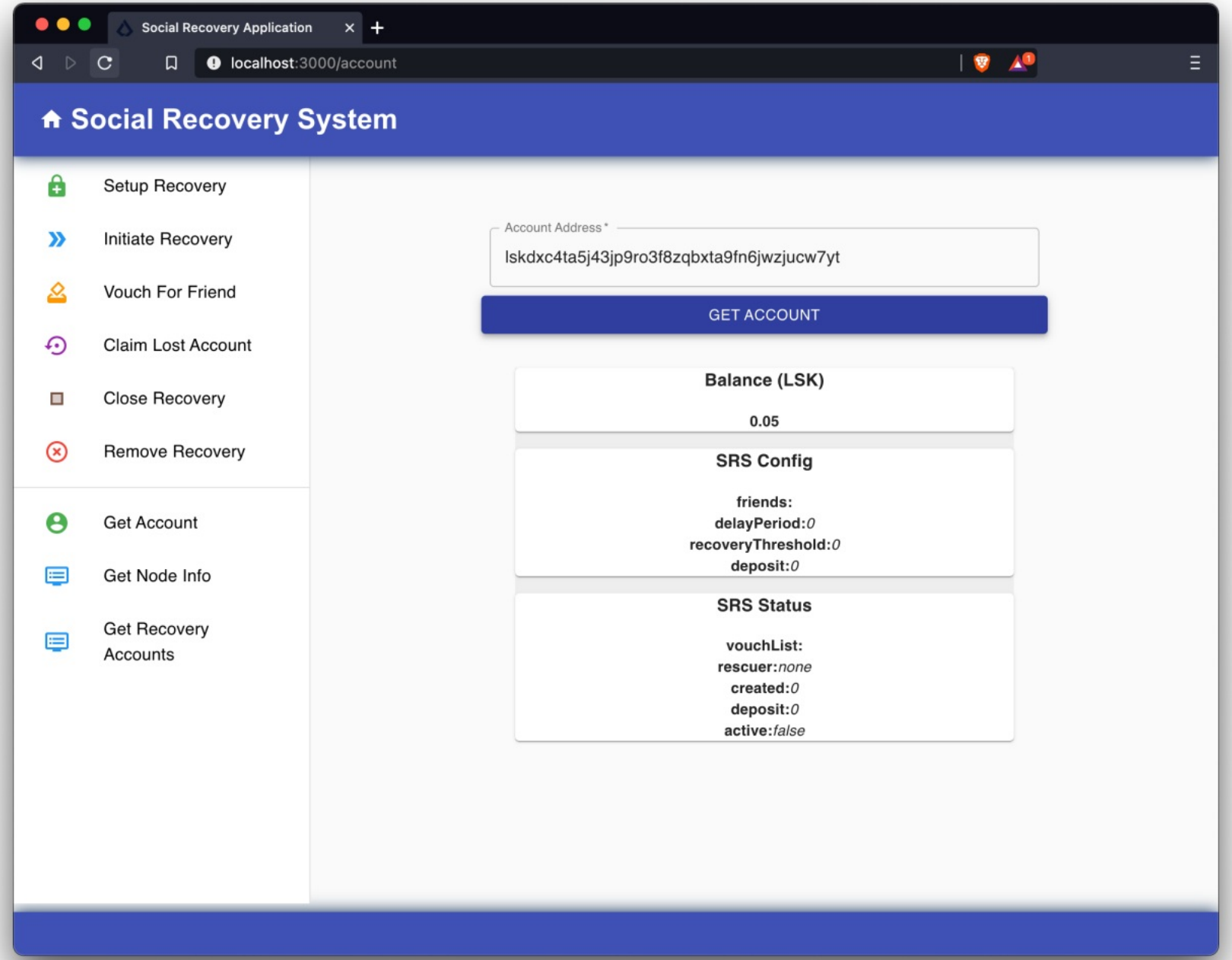


7.1.7. Claiming the recovery

The rescuer account can then claim the recovery. Click on `Claim Recovery` to claim the tokens in the original account. The address of the original account is pre-filled in the `Lost Account Address` field. The `Passphrase` field is pre-filled with the pasphrase of the rescuer account. Press the button `CLAIM RECOVERY` to finish the recovery process. As a result, all tokens in the balance of the original account will be transferred to the rescuer account.



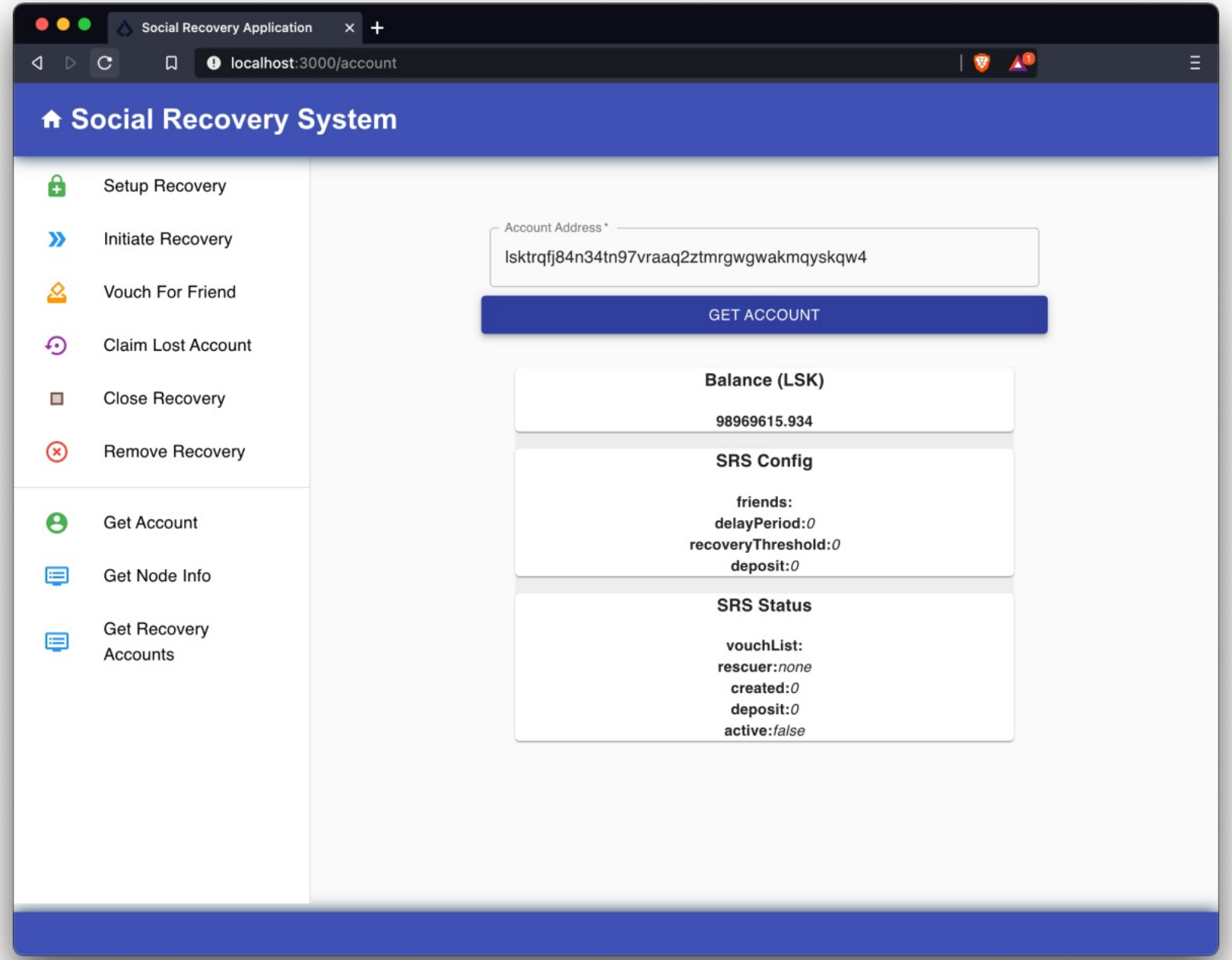
Empty balance of the original account


Social Recovery Application
localhost:3000/account
Social Recovery System
Setup Recovery
Initiate Recovery
Vouch For Friend
Claim Lost Account
Close Recovery
Remove Recovery
Get Account
Get Node Info
Get Recovery Accounts
Account Address *
lskdxc4ta5j43jp9ro3f8zqbxta9fn6jwzjucw7yt
GET ACCOUNT
Balance (LSK)
0.05
SRS Config
friends:
delayPeriod:0
recoveryThreshold:0
deposit:0
SRS Status
vouchList:
rescuer:none
created:0
deposit:0
active:false

Note

The 0.05 tokens remaining in the original account balance are the minimum required account balance.

Tokens of the original account have been transferred to the rescuer account.



That's it, the frontend walkabout in the browser is now complete.

Next let's take a dive into the most important parts of the frontend app, regarding the blockchain related logic.

7.2. API related functions

Multiple API-related functions are defined that fetch and post data from and to the blockchain application.

We will make use of the following APIs that are provided by the blockchain application:

- `http://localhost:4000/api/` : The HTTP API of the HTTPAPIPlugin. Used to retrieve general blockchain information from the database.
- `http://localhost:8080/api/` : The HTTP API of the SRSAPIPlugin. Used to post transactions to the blockchain application.
- `ws://localhost:8888/ws` : The WebSocket API of the blockchain application. Used to invoke actions in the blockchain application.

The following functions are defined:

- Fetching blockchain data
  - `fetchAccountInfo(address)`: Returns all account details for a given account address from the database.
  - `fetchNodeInfo()`: Returns various information about the node of the blockchain application, for example, the current block height.
- Posting transactions
  - `sendTransactions(tx, action)`: Sends a transaction to the blockchain application. Accepts two arguments:
    - `tx`(object): the transaction data
    - `action`(string): the endpoint of the SRSAPIplugin, that is used in combination with the transaction data.
- Invoking actions
  - `fetchRecoveryConfigs()`: Returns a list of all accounts with recovery configurations.

srs/frontent_app/src/api/index.js

```
import { apiClient } from '@liskhq/lisk-client';

const RPC_ENDPOINT = 'ws://localhost:8888/ws';
```

```
let clientCache;

export const getClient = async () => {
  if (!clientCache) {
    clientCache = await apiClient.createWSClient(RPC_ENDPOINT);
  }
  return clientCache;
};

export const fetchRecoveryConfigs = async () => {
  const client = await getClient();
  return client.invoke('SRSData:getAllRecoveryConfigs');
};

export const sendTransactions = async (tx, action) => {
    return fetch(`http://localhost:8080/api/recovery/${action}`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(tx),
    })
      .then((res) => res.json())
      .then((res) => res);
  };

export const fetchAccountInfo = async (address) => {
    return fetch(`http://localhost:4000/api/accounts/${address}`)
      .then((res) => res.json())
      .catch((res) => res.data);
};

export const fetchNodeInfo = async () => {
  return fetch("http://localhost:4000/api/node/info")
    .then((res) => res.json())
    .catch((res) => res.data);
};
```

7.3. Components

For the frontend the following components are defined:

- Home(): Renders the general landing page of the frontend application.
- CreateRecovery(): Renders the page for Creating a recovery config.
- InitiateRecovery(): Renders the page for Initiating an account recovery.
- VouchRecovery(): Renders the page for Vouching for a friend.
- ClaimRecovery(): Renders the page for Claiming the recovery.
- CloseRecovery(): Renders the page for closing a recovery.
- RemoveRecovery(): Renders the page for removing a previously created recovery configuration.
- GetAccount(): Renders the page for the account details.
- GetNodeInfo(): Displays the current node information.
- GetRecoveryConfigs(): Displays a list of all accounts with recovery configurations.
- SideMenu(): Renders the side navigation which allows the user to switch between the different pages of the frontend application.
- RecoveryManager(): Manages the routing of the different components of the frontend application.

The code examples for the `CreateRecovery` and `GetRecoveryConfigs` are explained in detail below. If you are curious how the other components of the frontend app are implemented, please directly view the example code. Most logic of the other components is implemented analog to the presented examples below:

7.3.1. CreateRecovery component

The `CreateRecovery` component shall allow the user to create a recovery configuration for their account.

To achieve this, the following information is required:

1. a list of trusted friends
2. the desired delay period
3. the desired recovery threshold

4. the passphrase of the account, to sign the transaction

`CreateRecovery` renders a form to receive this information from the user. When the user presses the button `CREATE RECOVERY CONFIG`, it sends the relevant data to the blockchain application.

srs/frontent_app/src/components/createRecovery.js

```
import React, { Fragment, useContext, useState } from 'react';
import Avatar from '@material-ui/core/Avatar';
import Button from '@material-ui/core/Button';
import CssBaseline from '@material-ui/core/CssBaseline';
import TextField from '@material-ui/core/TextField';
import Link from '@material-ui/core/Link';
import Grid from '@material-ui/core/Grid';
import Box from '@material-ui/core/Box';
import Typography from '@material-ui/core/Typography';
import { makeStyles } from '@material-ui/core/styles';
import Container from '@material-ui/core/Container';
import { grey, green } from '@material-ui/core/colors';
import EnhancedEncryptionIcon from '@material-ui/icons/EnhancedEncryption';
import { sendTransactions } from '../api';
import { createRecoveryDefaults } from "../utils/defaults";
import Snackbar from '@material-ui/core/Snackbar';
import MuiAlert from '@material-ui/lab/Alert';
import { cryptography } from '@liskhq/lisk-client';

function Alert(props) {
  return <MuiAlert elevation={6} variant="filled" {...props} />;
}

function Footer() {
  return (
    <Typography variant="body2" color="textSecondary" align="center">
      <Link style={{ color: grey[500] }} href="/">
        About Social Recovery System
      </Link>
    </Typography>
  );
}
```

```
const useStyles = makeStyles((theme) => ({
  paper: {
    marginTop: theme.spacing(8),
    display: 'flex',
    flexDirection: 'column',
    alignItems: 'center',
  },
  avatar: {
    margin: theme.spacing(1),
    backgroundColor: 'rgb(37 35 35 / 87%)',
  },
  form: {
    width: '100%',
    marginTop: theme.spacing(3),
  },
  submit: {
    margin: theme.spacing(3, 0, 2),
  },
}));

export default function CreateRecovery() {
  const classes = useStyles();
  const [open, setOpen ] = useState(false);

  const [data, setData] = useState({
    friends: createRecoveryDefaults.friends,
    delayPeriod: createRecoveryDefaults.delayPeriod,
    recoveryThreshold: createRecoveryDefaults.recoveryThreshold,
    passphrase: createRecoveryDefaults.passphrase,
    msg: '',
    severity: 'success',
  });

  const handleClose = (event, reason) => {
    if (reason === 'clickaway') {
      return;
    }

    setOpen(false);
```

```
  };

  const handleChange = (event) => {
    event.persist();
    setData({ ...data, [event.target.name]: event.target.value });
  };

  const handleSend = async (event) => {
    event.preventDefault();
    const { friends } = data;

    const friendList = friends ? friends.split(',').map(str => str.replace(/\s/g, '')): [];
    const binaryFriends = friendList.map(friend =>
cryptography.getAddressFromBase32Address(friend).toString('hex'));
    try {
        const result = await sendTransactions({ delayPeriod: data.delayPeriod, recoveryThreshold:
data.recoveryThreshold, friends: binaryFriends, passphrase: data.passphrase },
window.location.pathname.slice(1));
        if (result.errors) {
            setData({ msg: result.errors[0].message, severity: 'error' });
        } else {
            setData({ msg: `Transaction ID ${result.data.transactionId} is added`, severity: 'success'
});
        }
        setOpen(true);

    } catch (error) {}
  };

  return (
    <Container component="main" maxWidth="xs">
      <CssBaseline />
      <div className={classes.paper}>
        <Avatar className={classes.avatar}>
        <EnhancedEncryptionIcon style={{ color: green[500] }}/>
        </Avatar>
        <Typography component="h1" variant="h5">
          Setup Social Recovery
        </Typography>
        <Typography component="h4" style={{color: 'grey'}}>
```

```
          Create recovery configuration for your account
        </Typography>
        <form className={classes.form} noValidate autoComplete="off">
          <Grid container spacing={2}>
            <Grid item xs={12}>
              <TextField
                variant="outlined"
                required
                fullWidth
                id="friends"
                label="Comme separated addresses of friends"
                name="friends"
                multiline
                rows={5}
                onChange={handleChange}
                defaultValue={createRecoveryDefaults.friends}
              />
            </Grid>
            <Grid item xs={12}>
              <TextField
                variant="outlined"
                required
                fullWidth
                name="delayPeriod"
                label="Delay Period"
                id="delayPeriod"
                onChange={handleChange}
                defaultValue={createRecoveryDefaults.delayPeriod}
              />
            </Grid>
            <Grid item xs={12}>
              <TextField
                variant="outlined"
                required
                fullWidth
                name="recoveryThreshold"
                label="Recovery Threshold"
                id="threshold"
                onChange={handleChange}
                defaultValue={createRecoveryDefaults.recoveryThreshold}
```

```
            />
          </Grid>
          <Grid item xs={12}>
            <TextField
              variant="outlined"
              required
              fullWidth
              name="passphrase"
              label="Passphrase"
              id="passphrase"
              type="password"
              onChange={handleChange}
              defaultValue={createRecoveryDefaults.passphrase}
            />
          </Grid>
        <Button
          onClick={handleSend}
          fullWidth
          variant="contained"
          color="primary"
        >
          Create Recovery Config
        </Button>
        </Grid>
      </form>
      <Snackbar open={open} autoHideDuration={10000} onClose={handleClose}>
          <Alert onClose={handleClose} severity={data.severity}>
              <label id='msg'>{data.msg}</label>
          </Alert>
      </Snackbar>
    </div>
    <Box mt={5}>
      <Footer />
    </Box>
  </Container>
 );
}
```

7.3.2. GetRecoveryConfigs component

The `GetRecoveryConfigs` component shall display a complete list of all accounts that created a recovery configuration.

To achieve this, the function `fetchRecoveryConfigs()` from the API related functions is used to fetch all accounts with a recovery configuration from the blockchain application.

srs/frontent_app/src/components/getRecoveryAccounts.js

```
import React, {
  useEffect,
  useState
} from 'react';
import {
  Grid,
  CssBaseline,
  Container,
} from '@material-ui/core';
import {
  makeStyles
} from '@material-ui/core/styles';
import { fetchRecoveryConfigs } from '../api';
import RecoveryConfig from "./recoveryConfig";

const useStyles = makeStyles((theme) => ({
  root: {
    flexGrow: 1,
  },
  paper: {
    padding: theme.spacing(2),
    textAlign: 'center',
    color: theme.palette.text.primary,
  },
}));

export default function GetRecoveryConfigs () {
  const classes = useStyles();
  const [data, setData] = useState({
    result: []
  });
```

```
  useEffect(() => {
    async function getRecoveryConfigs() {
      const result = await fetchRecoveryConfigs();
      if ( result.length > 0 ) {
        setData({ result });
      }
    }
    getRecoveryConfigs()
  }, [])

  return (
    <Container component="main" className={classes.paper}>
      <CssBaseline />
      <div className={classes.root}>
        { data.result.length > 0
          ?
        <Grid container spacing={3}>
          {  data.result.map((config) => (
            <Grid item xs={12}>
              <RecoveryConfig item={config} key={config.address} />
            </Grid>
          )) }
        </Grid>
          : <p>No recoverable accounts found</p> }
      </div>
    </Container>
  );
}
```

8. Summary

That's it! You should now have a complete blockchain application running which allows users to recover their accounts in the case whereby they have lost their credentials.

It consists of the following components:

- a blockchain application with:
    - a custom module for a social account recovery system.

    - a custom plugin which provides a list of all accounts with created recovery configuration.
    - a custom plugin which provides additional HTTP API endpoints to conveniently create and post different transaction types.
- a frontend application which allows you to use and test the applications in the browser.

9. Optional exercises

Feel free to play around with the code example by changing/adjusting certain options, or by extending the application in your own way.

The following exercises might be interesting to get more familiar with the Lisk SDK.

9.1. Implement HTTP API endpoint as plugin actions

The `SRSAPIplugin` provides a lot of useful functions for creating and posting the different transaction objects to the blockchain application. These functions could alternatively be created as actions inside of the plugin. The frontend can then invoke the different actions analog to the function `fetchRecoveryConfigs()` which we defined in the API related functions. `fetchRecoveryConfigs()` invokes the action `SRSData:getAllRecoveryConfigs` to receive a list of all account with recovery configurations.

Try to re-implement the different HTTP API endpoints as actions in The SRS API plugin, and invoke those actions in the frontend to create and send the different transaction types.

9.2. Notify an account owner, if an account recovery for their account has been initialized

In the section Lifecycle hooks, we defined that the event `srs:recoveryInitiated` is published for every incoming "initiate recovery" transaction.

Try to use this event to warn an account owner that a recovery has been initiated for their account. The original account holder will be warned in case someone malicious starts a recovery on their account to steal their funds, and if it was a malicious actor who initiated the recovery, then the user can quickly close the recovery process before the delay period is passed and the set threshold number of friends that had already vouched for it.

Writing unit tests

In the final chapter of this tutorial, it is covered how to test the application with both unit and network tests.

Writing tests for the application is more and more important, the more complex the blockchain application becomes. Once a certain complexity is reached, it will not be convenient anymore to test the application functionality with the dashboard plugin, the CLI, or via a frontend. Therefore, writing tests is considered as crucial in order to verify the correct behavior of the blockchain application.

For this purpose, a couple of unit tests are added to the application in this chapter, which is followed by a couple of network tests in the next chapter.

To ensure the creation of the tests can be performed in a convenient manner, the The Lisk SDK testing utilities will be used.

Navigate into the folder of the LNS application. The folder is the correct location to store all kinds of different tests for the blockchain application.

The unit tests are stored in the folder, as the name suggests.

As can be seen, there are already some existing test files for the lns module and the assets. These files were auto-generated by Lisk Commander when the LNS module and the assets were generated.

The existing test files already contain test skeletons, providing a rough structure how to write the required tests.

Testing the Register asset

The complete code of the tests for the Register asset is described below. Most of the code is self-explanatory, however, the most important parts of the tests are summarized here to provide a better overview:

Testing the validate() function

Tests for the validate() function of the Register asset.

Write tests to check the following:

- It should not be possible to set the TTL in the asset to a value lower than 3600 seconds, (1 hour).
- It should throw an error if the name is registered for less than a year.
- It should throw an error if the name is registered for more than 5 years.
- It should throw an error if the domain contains an invalid tld.
- If no errors are thrown, then all asset parameters are valid.

Before each test:

- Create a new instance of the Register asset.

Use of the SDK testing utilities:

- : Returns valid parameters for the function. If the function is called with an empty object, it returns the default parameters for the function. For the test, overwrite the default value ( ), with a valid transaction asset for the Register asset. For the function to be called successfully, overwrite the default transaction value of the context ( ), with a transaction containing a

property with a Buffer of size 0. It is not necessary to put a real address here in this case, as it is not used in the tests.

Testing the apply() function

Tests for the apply() function of the Register asset.

Write tests to check the following:

- Valid cases:
  - It should update the state store with the name hash key.
  - It should update the state store with the updated sender account.
  - It should update the state store with the correct ttl value.
  - It should update the state store with the correct expiry date.
- Invalid cases:
  - It should throw an error if the name is already registered.

Before each test:

- Create a new default account for the LNS application.
- Add the newly created account to the list of the mock.
- Spy on the functions and . This allows checking in the tests if the respective functions have been called or not.

Use of the SDK testing utilities:

- : Returns valid parameters for the function. If the function is called with an empty object, it returns the default parameters for the function. For the test, overwrite the default value ( ), with a valid transaction asset for the Register asset. Additionally, overwrite the default transaction value of the context ( ), with a transaction containing a property which equals the address of the newly created account.
- : Used to create a default account for the LNS application.

- : Creates a mock for the StateStore.

Unit tests for the Register asset

Testing the Reverse Lookup asset

The entire code for the tests of the Reverse Lookup asset is described below. The majority of the code is self-explanatory, however, the most important parts of the tests are summarized here to provide a better overview:

Testing the apply() function

Tests for the apply() function of the Reverse Lookup asset.

Write tests to check the following:

- Valid cases:
  - It should update the lns reverse-lookup of the sender account with the given node if it is not already set.
  - It should update the lns reverse-lookup of the sender account with the given node even if it is already set.
- Invalid cases:
  - It should throw an error if the node to set-lookup is not owned by the sender.

Before each test:

- Create a new default account for the LNS application.
- Add two registered names to the account: and .
- Add the newly created account to the list of the mock.
- Spy on the functions and . This allows checking in the tests, if the respective functions have been called or not.

Use of the SDK testing utilities:

- : Returns valid parameters for the function. If the function is called with an empty object, it returns the default parameters for the function. For the test, overwrite the default value ( ), with a valid transaction asset for the Reverse Lookup asset. Additionally, overwrite the default transaction value of the context ( ), with a transaction containing a property which equals the address of the newly created account.

- : Used to create a default account for the LNS application.
- : Creates a mock for the StateStore.

Unit tests for the Reverse Lookup asset

Writing functional tests

Functional testing is a quality assurance (QA) process and a type of black-box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and in this case the internal program structure is rarely considered (unlike white-box testing). Functional testing is conducted to evaluate the compliance of a system or component with specified functional requirements. Functional testing usually describes what the system does.

As a final exercise, write a functional test, that checks if a domain name was correctly resolved after calling the action .

Create a new file in the folder:

The functional test should verify the following:

- action of the LNS module.
  - It should throw an error on resolving a non-registered name.
  - It should resolve the name after registration.

Before all tests:

- Create a new default environment for the LNS application.
- Start the application of the application environment.

Use of the SDK testing utilities:

- : Creates a default application environment for the functional test.

- : Creates a default account for the LNS application.
- : Use the passphrase of the default faucet account to send tokens from the faucet to the newly created default account.

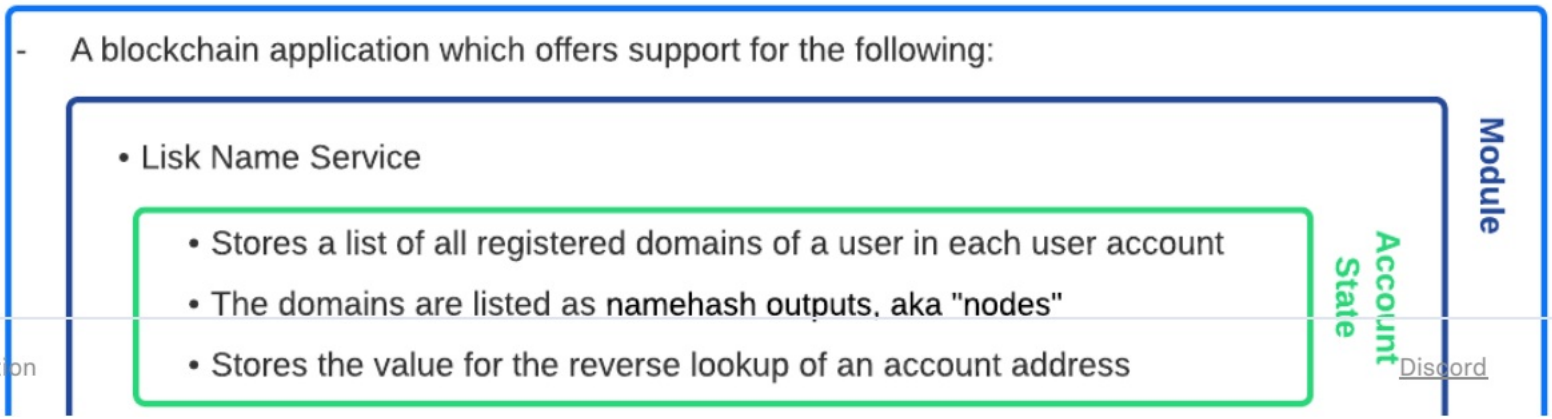
Functional tests for the action of the LNS module

How to create a blockchain application that offers a domain name service for blockchain accounts, similar to the Domain Name System, (DNS), and the Ethereum Name Service, (ENS).

This service allows an account to register one or multiple `.lsk` domains for a certain amount of time, (1-5 years).

If an account registers a domain, it becomes a human readable identifier for the account and can then be used to refer to the account instead of the (less human-readable) Lisk32 account address.

Additionally, it is possible to create TXT and CNAME records for each domain owned by an account.



Chain State

- Domains are stored in the database as LNS objects
- Each LNS object includes:
  - name: domain name
  - ttl: Time-To-Live, time that needs to pass, until records can be updated again for this domain
  - expiry: Duration how long the account has reserved this domain
  - ownerAddress: Address of the account who reserved this domain
  - records: List of created TXT & CNAME records for this domain

Transactions

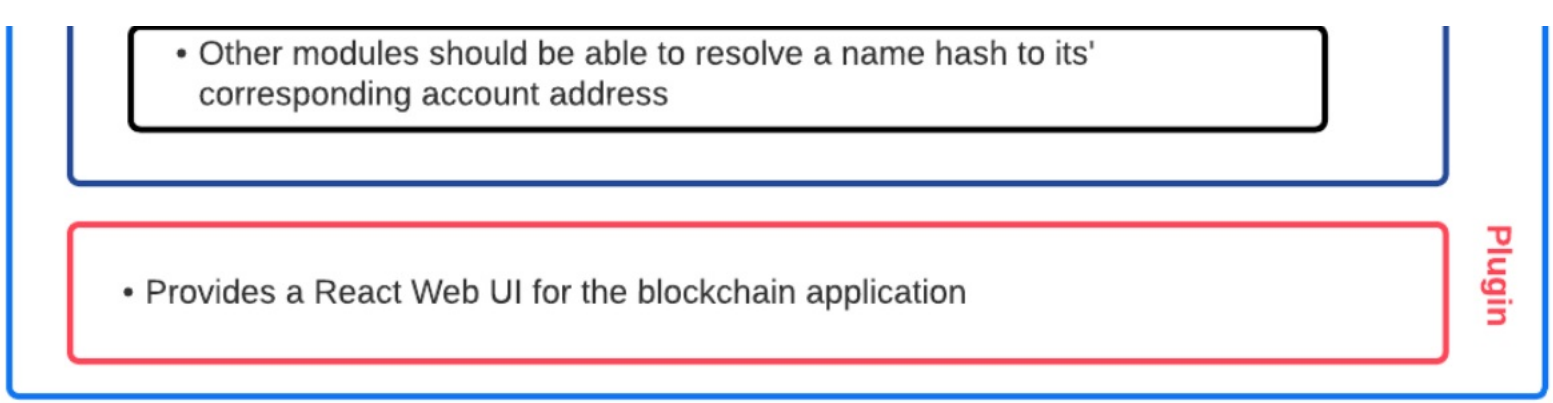
- Anyone can register an availlable .lsk domain for their account.
- The owner of a domain can create and update records for the domain
- The owner of a domain can update the reverse lookup domain for an account address

Actions

- Everyone should be able to perform a reverse lookup for a given account address
- Everyone should be able to resolve a name to its' corresponding account address
- Everyone should be able to resolve a name hash to its' corresponding account address

Reducers

- Other modules should be able to perform a reverse lookup for a given account address
- Other modules should be able to resolve a name to its' corresponding account address



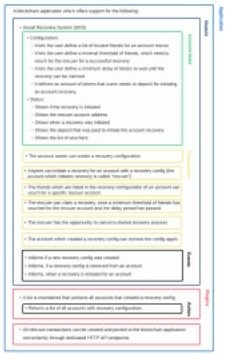
Chapter 1: Blockchain development	This chapter covers how to perform the following:
	• ...create a blockchain application with a module which provides a Lisk name service. The Lisk Name Service allows users to register domain names for their accounts. It can then resolve the human readable domain names to their corresponding account address, making the domain name a human readable alias for the address.
	• ...create three different transaction assets:
	  ◦ Register: To register a new domain name.
	  ◦ Reverse Lookup: To define the reverse lookup for an account address.
	  ◦ Update Records: To update the records of a domain name.
	• ...connect the Dashboard plugin to interact with the LNS app during development.

