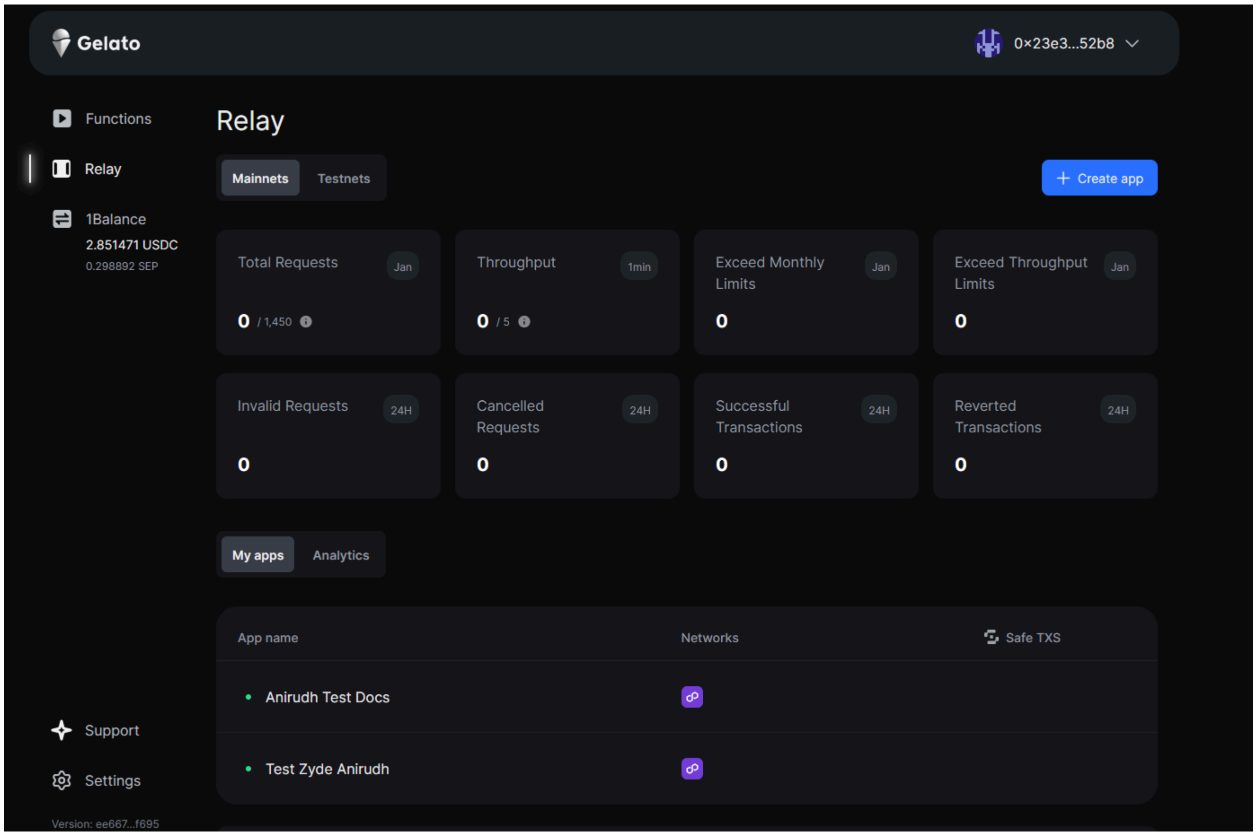
Gelato Relay

Introduction

Using Gelato Relay, we relay your user's transactions on-chain, enabling secure gasless transactions for an ultra-smooth UX for your app. This allows for a variety of new web3 experiences, as the user can now pay by only signing a message, or their transaction costs can be sponsored by the developer. As long as the gas costs are covered in one of the multiple payment methods that Gelato supports, we handle the rest reliably, quickly and securely.



Total Requests (Jan)	Throughput (1min)	Exceed Monthly Limits (Jan)	Exceed Throughput Limits (Jan)
0 / 1,450	0 / 5	0	0
Invalid Requests (24H)	Cancelled Requests (24H)	Successful Transactions (24H)	Reverted Transactions (24H)
0	0	0	0

App name	Networks	Safe TXS
Anirudh Test Docs		
Test Zyde Anirudh		

Prerequisites

- "node": ">=14.0.0"
- Basic JavaScript knowledge.
- ethers knowledge

Getting started

1: Installation

Install the Gelato Relay SDK
yarn add @gelatonetwork/relay-sdk

2: Choose the Method

At this point, you will need to answer the following questions, which will determine the method to use when calling the Gelato Relay.

