
ethers-provider-flashbots-bundle

This repository contains the `FlashbotsBundleProvider` ethers.js provider, an additional `Provider` to `ethers.js` to enable high-level access to `eth_sendBundle` and `eth_callBundle` rpc endpoint on mev-relay. **mev-relay is a hosted service; it is not necessary to run mev-relay or mev-geth to proceed with this example.**

Flashbots-enabled relays and miners expose two new jsonrpc endpoints: `eth_sendBundle` and `eth_callBundle`. Since these are non-standard endpoints, ethers.js and other libraries do not natively support these requests (like `getTransactionCount`). In order to interact with these endpoints, you will need access to another full-featured (non-Flashbots) endpoint for nonce-calculation, gas estimation, and transaction status.

One key feature this library provides is **payload signing**, a requirement to submit Flashbot bundles to the `mev-relay` service. This library takes care of the signing process via the `authSigner` passed into the constructor. Read more about relay signatures [here](#)

This library is not a fully functional ethers.js implementation, just a simple provider class, designed to interact with an existing ethers.js v5 installation.

Example

Install ethers.js and the Flashbots ethers bundle provider.

```
1 npm install --save ethers
2 npm install --save @flashbots/ethers-provider-bundle
```

Open up a new TypeScript file (this also works with JavaScript if you prefer)

```
1 import { providers, Wallet } from "ethers";
2 import { FlashbotsBundleProvider } from "@flashbots/ethers-provider-
  bundle";
3
4 // Standard json rpc provider directly from ethers.js (NOT Flashbots)
5 const provider = new providers.JsonRpcProvider({ url: ETHEREUM_RPC_URL
  }, 1)
6
7 // `authSigner` is an Ethereum private key that does NOT store funds
  and is NOT your bot's primary key.
8 // This is an identifying key for signing payloads to establish
  reputation and whitelisting
9 // In production, this should be used across multiple bundles to build
  relationship. In this example, we generate a new wallet each time
10 const authSigner = Wallet.createRandom();
11
```

```
12 // Flashbots provider requires passing in a standard provider
13 const flashbotsProvider = await FlashbotsBundleProvider.create(
14   provider, // a normal ethers.js provider, to perform gas estimations
15   authSigner // ethers.js signer wallet, only for signing request
16   payloads, not transactions
17 )
```

From here, you have a `flashbotsProvider` object setup which can now perform either an `eth_callBundle` (via `simulate()`) or `eth_sendBundle` (via `sendBundle`). Each of these functions act on an array of `Bundle Transactions`

Bundle Transactions

Both `simulate` and `sendBundle` operate on a bundle of strictly-ordered transactions. While the miner requires signed transactions, the provider library will accept a mix of pre-signed transaction and `TransactionRequest` + `Signer` transactions (which it will estimate, nonce-calculate, and sign before sending to the `mev-relay`)

```
1 const wallet = new Wallet(PRIVATE_KEY)
2 const transaction = {
3   to: CONTRACT_ADDRESS,
4   data: CALL_DATA
5 }
6 const transactionBundle = [
7   {
8     signedTransaction: SIGNED_ORACLE_UPDATE_FROM_PENDING_POOL //
9       serialized signed transaction hex
10   },
11   {
12     signer: wallet, // ethers signer
13     transaction: transaction // ethers populated transaction object
14   }
15 ]
```

Block Targeting

The last thing required for `sendBundle()` is block targeting. Every bundle specifically references a single block. If your bundle is valid for multiple blocks (including all blocks until it is mined), `sendBundle()` must be called for every block, ideally on one of the blocks immediately prior. This gives you a chance to re-evaluate the opportunity you are capturing and re-sign your transactions with a higher nonce, if necessary.

