docs

Introduction

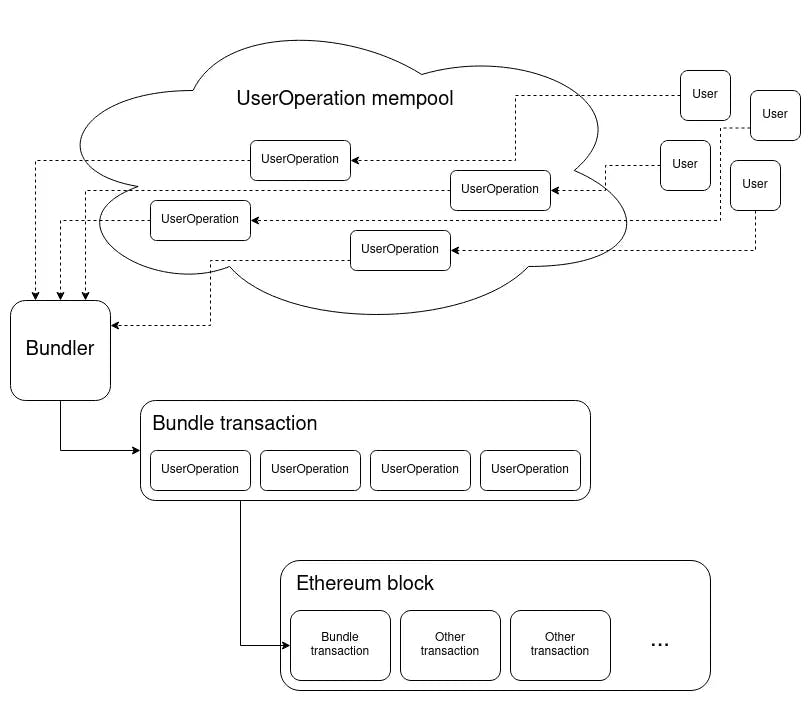
**ERC-4337 Documentation**

**Welcome to the docs! Here you will find guides, references and resources that will help you build with ERC-4337**

**What is ERC-4337?**

[ERC-4337(opens in a new tab)](https://eips.ethereum.org/EIPS/eip-4337) (Account Abstraction via Entry Point Contract specification) is a specification that aims to use an [entry point contract](https://www.erc4337.io/understanding-ERC-4337/entry-point-contract.md) to achieve account abstraction without changing the consensus layer protocol of Ethereum.

Instead of modifying the logic of the consensus layer itself, ERC-4337 replicates the functionality of the transaction mempool in a higher-level system. Users send UserOperation objects that package up the user’s intent along with signatures and other data for verification. Either miners or bundlers using services such as Flashbots can package up a set of UserOperation objects into a single “bundle transaction”, which then gets included into an Ethereum block.

[*Image source(opens in a new tab)*](https://medium.com/infinitism/erc-4337-account-abstraction-without-ethereum-protocol-changes-d75c9d94dc4a)

ERC-4337 also introduces a paymaster mechanism that can enable users to pay gas fees using ERC-20 tokens (e.g. USDC) instead of ETH or to allow a third party to sponsor their gas fees altogether, all in a decentralized fashion.

ERC-4337 is still in draft stage and not finalized yet. However, since ERC-4337 will not change the consensus layer, there are already several implementations available like [eth-infinitsm(opens in a new tab)](https://github.com/eth-infinitism/account-abstraction) and [Stackup(opens in a new tab)](https://github.com/stackup-wallet).

docs

Understanding ERC-4337

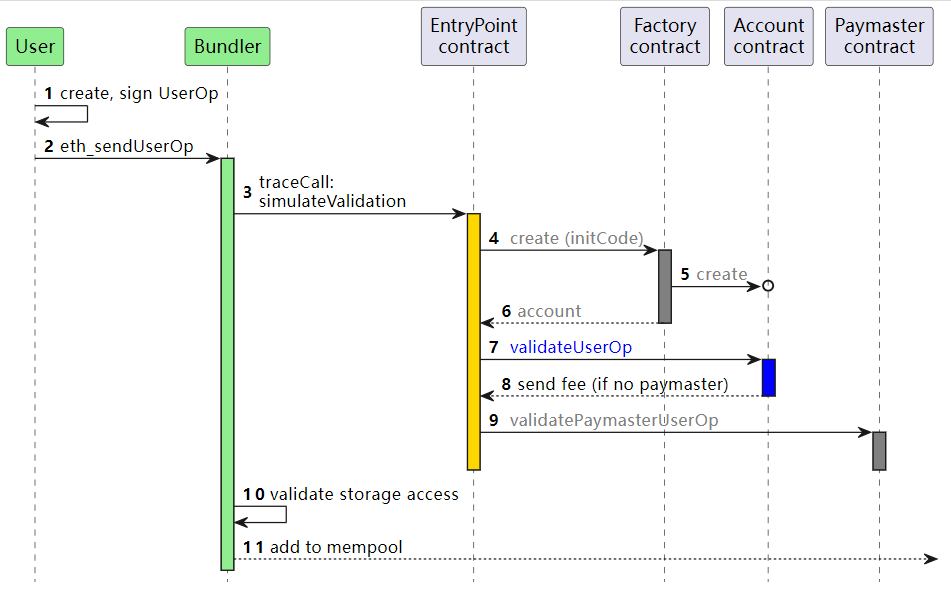
Architecture

**Architecture**

There are several main components to ERC-4337: UserOperation, Bundler, EntryPoint Contract, Account Contract, Account Factory Contract and Paymaster Contract.