Chapter 2: Frontend development	This chapter covers how to perform the following: • ...create a plugin that provides a React.js frontend for the LNS blockchain application.

Chapter 3: Extending the CLI	This chapter covers how to perform the following: • ...create lns commands. • ...try out the new CLI commands.
Chapter 4: Unit & functional tests	This chapter covers how to perform the following: • ...write unit tests for the newly implemented transactions assets. • ...write functional tests for the LNS module.

To become both familiar and fully conversant with the Lisk SDK, a step-by-step walk through guide is provided that simplifies the process of developing a proof of concept blockchain application.

Name	Estimated time required	Complexity	Content
Hello World 	~1-2 h	Beginner	Follow the development guides to learn how to perform the following: • ...bootstrap a simple blockchain application. • ...create a custom module. • ...create a custom plugin. • ...configure a blockchain application. • ...connect a dashboard to the application. • ...write unit tests modules and assets.
Non-fungible token (NFT) Tutorial 	~2-3 h	Beginner	Learn how to create: • ...a blockchain application that allows a user to create, transfer and purchase NFTs. • ...3 different transaction assets to create, transfer, and purchase NFTs. • ...a custom plugin which adds a new HTTP API that serves NFT-related data to the application. • ...a frontend application that allows the user to utilize the blockchain application in the browser.

Name	Estimated time required	Complexity	Content
Social Recovery System (SRS) Tutorial  	~4 h	Intermediate	Learn how to create: • ...a blockchain application that offers users the opportunity to recover their accounts if they have lost their credentials through a social recovery system. • ...6 different transaction assets to manage the recovery process. • ...2 custom plugins that provide new actions and API endpoints which are helpful in the frontend. • ...a frontend application that allows the user to utilize the blockchain application in the browser.

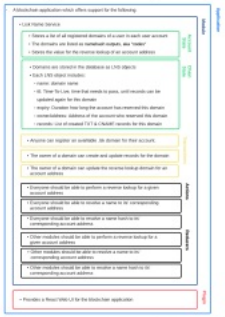
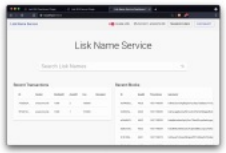
Name	Estimated time required	Complexity	Content
Lisk Name Service (LNS) Tutorial  	~4 h	Intermediate	Learn how to: • ...create a blockchain application with a module which provides a name service for Lisk addresses. The Lisk Name Service allows users to register domain names for their accounts. It can then resolve the human readable domain names to their corresponding account address, making the domain name a human readable alias for the address. • ...create three different transaction assets: ◦ Register: To register a new domain name. ◦ Reverse Lookup: To define the reverse lookup for an account address. ◦ Update Records: To update the records of a domain name. • ...connect the Dashboard plugin to interact with the LNS app during development. • ...create a plugin that provides a React.js frontend for the LNS blockchain application. • ...extend the LNS application CLI with additional commands. • ...write unit tests for the newly implemented transactions assets. • ...write functional tests for the LNS module.

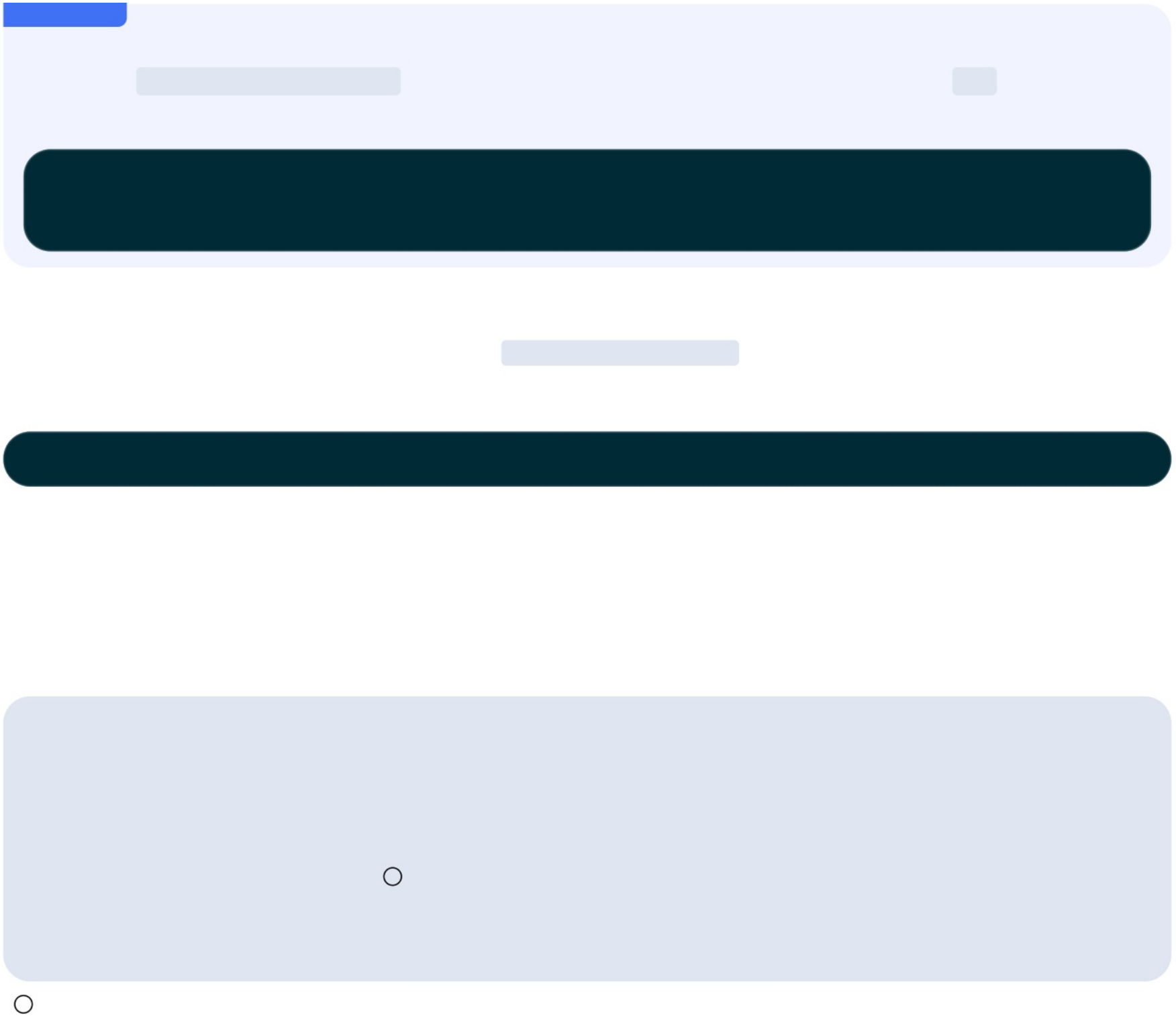
Table of Contents

-
-
-
-
-
-

4. Connecting the Dashboard and Faucet plugins

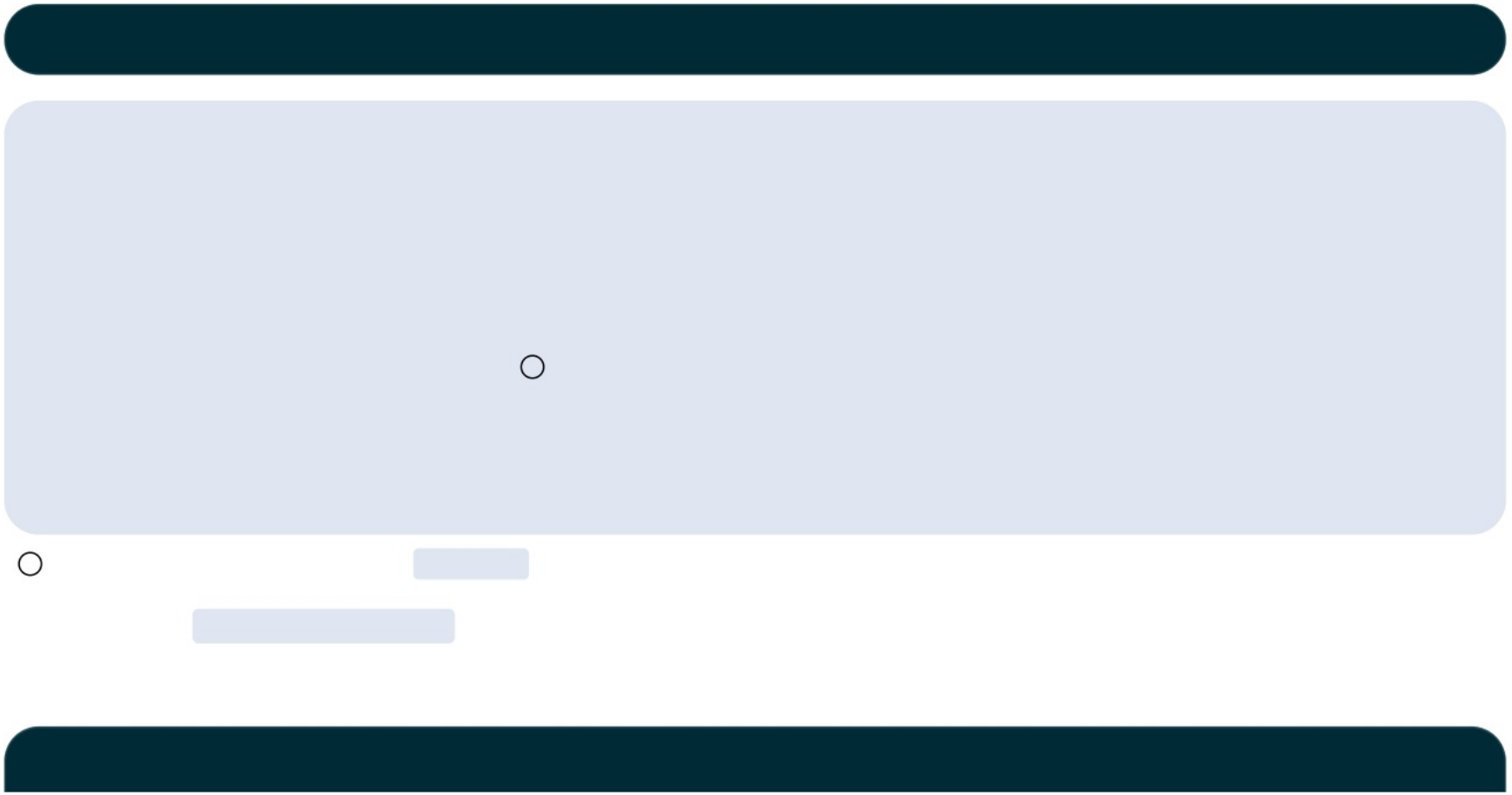
-
-
-
  -
  -
  -
  -
  -
  -
  -
  -
  -

1. Bootstrapping the default application

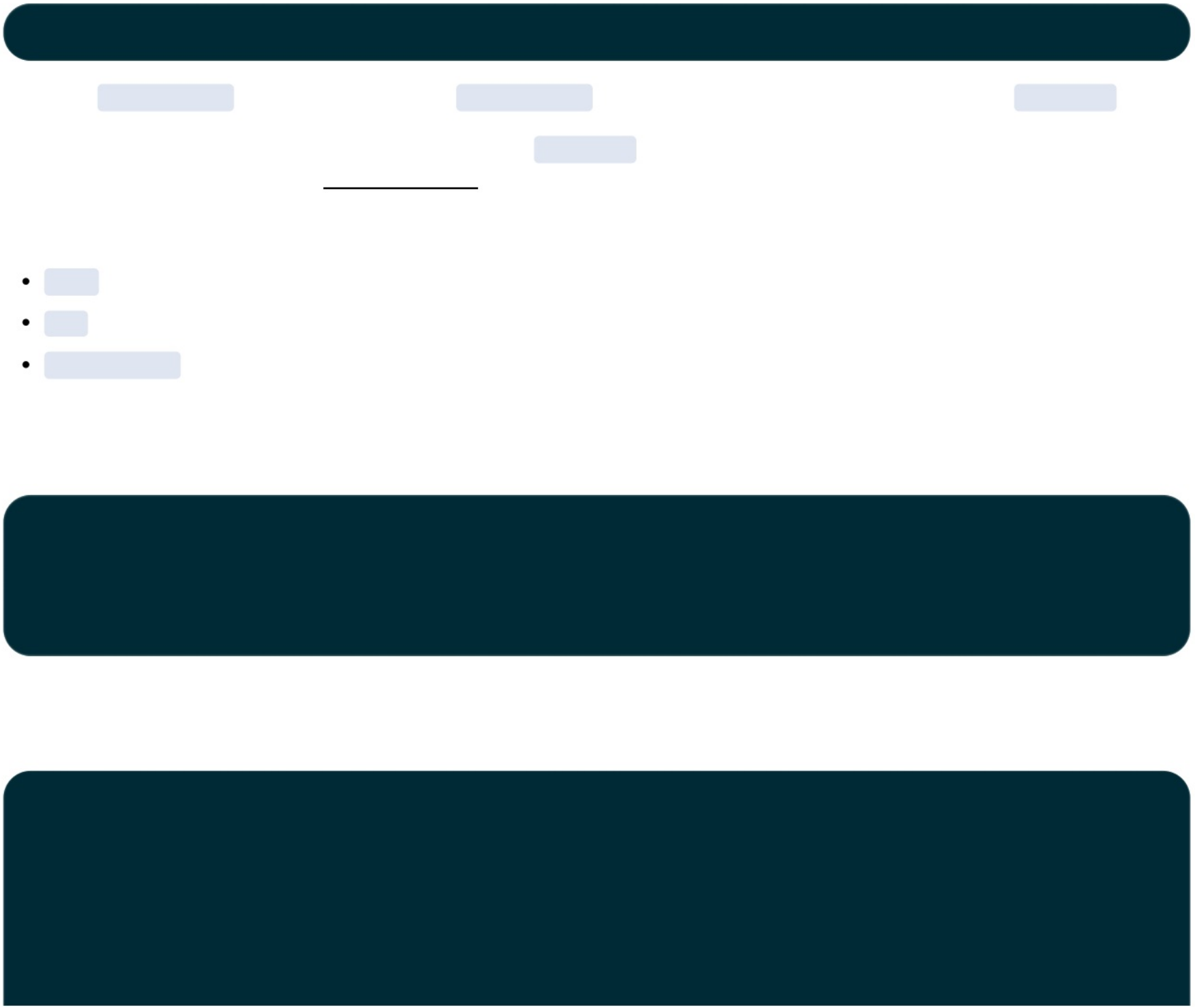


2. Creating the LNS module assets

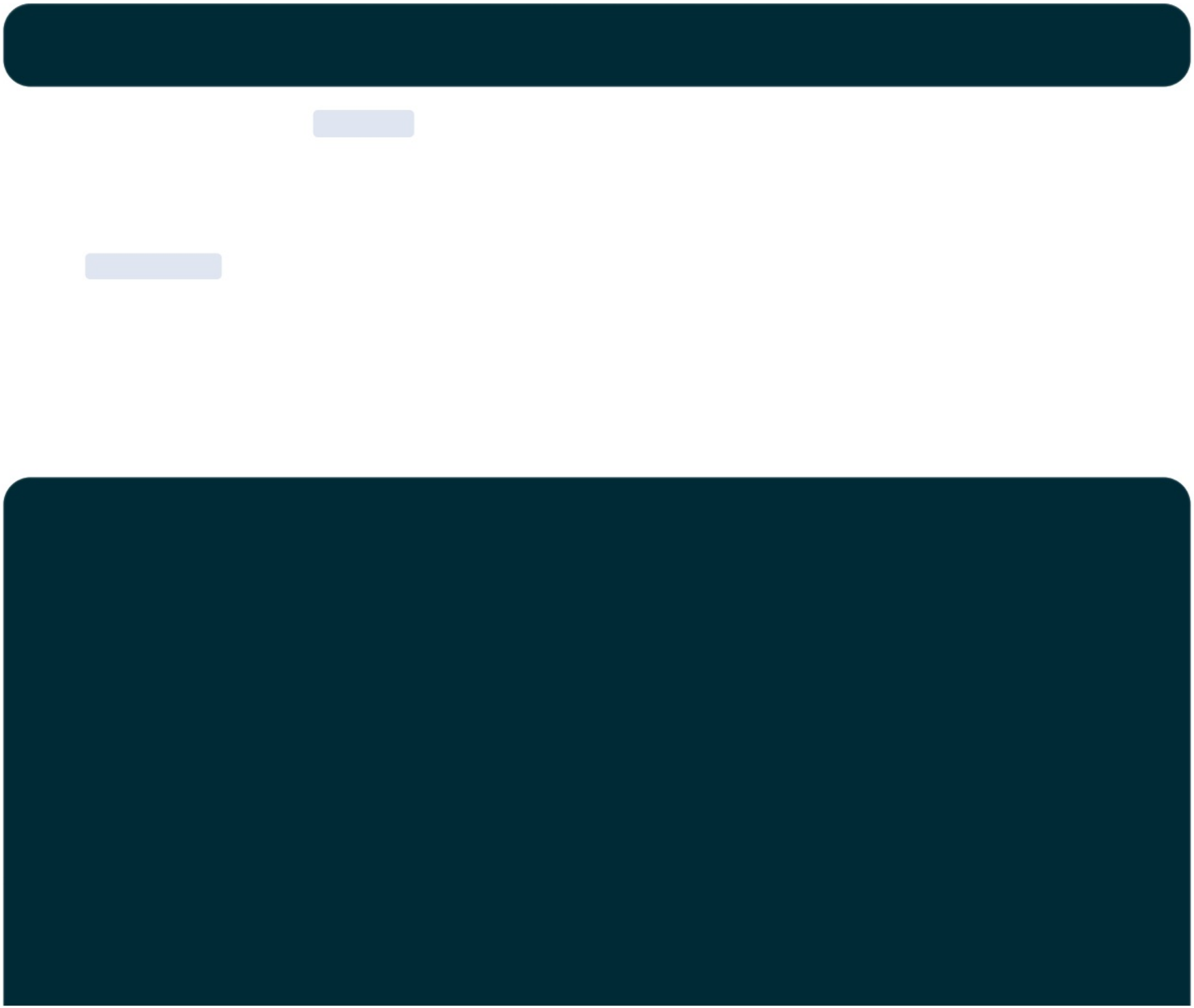
2.1. Register Domain



2.1.1. Schema



2.1.2. Validation



2.1.3. State change

-
-
-

2.1.4. Utility functions

-
-
-
-

2.1.5. Schemas

-
-

2.2. Update reverse lookup

2.2.1. Schema

2.2.2. State change

2.3. Update records

2.3.1. Schema

2.3.2. Validation

2.3.3. State change

-
-
-
-

2.3.4. Utility functions

3. Creating the LNS module

- [redacted]
- [redacted]

3.1. Account schema

3.2. Assets

3.3. Actions

3.4. Reducers

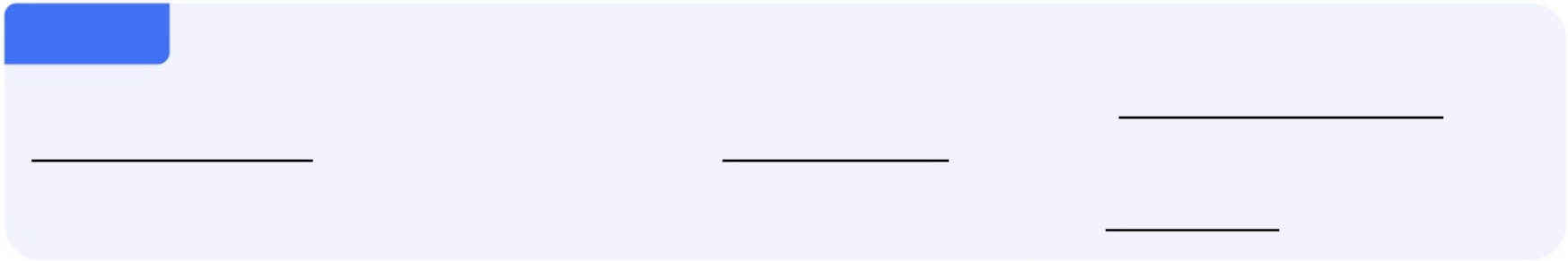
3.5. Utility functions

3.6. Updating the genesis block

4. Connecting the Dashboard and Faucet plugins

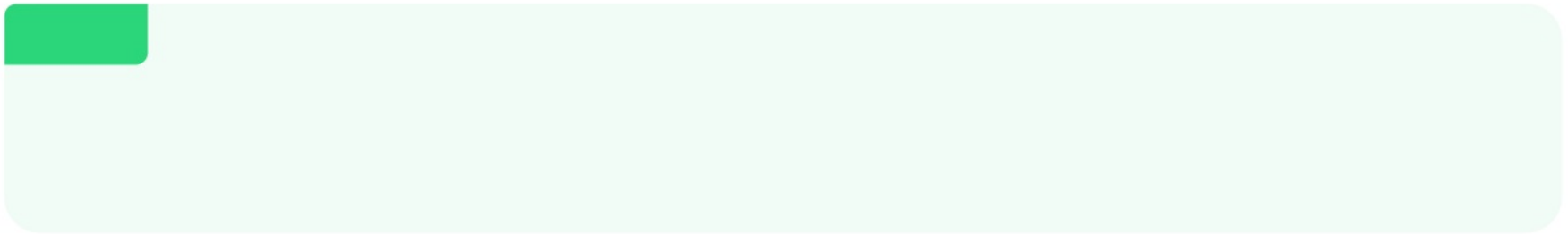
-
-
-

-
-
-
-
-
-
-
-

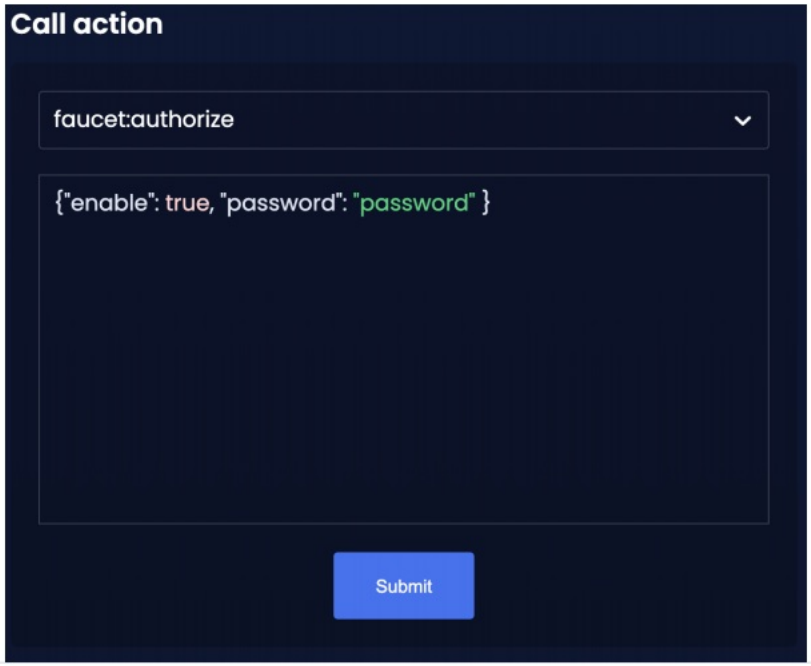


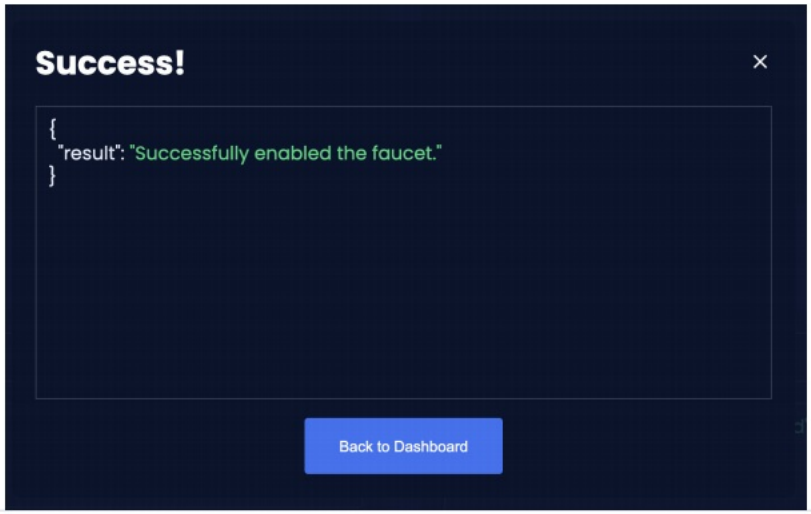
4.1. Registering the Dashboard plugin

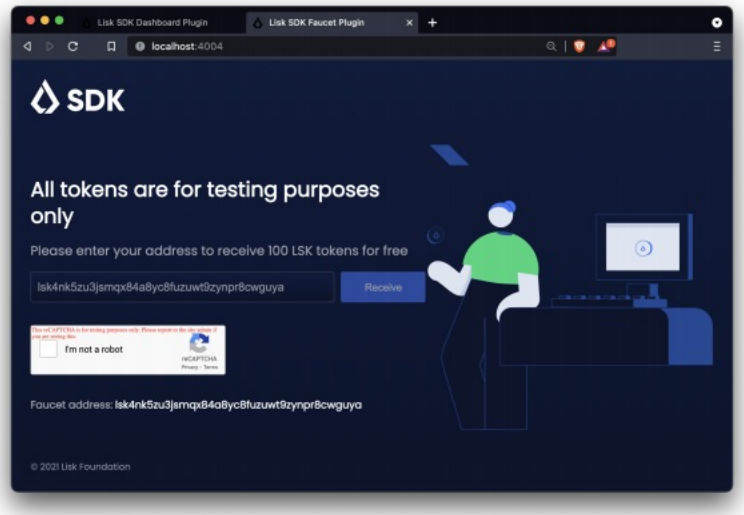
4.2. Registering the Faucet plugin



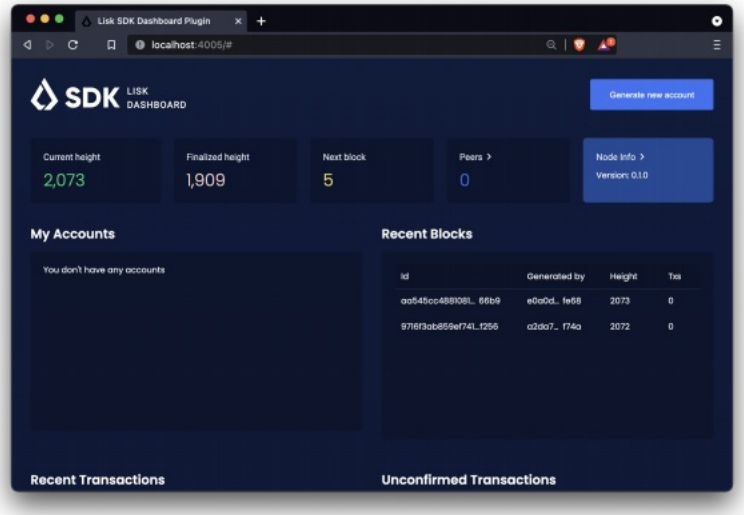
-
-
-


Call action
faucet:authorize
{"enable": true, "password": "password" }
Submit

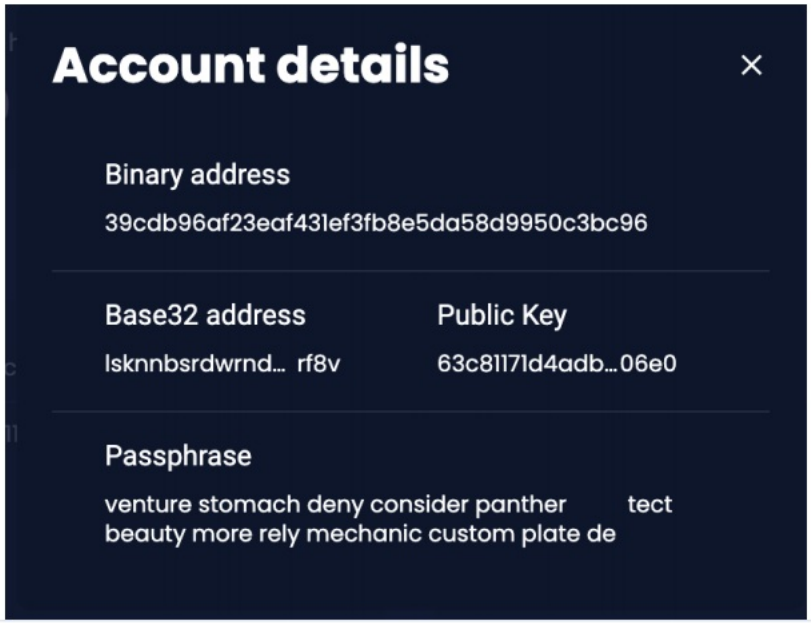




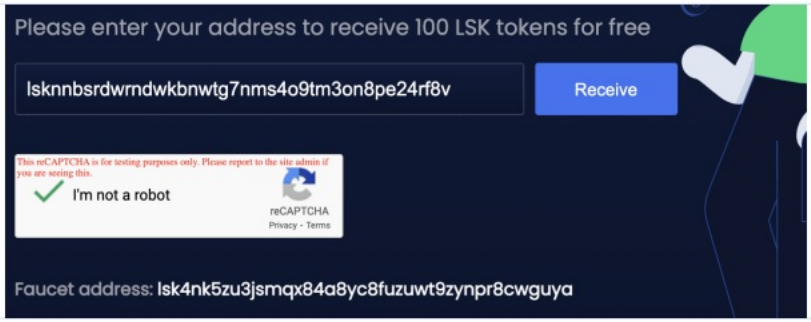
4.3. Checking the functionality of the LNS module

