Send a batch of transactions

Overview

This tutorial demonstrates how to send a batch of transactions using ethers.js/viem and the Biconomy Smart Account with the @biconomy/account SDK.

Prerequisites

Node.js installed on your machine

A Bundler url if you don't want to use the testnet one (for mumbai you can use https://bundler.biconomy.io/api/v2/80001/nJPK7B3ru.dd7f7861-190d-41bd-af80-6877f74b8f44)

An rpc url (for mumbai can use https://rpc.ankr.com/polygon_mumbai)

An address to send the transaction to (replace 0xaddress)

Step 1: Generate the config and Create Biconomy Smart Account

viem

ethers

```
import { createWalletClient } from "viem";

import { privateKeyToAccount } from "viem/accounts";

import { polygonMumbai } from "viem/chains";

import { createSmartAccountClient } from "@biconomy/account";


// Your configuration with private key and Biconomy API key

const config = {

  privateKey: "your-private-key",

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

};


// Generate EOA from private key using ethers.js

const account = privateKeyToAccount("0x" + config.privateKey);

const client = createWalletClient({

  account,

  chain: polygonMumbai,

  transport: http(),

});
```

```
// Create Biconomy Smart Account instance
const smartWallet = await createSmartAccountClient({
  signer: client,
  bundlerUrl: config.bundlerUrl,
});

const saAddress = await smartWallet.getAccountAddress();
console.log("SA Address", saAddress);
```

Get your signer from either ethers.js or viem and create a Biconomy Smart Account instance.

Step 2: Generate Transaction Data

```
const toAddress = "0xaddress"; // Replace with the recipient's address
const transactionData = "0x123"; // Replace with the actual transaction data

// Build the transactions
const tx = {
  to: toAddress,
  data: transactionData,
};
const txs = [tx, tx]; // Send the tx twice
```

Specify the recipient's address and transaction data to build the simple transaction. For this example we simply copy and paste the same dummy tx but the tx's here can be different.

Step 3: Send the Transaction and wait for the Transaction Hash

```
// Send the transaction and get the transaction hash
const userOpResponse = await smartWallet.sendTransaction(txs);
const { transactionHash } = await userOpResponse.waitForTxHash();
console.log("Transaction Hash", transactionHash);
```

```
const userOpReceipt  = await userOpResponse.wait();
if(userOpReceipt.success == 'true') {
  console.log("UserOp receipt", userOpReceipt)
  console.log("Transaction receipt", userOpReceipt.receipt)
}
```

Send the transaction using the Biconomy Smart Account and get the transaction hash. The transaction will be built into a User Operation and then sent to the Bundler.

That's it! You've successfully sent a simple transaction using ethers.js/viem and the Biconomy Smart Account. Feel free to customize this example based on your specific use case.

Send a gasless transaction

Overview

This tutorial demonstrates how to send a simple transaction using ethers.js/viem and the Biconomy Smart Account with the @biconomy/account SDK. The provided code assumes you have a Biconomy Paymaster API key.

You can get your Biconomy Paymaster API key from the dashboard here.

Prerequisites

- Node.js installed on your machine
- Biconomy Paymaster API key
- A Bundler url if you don't want to use the testnet one (for mumbai you can use https://bundler.biconomy.io/api/v2/80001/nJPK7B3ru.dd7f7861-190d-41bd-af80-6877f74b8f44)
- An rpc url (for mumbai can use https://rpc.ankr.com/polygon_mumbai)
- An address to send the transaction to (replace 0xaddress)

Step 1: Generate the config and Create Biconomy Smart Account

- viem
- ethers

```
import { createWalletClient } from "viem";
import { privateKeyToAccount } from "viem/accounts";
import { polygonMumbai } from "viem/chains";
import { createSmartAccountClient, PaymasterMode } from "@biconomy/account";
```

```
// Your configuration with private key and Biconomy API key
const config = {
  privateKey: "your-private-key",
  biconomyPaymasterApiKey: "your-biconomy-api-key",
  bundlerUrl: "", // <-- Read about this at https://docs.biconomy.io/dashboard#bundler-url
};

// Generate EOA from private key using ethers.js
const account = privateKeyToAccount("0x" + config.privateKey);
const client = createWalletClient({
  account,
  chain: polygonMumbai,
  transport: http(),
});

// Create Biconomy Smart Account instance
const smartWallet = await createSmartAccountClient({
  signer: client,
  biconomyPaymasterApiKey: config.biconomyPaymasterApiKey,
  bundlerUrl: config.bundlerUrl,
});

const saAddress = await smartWallet.getAccountAddress();
console.log("SA Address", saAddress);
```

Get your signer from either ethers.js or viem and create a Biconomy Smart Account instance.

