**Introduction to userop.js**

A simple JS library for building ERC-4337 UserOperations.

[Suggest Edits](https://docs.stackup.sh/edit/useropjs)

userop.js is an open-source library for building ERC-4337 User Operations for any bundler or paymaster. Similar to how[ethers.js](https://docs.ethers.org/v5/) allows developers to easily create traditional EVM transactions, userop.js allows developers to create and send User Operations to ERC-4337 Bundlers.

💪 **Implementation agnostic**: Can be used to create User Operations for any ERC-4337 Smart Account, Bundler service, or Paymaster. Completely open source and MIT licensed, so you can modify it however you like, and know you're never locked in.

🏗️ **Intuitive design**: Based on the [builder design pattern](https://refactoring.guru/design-patterns/builder), which mirrors how User Operations are built in production.

✨ **Native TypeScript support**: Builds on top of common ethers types like BigNumberish and BytesLike.

**Install**

userop.js is available as an npm package [here](https://www.npmjs.com/package/userop). To install, simply use your package manager of choice.

NPM

npm install userop

Yarn

yarn add userop

**Send a User Operation**

Open Recipe

Updated 9 days ago

# Client

Connecting to an ERC-4337 bundler is easy using userop.js

[Suggest Edits](https://docs.stackup.sh/edit/useropjs-client)

Userop.js allows you connect to a bundler RPC using the client interface.

An instance of a client is an abstraction for building and sending your User Operations to the eth\_sendUserOperation RPC method on a bundler.

## Interfaces

These interfaces are built using [UserOperation](https://docs.stackup.sh/docs/useropjs-builder" \l "useroperation" \t "_self), [UserOperationBuilder](https://docs.stackup.sh/docs/useropjs-builder" \l "useroperationbuilder" \t "_self), and [BigNumberish](https://docs.ethers.io/v5/api/utils/bignumber/" \l "BigNumberish" \t "_self) from [ethers.js](https://docs.ethers.io/" \t "_self).

### Client

An instance of client exposes methods to help get your UserOperation to an ERC-4337 Bundler.

TypeScript

interface IClient {

sendUserOperation: (

builder: IUserOperationBuilder,

opts?: ISendUserOperationOpts

) => Promise<ISendUserOperationResponse>;

buildUserOperation: (

builder: IUserOperationBuilder

) => Promise<IUserOperation>;

}

interface ISendUserOperationOpts {

onBuild?: (op: IUserOperation) => Promise<any> | any;

}

interface ISendUserOperationResponse {

userOpHash: string;

wait: () => Promise<UserOperationEventEvent | null>;

}

## Usage

One instance should be initialized for each node and network your application supports.

JavaScript

import { Client } from "userop";

const client = await Client.init(rpcUrl, entryPoint);

### sendUserOperation

A method for directing a [builder](https://docs.stackup.sh/docs/useropjs-builder) instance to create a User Operation and send it to a bundler via [eth\_sendUserOperation](https://docs.stackup.sh/reference/eth-senduseroperation" \t "_self).

TypeScript

const response = await client.sendUserOperation(builder);

const userOperationEvent = await response.wait();

📘

Note

This method will also call resetOp on a [builder](https://docs.stackup.sh/docs/useropjs-builder) if successful.

### buildUserOperation

This method can be used to direct a [builder](https://docs.stackup.sh/docs/useropjs-builder) using the client's entryPoint and chainID. However it will only return the UserOperation and not initiate a send request.

TypeScript

const userOp = await client.buildUserOperation(builder);

📘

Note

This method will **not** call resetOp on a [builder](https://docs.stackup.sh/docs/useropjs-builder).

### Constants

A instance of a client has several constants that can be set.

TypeScript

// The maximum amount of time to wait for the UserOperationEvent after calling response.wait()

client.waitTimeoutMs = 30000;

// The interval at which it will poll the node to look up UserOperationEvent.

client.waitIntervalMs = 5000;

Updated 9 days ago

# Builder

Learn how to use userop.js to build ERC-4337 UserOperations.

