Quickstart: Smart Account Native Transfer

In this guide, we will walk through creating a basic Node.js script using TypeScript with an implementation of the Smart Account Package from the Biconomy SDK. You will learn how to create a smart account and perform user operations by sending a native transfer of tokens.

info Please note that this tutorial assumes you have Node JS installed on your computer and have some working knowledge of Node.

Environment set up

We will clone a preconfigured Node.js project with TypeScript support to get started. Follow these steps to clone the repository to your local machine using your preferred command line interface:

1. Open your command line interface, Terminal, Command Prompt, or PowerShell.
2. Navigate to the desired directory where you would like to clone the repository.
3. Execute the following command to clone the repository from the provided GitHub link

git clone git@github.com:bcnmy/quickstart.git Note that this is the ssh example, use http or GithubCli options if you prefer.

git clone https://github.com/bcnmy/quickstart.git Once you have the repository on your local machine - start by installing all dependencies using your preferred package manager. In this tutorial we will use yarn.

yarn install yarn dev After running these two commands you should see the printed statement “Hello World!” in your terminal. Any changes made to theindex.ts file in the src directory should now automatically run in your terminal upon save.

All packages you need for this guide are all configured and installed for you, check out thepackage.json file if you want to explore the dependencies.

Click to learn more about the packages * The account package will help you with creating smart contract accounts and an interface with them to create transactions. * The bundler package helps you with interacting with our bundler or alternatively another bundler of your choice. * The core types package will give us Enums for the proper ChainId we may want to use * The paymaster package works similarly to the bundler package in that you can use our paymaster or any other one of your choice. * The core types package will give us Enums for the proper ChainId we may want to use. * The common package is needed by our accounts package as another dependency. * Finally the ethers package at version 5.7.2 will help us with giving our accounts an owner which will be our own EOA. Let’s first set up a .env file in the root of our project, this will need a Private Key of any Externally Owned Account (EOA) you would like to serve as the owner of the smart account we create. This is a private key you can get from wallets like MetaMask, TrustWallet, Coinbase Wallet etc. All of these wallets will have tutorials on how to export the Private key.

PRIVATE_KEY = "" Let’s give our script the ability to access this environment variable. Delete the console log inside ofsrc/index.ts and replace it with the code below. All of our work for the remainder of the tutorial will be in this file.

import

{ config }

from

"dotenv" ;

config ( ) ; Now our code is configured to access the environment variable as needed.

Initialization

Let’s import our bundler package, and providers from the ethers package:

import

{ IBundler , Bundler }

from

"@biconomy/bundler" ; import

{

DEFAULT_ENTRYPOINT_ADDRESS

}

from

"@biconomy/account" ; import

{ providers }

from

"ethers" ; import

{ ChainId }

from

"@biconomy/core-types" ; IBundler is the typing for the Bundler class that we will create a new instance of.

Initial Configuration

const bundler : IBundler =

new

Bundler ({ bundlerUrl : "https://bundler.biconomy.io/api/v2/80001/nJPK7B3ru.dd7f7861-190d-41bd-af80-6877f74b8f44" , chainId : ChainId . POLYGON_MUMBAI , entryPointAddress :

DEFAULT_ENTRYPOINT_ADDRESS , }) ; * Now we create an instance of our bundler with the following:* a bundler url which you can retrieve from the Biconomy Dashboard * * chain ID, in this case we're using Polygon Mumbai * * and default entry point address imported from the account package

import

{ BiconomySmartAccount , BiconomySmartAccountConfig , DEFAULT_ENTRYPOINT_ADDRESS , }

from

"@biconomy/account" ; import

{ Wallet , providers , ethers }

from

"ethers" ; Update your import from the account package to also include BiconomySmartAccount and BiconomySmartAccountConfig and also import Wallet, providers, and ethers from the ethers package.

const provider =

new

providers . JsonRpcProvider ("https://rpc.ankr.com/polygon_mumbai" ,) ; const wallet =

new

Wallet (process . env . PRIVATE_KEY

||

"" , provider) ; * We create a provider using a public RPC provider endpoint from ankr, feel free to use any service here such as Infura or Alchemy if you wish. * Next we create an instance of the wallet associated to our Private key.

We now need an object that will hold the configuration values for our Smart Account.

const biconomySmartAccountConfig : BiconomySmartAccountConfig =

{ signer : wallet , chainId : ChainId . POLYGON_MUMBAI , bundler : bundler , } ; Here we're using theBiconomySmartAccountConfig typing to help us structure the needed values in our configuration.

- signer
- : we pass the instance of our wallet
- chainId

- : we pass Polygon Mumbai or any supported chain you want to test with
- bundler
- : the instance of our bundler

async

function

createAccount ( )

{ let biconomySmartAccount =

new

BiconomySmartAccount (biconomySmartAccountConfig ,) ; biconomySmartAccount =

await biconomySmartAccount . init ( ) ; console . log ("owner: " , biconomySmartAccount . owner) ; console . log ("address: " ,

await biconomySmartAccount . getSmartAccountAddress ( )) ; return biconomySmartAccount ; }

createAccount ( ) ; We create a new instance of the account using the BiconomySmartAccount class and passing it the configuration.

