Sponsor UserOperations with Pimlico

In this guide, you will learn how to sponsor the deployment of an ERC-4337 Safe account and its user operations usingPimlico(opens in a new tab) infrastructure and thepermissionless(opens in a new tab) library.

This guide focuses on how user operations are built and what happens under the hood when a Safe is configured and deployed with theSafe4337Module enabled. For a quick start guide, feel free to checkHow to create and use a Safe account with permissionless.js(opens in a new tab) .

Pimlico is one of the most popular ERC-4337 account abstraction infrastructure platforms, which provides a suite of tools and services to help build, deploy, and manage smart accounts on EVM-compatible chains.

permissionless is a TypeScript library focused on building with the ERC-4337 stack, including smart accounts, bundlers, paymasters, and user operations. Some of its core principles are providing a great developer experience and avoiding vendor lock-in by supporting different providers and ERC-4337 smart accounts, including Safe.

Prerequisites

- Node.js and npm(opens in a new tab)
- .
- APimlico account(opens in a new tab)
- and an API key.

Steps

Install dependencies

Installviem(opens in a new tab) andpermissionless(opens in a new tab) dependencies by running the following command:

```
pnpm

install

viem

permissionless
```

Contracts

In this guide, we will use some specific versions for the following contracts deployed on Gnosis Chain.

- v0.6.0
- EntryPoint
- v1.4.1
- Safe Smart Account
- v0.2.0
- Safe4337Module
- v0.2.0
- AddModuleLib

Check the commented links in the code snippet to get the correct addresses if you use a different network.

```
const

ENTRYPOINT_ADDRESS_V06

=

'0x5FF137D4b0FDCD49DcA30c7CF57E578a026d2789'

// https://github.com/safe-global/safe-modules-deployments/blob/main/src/assets/safe-4337-module/v0.2.0/add-modules-lib.json#L8 const

ADD_MODULE_LIB_ADDRESS

=

'0x8EcD4ec46D4D2a6B64fE960B3D64e8B94B2234eb'
```

```
// https://github.com/safe-global/safe-modules-deployments/blob/main/src/assets/safe-4337-module/v0.2.0/safe-4337-module.json#L8 const

SAFE_4337_MODULE_ADDRESS

=

'0xa581c4A4DB7175302464fF3C06380BC3270b4037'

// https://github.com/safe-global/safe-deployments/blob/main/src/assets/v1.4.1/safe_proxy_factory.json#L13 const

SAFE_PROXY_FACTORY_ADDRESS

=

'0x4e1DCf7AD4e460CfD30791CCC4F9c8a4f820ec67'

// https://github.com/safe-global/safe-deployments/blob/main/src/assets/v1.4.1/safe.json#L13 const

SAFE_SINGLETON_ADDRESS

=

'0x41675C099F32341bf84BFc5382aF534df5C7461a'

// https://github.com/safe-global/safe-deployments/blob/main/src/assets/v1.4.1/multi_send.json#L13 const

SAFE_MULTISEND_ADDRESS

=

'0x38869bf66a61cF6bDB996A6aE40D5853Fd43B526'
```

Imports

These are all the imports required in the script we are building for this guide, which includespermissionless andviem packages.

```
import { bundlerActions , getAccountNonce } from

'permissionless' import { pimlicoBundlerActions , pimlicoPaymasterActions } from

'permissionless/actions/pimlico' import { Address , Client , Hash , Hex , PrivateKeyAccount , createClient , createPublicClient , encodeFunctionData , http } from

'viem' import { privateKeyToAccount } from

'viem/accounts' import { gnosis } from

'viem/chains'
```

Create a signer

First, we need a signer instance that will be the owner of the Safe account once it is deployed.

```
const

PRIVATE_KEY

=

'0x...'

const

signer

=

privateKeyToAccount ( PRIVATE_KEY

as
```

Hash)

Initialize the clients

We need to create a few client instances to query the blockchain network and operate with Pimlico infrastructure.

Firstly, we instantiate a standardpublicClient instance for regular Ethereum RPC calls. To do this, we must first define the corresponding RPC URL depending on our network.

const

rpcURL

=

'https://rpc.ankr.com/gnosis'

const

publicClient

=

createPublicClient ({ transport :

http (rpcURL) , chain : gnosis }) Secondly, we instantiate thebundlerClient using the Pimlico APIv1 , which is dedicated to the Bundler methods. This API requires aPIMLICO_API_KEY that we can get from theirdashboard(opens in a new tab) .

const

PIMLICO_API_V1

=

https://api.pimlico.io/v1/gnosis/rpc?apikey= { PIMLICO_API_KEY }

const

bundlerClient

=

createClient ({ transport :

http (PIMLICO_API_V1) , chain : gnosis }) .extend (bundlerActions (ENTRYPOINT_ADDRESS_V06)) .extend (pimlicoBundlerActions (ENTRYPOINT_ADDRESS_V06)) Lastly, we instantiate thepimlicoPaymasterClient using the Pimlico APIv2 , which is dedicated to the Paymaster methods and responsible for interacting with Pimlico's Verifying Paymaster endpoint and requesting sponsorship.

