Tutorial 2 — Submit a user operation with a Verifying Paymaster

This is a low-level tutorial that walks you through the steps of constructing a user operation from scratch. If you would like to leverage permissionless.js's high-level functions, take a look at tutorial 1 .

In this tutorial, you will generate a user operation, ask Pimlico's verifying paymaster to sponsor it, and then submit the sponsored user operation on-chain with Pimlico's Alto bundler.

Steps

Get a Pimlico API key

To get started, please go to our dashboard and generate a Pimlico API key.

Clone the Pimlico tutorial template repository

We have created a Pimlico tutorial template repository that you can use to get started. It comes set up with Typescript, viem, and permissionless.js.

```
gitclone https://github.com/pimlicolabs/tutorial-template.git pimlico-tutorial-2 cdpimlico-tutorial-2
```

Now, let's install the dependencies:

```
npminstall
```

The main file we will be working with isindex.ts . Let's run it to make sure everything is working:

```
npmstart
```

If everything has been set up correctly, you should seeHello world! printed to the console.

Create the viem clients

We will be using three different clients for this example.

1. Standard publicClient for normal Ethereum RPC calls —https://rpc.goerli.linea.build
2. Pimlico v1 api for the Bundler methods —https://api.pimlico.io/v1/linea-testnet/rpc?apikey=YOUR_PIMLICO_API_KEY
3. Pimlico v2 api for the Paymaster methods —https://api.pimlico.io/v2/linea-testnet/rpc?apikey=YOUR_PIMLICO_API_KEY

Make sure to replaceYOUR_PIMLICO_API_KEY in the code below with your actual Pimlico API key.

Let's open upindex.ts , and add the following to the bottom:

```
exportconstpublicClient=createPublicClient({ transport:http("https://rpc.ankr.com/eth_sepolia"), chain: sepolia, })

constapiKey="YOUR_PIMLICO_API_KEY"// REPLACE THIS constendpointUrl=https://api.pimlico.io/v2/sepolia/rpc?apikey={apiKey}

constbundlerClient=createClient({ transport:http(endpointUrl), chain: sepolia, }) .extend(bundlerActions(ENTRYPOINT_ADDRESS_V07))
.extend(pimlicoBundlerActions(ENTRYPOINT_ADDRESS_V07))

constpaymasterClient=createClient({ transport:http(endpointUrl), chain: sepolia, }).extend(pimlicoPaymasterActions(ENTRYPOINT_ADDRESS_V07))
```

Generate the factory and factoryData

For the purposes of this guide, we will be using theSimpleAccount.sol wallet found in the eth-infinitism repository. This Wallet is a simple ERC-4337 wallet controlled by a single EOA signer.

At0x91E60e0613810449d098b0b5Ec8b51A0FE8c8985 , most chain already have deployed aSimpleAccountFactory.sol contract, that is able to easily deploy new SimpleAccount instances via thecreateAccount function. We will be leveraging this contract to help us generate thefactory andfactoryData .

Requesting a smart contract deployment is done through thefactory andfactoryData field, where thefactory field specifies the address the EntryPoint will call, and thefactoryData corresponds to the data that will be called on that factory.

Add the following to the bottom ofindex.ts :

```
constSIMPLE_ACCOUNT_FACTORY_ADDRESS="0x91E60e0613810449d098b0b5Ec8b51A0FE8c8985"

constownerPrivateKey=generatePrivateKey() constowner=privateKeyToAccount(ownerPrivateKey)

console.log("Generated wallet with private key:", ownerPrivateKey)

constfactory=SIMPLE_ACCOUNT_FACTORY_ADDRESS constfactoryData=encodeFunctionData({ abi: [ { inputs: [ { name:"owner", type:"address"}, { name:"salt", type:"uint256"}, ],
name:"createAccount", outputs: [{ name:"ret", type:"address"}], stateMutability:"nonpayable", type:"function", }, ], args: [owner.address,0n], })

console.log("Generated factoryData:", factoryData)
```

Let's run this code withnpm start . You should see the generated initCode printed to the console.

```
Generated factoryData:
0x9406cc6185a346906296840746125a0e449764545fbfb9cf000000000000000000000000508a7005f997a21cd662d6fb41ad69f945c129c1000000000000000000000000000000000000000000
```

Calculate the sender address

Now that we have thefactory andfactoryData , we have to calculate the correspondingsender address, which is the address theSimpleAccount will be deployed to, and thereby the address which will handle the verification and execution steps of the UserOperation.

We do this by calling thegetSenderAddress utility function on the EntryPoint. Upon success, it will revert with a special error type that contains the counterfactual address of the smart contract wallet that will be deployed

Add the following to the bottom ofindex.ts :

```
constsenderAddress=awaitgetSenderAddress(publicClient, { factory, factoryData, entryPoint:ENTRYPOINT_ADDRESS_V07, }) console.log("Calculated sender address:", senderAddress)
```

```

Let's run this code withnpm start . You should see the address printed to the console.

```
Calculated sender address: 0xbAd38BdCf884ED92ab370f69C0CD0B7b8a1459A1
```

### Generate the callData

Now, let's decide on thecallData that we want the wallet to actually execute once the UserOperation passes verification.

Add the following to the bottom ofindex.ts :

```
constto="0xd8dA6BF26964aF9D7eEd9e03E53415D37aA96045"// vitalik constvalue=0n constdata="0x68656c6c6f"// "hello" encoded to utf-8 bytes

constcallData=encodeFunctionData({ abi: [ { inputs: [ { name:"dest", type:"address"}, { name:"value", type:"uint256"}, { name:"func", type:"bytes"}, ], name:"execute", outputs: [], stateMutability:"nonpayable", type:"function", }, ], args: [to, value, data], })

console.log("Generated callData:", callData)
```

Above, we are leveraging theexecute function of the SimpleWallet, which simply calls an arbitrary address, with arbitrary value and arbitrary callData.

Let's run this code withnpm start . You should see the callData printed to the console.