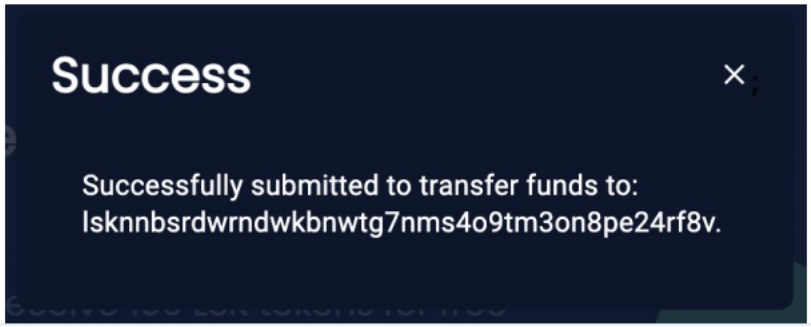


4.3.1. Create new account

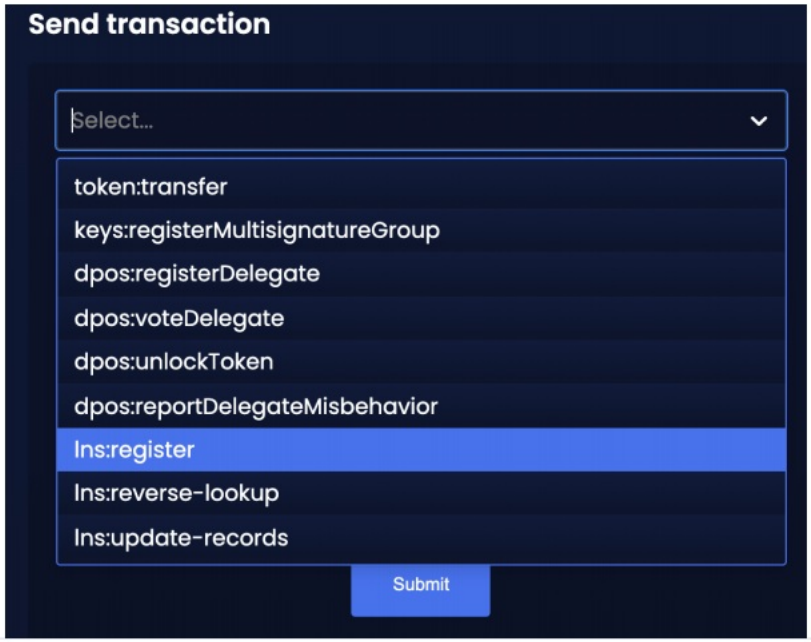


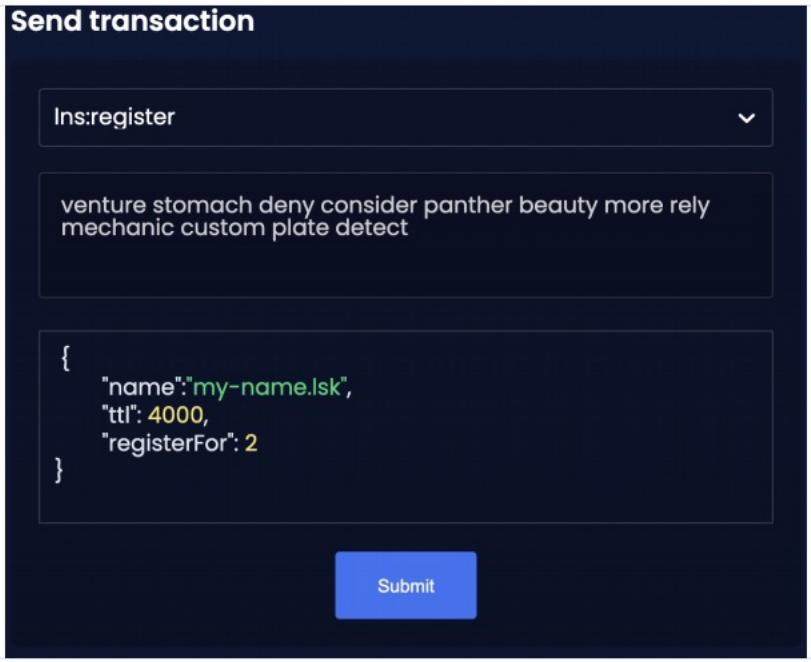
4.3.2. Fund new account

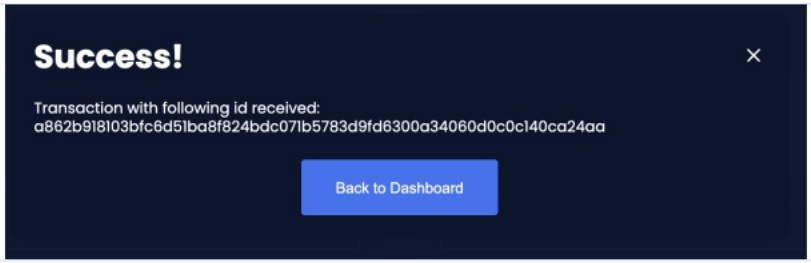




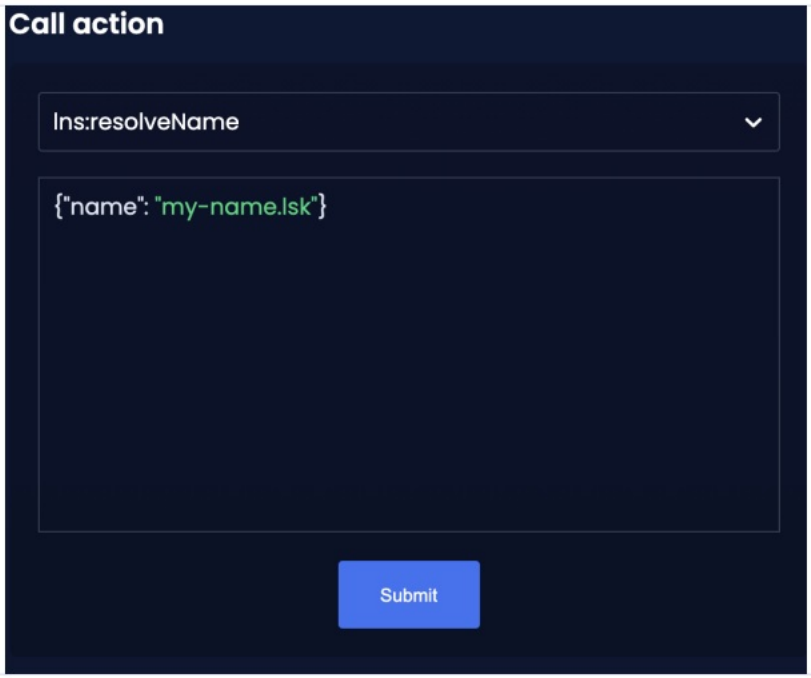
4.3.3. Register new domain

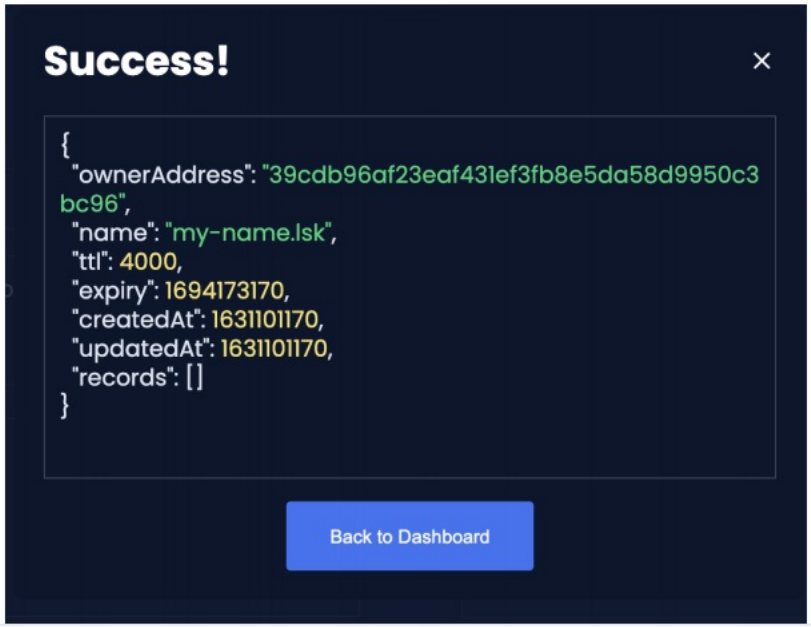

Send transaction
Select...
token:transfer
keys:registerMultisignatureGroup
dpos:registerDelegate
dpos:voteDelegate
dpos:unlockToken
dpos:reportDelegateMisbehavior
lns:register
lns:reverse-lookup
lns:update-records
Submit

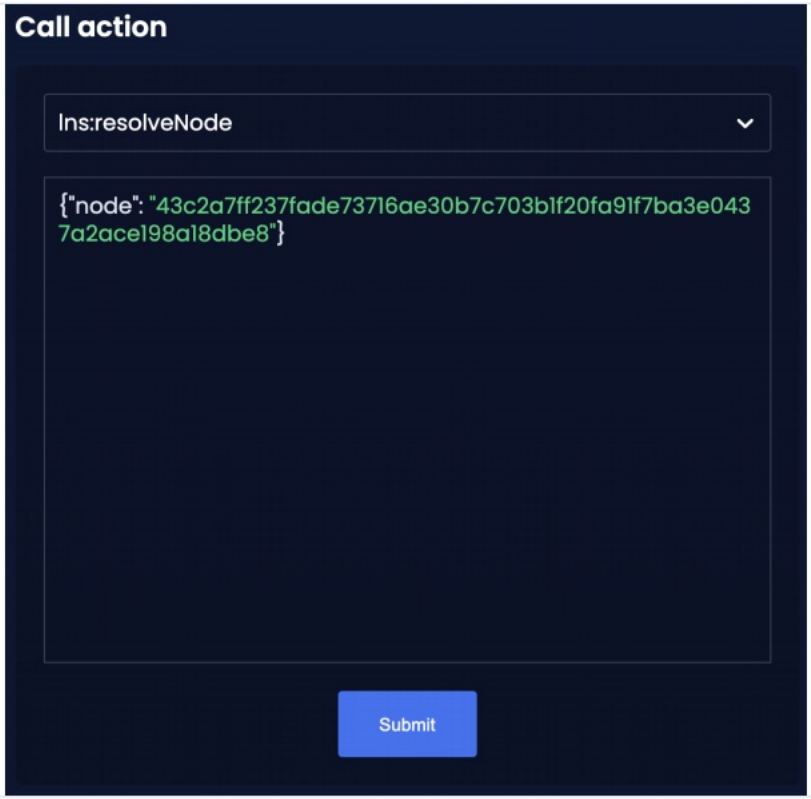

Send transaction
lns:register
venture stomach deny consider panther beauty more rely mechanic custom plate detect
{
"name":"my-name.lsk",
"ttl": 4000,
"registerFor": 2
}
Submit



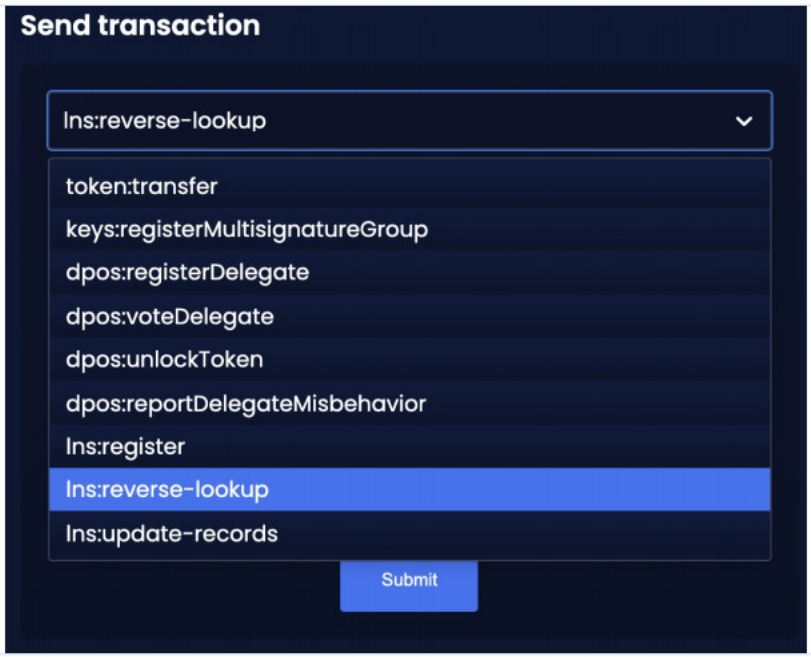
4.3.4. Call the actions resolveName and resolveNode

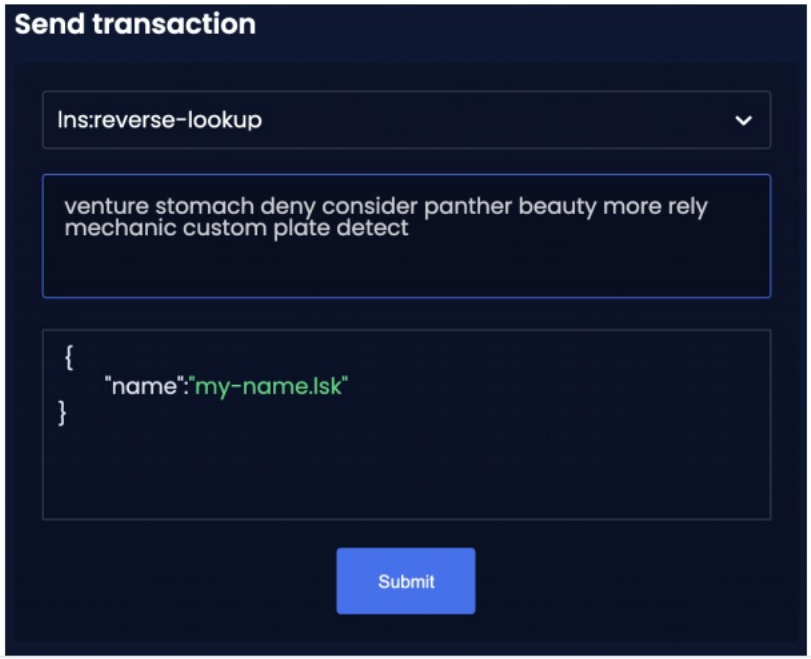


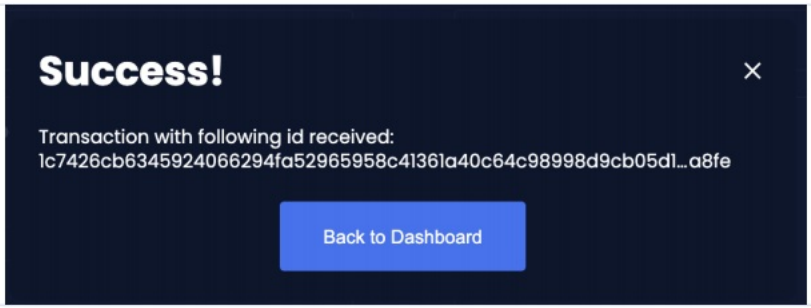

Success!
{
"ownerAddress": "39cdb96af23eaf431ef3fb8e5da58d9950c3bc96",
"name": "my-name.lsk",
"ttl": 4000,
"expiry": 1694173170,
"createdAt": 1631101170,
"updatedAt": 1631101170,
"records": []
}
Back to Dashboard


Call action
lns:resolveNode
{"node": "43c2a7ff237fade73716ae30b7c703b1f20fa91f7ba3e0437a2ace198a18dbe8"}
Submit

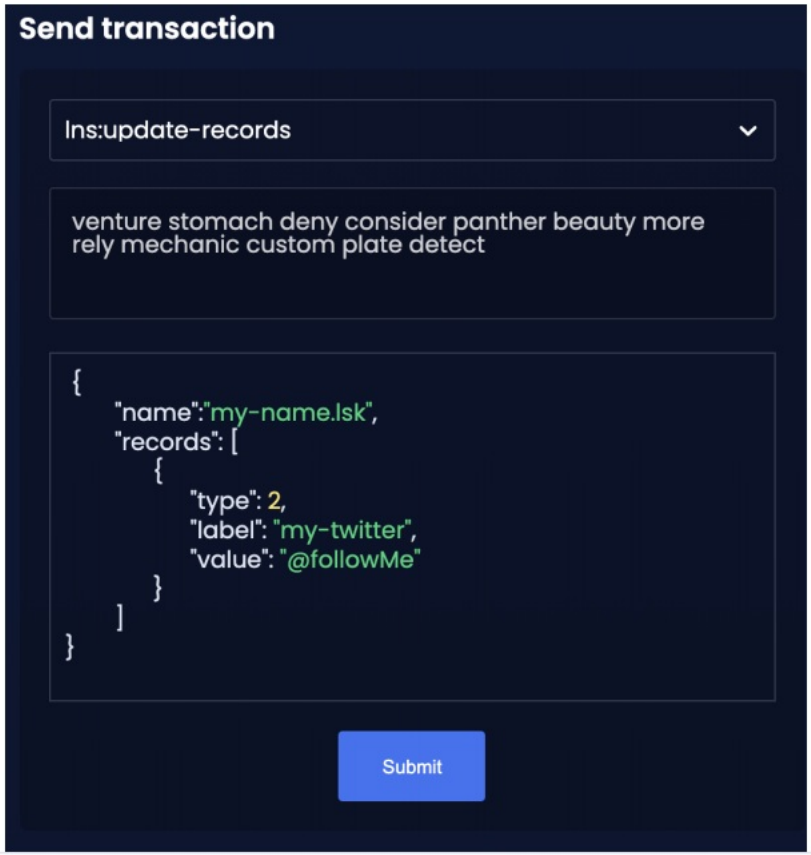
4.3.5. Define a reverse lookup address

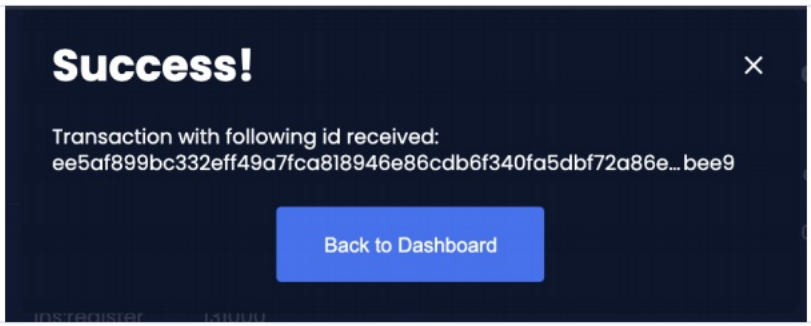



Send transaction
lns:reverse-lookup
venture stomach deny consider panther beauty more rely mechanic custom plate detect
{
"name":"my-name.lsk"
}
Submit

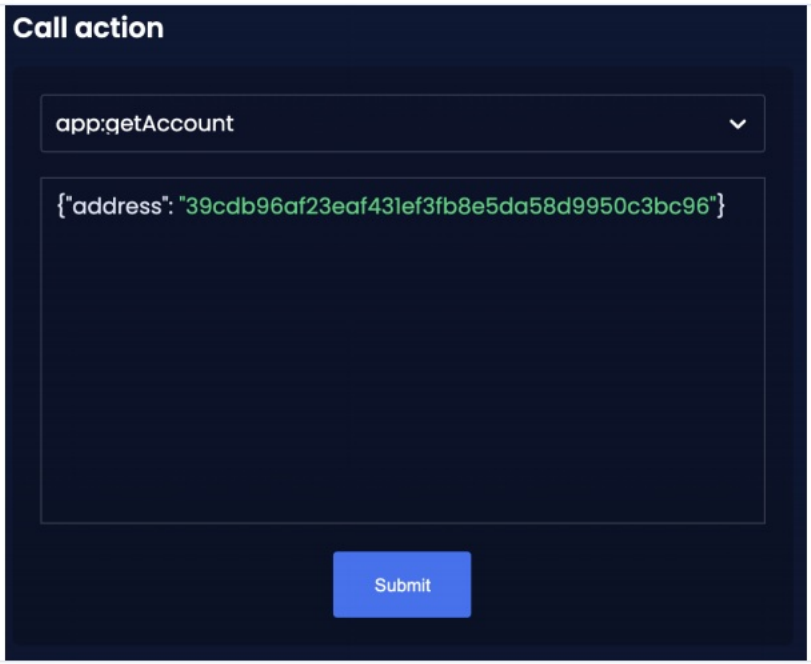

Success!
Transaction with following id received:
1c7426cb6345924066294fa52965958c41361a40c64c98998d9cb05d1...a8fe
Back to Dashboard

4.3.6. Update the records for a domain name


Send transaction
lns:update-records
venture stomach deny consider panther beauty more rely mechanic custom plate detect
{
"name":"my-name.lsk",
"records": [
{
"type": 2,
"label": "my-twitter",
"value": "@followMe"
}
]
}
Submit

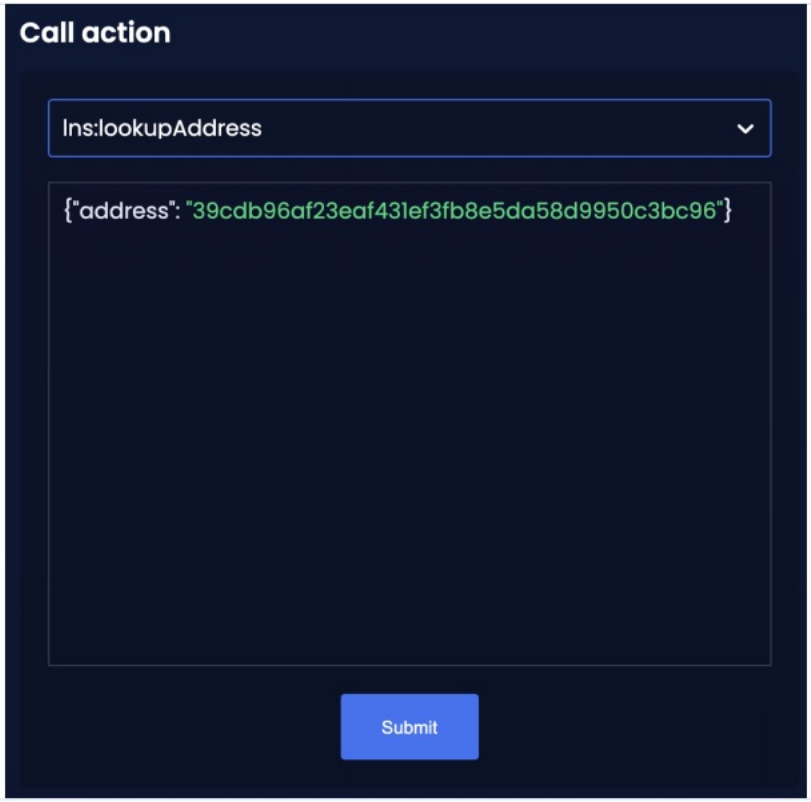

Success!
Transaction with following id received:
ee5af899bc332eff49a7fca818946e86cdb6f340fa5dbf72a86e... bee9
Back to Dashboard

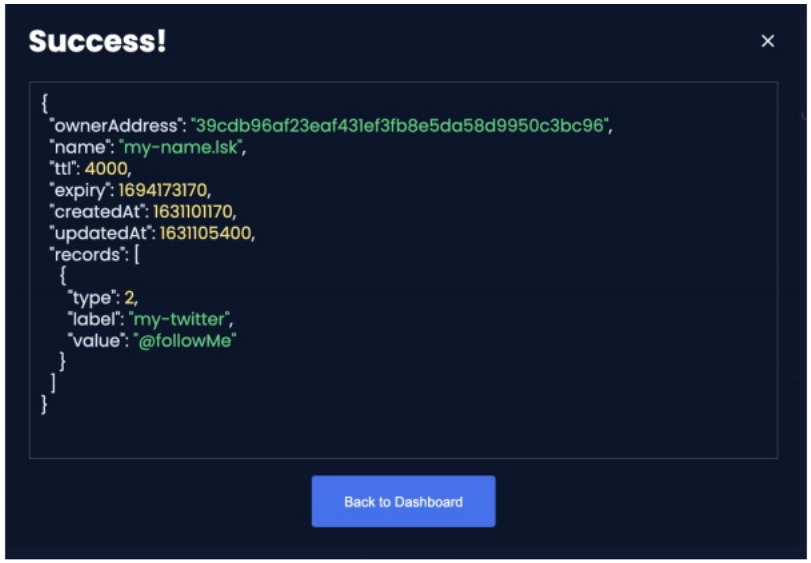
4.3.7. Check account details




Success!
{
"address": "39cdb96af23eaf431ef3fb8e5da58d9950c3bc96",
"token": {
"balance": "9999743000"
},
"sequence": {
"nonce": "2"
},
"keys": {
"numberOfSignatures": 0,
"mandatoryKeys": [],
"optionalKeys": []
},
"dpos": {
"delegate": {
"username": "",
"pomHeights": [],
"consecutiveMissedBlocks": 0,
"lastForgedHeight": 0,
"isBanned": false,
"totalVotesReceived": "0"
},
"sentVotes": [],
"unlocking": []
},
"lns": {
"reverseLookup": "43c2a7ff237fade73716ae30b7c703b1f20fa91f7ba3e0437a2ace198a18dbe8",
"ownNodes": [
"43c2a7ff237fade73716ae30b7c703b1f20fa91f7ba3e0437a2ace198a18dbe8"
]
}
}
Back to Dashboard

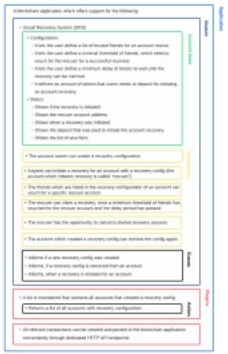
4.3.8. Perform a reverse lookup for an address

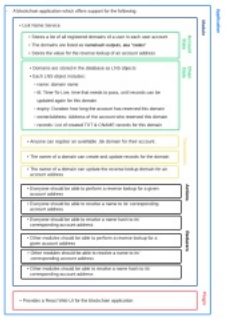
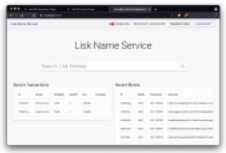



Success!
{
"ownerAddress": "39cdb96af23eaf431ef3fb8e5da58d9950c3bc96",
"name": "my-name.lsk",
"ttl": 4000,
"expiry": 1694173170,
"createdAt": 1631101170,
"updatedAt": 1631105400,
"records": [
{
"type": 2,
"label": "my-twitter",
"value": "@followMe"
}
]
}
Back to Dashboard

To become both familiar and fully conversant with the Lisk SDK, a step-by-step walk through guide is provided that simplifies the process of developing a proof of concept blockchain application.

Name	Estimated time required	Complexity	Content
Hello World 	~1-2 h	Beginner	Follow the development guides to learn how to perform the following: • ...bootstrap a simple blockchain application. • ...create a custom module. • ...create a custom plugin. • ...configure a blockchain application. • ...connect a dashboard to the application. • ...write unit tests modules and assets.
Non-fungible token (NFT) Tutorial 	~2-3 h	Beginner	Learn how to create: • ...a blockchain application that allows a user to create, transfer and purchase NFTs. • ...3 different transaction assets to create, transfer, and purchase NFTs. • ...a custom plugin which adds a new HTTP API that serves NFT-related data to the application. • ...a frontend application that allows the user to utilize the blockchain application in the browser.

Name	Estimated time required	Complexity	Content
Social Recovery System (SRS) Tutorial  	~4 h	Intermediate	Learn how to create: • ...a blockchain application that offers users the opportunity to recover their accounts if they have lost their credentials through a social recovery system. • ...6 different transaction assets to manage the recovery process. • ...2 custom plugins that provide new actions and API endpoints which are helpful in the frontend. • ...a frontend application that allows the user to utilize the blockchain application in the browser.

Name	Estimated time required	Complexity	Content
Lisk Name Service (LNS) Tutorial  	~4 h	Intermediate	Learn how to: • ...create a blockchain application with a module which provides a name service for Lisk addresses. The Lisk Name Service allows users to register domain names for their accounts. It can then resolve the human readable domain names to their corresponding account address, making the domain name a human readable alias for the address. • ...create three different transaction assets: ◦ Register: To register a new domain name. ◦ Reverse Lookup: To define the reverse lookup for an account address. ◦ Update Records: To update the records of a domain name. • ...connect the Dashboard plugin to interact with the LNS app during development. • ...create a plugin that provides a React.js frontend for the LNS blockchain application. • ...extend the LNS application CLI with additional commands. • ...write unit tests for the newly implemented transactions assets. • ...write functional tests for the LNS module.

Extending the application

To further enhance the LNS blockchain application, create two new LNS specific commands which can be executed directly from the command line:

- lns:resolve Command to resolve a provided domain name to an account address.
- lns:lookup: Command to perform a reverse lookup for a provided account address. Returns the default domain name of an account.

The application CLI already contains numerous general commands by default. They are directly created when the application is bootstrapped using Lisk Commander with .

An overview of all existing CLI commands can be seen by navigating to the root folder of the blockchain application, and running the following command:

This will return the command reference for the application CLI:

We already used the CLI in this tutorial to start the LNS application. Now, the plan is to create a new topic , and to define the two new commands and to be part of it.

Navigate to and create a new folder , which will contain the files for the new commands.

Tip

The application CLI commands are based on OCLIF.

Check out their documentation to get a deeper understanding on how the CLI commands are constructed.

lns:resolve

Create a new file and import the from Lisk Commander.

Create a new class , which extends the class BaseIPCClientCommand of Lisk Commander.

The already contains two optional default flags for the command: and . It also contains an API client to the node which will be used to invoke actions in the LNS application via the CLI.

The only method required for a new command is the function.

In this particular use case, one argument is needed, the name to resolve.

You may also add some examples about the usage of the command, which will be added to the auto-generated command reference.

This is all the code required to add the new command to the LNS blockchain application.

As can be seen, there is not much logic required to be implemented here. The action lns:resolveName can be reused to get the address, based on the address that was provided as the command argument.

lns:lookup

Create a new file [redacted] and paste the code snippet below. This is all the code required to add the [redacted] command to the LNS blockchain application.

The implementation is analog to the previous command, nevertheless, now use the corresponding action [redacted] to get the domain name for the provided account address.

Trying out the new CLI commands

Display the CLI reference once again. The new topic should now appear under :

If the LNS application is not already running, start it again:

Now resolve the domain name , which was registered previously.

This will return the corresponding LNS object:

When the resolve command works as expected, copy the from the returned LNS object and provide it as an argument for the command:

This will return the corresponding LNS object:

Note

Each account can register multiple domain names to their account. The address lookup returns a different object for the provided address, because the account has set to be the default domain name for this account.

How to display the topic reference:

How to display the command reference:

How to create a blockchain application that supports NFTs. You will learn how to create a custom module that adds the feature to create and purchase NFTs in a blockchain application, and a custom plugin that provides additional NFT-related features.

