Methods

The SDK provides the following API methods for a smart account.

createSmartAccountClient

This method is used to create an instance of the Biconomy Smart Account. This method requires a smart account configuration object to be passed and returns the smart account API instance.

This method creates the smart account based on the config object. To configure a smart account on another chain, you need to instantiate another smart account API instance with configuration of that chain.

Usage

let provider =

new

ethers . JsonRpcProvider ("rpcUrl") ; let signer =

new

ethers . Wallet ("private key" , provider) ;

const smartAccount =

await

createSmartAccountClient ({ signer , bundlerUrl :

"" ,

// <-- Read about this at https://docs.biconomy.io/dashboard#bundler-url biconomyPaymasterApiKey :

"" ,

// <-- Read about at https://docs.biconomy.io/dashboard/paymaster rpcUrl :

""

// Recommended for signers that are not a viem wallet or an ethers signer. It's advised to pass RPC url in case of custom signers such as privy, dynamic etc. If rpcUrl is not provided then a default public rpc will be used - which will likely be heavily throttled and can often silently fail }) ; Parameters

required params are explicitly mentioned

- config (object
- , required): ABiconomySmartAccountConfig
- object containing configuration options for creating the smart account.* signer(Signer
- , required) OR defaultValidationModule (BaseValidationModule
- , required): One either needs to pass the signer instance or the default validation module which gets used to detect address of the smart account. If not passed explictly, ECDSA module gets used as default.
- bundlerUrl (string
- , required) OR bundler (IBundler
- , required) : bundler url which will be internally used to create bundler instance or the bundler instance. Bundler instance can also be used if one wants to customise the bundler. Refer to bundlerintegration
- for more details on bundler.
- chainId (ChainId
- enum): The identifier for the blockchain network. (e.g., ChainId.POLYGON_MUMBAI).
- biconomyPaymasterApiKey(string
-

-) OR paymaster (IPaymaster
-): one can either pass paymaster API key or custom paymaster instance to use the paymaster.
- entryPointAddress (string
-): DEFAULT_ENTRY_POINT_ADDRESS will be used if not passed, otherwise the passed address will be used. On specific chains like Chiliz Mainnet it is a different address, so will need to be passed explicitly. Refer to below notes on this.
- activeValidationModule (BaseValidationModule
-): The run-time validation module (must be one of enabled validation modules) to sign and validate next userOp.
- rpcUrl (string
-): Recommended for signers that are not a viem wallet or an ethers signer. It's advised to pass RPC url in case of custom signers such as privy, dynamic etc. If rpcUrl is not provided then a default public rpc will be used - which will likely be heavily throttled and can often silently fail.
- index (number
-): index to create multiple smart accounts for an EOA. First account gets created with 0 index.
- accountAddress(string
-): smart account address. If you already have the smart account address, you can pass using this to get the instance.

Returns

- smartAccount
- (BiconomySmartAccount
-): An instance of the Biconomy Smart Account.

info Building on Chiliz Mainnet or the Spicy Testnet? Note that the entry point address on this is different as it was deployed by us on the Biconomy team. The address of the entry point is :0x00000061FEfce24A79343c27127435286BB7A4E1

createBundler

This method creates a custom bundler instance.

Usage

const address =

await

createBundler ({ bundlerUrl :

"BUNDLER_URL" }) ; Params

- config(BundlerConfig
-): configs to be passed to bundler

export type Bundlerconfig

=

{ bundlerUrl : string ; entryPointAddress ? : string ; chainId ? : number ; userOpReceiptIntervals ? :

{ [key in number] ? : number ;

// The polling interval per chain for the tx receipt in milliseconds. Default value is 5 seconds. } ;
userOpWaitForTxHashIntervals ? :

{ [key in number] ? : number ;

// The polling interval per chain for the tx result in milliseconds. Default value is 0.5 seconds. } ;

userOpReceiptMaxDurationIntervals ? :

{ [key in number] ? : number ;

// The maximum duration in milliseconds per chain to wait for the tx receipt. Default value is 30 seconds. } ;
userOpWaitForTxHashMaxDurationIntervals ? :

{ [key in number] ? : number ;

// The maximum duration in milliseconds per chain to wait for the tx hash. Default value is 20 seconds. } ; } ; Returns

- bundler(Promise
-): bundler instance

Smart Account Get Methods

getAccountAddress( )

This method retrieves the counterfactual address of thesmartAccount instance.