```
Generated callData:
0xb61d27f6000000000000000000000000d8da6bf26964af9d7eed9e03e53415d37aa9604500000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Fill out remaining UserOperation values

We're almost there, now let's fill out the rest of the UserOperation values.

Add the following to the bottom ofindex.ts :

```
constgasPrice=awaitbundlerClient.getUserOperationGasPrice()

constuserOperation={ sender: senderAddress, nonce:0n, factory: factoryasAddress, factoryData, callData, maxFeePerGas: gasPrice.fast.maxFeePerGas, maxPriorityFeePerGas: gasPrice.fast.maxPriorityFeePerGas, // dummy signature, needs to be there so the SimpleAccount doesn't immediately revert because of invalid signature length signature: "0xa15569dd8f8324dbeabf8073fdec36d4b754f53ce5901e283c6de79af177dc94557fa3c9922cd7af2a96ca94402d35c39f266925ee6407aeb32b31d76978d4ba1c"asHex, }
```

Request Pimlico verifying paymaster sponsorship

To make the operation gasless, we will leverage Pimlico's verifying paymaster. Using paymasters allows you to delegate the gas fee payment to a third-party contract that can decide whether it is willing to pay the gas fees for the user operation. In this case, Pimlico's verifying paymaster checks whether its off-chain signer has signed off on the user operation. To request Pimlico's signer to sign your user operation, call thepm_sponsorUserOperation endpoint or use thesponsorUserOperation method from the permissionless.js Pimlico paymaster actions.

Add the following to the bottom ofindex.ts :

```
constsponsorUserOperationResult=awaitpaymasterClient.sponsorUserOperation({ userOperation, })

constsponsoredUserOperation:UserOperation<"v0.7">={ ...userOperation, ...sponsorUserOperationResult, }

console.log("Received paymaster sponsor result:", sponsorUserOperationResult)
```

Let's run this code withnpm start . You should see something like this:

```
Received paymaster sponsor result: { paymaster: '0xcF60744ef322396a6d0a5B7d396F5814176855F1', paymasterVerificationGasLimit: 526114n, paymasterPostOpGasLimit: 75900n, paymasterData: '0000000000000000000000000000000000000000000000000000000000000000000006518a0c10000000000000000000000000000000000000000000000000000000000003251910f0e14691ca19a8e2cca216ce77
preVerificationGas: 247487n, verificationGasLimit: 526114n, callGasLimit: 75900n }
```

Great! Now we have received the gas limit estimates and added the paymaster-related fields containing Pimlico's signature to our UserOperation.

Sign the UserOperation

The last field to fill out is thesignature . This is a simple ECDSA signature consistent with typical Ethereum private key signing procedures.

Add the following to the bottom ofindex.ts :

```
constsignature=awaitsignUserOperationHashWithECDSA({ account: owner, userOperation: sponsoredUserOperation, chainId: sepolia.id, entryPoint:ENTRYPOINT_ADDRESS_V07, }) sponsoredUserOperation.signature=signature

console.log("Generated signature:", signature)
```

Let's run this code withnpm start . You should see something like this:

```
Generated signature: 0xcaae357fcd3882f1ea4b48f7dcce9d7f2482794ab72d1075ce5d7fcef4c5ec03265fe03e5fd7f8af65a4cd05b7e01300f3938f7e245dc8038748ddef93d5f4061c
```

Submit the UserOperation to be bundled

Finally, we're ready to submit the UserOperation to Pimlico's bundler, which will include it on-chain. Theeth_sendUserOperation RPC call or thesendUserOperation method from permissionless.js will submit the UserOperation to the bundler.

You can also query for receipts to keep checking the status of the UserOperation until it is included.

Add the following to the bottom ofindex.ts :
```

```
constuserOperationHash=awaitbundlerClient.sendUserOperation({ userOperation: sponsoredUserOperation, })

console.log("Received User Operation hash:", userOperationHash)

// let's also wait for the userOperation to be included, by continually querying for the receipts console.log("Querying for receipts...") constreceipt=awaitbundlerClient.waitForUserOperationReceipt({ hash: userOperationHash, }) consttxHash=receipt.receipt.transactionHash

console.log(UserOperation included: https://sepolia.etherscan.io/tx{txHash})
```

If we run this code withnpm start , we will go through the whole flow of executing the User Operation. You should see something like this:

```
UserOperation included: https://goerli.lineascan.build/tx/0x43bdf7e2dfc19bfb749376b91d872574668365493ea98f9c9a647a17f541fb96
```

Once the UserOperation is included, you can view the transaction on the Linea Goerli testnet explorer.

That's it! You've successfully generated a UserOperation and submitted it using Pimlico's Alto bundler.

By leveraging Pimlico's paymaster, you were able to make the User Operation completely gasless, and by using Pimlico's Alto bundler, you were able to submit the User Operation to the chain without having to worry about maintaining your own relaying infrastructure.

Congratulations, you are now a pioneer of Account Abstraction!

Please get in touch if you have any questions or if you'd like to share what you're building!

Combined code

If you want to see the complete code that combines all of the previous steps, we uploaded it to a separate repository . If you're looking to run it, remember to replace the API key with your own!