

Transfer Tokens with Data - Defensive Example

This tutorial extends the [programmable token transfers example](#) . It uses Chainlink CCIP to transfer tokens and arbitrary data between smart contracts on different blockchains, and focuses on defensive coding in the receiver contract. In the event of a specified error during the CCIP message reception, the contract locks the tokens. Locking the tokens allows the owner to recover and redirect them as needed. Defensive coding is crucial as it enables the recovery of locked tokens and ensures the protection of your users' assets.

Transferring tokens

This tutorial uses the term "transferring tokens" even though the tokens are not technically transferred. Instead, they are locked or burned on the source chain and then unlocked or minted on the destination chain. Read the [Token Pools](#) section to understand the various mechanisms that are used to transfer value across chains.

Before you begin

1. You should understand how to write, compile, deploy, and fund a smart contract. If you need to brush up on the basics, read this [tutorial](#) , which will guide you through using the [Solidity programming language](#) , interacting with the [MetaMask wallet](#) and working within the [Remix Development Environment](#) .
2. Your account must have some ETH and LINK tokens onEthereum Sepoliaand MATIC tokens onPolygon Mumbai. Learn how to [Acquire testnet LINK](#) .
3. Check the [Supported Networks page](#) to confirm that the tokens you will transfer are supported for your lane. In this example, you will transfer tokens fromEthereum SepoliatoPolygon Mumbaiso check the list of supported tokens [here](#) .
4. Learn how to [acquire CCIP test tokens](#) . Following this guide, you should have CCIP-BnM tokens, and CCIP-BnM should appear in the list of your tokens in MetaMask.
5. Learn how to [fund your contract](#) . This guide shows how to fund your contract in LINK, but you can use the same guide for funding your contract with any ERC20 tokens as long as they appear in the list of tokens in MetaMask.
6. Follow the previous tutorial [Transfer Tokens with Data](#) to learn how to make programmable token transfers using CCIP.

Tutorial

In this guide, you'll initiate a transaction from a smart contract onEthereum Sepolia, sending astringtext and CCIP-BnM tokens to another smart contract onPolygon Mumbaiusing CCIP. However, a deliberate failure in the processing logic will occur upon reaching the receiver contract. This tutorial will demonstrate a graceful error-handling approach, allowing the contract owner to recover the locked tokens.

Correctly estimate your gas limit

It is crucial to thoroughly test all scenarios to accurately estimate the required gas limit, including for failure scenarios. Be aware that the gas used to execute the error-handling logic for failure scenarios may be higher than that for successful scenarios.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;
import {IRouterClient} from "@chainlink/contracts-ccip/src/v0.8/ccip/interfaces/IRouterClient.sol";
import {OwnerIsCreator} from "@chainlink/contracts-ccip/src/v0.8/shared/access/OwnerIsCreator.sol";
import {Client} from "@chainlink/contracts-ccip/src/v0.8/ccip/libraries/Client.sol";
import {CCIPReceiver} from "@chainlink/contracts-ccip/src/v0.8/ccip/applications/CCIPReceiver.sol";
import {IERC20} from "@chainlink/contracts-ccip/src/v0.8/vendor/openzeppelin-solidity/v4.8.3/contracts/token/ERC20/IERC20.sol";
import {SafeERC20} from "@chainlink/contracts-ccip/src/v0.8/vendor/openzeppelin-solidity/v4.8.3/contracts/token/ERC20/utils/SafeERC20.sol";
import {EnumerableMap} from "@chainlink/contracts-ccip/src/v0.8/vendor/openzeppelin-solidity/v4.8.3/contracts/utils/structs/EnumerableMap.sol";

* THIS IS AN EXAMPLE CONTRACT THAT USES HARDCODED VALUES FOR CLARITY.
* THIS IS AN EXAMPLE CONTRACT THAT USES UN-AUDITED CODE.
* DO NOT USE THIS CODE IN PRODUCTION.

