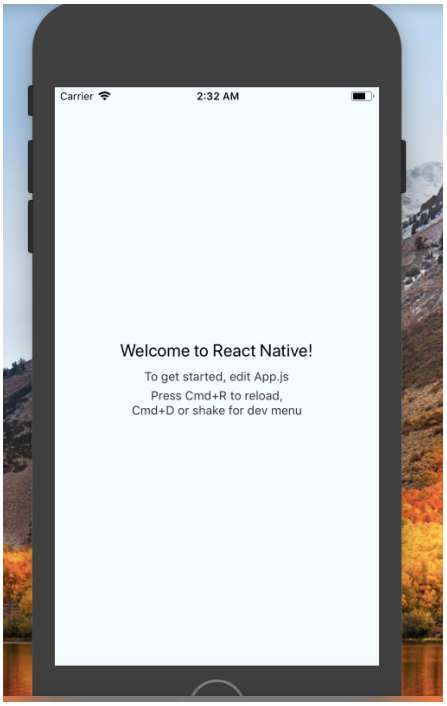
How to Build an NCNT Mobile Wallet Using Our Developer Tools

This walkthrough will guide you through the steps of building your own nCent mobile wallet.

You can use this to recreate our app, or modify these steps as needed to accommodate your own application ideas.

Step 1 - Setup a React Native Mobile Development Environment



We used a React Native environment for development of our Mobile Wallet.

1. **Install node.js**
    a. Download here.
    b. Run the following commands on a separate terminal window to make sure installation succeeded
        i. npm -v
        ii. node -v
        iii. Pro tip: npm install --save will automatically save anything you npm install to the dependency section of your package.json file
2. **Clone the nCent public repo and create a mobile wallet folder:**
    a. git clone https://github.com/ncent/ncent.github.io.git
    b. cd ncent.github.io
3. **Set up React Native:**

    a. Run the following command in the directory where you want to place your application folder:
        i. npm install -g react-native-cli
    b. Download Xcode via the Mac App store
    c. Open Xcode, choose preferences, go to the locations panel, and install the most recent tools from the command line drop-down menu
    d. From the the ncent.github.io root directory, run the following command
        i. react-native init MobileWallet
4. Run your application and it should pop up in a simulator. You can also run your application directly from xCode
    a. cd MobileWallet
    b. react-native run-ios

Resources:
There are a number of resources online to help you get started with react native:

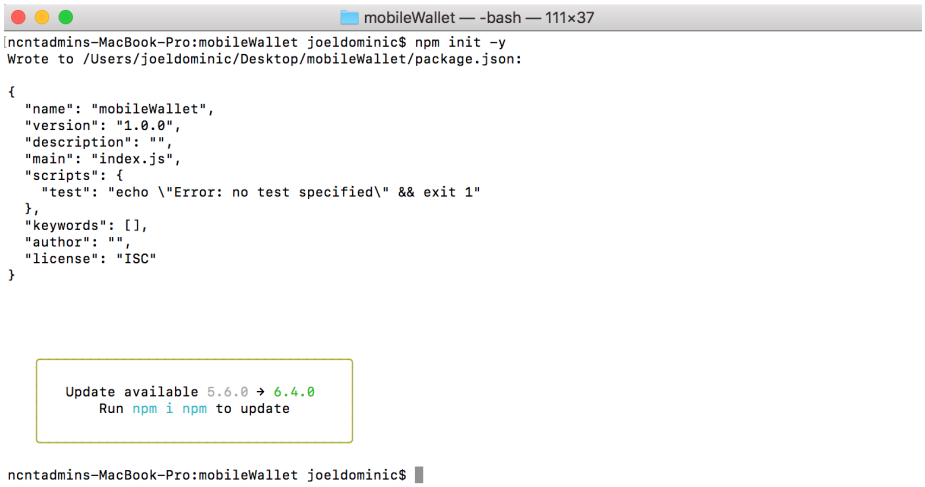
- Official React Native docs
- Official React Native "getting started" guide
- React Native tutorial video

PRO-TIP: Be careful using Create React Native App as some functions in the SDK require native code. Use React Native Cli instead, or you'll eventually have to detach.

Step 2 - Install and Setup Necessary Packages

We'll need to set up the following packages to begin development:

1. **Wallet dependencies**
    a. Navigate to the MobileWallet directory
        i. cd MobileWallet
    b. Initialize client-level node dependencies and packages.json file
        i. npm init -y



   c. Install the nCent SDK
      i. npm i ncent-sdk-public

2. **PostgreSQL:**
   a. Use this resource to gain access to the PSQL shell terminal
   b. Set defaults as follows: port=5432, password="dickey", user="postgres"
3. **nCent sandbox dependencies:**
   a. Install the Sandbox dependencies by navigating to the ncent.github.io root directory and running the following commands:
      i. cd Sandbox/Sandbox\ API
      ii. npm install