Step 2: Generate Transaction Data

```
const toAddress = "0xaddress"; // Replace with the recipient's address
const transactionData = "0x123"; // Replace with the actual transaction data

// Build the transaction
const tx = {
  to: toAddress,
  data: transactionData,
};
```

Specify the recipient's address and transaction data to build the simple transaction.

Step 3: Send the Transaction and wait for the Transaction Hash

```
// Send the transaction and get the transaction hash
const userOpResponse = await smartWallet.sendTransaction(tx, {
  paymasterServiceData: {mode: PaymasterMode.SPONSORED},
});
const { transactionHash } = await userOpResponse.waitForTxHash();
```

```
console.log("Transaction Hash", transactionHash);
const userOpReceipt  = await userOpResponse.wait();
if(userOpReceipt.success == 'true') {
  console.log("UserOp receipt", userOpReceipt)
  console.log("Transaction receipt", userOpReceipt.receipt)
}
```

Send the transaction using the Biconomy Smart Account and get the transaction hash. The transaction will be built into a User Operation and then send to the Bundler.

That's it! You've successfully sent a simple transaction using ethers.js/viem and the Biconomy Smart Account. Feel free to customize this example based on your specific use case.

Send a simple transaction

Overview

This tutorial demonstrates how to send a simple transaction using ethers.js/viem and the Biconomy Smart Account with the @biconomy/account SDK.

Prerequisites

- Node.js installed on your machine
- A Bundler url if you don't want to use the testnet one (for mumbai you can use https://bundler.biconomy.io/api/v2/80001/nJPK7B3ru.dd7f7861-190d-41bd-af80-6877f74b8f44)
- An rpc url (for mumbai can use https://rpc.ankr.com/polygon_mumbai)
- An address to send the transaction to (replace 0xaddress)

Step 1: Generate the config and Create Biconomy Smart Account

- viem
- ethers

```
import { createWalletClient } from "viem";
import { privateKeyToAccount } from "viem/accounts";
import { polygonMumbai } from "viem/chains";
import { createSmartAccountClient } from "@biconomy/account";

// Your configuration with private key and Biconomy API key
const config = {
  privateKey: "your-private-key",
  bundlerUrl: "", // <-- Read about this at https://docs.biconomy.io/dashboard#bundler-url
};

// Generate EOA from private key using ethers.js
const account = privateKeyToAccount("0x" + config.privateKey);
const signer = createWalletClient({
  account,
  chain: polygonMumbai,
  transport: http(),
});

// Create Biconomy Smart Account instance
const smartWallet = await createSmartAccountClient({
  signer,
  bundlerUrl: config.bundlerUrl,
});

const saAddress = await smartWallet.getAccountAddress();
console.log("SA Address", saAddress);
```

Get your signer from either ethers.js or viem and create a Biconomy Smart Account instance.

Step 2: Generate Transaction Data

```
const toAddress = "0xaddress"; // Replace with the recipient's address
const transactionData = "0x123"; // Replace with the actual transaction data

// Build the transaction
const tx = {
  to: toAddress,
  data: transactionData,
};
```

Specify the recipient's address and transaction data to build the simple transaction.

Step 3: Send the Transaction and wait for the Transaction Hash

```
// Send the transaction and get the transaction hash
const userOpResponse = await smartWallet.sendTransaction(tx);
const { transactionHash } = await userOpResponse.waitForTxHash();
console.log("Transaction Hash", transactionHash);

const userOpReceipt  = await userOpResponse.wait();
if(userOpReceipt.success == 'true') {
  console.log("UserOp receipt", userOpReceipt)
  console.log("Transaction receipt", userOpReceipt.receipt)
}
```

Send the transaction using the Biconomy Smart Account and get the transaction hash. The transaction will be built into a User Operation and then sent to the Bundler.

That's it! You've successfully sent a simple transaction using ethers.js/viem and the Biconomy Smart Account. Feel free to customize this example based on your specific use case.

Integration

To see how this interacts with other packages in the sdk you can view the Smart Accounts Integration page.

Installation

First, install the required packages for initializing the Paymaster.