/** @title A simple messenger contract for transferring/receiving tokens and data across chains.
    @dev - This example shows how to recover tokens in case of
    revert
    contract ProgrammableDefensiveTokenTransfers is CCIPReceiver, OwnerIsCreator {
        using EnumerableMap for EnumerableMap.Bytes32ToUintMap;
        using SafeERC20 for IERC20;

        // Custom errors to provide more descriptive revert messages.
        error NotEnoughBalance(uint256 currentBalance, uint256 calculatedFees); // Used to make sure contract has enough balance to cover the fees.
        error NothingToWithdraw(); // Used when trying to withdraw Ether but there's nothing to withdraw.
        error FailedToWithdrawEth(address owner, address target, uint256 value); // Used when the withdrawal of Ether fails.
        error DestinationChainNotAllowed(uint64 destinationChainSelector); // Used when the destination chain has not been allowedlist by the contract owner.
        error SourceChainNotAllowed(uint64 sourceChainSelector); // Used when the source chain has not been allowedlist by the contract owner.
        error InvalidReceiverAddress(); // Used when the receiver address is 0.
        error OnlySelf(); // Used when a function is called outside of the contract itself.
        error ErrorCase(); // Used when simulating a revert during message processing.
        error MessageNotFailed(bytes32 msgageld); // Example error code, could have many different error codes.
        enum ErrorCode {RESOLVED} // RESOLVED is first so that the default value is resolved.RESOLVED. // Could have any number of error codes
        here.
        FAILED struct FailedMessage {bytes32 msgageld; ErrorCode errorCode; } // Event emitted when a message is sent to another chain.
        event MessageSent(bytes32 indexed msgageld, The unique ID of the CCIP message.
        uint64 indexed destinationChainSelector, // The chain selector of the destination chain.
        address receiver, // The address of the receiver on the destination chain.
        string text, // The text being sent.
        address token, // The token address that was transferred.
        uint256 tokenAmount, // The token amount that was transferred.
        address feeToken, // the token address used to pay CCIP fees.
        uint256 fees, // The fees paid for sending the message.
        Event emitted when a message is received from another chain.
        event MessageReceived(bytes32 indexed msgageld, // The unique ID of the CCIP message.
        uint64 indexed sourceChainSelector, // The chain selector of the source chain.
        address sender, // The address of the sender from the source chain.
        string text, // The text that was received.
        address token, // The token address that was transferred.
        uint256 tokenAmount, // The token amount that was transferred.
        event MessageFailed(bytes32 indexed msgageld, bytes reason);
        event MessageRecovered(bytes32 indexed msgageld, bytes32 private _lastReceivedMessageageld; // Store the last received messageageld.
        address private _lastReceivedTokenAddress; // Store the last received token address.
        uint256 private _lastReceivedTokenAmount; // Store the last received amount.
        string private _lastReceivedText; // Store the last received text.
        Mapping to keep track of allowedlist destination chains.
        mapping(uint64 => bool) public allowedlistDestinationChains; // Mapping to keep track of allowedlist source chains.
        mapping(uint64 => bool) public allowedlistSourceChains; // Mapping to keep track of allowedlist senders.
        mapping(address => bool) public allowedlistSenders;
        IERC20 private _linkToken; // The message contents of failed messages are stored here.
        mapping(bytes32 msgageld => Client.Any2EVMMessage contents) public _messageContents; // Contains failed messages and their state.
        EnumerableMap.Bytes32ToUintMap internal _failedMessages; // This is used to simulate a revert in the processMessage function.
        bool internal _simRevert = false; // @notice Constructor initializes the contract with the router address.