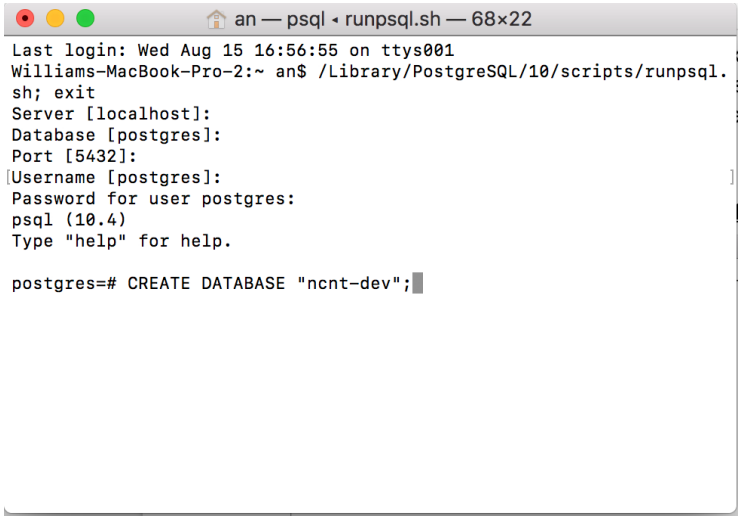
Additional Resources:

- nCent SDK installation tutorial
- Install PostgreSQL & Setup First Database Tutorial
- nCent Sandbox repository

Step 3 - Setup our Sandbox Environment Database Locally

Test your application using a local instance of our Sandbox.

1. Open PSQL shell and login with the permissions you set up in step 2
2. List all the databases by the following command in the shell using the following command:
   a. \l
3. Create a database instance for the Sandbox with the command:
   a. CREATE DATABASE "ncnt-dev";



4. Connect to your newly created database instance:
    a. \connect “ncnt-dev”
5. If you configured your permissions in step 2 differently, go to **Sandbox/Sandbox\ API/config/config.json**. Ensure that your information in the “development” object matches your setup in PostgreSQL

```
{
  "development": {
    "username": "postgres",
    "password": "dickey",
    "database": "ncnt-dev",
    "host": "127.0.0.1",
    "port": 5432,
    "dialect": "postgres"
  },
  "test": {
    "username": "postgres",
    "password": "dickey",
    "database": "ncnt-test",
    "host": "127.0.0.1",
    "port": 5432,
    "dialect": "postgres"
  }
}
```

6. Now, let’s migrate the schema into the database. In your current terminal, make your current directory **server** under **Sandbox\ API** and run:
    a. node_modules/.bin/sequelize db:migrate
7. You should see the following in your terminal:

```
admins-MacBook-Pro:server admin$ ../node_modules/.bin/sequelize db:migrate

Sequelize CLI [Node: 8.11.3, CLI: 4.0.0, ORM: 4.37.10]

Loaded configuration file "config/config.json".
Using environment "development".
sequelize deprecated String based operators are now deprecated. Please use Symbol
 based operators for better security, read more at http://docs.sequelizejs.com/ma
nual/tutorial/querying.html#operators ../node_modules/sequelize/lib/sequelize.js:
242:13
== 20180621230529-create-token-type: migrating =======
== 20180621230529-create-token-type: migrated (0.027s)

== 20180622161155-create-transaction: migrating =======
== 20180622161155-create-transaction: migrated (0.018s)

== 20180622225632-create-wallet: migrating =======
== 20180622225632-create-wallet: migrated (0.013s)

admins-MacBook-Pro:server admin$
```

8. Then, go back to the PSQL shell and check that you have the right tables:
    a. \dt
9. Your terminal should have printed the following:

```
ncnt-dev=# \dt
              List of relations
 Schema |     Name       | Type  |  Owner
--------+----------------+-------+----------
 public | SequelizeMeta  | table | postgres
 public | TokenTypes     | table | postgres
 public | Transactions   | table | postgres
 public | Wallets        | table | postgres
(4 rows)
```

10. Run the following command to see that the table is empty:
    a. **select * from “table_name”;**

```
ncnt-dev=# SELECT * FROM "Wallets";
 uuid | wallet_uuid | tokentype_uuid | balance | createdAt | updatedAt
------+-------------+----------------+---------+-----------+-----------
(0 rows)
```

Additional Resources:

- **nCent Tutorial video** (coming soon)
- PostgreSQL documentation
- PostgreSQL video tutorial

PRO-TIP: Do not use **createdb** command in the terminal. Create the database only in PSQL shell with the **CREATE DATABASE “database_name”** command. Using **createdb** will make duplicate databases that are disjointed.