- Do you require user authentication? When the use-case requires to authenticate the original user, you will need to implement the ERC2771 method where the user will sign the payload, and the original user will be decoded on-chain from the callData replacing msg.sender through _msgSender(), please see additional info here.
- What is the funding strategy? When relaying a transaction, the Gelato Nodes are paying the gas fees. There are two different ways of paying the fees back to Gelato. Either creating a 1Balance account and deposit USDC on polygon that will pay for all of the transactions on all EVM chains Gelato is deployed; or transferring back to gelato the fees while the transaction is executing, we call these methods syncFee, more info can be found here, in this latter case, the target contract would need to inherit the "Gelato Relay Context" contracts, so the methods to query and transfer the fee to Gelato are available.

If you require user authentication and you want to pay the transactions with a 1Balance account, the method to use is the sponsoredCallERC2771.
If you require user authentication and you want every transaction to pay for itself, transferring by execution the fees to Gelato, the method to use is the callWithSyncFeeERC2771.
If you don't require user authentication and you want to pay the transactions with a 1Balance account, the method to use is the sponsoredCall.
If you don't require user authentication and you want every transaction to pay for itself, transferring the fees by execution to Gelato, the method to use is the callWithSyncFee.

3: Implementation

We will require three simple steps to implement Gelato Relay. Here, we are going to showcase the three steps required to implement the method sponsoredCallERC2771, which is the most used one.

Step 1: Inherit Context Contract

Depending on the method, you must inherit different contracts as they will provide other methods. In this case, we will have to inherit the ERC2771Context. The ERC2771Context provide us with the methods _msgSender() and _msgData() that will allow us to recover the original user sending the transaction.
import {

```
  ERC2771Context
} from "@gelatonetwork/relay-context/contracts/vendor/ERC2771Context.sol";

contract CounterERC2771 is ERC2771Context {

  // ERC2771Context: setting the immutable trustedForwarder variable
  constructor(address trustedForwarder) ERC2771Context(trustedForwarder) {}

  function incrementContext() external {

      // Incrementing the counter mapped to the _msgSender!
      contextCounter[_msgSender()]++;

      // Emitting an event for testing purposes
      emit IncrementContextCounter(_msgSender());
  }
}
```

Step 2: Import the relay SDK

In your frontend/backend, you would need to import and instantiate the relay class.

```
import { GelatoRelay, SponsoredCallERC2771Request } from "@gelatonetwork/relay-sdk";
const relay = new GelatoRelay(API_KEY);
```

Step 3: Send the payload to Gelato

This is an example using Gelato's CounterERC2771.sol, which is deployed on these networks.

```
// Set up on-chain variables, such as target address
const counter = "0x00172f67db60E5fA346e599cdE675f0ca213b47b";
const abi = ["function incrementContext()"];
const provider = new ethers.BrowserProvider(window.ethereum);
const signer = provider.getSigner();
const user = signer.getAddress();

// Generate the target payload
const contract = new ethers.Contract(counter, abi, signer);
const { data } = await contract.incrementContext.populateTransaction();

// Populate a relay request
const request: CallWithERC2771Request = {
 chainId: (await provider.getNetwork()).chainId,
 target: counter;
 data: data;
 user: user;
};
```

```
// Without a specific API key, the relay request will fail!
// Go to https://relay.gelato.network to get a testnet API key with 1Balance.
// Send a relay request using Gelato Relay!
const relayResponse = await relay.sponsoredCallERC2771(request, provider, apiKey);
```