        @param _router The address of the router contract.
        @param _link The address of the link contract.
        constructor(address _router, address _link) CCIPReceiver(_router) {
            _linkToken = IERC20(_link);
        }
        @dev Modifier that checks if the chain with the given destinationChainSelector is allowedlist.
        @param _destinationChainSelector The selector of the destination chain.
        modifier onlyAllowedlistDestinationChain(uint64 _destinationChainSelector) {
            if(!_allowedlistDestinationChains[_destinationChainSelector]) revert DestinationChainNotAllowed(_destinationChainSelector);
        }
        @dev Modifier that checks if the chain with the given sourceChainSelector is allowedlist and if the sender is allowedlist.
        @param _sourceChainSelector The selector of the destination chain.
        @param _sender The address of the sender.
        modifier onlyAllowedlist(uint64 _sourceChainSelector, address _sender) {
            if(!_allowedlistSourceChains[_sourceChainSelector]) revert SourceChainNotAllowed(_sourceChainSelector);
            if(!_allowedlistSenders[_sender]) revert SenderNotAllowed(_sender);
        }
        @dev Modifier that checks the receiver address is not 0.
        @param _receiver The receiver address.
        modifier validateReceiver(address _receiver) {
            if(_receiver == address(0)) revert InvalidReceiverAddress();
        }
        @dev Modifier to allow only the contract itself to execute a function.
        Throws an exception if called by any account other than the contract itself.
        modifier onlySelf() {
            if(msg.sender != address(this)) revert OnlySelf();
        }
        @dev Updates the allowlist status of a destination chain for transactions.
        @notice This function can only be called by the owner.
        @param _destinationChainSelector The selector of the destination chain to be updated.
        @param _allowed The allowlist status to be set for the destination
        chain.
        functional allowlistDestinationChain(uint64 _destinationChainSelector, bool _allowed) external onlyOwner {
            _allowedlistDestinationChains[_destinationChainSelector] = _allowed;
        }
        @dev Updates the allowlist status of a source chain.
        @notice This function can only be called by the owner.
        @param _sourceChainSelector The selector of the source chain to be updated.
        @param _allowed The allowlist status to be set for the source
        chain.
        functional allowlistSourceChain(uint64 _sourceChainSelector, bool _allowed) external onlyOwner {
            _allowedlistSourceChains[_sourceChainSelector] = _allowed;
        }
        @dev Updates the allowlist status of a sender for transactions.
        @notice This function can only be called by the owner.
        @param _sender The address of the sender to be updated.
        @param _allowed The allowlist status to be set for the sender.
        functional allowlistSender(address _sender, bool _allowed) external onlyOwner {
            _allowedlistSenders[_sender] = _allowed;
        }
        @notice Sends data and transfer tokens to receiver on the destination chain.
        @notice Pay for fees in LINK.
        @dev Assumes your contract has sufficient LINK to pay for CCIP fees.
        @param _destinationChainSelector The identifier (aka selector) for the destination blockchain.
        @param _receiver The address of the recipient on the destination blockchain.
        @param _text The string data to be sent.
        @param _token token address.
        @param _amount token amount.
        @return messageageld The ID of the CCIP message that was sent.
        function sendMessagePayLINK(uint64 _destinationChainSelector, address _receiver, string calldata _text, address _token, uint256 _amount) external onlyOwner onlyAllowedlistDestinationChain {
            // Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain message.
            // address(linkToken) means fees are paid in LINK.