Important

The example app for this tutorial was build without bootstrapping via Lisk Commander, therefore the project setup and file structure are different in this tutorial. If you wish to see an example how to easily bootstrap a blockchain application with the Lisk Commander, check out how to bootstrap the Hello World application.

Tip

For the **full code example** please see the NFT app on Github.

Table of Contents

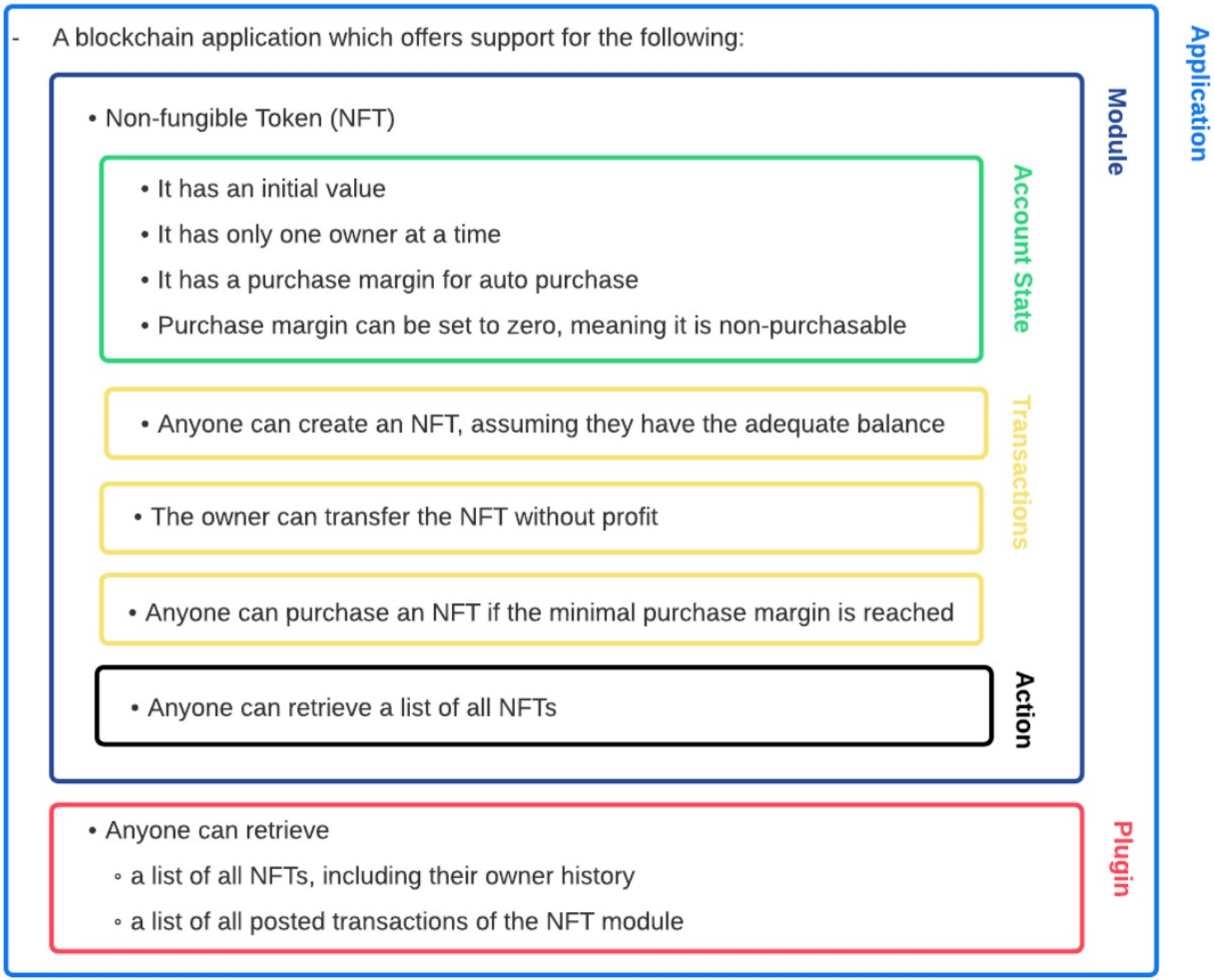
Fungible vs non-fungible tokens

	Fungible tokens	**Non-fungible tokens**
Description	Tokens that can be readily interchanged with each other.	Each token has unique characteristics which make it special.
Characteristics	• Not unique. • Divisible. • Mutually interchangeable.	• Unique. • Indivisible. • Not mutually interchangeable.

	Fungible tokens	Non-fungible tokens
Examples	• Real world: Fiat currencies like the Dollar($) or Euro(€). • Digital tokens: Bitcoin(BTC) or Lisk(LSK).	• Real world: Paintings or music. • Digital tokens: Crypto collectibles (e.g. Cryptokitties), event tickets, or other virtual assets.

NFT blockchain overview

In our example, we will design the following characteristics for the blockchain application:



As can be seen in the above image, it is necessary to create the following components:

1. The NFT module (on-chain).
2. Three new transaction assets for the NFT module.
3. The NFT plugin (off-chain).

In addition to the blockchain application, we will also implement a **frontend application**, which allows us to interact with the blockchain application through a UI (User Interface), in the browser.

1. Project setup

Create a new folder, which will contain all the files for the NFT app:

```
mkdir nft
mkdir nft/blockchain_app
cd nft/blockchain_app
npm init
npm i lisk-sdk
```

Next, create a new file `index.js` then copy and paste the following:

nft/blockchain_app/index.js

```
const { Application, genesisBlockDevnet, configDevnet } = require('lisk-sdk');

const app = Application.defaultApplication(genesisBlockDevnet, configDevnet);

app
	.run()
	.then(() => app.logger.info('App started...'))
	.catch(error => {
		console.error('Faced error in application', error);
		process.exit(1);
	});
```

This code snippet creates a default blockchain application, which is configured for development purposes. We will use this app as a basis for the NFT app and extend it with a module and a plugin in the following steps, to suit the desired use case.

Create a new folder `nft_module/` and inside a new file `index.js`:

nft/blockchain_app/

```
mkdir nft_module
```

2. NFT related functions

For a better overview, create a new file `nft.js` in the `nft_module/` folder.

Now open `nft.js` and define the functions related to the NFT, which will be used in the assets and the module in the next steps.

The following functions are implemtented:

- `registeredNFTTokensSchema`: The schema, how the NFTs are saved in the database. Each NFT has the following properties:
  - id: The unique ID of the NFT.
  - value: The value of the NFT.
  - ownerAddress: The address of the owner of the NFT.
  - minPurchaseMargin: The minimum price increase, that should happen with every purchase of the NFT in percentage.
  - name: The NFT name.
- `CHAIN_STATE_NFT_TOKENS`: The key under which the NFTs are saved in the database.
- `createNFTToken()`: Creates a new NFT based on the provided `name`, `ownerAddress`, `nonce`, `value`, and `minPurchaseMargin`. The ID for the NFT is created by combining and hashing the owner address and its' current nonce, which in combination always creates a unique value. This way it is ensured that each newly created NFT has a unique ID in the database.
- `setAllNFTTokens()`: Saves provided NFTs to the database.
- `getAllNFTTokens()`: Retrieves NFTs from the database.
- `getAllNFTTokensAsJSON()`: Returns all NFTs from the database as JSON.

Note

You may have noticed that we use special parameters in the functions, such as `stateStore` and `_dataAccess`. These variables are available in the module and are explained in more detail in the coming sections The apply function and Actions.

nft/blockchain_app/nft_module/nft.js

```
const { codec, cryptography } = require("lisk-sdk");

const registeredNFTTokensSchema = {
  $id: "lisk/nft/registeredTokens",
  type: "object",
  required: ["registeredNFTTokens"],
  properties: {
    registeredNFTTokens: {
      type: "array",
      fieldNumber: 1,
      items: {
        type: "object",
        required: ["id", "value", "ownerAddress", "minPurchaseMargin", "name"],
        properties: {
          id: {
            dataType: "bytes",
            fieldNumber: 1,
          },
          value: {
            dataType: "uint64",
            fieldNumber: 2,
          },
          ownerAddress: {
            dataType: "bytes",
            fieldNumber: 3,
          },
          minPurchaseMargin: {
            dataType: "uint32",
            fieldNumber: 4,
          },
          name: {
```

```
          name: {
            dataType: "string",
            fieldNumber: 5,
          },
        },
      },
    },
  },
};

const CHAIN_STATE_NFT_TOKENS = "nft:registeredNFTTokens";

const createNFTToken = ({ name, ownerAddress, nonce, value, minPurchaseMargin }) => {
  const nonceBuffer = Buffer.alloc(8);
  nonceBuffer.writeBigInt64LE(nonce);
  // Create a unique seed by using a combination of the owner account address and the current nonce of
the account.
  const seed = Buffer.concat([ownerAddress, nonceBuffer]);
  const id = cryptography.hash(seed);

  return {
    id,
    minPurchaseMargin,
    name,
    ownerAddress,
    value,
  };
};

const getAllNFTTokens = async (stateStore) => {
  const registeredTokensBuffer = await stateStore.chain.get(
    CHAIN_STATE_NFT_TOKENS
  );
  if (!registeredTokensBuffer) {
    return [];
  }

  const registeredTokens = codec.decode(
    registeredNFTTokensSchema,
    registeredTokensBuffer
  );
```

```
  );

  return registeredTokens.registeredNFTTokens;
};

const getAllNFTTokensAsJSON = async (dataAccess) => {
  const registeredTokensBuffer = await dataAccess.getChainState(
    CHAIN_STATE_NFT_TOKENS
  );

  if (!registeredTokensBuffer) {
    return [];
  }

  const registeredTokens = codec.decode(
    registeredNFTTokensSchema,
    registeredTokensBuffer
  );

  return codec.toJSON(registeredNFTTokensSchema, registeredTokens)
    .registeredNFTTokens;
};

const setAllNFTTokens = async (stateStore, NFTTokens) => {
  const registeredTokens = {
    registeredNFTTokens: NFTTokens.sort((a, b) => a.id.compare(b.id)),
  };

  await stateStore.chain.set(
    CHAIN_STATE_NFT_TOKENS,
    codec.encode(registeredNFTTokensSchema, registeredTokens)
  );
};

module.exports = {
  registeredNFTTokensSchema,
  CHAIN_STATE_NFT_TOKENS,
  getAllNFTTokens,
  setAllNFTTokens,
  getAllNFTTokensAsJSON,
  createNFTToken
```

```
};
```

3. Transaction assets

The user shall have the ability to create, purchase, and transfer NFTs in the network. To do this, we create the corresponding transaction assets for the NFT module. These transaction assets each define both the asset schema for the transaction data, and the logic, which determine how this data is applied and stored in the database.

nft/blockchain_app/nft_module/

```
mkdir transactions ①
cd transactions/
```

① Create a new folder `transactions/`, which will contain the files for the transaction assets.

3.1. CreateNFT asset

Create a new file `create_nft_asset.js` inside the newly created `transactions/` folder.

Now open the file and copy and paste the code below:

nft/blockchain_app/nft_module/transactions/create_nft_asset.js

```
const { BaseAsset } = require("lisk-sdk");

// extend base asset to implement your custom asset
class CreateNFTAsset extends BaseAsset { ①

}

module.exports = CreateNFTAsset; ②
```

① Extend from the base asset to implement a custom asset.

② Export the asset, so it can be imported later into the custom module.

Now define all required properties for the transaction asset one after another.

3.1.1. Asset ID and name

nft/blockchain_app/nft_module/transactions/create_nft_asset.js

```
const { BaseAsset } = require("lisk-sdk");

// extend base asset to implement your custom asset
class CreateNFTAsset extends BaseAsset {
  // define unique asset name and id
  name = "createNFT"; ①
  id = 0; ②
}

module.exports = CreateNFTAsset;
```

① Set the asset name to `"createNFT"`.

② Set the asset id to `0`.

3.1.2. Asset schema

The asset schema describes the required datatypes and the structure of the data in the respective transaction asset.

> **Tip**
>
> For more information how schemas are used in the application, check out the Codec & schema reference.

For creating a new NFT, we require the following information:

- Name: The name of the NFT.
- Initial value: The inital value of the NFT.
- Minimal purchase margin: The % value of the initial value, that is added to the initial value when purchasing the NFT.

Therefore, create the schema as described below:

nft/blockchain_app/nft_module/transactions/create_nft_asset.js

```
const { BaseAsset } = require("lisk-sdk");
```

```
// extend base asset to implement your custom asset
class CreateNFTAsset extends BaseAsset {
  // define unique asset name and id
  name = "createNFT";
  id = 0;
  // define asset schema for serialization
  schema = {
    $id: "lisk/nft/create",
    type: "object",
    required: ["minPurchaseMargin", "initValue", "name"],
    properties: {
      minPurchaseMargin: {
        dataType: "uint32",
        fieldNumber: 1,
      },
      initValue: {
        dataType: "uint64",
        fieldNumber: 2,
      },
      name: {
        dataType: "string",
        fieldNumber: 3,
      },
    },
  };
}

module.exports = CreateNFTAsset;
```

Now that the `schema` defines what data to expect, we can use the `validate()` function to validate the received data of the transaction asset before applying it.

3.1.3. The `validate()` function

Before the data in the transaction asset is applied in the next step, use the `validate()` function to verify the correctness of the submitted data.

The `validate()` function has access to:

- `asset`: the posted transaction asset.

nft/blockchain_app/nft_module/transactions/create_nft_asset.js

```
const { BaseAsset } = require("lisk-sdk");

// extend base asset to implement your custom asset
class CreateNFTAsset extends BaseAsset {
  // define unique asset name and id
  name = "createNFT";
  id = 0;
  // define asset schema for serialization
  schema = {
    $id: "lisk/nft/create",
    type: "object",
    required: ["minPurchaseMargin", "initValue", "name"],
    properties: {
      minPurchaseMargin: {
        dataType: "uint32",
        fieldNumber: 1,
      },
      initValue: {
        dataType: "uint64",
        fieldNumber: 2,
      },
      name: {
        dataType: "string",
        fieldNumber: 3,
      },
    },
  };

  // verify data in the transaction asset
  validate({asset}) {
    if (asset.initValue <= 0) { ①
      throw new Error("NFT init value is too low.");
    } else if (asset.minPurchaseMargin < 0 || asset.minPurchaseMargin > 100) { ②
      throw new Error("The NFT minimum purchase value needs to be between 0-100.");
    }
  };
```

```
}


module.exports = CreateNFTAsset;
```

① Verifies that the initial value of the NFT is greater than 0. If not, the transaction is not applied, and a corresponding error is thrown.

② Verifies that the minimal purchase margin is a value between 0 and 100.

3.1.4. The apply function

The `apply()` function has access to:

- `asset`: the posted transaction asset.
- `stateStore`: The state store is a data structure that holds a temporary state while processing a block. It is used here to get and set certain data from and to the database.
- `reducerHandler`: Allows the user to utilize reducer functions of other modules inside the `apply()` function.
- `transaction`: the complete transaction object.

nft/blockchain_app/nft_module/transactions/create_nft_asset.js

```
const { BaseAsset } = require("lisk-sdk");
const {
  getAllNFTTokens,
  setAllNFTTokens,
  createNFTToken,
} = require("../nft"); ①

// extend base asset to implement your custom asset
class CreateNFTAsset extends BaseAsset {
  // define unique asset name and id
  name = "createNFT";
  id = 0;
  // define asset schema for serialization
  schema = {
    $id: "lisk/nft/create",
    type: "object",
    required: ["minPurchaseMargin", "initValue", "name"],
    properties: {
      minPurchaseMargin: {
```

```
      minPurchaseMargin: {
        dataType: "uint32",
        fieldNumber: 1,
      },
      initValue: {
        dataType: "uint64",
        fieldNumber: 2,
      },
      name: {
        dataType: "string",
        fieldNumber: 3,
      },
    },
  };

  // verify data in the transaction asset
  validate({asset}) {
    if (asset.initValue <= 0) {
      throw new Error("The NFT init value is too low.");
    } else if (asset.minPurchaseMargin < 0 || asset.minPurchaseMargin > 100) {
      throw new Error("The NFT minimum purchase value needs to be between 0-100.");
    }
  };

  async apply({ asset, stateStore, reducerHandler, transaction }) {
    // create NFT ②
    const senderAddress = transaction.senderAddress;
    const senderAccount = await stateStore.account.get(senderAddress);
    const nftToken = createNFTToken({
      name: asset.name,
      ownerAddress: senderAddress,
      nonce: transaction.nonce,
      value: asset.initValue,
      minPurchaseMargin: asset.minPurchaseMargin,
    });

    // update sender account with unique NFT ID ③
    senderAccount.nft.ownNFTs.push(nftToken.id);
    await stateStore.account.set(senderAddress, senderAccount);

    // debit tokens from sender account to create an NFT ④
```

```
    await reducerHandler.invoke("token:debit", {
      address: senderAddress,
      amount: asset.initValue,
    });

    // save NFTs ⑤
    const allTokens = await getAllNFTTokens(stateStore);
    allTokens.push(nftToken);
    await setAllNFTTokens(stateStore, allTokens);
  }
}

module.exports = CreateNFTAsset;
```

① Import `getAllNFTTokens`, `setAllNFTTokens` and `createNFTToken` from the `nft.js` file.

② Create the NFT based on the asset data and the sender address of the transaction.

③ Push the ID of the newly created NFT into the sender account and save the updated sender account in the database.

④ Debit the initial value of the NFT from the sender account.

⑤ Push the newly created NFT into the list of all NFTs and save it in the database.

3.2. PurchaseNFT asset

Create a new file `purchase_nft_asset.js` inside the `transactions/` folder.

Analog to the implementation of the `createNFT` asset, create the `purchaseNFT` asset by pasting the snippet below.

Note

The validation of the asset inputs is done in the `apply()` function, as it is necessary to access the database in order to validate the transaction inputs.

nft/blockchain_app/nft_module/transactions/purchase_nft_asset.js

```
const { BaseAsset } = require("lisk-sdk");
const { getAllNFTTokens, setAllNFTTokens } = require("../nft");

// extend base asset to implement your custom asset
```

```
// extend base asset to implement your custom asset
class PurchaseNFTAsset extends BaseAsset {
  // define unique asset name and id
  name = "purchaseNFT";
  id = 1;
  // define asset schema for serialization
  schema = {
    $id: "lisk/nft/purchase",
    type: "object",
    required: ["nftId", "purchaseValue", "name"],
    properties: {
      nftId: {
        dataType: "bytes",
        fieldNumber: 1,
      },
      purchaseValue: {
        dataType: "uint64",
        fieldNumber: 2,
      },
      name: {
        dataType: "string",
        fieldNumber: 3,
      },
    },
  };

  async apply({ asset, stateStore, reducerHandler, transaction }) {
    // verify if purchasing nft exists ①
    const nftTokens = await getAllNFTTokens(stateStore);
    const nftTokenIndex = nftTokens.findIndex((t) => t.id.equals(asset.nftId));

    if (nftTokenIndex < 0) {
      throw new Error("Token id not found");
    }
    // verify if minimum nft purchasing condition met ②
    const token = nftTokens[nftTokenIndex];
    const tokenOwner = await stateStore.account.get(token.ownerAddress);
    const tokenOwnerAddress = tokenOwner.address;

    if (token && token.minPurchaseMargin === 0) {
      throw new Error("This NFT can not be purchased");
```

```
    }

    const tokenCurrentValue = token.value;
    const tokenMinPurchaseValue =
      tokenCurrentValue +
      (tokenCurrentValue * BigInt(token.minPurchaseMargin)) / BigInt(100);
    const purchaseValue = asset.purchaseValue;

    if (tokenMinPurchaseValue > purchaseValue) {
      throw new Error("Token can not be purchased. Purchase value is too low. Minimum value: " +
tokenMinPurchaseValue);
    }

    // remove nft from owner account ③
    const purchaserAddress = transaction.senderAddress;
    const purchaserAccount = await stateStore.account.get(purchaserAddress);

    const ownerTokenIndex = tokenOwner.nft.ownNFTs.findIndex((a) =>
      a.equals(token.id)
    );
    tokenOwner.nft.ownNFTs.splice(ownerTokenIndex, 1);
    await stateStore.account.set(tokenOwnerAddress, tokenOwner);

    // add nft to purchaser account ④
    purchaserAccount.nft.ownNFTs.push(token.id);
    await stateStore.account.set(purchaserAddress, purchaserAccount);

    token.ownerAddress = purchaserAddress;
    token.value = purchaseValue;
    nftTokens[nftTokenIndex] = token;
    await setAllNFTTokens(stateStore, nftTokens);

    // debit LSK tokens from purchaser account ⑤
    await reducerHandler.invoke("token:debit", {
      address: purchaserAddress,
      amount: purchaseValue,
    });

    // credit LSK tokens to purchaser account ⑥
    await reducerHandler.invoke("token:credit", {
```

```
      address: tokenOwnerAddress,
      amount: purchaseValue,
    });
  }
}

module.exports = PurchaseNFTAsset;
```

① Verify, that the NFT which is purchased exists in the database. To do this, we request all NFTs with `getAllNFTTokens()` and search inside the returned list for the desired NFT ID. If no NFT is found, a corresponding error is thrown.

② If the NFT was found, it is retrieved from the database, and the minimum purchase value of the token is compared to the purchase value in the transaction asset. If the purchase value in the transaction asset is equal or higher than the minimal purchase value of the NFT, the NFT can be purchased. Otherwise, an error will be thrown.

③ Remove the NFT from the current owner account. The `StateStore` is used here to retrieve the owner account data from the database and later to update the owner account in the database, after the token ID has been removed from their owned tokens.

④ Add the NFT to the account of the purchaser. The `StateStore` is used again to update the purchaser account in the database, after the token ID has been added to their owned tokens.

⑤ Debit the purchase value from the purchasers account. We use the `reducerHandler` here and invoke `token:debit` from Token module which allows us to conveniently debit tokens from an account.

⑥ Credit the purchase value to the owners account. We use the `reducerHandler` here and invoke `token:debit` from Token module which allows us to conveniently credit tokens to an account.

3.3. TransferNFT asset

The last transaction asset that we want to implement in this tutorial is the `transferNFT` transaction.

Create a new file `transfer_nft_asset.js` inside the `transactions/` folder. Create the `transferNFT` asset by pasting the snippet below.

nft/blockchain_app/nft_module/transactions/transfer_nft_asset.js

```
const { BaseAsset } = require("lisk-sdk");
const { getAllNFTTokens, setAllNFTTokens } = require("../nft");

// 1.extend base asset to implement your custom asset
class TransferNFTAsset extends BaseAsset {
  // 2.define unique asset name and id
  name = "transferNFT";
```

```
  id = 2;

  // 3.define asset schema for serialization
  schema = {
    $id: "lisk/nft/transfer",
    type: "object",
    required: ["nftId", "recipient"],
    properties: {
      nftId: {
        dataType: "bytes",
        fieldNumber: 1,
      },
      recipient: {
        dataType: "bytes",
        fieldNumber: 2,
      },
      name: {
        dataType: "string",
        fieldNumber: 3,
      },
    },
  };

  async apply({ asset, stateStore, transaction }) {
    const nftTokens = await getAllNFTTokens(stateStore);
    const nftTokenIndex = nftTokens.findIndex((t) => t.id.equals(asset.nftId));

    // 4.verify if the nft exists ①
    if (nftTokenIndex < 0) {
      throw new Error("Token id not found");
    }
    const token = nftTokens[nftTokenIndex];
    const tokenOwnerAddress = token.ownerAddress;
    const senderAddress = transaction.senderAddress;
    // 5.verify that the sender owns the nft ②

    if (!tokenOwnerAddress.equals(senderAddress)) {
      throw new Error("An NFT can only be transferred by the owner of the NFT.");
    }

    const tokenOwner = await stateStore.account.get(tokenOwnerAddress);
```

```
    // 6.remove nft from the owner account ③
    const ownerTokenIndex = tokenOwner.nft.ownNFTs.findIndex((a) =>
      a.equals(token.id)
    );
    tokenOwner.nft.ownNFTs.splice(ownerTokenIndex, 1);
    await stateStore.account.set(tokenOwnerAddress, tokenOwner);

    // 7.add nft to the recipient account ④
    const recipientAddress = asset.recipient;
    const recipientAccount = await stateStore.account.get(recipientAddress);
    recipientAccount.nft.ownNFTs.push(token.id);
    await stateStore.account.set(recipientAddress, recipientAccount);

    token.ownerAddress = recipientAddress;
    nftTokens[nftTokenIndex] = token;
    await setAllNFTTokens(stateStore, nftTokens);
  }
}

module.exports = TransferNFTAsset;
```

① Verify, that the NFT which is purchased exists in the database. To do this, we request all NFTs with `getAllNFTTokens()` and search inside the returned list for the desired NFT ID. If no NFT is found, a corresponding error is thrown.

② Verify, that the account who wants to transfer the NFT actually owns the NFT.

③ Remove the NFT from the current owner account. The `StateStore` is used here to retrieve the owner account data from the database and later to update the owner account in the database, after the token ID has been removed from their owned tokens.

④ Add the NFT to the account of the recipient. The `StateStore` is used again to update the recipient account in the database, after the token ID has been added to their owned tokens.

4. The NFT module

Inside the `nft_module/` folder, create a new file `index.js`.

Open `index.js` and create the skeleton, which will contain all parts of the NFT module:

nft/blockchain_app/nft_module/index.js

```
const { BaseModule } = require("lisk-sdk");
```

```
const { BaseModule } = require("lisk-sdk");

class NFTModule extends BaseModule { ①

}

module.exports = { NFTModule }; ②
```

① Extend from the base module to implement a custom module.

② Export the module, so it can be imported into the application later.

Now define all required properties for the module one after another.

4.1. Module ID and name

It is recommended to start with the easiest ones: defining the module name and ID.

nft/blockchain_app/nft_module/index.js

```
const { BaseModule } = require("lisk-sdk");

class NFTModule extends BaseModule {
  public name = "nft"; ①
  public id = 1024; ②
}

module.exports = { NFTModule };
```

① Set the module name to `"nft"`.

② Set the module id to `1024`.

The module ID has to be unique within the network. The minimum value for it is `1000`, as the other IDs are reserved for future default modules of the Lisk SDK. If the module ID is not unique, it will cause forks in the network.

The module name should be unique within the network as well, otherwise, it will lead to confusion. For example, when subscribing to events or invoking actions of that module.

4.2. The account schema

Next, define the account schema. This defines the properties that are added to each network account by the module if it is registered with the application later.

Here, we expect each account to have a property `ownNFTs`, which is an array of NFTs which the account owns. By default, it is empty.

nft/blockchain_app/nft_module/index.js

```
const { BaseModule } = require("lisk-sdk");

// Extend base module to implement your custom module
class NFTModule extends BaseModule {
  public name = "nft";
  public id = 1024;
  public accountSchema = {
    type: "object",
    required: ["ownNFTs"],
    properties: {
      ownNFTs: {
        type: "array",
        fieldNumber: 1,
        items: {
          dataType: "bytes",
        },
      },
    },
    default: {
      ownNFTs: [],
    },
  };
}

module.exports = { NFTModule };
```

4.3. Importing transaction assets into the module

Now, import the transactions which were created in section 2: Transaction assets into the module.

Add them to the `transactionAssets` property as shown in the snippet below.

Tip

Best practice

It is good practice to name the imported transaction assets after their corresponding classname.

In this example: `CreateNFTAsset`, `PurchaseNFTAsset`, and `TransferNFTAsset`.

nft/blockchain_app/nft_module/index.js

```
const { BaseModule } = require("lisk-sdk");

const CreateNFTAsset = require("./transactions/create_nft_asset");
const PurchaseNFTAsset = require("./transactions/purchase_nft_asset");
const TransferNFTAsset = require("./transactions/transfer_nft_asset");

// Extend base module to implement your custom module
class NFTModule extends BaseModule {
  public name = "nft";
  public id = 1024;
  public accountSchema = {
    type: "object",
    required: ["ownNFTs"],
    properties: {
      ownNFTs: {
        type: "array",
        fieldNumber: 4,
        items: {
          dataType: "bytes",
        },
      },
    },
    default: {
      ownNFTs: [],
    },
  };
  // Add the transaction assets to the module
  public transactionAssets = [new CreateNFTAsset(), new PurchaseNFTAsset(), new TransferNFTAsset()];
}
```

```
module.exports = { NFTModule };
```

4.4. Actions

The last remaining feature for the NFT module consists of creating the following action `getAllNFTTokens` which will allow the NFT plugin to retrieve a list of all existing NFT tokens later.

To implement it, we use the function `getAllNFTTokensAsJSON()`, which has been described in the section NFT related functions.

The variable `this._dataAccess` is passed as a parameter. This variable is available throughout the module and is used in the `actions` to access data from the database.

nft/blockchain_app/nft_module/index.js

```
const { BaseModule } = require("lisk-sdk");
const { getAllNFTTokensAsJSON } = require("./nft");

const CreateNFTAsset = require("./transactions/create_nft_asset");
const PurchaseNFTAsset = require("./transactions/purchase_nft_asset");
const TransferNFTAsset = require("./transactions/transfer_nft_asset");

// Extend from the base module to implement the NFT module
class NFTModule extends BaseModule {
  public name = "nft";
  public id = 1024;
  public accountSchema = {
    type: "object",
    required: ["ownNFTs"],
    properties: {
      ownNFTs: {
        type: "array",
        fieldNumber: 1,
        items: {
          dataType: "bytes",
        },
```

```
      },
    },
    default: {
      ownNFTs: [],
    },
  };
  public transactionAssets = [new CreateNFTAsset(), new PurchaseNFTAsset(), new TransferNFTAsset()];
  public actions = {
    // get all the registered NFT tokens from blockchain
    getAllNFTTokens: async () => getAllNFTTokensAsJSON(this._dataAccess),
  };
}

module.exports = { NFTModule };
```

5. The NFT plugin

Now that all on-chain logic for the NFTs is defined in the NFT module, let's add a corresponding NFT plugin, which will handle the off-chain logic for the NFT app.

The NFT plugin shall provide an HTTP API that offers new endpoints for NFT related data from the blockchain.

Navigate out of the `nft_module` folder back into the `blockchain_app` folder.

If you haven't done it yet, now add an author to your `package.json` file. This information will be used later in the plugin.

nft/blockchain_app/

```
vim package.json
```

Press `i` to switch to the insert mode.

Set the author name as a string of your choice and exit the insert mode by pressing `Esc`.

Save and exit the file by pressing `:wq` and `Enter`.

Create a new folder `nft_api_plugin/`.

Inside the `nft_api_plugin/` folder, create a new file `index.js`.

nft/blockchain_app/

```
mkdir nft_api_plugin
touch nft_api_plugin/index.js
```

Open `index.js` and create the skeleton, which will contain all parts of the NFT plugin:

nft/blockchain_app/nft_api_plugin/index.js

```
const { BasePlugin } = require("lisk-sdk");
const pJSON = require("../package.json");

class NFTAPIPlugin extends BasePlugin { ①
  _server = undefined;
  _app = undefined;
  _channel = undefined;
  _db = undefined;
  _nodeInfo = undefined;

  static get alias() { ②
    return "NFTHttpApi";
  }

  static get info() { ③
    return {
      author: pJSON.author,
      version: pJSON.version,
      name: pJSON.name,
    };
  }

  get defaults() {
    return {};
  }

  get events() {
    return [];
```

```
  }

  get actions() {
    return {};
  }
}

module.exports = { NFTAPIPlugin }; ④
```

① Extend from the base plugin to implement a custom plugin.

