TestHelper (Foundry)

Overview

The TestHelper contract is designed to facilitate the testing of Omnichain Applications (OApps) developed using LayerZero V2, specifically within the Foundry test framework.

This contract provides a suite of functions to simulate cross-chain transactions and validate the behavior of OApps locally in your Foundry unit tests. The full code to this contract can be found inMonorepo .

info For developers new to Foundry or those looking to deepen their understanding of its capabilities in Solidity testing, the following resources can be helpful:

1. Getting Started with Foundry
2. : To begin your journey with Foundry, theFoundry Book
3. offers a detailed guide on installation, setup, and basic usage. It's an excellent starting point for understanding the fundamentals of Foundry and its role in smart contract development.
4. Solidity Testing with Foundry
5. : For a deeper dive into testing Solidity contracts using Foundry, theFoundry GitHub
6. provides comprehensive documentation, examples, and community contributions. This resource is invaluable for learning best practices and advanced techniques in contract testing.

Installation

To install the TestHelper package in Foundry, run the following command:

forge install LayerZero-Labs/devtools And then add the following remapping to yourremappings.txt file:

@layerzerolabs/test-devtools-evm-foundry/=lib/devtools/packages/test-devtools-evm-foundry

NPM

If you have a hybrid Foundry and NPM setup you can use following command to install the tool:

npm

install @layerzerolabs/test-devtools-evm-foundry

Sample Implementation

// SPDX-License-Identifier: UNLICENSED

pragma

solidity

^ 0.8.22 ;

import

{ Packet }

from

"@layerzerolabs/lz-evm-protocol-v2/contracts/interfaces/ISendLib.sol" ; import

{ OptionsBuilder }

from

"@layerzerolabs/lz-evm-oapp-v2/contracts/oapp/libs/OptionsBuilder.sol" ; import

{ MessagingFee }

from

"@layerzerolabs/lz-evm-oapp-v2/contracts/oapp/OApp.sol" ; import

{ MessagingReceipt }

```
from

"@layerzerolabs/lz-evm-oapp-v2/contracts/oapp/OAppSender.sol" ; // The unique path location of your OApp import

{ MyOApp }

from

"../../contracts/MyOApp.sol" ; import

{ TestHelperOz5 }

from

"@layerzerolabs/test-devtools-evm-foundry/contracts/TestHelperOz5.sol" ;

import

"forge-std/console.sol" ;

/// @notice Unit test for MyOApp using the TestHelper. /// @dev Inherits from TestHelper to utilize its setup and utility functions. contract

MyOAppTest

is TestHelperOz5 { using

OptionsBuilder

for

bytes ;

// Declaration of mock endpoint IDs. uint16 aEid =

1 ; uint16 bEid =

2 ;

// Declaration of mock contracts. MyOApp aMyOApp ;

// OApp A MyOApp bMyOApp ;

// OApp B

/// @notice Calls setUp from TestHelper and initializes contract instances for testing. function

setUp ( )

public virtual override { super . setUp ( ) ;

// Setup function to initialize 2 Mock Endpoints with Mock MessageLib. setUpEndpoints ( 2 , LibraryType . UltraLightNode ) ;

// Initializes 2 MyOApps; one on chain A, one on chain B. address [ ]

memory sender =

setupOApps ( type ( MyOApp ) . creationCode ,

1 ,

2 ) ; aMyOApp =

MyOApp ( payable ( sender [ 0 ] ) ) ; bMyOApp =

MyOApp ( payable ( sender [ 1 ] ) ) ; }

/// @notice Tests the send and multi-compose functionality of MyOApp. /// @dev Simulates message passing from A -> B and checks for data integrity. function

test_send ( )

public
```

{ // Setup variable for data values before calling send(). string

memory dataBefore = aMyOApp . data ( ) ;

// Generates 1 lzReceive execution option via the OptionsBuilder library. // STEP 0: Estimating message gas fees via the quote function. bytes

memory options = OptionsBuilder . newOptions ( ) . addExecutorLzReceiveOption (150000 ,

0) ; MessagingFee memory fee = aMyOApp . quote (bEid ,

"test message" , options ,

false) ;

// STEP 1: Sending a message via the _lzSend() method. MessagingReceipt memory receipt = aMyOApp . send { value : fee . nativeFee } (bEid ,

"test message" , options) ;

// Asserting that the receiving OApps have NOT had data manipulated. assertEq (bMyOApp . data ( ) , dataBefore ,

"shouldn't be changed until lzReceive packet is verified") ;

// STEP 2 & 3: Deliver packet to bMyOApp manually. verifyPackets (bEid ,

addressToBytes32 (address (bMyOApp))) ;

// Asserting that the data variable has updated in the receiving OApp. assertEq (bMyOApp . data ( ) ,

"test message" ,

"lzReceive data assertion failure") ; } }

Key Functions

The TestHelper contract, integral to testing Omnichain Applications (OApps) with Foundry, is equipped with a variety of functions. While some of these functions are geared towards internal mechanics of the contract and may not be directly utilized by developers, others are crucial for effectively testing cross-chain functionalities in OApps. Below, we delve into those key functions that are particularly important for external use in testing scenarios.