            Client.EVM2AnyMessage memory evm2AnyMessage = buildCCIPMessage(_receiver, _text, _token, _amount, address(_linkToken));
            // Initialize a router client instance to interact with cross-chain router.
            IRouterClient router = IRouterClient(this.getRouter());
            // Get the fee required to send the CCIP message.
            uint256 fees = router.getFee(_destinationChainSelector, evm2AnyMessage);
            if(fees > s._linkToken.balanceOf(address(this))) revert NotEnoughBalance(s._linkToken.balanceOf(address(this)));
            // approve the Router to transfer LINK tokens on contract's behalf. It will spend the fees in LINKs.
            _linkToken.approve(address(router), fees);
            // approve the Router to spend tokens on contract's behalf. It will spend the amount of the given token.
            IERC20(_token).approve(address(router), _amount);
            // Send the message through the router and store the returned message ID.
            messageageld = router.cipSend(_destinationChainSelector, evm2AnyMessage);
            // Emit an event with message details.
            emit MessageSent(messageageld, _destinationChainSelector, _receiver, _text, _token, _amount, address(_linkToken), fees);
            // Return the message ID.
            return messageageld;
        }
        @notice Sends data and transfer tokens to receiver on the destination chain.
        @notice Pay for fees in native gas.
        @dev Assumes your contract has sufficient native gas like ETH on Ethereum or MATIC on Polygon.
        @param _destinationChainSelector The identifier (aka selector) for the destination blockchain.
        @param _receiver The address of the recipient on the destination blockchain.
        @param _text The string data to be sent.
        @param _token token address.
        @param _amount token amount.
        @return messageageld The ID of the CCIP message that was sent.
        function sendMessagePayNative(uint64 _destinationChainSelector, address _receiver, string calldata _text, address _token, uint256 _amount) external onlyOwner onlyAllowedlistDestinationChain {
            // Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain message.
            // address(0) means fees are paid in native gas.
            Client.EVM2AnyMessage memory evm2AnyMessage = buildCCIPMessage(_receiver, _text, _token, _amount, address(0));
            // Initialize a router client instance to interact with cross-chain router.
            IRouterClient router = IRouterClient(this.getRouter());
            // Get the fee required to send the CCIP message.
            uint256 fees = router.getFee(_destinationChainSelector, evm2AnyMessage);
            if(fees > address(this).balance) revert NotEnoughBalance(address(this).balance, fees);
            // approve the Router to spend tokens on contract's behalf. It will spend the amount of the given token.
            IERC20(_token).approve(address(router), _amount);
            // Send the message through the router and store the returned message ID.
            messageageld = router.cipSend(value: fees)(_destinationChainSelector, evm2AnyMessage);
            // Emit an event with message details.
            emit MessageSent(messageageld, _destinationChainSelector, _receiver, _text, _token, _amount, address(0), fees);
            // Return the message ID.
            return messageageld;
        }
        * @notice Returns the details of the last CCIP received message.
        * @dev This function retrieves the ID, text, token address, and token amount of the last received CCIP message.
        * @return messageageld The ID of the last received CCIP message.
        * @return text The text of the last received CCIP message.
        * @return tokenAddress The address of the token in the last CCIP received message.
        * @return tokenAmount The amount
```

of the token in the last CCIP received message. `/functiongetLastReceivedMessageDetails()` (public view returns (bytes32 messageId, string memory text, address tokenAddress, uint256 tokenAmount))