② Set the alias for the plugin to `NFTHttpApi`.

③ Set the meta information for the plugin. Here, we use the data from the `package.json` file.

④ Export the plugin, so it can be imported into the application later.

5.1. Database related functions

For an enhanced overview, first, create another file that will contain the functions related to the plugin database. Here, we use a key-value store to save the data, similar to how the on-chain related data is saved in the default key-value store of the blockchain application.

Create a new file `db.js`. Copy and paste the below snippet into the newly created file `db.js`.

`db.js` contains various functions that take care of the following aspects:

- `getDBInstance(dataPath,dbName)`: Creates a new key-value store `nft_plugin.db` for the NFT plugin. The key-value store is used to store NFT related blockchain information of the plugin in a schema that makes it conveniently accessible for third party services.
- `saveTransactions(db, payload)`: Saves new transactions to the DB.
- `getAllTransactions(db, registeredSchema)`: Returns a list of all transactions from the DB.
- `getNFTHistory(db, dbKey)`: Returns the owner history of an NFT.
- `saveNFTHistory(db, decodedBlock, registeredModules)`: Filters for transactions of the NFT module in the decoded block. Updates the NFT history based on the found NFT transaction in the block. Saves the individual owner history for the corresponding NFTs in the database.

nft/blockchain_app/nft_api_plugin/db.js

```
const fs-extra = require("fs-extra");
```

```
const fs_extra = require('fs-extra');
const os = require("os");
const path = require("path");
const { cryptography, codec, db } = require("lisk-sdk");

const DB_KEY_TRANSACTIONS = "nft:transactions"; ①
const CREATENFT_ASSET_ID = 0;
const TRANSFERNFT_ASSET_ID = 2;

// Schemas
const encodedTransactionSchema = { ②
  $id: 'nft/encoded/transactions',
  type: 'object',
  required: ['transactions'],
  properties: {
    transactions: {
      type: 'array',
      fieldNumber: 1,
      items: {
        dataType: 'bytes',
      },
    },
  },
};

const encodedNFTHistorySchema = { ③
  $id: 'nft/encoded/nftHistory',
  type: 'object',
  required: ['nftHistory'],
  properties: {
    nftHistory: {
      type: 'array',
      fieldNumber: 1,
      items: {
        dataType: 'bytes',
      },
    },
  },
};

const getDBInstance = async (dataPath = '~/.lisk/nft-app/', dbName = 'nft_plugin.db') => {
```

```
  const dirPath = path.join(dataPath.replace('~', os.homedir()), 'plugins/data', dbName);
  await fs_extra.ensureDir(dirPath);
  return new db.KVStore(dirPath);
};

const saveTransactions = async (db, payload) => {
  const savedTransactions = await getTransactions(db);
  const transactions = [...savedTransactions, ...payload];
  const encodedTransactions = codec.encode(encodedTransactionSchema, { transactions });
  await db.put(DB_KEY_TRANSACTIONS, encodedTransactions);
};

const getTransactions = async (db) => {
  try {
    const encodedTransactions = await db.get(DB_KEY_TRANSACTIONS);
    const { transactions } = codec.decode(encodedTransactionSchema, encodedTransactions);
    return transactions;
  }
  catch (error) {
    return [];
  }
};

const getAllTransactions = async (db, registeredSchema) => {
  const savedTransactions = await getTransactions(db);
  const transactions = [];
  for (const trx of savedTransactions) {
    transactions.push(decodeTransaction(trx, registeredSchema));
  }
  return transactions;
};

const getNFTHistory = async (db, dbKey) => {
  try {
    const encodedNFTHistory = await db.get(dbKey);
    const { nftHistory } = codec.decode(encodedNFTHistorySchema, encodedNFTHistory);

    return nftHistory;
  }
  catch (error) {
```

```
    return [];
  }
};

const saveNFTHistory = async (db, decodedBlock, registeredModules, channel) => {
  decodedBlock.payload.map(async trx => {
    const module = registeredModules.find(m => m.id === trx.moduleID);
    if (module.name === 'nft') {
      let dbKey, savedHistory, base32Address, nftHistory, encodedNFTHistory;
      if (trx.assetID === CREATENFT_ASSET_ID){
        channel.invoke('nft:getAllNFTTokens').then(async (val) => {
          for (let i = 0; i < val.length; i++) {
            const senderAdress = cryptography.getAddressFromPublicKey(Buffer.from(trx.senderPublicKey,
'hex'));
            if (val[i].ownerAddress === senderAdress.toString('hex')) {
              dbKey = `nft:${val[i].id}`; ④
              savedHistory = await getNFTHistory(db, dbKey);
              if (savedHistory && savedHistory.length < 1) {
                base32Address =
cryptography.getBase32AddressFromPublicKey(Buffer.from(trx.senderPublicKey, 'hex'), 'lsk');
                nftHistory = [Buffer.from(base32Address, 'binary'), ...savedHistory];
                encodedNFTHistory = codec.encode(encodedNFTHistorySchema, { nftHistory });
                await db.put(dbKey, encodedNFTHistory);
              }
            }
          };
        });
      } else {
        dbKey = `nft:${trx.asset.nftId}`; ⑤
        base32Address = (trx.assetID === TRANSFERNFT_ASSET_ID) ?
cryptography.getBase32AddressFromAddress(Buffer.from(trx.asset.recipient, 'hex')) :
cryptography.getBase32AddressFromPublicKey(Buffer.from(trx.senderPublicKey, 'hex'), 'lsk');
        savedHistory = await getNFTHistory(db, dbKey);
        nftHistory = [Buffer.from(base32Address, 'binary'), ...savedHistory];
        encodedNFTHistory = codec.encode(encodedNFTHistorySchema, { nftHistory });
        await db.put(dbKey, encodedNFTHistory);
      }
    }
  });
};
```

```
};

const decodeTransaction = (
  encodedTransaction,
  registeredSchema,
) => {
  const transaction = codec.decode(registeredSchema.transaction, encodedTransaction);
  const assetSchema = getTransactionAssetSchema(transaction, registeredSchema);
  const asset = codec.decode(assetSchema, transaction.asset);
  const id = cryptography.hash(encodedTransaction);
  return {
    ...codec.toJSON(registeredSchema.transaction, transaction),
    asset: codec.toJSON(assetSchema, asset),
    id: id.toString('hex'),
  };
};

const getTransactionAssetSchema = (
  transaction,
  registeredSchema,
) => {
  const txAssetSchema = registeredSchema.transactionsAssets.find(
    assetSchema =>
      assetSchema.moduleID === transaction.moduleID && assetSchema.assetID === transaction.assetID,
  );
  if (!txAssetSchema) {
    throw new Error(
      // eslint-disable-next-line @typescript-eslint/restrict-template-expressions
      `ModuleID: ${transaction.moduleID} AssetID: ${transaction.assetID} is not registered.`,
    );
  }
  return txAssetSchema.schema;
};

module.exports = {
  getDBInstance,
  getAllTransactions,
  getTransactions,
  saveTransactions,
  saveNFTHistory,
  getNFTHistory
```

```
}
```

① `DB_KEY_TRANSACTIONS`: The key used to save the transactions in the key-value store.

② `encodedTransactionSchema`: The schema how the transactions will be saved in the key-value store. Here, we define a simple array which contains the encoded transactions.

③ `encodedNFTHistorySchema`: The schema how the owner history of an NFT is saved in the key-value store. Here, we define a simple array, which contains the addresses of the current and all previous owners of the NFT.

④ `dbKey`: The key we use to save the owner history of a newly created NFT in the key-value store.

⑤ `dbKey`: The same key as in <4>, but it can be retrieved directly from the transaction asset for an existing NFT in the key-value store.

5.2. The plugin logic

Now go back to `index.js` and implement the desired plugin logic.

The plugin shall provide the following additional data to the application:

- a list of all existing NFTs and their corresponding owner history.
- details of an NFT including the owner history, by NFT ID.
- a list of all transactions, including their module and asset IDs and the transaction asset.

To create this data, the plugin needs to listen to events for new blocks in the blockchain application. When a new block is created, the plugin checks if the block contains transactions and if so, saves them in the key-value store of the plugin. In case the block contains transactions of the NFT module, it will also update the owner history of the NFTs in the key-value store.

We will then create an HTTP server and also create the required API endpoints to serve the data to the frontend application.

This will all be defined inside of the `load()` function. All code inside the `load()` function is executed by the blockchain application when it loads the plugin.

The code in the `unload()` function is executed in complement by the blockchain application when it unloads the plugin.

nft/blockchain_app/nft_api_plugin/index.js

```
const express = require("express");
const cors = require("cors");
```

```
const { BasePlugin, codec } = require("lisk-sdk");

const pJSON = require("../package.json");
const { getDBInstance, getNFTHistory, getAllTransactions, saveNFTHistory, saveTransactions } =
require("./db");

// 1.plugin can be a daemon/HTTP/Websocket service for off-chain processing
class NFTAPIPlugin extends BasePlugin {
  _server = undefined;
  _app = undefined;
  _channel = undefined;
  _db = undefined;
  _nodeInfo = undefined;

  static get alias() {
    return "NFTHttpApi";
  }

  static get info() {
    return {
      author: pJSON.author,
      version: pJSON.version,
      name: pJSON.name,
    };
  }

  get defaults() {
    return {};
  }

  get events() {
    return [];
  }

  get actions() {
    return {};
  }

  async load(channel) {
    this._app = express();
    this._channel = channel;
```

```
    this._db = await getDBInstance();
    this._nodeInfo = await this._channel.invoke("app:getNodeInfo");


    this._app.use(cors({ origin: "*", methods: ["GET", "POST", "PUT"] }));
    this._app.use(express.json());

    this._app.get("/api/nft_tokens", async (_req, res) => {
      const nftTokens = await this._channel.invoke("nft:getAllNFTTokens");
      const data = await Promise.all(nftTokens.map(async token => {
        const dbKey = `${token.name}`;
        let tokenHistory = await getNFTHistory(this._db, dbKey);
        tokenHistory = tokenHistory.map(h => h.toString('binary'));
        return {
          ...token,
          tokenHistory,
        }
      }));

      res.json({ data });
    });

    this._app.get("/api/nft_tokens/:id", async (req, res) => {
      const nftTokens = await this._channel.invoke("nft:getAllNFTTokens");
      const token = nftTokens.find((t) => t.id === req.params.id);
      const dbKey = `${token.name}`;
      let tokenHistory = await getNFTHistory(this._db, dbKey);
      tokenHistory = tokenHistory.map(h => h.toString('binary'));

      res.json({ data: { ...token, tokenHistory } });
    });

    this._app.get("/api/transactions", async (_req, res) => {
      const transactions = await getAllTransactions(this._db, this.schemas);

      const data = transactions.map(trx => {
        const module = this._nodeInfo.registeredModules.find(m => m.id === trx.moduleID);
        const asset = module.transactionAssets.find(a => a.id === trx.assetID);
        return {
          trx
```

```
        ...trx.asset,
        moduleName: module.name,
        assetName: asset.name,
      }
    })
    res.json({ data });
  });

  this._subscribeToChannel();

  this._server = this._app.listen(8080, "0.0.0.0");
}

_subscribeToChannel() {
  // listen to application events and enrich blockchain data for UI/third party application
  this._channel.subscribe('app:block:new', async (data) => {
    const { block } = data;
    const { payload } = codec.decode(
      this.schemas.block,
      Buffer.from(block, 'hex'),
    );
    if (payload.length > 0) {
      await saveTransactions(this._db, payload);
      const decodedBlock = this.codec.decodeBlock(block);
      // save NFT transaction history
      await saveNFTHistory(this._db, decodedBlock, this._nodeInfo.registeredModules, this._channel);
    }
  });
}

async unload() {
  // close http server
  await new Promise((resolve, reject) => {
    this._server.close((err) => {
      if (err) {
        reject(err);
        return;
      }
      resolve();
    });
```

```
    });
    // close database connection
    await this._db.close();
  }
}

module.exports = { NFTAPIPlugin };
```

6. Registering module and plugin

Now that the NFT module and plugin have been implemented, it is necessary to inform the blockchain application about them.

This is done by registering them with the blockchain application as shown below.

Open the `nft/blockchain_app/index.js` file again and copy and paste the following code:

nft/blockchain_app/index.js

```
// 1.Import lisk sdk to create the blockchain application
const {
        Application,
        configDevnet,
        genesisBlockDevnet,
        HTTPAPIPlugin,
        utils,
} = require('lisk-sdk');

// 2.Import NFT module and Plugin
const { NFTModule } = require('./nft_module');
const { NFTAPIPlugin } = require('./nft_api_plugin');

// 3.Update the genesis block accounts to include NFT module attributes
genesisBlockDevnet.header.timestamp = 1605699440;
genesisBlockDevnet.header.asset.accounts = genesisBlockDevnet.header.asset.accounts.map(
        (a) =>
                utils.objects.mergeDeep({}, a, {
                        nft: {
```

```
                        ownNFTs: [],
                    },
                }),
);

// 4.Update application config to include unique label
// and communityIdentifier to mitigate transaction replay
const appConfig = utils.objects.mergeDeep({}, configDevnet, {
        label: 'nft-app',
        genesisConfig: { communityIdentifier: 'NFT' }, //In order to have a unique networkIdentifier
        logger: {
                consoleLogLevel: 'info',
        },
});

// 5.Initialize the application with genesis block and application config
const app = Application.defaultApplication(genesisBlockDevnet, appConfig);

// 6.Register custom NFT Module and Plugins
app.registerModule(NFTModule);
app.registerPlugin(HTTPAPIPlugin);
app.registerPlugin(NFTAPIPlugin);

// 7.Run the application
app
        .run()
        .then(() => console.info('NFT Blockchain running....'))
        .catch(console.error);
```

Save and close the `index.js`.

Now when the application is started again with `node index.js`, the blockchain application will load the newly created NFT module and the plugins, and the new features will become available to the blockchain application.

In the next step, we will build a simple React frontend, which allows us to interact with the blockchain application through the browser.

7. Frontend application

The final part of the NFT application is the frontend application.

> **Note**
>
> The development of the frontend application is completely flexible, and you can use any technology stack that you feel comfortable with.
>
> In this example, we use React to build the client application.

This tutorial is mainly about explaining how to build with the Lisk SDK, therefore other parts of the frontend app are not explained in much detail here. For example, you can find more information about how to build a React application in the official React documentation.

For convenience, clone the `development` branch from the `lisk-sdk-examples` GitHub repository and use the prepared NFT frontend_app from the sdk examples.

nft/

```
git clone https://github.com/LiskHQ/lisk-sdk-examples.git
mv lisk-sdk-examples/tutorials/nft/frontend_app frontend_app
rm -r ./lisk-sdk-examples
cd frontend_app
npm i
```

At this point it is now possible to already try out the frontend and verify that the NFT blockchain application works as expected:

First, open a second terminal window and navigate to the `nft/blockchain_app`. Now start the blockchain application with `node index.js`, if it is not already running.

In the first terminal window, start the frontend application with the following:

```
npm start
```

This should open the React app in the browser under http://localhost:3000/.

7.1. Frontend walkabout

Before we explore the code of the frontend app, let's first take a tour through the frontend in the browser to see how it all works together.

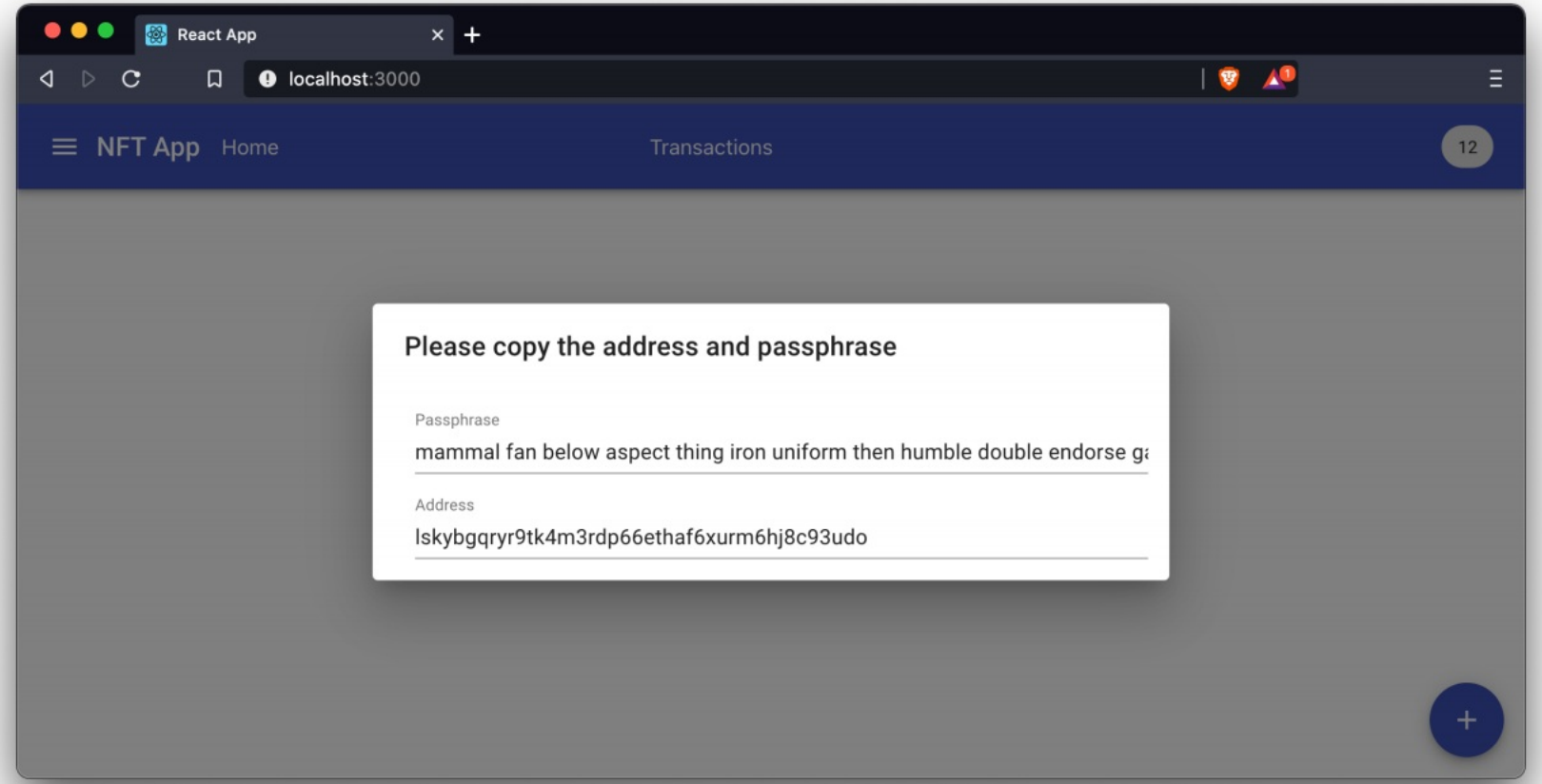
We will perform the following:

1. Create 3 different accounts.
2. Send an initial amount of tokens to each account.
3. Create a new NFT with the first account.
4. Transfer the newly created NFT to the second account.
5. Purchase the NFT with the third account.
6. Create a second NFT with the first account, which is non-purchasable.

In the example screenshots we use the following account credentials:

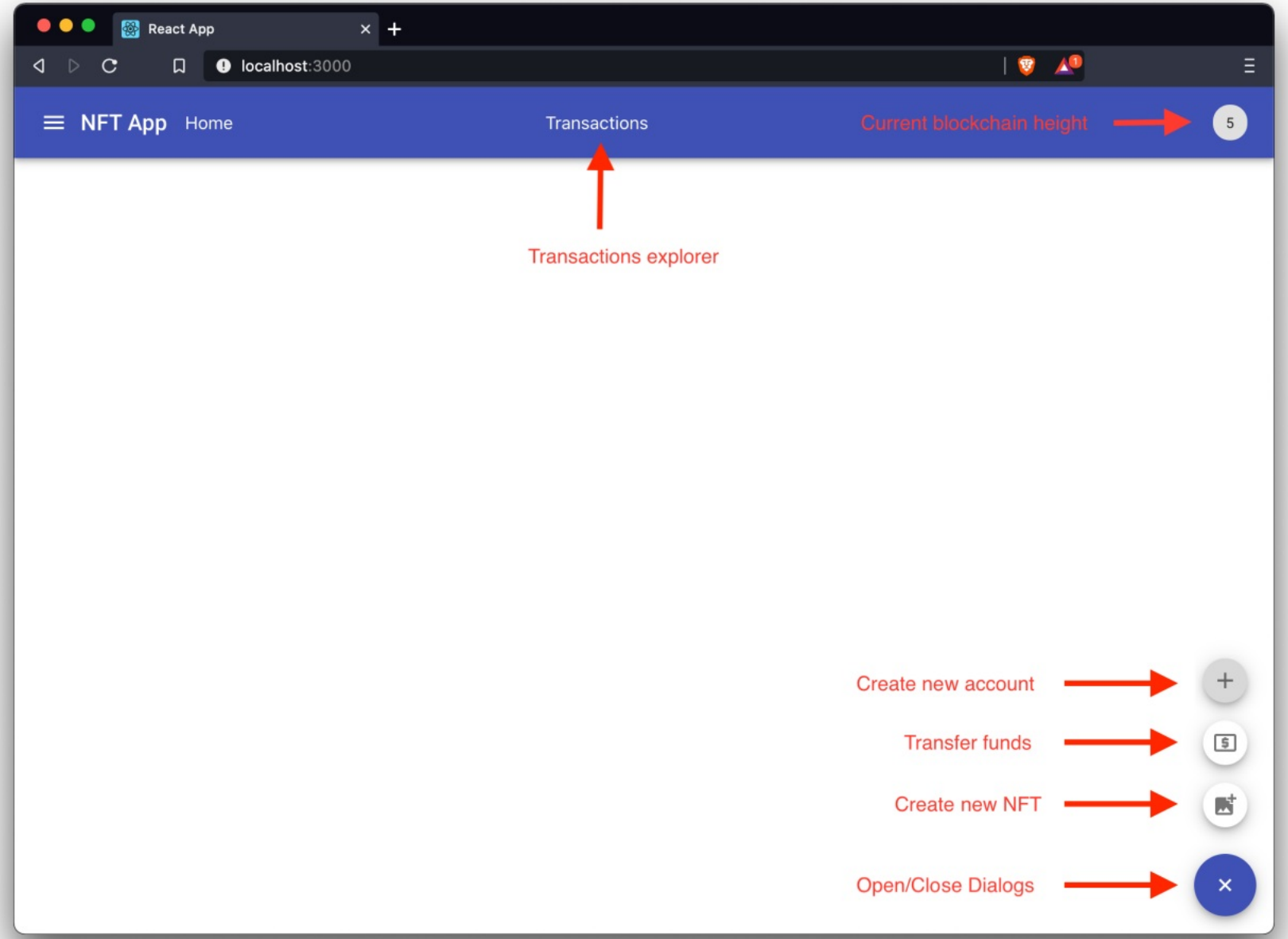
```
================================
Artist
passphrase: boss annual room suspect ride robot connect repeat relax govern dolphin depth
address: lsktxksfsbmkmoto68y7edszaecgpnaxqqg7cs43d
--------------------------------
Collector1
passphrase: mammal fan below aspect thing iron uniform then humble double endorse gauge
address: lskybgqryr9tk4m3rdp66ethaf6xurm6hj8c93udo
--------------------------------
Collector2
passphrase: emotion project prepare cream double damage gentle basket submit enhance between drill
address: lskha38ewso7do8zeuqx8qnyoqd8962mk48atknbs
================================
```

However, you can also create new credentials by using the `Create Account` dialog:



7.1.1. The home page

The home page is the landing page you see when opening the frontend app under `http://localhost:3000/` in the browser.



On the first start of the app, the page is quite empty. However, once we have created the first NFTs, the home page will display all existing NFTs and their details.

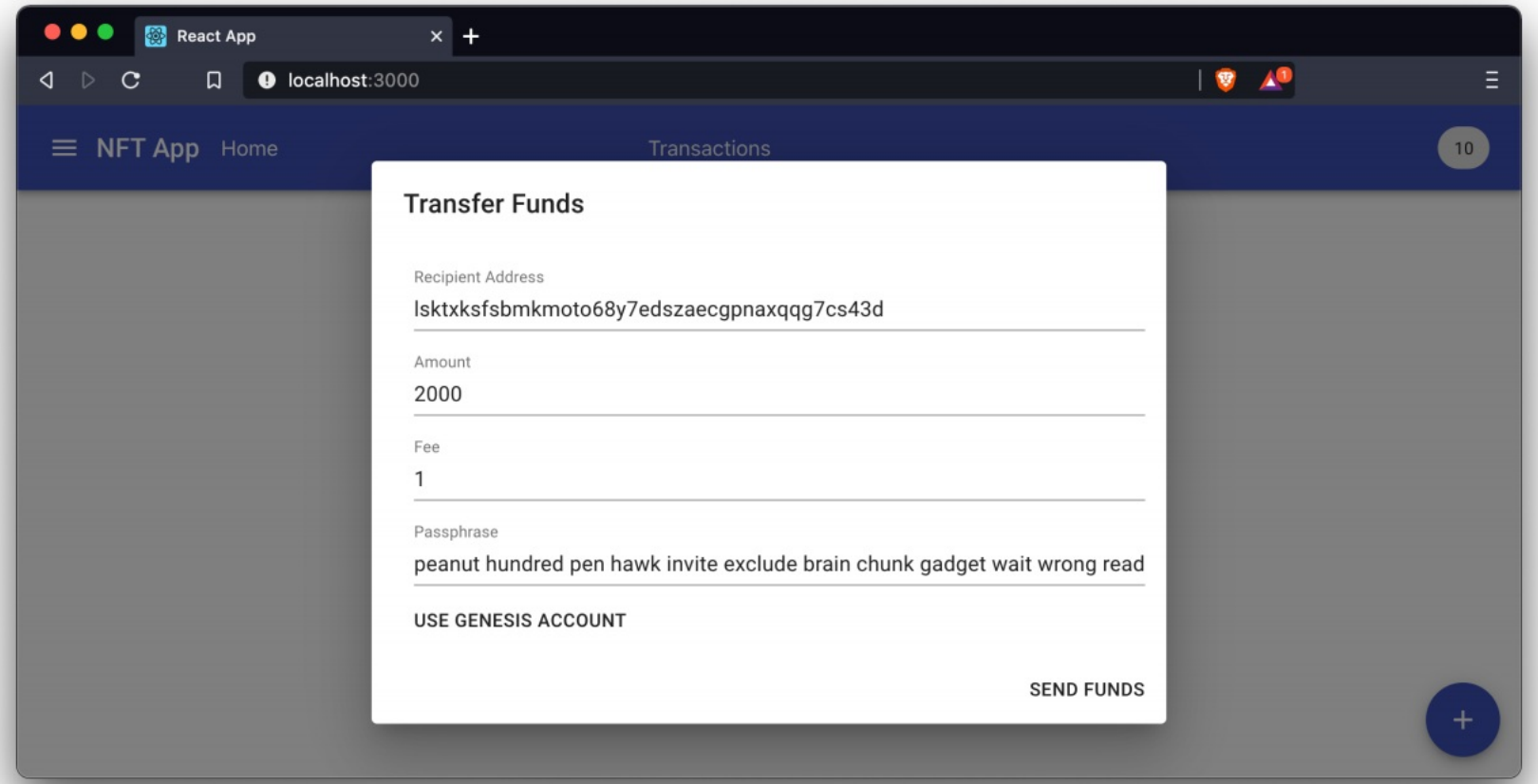
In the top bar there is a link to the transactions explorer, which displays a list of all transactions which are included in the blockchain. Currently, it should also be empty as we haven't sent any transactions yet.

On the bottom right, there is a button that can be used for opening the different dialogs for creating a new account, transferring tokens from one account to another, and most importantly, for creating new NFTs.

To be able to create and purchase a new NFT, an account needs to have some tokens in the account balance. Therefore, we first need to transfer some tokens to the created accounts.

7.1.2. Transferring funds

Click on the `Transfer tokens` dialog and transfer an adequate amount of tokens to the above described demo accounts. In our example, we transferred 2000 tokens to the artist account and 1000 tokens each to the collector accounts.



For the passphrase, use the passphrase of the genesis account, by clicking on the button `Use genesis account`. Now click on `Send funds`. This will send the specified tokens from the genesis account to the specified account in the `Recipient Address` field.

This will post a corresponding transfer transaction to the blockchain application.

