D4D4D4;--ch-t-background: #1E1E1E;--ch-t-lighter-inlineBackground: #1e1e1ee6;--ch-t-editor-background: #1E1E1E;--ch-t-editor-foreground: #D4D4D4;--ch-t-editor-rangeHighlightBackground: #ffffff0b;--ch-t-editor-infoForeground: #3794FF;--ch-t-editor-selectionBackground: #264F78;--ch-t-focusBorder: #007FD4;--ch-t-tab-activeBackground: #1E1E1E;--ch-t-tab-activeForeground: #ffffff;--ch-t-tab-inactiveBackground: #2D2D2D;--ch-t-tab-inactiveForeground: #ffffff80;--ch-t-tab-border: #252526;--ch-t-tab-activeBorder: #1E1E1E;--ch-t-editorGroup-border: #444444;--ch-t-editorGroupHeader-tabsBackground: #252526;--ch-t-editorLineNumber-foreground: #858585;--ch-t-input-background: #3C3C3C;--ch-t-input-foreground: #D4D4D4;--ch-t-icon-foreground: #C5C5C5;--ch-t-sideBar-background: #252526;--ch-t-sideBar-foreground: #D4D4D4;--ch-t-sideBar-border: #252526;--ch-t-list-activeSelectionBackground: #094771;--ch-t-list-activeSelectionForeground: #fffffe;--ch-t-list-hoverBackground: #2A2D2E; }

Permissionless.js Detailed Guide

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 quickstart guide, check thePermissionless quickstart guide .

â¹ï¸ If you are already building with the Safe{Core} SDK, you may want to follow theSafe4337Pack guide instead of integrating thepermissionless library directly into your application.

Prerequisites

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

Install dependencies

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

_10 pnpm install viem permissionless

Steps

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.

_16 const ENTRYPOINT_ADDRESS_V06 = '0x5FF137D4b0FDCD49DcA30c7CF57E578a026d2789' _16 _16 // https://github.com/safe-global/safe-modules-deployments/blob/main/src/assets/safe-4337-module/v0.2.0/add-modules-lib.json#L8 _16 const ADD_MODULE_LIB_ADDRESS = '0x8EcD4ec46D4D2a6B64fE960B3D64e8B94B2234eb' _16 _16 // https://github.com/safe-global/safe-modules-deployments/blob/main/src/assets/safe-4337-module/v0.2.0/safe-4337-module.json#L8 _16 const SAFE_4337_MODULE_ADDRESS = '0xa581c4A4DB7175302464fF3C06380BC3270b4037' _16 _16 // https://github.com/safe-global/safe-deployments/blob/main/src/assets/v1.4.1/safe_proxy_factory.json#L13 _16 const SAFE_PROXY_FACTORY_ADDRESS = '0x4e1DCf7AD4e460CfD30791CCC4F9c8a4f820ec67' _16 _16 // https://github.com/safe-global/safe-deployments/blob/main/src/assets/v1.4.1/safe.json#L13 _16 const SAFE_SINGLETON_ADDRESS = '0x41675C099F32341bf84BFc5382aF534df5C7461a' _16 _16 // https://github.com/safe-global/safe-deployments/blob/main/src/assets/v1.4.1/multi_send.json#L13 _16 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.

_18 import { bundlerActions, getAccountNonce } from 'permissionless' _18 import { _18 pimlicoBundlerActions, _18 pimlicoPaymasterActions _18 } from 'permissionless/actions/pimlico' _18 import { _18 Address, _18 Client, _18 Hash, _18 Hex, _18 PrivateKeyAccount, _18 createClient, _18 createPublicClient, _18 encodeFunctionData, _18 http _18 } from 'viem' _18 import { privateKeyToAccount } from 'viem/accounts' _18 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.

_10 const PRIVATE_KEY = '0x...' _10 _10 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.

_10 const rpcURL = 'https://rpc.ankr.com/gnosis' _10 _10 const publicClient = createPublicClient({ _10 transport: http(rpcURL), _10 chain: gnosis _10 }) 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 their[dashboard(opens in a new tab)](#) .

_10 const PIMLICO_API_V1 = https://api.pimlico.io/v1/gnosis/rpc?apikey={PIMLICO_API_KEY} _10 _10 const bundlerClient = createClient({ _10 transport: http(PIMLICO_API_V1), _10 chain: gnosis _10 }) _10 .extend(bundlerActions(ENTRYPOINT_ADDRESS_V06)) _10 .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.

_10 const PIMLICO_API_V2 = https://api.pimlico.io/v2/gnosis/rpc?apikey={PIMLICO_API_KEY} _10 _10 const pimlicoPaymasterClient = createClient({ _10 transport: http(PIMLICO_API_V2), _10 chain: gnosis _10 }).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.