[Suggest Edits](https://docs.stackup.sh/edit/useropjs-builder)

A [UserOperation](https://docs.stackup.sh/docs/erc-4337-overview" \l "user-operations" \t "_self) is a pseudo-transaction object used to execute actions through a smart contract account. Although it can be quite complex to create, the UserOperationBuilder simplifies this process using the [builder pattern](https://refactoring.guru/design-patterns/builder). The interface is also agnostic to any ERC-4337 Smart Account or Paymaster implementation.

## Interfaces

These interfaces are built using common [ethers.js](https://docs.ethers.io/) types. More specifically [BigNumberish](https://docs.ethers.io/v5/api/utils/bignumber/" \l "BigNumberish" \t "_self) and [BytesLike](https://docs.ethers.io/v5/api/utils/bytes/" \l "BytesLike" \t "_self).

### UserOperation

An interface for an ERC-4337 User Operation. Building a UserOperation involves constructing multiple parts and merging them together.

TypeScript

interface IUserOperation {

sender: string;

nonce: BigNumberish;

initCode: BytesLike;

callData: BytesLike;

callGasLimit: BigNumberish;

verificationGasLimit: BigNumberish;

preVerificationGas: BigNumberish;

maxFeePerGas: BigNumberish;

maxPriorityFeePerGas: BigNumberish;

paymasterAndData: BytesLike;

signature: BytesLike;

}

### UserOperationBuilder

An instance of UserOperationBuilder can help build a UserOperation that can be passed to the [client](https://docs.stackup.sh/docs/useropjs-client#senduseroperation).

TypeScript

interface IUserOperationBuilder {

// get methods.

getSender: () => string;

getNonce: () => BigNumberish;

getInitCode: () => BytesLike;

getCallData: () => BytesLike;

getCallGasLimit: () => BigNumberish;

getVerificationGasLimit: () => BigNumberish;

getPreVerificationGas: () => BigNumberish;

getMaxFeePerGas: () => BigNumberish;

getMaxPriorityFeePerGas: () => BigNumberish;

getPaymasterAndData: () => BytesLike;

getSignature: () => BytesLike;

getOp: () => IUserOperation;

// set methods.

setSender: (address: string) => IUserOperationBuilder;

setNonce: (nonce: BigNumberish) => IUserOperationBuilder;

setInitCode: (code: BytesLike) => IUserOperationBuilder;

setCallData: (data: BytesLike) => IUserOperationBuilder;

setCallGasLimit: (gas: BigNumberish) => IUserOperationBuilder;

setVerificationGasLimit: (gas: BigNumberish) => IUserOperationBuilder;

setPreVerificationGas: (gas: BigNumberish) => IUserOperationBuilder;

setMaxFeePerGas: (fee: BigNumberish) => IUserOperationBuilder;

setMaxPriorityFeePerGas: (fee: BigNumberish) => IUserOperationBuilder;

setPaymasterAndData: (data: BytesLike) => IUserOperationBuilder;

setSignature: (bytes: BytesLike) => IUserOperationBuilder;

setPartial: (partialOp: Partial<IUserOperation>) => IUserOperationBuilder;

// Sets the default values that won't be wiped on reset.

useDefaults: (partialOp: Partial<IUserOperation>) => IUserOperationBuilder;

resetDefaults: () => IUserOperationBuilder;

// Some fields may require arbitrary logic to build an op.

// Middleware functions allow you to set custom logic for building op fragments.

useMiddleware: (fn: UserOperationMiddlewareFn) => IUserOperationBuilder;

resetMiddleware: () => IUserOperationBuilder;

// This will construct a UserOperation that can be sent to a client.

// It will run through your entire middleware stack in the process.

buildOp: (

entryPoint: string,

chainId: BigNumberish

) => Promise<IUserOperation>;

// Will reset all fields back to default value.

resetOp: () => IUserOperationBuilder;

}

## Usage

TypeScript

import { UserOperationBuilder } from "userop";

const builder = new UserOperationBuilder().useDefaults({ sender });

🚧

Using Defaults

The useDefaults method will set fields that will persist after calling resetOp. This could be for fields like sender which you don't expect to change across different operations.