The block should always be a *future* block, never the current one.

```
1 const targetBlockNumber = (await provider.getBlockNumber()) + 1
```

Gas Prices and EIP-1559

Before EIP-1559 was activated, the most common way for searchers to submit transactions is with `gasPrice=0`, with an on-chain payment to `block.coinbase` conditional on the transaction's success. All transactions must pay `baseFee` now, an attribute of a block. For an example of how to ensure you are using this `baseFee`, see `demo.ts` in this repository.

```
1 const block = await provider.getBlock(blockNumber)
2 const maxBaseFeeInFutureBlock = FlashbotsBundleProvider.
  getMaxBaseFeeInFutureBlock(block.baseFeePerGas, BLOCKS_IN_THE_FUTURE
  )
3 const eip1559Transaction = {
4   to: wallet.address,
5   type: 2,
6   maxFeePerGas: PRIORITY_FEE.add(maxBaseFeeInFutureBlock),
7   maxPriorityFeePerGas: PRIORITY_FEE,
8   gasLimit: 21000,
9   data: '0x',
10  chainId: CHAIN_ID
11 }
```

`FlashbotsBundleProvider.getMaxBaseFeeInFutureBlock` calculates the maximum `baseFee` that is possible `BLOCKS_IN_THE_FUTURE` blocks, given maximum expansion on each block. You won't pay this fee, so long as it is specified as `maxFeePerGas`, you will only pay the block's `baseFee`.

Additionally if you have the `baseFeePerGas`, `gasUsed`, and `gasLimit` from the current block and are only targeting one block in the future you can get the exact base fee for the next block using `FlashbotsBundleProvider.getBaseFeeInNextBlock`. This method implements the math based on the EIP1559 definition

Simulate and Send

Now that we have:

1. Flashbots Provider `flashbotsProvider`
2. Bundle of transactions `transactionBundle`
3. Block Number `targetBlockNumber`

We can run simulations and submit directly to miners, via the `mev-relay`.

Simulate:

```
1  const signedTransactions = await flashbotsProvider.signBundle(
    transactionBundle)
2  const simulation = await flashbotsProvider.simulate(
    signedTransactions, targetBlockNumber)
3  console.log(JSON.stringify(simulation, null, 2))
```

Send:

```
1  const flashbotsTransactionResponse = await flashbotsProvider.sendBundle
    (
2    transactionBundle,
3    targetBlockNumber,
4  )
```

FlashbotsTransactionResponse

After calling `sendBundle`, this provider will return a Promise of an object with helper functions related to the bundle you submitted.

These functions return metadata available at transaction submission time, as well as the following functions which can wait, track, and simulate the bundle's behavior.

- `bundleTransactions()` - An array of transaction descriptions sent to the relay, including hash, nonce, and the raw transaction.
- `receipts()` - Returns promise of an array of transaction receipts corresponding to the transaction hashes that were relayed as part of the bundle. Will not wait for block to be mined; could return incomplete information
- `wait()` - Returns a promise which will wait for target block number to be reached *OR* one of the transactions to become invalid due to nonce-issues (including, but not limited to, one of the transactions from your bundle being included too early). Returns the wait resolution as a status enum
- `simulate()` - Returns a promise of the transaction simulation, once the proper block height has been reached. Use this function to troubleshoot failing bundles and verify miner profitability

Optional `eth_sendBundle` arguments

Beyond target block number, an object can be passed in with optional attributes:

```
1  {
```

```
2   minTimestamp, // optional minimum timestamp at which this bundle is
    valid (inclusive)
3   maxTimestamp, // optional maximum timestamp at which this bundle is
    valid (inclusive)
4   revertingTxHashes: [tx1, tx2] // optional list of transaction hashes
    allowed to revert. Without specifying here, any revert invalidates
    the entire bundle.
5 }
```

minTimestamp / maxTimestamp

While each bundle targets only a single block, you can add a filter for validity based on the block's timestamp. This does *not* allow for targeting any block number based on a timestamp or instruct miners on what timestamp to use, it merely serves as a secondary filter.

If your bundle is not valid before a certain time or includes an expiring opportunity, setting these values allows the miner to skip bundle processing earlier in the phase.

Additionally, you could target several blocks in the future, but with a strict maxTimestamp, to ensure your bundle is considered for inclusion up to a specific time, regardless of how quickly blocks are mined in that timeframe.

Reverting Transaction Hashes

Transaction bundles will not be considered for inclusion if they include *any* transactions that revert or fail. While this is normally desirable, there are some advanced use-cases where a searcher might WANT to bring a failing transaction to the chain. This is normally desirable for nonce management. Consider:

Transaction Nonce #1 = Failed (unrelated) token transfer Transaction Nonce #2 = DEX trade

If a searcher wants to bring #2 to the chain, #1 must be included first, and its failure is not related to the desired transaction #2. This is especially common during high gas times.