Tracking your Request

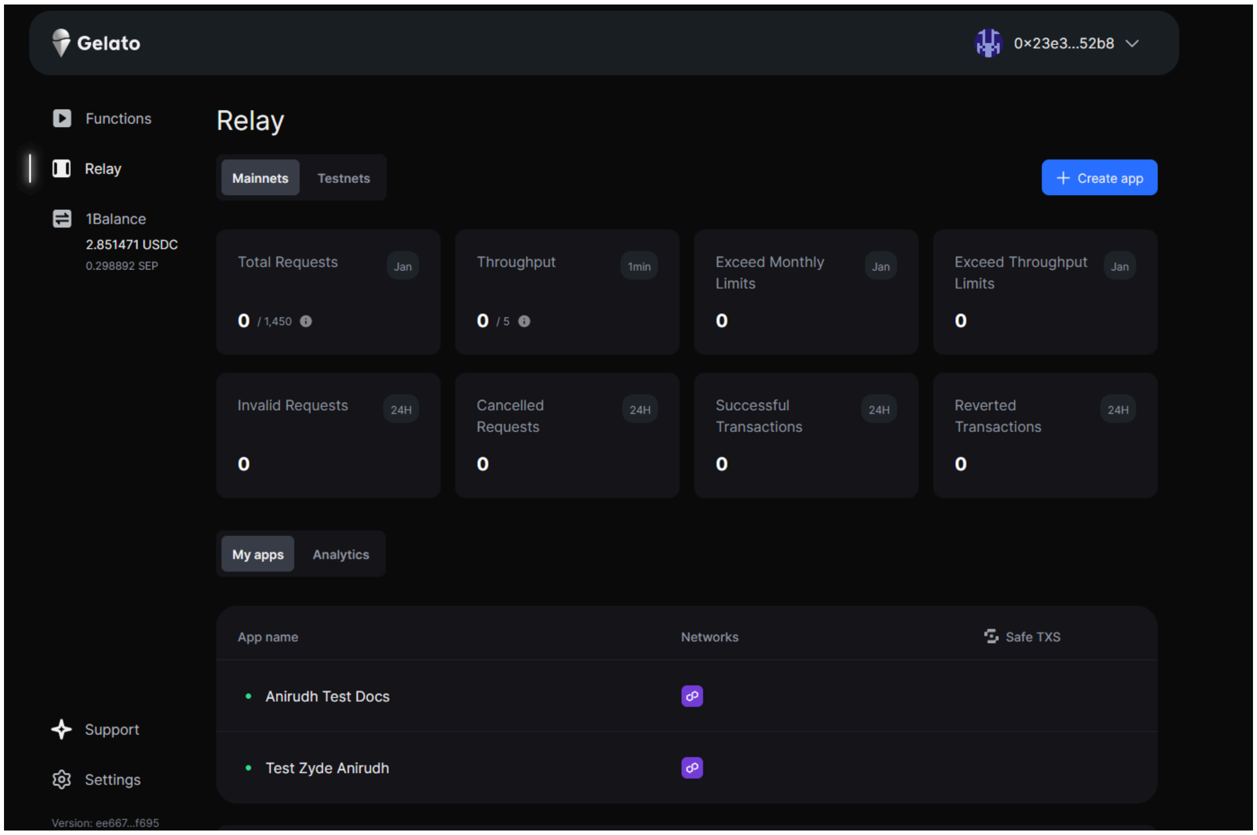
When submitting your Gelato Relay requests, you'll receive a taskId in response. This taskId allows you to track the status of your request in two primary ways:

1. WebSocket Subscriptions: This is the recommended and most efficient method. By subscribing via WebSocket, the Gelato backend will automatically push updates for all your tasks to your Relay SDK client. To start receiving these updates, you must register a callback function, which will be triggered every time one of your tasks gets updated. Detailed implementation can be found here.
2. Polling for Updates: Alternatively, you can periodically query the Gelato task status API for updates. If you're using the Gelato Relay SDK, the getTaskStatus method makes this easy. Detailed implementation can be found here.

to Relay

Introduction

Using Gelato Relay, we relay your user's transactions on-chain, enabling secure gasless transactions for an ultra-smooth UX for your app. This allows for a variety of new web3 experiences, as the user can now pay by only signing a message, or their transaction costs can be sponsored by the developer. As long as the gas costs are covered in one of the multiple payment methods that Gelato supports, we handle the rest reliably, quickly and securely.



Relay

Total Requests (Jan)	Throughput (1min)	Exceed Monthly Limits (Jan)	Exceed Throughput Limits (Jan)
0 / 1,450	0 / 5	0	0

Invalid Requests (24H)	Cancelled Requests (24H)	Successful Transactions (24H)	Reverted Transactions (24H)
0	0	0	0

App name	Networks	Safe TXS
Anirudh Test Docs		
Test Zyde Anirudh		

Prerequisites

- "node": ">=14.0.0"
- Basic JavaScript knowledge.
- ethers knowledge

Getting started

1: Installation

Install the Gelato Relay SDK
yarn add @gelatonetwork/relay-sdk

2: Choose the Method

At this point, you will need to answer the following questions, which will determine the method to use when calling the Gelato Relay.