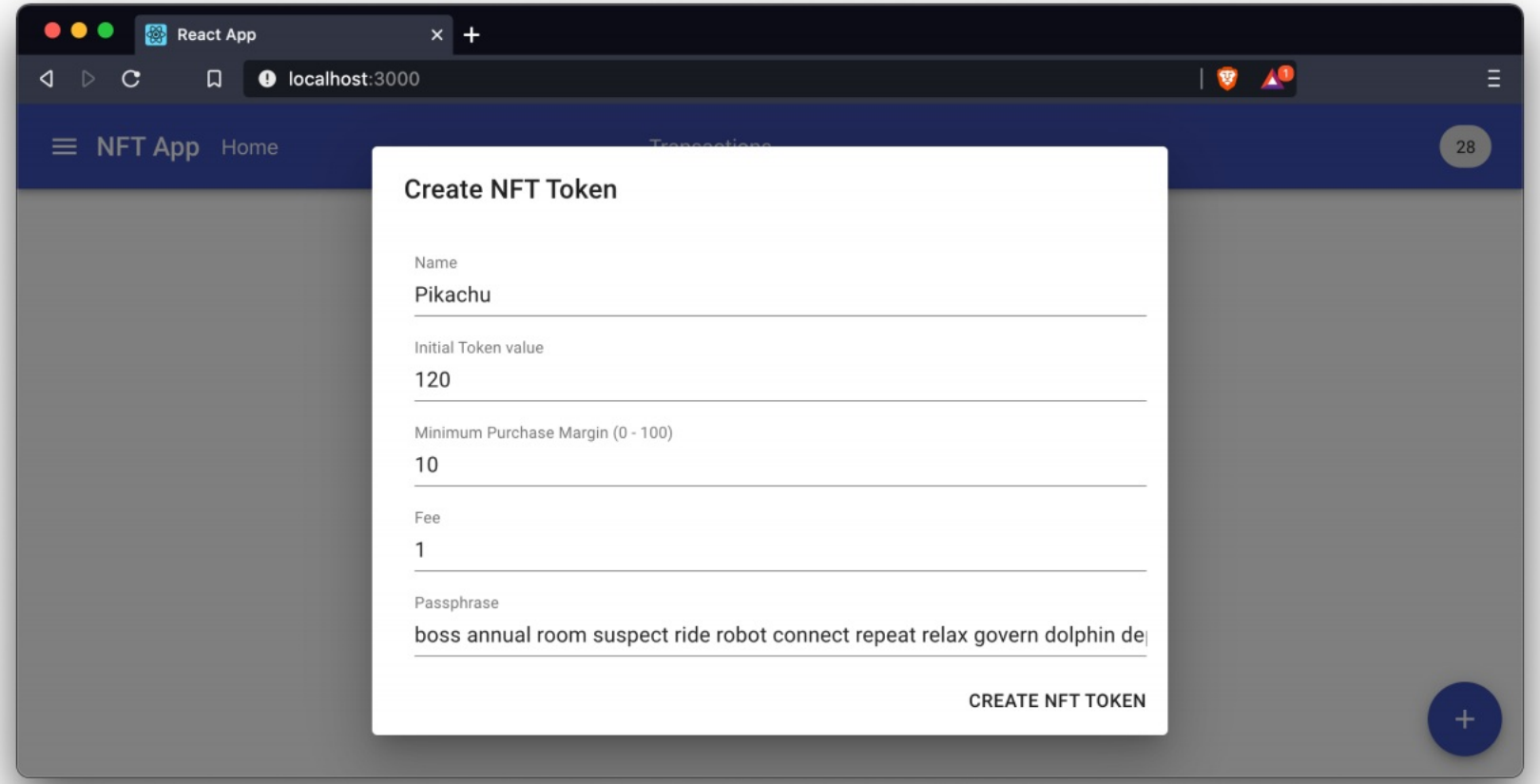
It is possible to verify the transaction was included in a block by observing the logs of the blockchain application:

Logs of the blockchain app when a valid transaction is posted to the node

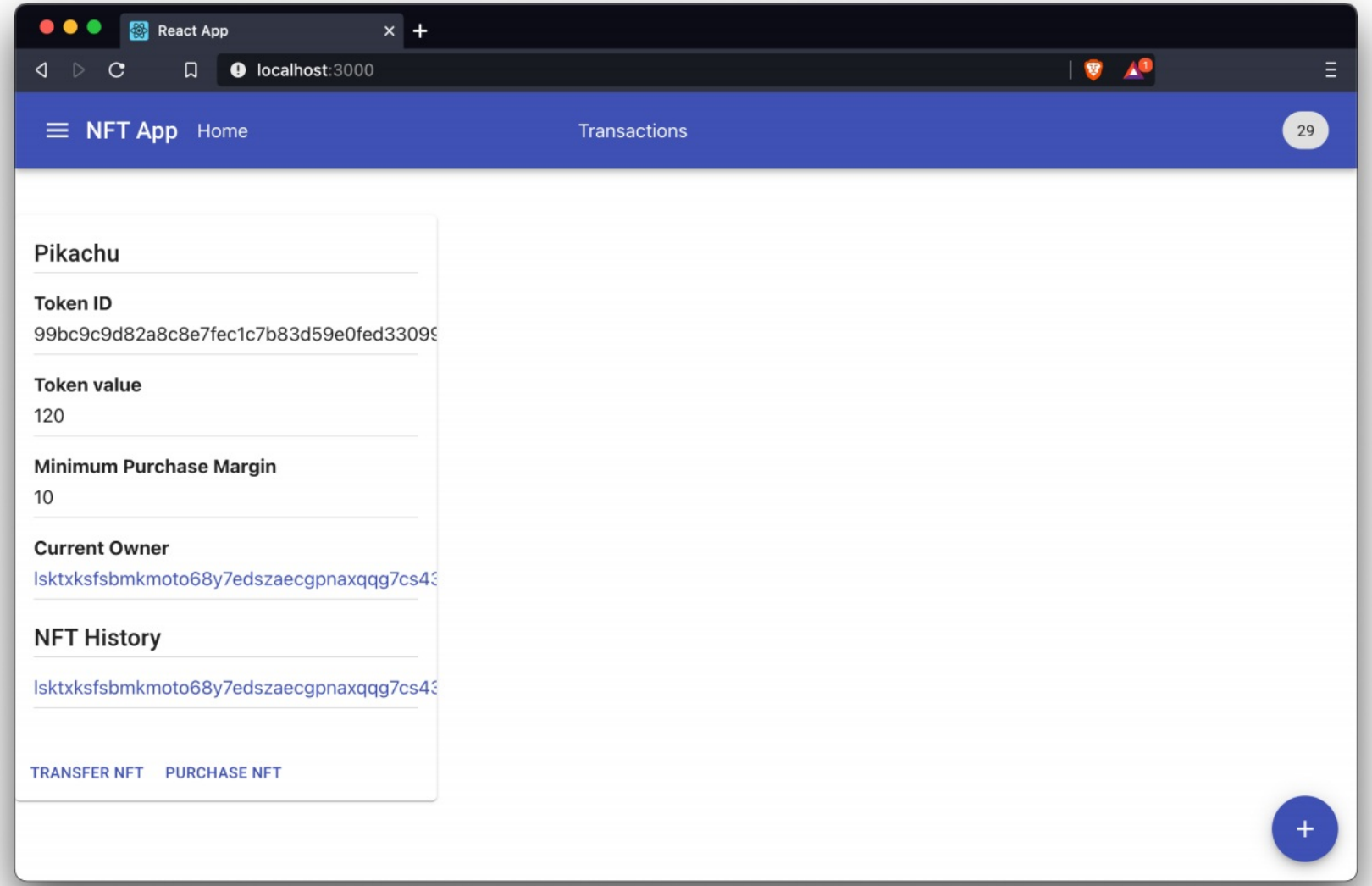
```
15:20:56 INFO lisk-framework: Added transaction to pool (module=lisk:app)
{
 "id": "b9d81d996886f28d2f9fd102c2d8407dc86df941eeea0b03c004080e0f100f27",
 "nonce": "0",
 "senderPublicKey": "836d4f07c7db6d10c84394c60549d3f95cf61354e2ab5b0965a3fe7120e2f70d"
}
15:21:00 INFO lisk-framework: New block added to the chain (module=lisk:app)
{
 "id": "f6d2ee7cb0e76938340f0b8a946389d518e7f27c062759f2c78f47d2841a7010",
 "height": 787,
 "numberOfTransactions": 1
}
```

7.1.3. Creating a new NFT

Now that all accounts have some tokens in their account balance, use the artist account passphrase to create a new NFT.



After approximately 10 seconds, the NFT should appear on the home page, after refreshing the page.



At the bottom of the NFT card, the user now has the option to `transfer` or `purchase` the NFT.

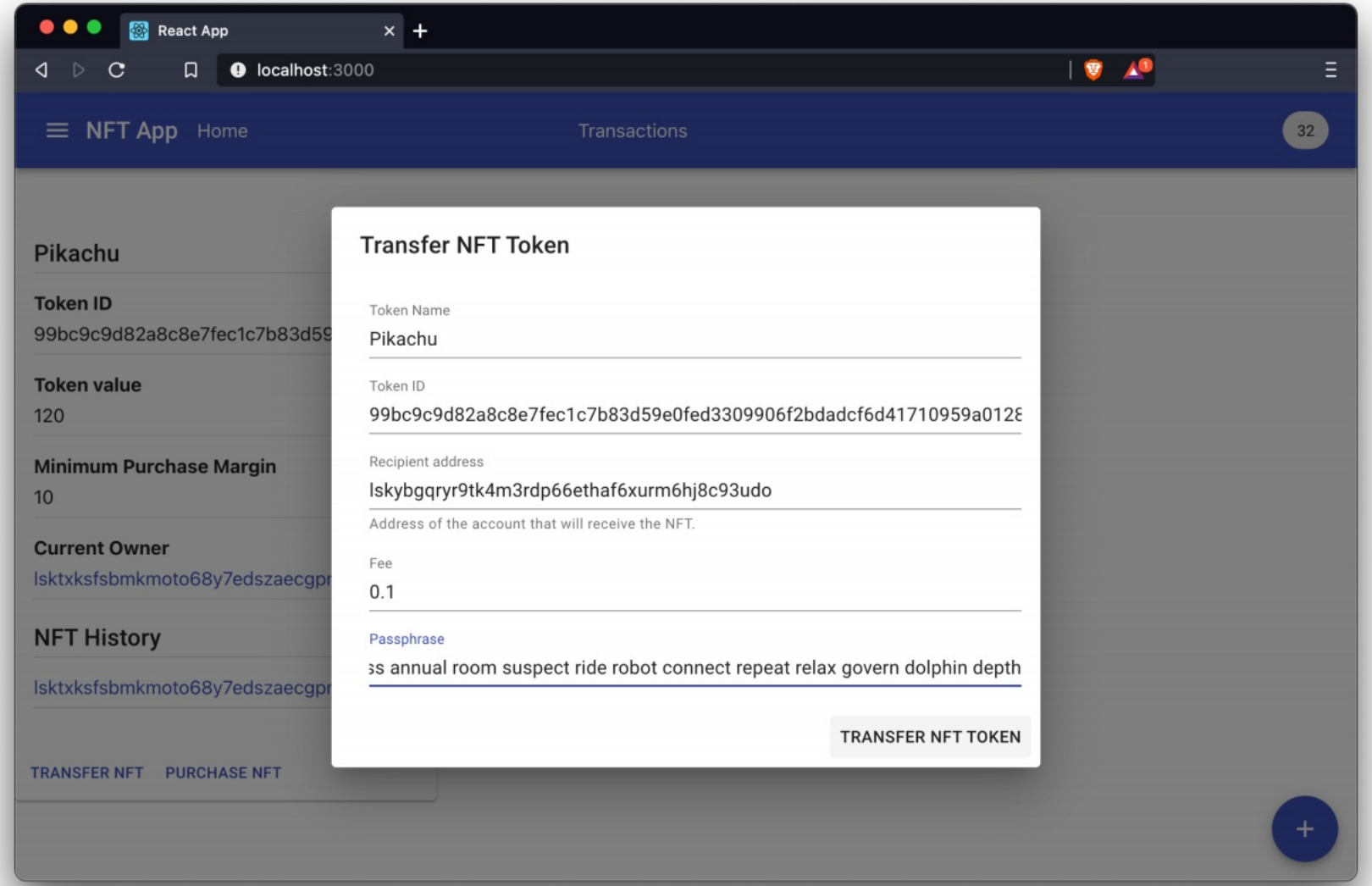
7.1.4. Transferring an NFT

To test the `Transfer NFT` option, transfer the NFT now to the Collector1 account:

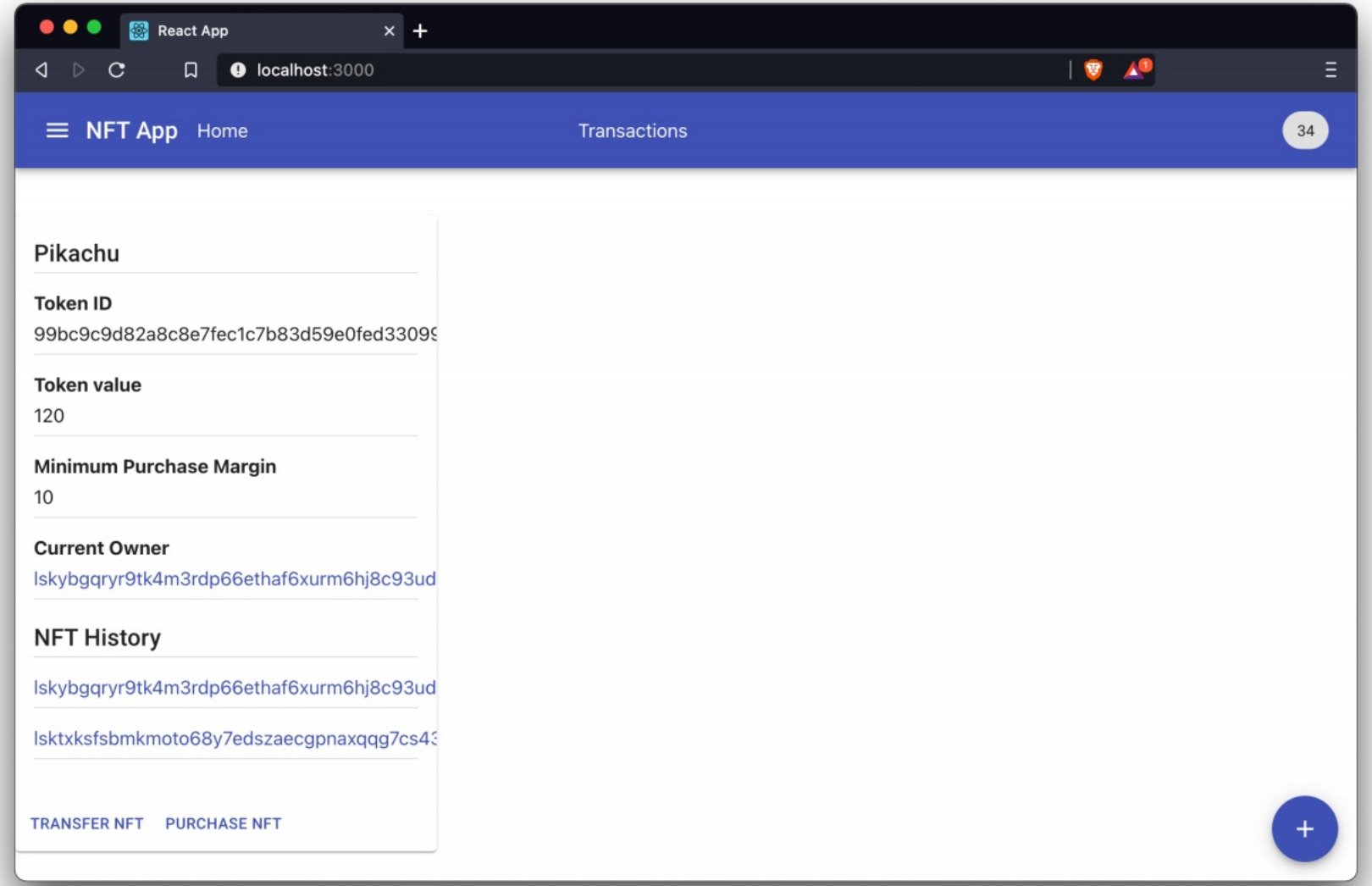
Important

You must use the passphrase of the current owner of the NFT to sign this transaction.

If a different account than the owner tries to transfer the NFT, the application will throw an error `An NFT can only be transferred by the owner of the NFT.`, which we defined previously in the TransferNFT asset section.

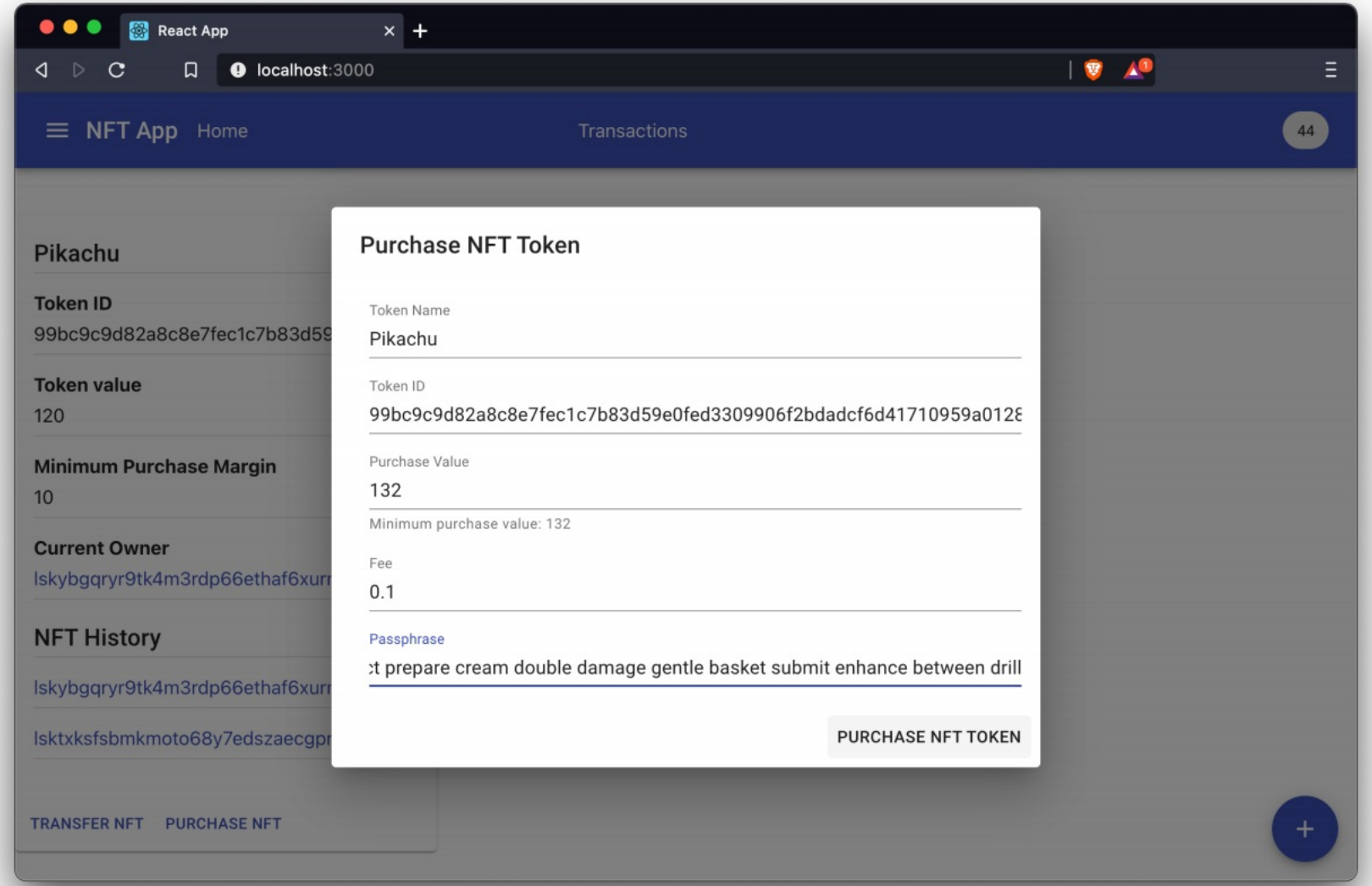


Now wait 10 seconds and refresh the home page again, to see the updated owner and history of the NFT.



7.1.5. Purchasing an NFT

To test the `Purchase NFT` option, purchase the NFT with the Collector2 account:



Now wait again 10 seconds, to see the updated owner and history of the NFT.

Click on the account addresses in the NFT history, to view the account details of the corresponding account on a new page:

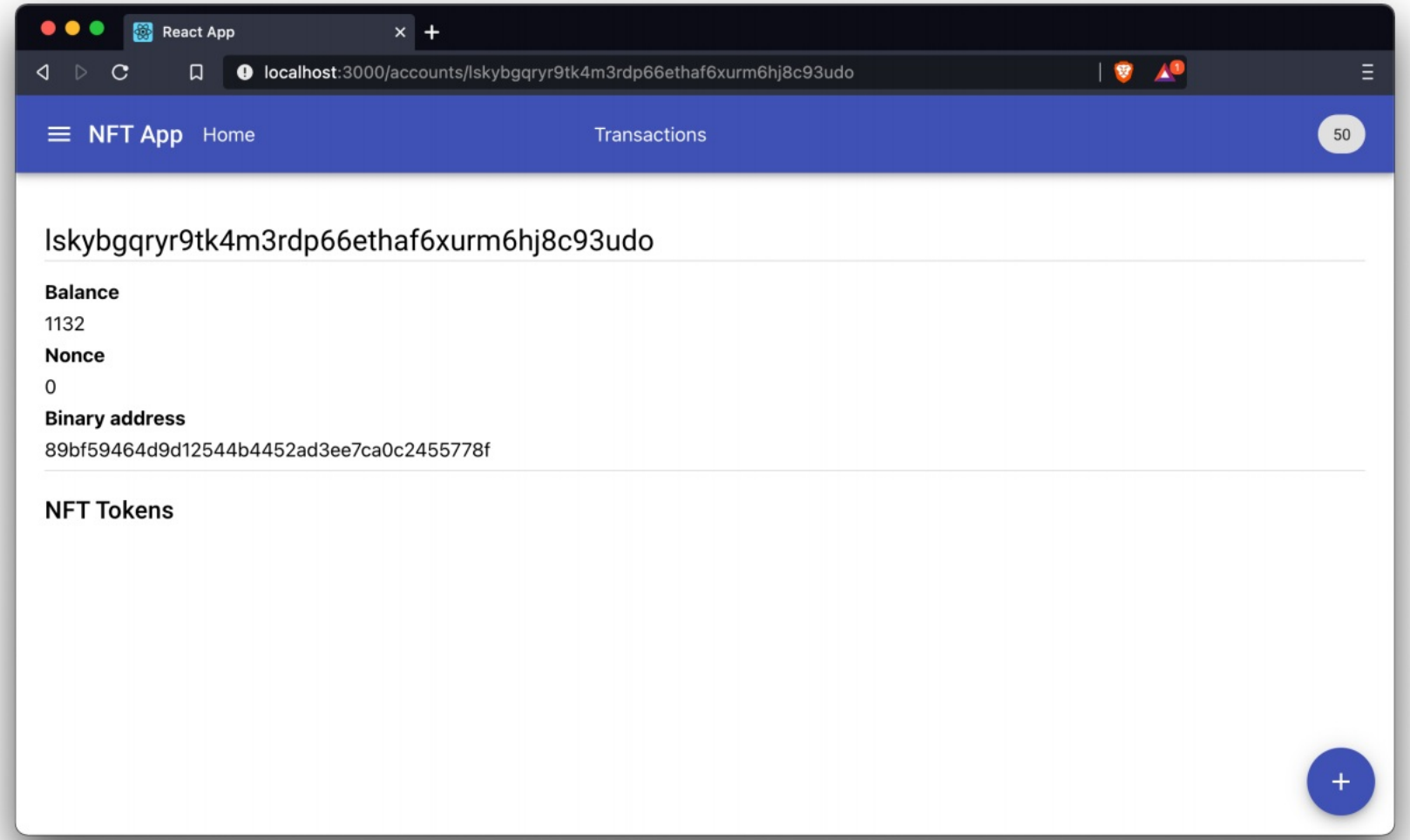


Figure 1. Collector1 account details

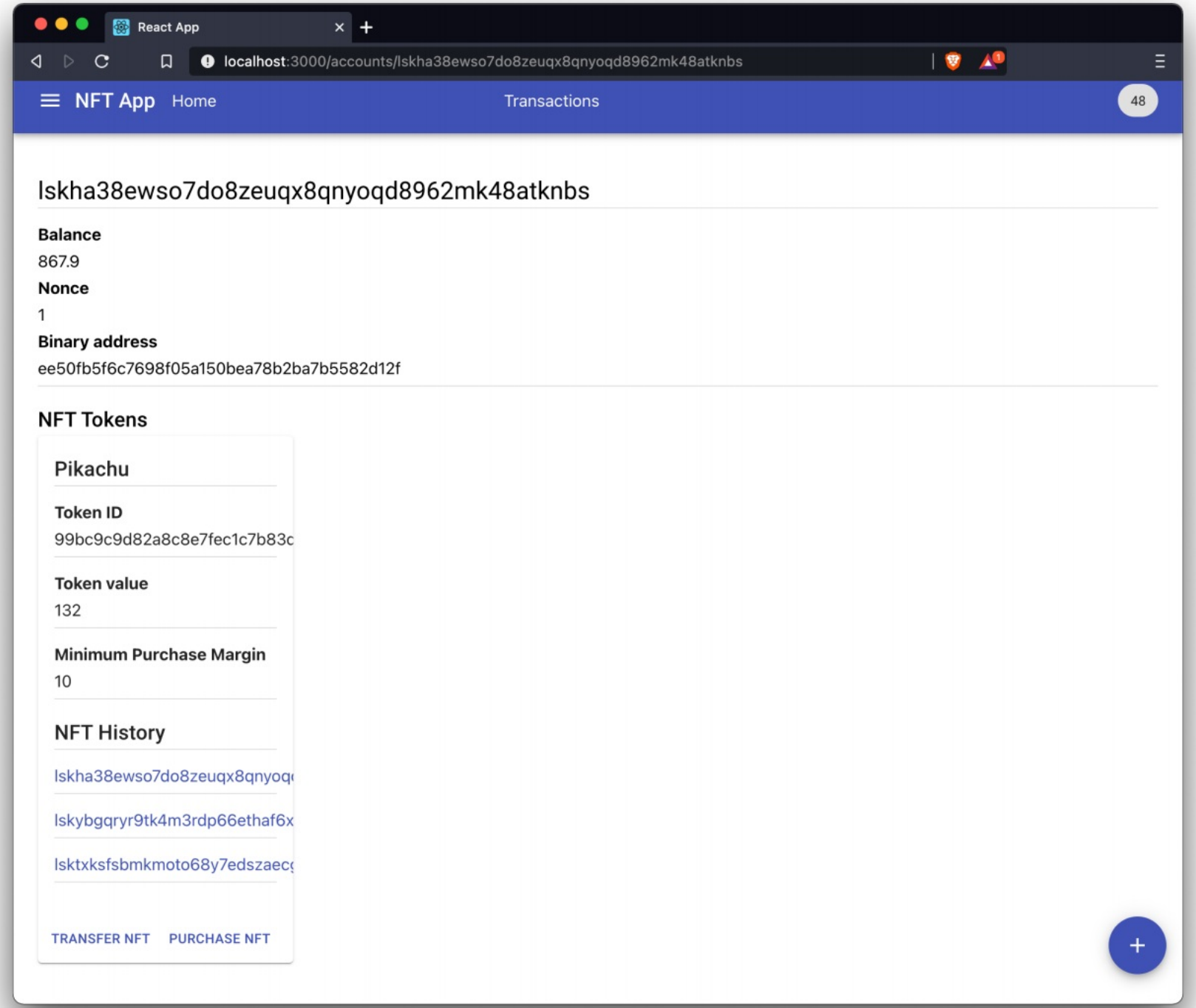

React App
localhost:3000/accounts/lskha38ewso7do8zeuqx8qnyoqd8962mk48atknbs
NFT App
Home
Transactions
48
lskha38ewso7do8zeuqx8qnyoqd8962mk48atknbs
Balance
867.9
Nonce
1
Binary address
ee50fb5f6c7698f05a150bea78b2ba7b5582d12f
NFT Tokens
Pikachu
Token ID
99bc9c9d82a8c8e7fec1c7b83c
Token value
132
Minimum Purchase Margin
10
NFT History
lskha38ewso7do8zeuqx8qnyoq
lskybgqryr9tk4m3rdp66ethaf6x
lsktxksfsbmkmoto68y7edszaec
TRANSFER NFT
PURCHASE NFT

Figure 2. Collector2 account details

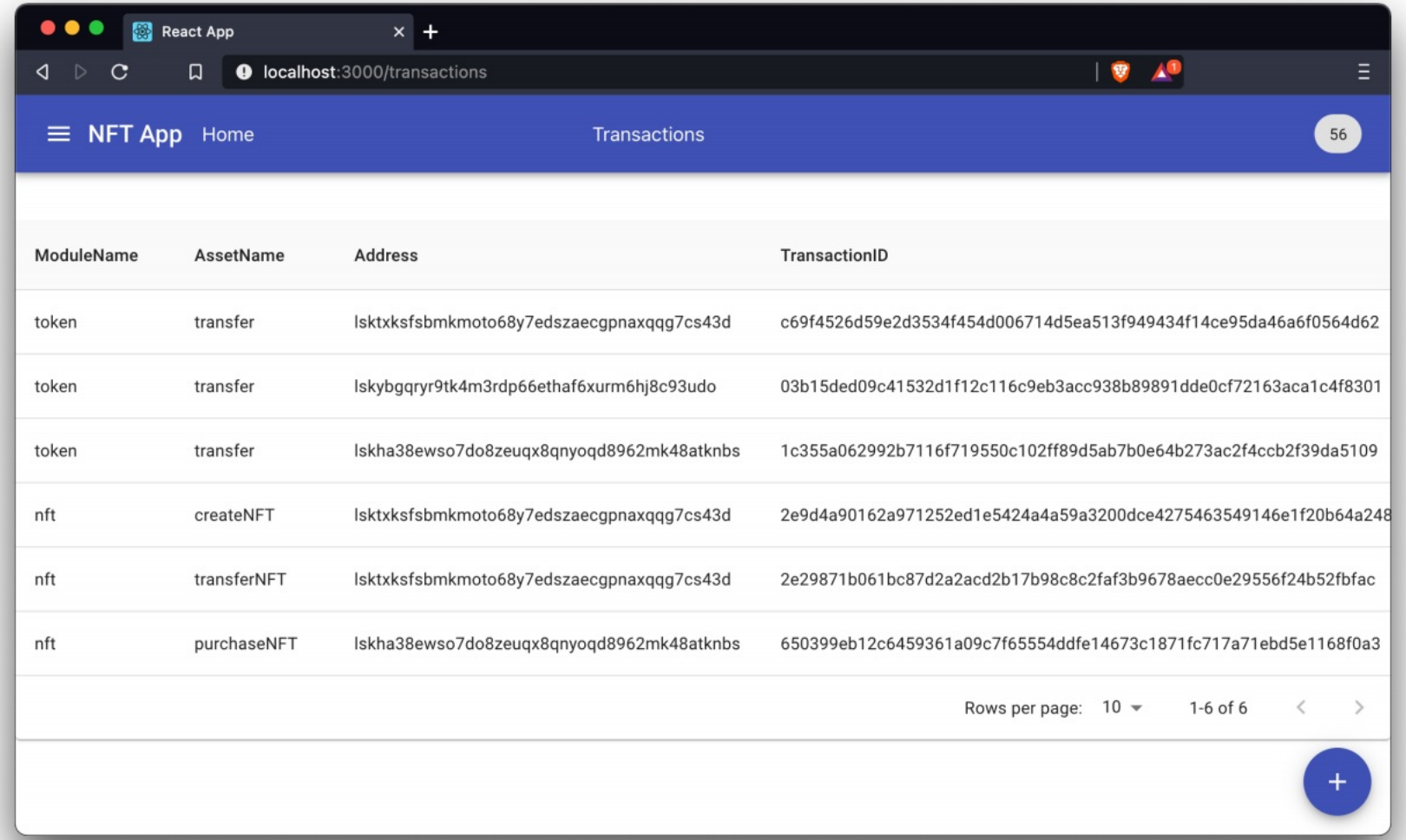
On the account page of Collector2, the NFT is now included.

Have a close look at the `Token value` property, which has increased from 120 to 132 due to the purchase.

7.1.6. The transactions explorer

Click on the `Transactions` link in the top bar, to go to the transaction explorer.