Usage

const address =

await smartAccount . getAccountAddress ( ) ; Returns

- address (Promise
-): A Promise resolving to the account address associated with the smartAccount instance.

getNonce( )

This method is used to retrieve the nonce associated with thesmartAccount instance.

Usage

const nonce =

await smartAccount . getNonce ( ) ; console . log (nonce . toNumber ( )) ;

const nonceKey =

10 ; const nonce =

await biconomySmartAccount . getNonce (nonceKey) ; Parameters

- nonceKey(number
-): one can also pass the optional parameter nonceKey in the case of two-dimensional nonces.

Returns

- address (Promise
-): A Promise resolving to current nonce of the smart account.

index

This is used to retrieve the index of the current active smart account.

Usage

const index = smartAccount . index ; Returns

- index (number
-): A number indicating the index of current active smart account.

deploy()

getBalances()

[getSupportedTokens()]

Transaction Methods

sendTransaction( )

This method is used to Send a transaction to a bundler for execution. It internally executes a build and send UserOp.

Usage

```
import
{ createClient }
from
"viem" ; import
{ createSmartAccountClient ,
PaymasterMode
}
from
"@biconomy/account" ; import
{ createWalletClient , http }
from
"viem" ; import
{ polygonMumbai }
from
"viem/chains" ;
const signer =
createWalletClient ( { account , chain : polygonMumbai , transport :
http ( ) , } ) ;
const smartWallet =
await
createSmartAccountClient ( { signer , bundlerUrl } ) ;
// Retrieve bundler url from dasboard const encodedCall =
encodeFunctionData ( { abi :
parseAbi ( [ "function safeMint(address to) public" ] ) , functionName :
"safeMint" , args :
[ "0x..." ] , } ) ;
const transaction =
{ to : nftAddress , data : encodedCall , } ;
const
{ waitForTxHash }
=
```

await smartWallet . sendTransaction (transaction) ; const

{ transactionHash , userOperationReceipt }

=

await

waitForTxHash ( ) ; Parameters

- manyOrOneTransactions (Transaction | Transaction[]
- , required): An array of transactions to be batched which will be executed in the provided order. You can also pass a single transaction.
- Transaction
- :
- {
- to
- :
- string
- ;
- }
- &
- ValueOrData
- buildUseropDto (BuildUserOpOptions
- , optional): These options can be passed to customize how a userOp is built.
- type
- BuildUserOpOptions
- =
- {
- forceEncodeForBatch
- ?
- :
- boolean
- ;
- nonceOptions
- ?
- :
- NonceOptions
- ;
- params
- ?
- :
- ModuleInfo
- ;
- paymasterServiceData
- ?
- :
- PaymasterUserOperationDto
- ;
- simulationType
- ?
- :
- SimulationType
- ;
- stateOverrideSet
- ?
- :
- StateOverrideSet
- ;
- useEmptyDeployCallData
- ?
- :
- boolean
- ;
- }
- ;
- Let's look at each of these params:
-
    1. forceEncodeForBatch (boolean

- 1.): When a transactions array is passed, by default the Biconomy SDK encodes it for executeBatch() executor function and execute() function for single transaction. However, in some cases, there may be a preference to encode a single transaction for a batch, especially if the custom module only decodes for executeBatch. In such cases, set this flag to true; otherwise, it remains false by default.
- 1. nonceOptions(NonceOptions
- 1.) : This can be used to execute multiple user operations in parallel for the same smart account.
- 1. type
- 1. NonceOptions
- 1. =
- 1. {
- 1. nonceKey
- 1. ?
- 1. :
- 1. number
- 1. ;
- 1. nonceOverride
- 1. ?
- 1. :
- 1. number
- 1. ;
- 1. }
- 1. ;
- 1. // nonceOptions usage
- 1. let
- 1. i
- 1. =
- 1. 0
- 1. ;
- 1. const
- 1. userOp
- 1. =
- 1. await
- 1. smartAccount
- 1. .
- 1. buildUserOp
-