const

PIMLICO_API_V2

=

https://api.pimlico.io/v2/gnosis/rpc?apikey= { PIMLICO_API_KEY }

const

pimlicoPaymasterClient

=

createClient ({ transport :

http (PIMLICO_API_V2) , chain : gnosis }) .extend (pimlicoPaymasterActions (ENTRYPOINT_ADDRESS_V06))

Create a UserOperation

We now define the user operation object we want to execute following the structure of theUserOperation type.

type

```
UserOperation

= { sender :

Address nonce :

bigint initCode :

Hex callData :

Hex callGasLimit :

bigint verificationGasLimit :

bigint preVerificationGas :

bigint maxFeePerGas :

bigint maxPriorityFeePerGas :

bigint paymasterAndData :

Hex signature :
```

Hex } We are currently missing the values for thesender ,nonce ,initCode , andcallData properties, so we need to calculate them. The gas-related properties will be calculated later in the next step, and thesignature in the following one.

After getting these properties, we can instantiate thesponsoredUserOperation object.

```
const

contractCode

=

await

publicClient .getBytecode ({ address : sender })

const

sponsoredUserOperation :

UserOperation

= { sender , nonce , initCode : contractCode ?

'0x'

: initCode , callData , callGasLimit :

1 n ,

// All gas values will be filled by Estimation Response Data. verificationGasLimit :

1 n , preVerificationGas :

1 n , maxFeePerGas :

1 n , maxPriorityFeePerGas :

1 n , paymasterAndData :

ERC20_PAYMASTER_ADDRESS , signature :

'0x' }
```

Get theinitCode

TheinitCode encodes the instructions for deploying the ERC-4337 smart account. For this reason, it's not needed when the account has already been deployed.

If we are deploying a new account, we can calculate it with thegetAccountInitCode utility function defined in the second tab, which returns the concatenation of theSafeProxyFactory contract address and theinitCodeCallData .

TheinitCodeCallData encodes the call to thecreateProxyWithNonce function in theSafeProxyFactory contract with theinitializer and asaltNonce .

Theinitializer is calculated using thegetInitializerCode function from its corresponding tab. This function returns the encoding of the call to thesetup function in the Safe contract to initialize the account with itsowners ,threshold ,fallbackHandler , etc.

In this case, we are creating a Safe account with one owner (our signer), threshold one, and theSafe4337Module as thefallbackHandler .

This initialization also includes the option to execute a call by using theto anddata parameters, which we will use to enable theSafe4337Module contract in the Safe and give an allowance to theEntryPoint contract to pay the gas fees in an ERC-20 token like USDC. As we are performing multiple calls, we need to encode a call to theMultiSend contract using theencodeMultiSend function, setting theSAFE_MULTISEND_ADDRESS as theto and its encoding as thedata .

To enable the module in theenableModuleCallData function, we will encode a call to theAddModuleLib contract by passing the address of theSafe4337Module .

script.ts getAccountInitCode.ts getInitializerCode.ts enableModuleCallData.ts encodeMultiSend.ts const

initCode

=

await

getAccountInitCode ({ owner :

signer .address , addModuleLibAddress :

ADD_MODULE_LIB_ADDRESS , safe4337ModuleAddress :

SAFE_4337_MODULE_ADDRESS , safeProxyFactoryAddress :

SAFE_PROXY_FACTORY_ADDRESS , safeSingletonAddress :

SAFE_SINGLETON_ADDRESS , saltNonce , multiSendAddress :

SAFE_MULTISEND_ADDRESS , erc20TokenAddress :

USDC_TOKEN_ADDRESS , paymasterAddress :

ERC20_PAYMASTER_ADDRESS }) In case of doing the token approval to theEntryPoint contract, check the list of ERC-20 Pimlico paymasters and USDC tokens addresses(opens in a new tab) to select the correct addresses for these contracts depending on the network.

Get the Safe address

We implemented thegetAccountAddress utility function to calculate the' sender'. This function calls the viemgetContractAddress function to get the address based on:

- TheSAFE_PROXY_FACTORY_ADDRESS
- The bytecode of the deployed contract (the Safe Proxy)
- ThesaltNonce

Notice that thesender address will depend on the value of the Safe configuration properties and thesaltNonce .

script.ts getAccountAddress.ts const

sender

=

await

getAccountAddress ({ client : publicClient , owner :

signer .address , addModuleLibAddress :

ADD_MODULE_LIB_ADDRESS , safe4337ModuleAddress :

SAFE_4337_MODULE_ADDRESS , safeProxyFactoryAddress :

SAFE_PROXY_FACTORY_ADDRESS , safeSingletonAddress :

SAFE_SINGLETON_ADDRESS , saltNonce , multiSendAddress :

SAFE_MULTISEND_ADDRESS , erc20TokenAddress :

USDC_TOKEN_ADDRESS , paymasterAddress :

ERC20_PAYMASTER_ADDRESS }) After calculating the predicted address of the counterfactual ERC-4337 Safe account, thesender , we can check on theGnosis Chain block explorer(opens in a new tab) that the account is not deployed yet.