Initializers

setUp()

ThesetUp() function initializes the test environment. This function can be overridden in derived contracts to set up specific test conditions.

function

setUp ( )

public virtual { } It is called at the beginning of each test to prepare the test environment.

setUpEndpoints

ThesetUpEndpoints function is designed to initialize a specified number of mock endpoints. This function allows for the creation of multiple endpoints, each potentially representing different blockchains or networks, and configures them with a chosen library type (e.g., Ultra Light Node, Simple Message Lib).

/* * *@dev setup the endpoints * @param _endpointNum num of endpoints*/ function

setUpEndpoints (uint8 _endpointNum , LibraryType _libraryType)

public

{ // Full code implementation here: https://github.com/LayerZero-Labs/devtools/blob/30e92c638876c5aa435ae0a5c856f15cfed2a706/packages/test-devtools-evm-foundry/contracts/TestHelperOz5.sol } * _endpointNum * : The number of endpoints to set up. * _libraryType * : The type of library to use (Ultra Light Node or Simple Message Lib).* UltraLightNode * * : A messaging library featuring Mock Decentralized Verifier Networks (DVNs) and Executors for complex cross-chain message verification and execution. * * SimpleMessageLib * * : A streamlined library for basic cross-chain message passing, lacking additional functionalities like Mock DVNs and Executors found in more complex libraries.

setupOApps

setupOApps automates the deployment and wiring of OApp instances. It enables developers to simulate multiple instances of their OApps on different mock chains, providing a comprehensive testing landscape.

/* * *@dev setup UAs, only if the UA has*endpoint *address as the unique parameter*/ function

setupOApps (bytes

memory _oappCreationCode , uint8 _startEid , uint8 _oappNum)

public

returns

(address [ ]

memory oapps)

{ // Full code implementation here: https://github.com/LayerZero-Labs/monorepo/blob/main/packages/layerzero-v2/evm/oapp/test/TestHelper.sol } * bytes _oappCreationCode * : Represents the bytecode (creation code) of the Omnichain Application (OApp) to be deployed. It is essentially the compiled code of the OApp contract. * uint8 _startEid * : Specifies the starting mock Endpoint ID (Eid) for the OApps being set up. In the context of LayerZero and cross-chain applications, an Endpoint ID uniquely identifies a specific blockchain or network endpoint. * uint8 _oappNum * : Indicates the number of OApp instances to deploy.

This function returns an array of mock OApp addresses that are deployed and wired together (viasetPeer) in the test environment.

Sample Implementation

The example implementation below demonstrates how to utilize initializers in TestHelper likesetUpEndpoints andsetupOApps to create a testing environment.

function

setUp ( )

public virtual override { super . setUp ( ) ;

// Setup function to initialize 2 Mock Endpoints with Mock MessageLib. setUpEndpoints (2 , LibraryType . UltraLightNode) ;

// Initializes 2 Sample OApps; one on chain A, one on chain B. address [ ]

memory sender =

setupOApps (type (SampleOApp) . creationCode ,

1 ,

2) ; aSampleOApp =

SampleOApp (payable (sender [0])) ; bSampleOApp =

SampleOApp (payable (sender [1])) ; }

Simulate Transactions

verifyPackets

verifyPackets simulates the receipt and processing of packets on the destination chain.

/* * @dev dst UA receive/execute packets * @dev will NOT work calling this directly with composer IF the composed payload is different from the lzReceive msg payload */ function

verifyPackets (uint32 _dstEid ,

bytes32 _dstAddress ,

uint256 _packetAmount ,

address _composer)

public

{ // Full code implementation here: https://github.com/LayerZero-Labs/monorepo/blob/main/packages/layerzero-v2/evm/oapp/test/TestHelper.sol } * _dstEid * : The destination endpoint Id * _dstAddress * : The destination address (asbytes32 *) * _packetAmount * : Specifies the number of packets to verify. Used to limit the number of packets that will be processed during the simulation. This can be useful for testing scenarios where you need to control the volume of packets being verified in a single function call. * _composer * : The address of the composer. Used when the verification process involves composed messaging. * Overloads * : * 1. verifyPackets(uint32 _dstEid, bytes32 _dstAddress) * 2. verifyPackets(uint32 _dstEid, address _dstAddress)