```
(return(s_lastReceivedMessageId,s_lastReceivedText,s_lastReceivedTokenAddress,s_lastReceivedTokenAmount));
```

* `@notice` Retrieves a paginated list of failed messages. * `@dev` This function returns a subset of failed messages defined by `offset` and `limit` parameters. It ensures that the pagination parameters are within the bounds of the available data set. * `@param offset` The index of the first failed message to return, enabling pagination by skipping a specified number of messages from the start of the dataset. * `@param limit` The maximum number of failed messages to return, restricting the size of the returned array. * `@return failedMessages` An array of `FailedMessage` struct, each containing a `messageId` and an `errorCode` (`RESOLVED` or `FAILED`), representing the requested subset of failed messages. The length of the returned array is determined by the `limit` and the total number of failed messages.

```
/functiongetFailedMessages(uint256offset,uint256limit)externalviewreturns(FailedMessage[]memory){uint256length=s_failedMessages.length();// Calculate the actual number of items to return (can't exceed total length or requested limit)uint256returnLength=(offset+limit-length)?length-offset:limit;FailedMessage[]memoryfailedMessages=newFailedMessage();// Adjust loop to respect pagination (start at offset, end at offset + limit or total length)for(uint256i=0;i<returnLength;i++){(bytes32messageId,uint256errorCode)=s_failedMessages.at(offset+i);failedMessages[i]=FailedMessage(messageId,ErrorCode(errorCode));}returnfailedMessages;}// @notice The endpoint for the CCIP router to call. This function should // never revert, all errors should be handled internally in this contract.// @param any2EvmMessage The message to process.// @dev Extremely important to ensure only router calls this.functionccipReceive(Client.Any2EVMMessagecalldataany2EvmMessage)externaloverrideonlyRouteronlyAllowlisted(any2EvmMessage.sourceChainSelector,abi.decode(any2EvmMessage.sender,(address))){// Make sure the source chain and sender are allowlisted(// solhint-disable no-empty-blocks /try{this.processMessage(any2EvmMessage)}// Intentionally empty in this example; no action needed if processMessage succeeds}catch(bytesmemoryerr){}// Could set different error codes based on the caught error. Each could be// handled differently.s_failedMessages.set(any2EvmMessage.messageId,uint256(ErrorCode.FAILED));s_messageContents[any2EvmMessage.messageId]=any2EvmMessage;// Don't revert so CCIP doesn't revert. Emit event instead.// The message can be retried later without having to do manual execution of CCIP.emitMessageFailed(any2EvmMessage.messageId,err);return;}// @notice Serves as the entry point for this contract to process incoming messages.// @param any2EvmMessage Received CCIP message.// @dev Transfers specified token amounts to the owner of this contract. This function// must be external because of the try/catch for error handling.// It uses the onlySelf: can only be called from the contract.functionprocessMessage(Client.Any2EVMMessagecalldataany2EvmMessage)externalonlySelfonlyAllowlisted(any2EvmMessage.sourceChainSelector,abi.decode(any2EvmMessage.sender,(address))){// Make sure the source chain and sender are allowlisted(// Simulate a revert for testing purposesif(s_simRevert)revertErrorCase();_ccipReceive(any2EvmMessage);// process the message - may revert as well)// @notice Allows the owner to retry a failed message in order to unblock the associated tokens.// @param messageId The unique identifier of the failed message.// @param tokenReceiver The address to which the tokens will be sent.// @dev This function is only callable by the contract owner. It changes the status of the message// from 'failed' to 'resolved' to prevent reentry and multiple retries of the same message.functionretryFailedMessage(bytes32messageId,addresstokenReceiver)externalonlyOwner{// Check if the message has failed; if not, revert the transaction.if(s_failedMessages.get(messageId)!=uint256(ErrorCode.FAILED))revertMessageNotFailed(messageId);// Set the error code to RESOLVED to disallow reentry and multiple retries of the same failed message.s_failedMessages.set(messageId,uint256(ErrorCode.RESOLVED));// Retrieve the content of the failed message.Client.Any2EVMMessagememorymessage=s_messageContents[messageId];// This example expects one token to have been sent, but you can handle multiple tokens.// Transfer the associated tokens to the specified receiver as an escape hatch.IERC20(message.destTokenAmounts[0].token).safeTransfer(tokenReceiver,message.destTokenAmounts[0].amount);// Emit an event indicating that the message has been recovered.emitMessageRecovered(messageId);}// @notice Allows the owner to toggle simulation of reversion for testing purposes.// @param simRevert If true, simulates a revert condition; if false, disables the simulation.