Get thenonce

To get the nonce, we can use thegetAccountNonce function.

const

nonce

=

await

getAccountNonce (publicClient as

Client , { entryPoint :

ENTRYPOINT_ADDRESS_V06 , sender })

Get thecallData

ThecallData encodes a call to theexecuteUserOp function and represents the action(s) that will be executed from the Safe account. In this example we are sending a transaction to the Safe account with no value and no data, resulting in an increase of the nonce of the account. However, this can be any action like a transfer of the native or an ERC-20 token, a call to another contract, etc.

Check theencodeCallData tab to see how the encoding is implemented.

script.ts encodeCallData.ts const

callData :

0x { string }

=

encodeCallData ({ to : sender , data :

'0x' , value :

0 n })

Estimate the UserOperation gas

To estimate the gas limits for aUserOperation , we call theestimateUserOperationGas method from the bundler API, which receives theuserOperation andentryPoint as parameters.

After that, we call thegetUserOperationGasPrice method to get the maximum gas price and add all the returned values to thesponsoredUserOperation .

const

gasEstimate

=

await

bundlerClient .estimateUserOperationGas ({ userOperation : sponsoredUserOperation , entryPoint :

ENTRYPOINT_ADDRESS_V06 }) const

maxGasPriceResult

=

await

bundlerClient .getUserOperationGasPrice ()

sponsoredUserOperation .callGasLimit =

gasEstimate .callGasLimit sponsoredUserOperation .verificationGasLimit =

gasEstimate .verificationGasLimit sponsoredUserOperation .preVerificationGas =

gasEstimate .preVerificationGas sponsoredUserOperation .maxFeePerGas =

maxGasPriceResult . fast .maxFeePerGas sponsoredUserOperation .maxPriorityFeePerGas =

maxGasPriceResult . fast .maxPriorityFeePerGas To use the Paymaster to pay for the fees, we need to provide aSPONSORSHIP_POLICY_ID that can be provided by a third party willing to sponsor our user operations, or it can be generated in thePimlico dashboard(opens in a new tab) . Sponsorship policies allow the definition of custom rules for sponsorships with various options to limit the total sponsored amount, per user, and per user operation.

On top of that, we need to overwrite some gas values from the Paymaster and add thepaymasterAndData to thesponsoredUserOperation .

if (usePaymaster) { const

sponsorResult

=

await

pimlicoPaymasterClient .sponsorUserOperation ({ userOperation : sponsoredUserOperation , entryPoint :

ENTRYPOINT_ADDRESS_V06 , sponsorshipPolicyId :

SPONSORSHIP_POLICY_ID })

sponsoredUserOperation .callGasLimit =

sponsorResult .callGasLimit sponsoredUserOperation .verificationGasLimit =

sponsorResult .verificationGasLimit sponsoredUserOperation .preVerificationGas =

sponsorResult .preVerificationGas sponsoredUserOperation .paymasterAndData =

sponsorResult .paymasterAndData } If we don't want to use a Paymaster to pay the gas fees, we need to ensure the Safe account holds at least a few USDC tokens because the fees would be extracted from the Safe itself. Be cautious with the amount as it will depend on thecallData , and the networkgasPrice .

Sign the UserOperation

To sign thesponsoredUserOperation , we have created thesignUserOperation utility function that returns the signature from the signer and accepts the following parameters. Check the second tab to see its implementation.

script.ts signUserOperation.ts const

chainId

=

100

sponsoredUserOperation .signature =

await

signUserOperation (sponsoredUserOperation , signer , chainId , SAFE_4337_MODULE_ADDRESS)

Submit the UserOperation

Call thesendUserOperation method from the bundler to submit thesponsoredUserOperation to theEntryPoint contract.

const

userOperationHash

=

await

bundlerClient .sendUserOperation ({ userOperation : sponsoredUserOperation , entryPoint :

ENTRYPOINT_ADDRESS_V06 }) To get more details about the submittedUserOperation copy the value of theuserOperationHash returned, visit theUserOp Explorer(opens in a new tab), and paste it into the search bar.

Lastly, to get more details about the transaction, we can get the receipt of thesponsoredUserOperation , get thetransactionHash , and check the tansaction details in theGnosis Chain block explorer(opens in a new tab).

const

receipt

=

await

bundlerClient .waitForUserOperationReceipt ({ hash : userOperationHash })

const

transactionHash

=

receipt . receipt .transactionHash

Recap and further reading

This guide covered how to sponsor the deployment of a new ERC-4337 Safe and its user operations with Pimlico infrastructure using a Paymaster.

Feel free to try out other ideas and possibilities, as there are many more regarding:

- The deployment and initial setup of ERC-4337 accounts.
- The entity responsible for paying the transaction fees.
- The tokens used to pay the transaction fees.

Explore our4337-gas-metering(opens in a new tab) repository on GitHub to see how most of these options work with Safe and notice the integrations with different providers like Alchemy, Gelato, and Pimlico (where you will find most of the code used in this guide).

Supported Networks

Was this page helpful?

Report issue