We then await the initialization of the account and log out two values to out terminal: the owner of the account and the smart account address. The owner should be the EOA that you got your private key from and the smart account address will be a new address referring to the address of this wallet.

info Smart accounts are counterfactual in nature. We know their address before they are even deployed. In this instance we won't immediately deploy it, it will automatically be deployed on the first transaction it initiates and the gas needed for deployment will be added to that first transaction. Before continuing, now that we have our smart account address we need to fund it with some test network tokens! Since we are using the Polygon Mumbai network head over to thePolygon Fuacet and paste in your smart account address and get some test tokens!

Once you have tokens available it is time to start constructing our first userOps for a native transfer.

Execute your first userOp

async

function

createTransaction ( )

{ console . log ("creating account") ;

const smartAccount =

await

createAccount ( ) ;

const transaction =

{ to :

"0x322Af0da66D00be980C7aa006377FCaaEee3BDFD" , data :

"0x" , value : ethers . utils . parseEther ("0.1") , } ;

const userOp =

await smartAccount . buildUserOp ([transaction]) ; userOp . paymasterAndData =

"0x" ;

const userOpResponse =

await smartAccount . sendUserOp (userOp) ;

const transactionDetail =

await userOpResponse . wait ( ) ;

console . log ("transaction detail below") ; console . log (transactionDetail) ; } Let’s move the call to create account into the create transaction function and have it assigned to the value smartAccount.

Now we need to construct our transaction which will take the following values:

- to
- : the address this interaction is directed towards, (in this case here is an address I own, feel free to change this to your own or send me more test tokens)
- data
- : we are defaulting to ‘0x’ as we do not need to send any specific data for this transaction to work since it is a native transfer
- value
- : we indicate the amount we would like to transfer here and use theparseEther
- utility function to make sure our value is formatted the way we need it to be

Next up is building the userOp, feel free to add an additional log here if you would like to see how the partial userOp looks in this case. We’re also going to add “0x” for the paymasterAndData value as we just want this to be a regular transaction where the gas is paid for by the end user.

Finally we send the userOp and save the value to a variable named userOpResponse and get the transactionDetail after calling userOpResponse.wait(). This function can optionally take a number to specify the amount of network confirmations you would like before returning a value. For example, if you passeduserOpResponse.wait(5) this would wait for 5 confirmations on the network before giving you the necessary value.

Check out the long transaction details available now in your console! You just created and executed your first userOps using the Biconomy SDK!

Click to view final code import

{ config }

from

"dotenv" import

{ IBundler , Bundler }

from

'@biconomy/bundler' import

{ ChainId }

from

"@biconomy/core-types" ; import

{ BiconomySmartAccount , BiconomySmartAccountConfig ,

DEFAULT_ENTRYPOINT_ADDRESS

}

from

"@biconomy/account" import

{ Wallet , providers , ethers }

from

'ethers'

config ( ) const provider =

new

providers . JsonRpcProvider ("https://rpc.ankr.com/polygon_mumbai") const wallet =

new

Wallet (process . env . PRIVATE_KEY

||

"" , provider) ; const bundler : IBundler =

new

Bundler ({ bundlerUrl : 'https : / / bundler . biconomy . io / api / v2 / 80001 / nJPK7B3ru . dd7f7861 - 190d - 41bd - af80 - 6877f74b8f44 , chainId : ChainId . POLYGON_MUMBAI , entryPointAddress :

DEFAULT_ENTRYPOINT_ADDRESS , }) const biconomySmartAccountConfig : BiconomySmartAccountConfig =

{ signer : wallet , chainId : ChainId . POLYGON_MUMBAI , bundler : bundler } async

function

createAccount ( )

{ const biconomyAccount =

new

BiconomySmartAccount (biconomySmartAccountConfig) const biconomySmartAccount =

await biconomyAccount . init ( ) console . log ("owner: " , biconomySmartAccount . owner) console . log ("address: " ,

await biconomySmartAccount . getAccountAddress ( )) return biconomyAccount } async

function

createTransaction ( )

{ console . log ("creating account") const smartAccount =

await

createAccount ( ) ; const transaction =

{ to :

'0x322Af0da66D00be980C7aa006377FCaaEee3BDFD' , data :

'0x' , value : ethers . utils . parseEther ('0.1') , }

const userOp =

await smartAccount . buildUserOp ([transaction]) userOp . paymasterAndData =

"0x"

const userOpResponse =

await smartAccount . sendUserOp (userOp)

const transactionDetail =

await userOpResponse . wait ( )

console . log ("transaction detail below") console . log (transactionDetail) }

createTransaction ( ) Now that you have completed our quickstart take a look at exploring further usecases in our Quick Explore guide or our Node JS guides! Previous Legacy Smart Account V1 Next Methods