// @dev This function is only callable by the contract owner.functionsetSimRevert(boolsimRevert)externalonlyOwner(s_simRevert=simRevert);function_ccipReceive(Client.Any2EVMMessagememoryany2EvmMessage)internaloverride{s_lastReceivedMessageId=s_lastReceivedText=abi.decode(any2EvmMessage.data,(string));// abi-decoding of the sent text// Expect one token to be transferred at once, but you can transfer several tokens.s_lastReceivedTokenAddress=any2EvmMessage.destTokenAmounts[0].token;s_lastReceivedTokenAmount=any2EvmMessage.destTokenAmounts[0].amount;emitMessageReceived(any2EvmM fetch the source chain identifier (aka selector)abi.decode(any2EvmMessage.sender,(address));// abi-decoding of the sender address,abi.decode(any2EvmMessage.data,(string)),any2EvmMessage.destTokenAmounts[0].token,any2EvmMessage.destTokenAmounts[0].amount);}// @notice Construct a CCIP message.// @dev This function will create an EVM2AnyMessage struct with all the necessary information for programmable tokens transfer.// @param _receiver The address of the receiver.// @param _text The string data to be sent.// @param _token The token to be transferred.// @param _amount The amount of the token to be transferred.// @param _feeTokenAddress The address of the token used for fees. Set address(0) for native gas.// @return Client.EVM2AnyMessage Returns an EVM2AnyMessage struct which contains information for sending a CCIP message.function_buildCCIPMessage(address_receiver,stringcalldata_text,address_token,uint256_amount,address_feeTokenAddress)privatepurereturns(Client.EVM2AnyMessagememory){// Set the token amounts.Client.EVMTokenAmount[]memorytokenAmounts=newClient.EVMTokenAmount;Client.EVMTokenAmount(token:_token,amount:_amount);tokenCreate an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain message.Client.EVM2AnyMessagememoryevm2AnyMessage=Client.EVM2AnyMessage((receiver:abi.encode(_receiver),// ABI-encoded receiver addressdata:abi.encode(_text),// ABI-encoded stringtokenAmounts:tokenAmounts,// The amount and type of token being transferredextraArgs:Client._argsToBytes(),// Additional arguments, setting gas limitClient.EVMExtraArgsV1((gasLimit:400_000)),// Set the feeToken to a feeTokenAddress, indicating specific asset will be used for feesfeeToken:_feeTokenAddress));returnevm2AnyMessage;}// @notice Fallback function to allow the contract to receive Ether.// @dev This function has no function body, making it a default function for receiving Ether.// It is automatically called when Ether is sent to the contract without any data.receive()externalpayable{}/} // @notice Allows the contract owner to withdraw the entire balance of Ether from the contract.// @dev This function reverts if there are no funds to withdraw or if the transfer fails.// It should only be callable by the owner of the contract.// @param _beneficiary The address to which the Ether should be sent.functionwithdraw(address_beneficiary)publiconlyOwner{// Retrieve the balance of this contractuint256amount=address(this).balance;// Revert if there is nothing to withdrawif(amount==0)revertNothingToWithdraw();// Attempt to send the funds, capturing the success status and discarding any return data(boolsent,)=_beneficiary.call{value:amount}("");// Revert if the send failed, with information about the attempted transferif(!sent)revertFailedToWithdrawEth(msg.sender,_beneficiary,amount);}// @notice Allows the owner of the contract to withdraw all tokens of a specific ERC20 token.// @dev This function reverts with a 'NothingToWithdraw' error if there are no tokens to withdraw.// @param _beneficiary The address to which the tokens will be sent.// @param _token The contract address of the ERC20 token to be withdrawn.functionwithdrawToken(address_beneficiary,address_token)publiconlyOwner{// Retrieve the balance of this contractuint256amount=IERC20(_token).balanceOf(address(this));// Revert if there is nothing to withdrawif(amount==0)revertNothingToWithdraw();IERC20(_token).safeTransfer(_beneficiary,amount);}
```

[Open in Remix](#) [What is Remix?](#)

Deploy your contracts

To use this contract:

1. [Open the contract in Remix](#).
2. Compile your contract.
3. Deploy, fund your sender contract onEthereum Sepoliaand enable sending messages toPolygon Mumbai:
4. Open MetaMask and select the networkEthereum Sepolia.
5. In Remix IDE, click onDeploy & Run Transactionsand selectInjected Provider - MetaMaskfrom the environment list. Remix will then interact with your MetaMask wallet to communicate withEthereum Sepolia.
6. Fill in your blockchain's router and LINK contract addresses. The router address can be found on the[supported networks page](#) and the LINK contract address on the[LINK token contracts page](#). ForEthereum Sepolia, the router address is0x0BF3dE8c5D3e8A2B34D2BEb17ABfCeBa363A59and the LINK contract address is0x779877A7B0D9E8603169DdbD7836e478b4624789.
7. Click thetransactbutton. After you confirm the transaction, the contract address appears on theDeployed Contractslist. Note your contract address.