1. (
- 1. [
- 1. tx1
- 1.]
- 1. ,
- 1. {
- 1. nonceOptions
- 1. :
- 1. {
- 1. nonceKey
- 1. :
- 1. i
- 1. ++
- 1. }
- 1. ,
- 1. }
- 1.)
- 1. ;
- 1. nonceKey can be initialised at any arbitrary number and incremented as one builds user operations to be sent in parallel. The nonceKey will create a batch or space in which the nonce can safely increment without colliding with other transactions. The nonceOverride will directly override the nonce and should only be used if you know the order in which you are sending the userOps.
- 1. params (ModuleInfo
- 1.): This param can be used to pass session validation module parameters. Refer to the tutorial to learn more about the session keys.
- 1. type
- 1. ModuleInfo
- 1. =
- 1. {
- 1. sessionID
- 1. ?
- 1. :
- 1. string
- 1. ;
- 1. sessionSigner
- 1. ?
-

1. :
- 1. Signer
- 1. ;
- 1. sessionValidationModule
- 1. ?
- 1. :
- 1. string
- 1. ;
- 1. additionalSessionData
- 1. ?
- 1. :
- 1. string
- 1. ;
- 1. batchSessionParams
- 1. ?
- 1. :
- 1. SessionParams
- 1. [
- 1.]
- 1. ;
- 1. }
- 1. ;
- 1. paymasterServiceData (PaymasterUserOperationDto
- 1.): ThepaymasterServiceData
- 1. includes details about the kind of sponsorship and payment token in case mode is ERC20. It contains information about the paymaster service, which is used to calculate thepaymasterAndData
- 1. field in the user operation. Note that this is only applicable if you're using Biconomy paymaster.
- 1. type
- 1. PaymasterUserOperationDto
- 1. =
- 1. {
- 1. mode
- 1. :
- 1. PaymasterMode
- 1. ;

- 1. calculateGasLimits
- 1. ?
- 1. :
- 1. boolean
- 1. ;
- 1. //this flag defaults to true, signifying the paymaster will undertake gas limit calculations to ensure efficient operation execution
- 1. expiryDuration
- 1. ?
- 1. :
- 1. number
- 1. ;
- 1. webhookData
- 1. ?
- 1. :
- 1. Record
- 1. <
- 1. string
- 1. ,
- 1. any
- 1.
- 1. ;
- 1. smartAccountInfo
- 1. ?
- 1. :
- 1. SmartAccountData
- 1. ;
- 1. feeTokenAddress
- 1. ?
- 1. :
- 1. string
- 1. ;
- 1. }
- 1. ;
-

1. simulationType(SimulationType
- 1. , enum): When you send a transaction, bundler performs simulations to check for any reverts. simulationType determines how and when the UserOperation is simulated. This parameter can be set in two ways:
- 1.
    - validation
- 1.
    - which will only simulate the validation phase, checks if user op is valid but does not check if execution will succeed. By default this flag is set to validation.
- 1.
    - validation_and_execution
- 1.
    - checks if user op is valid and if user op execution will succeed based on the callData. It can be useful during development for error logging purposes in case of any issues, but it may lead to increased latency.
- 1. stateOverrideSet(StateOverrideSet
- 1.): for overriding the blockchain state during simulations or gas estimation.