Step 4 - Run the Sandbox and Test the Environment

1. Enter the following command from your Sandbox terminal

   npm run start:dev {path to your Sandbox}
2. You should see the following:

```
> ncnt_api@1.0.0 start:dev /Users/admin/Documents/ncnt/Sandbox/Sandbox API
> nodemon ./bin/www

[nodemon] 1.17.5
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node ./bin/www`
sequelize deprecated String based operators are now deprecated. Please use Symbol based operators for better security, read more at http://docs.sequelizejs.com/manual/tutorial/querying.html#operators node_modules/sequelize/lib/sequelize.js:242:13
Executing (default): CREATE TABLE IF NOT EXISTS "TokenTypes" ("Name" VARCHAR(255) NOT NULL UNIQUE, "uuid" UUID NOT NULL , "ExpiryDate" TIMESTAMP WITH TIME ZONE NOT NULL, "sponsor_uuid" VARCHAR(255) NOT NULL, "totalTokens" INTEGER NOT NULL, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, PRIMARY KEY ("uuid"));
Executing (default): CREATE TABLE IF NOT EXISTS "TokenTypes" ("Name" VARCHAR(255) NOT NULL UNIQUE, "uuid" UUID NOT NULL , "ExpiryDate" TIMESTAMP WITH TIME ZONE NOT NULL, "sponsor_uuid" VARCHAR(255) NOT NULL, "totalTokens" INTEGER NOT NULL, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, PRIMARY KEY ("uuid"));
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'TokenTypes' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'TokenTypes' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): CREATE TABLE IF NOT EXISTS "Transactions" ("uuid" UUID NOT NULL , "amount" INTEGER NOT NULL, "fromAddress" VARCHAR(255) NOT NULL, "toAddress" VARCHAR(255) NOT NULL, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, "tokentype_uuid" UUID REFERENCES "TokenTypes" ("uuid") ON DELETE SET NULL ON UPDATE CASCADE, PRIMARY KEY ("uuid"));
Executing (default): CREATE TABLE IF NOT EXISTS "Transactions" ("uuid" UUID NOT NULL , "amount" INTEGER NOT NULL, "fromAddress" VARCHAR(255) NOT NULL, "toAddress" VARCHAR(255) NOT NULL, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, "tokentype_uuid" UUID REFERENCES "TokenTypes" ("uuid") ON DELETE SET NULL ON UPDATE CASCADE, PRIMARY KEY ("uuid"));
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'Transactions' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'Transactions' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): CREATE TABLE IF NOT EXISTS "Wallets" ("uuid" UUID NOT NULL , "wallet_uuid" VARCHAR(255) NOT NULL, "tokentype_uuid" UUID NOT NULL DEFAULT '498cc1fe-62d3-4863-a0e0-a42049b90fff', "balance" INTEGER NOT NULL DEFAULT 0, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, PRIMARY KEY ("uuid"));
Executing (default): CREATE TABLE IF NOT EXISTS "Wallets" ("uuid" UUID NOT NULL , "wallet_uuid" VARCHAR(255) NOT NULL, "tokentype_uuid" UUID NOT NULL DEFAULT '498cc1fe-62d3-4863-a0e0-a42049b90fff', "balance" INTEGER NOT NULL DEFAULT 0, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, PRIMARY KEY ("uuid"));
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'Wallets' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'Wallets' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): CREATE UNIQUE INDEX "wallets_wallet_uuid_tokentype_uuid" ON "Wallets" ("wallet_uuid", "tokentype_uuid")
Executing (default): CREATE UNIQUE INDEX "wallets_wallet_uuid_tokentype_uuid" ON "Wallets" ("wallet_uuid", "tokentype_uuid")
```

3. If any errors appear enter the following command:

   rs
4. Now, open another terminal and go to the SDK directory in your local git repository.
5. Use the following command to test the tables in the database:

   node test.js
6. You should see the following:

```
ncnt-dev=# SELECT * FROM "Wallets";
                 uuid                 |                wallet_uuid                |            tokentype_uuid            | balance |        createdAt         |        updatedAt
--------------------------------------+-------------------------------------------+--------------------------------------+---------+--------------------------+--------------------------
 0d475e8f-1e78-4b1e-90fc-e067f56ca003 | GBKEY62GRQ4CTEZ4XDUKMXDBHHF34CZE4UVFFRHW4FJCUERXIISNTOFL | dd8f431e-9641-4e57-bfb0-59cc6e65b86d |  999990 | 2018-08-15 17:55:21.093-07 | 2018-08-15 17:55:21.17-07
 9659a087-c752-4626-a1c7-f55adad9ce85 | GCZFSFCAISMRO236EVOXSKU56726OCRARY54KAB6NBJNWW6FIQZKMSVF | dd8f431e-9641-4e57-bfb0-59cc6e65b86d |      10 | 2018-08-15 17:55:21.179-07 | 2018-08-15 17:55:21.182-07
(2 rows)
```

7. Now, we will clear the database. In your terminal with the sandbox is running, enter the following commands:
   a. ctrl-c
   b. node_modules/.bin/sequelize db:migrate:undo:all
   c. node_modules/.bin/sequelize db:migrate
8. Now, your tables will be empty. Check the tables on your shell to see that it is clear.