_13 type UserOperation = { _13 sender: Address _13 nonce: bigint _13 initCode: Hex _13 callData: Hex _13 callGasLimit: bigint _13 verificationGasLimit: bigint _13 preVerificationGas: bigint _13 maxFeePerGas: bigint _13 maxPriorityFeePerGas: bigint _13 paymasterAndData: Hex _13 signature: Hex _13 } 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.

_15 const contractCode = await publicClient.getBytecode({ address: sender }) _15 _15 const sponsoredUserOperation: UserOperation = { _15 sender, _15 nonce, _15 initCode: contractCode ? '0x' : initCode, _15 callData, _15 callGasLimit: 1n, // All gas values will be filled by Estimation Response Data. _15 verificationGasLimit: 1n, _15 preVerificationGas: 1n, _15 maxFeePerGas: 1n, _15 maxPriorityFeePerGas: 1n, _15 paymasterAndData: ERC20_PAYMASTER_ADDRESS, _15 signature: '0x' _15 }

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 _11 const initCode = await getAccountInitCode({ _11 owner: signer.address, _11 addModuleLibAddress: ADD_MODULE_LIB_ADDRESS, _11 safe4337ModuleAddress: SAFE_4337_MODULE_ADDRESS, _11 safeProxyFactoryAddress: SAFE_PROXY_FACTORY_ADDRESS, _11 safeSingletonAddress: SAFE_SINGLETON_ADDRESS, _11 saltNonce, _11 multiSendAddress: SAFE_MULTISEND_ADDRESS, _11 erc20TokenAddress: USDC_TOKEN_ADDRESS, _11 paymasterAddress: ERC20_PAYMASTER_ADDRESS _11 }) In case of doing the token approval to theEntryPoint contract, check the list ofERC-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 _12 const sender = await getAccountAddress({ _12 client: publicClient, _12 owner: signer.address, _12 addModuleLibAddress: ADD_MODULE_LIB_ADDRESS, _12 safe4337ModuleAddress: SAFE_4337_MODULE_ADDRESS, _12 safeProxyFactoryAddress: SAFE_PROXY_FACTORY_ADDRESS, _12 safeSingletonAddress: SAFE_SINGLETON_ADDRESS, _12 saltNonce, _12 multiSendAddress: SAFE_MULTISEND_ADDRESS, _12 erc20TokenAddress: USDC_TOKEN_ADDRESS, _12 paymasterAddress: ERC20_PAYMASTER_ADDRESS _12 }) 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.

_10 const nonce = await getAccountNonce(publicClient as Client, { _10 entryPoint: ENTRYPOINT_ADDRESS_V06, _10 sender _10 })

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 _10 const callData: 0x{string} = encodeCallData({ _10 to: sender, _10 data: '0x', _10 value: 0n _10 })

Estimate the UserOperation gas

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

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

_11 const gasEstimate = await bundlerClient.estimateUserOperationGas({ _11 userOperation: sponsoredUserOperation, _11 entryPoint: ENTRYPOINT_ADDRESS_V06 _11 }) _11 const maxGasPriceResult = await bundlerClient.getUserOperationGasPrice() _11 _11 sponsoredUserOperation.callGasLimit = gasEstimate.callGasLimit _11 sponsoredUserOperation.verificationGasLimit = gasEstimate.verificationGasLimit _11 sponsoredUserOperation.preVerificationGas = gasEstimate.preVerificationGas _11 sponsoredUserOperation.maxFeePerGas = maxGasPriceResult.fast.maxFeePerGas _11 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 .

_12 if (usePaymaster) { _12 const sponsorResult = await pimlicoPaymasterClient.sponsorUserOperation({ _12 userOperation: sponsoredUserOperation, _12 entryPoint: ENTRYPOINT_ADDRESS_V06, _12 sponsorshipPolicyId: SPONSORSHIP_POLICY_ID _12 }) _12 _12 sponsoredUserOperation.callGasLimit = sponsorResult.callGasLimit _12 sponsoredUserOperation.verificationGasLimit = sponsorResult.verificationGasLimit _12 sponsoredUserOperation.preVerificationGas = sponsorResult.preVerificationGas _12 sponsoredUserOperation.paymasterAndData = sponsorResult.paymasterAndData _12 } 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 _10 const chainId = 100 _10 _10 sponsoredUserOperation.signature = await signUserOperation(_10 sponsoredUserOperation, _10 signer, _10 chainId, _10 SAFE_4337_MODULE_ADDRESS _10)

Submit the UserOperation

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

_10 const userOperationHash = await bundlerClient.sendUserOperation({ _10 userOperation: sponsoredUserOperation, _10 entryPoint: ENTRYPOINT_ADDRESS_V06 _10 }) To get more details about the submittedUserOperation object 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 transaction details in theGnosis Chain block explorer(opens in a new tab).

_10 const receipt = await bundlerClient.waitForUserOperationReceipt({ _10 hash: userOperationHash _10 }) _10 _10 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).

Permissionless.js Quickstart