Returns

- userOpResponse (Promise
-): The method returns an object of typeUserOpResponse
- which has auserOpHash
- and two methods:wait()
- andwaitForTxHash()
- .
- type
- UserOpResponse
- =
- {
- userOpHash
- :
- string
- ;
- wait
- (
- _confirmations
- ?
- :
- number
-)
- :
- Promise
- <
- UserOpReceipt
-
- ;
- waitForTxHash
- (
-)
- :
- Promise
- <
- UserOpStatus
-
- ;
- }
- ;
- type
- UserOpReceipt
- =
- {

- / *The request hash of the UserOperation.*/
- userOpHash
- :
- string
- ;
- / *The entry point address used for the UserOperation.*/
- entryPoint
- :
- string
- ;
- / *The paymaster used for this UserOperation (or empty).*/
- paymaster
- :
- string
- ;
- / *The actual amount paid (by account or paymaster) for this UserOperation.*/
- actualGasCost
- :
- Hex
- ;
- / *The total gas used by this UserOperation (including preVerification, creation, validation, and execution).*/
- actualGasUsed
- :
- Hex
- ;
- / *Indicates whether the execution completed without reverting.*/
- success
- :
- "true"
- |
- "false"
- ;
- / *In case of revert, this is the revert reason.*/
- reason
- :
- string
- ;
- / *The logs generated by this UserOperation (not including logs of other UserOperations in the same bundle).*/
- logs
- :
- Array
- <
- any
-

- ;
- // The logs generated by this UserOperation (not including logs of other UserOperations in the same bundle)
- / *The TransactionReceipt object for the entire bundle, not only for this UserOperation.*/
- receipt
- :
- any
- ;
- }
- ;
- Thewait()
- method resolves when the user operation is dispatched by the bundler on-chain and gets mined. ThewaitForTxHash()
- method returns aUserOpStatus
- object which includes the transaction hash and the receipt once added on-chain.
- const
- {
- transactionHash
- }
- =
- await
- userOpResponse
- .
- waitForTxHash

- (
-)
- ;
- console
- .
- log
- (
- "transaction Hash"
- ,
- transactionHash
-)
- ;
- const
- userOpReceipt
- =
- await
- userOpResponse
- .
- wait
- (
-)
- ;
- if
- (
- userOpReceipt
- .
- success
- ==
- 'true'
-)
- {
- // indicates that the user operation was successful without any revert
- console
- .
- log
- (
- "UserOp receipt"
- ,
- userOpReceipt
-)
- console
- .
- log
- (
- "actual transaction receipt"
- ,
- userOpReceipt
- .
- receipt
-)
- }

buildUserOp( )

This method is used for configuring and setting up properties of the partialuserOp object. It converts an individual transaction or batch of transactions into a partial user operation populating fields such as initCode, sender, nonce, maxFeePerGas, maxPriorityFeePerGas, callGasLimit, verificationGasLimit and preVerificationGas (as this step also involves estimating gas for the userOp internally)

Usage

For example, in the context of creating a userOp for anaddComment transaction, an instance of the contract is created. A basic transaction object is then created that holds the necessary address and data from the transaction. Finally, a partial userOp is created using the Smart Account'sbuildUserOp method. Now this can be signed and sent to the bundler.

const contractAddress =

"contract address" ; const provider =

new

ethers . JsonRpcProvider ("rpc url") ;

const blogContract =

new

ethers . Contract (contractAddress , abi ,

// contract abi provider) ;

const createComment =

await blogContract . populateTransaction . addComment ("comment") ;

const tx1 =

{ to : contractAddress , data : createComment . data , } ;

const userOp =

await smartAccount . buildUserOp ([tx1]) ; Parameters

- transactions (Transaction[]
- , required): The required argument is an array of transactions which will be executed in provided order. You can pass multiple transactions into a userOp if you would like to batch them together into one transaction.
- buildUseropDto (BuildUserOpOptions
-): One can also pass these options to customize how a userOp is built.

Returns

- partialUserOp (Promise>
-): A Promise resolving toUserOperationStruct
- which can be further signed and sent to the bundler.

senduserOp( )

This method is used to submit a User Operation object to the User Operation pool of the client. It signs the UserOperation using activeValidationModule instance and submits it to the bundler for on-chain processing.

