How to send a userOperation from a EOA using EIP-7702

This guide showcases a simple demo that uses ERC-4337 and EIP-7702 to send a sponsored userOperation from a EOA. We will use Safe as our smart account implementation of choice, but the same applies for any ERC-4337 compatible smart account.

For a high level overview of EIP-7702, checkout our EIP-7702 conceptual guide and for a more technical overview, please refer to the EIP-7702 proposal .

Steps

This guide is divided into two parts. The first part will walk you through how to turn your EOA into a ERC-4337 compatible smart account using EIP-7702. The second part will show you how to send a sponsored userOperation originating from your EOA.

Part 1: Sending a EIP-7702 set code transaction

Setup

This demo will be ran on Odyssey Testnet , which already implements EIP-7702.

To get started, you can bridge funds from Sepolia to Odyssey through Conduit's SuperBridge .

You can confirm the bridge transfer by checking the Odyssey blockchain explorer .

Confirming the EOA has no code

Before starting the demo, we can quickly confirm that our EOA has no code attached to it by running the following command:

```
castcodeYOUR_EOA_ADDRESS--rpc-urlhttps://odyssey.ithaca.xyz
```

If should return back the following:

```
0x
```

Signing the Authorization Request

We first need to prepare our EOA by signing a authorization request to set the Safe Singleton contract as our designated delegator. To do this, we extend our wallet client with Viem's experimental EIP-7702 actions .

```
import{ createWalletClient, Hex, http, zeroAddress }from"viem" import{ privateKeyToAccount, privateKeyToAddress }from"viem/accounts" import{ odysseyTestnet }from"viem/chains" import{ eip7702Actions }from"viem/experimental" import{ safeAbiImplementation }from"./safeAbi" import{ getSafeModuleSetupData }from"./setupData" importdotenvfrom"dotenv" dotenv.config()

consteoaPrivateKey=process.env.EOA_PRIVATE_KEYasHex if(!eoaPrivateKey)thrownewError("EOA_PRIVATE_KEY is required")

constaccount=privateKeyToAccount(eoaPrivateKey)

constwalletClient=createWalletClient({ account, chain: odysseyTestnet, transport:http("https://odyssey.ithaca.xyz"), }).extend(eip7702Actions())

constSAFE_SINGLETON_ADDRESS="0x41675C099F32341bf84BFc5382aF534df5C7461a"

constauthorization=awaitwalletClient.signAuthorization({ contractAddress:SAFE_SINGLETON_ADDRESS, })
```

Sending the Authorization Request

Before we can interact with our smart account, we also need to initialize it by populating it's storage. With Safe, this is done by calling thesetup function.

We can make a slight optimization by sending both the Authorization andsetup call in one transaction which would both set our EOA's code and setup our smart account.

Note: We are using a seperate private key,safePrivateKey instead of our EOA's private key to act as our smart account owner. This is because Safe currently doesn't allow the account owner to equal address(this). index.ts setupData.ts safeAbi.ts ```

File index.ts constSAFE_MULTISEND_ADDRESS="0x38869bf66a61cF6bDB996A6aE40D5853Fd43B526" constSAFE_4337_MODULE_ADDRESS="0x75cf11467937ce3F2f357CE24ffc3DBF8fD5c226"

constsafePrivateKey=process.env.SAFE_PRIVATE_KEYasHex|undefined if(!safePrivateKey)thrownewError("SAFE_PRIVATE_KEY is required")