- Do you require user authentication? When the use-case requires to authenticate the original user, you will need to implement the ERC2771 method where the user will sign the payload, and the original user will be decoded on-chain from the callData replacing msg.sender through _msgSender(), please see additional info here.

- What is the funding strategy? When relaying a transaction, the Gelato Nodes are paying the gas fees. There are two different ways of paying the fees back to Gelato. Either creating a 1Balance account and deposit USDC on polygon that will pay for all of the transactions on all EVM chains Gelato is deployed; or transferring back to gelato the fees while the transaction is executing, we call these methods syncFee, more info can be found here, in this latter case, the target contract would need to inherit the "Gelato Relay Context" contracts, so the methods to query and transfer the fee to Gelato are available.

If you require user authentication and you want to pay the transactions with a 1Balance account, the method to use is the sponsoredCallERC2771.
If you require user authentication and you want every transaction to pay for itself, transferring by execution the fees to Gelato, the method to use is the callWithSyncFeeERC2771.
If you don't require user authentication and you want to pay the transactions with a 1Balance account, the method to use is the sponsoredCall.
If you don't require user authentication and you want every transaction to pay for itself, transferring the fees by execution to Gelato, the method to use is the callWithSyncFee.

3: Implementation

We will require three simple steps to implement Gelato Relay. Here, we are going to showcase the three steps required to implement the method sponsoredCallERC2771, which is the most used one.

Step 1: Inherit Context Contract

Depending on the method, you must inherit different contracts as they will provide other methods. In this case, we will have to inherit the ERC2771Context. The ERC2771Context provide us with the methods _msgSender() and _msgData() that will allow us to recover the original user sending the transaction.

```
import {
  ERC2771Context
} from "@gelatonetwork/relay-context/contracts/vendor/ERC2771Context.sol";

contract CounterERC2771 is ERC2771Context {

  // ERC2771Context: setting the immutable trustedForwarder variable
  constructor(address trustedForwarder) ERC2771Context(trustedForwarder) {}

  function incrementContext() external {

     // Incrementing the counter mapped to the _msgSender!
     contextCounter[_msgSender()]++;

     // Emitting an event for testing purposes
     emit IncrementContextCounter(_msgSender());
  }
}
```

Step 2: Import the relay SDK

In your frontend/backend, you would need to import and instantiate the relay class.

```
import { GelatoRelay, SponsoredCallERC2771Request } from "@gelatonetwork/relay-sdk";
const relay = new GelatoRelay(API_KEY);
```

Step 3: Send the payload to Gelato

This is an example using Gelato's CounterERC2771.sol, which is deployed on these networks.

```
// Set up on-chain variables, such as target address
const counter = "0x00172f67db60E5fA346e599cdE675f0ca213b47b";
const abi = ["function incrementContext()"];
const provider = new ethers.BrowserProvider(window.ethereum);
const signer = provider.getSigner();
const user = signer.getAddress();

// Generate the target payload
const contract = new ethers.Contract(counter, abi, signer);
const { data } = await contract.incrementContext.populateTransaction();

// Populate a relay request
const request: CallWithERC2771Request = {
 chainId: (await provider.getNetwork()).chainId,
 target: counter;
 data: data;
 user: user;
};

// Without a specific API key, the relay request will fail!
// Go to https://relay.gelato.network to get a testnet API key with 1Balance.
// Send a relay request using Gelato Relay!
const relayResponse = await relay.sponsoredCallERC2771(request, provider, apiKey);
```

Tracking your Request

When submitting your Gelato Relay requests, you'll receive a taskId in response. This taskId allows you to track the status of your request in two primary ways:

1. WebSocket Subscriptions: This is the recommended and most efficient method. By subscribing via WebSocket, the Gelato backend will automatically push updates for all your tasks to your Relay SDK client. To start receiving these updates, you must register a callback function, which will be triggered every time one of your tasks gets updated. Detailed implementation can be found here.

2. Polling for Updates: Alternatively, you can periodically query the Gelato task status API for updates. If you're using the Gelato Relay SDK, the getTaskStatus method makes this easy. Detailed implementation can be found here.