### Building a UserOperation

A UserOperation is built using the buildOp method once it is properly configured. The easiest way to avoid passing around entryPoint and chainID values is to [use the client as a director](https://docs.stackup.sh/docs/useropjs-client).

TypeScript

// If you only want to build.

const userOp = await client.buildUserOperation(builder);

// If you want to build and send.

const result = await client.sendUserOperation(builder);

📘

Note

Using the above methods on a client to direct a builder will also call resetOp if successful.

Alternatively, if you want to control the build process:

TypeScript

// Build op with the middleware stack.

let userOp = await builder.buildOp(entryPoint, chainId);

// Or get the latest built op. Will not use the middleware stack.

let userOp = await builder.getOp();

// Reset op back to default values when you're done.

builder.resetOp();

### get and set Functions

These are basic getters and setters for all fields on a UserOperation. Getters return the field type whereas setters will return the instance to enable chaining.

For example:

TypeScript

const builder = new UserOperationBuilder()

.setCallData(callData)

.setCallGasLimit(callGas);

🚧

Careful when setting gas values

Avoid setting hard-coded gas prices in your user operations. We recommend using the built-in gas price middleware to set gas prices or a gas price oracle.

### Middleware Functions

Some fragments on a UserOperation may depend on custom logic in order to be built. For example, based on your Smart Account, there might be a specific ways to sign an operation which aren't specified in the standard.

For such cases we can set custom middleware functions. During buildOp, a middleware will be called in the order they are set. Here is a example of middleware functions you might have in your application:

TypeScript

const resolveAccount => async (ctx) => {

// Fetch the latest nonce and initCode if required.

ctx.op.nonce = nonce;

ctx.op.initCode = initCode;

};

const fetchGasPrice = async (ctx) => {

// Fetch the latest gas prices.

ctx.op.maxFeePerGas = maxFeePerGas;

ctx.op.maxFeePerGas = maxPriorityFeePerGas;

};

const verifyingPaymaster = async (ctx) => {

// Request gas sponsorship from a paymaster provider.

ctx.op.paymasterAndData = paymasterAndData;

ctx.op.preVerificationGas = preVerificationGas;

ctx.op.verificationGasLimit = verificationGasLimit;

ctx.op.callGasLimit = callGasLimit;

};

const signUserOperation = async (ctx) => {

// Use the required signature scheme based on your wallet.

// ctx.getRequestId() will generate the required hash for verification.

// Multisig, ECDSA, etc.

ctx.op.signature = signature;

};

const builder = new UserOperationBuilder()

.useMiddleware(resolveAccount)

.useMiddleware(fetchGasPrice)

.useMiddleware(verifyingPaymaster)

.useMiddleware(signUserOperation);

📘

Looking for a concrete example?

Checkout the [SimpleAccount](https://github.com/stackup-wallet/userop.js/blob/main/src/preset/builder/simpleAccount.ts" \t "_self) preset based on [SimpleAccount.sol](https://github.com/stackup-wallet/userop.js/blob/main/src/preset/builder/simpleAccount.ts" \t "_self).

Updated 3 months ago

# Presets

Userop.js ships with common presets that can help you get started even quicker with certain use cases.

[Suggest Edits](https://docs.stackup.sh/edit/useropjs-presets)

The [builder](https://docs.stackup.sh/docs/useropjs-builder) interface is flexible enough to support any implementation of an ERC-4337 Smart Account. However, there are already a few common implementations used within the ecosystem already. Rather then create the same builder and middleware functions over and over again, we've shipped a few useful presets for commonly used implementations.