// Parameters for Safe's setup call. constowners=[privateKeyToAddress(safePrivateKey)] constsignerThreshold=1n constsetupAddress=SAFE_MULTISEND_ADDRESS constsetupData=getSafeModuleSetupData() constfallbackHandler=SAFE_4337_MODULE_ADDRESS constpaymentToken=zeroAddress constpaymentValue=0n constpaymentReceiver=zeroAddress

consttxHash=awaitwalletClient.writeContract({ address: account.address, abi: safeAbiImplementation, functionName:"setup", args: [owners, signerThreshold, setupAddress, setupData, fallbackHandler, paymentToken, paymentValue, paymentReceiver,], authorizationList: [authorization], })

console.log(Submitted: https://odyssey-explorer.ithaca.xyz/tx{txHash})

``` Warning : This demo is meant to serve as a overview of what EIP-7702 and ERC-4337 could look like. This approach should not be used in production as a malicious entity could take over your EOA by frontrunning the setup transaction and setting theowners field to an address they control.

### Confirming the EOA has code

Now that the authorization request has been sent, we can confirm that our EOA has code attached to it by running the following command:

```

castcodeYOUR_EOA_ADDRESS--rpc-urlhttps://odyssey.ithaca.xyz

```

If should return back the following:

```

0xef010041675c099f32341bf84bfc5382af534df5c7461a

```

Here the EOA's code is in the format of(0xef0100 ++ address) where0xef0100 are magic bytes that indicate the EOA has a active delegation designator. The remaining bytes0x41675c099f32341bf84bfc5382af534df5c7461a is the Safe Singleton's address.

## Part 2: Sending the UserOperation

### Preparing the clients

The setup process follows the typical flow of sending a userOperation. The only difference is that when creating the Safe smart account instance, we set the sender address as our EOA's address.

```

import{ toSafeSmartAccount }from"permissionless/accounts" import{ createPimlicoClient }from"permissionless/clients/pimlico" import{ createPublicClient, Hex, http, zeroAddress }from"viem" import{ odysseyTestnet }from"viem/chains" import{ privateKeyToAccount, privateKeyToAddress }from"viem/accounts" importdotenvfrom"dotenv" import{ createSmartAccountClient }from"permissionless" dotenv.config()

```
consteoaPrivateKey=process.env.EOA_PRIVATE_KEYasHex|undefined
if(!eoaPrivateKey)thrownewError("EOA_PRIVATE_KEY is required")

constsafePrivateKey=process.env.SAFE_PRIVATE_KEYasHex|undefined
if(!safePrivateKey)thrownewError("SAFE_PRIVATE_KEY is required")

constpimlicoApiKey=process.env.PIMLICO_API_KEYasHex|undefined
if(!pimlicoApiKey)thrownewError("PIMLICO_API_KEY is required")

constpimlicoUrl=https://api.pimlico.io/v2{odysseyTestnet.id}/rpc?apikey={pimlicoApiKey}

constpimlicoClient=createPimlicoClient({ transport:http(pimlicoUrl), })

constpublicClient=createPublicClient({ chain: odysseyTestnet, transport:http("https://odyssey.ithaca.xyz"), })

constsafeAccount=awaittoSafeSmartAccount({ address:privateKeyToAddress(eoaPrivateKey), owners:
[privateKeyToAccount(safePrivateKey)], client: publicClient, version:"1.4.1", })

constsmartAccountClient=createSmartAccountClient({ account: safeAccount, paymaster: pimlicoClient,
bundlerTransport:http(pimlicoUrl), userOperation: { estimateFeesPerGas:async()=>
(awaitpimlicoClient.getUserOperationGasPrice()).fast, }, })
```

Sending the UserOperation

We can now send the userOperation as usual.

```
constuserOperationHash=awaitsmartAccountClient.sendUserOperation({ calls: [ { to: zeroAddress, value:0n, data:"0x", }, ],
})

const{receipt}=awaitsmartAccountClient.waitForUserOperationReceipt({ hash: userOperationHash, })

console.log( UserOperation included: https://odyssey-explorer.ithaca.xyz/tx{receipt.transactionHash}, )
```

Review

Congratulations! You have successfully sent a sponsored userOperation from your EOA, if you review the transaction on the blockchain explorer, you will see that the userOperation's sender address is equal to your EOA's address.

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!