Resources:

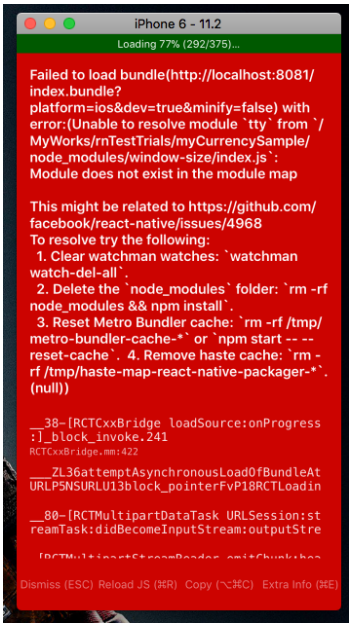
- nCent Video Tutorial on testing the SDK
- Sequalize documentation
- Sequalize tutorial video

PROTIP: Often, when you are testing your backend, it can help to clear your database for easier visibility into how your data is being handled. In order to do this, use the command **DROP DATABASE “table_name”;**

Step 5 - Run the SDK from your React Native App

Try to require the nCentSDK with **require('ncent-sdk-public')** In one of your app's components and run some of its methods. Hopefully, you'll be able to run the functions with no errors. If you can, fantastic! You're ready to start developing with our SDK.

However, you will likely get a haste module error such as the one below, as the SDK uses libraries such as crypto which, in react native, will require linking.



Troubleshooting Linking Errors
If you developed in Expo, it is now time to detach since you cannot link libraries in Expo. To make libraries appear in the haste module map, run the command:
react-native link

However, depending on your exact development environment, you may have to go through additional steps. There are many resources for linking libraries, and you may have to manually link in android or ios environments. There are countless possible errors and this can be a difficult process, but we have complied some examples and links below.

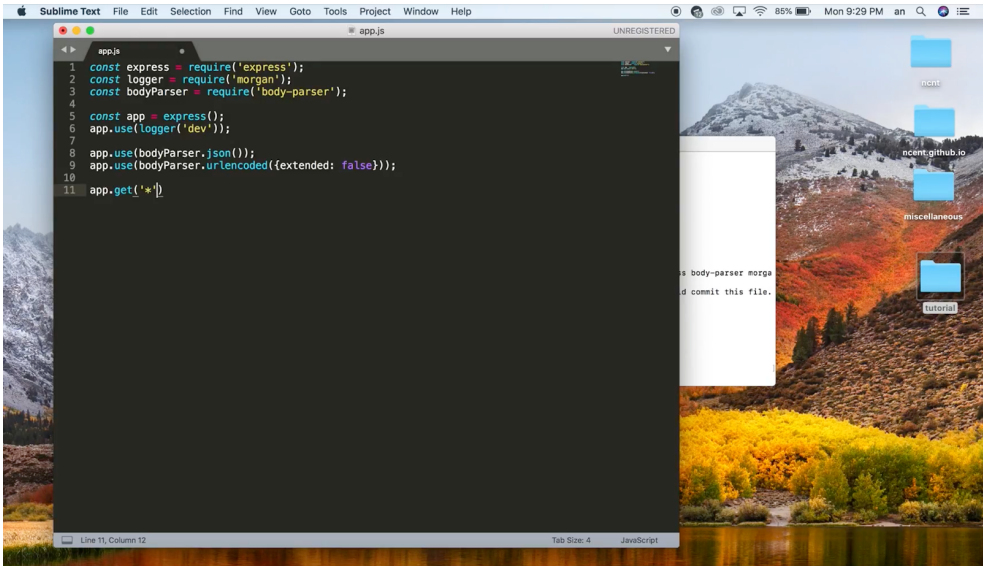
Resources:

- React Native Crypto docs
- Linking libraries on iOS
- Linking on React Native Tutorial video
- nCent Video Tutorial (coming soon)

PRO-TIP: If you are using Pods and xCode, don't forget to run pod install once you've linked to update your pods, and then don't forget to rebuild your project from scratch in your xCode workspace (not proj).

Note: At this point, you are ready to start developing with our SDK!

Step 6 - Set up your own server and database (optional)



Only follow these steps if your application requires a server and a database.

1. **Follow [our tutorial] for setting up a server**
    a. Make the directory for your project. Include a bin and a server folder
    b. Initialize your directory
        i. npm init -y
    c. If you haven't already, install express, body-parser, and morgan npm packages
        i. npm install express body-parser morgan
    d. Create an app.js file like the following:

```
const express = require('express');
const logger = require('morgan');
const bodyParser = require('body-parser');

// Set up the express app
const app = express();

// Log requests to the console.
app.use(logger('dev'));

// Parse incoming requests data (https://github.com/expressjs/body-parser)
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

// Setup a default catch-all route that sends back a welcome message in JSON format.
app.get('*', (req, res) => res.status(200).send({
  message: 'Welcome to the beginning of nothingness.',
}));

module.exports = app;
```

    e. Make a file called 'www' in the bin folder like the following:

```
// This will be our application entry. We'll setup our server here.
const http = require('http');
const app = require('../app'); // The express app we just created

const port = parseInt(process.env.PORT, 10) || 8000;
app.set('port', port);

const server = http.createServer(app);
server.listen(port);
```

   f. Use nodemon so that your server restarts every time you change code
      i. npm i -D nodemon
   g. Navigate to your package.json file, and under the scripts section add
      i. "start:dev": "nodemon ./bin/www"
   h. Run the following command:
      i. npm run start:dev
   i. Navigate to localhost:8000 to see your default message