📘

Preset for [insert favourite contract account] is not included?? :(

No worries! Userop.js is open for PRs. Alternatively, let us know what type of presets you'd like to see and we'll work together to get it shipped. 🤝

## Builder

Builder presets are pre-configured [builders](https://docs.stackup.sh/docs/useropjs-builder) over a known contract account implementation. Presets can be used as-is or modified with the[get and set functions](https://docs.stackup.sh/docs/useropjs-builder#get-and-set-functions).

📘

Looking for a concrete example?

All builder presets are also used in the [ERC-4337 examples repo](https://docs.stackup.sh/docs/erc-4337-examples). For each preset, you can find working code samples in the [scripts directory](https://github.com/stackup-wallet/erc-4337-examples/tree/main/scripts).

### SimpleAccount

The SimpleAccount preset is an abstraction to build User Operations for an ERC-4337 account based on [SimpleAccount.sol](https://github.com/eth-infinitism/account-abstraction/blob/develop/contracts/samples/SimpleAccount.sol" \t "_self).

TypeScript

import { Client, Presets } from "userop";

const simpleAccount = await Presets.Builder.SimpleAccount.init(

signer, // Any object compatible with ethers.Signer

config.rpcUrl

);

const client = await Client.init(config.rpcUrl);

const res = await client.sendUserOperation(

simpleAccount.execute(target, value, "0x"),

{ onBuild: (op) => console.log("Signed UserOperation:", op) }

);

console.log(`UserOpHash: ${res.userOpHash}`);

console.log("Waiting for transaction...");

const ev = await res.wait();

console.log(`Transaction hash: ${ev?.transactionHash ?? null}`);

### Kernel

The Kernel preset is an abstraction to build User Operations for an ERC-4337 account based on [ZeroDev Kernel V2](https://github.com/zerodevapp/kernel/blob/main/src/Kernel.sol) - a modular contract account framework. It deploys with the [ECDSA validator](https://github.com/zerodevapp/kernel/blob/main/src/validator/ECDSAValidator.sol) by default.

🚧

Only available on Polygon Mumbai

Kernel V2 factories and validators are currently only deployed on Polygon Mumbai.

TypeScript

import { Client, Presets } from "userop";

const kernel = await Presets.Builder.Kernel.init(

signer, // Any object compatible with ethers.Signer

config.rpcUrl,

);

const client = await Client.init(config.rpcUrl);

const res = await client.sendUserOperation(

kernel.execute({to, value, data: "0x"}),

{ onBuild: (op) => console.log("Signed UserOperation:", op) }

);

console.log(`UserOpHash: ${res.userOpHash}`);

console.log("Waiting for transaction...");

const ev = await res.wait();

console.log(`Transaction hash: ${ev?.transactionHash ?? null}`);

## Middleware

Middleware presets are common implementations of [middleware functions](https://docs.stackup.sh/docs/useropjs-builder#middleware-functions) that can be re-used for different builder instances.

### estimateUserOperationGas

A middleware function for sending UserOperations to the eth\_estimateUserOperationGas endpoint in order to estimate reasonable gas limits for preVerificationGas, verificationGasLimit, and callGasLimit.

TypeScript

import { Presets } from "userop";

// provider is an ethers.js JSON-RPC provider.

builder = builder.useMiddleware(Presets.Middleware.estimateUserOperationGas(provider))

### getGasPrice

A middleware function for getting the latest values for maxFeePerGas and maxPriorityFeePerGas.

TypeScript

import { Presets } from "userop";

// provider is an ethers.js JSON-RPC provider.

builder = builder.useMiddleware(Presets.Middleware.getGasPrice(provider))

### verifyingPaymaster

A middleware function for requesting gas sponsorship from a Paymaster service. The middleware assumes that the service implements this proposed [JSON-RPC API for verifying paymasters](https://hackmd.io/@stackup/H1oIvV-qi).

**Use a Paymaster**

Open Recipe

TypeScript

import { Presets } from "userop";

builder = builder.useMiddleware(

Presets.Middleware.verifyingPaymaster(paymasterRpc, paymasterCtx)

)

### EOASignature

A middleware function for signing the User Operation with an EOA private key.