- npm
- yarn
- pnpm

```
npm install @biconomy/paymaster
```

Integration

Set up a paymaster for the smart account by providing only the paymaster API key. The default mode used is Sponsored mode in this case.

INFO

Click here to learn more about our dashboard and how to get your Paymaster api key.

```
 import { createSmartAccountClient } from "@biconomy/account";

 const biconomySmartAccount = await createSmartAccountClient(
  {
    signer: signer,
    chainId: 80001
 bundlerUrl:
   "https://docs.biconomy.io/dashboard#bundler-url", // <-- Read about this here
 biconomyPaymasterApiKey: "", // <-- Read about at
https://docs.biconomy.io/dashboard/paymaster
   }
 );
```

If you wish to pass "strictMode" to your Paymaster you need to create the instace.

```
 import { createSmartAccountClient, IPaymaster, BiconomyPaymaster } from
"@biconomy/account";

 const paymaster: IPaymaster = new BiconomyPaymaster({
   //https://dashboard.biconomy.io/ get paymaster urls from your dashboard
   paymasterUrl: "",
   strictMode: true
 });

 const biconomySmartAccount = await createSmartAccountClient(
  {
    signer: signer,
    chainId: 80001
    paymaster,
  bundlerUrl:
   "https://bundler.biconomy.io/api/v2/{chain-id-here}/nJPK7B3ru.dd7f7861-190d-41bd-af80-
6877f74b8f44"    }
  );
```

You have now assigned the Paymaster to the smart account. See our tutorials for in depth integrations of the Smart Account and Paymaster.

Paymaster Methods

Imports needed for these methods:

```
import {
  IHybridPaymaster,
  PaymasterFeeQuote,
  PaymasterMode,
  SponsorUserOperationDto,
  createPaymaster,
} from "@biconomy/account";

const biconomyPaymaster =
  smartAccount.paymaster as IHybridPaymaster<SponsorUserOperationDto>;

// Or...

const biconomyPaymaster = await createPaymaster({ paymasterUrl }); // Found at
https://dashboard.biconomy.io
```

After setting up your smart account as mentioned above you can start using the Paymaster.

getPaymasterFeeQuotesOrData()

This method is particularly useful for ERC20 mode and is used to get the fee quotes information for the ERC20 tokens.

Usage

```
const feeQuotesResponse = await biconomyPaymaster.getPaymasterFeeQuotesOrData(
  userOp,
  {
    mode: PaymasterMode.ERC20,
    tokenList: ["0xdA5289FCAAF71d52A80A254dA614A192B693e975"],
    preferredToken: "0xdA5289FCAAF71d52A80A254dA614A192B693e975",
  }
);
```

As per the code

- We set the mode to ERC20
- In the token list we can specify a list of addresses of the ERC20 tokens we want to have our users pay in.
- We can also decide to choose a preferred token for the response to include.

Parameters

- userOp(Partial<UserOperation>, required): A UserOperation object representing the user's request that needs to be processed.
- paymasterServiceData(FeeQuotesOrDataDto, required): The paymaster service data containing token information and sponsorship details. You can send just the preferred token or an array of token addresses.

```
type FeeQuotesOrDataDto = {
 mode?: PaymasterMode; // enum values ERC20 and SPONSORED
 expiryDuration?: number; // Specifies the duration, in seconds, for which the user intends the
paymasterAndData to remain valid, minimum duration is 300 secs.
 calculateGasLimits?: boolean;
 tokenList?: string[]; // If you pass tokenList as an empty array. and it
 // would return fee quotes for all tokens supported by the Biconomy paymaster
 preferredToken?: string; // If you want to pass only one token
 webhookData?: Record<string, any>;
 smartAccountInfo?: SmartAccountData; // use this to calulate the gas for smart account v1,
otherwise by default it will return for latest smart account version
};
```

Returns

- response(Promise<FeeQuotesOrDataResponse>): It returns a promise that resolves to the following object containing a feeQuotes Array. A paymaster Fee quote will have additional information about the fee being paid in the specified ERC20 token, how long this fee is valid, what type of premium is going to be paid, as well as general token information and the amount in USD. Here is the full typing for a single fee quote:

```
type FeeQuotesOrDataResponse = {
 feeQuotes?: PaymasterFeeQuote[];
 tokenPaymasterAddress?: string;
 paymasterAndData?: string;
 preVerificationGas?: BigNumberish;
 verificationGasLimit?: BigNumberish;
 callGasLimit?: BigNumberish;
};

type PaymasterFeeQuote = {
 symbol: string; // ERC20 token symbol
 tokenAddress: string; // ERC20 token address
 decimal: number; //  number of decimal places used to represent the token
 logoUrl?: string; // logo url of the ERC20 token
 maxGasFee: number;
 maxGasFeeUSD?: number;
 usdPayment?: number; // the fee converted into the USD
 premiumPercentage: number; // premium in percentage that biconomy charges for the txn,
generally ranges between 7-12
 validUntil?: number; // number (epoch time in milliseconds, till these fee quotes are valid)
};
```

getPaymasterAndData()