* UserOperations are pseudo-transaction objects that are used to execute transactions with contract accounts. These are created by the dapp. Wallets should be able to translate regular transactions into UserOperations so dapps' frontends don't need to change anything to support ERC-4337
* Bundlers are actors that package UserOperations from a mempool and send them to the EntryPoint contract on the blockchain. For more detailed documentation on bundlers head on to [this part](https://www.erc4337.io/docs/understanding-ERC-4337/link-to-relevant-docs-page) of the documentation.
* EntryPoint is a smart contract that handles the verification and execution logic for transactions. Account Contracts are smart contract accounts owned by a user.
* Account Contract is the smart contract wallet of a user. Wallet developers are required to implement at least two custom functions - one to verify signatures, and another to process transactions.
* Factory Contract - When using a wallet for the first time, the initCode field of the UserOperation is used to specify creation of the smart contract wallet. This is used concurrently with the first actual operation of the wallet (in the same UserOperation). Therefore, wallet developers also need to implement the account factory contract (for example: [BLSAccountFactory.sol(opens in a new tab)](https://github.com/eth-infinitism/account-abstraction/blob/develop/contracts/samples/bls/BLSAccountFactory.sol)). Creating new wallets should use the CREATE2 method to ensure the determinacy of generated addresses.
* Paymaster Contracts are optional smart contract accounts that can sponsor gas fees for Account Contracts, or allow their owners to pay for those fees with ERC-20 tokens instead of ETH. For more detailed documentation on paymasters head on to [this part](https://www.erc4337.io/docs/understanding-ERC-4337/link-to-relevant-docs-page) of the documentation.

In practice the process is complex. The typical lifecycle of a transaction looks like this:



docs

Understanding ERC-4337

UserOperation

**UserOperation**

All components of ERC-4337 revolve around a pseudo-transaction object called a UserOperation which is used to execute actions through a smart contract account. This isn't to be mistaken for a regular transaction type.

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| sender | address | The address of the smart contract account |
| nonce | uint256 | Anti-replay protection; also used as the salt for first-time account creation |
| initCode | bytes | Code used to deploy the account if not yet on-chain |
| callData | bytes | Data that's passed to the sender for execution |
| callGasLimit | uint256 | Gas limit for execution phase |
| verificationGasLimit | uint256 | Gas limit for verification phase |
| preVerificationGas | uint256 | Gas to compensate the bundler |
| maxFeePerGas | uint256 | Maximum fee per gas (similar to [EIP-1559(opens in a new tab)](https://eips.ethereum.org/EIPS/eip-1559) max\_fee\_per\_gas) |
| maxPriorityFeePerGas | uint256 | Maximum priority fee per gas (similar to EIP-1559 max\_priority\_fee\_per\_gas) |
| paymasterAndData | bytes | Paymaster Contract address and any extra data required for verification and execution (empty for self-sponsored transaction) |
| signature | bytes | Used to validate a UserOperation along with the nonce during verification |

**UserOperation mempool**

Instead of going to the traditional public mempool that hosts pending transactions for EOAs, UserOperations will instead be sent to the UserOperation mempool: a dedicated, higher-level mempool specifically for UserOperations.

Bundlers listen to the UserOperation mempool and bundle multiple UserOperations together into a "classic" transaction. They first verify the validity of the UserOperations using the relevant EntryPoint methods. The bundler then includes that multi-UserOperation transaction in the next block they propose to the network. To clarify, that transaction isn't sent to the regular mempool. Bundlers are either block builders themselves, or work together with block builders.

docs

Understanding ERC-4337

EntryPoint contract

**EntryPoint Contract**

The EntryPoint contract (which also includes a couple of others, such as StakeManager) is a singleton.  There should only be one implementation because only one instance should exist on each chain.

This contract verifies and executes the bundles of UserOperations sent to it.

The use of a single EntryPoint contract simplifies the logic used by smart contract wallets, making sure the more complicated smart contract functions needed to ensure safety are tried and battle tested. This allows wallets to focus primarily on the core smart account funcitonality (e.g. signature validation rules).

Bundlers/Clients should whitelist their supported EntryPoint contract addresses.

docs

Understanding ERC-4337

Account contract

# Account Contract

With EOAs, the address is consistent across all EVM networks. As long as the user has access to the private key they can access the same address on any network. Ideally, we would like to create the same user experience with contract accounts.

Users should be able to deterministically know their account address and keep it consistent on every EVM network irrespective of whether the account has been deployed or not. This means that they can generate an account and start sending funds to it with the knowledge that they'll be able to control those funds at any time, provided they have the correct verification method.

ERC-4337 does this by using the CREATE2 opcode through a [Singleton Factory(opens in a new tab)](https://eips.ethereum.org/EIPS/eip-2470) - the [Account Factory contract](https://www.erc4337.io/docs/understanding-ERC-4337/account-factory-contract.md). For example, here is how you can calculate a CREATE2 address with ethers.js:

const accountAddress = ethers.utils.getCreate2Address( fromAddress, salt, initCodeHash);

The contract address would be determined by fromAddress, salt, and initCodeHash.

fromAddress The fromAddress is the address of the Singleton Factory. This factory receives the salt and initCode as input and uses CREATE2 to deploy the contract on-chain.

Because the Account Factory contract address is the same on every chain, we can rely on it to also deploy our smart contract code on all networks under the same address too.

salt For an ERC-4337 account, the salt parameter is the first nonce value. This is most likely 0.

initCodeHash The initCode is passed along as a field on the UserOperation is the smart contract code and arguments used for initializing it. It is hashed using keccak256 to derive the initCodeHash.

## Authentication and authorization

## Nonces and replay protection