TypeScript

import { ethers } from "ethers";

import { Presets } from "userop";

// signer is an ethers.js Wallet instance.

const signer = new ethers.Wallet(signingKey);

builder = builder.useMiddleware(Presets.Middleware.EOASignature(signer))

Updated 9 days ago

# Constants

A useful collection of reference values related to the ERC-4337 protocol.

[Suggest Edits](https://docs.stackup.sh/edit/useropjs-constants)

While building out your applications, you may require a reference to certain static values that don't change. For example, the EntryPoint contract or various account factories. The Constants module provides a quick way to access these values.

TypeScript

import { Constants } from "userop";

## ERC4337

A collection of values related to the [core ERC-4337 protocol](https://github.com/eth-infinitism/account-abstraction/).

### EntryPoint

The address to the canonical EntryPoint contract.

TypeScript

Constants.ERC4337.EntryPoint

### SimpleAccount factory

The address to the factory contract used to deploy the sample SimpleAccount.sol.

TypeScript

Constants.ERC4337.SimpleAccount.Factory

## Kernel

A collection of values related to the [ZeroDev Kernel](https://github.com/zerodevapp/kernel" \t "_self) contract account framework.

### Factory

The address to the factory used to deploy a Kernel account. This is the factory to use if you need to deploy a Kernel account with a custom [default validator](https://docs.zerodev.app/extend-wallets/build-a-plugin#default-validator).

TypeScript

Constants.Kernel.factory

### ECDSA factory

The address to the factory used to deploy a Kernel account with an [ECDSA validator](https://github.com/zerodevapp/kernel/blob/main/src/validator/ECDSAValidator.sol) as the default. This is equivalent to creating an account with the base factory using the ECDSA validator address.

TypeScript

Constants.Kernel.ECDSAFactory

### ECDSA validator

The address to the [ECDSA validator](https://github.com/zerodevapp/kernel/blob/main/src/validator/ECDSAValidator.sol) contract. This is a plugin that allows User Operations sent to a Kernel account to be validated with an ECDSA key pair.

TypeScript

Constants.Kernel.ECDSAValidator

### Modes

A Kernel account can be executed under different [modes](https://docs.zerodev.app/extend-wallets/overview#validation-phase). These are constant values which can be concatenated to the start of the UserOperation signature field.

TypeScript

Constants.Kernel.Modes.Sudo

Constants.Kernel.Modes.Plugin

Constants.Kernel.Modes.Enable

## Safe

A collection of values related to the [Safe](https://github.com/safe-global/safe-contracts) contract ecosystem.

### MultiSend

The address to the [MultiSend.sol](https://github.com/safe-global/safe-contracts/blob/v1.3.0/contracts/libraries/MultiSend.sol) contract which allows batching of multiple transactions. The addresses here are equivalent to [v1.3.0 of the safe deployments](https://github.com/safe-global/safe-deployments/blob/main/src/assets/v1.3.0/multi_send.json).

TypeScript

// Addresses are not the same for all networks and requires a lookup by the chainID string.

Constants.Safe.MultiSend[chainID]

Updated 3 months ago

**Provider**

A simple wrapper over ethers.js JsonRpcProvider with the ability to re-route bundler methods.

[Suggest Edits](https://docs.stackup.sh/edit/useropjs-provider)

By default userop.js assumes both bundler and node methods share the same RPC url. However, this may not always be the case. This module is a simple wrapper over [ethers.js JsonRpcProvider](https://docs.ethers.org/v5/api/providers/jsonrpc-provider/) with the added ability to override the bundler RPC. This allows all bundler RPC methods to be re-routed to a different endpoint.

**BundlerJsonRpcProvider**

TypeScript

import { BundlerJsonRpcProvider } from "userop";

// provider will have the same interface as a regular JsonRpcProvider.

const provider = new BundlerJsonRpcProvider(rpcUrl).setBundlerRpc(config.overrideBundlerRpc);

Updated 5 months ago