Sample Implementation

/// @notice Tests the send and receive functionality. /// @dev Simulates message passing from A -> B and checks for data integrity. function

test_send_and_compose ( )

public

{

// Setup variables for data values before calling send(). string

memory dataBefore = bSampleOApp . data ( ) ;

// STEP 0: Estimating message gas fees via the quote function. bytes

memory _payload = abi . encode ("test message") ;

// Generates 1 lzReceive execution option via the OptionsBuilder library. bytes

memory options = OptionsBuilder . newOptions ( ) . addExecutorLzReceiveOption (150000 ,

0) ;

MessagingFee memory fee = aSampleApp . quote (bEid ,

"test message" , options ,

false) ;

// STEP 1: Sending a message via the _lzSend() method. MessagingReceipt memory receipt = aSampleOApp . send { value : fee . nativeFee } (bEid ,

"test message" , options) ;

// Asserting that the receiving OApps have NOT had data manipulated. assertEq (bSampleOApp . data ( ) , dataBefore ,

"shouldn't be changed until lzReceive packet is verified") ;

// STEP 2 & 3: Deliver packet to bSampleOApp. verifyPackets (bEid ,

addressToBytes32 (address (bSampleOApp))) ;

// Asserting that the data variable has updated in both receiving OApps. assertEq (bSampleOApp . data ( ) ,

"test message" ,

"lzReceive data assertion failure") ; }

Helper Functions

In addition to its main testing functions, theTestHelper.sol contract includes helper functions that enhance its capability to handle various scenarios in the testing of Omnichain Applications (OApps). These functions are critical for ensuring a thorough and versatile testing environment, particularly when dealing with the complexities of cross-chain communication.

addressToBytes32

addressToBytes32 converts an Ethereum address to abytes32 format. This is useful in scenarios where addresses need to be handled in a fixed-size byte format, which is common in many blockchain protocols and LayerZero operations.

function

addressToBytes32 (address _addr)

internal

pure

returns

(bytes32)

{ return

bytes32 (uint256 (uint160 (_addr))) ; } * _addr * : The Ethereum address to convert. * Returns * :bytes32 * representation of the address.

getNextInflightPacket

getNextInflightPacket Retrieves the next packet in line for delivery to a specified destination. This is crucial for testing the order and integrity of packet delivery in cross-chain communications. Use it to inspect and verify the sequence and content of packets destined for a particular chain or address.

function

getNextInflightPacket (uint16 _dstEid ,

bytes32 _dstAddress)

public

view

returns

(bytes

memory packetBytes)

{ DoubleEndedQueue . Bytes32Deque storage queue = packetsQueue [_dstEid] [_dstAddress] ; if

(queue . length ( )

0)

{ bytes32 guid = queue . back ( ) ; packetBytes = packets [guid] ; } } * _dstEid * : The destination Endpoint ID. * _dstAddress * : The destination address (asbytes32 *). * Returns * : The next packet (as bytes) scheduled for delivery.

hasPendingPackets

hasPendingPackets checks if there are any pending packets for a given destination. Useful for verifying if packets are scheduled for delivery.

function

hasPendingPackets (uint16 _dstEid ,

bytes32 _dstAddress)

public

view

returns

(bool flag)

{ DoubleEndedQueue . Bytes32Deque storage queue = packetsQueue [_dstEid] [_dstAddress] ; return queue . length ( )

0 ; } * _dstEid * : Destination endpoint ID * _dstAddress * : Destination Address (asbytes32 *). * Returns * : Boolean indicating the presence of pending packets.

assertGuid

assertGuid validates that a given packet has the correct Global Unique Identifier (GUID) which is defined in the messageLib.

function

assertGuid (bytes

calldata packetBytes ,

bytes32 guid)

external

pure

{ bytes32 packetGuid = packetBytes . guid ( ) ; require (packetGuid == guid ,

"guid not match") ; } * _packetBytes * : The packet content. * guid * : The Global Unique Identifier of the specific packet being checked. * Usage * :1. Testing Packet Integrity: This function is instrumental in testing scenarios to ensure that packets being sent and received in a cross-chain setup are correctly identified and match their intended GUIDs. * 2. Debugging and Validation: It's a useful tool for debugging and validating that the packet creation, modification, or routing processes are functioning correctly, as any discrepancy in GUIDs would be a clear indicator of an issue. Edit this page

Previous Deterministic Deployment Next LayerZero Scan