Usage

const userOp =

await smartAccount . buildUserOp ([transaction]) ; const userOpResponse =

await smartAccount . sendUserOp (userOp) ;

const

{ receipt }

=

await userOpResponse . wait (1) ; Parameters

- userOp (Partial
- , required): TheuserOp
- object includes essential fields likesender
- ,nonce
- ,callData
- ,callGasLimit
- andgas
- related properties.
- params (SendUserOpParams
- , optional): This gets used when the active validation module is complex and requires additional information for signature generation. TheSendUserOpParams
- object can contain fields such assessionID
- ,sessionSigner
- ,sessionValidationModule
- ,additionalSessionData

- ,batchSessionParams
- , andsimulationType
- . These parameters are used to customize the behavior of thesendUserOp
- method and are optional.
- const
- userOpResponse
- =
- await
- moduleSmartAccount
- ?.
- sendUserOp
- (
- userOp
- ,
- {
- sessionSigner
- :
- sessionSigner
- ,
- sessionValidationModule
- :
- moduleAddr
- ,
- }
-)
- ;
- Similar to building auserOp
- we need to ensure that any modules used for additional validation or execution logic are specified in thesendUserOp
- method. Currently, this only applies for session key module requirements.

note Please note thatsimulationType allows for more debugging insights aboutcallData on why an internal transaction fails. It is set to "validation" by default, but can be changed to "validation_and_execution" for more detailed tracing. Returns

- userOpsResponse (UserOpResponse
-): The method returns an object of typeUserOpResponse
- which has auserOpHash
- and two methods:wait()
- andwaitForTxHash()
- .
- type
- UserOpResponse
- =
- {
- userOpHash
- :
- string
- ;
- wait
- (
- _confirmations
- ?
- :
- number
-)
- :
- Promise
- <
- UserOpReceipt
-

- ;
- waitForTxHash
- (
-)
- :
- Promise
- <
- UserOpStatus
-

- ;
- }
- ;
- type
- UserOpReceipt
- =
- {
- userOpHash
- :
- string
- ;
- entryPoint
- :
- string
- ;
- sender
- :
- string
- ;
- nonce
- :
- number
- ;
- paymaster
- :
- string
- ;
- actualGasCost
- :
- BigNumber
- ;
- actualGasUsed
- :
- BigNumber
- ;
- success
- :
- boolean
- ;
- reason
- :
- string
- ;
- logs
- :
- Array
- <
- ethers
- .
- providers
- .
- Log
-

- ;
- receipt
- :
- ethers
- .
- providers
- .
- TransactionReceipt
- ;
- }
- ;
- Thewait()
- method resolves when the user operation is dispatched by the bundler on-chain and gets mined. ThewaitForTxHash()
- method returns aUserOpStatus
- object which includes the transaction hash and the receipt once added on-chain.

sendSignedUserOp( )

This method is designed to dispatch signed user operations to the bundler. This method is particularly useful when handling operations that have been grouped together with a multi-chain module, as it allows for the submission of these combined operations in a single request. It can also be useful in the case of two different instances of smart account, for example one backend instance to build userOp, while another instance to obtain the signed userOp on the frontend and subsequently dispatch the signed userOp using the backend instance.

Usage

const userOpResponse =

await smartAccount . sendSignedUserOp (userOp) ; note Please ensure that the user operations have been correctly signed before using this method Parameters

- userOp (UserOperation
- , required): TheuserOp
- object includes essential fields likesender
- ,nonce
- ,callData
- ,gas
- related properties, andsignature
- .
- params (SendUserOpParams
-): TheSendUserOpParams
- object can contain fields such assessionID
- ,sessionSigner
- ,sessionValidationModule
- ,additionalSessionData
- ,batchSessionParams
- , andsimulationType
- . These parameters are used to customize the behavior of thesendUserOp
- method and are optional.

Returns

- userOpsResponse (UserOpResponse
-): The method returns an object of typeUserOpResponse
- which has auserOpHash
- and two methods:wait()
- andwaitForTxHash()
- .
- Thewait()
- method resolves when the user operation is dispatched by the bundler on-chain and gets mined. ThewaitForTxHash()
- method returns aUserOpStatus
- object which includes the transaction hash and the receipt once added on-chain.Previous Integration Next Signers