2. **Ensure you have sequelize installed:**
   a. npm install --save sequelize pg pg-hstore
3. **Initialize sequelize:**
   a. sequelize init
4. **Create your database:**
   a. createdb ‘database_name’
5. **Create your models:**
   a. Execute the following command
      i. sequelize model:create --name ‘{model_name}’ --attributes title:string
   b. This is to create a model with a single string attribute that is the title. You can add more attributes later on in the file or list them in this command
   c. Do this for each model you would like to create
   d. This will also create your migrations
6. **Migrate your database:**
   a. sequelize migrate
7. **Create controllers:**
   a. Require the relevant models
   b. Develop the functionality of the controller by writing the methods you will need.
   c. See example below:

```
const Bug = require('../models').Bug;
const User = require('../models').User;
const bugUser = require('../models').bugUser;
const bcrypt = require('bcrypt');
const path = require('path');
const ncentSDK = require('../../../../../../SDK/source/');
const ncentSdkInstance = new ncentSDK();

module.exports = {
  getBalance(req, res){
    return User
      .findById(req.session.user.uuid, {
      })
      .then(user => {
        console.log('here');
        if (!user) {
          return res.status(404).send({
            message: 'User Not Found',
          });
        }
        return new Promise(function(resolve, reject) {
          return ncentSdkInstance.getTokenBalance(user.email,'9d91db6b-f33a-4392-a583-a6ea14bd368f',resolve);
        })
        .then(data => res.status(200).send(data))
        .catch(error=> console.log(error));

      })
      .catch(error => res.status(400).send(error));
  },
  updateBugPage(req, res){
    res.sendFile(path.resolve(__dirname + '/public/updatebug.html'));
  },
  logOut(req, res){
    if (req.session.user && req.cookies.user_sid) {
      res.clearCookie('user_sid');
    }
    res.sendFile(path.resolve('__dirname' + '../../../../index.html'));
  },
```

8. **Create an index.js file in controllers and export all your controllers**
    a. Navigate to the controllers
        i. cd controllers
    b. Create an index.js file
        i. touch index.js
    c. Add the following to your index.js file:
    ```
    const controller_name = require('./controller_name');

    module.exports = {
        controller_name
    };
    ```
    *Replace controller_name with the name of your controllers. Make sure you do this for all of your controllers.
9. **In the index.js file in the routes folder,**
    a. Require and define all the controllers
    b. Define your routes and the methods they will use from the appropriate controllers.
    c. As an example (where wallet is a model):
        ‘’’
        const walletController = require('../controllers').wallet;
        app.get(‘/getBalance’, walletController.getBalance);
        ‘’’
    d. You must decide whether it will be a post (for creating new information), put (for updating information), or get (for retrieving information), then add the route as a

string for the first parameter and the controller_name.functionName as the second parameter.

10. **Require the routes you just made in the app.js file with this code:**

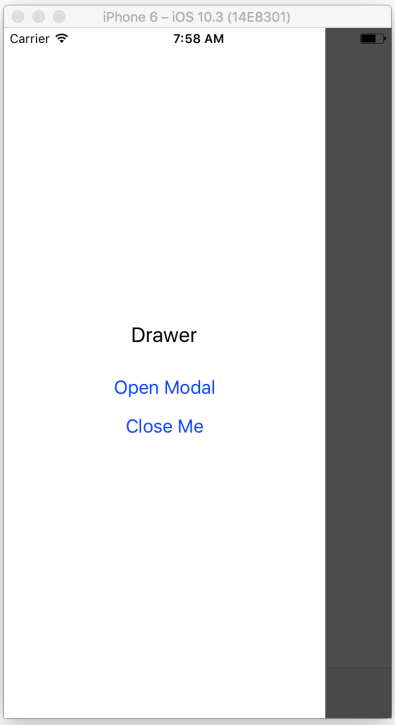
```
require('./server/routes')(app);
```

Additional Resources:

- Getting started with node express and postgres using sequelize
- Our tutorial on setting up a server
- Firebase tutorial video

PRO-TIP: For speed, you can use a free server such as Firebase to get your app running quickly.

Step 7 - Choose a Navigation Scheme



Options:

- **React Navigation (recommended):** A great go-to navigation library. The only downside is that it adds a layer of complexity to integrate Redux, and they will soon not provide supporting documentation for Redux. That being said, the only true issue is the difficulty of calling functions outside components (like when using Redux), and documentation is provided to assist this

- **React Native Router Flux**: A nice and popular implementation of React Navigation, optimized for Redux. Not as well documented and a more questionable long term scheme, but for now it also works. Our app currently uses this library, although we may switch to React Navigation in the future.