The getPaymasterAndData method is used to get the information on how the transaction fees will be covered for the transactions. The returned PaymasterAndDataResponse includes a signed string (paymasterAndData) that signifies the paymaster's commitment to covering the transaction fee.

Parameters

- userOperation (UserOperation, required): A UserOperation object representing the user's request that needs to be processed.
- paymasterServiceData (SponsorUserOperationDto, Optional): An optional parameter that allows you to provide additional data specific to the paymaster service.

```
type SponsorUserOperationDto = {
  mode: PaymasterMode;
  calculateGasLimits?: boolean;
  expiryDuration?: number;
  webhookData?: {
    [key: string]: any;
  };
  smartAccountInfo?: SmartAccountData;
  feeTokenAddress?: string;
};
```

Returns

- paymasterAndDataResponse(Promise<PaymasterAndDataResponse>): It returns a promise that resolves to a Paymaster and Data Response object. The data in this response can be used to update the userOp to include sponsorship data or data for paying gas in ERC20 tokens.

```
type PaymasterAndDataResponse = {
  paymasterAndData: string;
  preVerificationGas?: BigNumberish;
  verificationGasLimit?: BigNumberish;
  callGasLimit?: BigNumberish;
};
```

Usage Mode: SPONSORED

There are two modes for using this function: SPONSORED and ERC20.

Here it is meant to act as Sponsorship/Verifying paymaster hence we send mode: PaymasterMode.SPONSORED which is required

```
let paymasterServiceData: SponsorUserOperationDto = {
  mode: PaymasterMode.SPONSORED,
  // optional params...
  calculateGasLimits: true,
};
const paymasterAndDataResponse = await biconomyPaymaster.getPaymasterAndData(
  userOp,
  paymasterServiceData
);

userOp.paymasterAndData = paymasterAndDataResponse.paymasterAndData;
```

As per the code we get the Paymaster Data Response and update the userOp, to specify that this will be a sponsored transaction by the Biconomy Paymaster.

Usage Mode: ERC20

ERC20 mode enables you to pay the gas fees of your users in exchange for ERC-20 tokens. When switching mode to ERC20 there are additional steps that need to be considered.

WARNING

Important: When using **Token Paymaster** with ERC20 tokens, always ensure to calculate the **feeQuote** correctly. This is crucial to avoid transaction reverts due to insufficient token balance after execution. The feeQuote should consider both the transaction cost and any other **token** movements within the same operation.

Example: If a user is transacting with **USDC**, and the feeQuote is **2 USDC**, the DApp must ensure that the user's balance post-callData execution is sufficient to cover this fee. Incorrect fee calculations can lead to transaction failures and a degraded user experience.

1. Get paymaster fee quote: We need to get the swap quotes of the ERC20 tokens that the user is paying for the native token of the network we are on.

```
const feeQuotesResponse = await biconomyPaymaster.getPaymasterFeeQuotesOrData(
  userOp,
  {
    mode: PaymasterMode.ERC20,
    tokenList: [],
    preferredToken: "",
  }
);
```

2. Update userOp: After getting this information we need to build our updated userOp with the preferred feeQuote from the feeQuotes array.

```
finalUserOp = await smartAccount.buildTokenPaymasterUserOp(userOp, {
  feeQuote: selectedFeeQuote,
  spender: spender,
  maxApproval: false,
});
```

3. Create paymaster data: Now we create our paymasterServiceData again and call the getPaymasterAndData method

```
let paymasterServiceData = {
  mode: PaymasterMode.ERC20,
  feeTokenAddress: selectedFeeQuote.tokenAddress,
  calculateGasLimits: true, // Always recommended and especially when using token paymaster
};
const paymasterAndDataWithLimits = await biconomyPaymaster.getPaymasterAndData(
  finalUserOp,
  paymasterServiceData
);
finalUserOp.paymasterAndData = paymasterAndDataWithLimits.paymasterAndData;
```