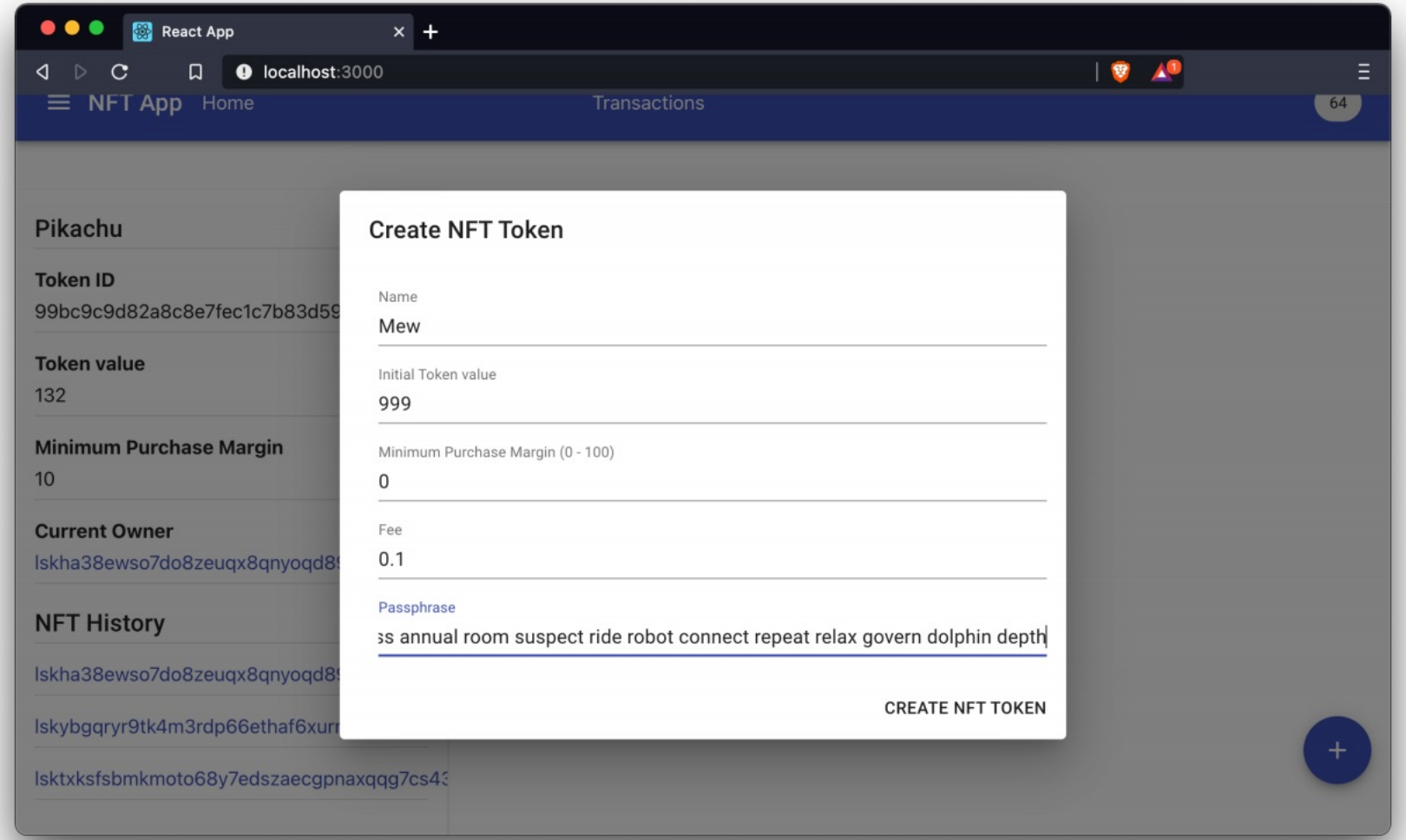
Here you can see a list of all transactions, which have been posted in the network so far, including a few details such as their respective module and asset name, the sender address, and the transaction ID.



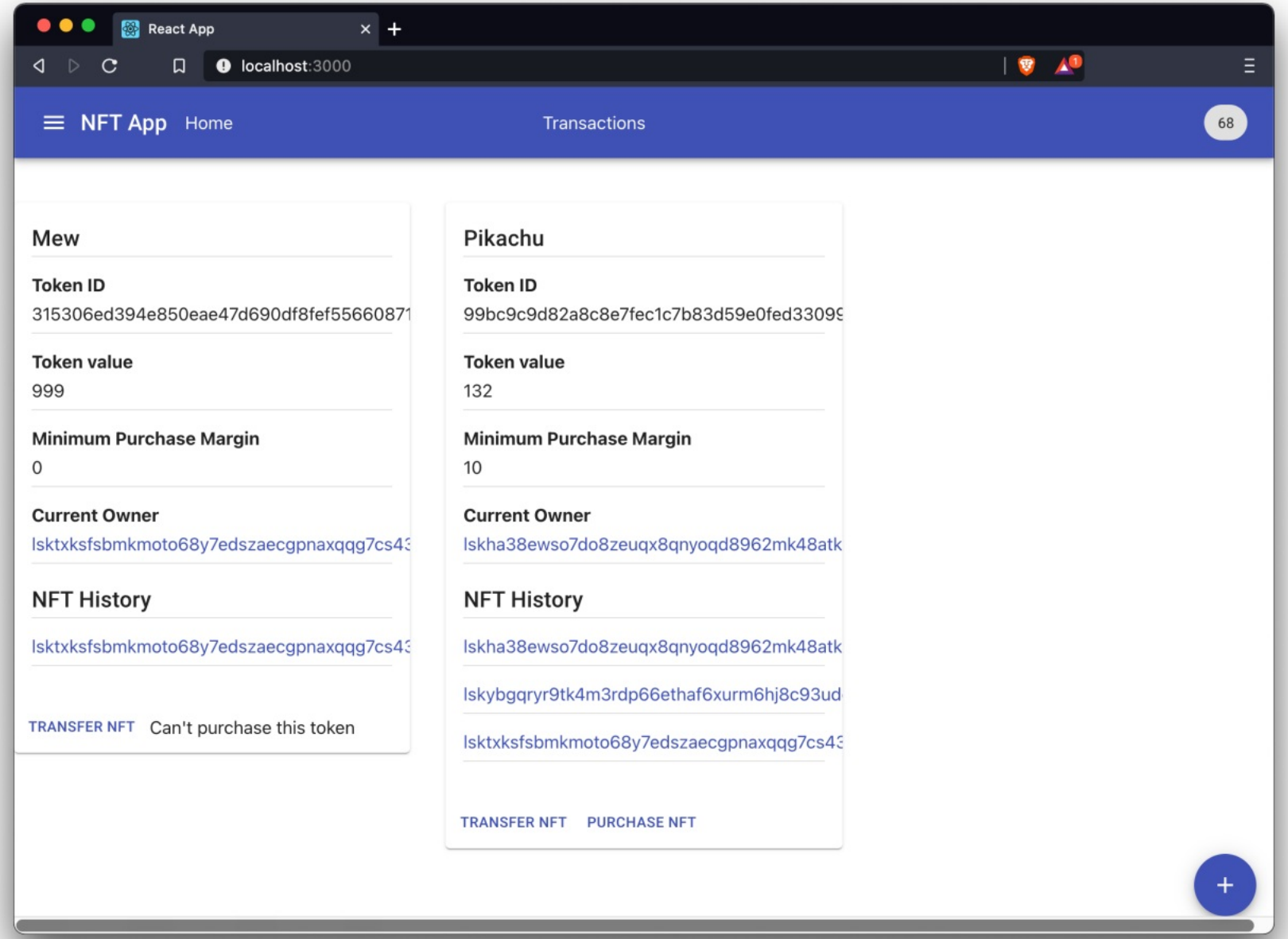
7.1.7. Creating non-purchasable NFTs

To become more familiar with the process, play around a little more with the NFT app by creating more NFTs.

For example, it is also possible to create non-purchasable NFTs by setting the minimum purchase margin to zero.



The refreshed home page will then look like this:



That's it, the frontend walkabout in the browser is now complete.

Next let's take a dive into the most important parts of the frontend app, regarding the blockchain related logic.

7.2. API related functions

At first define multiple functions that fetch data from the HTTP API of the blockchain information.

The NFT blockchain app offers two different HTTP APIs:

- http://localhost:4000/api/ : The API of the `HTTPAPIPlugin`. Used to retrieve general blockchain information from the database.
- http://localhost:8080/api/ : The API of the `NFTAPIPlugin`. Used to retrieve NFT related information from the database.

We will use both APIs and their provided endpoints to retrieve or post the following data:

- General blockchain information
  - `fetchNodeInfo()`: Returns information about the connected node.
  - `fetchAccountInfo(address)`: Returns details of a specific account, based on its address.
  - `sendTransactions(tx)`: Sends a specified transaction object `tx` to the node.
- NFT related information
  - `fetchAllNFTTokens()`: Fetches a list of all registered NFTs in the network.
  - `fetchNFTToken()`: Returns details of a specific NFT, based on its ID.
  - `getAllTransactions()`: Returns a list of all posted transactions in the network.

frontend_app/src/api/index.js

```
export const fetchNodeInfo = async () => {
  return fetch("http://localhost:4000/api/node/info")
    .then((res) => res.json())
    .then((res) => res.data);
};

export const fetchAccountInfo = async (address) => {
  return fetch(`http://localhost:4000/api/accounts/${address}`)
    .then((res) => res.json())
    .then((res) => res.data);
};
```

```
export const sendTransactions = async (tx) => {
  return fetch("http://localhost:4000/api/transactions", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(tx),
  })
    .then((res) => res.json())
    .then((res) => res.data);
};

export const fetchAllNFTTokens = async () => {
  return fetch("http://localhost:8080/api/nft_tokens")
    .then((res) => res.json())
    .then((res) => res.data);
};

export const fetchNFTToken = async (id) => {
  return fetch(`http://localhost:8080/api/nft_tokens/${id}`)
    .then((res) => res.json())
    .then((res) => res.data);
};

export const getAllTransactions = async () => {
  return fetch(`http://localhost:8080/api/transactions`)
    .then((res) => res.json())
    .then((res) => {
      return res.data;
    });
};
```

7.3. Functions for creating transactions

The `utils/` folder of the frontend app stores certain utility functions for creating the different transaction types.

The functions will be used in the Dialogs below to create and send the transaction objects based on the form data in the dialog.

The transactions are created and signed by utilizing the `signTransaction()` from the `@liskhq/lisk-client` package.

The nonce for each transaction is retrieved from the sender account by running `fetchAccountInfo()` from the API related functions section.

7.3.1. Create NFT

frontend_app/src/utils/transactions/create_nft_token.js

```
/* global BigInt */

import { transactions, codec, cryptography } from "@liskhq/lisk-client";
import { getFullAssetSchema, calcMinTxFee } from "../common";
import { fetchAccountInfo } from "../../api";

export const createNFTTokenSchema = {
  $id: "lisk/create-nft-asset",
  type: "object",
  required: ["minPurchaseMargin", "initValue", "name"],
  properties: {
    minPurchaseMargin: {
      dataType: "uint32",
      fieldNumber: 1,
    },
    initValue: {
      dataType: "uint64",
      fieldNumber: 2,
    },
    name: {
      dataType: "string",
      fieldNumber: 3,
    },
  },
};

export const createNFTToken = async ({
  name,
  initValue,
  minPurchaseMargin,
```

```
  passphrase,
  fee,
  networkIdentifier,
  minFeePerByte,
}) => {
  const { publicKey } = cryptography.getPrivateAndPublicKeyFromPassphrase(
    passphrase
  );
  const address = cryptography.getAddressFromPassphrase(passphrase).toString("hex");

  const {
    sequence: { nonce },
  } = await fetchAccountInfo(address);

  const { id, ...rest } = transactions.signTransaction(
    createNFTTokenSchema,
    {
      moduleID: 1024,
      assetID: 0,
      nonce: BigInt(nonce),
      fee: BigInt(transactions.convertLSKToBeddows(fee)),
      senderPublicKey: publicKey,
      asset: {
        name,
        initValue: BigInt(transactions.convertLSKToBeddows(initValue)),
        minPurchaseMargin: parseInt(minPurchaseMargin),
      },
    },
    Buffer.from(networkIdentifier, "hex"),
    passphrase
  );

  return {
    id: id.toString("hex"),
    tx: codec.codec.toJSON(getFullAssetSchema(createNFTTokenSchema), rest),
    minFee: calcMinTxFee(createNFTTokenSchema, minFeePerByte, rest),
  };
};
```

7.3.2. Purchase NFT

frontend_app/src/utils/transactions/purchase_nft_token.js

```
/* global BigInt */

import { transactions, codec, cryptography } from "@liskhq/lisk-client";
import { getFullAssetSchema, calcMinTxFee } from "../common";
import { fetchAccountInfo } from "../../api";

export const purchaseNFTTokenSchema = {
  $id: "lisk/nft/purchase",
  type: "object",
  required: ["nftId", "purchaseValue"],
  properties: {
    nftId: {
      dataType: "bytes",
      fieldNumber: 1,
    },
    purchaseValue: {
      dataType: "uint64",
      fieldNumber: 2,
    },
    name: {
      dataType: "string",
      fieldNumber: 3,
    },
  },
};

export const purchaseNFTToken = async ({
  name,
  nftId,
  purchaseValue,
  passphrase,
  fee,
  networkIdentifier,
  minFeePerByte,
}) => {
  const { publicKey } = cryptography.getPrivateAndPublicKeyFromPassphrase(
    passphrase
  );
```

```
  const address = cryptography.getAddressFromPassphrase(passphrase);
  const {
    sequence: { nonce },
  } = await fetchAccountInfo(address.toString("hex"));

  const { id, ...rest } = transactions.signTransaction(
    purchaseNFTTokenSchema,
    {
      moduleID: 1024,
      assetID: 1,
      nonce: BigInt(nonce),
      fee: BigInt(transactions.convertLSKToBeddows(fee)),
      senderPublicKey: publicKey,
      asset: {
        name,
        nftId: Buffer.from(nftId, "hex"),
        purchaseValue: BigInt(transactions.convertLSKToBeddows(purchaseValue)),
      },
    },
    Buffer.from(networkIdentifier, "hex"),
    passphrase
  );

  return {
    id: id.toString("hex"),
    tx: codec.codec.toJSON(getFullAssetSchema(purchaseNFTTokenSchema), rest),
    minFee: calcMinTxFee(purchaseNFTTokenSchema, minFeePerByte, rest),
  };
};
```

7.3.3. Transfer funds

frontend_app/src/utils/transactions/transfer.js

```
/* global BigInt */

import { transactions, codec, cryptography } from "@liskhq/lisk-client";
import { getFullAssetSchema, calcMinTxFee } from "../common";
```

```
import { fetchAccountInfo } from "../../api";

export const transferAssetSchema = {
  $id: "lisk/transfer-asset",
  title: "Transfer transaction asset",
  type: "object",
  required: ["amount", "recipientAddress", "data"],
  properties: {
    amount: {
      dataType: "uint64",
      fieldNumber: 1,
    },
    recipientAddress: {
      dataType: "bytes",
      fieldNumber: 2,
      minLength: 20,
      maxLength: 20,
    },
    data: {
      dataType: "string",
      fieldNumber: 3,
      minLength: 0,
      maxLength: 64,
    },
  },
};

export const transfer = async ({
  recipientAddress,
  amount,
  passphrase,
  fee,
  networkIdentifier,
  minFeePerByte,
}) => {
  const { publicKey } = cryptography.getPrivateAndPublicKeyFromPassphrase(
    passphrase
  );
  const address = cryptography.getAddressFromPassphrase(passphrase);
  const {
```

```
    sequence: { nonce },
  } = await fetchAccountInfo(address.toString("hex"));
  const recipient = cryptography.getAddressFromBase32Address(recipientAddress);
  const { id, ...rest } = transactions.signTransaction(
    transferAssetSchema,
    {
      moduleID: 2,
      assetID: 0,
      nonce: BigInt(nonce),
      fee: BigInt(transactions.convertLSKToBeddows(fee)),
      senderPublicKey: publicKey,
      asset: {
        amount: BigInt(transactions.convertLSKToBeddows(amount)),
        recipientAddress: recipient,
        data: "",
      },
    },
    Buffer.from(networkIdentifier, "hex"),
    passphrase
  );

  return {
    id: id.toString("hex"),
    tx: codec.codec.toJSON(getFullAssetSchema(transferAssetSchema), rest),
    minFee: calcMinTxFee(transferAssetSchema, minFeePerByte, rest),
  };
};
```

7.3.4. Transfer NFT

frontend_app/src/utils/transactions/transfer_nft.js

```
/* global BigInt */

import { transactions, codec, cryptography } from "@liskhq/lisk-client";
import { getFullAssetSchema, calcMinTxFee } from "../common";
import { fetchAccountInfo } from "../../api";

export const transferNFTSchema = {
```

```
  $id: "lisk/nft/transfer",
  type: "object",
  required: ["nftId", "recipient"],
  properties: {
    nftId: {
      dataType: "bytes",
      fieldNumber: 1,
    },
    recipient: {
      dataType: "bytes",
      fieldNumber: 2,
    },
    name: {
      dataType: "string",
      fieldNumber: 3,
    },
  },
};

export const transferNFT = async ({
                                    name,
                                    nftId,
                                    recipientAddress,
                                    passphrase,
                                    fee,
                                    networkIdentifier,
                                    minFeePerByte,
                                  }) => {
  const { publicKey } = cryptography.getPrivateAndPublicKeyFromPassphrase(
    passphrase
  );
  const address = cryptography.getAddressFromPassphrase(passphrase);
  const recipient = cryptography.getAddressFromBase32Address(recipientAddress);
  const {
    sequence: { nonce },
  } = await fetchAccountInfo(address.toString("hex"));

  const { id, ...rest } = transactions.signTransaction(
    transferNFTSchema,
    {
```

```
      moduleID: 1024,
      assetID: 2,
      nonce: BigInt(nonce),
      fee: BigInt(transactions.convertLSKToBeddows(fee)),
      senderPublicKey: publicKey,
      asset: {
        name,
        nftId: Buffer.from(nftId, "hex"),
        recipient: recipient,
      },
    },
    Buffer.from(networkIdentifier, "hex"),
    passphrase
  );

  return {
    id: id.toString("hex"),
    tx: codec.codec.toJSON(getFullAssetSchema(transferNFTSchema), rest),
    minFee: calcMinTxFee(transferNFTSchema, minFeePerByte, rest),
  };
};
```

7.4. Dialogs

7.4.1. Create account dialog

The create account dialog creates new account details each time it is opened.

Note, that these account details are only created locally, and are not included in the blockchain yet.

To include an account in the blockchain, simply send some funds to the account with the Transfer funds dialog.

To create the account details, the `passphrase` and `cryptography` library of the `@liskhq/lisk-client` package are used.

frontend_app/src/components/dialogs/CreateAccountDialog.js

```
import React, { Fragment, useState, useEffect } from "react";
import {
```

```
  Dialog,
  DialogTitle,
  DialogContent,
  TextField,
} from "@material-ui/core";
import { makeStyles } from "@material-ui/core/styles";
import { passphrase, cryptography } from "@liskhq/lisk-client";

const useStyles = makeStyles((theme) => ({
  root: {
    "& .MuiTextField-root": {
      margin: theme.spacing(1),
    },
  },
}));

export default function CreateAccountDialog(props) {
  const [data, setData] = useState({ passphrase: "", address: "" });
  const classes = useStyles();

  useEffect(() => {
    const pw = passphrase.Mnemonic.generateMnemonic();
    const address = cryptography.getBase32AddressFromPassphrase(pw).toString("hex");
    setData({ passphrase: pw, address });
  }, [props.open]);

  return (
    <Fragment>
      <Dialog open={props.open} onBackdropClick={props.handleClose} fullWidth>
        <DialogTitle id="alert-dialog-title">
          {"Please copy the address and passphrase"}
        </DialogTitle>
        <DialogContent>
          <form noValidate autoComplete="off" className={classes.root}>
            <TextField
              label="Passphrase"
              value={data.passphrase}
              fullWidth
              InputProps={{
                readOnly: true,
```

```
            }}
          />
          <TextField
            label="Address"
            value={data.address}
            fullWidth
            InputProps={{
              readOnly: true,
            }}
          />
        </form>
      </DialogContent>
    </Dialog>
  </Fragment>
 );
}
```

7.4.2. Create NFT dialog

The create NFT dialog allows a user to create a new NFT.

It renders a form where a user can enter all important information to create the NFT:

- **Name**: The name of the NFT.
- **Initial Token value**: The inital value of the token. The amount will be debited from the balance of the account which creates the NFT.
- **Minimum Purchase Margin**: The minimum margin in %, which is added to the token value on purchase.
- **Fee**: The transaction fee for the `createNFT` transaction.
- **Passphrase**: The passphrase of the account which creates the NFT.

It then uses the `createNFTToken()` function we defined in the Create NFT section to create the `createNFT` transaction and the `sendTransactions()` function from the API related functions section to post the transaction to the blockchain application.

frontend_app/src/components/dialogs/CreateNFTTokenDialog.js

```
import React, { Fragment, useContext, useState } from "react";
```

```
import {
  Dialog,
  DialogTitle,
  DialogContent,
  TextField,
  Button,
  DialogActions,
} from "@material-ui/core";
import { makeStyles } from "@material-ui/core/styles";
import { NodeInfoContext } from "../../context";
import { createNFTToken } from "../../utils/transactions/create_nft_token";
import * as api from "../../api";

const useStyles = makeStyles((theme) => ({
  root: {
    "& .MuiTextField-root": {
      margin: theme.spacing(1),
    },
  },
}));

export default function CreateNFTTokenDialog(props) {
  const nodeInfo = useContext(NodeInfoContext);
  const classes = useStyles();
  const [data, setData] = useState({
    name: "",
    initValue: "",
    minPurchaseMargin: "",
    fee: "",
    passphrase: "",
  });

  const handleChange = (event) => {
    event.persist();
    setData({ ...data, [event.target.name]: event.target.value });
  };

  const handleSend = async (event) => {
    event.preventDefault();
```

```
    const res = await createNFTToken({
      ...data,
      networkIdentifier: nodeInfo.networkIdentifier,
      minFeePerByte: nodeInfo.minFeePerByte,
    });
    await api.sendTransactions(res.tx);
    props.handleClose();
  };

  return (
    <Fragment>
      <Dialog open={props.open} onBackdropClick={props.handleClose}>
        <DialogTitle id="alert-dialog-title">{"Create NFT"}</DialogTitle>
        <DialogContent>
          <form className={classes.root} noValidate autoComplete="off">
            <TextField
              label="Name"
              value={data.name}
              name="name"
              onChange={handleChange}
              fullWidth
            />
            <TextField
              label="Initial Token value"
              value={data.initValue}
              name="initValue"
              onChange={handleChange}
              fullWidth
            />
            <TextField
              label="Minimum Purchase Margin (0 - 100)"
              value={data.minPurchaseMargin}
              name="minPurchaseMargin"
              onChange={handleChange}
              fullWidth
            />
            <TextField
              label="Fee"
              value={data.fee}
              name="fee"
```

```
              onChange={handleChange}
              fullWidth
            />
            <TextField
              label="Passphrase"
              value={data.passphrase}
              name="passphrase"
              onChange={handleChange}
              fullWidth
            />
          </form>
        </DialogContent>
        <DialogActions>
          <Button onClick={handleSend}>Create NFT</Button>
        </DialogActions>
      </Dialog>
    </Fragment>
  );
}
```

7.4.3. Purchase NFT dialog

The purchase NFT dialog allows a user to purchase an existing NFT.

It renders a form where a user can enter all important information to purchase the NFT:

- **Token Name**(pre-filled): The name of the NFT.
- **Token ID**(pre-filled): The ID of the NFT.
- **Purchase Value**: The value the purchaser wants to pay for the NFT. For assistance, the minimum valid purchase margin for this particular NFT is displayed below.
- **Fee**: The transaction fee for the `purchaseNFT` transaction.
- **Passphrase**: The passphrase of the account which purchases the NFT.

It then uses the `purchaseNFTToken()` function we defined in the Purchase NFT section to create the `purchaseNFT` transaction and the `sendTransactions()` function from the API related functions section to post the transaction to the blockchain application.

frontend_app/src/components/dialogs/PurchaseNFTTokenDialog.js

```
import React, { Fragment, useContext, useState } from "react";
import {
  Dialog,
  DialogTitle,
  DialogContent,
  TextField,
  Button,
  DialogActions,
} from "@material-ui/core";
import { makeStyles } from "@material-ui/core/styles";
import { NodeInfoContext } from "../../context";
import { purchaseNFTToken } from "../../utils/transactions/purchase_nft_token";
import * as api from "../../api";
import { transactions } from "@liskhq/lisk-client";

const useStyles = makeStyles((theme) => ({
  root: {
    "& .MuiTextField-root": {
      margin: theme.spacing(1),
    },
  },
}));

export default function PurchaseNFTTokenDialog(props) {
  const nodeInfo = useContext(NodeInfoContext);
  const classes = useStyles();
  const currentValue = parseFloat(
    transactions.convertBeddowsToLSK(props.token.value)
  );
  const minPurchaseMargin = parseFloat(props.token.minPurchaseMargin);
  const minPurchaseValue =
    currentValue + (currentValue * minPurchaseMargin) / 100.0;

  const [data, setData] = useState({
    name: props.token.name,
    nftId: props.token.id,
    purchaseValue: "",
    fee: "",
```

```
    passphrase: "",
  });

  const handleChange = (event) => {
    event.persist();
    setData({ ...data, [event.target.name]: event.target.value });
  };

  const handleSend = async (event) => {
    event.preventDefault();

    const res = await purchaseNFTToken({
      ...data,
      networkIdentifier: nodeInfo.networkIdentifier,
      minFeePerByte: nodeInfo.minFeePerByte,
    });
    await api.sendTransactions(res.tx);
    props.handleClose();
  };

  return (
    <Fragment>
      <Dialog open={props.open} onBackdropClick={props.handleClose}>
        <DialogTitle id="alert-dialog-title">
          {"Purchase NFT"}
        </DialogTitle>
        <DialogContent>
          <form className={classes.root} noValidate autoComplete="off">
            <TextField
              label="Token Name"
              value={data.name}
              name="name"
              onChange={handleChange}
              fullWidth
            />
            <TextField
              label="Token ID"
              value={data.nftId}
              name="nftId"
              onChange={handleChange}
```

```
            fullWidth
          />
          <TextField
            label="Purchase Value"
            value={data.purchaseValue}
            name="purchaseValue"
            onChange={handleChange}
            helperText={`Minimum purchase value: ${minPurchaseValue}`}
            fullWidth
          />
          <TextField
            label="Fee"
            value={data.fee}
            name="fee"
            onChange={handleChange}
            fullWidth
          />
          <TextField
            label="Passphrase"
            value={data.passphrase}
            name="passphrase"
            onChange={handleChange}
            fullWidth
          />
        </form>
      </DialogContent>
      <DialogActions>
        <Button onClick={handleSend}>Purchase NFT</Button>
      </DialogActions>
    </Dialog>
  </Fragment>
 );
}
```

7.4.4. Transfer funds dialog

The transfer funds dialog allows a user to transfer tokens from one account to another.

It renders a form where a user can enter all important information to transfer the tokens:

- **Recipient Address**: The base 32 address of the account which receives the funds.
- **Amount**: The amount of tokens being transfered.
- **Fee**: The transaction fee for the `transfer` transaction.
- **Passphrase**: The passphrase of the account which sends the funds.

It then uses the `transfer()` function we defined in the Transfer funds section to create the `transfer` transaction and the `sendTransactions()` function from the API related functions section to post the transaction to the blockchain application.

frontend_app/src/components/dialogs/TransferFundsDialog.js

```
import React, { Fragment, useContext, useState } from "react";
import {
  Dialog,
  DialogTitle,
  DialogContent,
  TextField,
  Button,
  DialogActions,
} from "@material-ui/core";
import { makeStyles } from "@material-ui/core/styles";
import { NodeInfoContext } from "../../context";
import { transfer } from "../../utils/transactions/transfer";
import * as api from "../../api";

const useStyles = makeStyles((theme) => ({
  root: {
    "& .MuiTextField-root": {
      margin: theme.spacing(1),
    },
  },
}));

export default function TransferFundsDialog(props) {
  const nodeInfo = useContext(NodeInfoContext);
  const classes = useStyles();
  const [data, setData] = useState({
    recipientAddress: "",
```

```
    passphrase: "",
    amount: "",
    fee: "",
  });

  const handleChange = (event) => {
    event.persist();
    setData({ ...data, [event.target.name]: event.target.value });
  };

  const handleSend = async (event) => {
    event.preventDefault();

    const res = await transfer({
      ...data,
      networkIdentifier: nodeInfo.networkIdentifier,
      minFeePerByte: nodeInfo.minFeePerByte,
    });
    await api.sendTransactions(res.tx);
    props.handleClose();
  };

  return (
    <Fragment>
      <Dialog open={props.open} onBackdropClick={props.handleClose}>
        <DialogTitle id="alert-dialog-title">{"Transfer Funds"}</DialogTitle>
        <DialogContent>
          <form className={classes.root} noValidate autoComplete="off">
            <TextField
              label="Recipient Address"
              value={data.recipientAddress}
              name="recipientAddress"
              onChange={handleChange}
              fullWidth
            />
            <TextField
              label="Amount"
              value={data.amount}
              name="amount"
              onChange={handleChange}
```

```
            fullWidth
          />
          <TextField
            label="Fee"
            value={data.fee}
            name="fee"
            onChange={handleChange}
            fullWidth
          />
          <TextField
            label="Passphrase"
            value={data.passphrase}
            name="passphrase"
            onChange={handleChange}
            fullWidth
          />

          <Button
            onClick={() => {
              setData({
                ...data,
                passphrase:
                  "peanut hundred pen hawk invite exclude brain chunk gadget wait wrong ready",
              });
            }}
          >
            Use Genesis Account
          </Button>
        </form>
      </DialogContent>
      <DialogActions>
        <Button onClick={handleSend}>Send Funds</Button>
      </DialogActions>
    </Dialog>
  </Fragment>
 );
}
```

7.4.5. Transfer NFT dialog

The transfer NFT dialog allows the owner of a NFT to transfer the NFT.

It renders a form where the current owner can enter all important information to transfer the NFT:

- **Token Name**(pre-filled): The name of the NFT.
- **Token ID**(pre-filled): The ID of the NFT.
- **Recipient Address**: The base 32 address of the account which receives the NFT.
- **Fee**: The transaction fee for the `purchaseNFT` transaction.
- **Passphrase**: The passphrase of the owner of the NFT.