To get React Navigation up and running:

1. npm install --save react-navigation
2. Create your navigator in your App component

```
import {
  createStackNavigator,
} from 'react-navigation';

const App = createStackNavigator({
  Home: { screen: HomeScreen },
  Profile: { screen: ProfileScreen },
});

export default App;
```

3. Create a few screens and try navigating

```
class HomeScreen extends React.Component {
  static navigationOptions = {
    title: 'Welcome',
  };
  render() {
    const { navigate } = this.props.navigation;
    return (
      <Button
        title="Go to Jane's profile"
        onPress={() =>
          navigate('Profile', { name: 'Jane' })
        }
      />
    );
  }
}
```

Additional Resources:

- Video tutorial on managing navigation state with Redux
- nCent tutorial video (coming soon)

PRO-TIP: It's probably safer to stick with React Navigation, but if you are having lots of trouble setting up Redux and really want to, React Native Router Flux can get your application running quickly

Step 8 - Test Functionality with Demo Screens

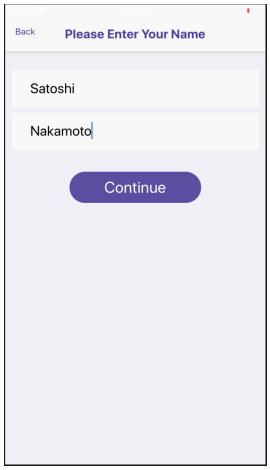
Set up the following components to test for basic SDK functionality. You can make them as simple as you want. These are the bare necessities for a functioning wallet. You'll call the SDK functions from these components.

The following is code for a very basic component. Follow the links in resources to style however you see fit.

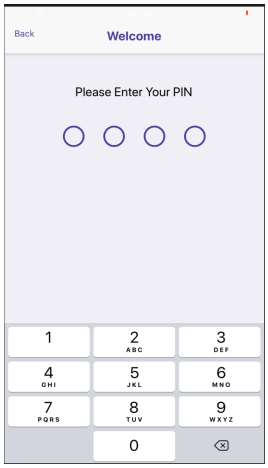
```
import React, { Component } from 'react';
import { Text, View } from 'react-native';

export default class HelloWorldApp extends Component {
  render() {
    return (
      <View>
        <Text>Hello world!</Text>
      </View>
    );
  }
}
```

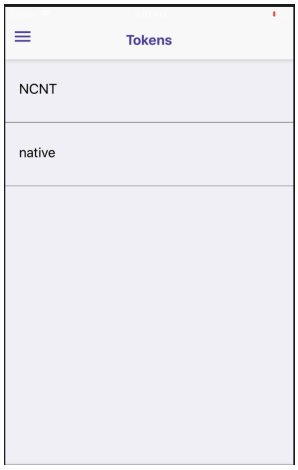
Sign up screen



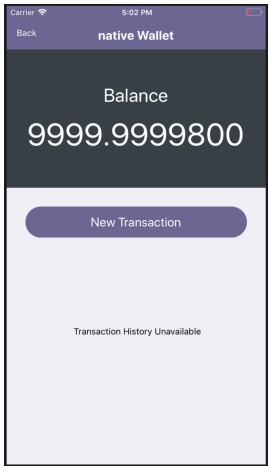
Log in screen



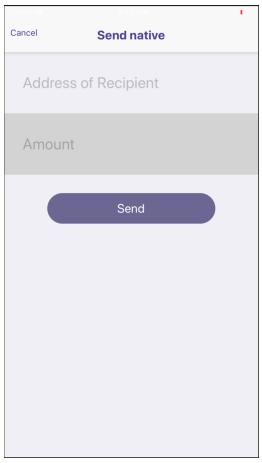
Tokens Screen



Balance screen



Transfer screen



Additional Resources:
Follow these guides to learn to set up and style your components:
https://facebook.github.io/react-native/docs/tutorial
https://facebook.github.io/react-native/docs/style
https://facebook.github.io/react-native/docs/flexbox
https://facebook.github.io/react-native/docs/handling-text-input

Pro-tip: For speed while testing, simplify your authentication system temporarily. This also enables you to test it step by step.

Step 9 - Create Accounts and Authenticate Users

Credential options:

1. A very standard approach is to store user password credentials in a database and only allow a user access with these credentials (see Coinbase).
2. You could also do something more local, such as a pin stored on the device (see Abra). As of now, we authenticate with a pin in our wallet.

Key storage options:

1. Again, you have two options. You can store keys securely in a centralized server to allow for cross device access to an account (see Coinbase).
2. You can also store the user's keys locally to give them autonomy over their keys (see Abra). Local key storage promotes decentralized control and is currently implemented in our wallet.

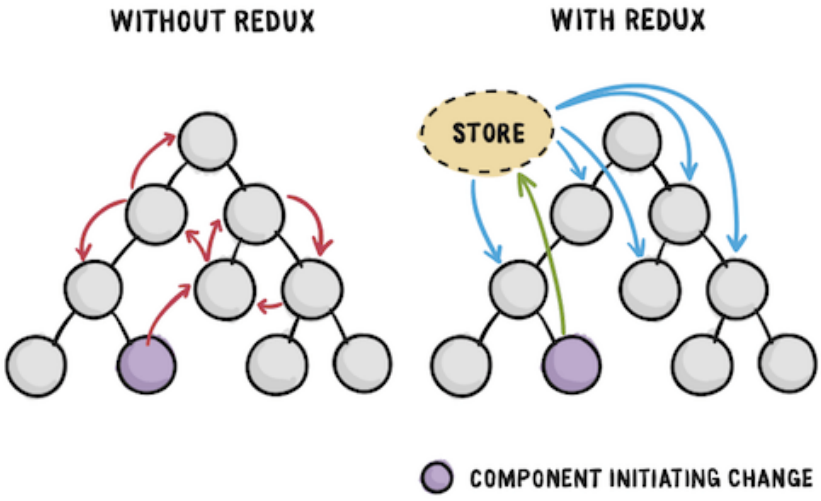
Additional Resources:

- **Secure Local Storage Links in react native:**

    - Expo SecureStore docs
    - React Native Keychain docs
- **Information on Android and IOS secure local storage:**
    - iOS Keychain Services documentation
    - Android Keystore documentation
    - Android shared preferences documentation

PRO-TIP: Many modern apps generate keys deterministically from a randomly generated mnemonic phrase from which the keys can be recovered, allowing users to simply remember a 12 word seed phrase, while providing the same amount of entropy as random Keypair generation.

Step 10 - Set up your application with Redux (optional, advanced)



Redux can appear intimidating, but once it is in place, it organizes the flow of application data very nicely, which proves critical as your application begins to scale. We recommend implementing Redux in your mobile application, although if you are unfamiliar with React Native it may be better to get your project working before using Redux.

1. **Install redux and react-redux**
    a. npm install --save redux react-redux
2. **Import the libraries in your App.js file or wherever you keep your App component**
    a. import { Provider } from 'react-redux';
    b. import { createStore } from 'redux';
3. **Inside of your App component, return a Provider, and pass your provider a store**

a.
```
return (
    <Provider store={createStore}
        //… your app components
    </Provider>

);
```

4. **Create a reducers folder, and in your folder create an index.js file, and in this file**
    a. import { combineReducers } from ‘redux’
    b. Create a dummy reducer
        i.
```
import { combineReducers } from 'redux';

export default combineReducers({
    placeholder: () => [ ]
});
```
    c. Our file structure is eventually going to look like:
        i. Components
            1. TokenDetails.js
        ii. Reducers
            1. TokenDetailsReducer.js
            2. index.js
        iii. Actions
            1. Index.js
        iv. App.js
5. **Now back to our App file, at the top add an import statement and import the dummy reducer from our reducers file**
    a. import {reducers} from ‘./reducers’;
    b. Update our store
        i.
```
import { combineReducers } from 'redux';

export default combineReducers({
    placeholder: () => [ ]
});
```

6. **You’re now all set up with Redux. Now we’ll make an actual reducer now to replace our dummy reducer.** We’ll want a reducer to manage different parts of our application state. You can come back to this part of the tutorial once you’ve got an actual component set up, but now we’ll go over how to update the redux store with different reducers.

    Let’s say we’ve built a page where we want to show specific token details, such as a balance.
    a. Create a new file in our ‘reducers’ folder called TokenDetailsReducer.js

b. In our ‘reducers/index.js’ file,
    i. import TokenDetailsReducer from ‘./TokenDetailsReducer’
    ii. Update the combineReducers call as:

```
export default combineReducers({
    tokenDetails: TokenDetailsReducer
});
```

c. In our ‘reducers/TokenDetailsReducer.js’ file, we’ll export some dummy data for our balance. Let’s just make a function that returns 5. Your entire TokendDetailsReducer file should look like this
    i.

```
export default () => 5;
```

7. In this example, we’re assuming that we already have a component (let’s call it TokenDetails) where we want to show the balance for a specific user’s specific token type. **We now want to ask the reducer we just made for this information from our component.** This way, we store all this information in the “store” we made in step 3 instead of in this component. This is why we use Redux, to move state logic outside the components.

   But this means we need to somehow connect our TokenDetails component to the TokenDetailsReducer. We will use a “connect” helper function.
    a. If your component used to look like this:
        i. import ……;
        class TokenDetails extends Component {
            …...
        }
        export default TokenDetails;

    b. Change it to this, importing connect, and calling the connect function
        i. **import {connect} from ‘react-redux’;**
        import ….;

        class TokenDetails extends Component {
            …..
        }
        export default **connect()(TokenDetails);**

    c. Now we tie redux state and component props together with a mapStateToProps function. Your file should now look like:

i.

```
import {connect} from 'react-redux';
// import ....

class TokenDetails extends Component {
    // .....
}

const mapStateToProps = state => {
    return {balance: state.tokenDetails};
};
export default connect(mapStateToProps)(TokenDetails);
```

d. You should now be able to access the value of 5 returned by our reducer in our component through **this.props.balance**. console.log the balance in your render function to make sure this worked.