8. Open MetaMask and fund your contract with CCIP-BnM tokens. You can transfer0.002CCIP-BnMto your contract.
9. Enable your contract to send CCIP messages toPolygon Mumbai:1. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onEthereum Sepolia.
10. Call theallowlistDestinationChainwith12532609583862916517as the destination chain selector, andtrueas allowed. Each chain selector is found on the[supported networks page](#).
11. Deploy your receiver contract onPolygon Mumbaiand enable receiving messages from your sender contract:
12. Open MetaMask and select the networkPolygon Mumbai.
13. In Remix IDE, underDeploy & Run Transactions, make sure the environment is stillInjected Provider - MetaMask.
14. Fill in your blockchain's router and LINK contract addresses. The router address can be found on the[supported networks page](#) and the LINK contract address on the[LINK token contracts page](#). ForPolygon Mumbai, the router address is0x1035CabC275068e0f4b745A29CEDf38E13aF41b1and the LINK contract address is0x326C977E6efc84E512bB9C30f76E30c160eD06FB.
15. Click thetransactbutton. After you confirm the transaction, the contract address appears on theDeployed Contractslist. Note your contract address.
16. Enable your contract to receive CCIP messages fromEthereum Sepolia:1. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
17. Call theallowlistSourceChainwith16015286601757825753as the source chain selector, andtrueas allowed. Each chain selector is found on the[supported networks page](#).
18. Enable your contract to receive CCIP messages from the contract that you deployed onEthereum Sepolia:1. In Remix IDE, underDeploy & Run Transactions, open the list of transactions of your smart contract deployed onPolygon Mumbai.
19. Call theallowlistSenderwith the contract address of the contract that you deployed onEthereum Sepolia, andtrueas allowed.
20. Call thesetSimRevertfunction, passingtrueas a parameter, then wait for the transaction to confirm. Settings_simRevertto true simulates a failure when processing the received message. Read the[explanation](#) section for more details.

At this point, you have onesendercontract onEthereum Sepoliaand onereceivercontract onPolygon Mumbai. As security measures, you enabled the sender contract to send CCIP messages toPolygon Mumbaiand the receiver contract to receive CCIP messages from the sender andEthereum Sepolia. The receiver contract cannot process the message, and therefore, instead of throwing an exception, it will lock the received tokens, enabling the owner to recover them.

Note: Another security measure enforces that only the router can call the `_ccipReceive` function. Read the [explanation](#) section for more details.

Recover the locked tokens

You will transfer0.001 CCIP-BnMand a text. The CCIP fees for using CCIP will be paid in LINK.

1. Open MetaMask and connect toEthereum Sepolia. Fund your contract with LINK tokens. You can transfer0.5LINKto your contract. In this example, LINK is used to pay the CCIP fees.
2. Send a string data with tokens fromEthereum Sepolia:
3. Open MetaMask and select the networkEthereum Sepolia.

4. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Ethereum Sepolia.
5. Fill in the arguments of the `sendMessagePayLINK` function:

ArgumentValue and Description_destinationChainSelector12532609583862916517CCIP Chain identifier of the destination blockchain (Polygon Mumbai in this example). You can find each chain selector on the [supported networks page](#) . receiverYour receiver contract address at Polygon Mumbai. The destination contract address . textHelloWorld!Anystring_token0xFd57b4ddBf88a4e071F4e34C487b99af2Fe82a05TheCCIP-BnMcontract address at the source chain (Ethereum Sepolia in this example). You can find all the addresses for each supported blockchain on the [supported networks page](#) . amount1000000000000000The token amount (0.001 CCIP-BnM). 4. Click on `transact` and confirm the transaction on MetaMask. 5. After the transaction is successful, record the transaction hash. Here is an [example](#) of a transaction on Ethereum Sepolia.

note

During gas price spikes, your transaction might fail, requiring more than 0.5 LINK to proceed. If your transaction fails, fund your contract with more LINK tokens and try again. 3. Open the [CCIP explorer](#) and search your cross-chain transaction using the transaction hash. 4. The CCIP transaction is completed once the status is marked as "Success". In this example, the CCIP message ID is `0x2fb721d506350a75439b16f7dab551cf518c6dcc473b4e9271218f8f470533f8`. 5. Check the receiver contract on the destination chain:

1. Open MetaMask and select the network Polygon Mumbai.
2. In Remix IDE, under Deploy & Run Transactions, open the list of transactions of your smart contract deployed on Polygon Mumbai.
3. Call the `getFailedMessages` function with an `offset` of 0 and an `limit` of 1 to retrieve the first failed message.
4. Notice the returned values are: `0x2fb721d506350a75439b16f7dab551cf518c6dcc473b4e9271218f8f470533f8` (the message ID) and `1` (the error code indicating failure).