Optional parameter `revertingTxHashes` allows a searcher to specify an array of transactions that can (but are not required to) revert.

Paying for your bundle

In addition to paying for a bundle with gas price, bundles can also conditionally pay a miner via: `block.coinbase.transfer(_minerReward)` or `block.coinbase.call{value: _minerReward}("");`

(assuming `_minerReward` is a solidity `uint256` with the wei-value to be transferred directly to the miner)

The entire value of the bundle is added up at the end, so not every transaction needs to have a gas price or `block.coinbase` payment, so long as at least one does, and pays enough to support the gas used in non-paying transactions.

Note: Gas-fees will ONLY benefit your bundle if the transaction is not already present in the mempool. When including a pending transaction in your bundle, it is similar to that transaction having a gas price of 0; other transactions in your bundle will need to pay more for the gas it uses.

Bundle and User Statistics

The Flashbots relay can also return statistics about you as a user (identified solely by your signing address) and any bundle previously submitted.

- `getUserStats()` returns aggregate metrics about past performance, including if your signing key is currently eligible for the “high priority” queue. Read more about searcher reputation [here](#)
- `getBundleStats(bundleHash, targetBlockNumber)` returns data specific to a single bundle submission, including detailed timestamps for the various phases a bundle goes before reaching miners. You can get the `bundleHash` from the simulation:

```
1 const simulation = await flashbotsProvider.simulate(  
    signedTransactions, targetBlockNumber)  
2 console.log(simulation.bundleHash)
```

Investigating Losses

When your bundle fails to land in the specified block, there are many reasons why this could have occurred. For a list of reasons, check out [Flashbots Docs : Why didn't my transaction get included?](#). To aid in troubleshooting, this library offers a method that will simulate a bundle in multiple positions to identify the competing bundle that landed on chain (if there was one) and calculate the relevant pricing.

For usage instructions, check out the [demo-research.ts](#).

How to run demo.ts

Included is a simple demo of how to construct the `FlashbotsProvider` with auth signer authentication and submit a [non-functional] bundle. This will not yield any mev, but serves as a sample initialization

to help integrate into your own functional searcher.

Sending a Private Transaction

To send a *single* transaction without having to send it as a bundle, use the `sendPrivateTransaction` function. This method allows us to process transactions faster and more efficiently than regular bundles. When you send a transaction using this method, we will try to send it to miners over the next 25 blocks (up to, but not past the target block number).

```
1  const tx = {
2    from: wallet.address,
3    to: wallet.address,
4    value: "0x42",
5    gasPrice: BigNumber.from(99).mul(1e9), // 99 gwei
6    gasLimit: BigNumber.from(21000),
7  }
8  const privateTx = {
9    transaction: tx,
10   signer: wallet,
11 }
12
13 const res = await flashbotsProvider.sendPrivateTransaction(privateTx)
```

Optionally, you can set the following parameters to fine-tune your submission:

```
1  // highest block number your tx can be included in
2  const maxBlockNumber = (await provider.getBlockNumber()) + 10;
3  // timestamp for simulations
4  const minTimestamp = 1645753192;
5
6  const res = await flashbotsProvider.sendPrivateTransaction(
7    privateTx,
8    {maxBlockNumber, minTimestamp}
9  )
```

Flashbots on Goerli

To test Flashbots before going to mainnet, you can use the Goerli Flashbots relay, which works in conjunction with a Flashbots-enabled Goerli validator. Flashbots on Goerli requires two simple changes:

1. Ensure your genericProvider passed in to the FlashbotsBundleProvider constructor is connected to Goerli (gas estimates and nonce requests need to correspond to the correct chain):

```
1 import { providers } from 'ethers'
2 const provider = providers.getDefaultProvider('goerli')
```

2. Set the relay endpoint to <https://relay-goerli.flashbots.net/>

```
1 const flashbotsProvider = await FlashbotsBundleProvider.create(
2   provider,
3   authSigner,
4   'https://relay-goerli.flashbots.net/',
5   'goerli')
```