8. So now we've moved our balance getting logic outside our component and into the reducer. **Now, let's actually get the balance.** For this, we are going to use an Action. Redux works by having the component call an action which updates the reducer, updating the application state which the component then has access to.
   a. Make an 'actions' folder and in it make an 'index.js' file. We'll dispatch our actions from 'index.js' for now, but best practice is to create different files for different categories of actions and export them through index.js.
   b. In 'actions/index.js', we will create our get balance action with two parameters we assume we know, the user's id that we want the balance for and his token type
      i.

```
export const getTokenBalance (user_id, tokentype_id) => {
    return {
        type: 'get_balance', payload: 6
    }
}
```

   c. Now, when getTokenBalance is called, we are dispatching an action of type 'get_balance' and a payload of 6. Back to our reducer, in 'reducers/TokenDetailsReducer.js', we're going to want to update our state when this action is dispatched. Replace our dummy reducer with this code:
      i.

```
const INITIAL_STATE = {balance: ''};
export default (state = INITIAL_STATE, action) => {
    switch(action.type) {
        case 'get_balance':
            return {...state, balance: action.payload}
        default:
            return state;
    }
}
```

   d. Finally, let's wire up this function to our component. Our TokenDetails component should now look like:

i.

```
import {connect} from 'react-redux';
import {getTokenBalance} from '../Actions'
// import ….

class TokenDetails extends Component {
    //…..
}

const mapStateToProps = state => {
    const {balance} = state.tokenDetails;
    return {balance};
};
export default connect(mapStateToProps, {getTokenBalance})(TokenDetails);
```

9. We're almost done. We just need to be able to call our SDK function in our getTokenBalance function to get the balance. Only problem is, our function is asynchronous and Redux wants a response now. Luckily for this problem, **there's Redux Thunk**.
    a. npm install --save redux-thunk
    b. In our App.js file

i.

```
import ReduxThunk from 'redux-thunk';
import { createStore, applyMiddleware } from 'redux';
// … app stuff
// app class {
    const store = createStore(reducers, {}, applyMiddleware(ReduxThunk));
    <Provider store={store}>
        //… your app
    </Provider>
// }
// more stuff
```

    c. Back in our 'actions/index.js' file in our getTokenBalance function, Redux Thunk now allows us to return a function instead of simply dispatching an action immediately.

i.

```
const mySDK = require('../source/ncentSDK.js');
const sdk = new mySDK();

export const getTokenBalance = (user_id, tokentype_id) => {
    return (dispatch) {
        new Promise(function(resolve, reject) {
            sdk.getTokenBalance(user_id, tokentype_id, resolve, reject)
        })
        .then(response => {
            dispatch({type: 'get_balance', payload: response.data.balance})
        })
    }
}
```

    d. Now, in our TokenDetails component, this.props.balance should be updated whenever we call getTokenBalance from our component, and all the state updating is taken care of outside the component.

For other tutorials, check out these resources.

Resources:

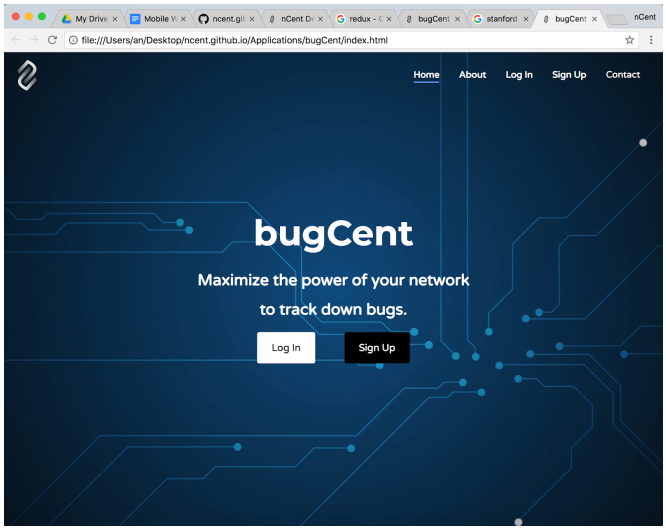
- nCent Video Tutorial (coming soon)
- React Native Redux boilerplate tutorial
- React Native with Redux video tutorial

PRO-TIP: To handle tricky middleware in Redux, use Redux Thunk.

Step 11 - Get Creative!

We chose to integrate with the SDK to make a mobile wallet application that you can use to view and manage your various NCNT tokens. However, we encourage developers using our SDK to get creative and build whatever applications they can imagine on our platform!

Here's an example of our bugCent Application, which we plan to use to find and squash bugs in our source code:



Please feel free to visit our Requests For Startups document to see what our team has brainstormed and get inspired!

If you are really serious about building on our platform, check out yCent to view more details about our up-and-coming incubator program, and submit your application!

PRO-TIP: Check out our founder's Fireside Chat with Steve Jurvetson to learn more about who we are and what our mission is on your journey to build your best application.

Step 12 - Publish Your App for Android and IOS

Before deploying your app, it is a good idea to test it on a device. If you weren't developing with Expo, follow this link to test the app on your device. How you publish your app to the different app stores depends on whether you used Create React Native App or React Native Cli.

- If you used Create React Native App and Expo, publishing is very easy. Follow the docs here
- If you used React Native Cli:
    - Check out these links for the Google Play Store
    - And this link for publishing your app to IOS:

PRO-TIP: It is much faster to publish an App to the Google Play Store (under a day) than to the Apple Store (can take 1 to 2 weeks). Plan accordingly