5. To recover the locked tokens, call the `retryFailedMessage` function:

ArgumentDescriptionmessageIdThe unique identifier of the failed message.tokenReceiverThe address to which the tokens will be sent. 7. After confirming the transaction, you can open it in a block explorer. Notice that the locked funds were transferred to the `tokenReceiver` address. 8. Call again the `getFailedMessages` function with an `offset` of 0 and an `limit` of 1 to retrieve the first failed message. Notice that the error code is now 0, indicating that the message was resolved.

Note: These example contracts are designed to work bi-directionally. As an exercise, you can use them to transfer tokens with data from Ethereum Sepolia to Polygon Mumbai and from Polygon Mumbai back to Ethereum Sepolia.

Explanation

Integrate Chainlink CCIP into your project

npm yarn foundryIf you use NPM , install the [@chainlink/contracts-ccip NPM package](#) and set it to the v1.4.0 release:

npm install @chainlink/If you use Yarn , install the [@chainlink/contracts-ccip NPM package](#) and set it to the v1.4.0 release:

yarn add @chainlink/If you use Foundry , install the v1.4.0 release:

forge install smartcontractkit/ccip@b06a3c2e6cb9892ec6f76a015624413ffa1a122

The smart contract featured in this tutorial is designed to interact with CCIP to transfer and receive tokens and data. The contract code is similar to the [Transfer Tokens with Data](#) tutorial. Hence, you can refer to its [code explanation](#) . We will only explain the main differences.

Sending messages

The `sendMessagePayLINK` function is similar to the `sendMessagePayLINK` function in the [Transfer Tokens with Data](#) tutorial. The main difference is the increased gas limit to account for the additional gas required to process the error-handling logic.

Receiving and processing messages

Upon receiving a message on the destination blockchain, the `ccipReceive` function is called by the CCIP router. This function serves as the entry point to the contract for processing incoming CCIP messages, enforcing crucial security checks through the `onlyRouter`, and `onlyAllowlistedmodifiers`.

Here's the step-by-step breakdown of the process:

1. Entrance through `ccipReceive`:
2. The `ccipReceive` function is invoked with an `Any2EVMMessage` struct containing the message to be processed.
3. Security checks ensure the call is from the authorized router, an allowlisted source chain, and an allowlisted sender.
4. Processing Message:
5. `ccipReceive` calls the `processMessage` function, which is external to leverage Solidity's try/catch error handling mechanism. Note: The `onlySelf` modifier ensures that only the contract can call this function.
6. Inside `processMessage`, a check is performed for a simulated revert condition using the `_simRevert` state variable. This simulation is toggled by the `setSimRevert` function, callable only by the contract owner.
7. If `_simRevert` is false, `processMessage` calls the `_ccipReceive` function for further message processing.
8. Message Processing in `_ccipReceive`:
9. `_ccipReceive` extracts and stores various information from the message, such as the `messageId`, `decodedSenderAddress`, token amounts, and data.
10. It then emits a `MessageReceived` event, signaling the successful processing of the message.
11. Error Handling:
12. If an error occurs during the processing (or a simulated revert is triggered), the catch block within `ccipReceive` is executed.
13. The `messageId` of the failed message is added to `_failedMessages`, and the message content is stored in `_messageContents`.
14. A `MessageFailed` event is emitted, which allows for later identification and reprocessing of failed messages.

Reprocessing of failed messages

The `retryFailedMessage` function provides a mechanism to recover assets if a CCIP message processing fails. It's specifically designed to handle scenarios where message data issues prevent entire processing yet allow for token recovery:

1. Initiation:
2. Only the contract owner can call this function, providing the `messageId` of the failed message and the `tokenReceiverAddress` for token recovery.
3. Validation:
4. It checks if the message has failed using `_failedMessages.get(messageId)`. If not, it reverts the transaction.
5. Status Update:
6. The error code for the message is updated to `RESOLVED` to prevent reentry and multiple retries.
7. Token Recovery:
8. Retrieves the failed message content using `_messageContents[messageId]`.
9. Transfers the locked tokens associated with the failed message to the specified `tokenReceiver` as an escape hatch without processing the entire message again.
10. Event Emission:
11. An event `MessageRecovered` is emitted to signal the successful recovery of the tokens.

This function showcases a graceful asset recovery solution, protecting user values even when message processing encounters issues.