It then uses the `transferNFT()` function we defined in the Transfer NFT section to create the `transferNFT` transaction and the `sendTransactions()` function from the API related functions section to post the transaction to the blockchain application.

frontend_app/src/components/dialogs/TransferNFTDialog.js

```
import React, { Fragment, useContext, useState } from "react";
import {
  Dialog,
  DialogTitle,
  DialogContent,
  TextField,
  Button,
  DialogActions,
} from "@material-ui/core";
import { makeStyles } from "@material-ui/core/styles";
import { NodeInfoContext } from "../../context";
import { transferNFT } from "../../utils/transactions/transfer_nft";
import * as api from "../../api";

const useStyles = makeStyles((theme) => ({
  root: {
    "& .MuiTextField-root": {
      margin: theme.spacing(1),
    },
  },
}));
```

```
export default function TransferNFTDialog(props) {
  const nodeInfo = useContext(NodeInfoContext);
  const classes = useStyles();

  const [data, setData] = useState({
    name: props.token.name,
    nftId: props.token.id,
    recipientAddress: "",
    fee: "",
    passphrase: "",
  });

  const handleChange = (event) => {
    event.persist();
    setData({ ...data, [event.target.name]: event.target.value });
  };

  const handleSend = async (event) => {
    event.preventDefault();

    const res = await transferNFT({
      ...data,
      networkIdentifier: nodeInfo.networkIdentifier,
      minFeePerByte: nodeInfo.minFeePerByte,
    });
    await api.sendTransactions(res.tx);
    props.handleClose();
  };

  return (
    <Fragment>
      <Dialog open={props.open} onBackdropClick={props.handleClose}>
        <DialogTitle id="alert-dialog-title">
          {"Transfer NFT"}
        </DialogTitle>
        <DialogContent>
          <form className={classes.root} noValidate autoComplete="off">
            <TextField
              label="Token Name"
```

```
            value={data.name}
            name="name"
            onChange={handleChange}
            fullWidth
          />
          <TextField
            label="Token ID"
            value={data.nftId}
            name="nftId"
            onChange={handleChange}
            fullWidth
          />
          <TextField
            label="Recipient address"
            value={data.recipientAddress}
            name="recipientAddress"
            onChange={handleChange}
            helperText={`Address of the account that will receive the NFT.`}
            fullWidth
          />
          <TextField
            label="Fee"
            value={data.fee}
            name="fee"
            onChange={handleChange}
            fullWidth
          />
          <TextField
            label="Passphrase"
            value={data.passphrase}
            name="passphrase"
            onChange={handleChange}
            fullWidth
          />
        </form>
      </DialogContent>
      <DialogActions>
        <Button onClick={handleSend}>Transfer NFT</Button>
      </DialogActions>
    </Dialog>
```

```
    </Fragment>
  );
}
```

7.5. Components

In React, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

For the frontend we define the following components:

- `HomePage`: A component for rendering The home page.
- `NFTToken`: A component for rendering an NFT including its details and related actions.
- `TransactionsPage`: A component for rendering The transactions explorer.
- `AccountPage`: A component for rendering the account page.
- `Account`: A component for rendering details to a specific account.

The code examples for the `HomePage` and `NFTToken` components can be found below.

Tip

Account addresses in the frontend

As you may be aware, the account address can be represented in different formats, such as `bytes`, `Buffer`, `hex` or `Lisk32`.

The `Lisk32` format is the most human-readable representation of an account address, and therefore users of the application should only see this address in the UI.

More information about the different data formats can be found on the Codec & schema reference page.

7.5.1. NFT component

The NFT component renders the following information:

- **Name**: The name of the NFT as the title.
- **Token ID**: The unique ID of the NFT.
- **Token value**: The current value of the token.
- **Minimum Purchase Margin**: The minimum margin in %, which is added to the token value on purchase.
- **Current owner**: The current owner of the NFT.
- **NFT history**: The complete owner history of the NFT.

This component makes use of the previously created `PurchaseNFTTokenDialog` and `TransferNFTDialog` Dialogs and attaches them at the bottom of the NFT.

frontend_app/src/components/NFTToken.js

```
import React, { useState } from "react";
import {
  Card,
  CardContent,
  CardActions,
  Typography,
  Link,
  Divider,
  Button,
} from "@material-ui/core";
import { makeStyles } from "@material-ui/core/styles";
import { Link as RouterLink } from "react-router-dom";
import { transactions, cryptography, Buffer } from "@liskhq/lisk-client";

import PurchaseNFTTokenDialog from "./dialogs/PurchaseNFTTokenDialog";
import TransferNFTDialog from "./dialogs/TransferNFTDialog";

const useStyles = makeStyles((theme) => ({
  propertyList: {
    listStyle: "none",

    "& li": {
      margin: theme.spacing(2, 0),
      borderBottomColor: theme.palette.divider,
      borderBottomStyle: "solid",
```

```
      borderBottomWidth: 1,

      "& dt": {
        display: "block",
        width: "100%",
        fontWeight: "bold",
        margin: theme.spacing(1, 0),
      },
      "& dd": {
        display: "block",
        width: "100%",
        margin: theme.spacing(1, 0),
      },
    },
  },
}));

export default function NFTToken(props) {
  const classes = useStyles();
  const [openPurchase, setOpenPurchase] = useState(false);
  const [openTransfer, setOpenTransfer] = useState(false);
  const base32UIAddress = cryptography.getBase32AddressFromAddress(Buffer.from(props.item.ownerAddress,
'hex'), 'lsk').toString('binary');
  return (
    <Card>
      <CardContent>
        <Typography variant="h6">{props.item.name}</Typography>
        <Divider />
        <dl className={classes.propertyList}>
          <li>
            <dt>Token ID</dt>
            <dd>{props.item.id}</dd>
          </li>
          <li>
            <dt>Token value</dt>
            <dd>{transactions.convertBeddowsToLSK(props.item.value)}</dd>
          </li>
          <li>
            <dt>Minimum Purchase Margin</dt>
            <dd>{props.item.minPurchaseMargin}</dd>
```

```
          </li>
          {!props.minimum && (
            <li>
              <dt>Current Owner</dt>
              <dd>
                <Link
                  component={RouterLink}
                  to={`/accounts/${base32UIAddress}`}
                >
                  {base32UIAddress}
                </Link>
              </dd>
            </li>
          )}
        </dl>
        <Typography variant="h6">NFT History</Typography>
        <Divider />
        {props.item.tokenHistory.map((base32UIAddress) => (
          <dl className={classes.propertyList}>
            <li>
              <dd>
                <Link
                  component={RouterLink}
                  to={`/accounts/${base32UIAddress}`}
                >
                  {base32UIAddress}
                </Link>
              </dd>
            </li>
          </dl>
        ))}

      </CardContent>
      <CardActions>
        <>
          <Button
            size="small"
            color="primary"
            onClick={() => {
              setOpenTransfer(true);
```

```
            }}
          >
            Transfer NFT
          </Button>
          <TransferNFTDialog
            open={openTransfer}
            handleClose={() => {
              setOpenTransfer(false);
            }}
            token={props.item}
          />
        </>
        {props.item.minPurchaseMargin > 0 ? (
          <>
            <Button
              size="small"
              color="primary"
              onClick={() => {
                setOpenPurchase(true);
              }}
            >
              Purchase NFT
            </Button>
            <PurchaseNFTTokenDialog
              open={openPurchase}
              handleClose={() => {
                setOpenPurchase(false);
              }}
              token={props.item}
            />
          </>
        ) : (
          <Typography variant="body">Can't purchase this token</Typography>
        )}
      </CardActions>
    </Card>
  );
}
```

7.5.2. Home page component

The home page component renders the following information:

A list of all existing NFTs, rendered as an NFT component.

The NFTs are fetched by utilizing the `fetchAllNFTTokens()` function from the API related functions.

frontend_app/src/components/HomePage.js

```
import React, { Fragment, useEffect, useState } from "react";
import NFTToken from "./NFTToken";
import { Grid } from "@material-ui/core";
import { fetchAllNFTTokens } from "../api";

function HomePage() {
  const [NFTAccounts, setNFTAccounts] = useState([]);

  useEffect(() => {
    async function fetchData() {
      setNFTAccounts(await fetchAllNFTTokens());
    }
    fetchData();
  }, []);

  return (
    <Fragment>
      <Grid container spacing={4}>
        {NFTAccounts.map((item) => (
          <Grid item md={4}>
            <NFTToken item={item} key={item.id} />
          </Grid>
        ))}
      </Grid>
    </Fragment>
  );
}

export default HomePage;
```

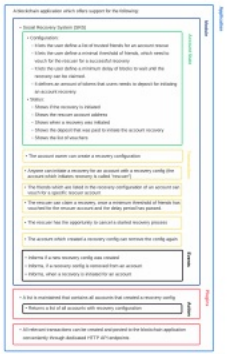
Summary

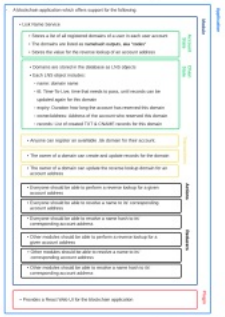
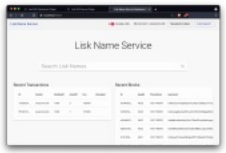
That's it! You should now have a complete example NFT blockchain application running, including:

- a blockchain application with the following:
  - a custom module for handling NFTs.
  - a custom plugin that adds additional features for the NFTs.
- a frontend application which allows you to use and test the applications in the browser.

To become both familiar and fully conversant with the Lisk SDK, a step-by-step walk through guide is provided that simplifies the process of developing a proof of concept blockchain application.

Name	Estimated time required	Complexity	Content
Hello World 	~1-2 h	Beginner	Follow the development guides to learn how to perform the following: • ...bootstrap a simple blockchain application. • ...create a custom module. • ...create a custom plugin. • ...configure a blockchain application. • ...connect a dashboard to the application. • ...write unit tests modules and assets.
Non-fungible token (NFT) Tutorial 	~2-3 h	Beginner	Learn how to create: • ...a blockchain application that allows a user to create, transfer and purchase NFTs. • ...3 different transaction assets to create, transfer, and purchase NFTs. • ...a custom plugin which adds a new HTTP API that serves NFT-related data to the application. • ...a frontend application that allows the user to utilize the blockchain application in the browser.

Name	Estimated time required	Complexity	Content
Social Recovery System (SRS) Tutorial  	~4 h	Intermediate	Learn how to create: • ...a blockchain application that offers users the opportunity to recover their accounts if they have lost their credentials through a social recovery system. • ...6 different transaction assets to manage the recovery process. • ...2 custom plugins that provide new actions and API endpoints which are helpful in the frontend. • ...a frontend application that allows the user to utilize the blockchain application in the browser.

Name	Estimated time required	Complexity	Content
Lisk Name Service (LNS) Tutorial  	~4 h	Intermediate	Learn how to: • ...create a blockchain application with a module which provides a name service for Lisk addresses. The Lisk Name Service allows users to register domain names for their accounts. It can then resolve the human readable domain names to their corresponding account address, making the domain name a human readable alias for the address. • ...create three different transaction assets: ◦ Register: To register a new domain name. ◦ Reverse Lookup: To define the reverse lookup for an account address. ◦ Update Records: To update the records of a domain name. • ...connect the Dashboard plugin to interact with the LNS app during development. • ...create a plugin that provides a React.js frontend for the LNS blockchain application. • ...extend the LNS application CLI with additional commands. • ...write unit tests for the newly implemented transactions assets. • ...write functional tests for the LNS module.

Using a plugin as frontend

The SDK Dashboard plugin is convenient to use during development of the blockchain application, as it offers a simple way to interact with the blockchain application through a user interface. To make the application more user-friendly, add a frontend which is specialized for the respective use case of the LNS application.

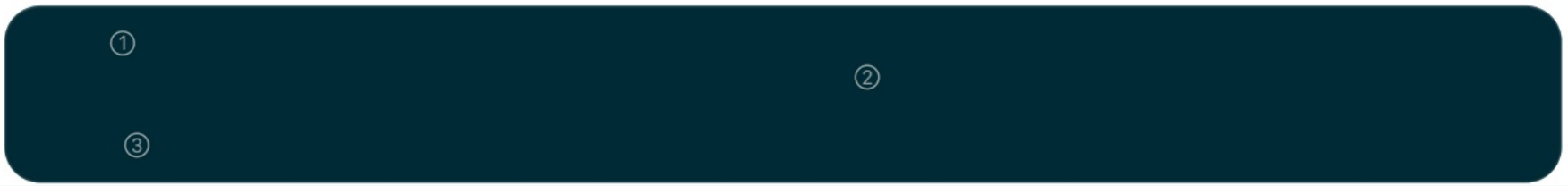
For the LNS app, we want to provide a simple , which is registered to the LNS blockchain application. See the Frontend & Backend section for more information about the different possibilities to provide a frontend for a blockchain application.

The React frontend application

A simple React.js web application is used as a frontend. The development of the React application is not covered in detail in this tutorial. Instead, use the existing React frontend in the lisk-sdk-examples repository and see how it can be included as a standalone UI plugin for a blockchain application.

Tip

If you wish to learn more about how to develop a React frontend application, check out the React.js documentation.



① Navigate out of the folder.

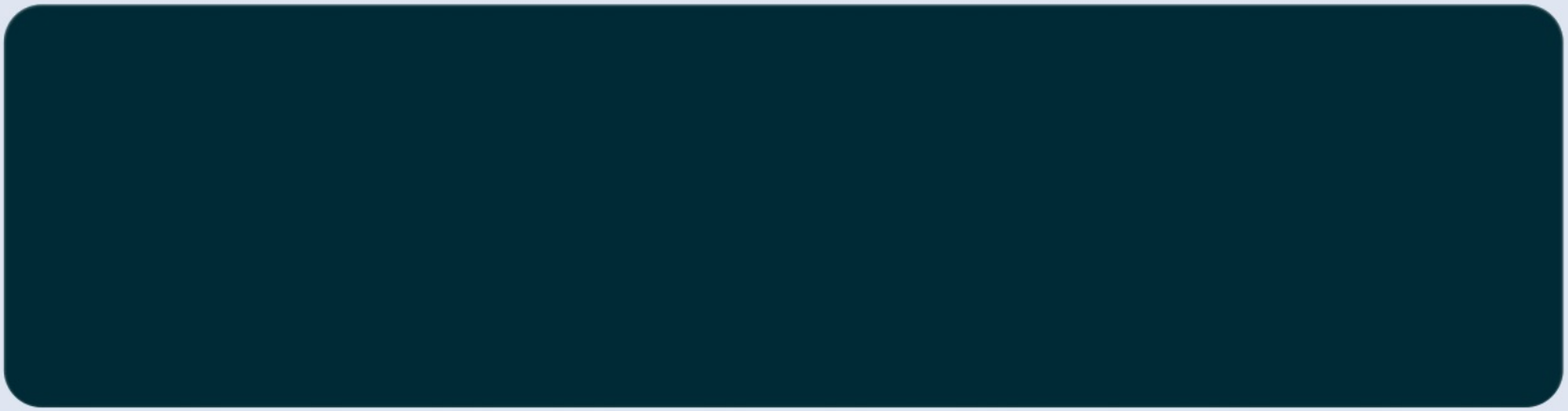
② Clone the repository.

③ Copy the LNS Dashboard plugin to the root folder of your LNS blockchain application.

As you may notice when viewing the files, the LNS Dashboard plugin is based on the code of the Dashboard plugin from the Lisk SDK Framework.

The file structure of the UI plugin is basically a merge of a React.js application and a Lisk plugin. The main logic of the React application is located under , and the main logic of the LNS UI Plugin is located in .

It is not necessary to generate any new files, as the process has been simplified by downloading the prepared frontend plugin for the LNS blockchain application. However, in the case whereby you wish to create your own standalone plugin from scratch, it is possible to use Lisk Commander to generate a plugin skeleton, just as was done when generating the Modules and [assets] skeletons:



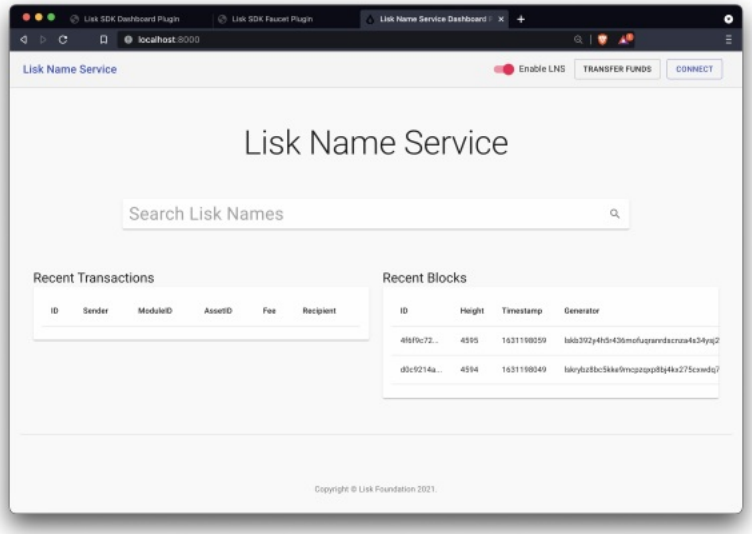
Adding the plugin to the LNS blockchain application

Open the file and add the downloaded LNS Dashboard plugin to the dependencies:

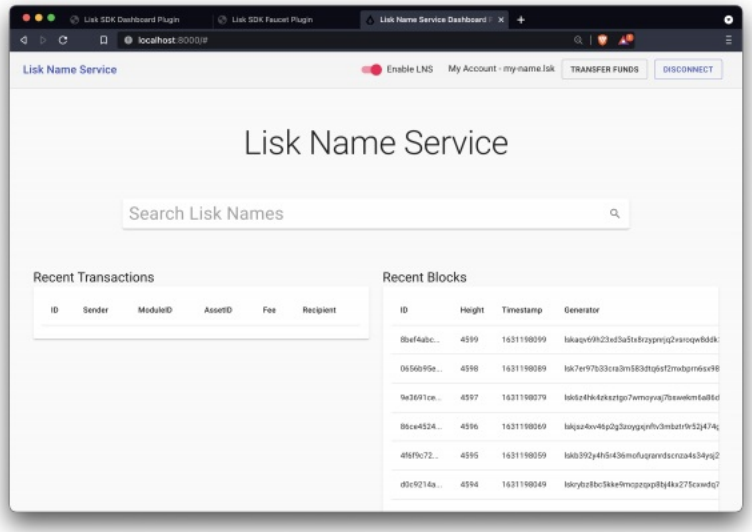
Now open , import the UI plugin, and register it with the application as shown below:

Save and close . Restart the LNS blockchain application to apply the changes.

After the application has loaded, it is possible to acceess the LNS Dashboard under http://localhost:8000.



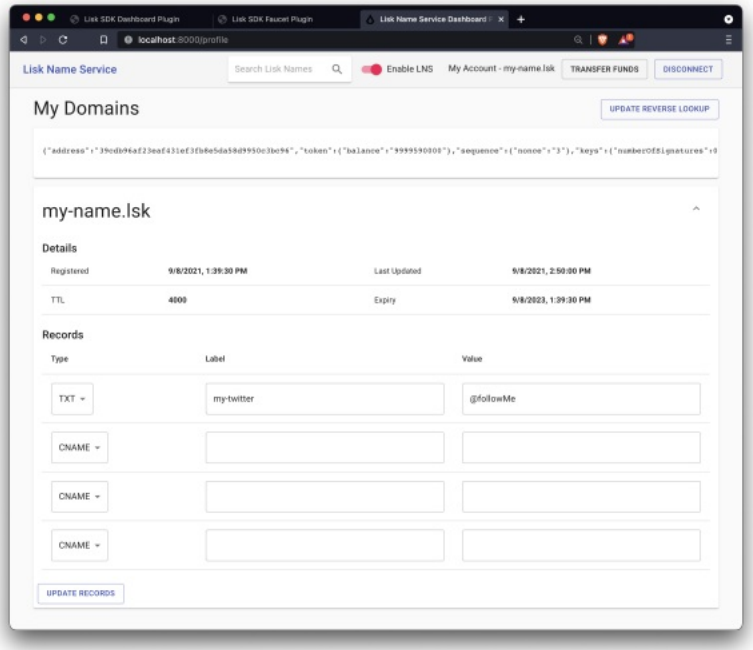
Use the credentials of the account that was newly created in the previous step Create new account of Chapter 1 to connect to the LNS frontend.



In the top right corner you will now see the domain that was defined as reverse lookup for the account address. If you tick off the slider , the address of the account will be displayed again, instead of the domain name.

By clicking on your account you will reach the following page, which contains all important information about your account, and the domain names that are registered for this account. Currently, one domain name is registered to the account. This was done in the step Register Domain of Chapter 1 via the Dashboard plugin.

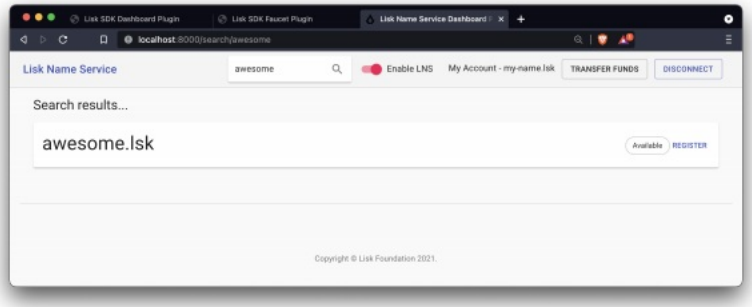
The new TXT record, which was added to the LNS object, can also be seen.



Go back to the index page of the LNS Dashboard and search for a new domain name to add to your account.

The LNS Dashboard will automatically check its' availability. If no other user has registered this domain at the moment, it will provide a link to a dialog to register this new domain.

In the screenshot below, searching for the domain `awesome.lsk`is depicted, and as can be seen here, it is still available.

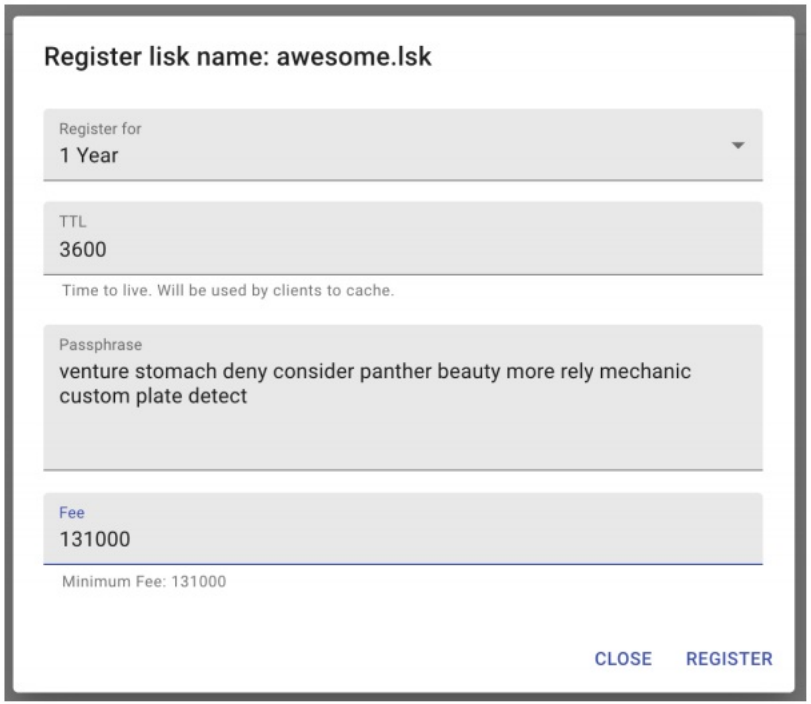


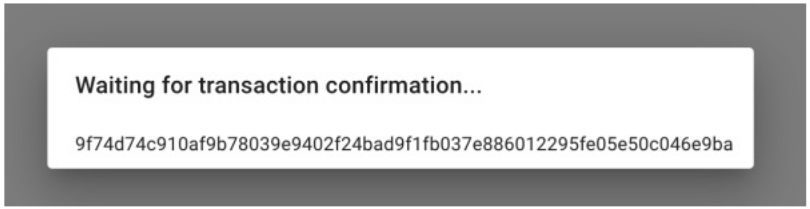
Click on the link to open the dialog to register the domain.

The minimum fee of the Register transaction is calculated automatically, after all required transaction data is pasted in the fields. Please ensure to always use at least the minimum fee for the transaction, or it will be rejected by the blockchain application.

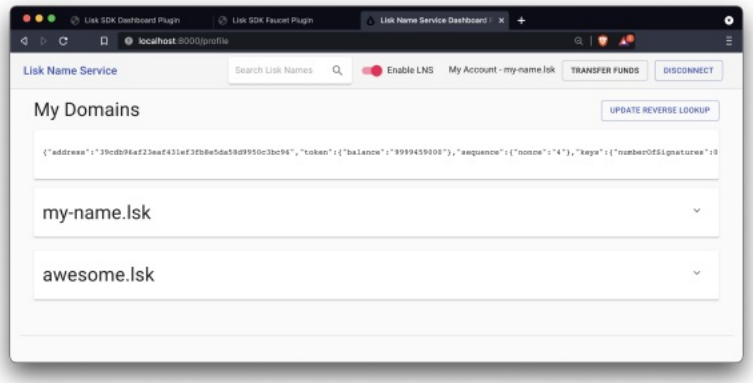
Click on the button to send the domain name registration to the LNS blockchain application.

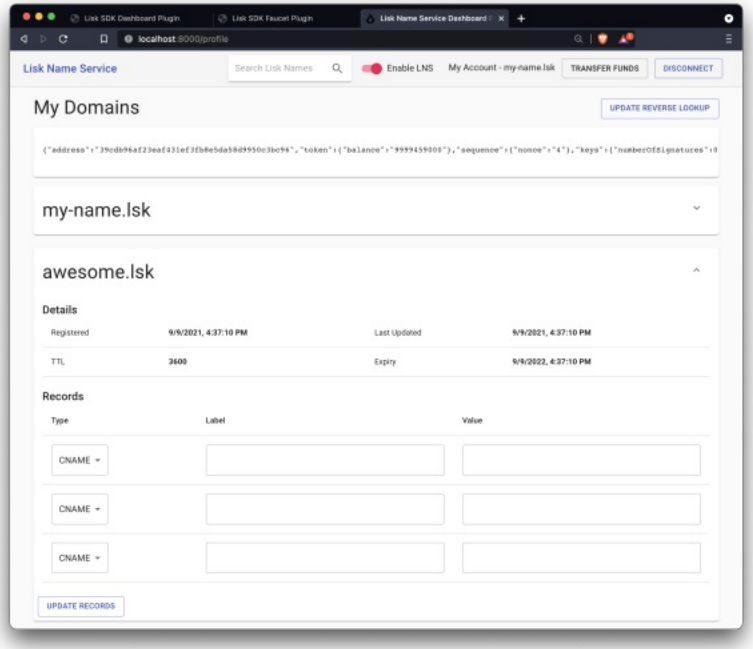
Wait for confirmation that the LNS app has received your transaction.





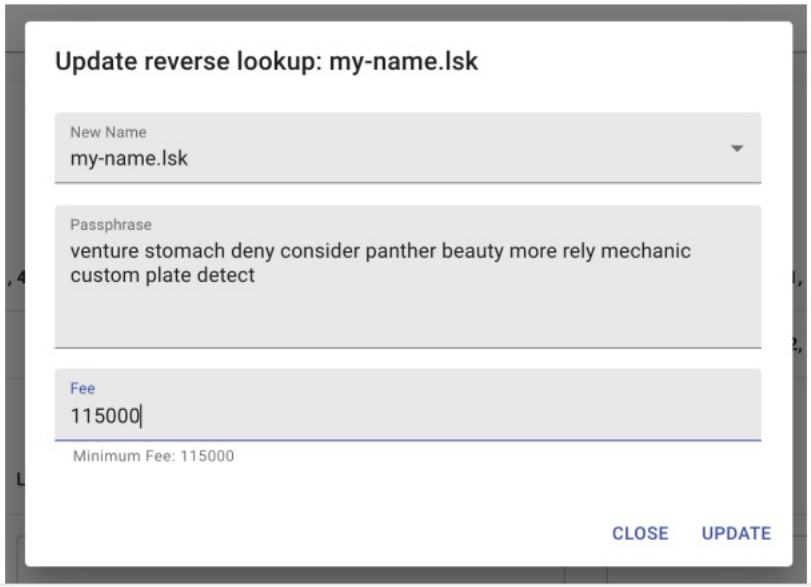
Now return back to your account page. The new domain name should now be visible there.





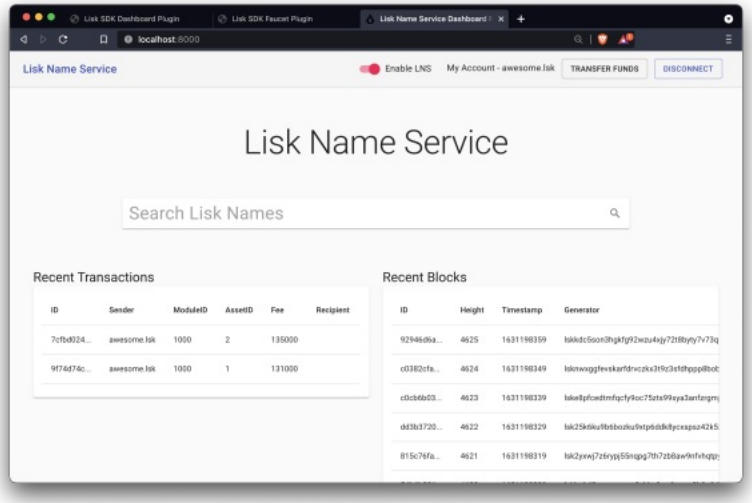
Unfortunately, as can be seen on the top right, our address is still resolving to the domain (if LNS is enabled). So now update the reverse lookup of the account to point to the new domain name .

Open the Dialog for updating the reverse lookup by clicking on the button in the top right.

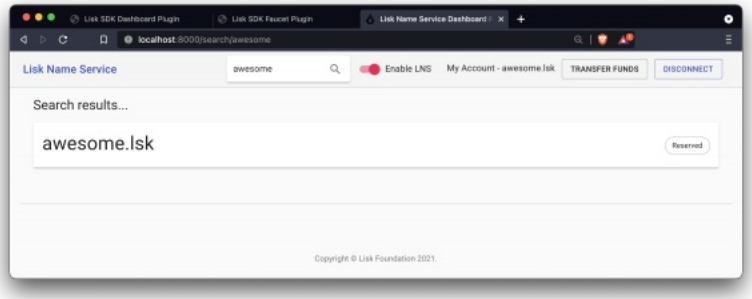


Choose from the dropdown menu, enter the account passphrase and the minimum fee.

Click the button to update the reverse lookup entry for this account. As a result, it is possible to verify that the LNS Dashboard now resolves the account address automatically to the new domain name .



Try to search again for the domain name . The search results should now indicate that this domain name is already reserved.



It is recommended to fully familiarize yourself with the LNS application UI before moving on.

The development of the frontend and backend of the LNS application is now complete, and the application has all the features that were described in the LNS application overview at the top.

In the next chapter, the LNS application CLI will be enhanced with commands specific to the LNS module. This allows interaction with the blockchain application directly via the command-line, which can be beneficial for developers and/or node operators.