

Send Arbitrary Data

In this tutorial, you will use Chainlink CCIP to send data between smart contracts on different blockchains. First, you will pay for the CCIP fees on the source blockchain using LINK. Then, you will use the same contract to pay CCIP fees in native gas tokens. For example, you would use ETH on Ethereum or MATIC on Polygon.

Node Operator Rewards

CCIP rewards the oracle node and Risk Management node operators in LINK.

Before you begin

- You should understand how to write, compile, deploy, and fund a smart contract. If you need to brush up on the basics, read the [tutorial](#), which will guide you through using the [Solidity programming language](#), interacting with the [MetaMask wallet](#) and working within the [Remix Development Environment](#).
- Your account must have some ETH tokens on Ethereum Sepolia and MATIC tokens on Polygon Mumbai.
- Learn how to [Acquire testnet LINK](#) and [Fund your contract with LINK](#).

Tutorial

In this tutorial, you will send `asStringText` between smart contracts on Ethereum Sepolia and Polygon Mumbai using CCIP. First, you will pay [CCIP fees in LINK](#), then you will pay [CCIP fees in native gas](#).

```
SPDX-License-Identifier: MIT;pragma solidity 0.8.19;import {RouterClient} from '@chainlink/contracts-ccip/src/v0.8/ccip/interfaces/RouterClient.sol';import {OwnersCreator} from '@chainlink/contracts-ccip/src/v0.8/shared/access/OwnersCreator.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';/* * 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 sending/receiving string data across chains.contract Messenger is CCIPReceiver, OwnersCreator {using SafeERC20 for IERC20; // Custom errors to provide more descriptive revert messages.errorNotEnoughBalance(uint256 currentBalance, uint256 calculatedFees) { // Used to make sure contract has enough balance.errorNothingToWithdraw() { // Used when trying to withdraw Ether but there's nothing to withdraw.errorFailedToWithdrawEth(address owner, address target, uint256 value) { // Used when the withdrawal of Ether fails.errorDestinationChainNotAllowed(uint64 destinationChainSelector) { // Used when the destination chain has not been allowed by the contract owner.errorSourceChainNotAllowed(uint64 sourceChainSelector) { // Used when the source chain has not been allowed by the contract owner.errorSenderNotAllowed(address sender) { // Used when the sender has not been allowed by the contract owner.errorInvalidReceiverAddress() { // Used when the receiver address is 0. // Event emitted when a message is sent to another chain.event MessageSent(bytes32 indexed msgid, string text) { // 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 feeToken, // the token address used to pay CCIP fees, uint256 fees // The fees paid for sending the CCIP message. // } // Event emitted when a message is received from another chain.event MessageReceived(bytes32 indexed msgid, string text) { // 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. // } bytes32 private _lastReceivedMessageId; // Store the last received messageId, string private _lastReceivedText; // Store the last received text. // Mapping to keep track of allowed destination chains, mapping(uint64 => bool) public allowedDestinationChains; // Mapping to keep track of allowed source chains, mapping(uint64 => bool) public allowedSourceChains; // Mapping to keep track of allowed senders, mapping(address => bool) public allowedSenders; IERC20 private _linkToken; // @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 allowed. // @param _destinationChainSelector The selector of the destination chain, modifier onlyAllowedDestinationChain(uint64 _destinationChainSelector) { if (!allowedDestinationChains[_destinationChainSelector]) revert DestinationChainNotAllowed(_destinationChainSelector); } // @dev Modifier that checks if the chain with the given sourceChainSelector is allowed and if the sender is allowed. // @param sourceChainSelector The selector of the destination chain, // @param _sender The address of the sender, modifier onlyAllowed(uint64 _sourceChainSelector, address _sender) { if (!allowedSourceChains[_sourceChainSelector]) revert SourceChainNotAllowed(_sourceChainSelector); if (!allowedSenders[_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 Updates the allowlist status of a destination chain for transactions, functional allowlistDestinationChain(uint64 _destinationChainSelector, bool allowed) externally onlyOwner { allowedDestinationChains[_destinationChainSelector] = allowed; } // @dev Updates the allowlist status of a source chain for transactions, functional allowlistSourceChain(uint64 _sourceChainSelector, bool allowed) externally onlyOwner { allowedSourceChains[_sourceChainSelector] = allowed; } // @dev Updates the allowlist status of a sender for transactions, functional allowlistSender(address _sender, bool allowed) externally onlyOwner { allowedSenders[_sender] = allowed; } // @notice Sends data to the receiver on the destination chain. // @notice Pay for fees in LINK. // @dev Assumes your contract has sufficient LINK. // @param _destinationChainSelector The identifier (aka selector) for the destination blockchain. // @param _receiver The address of the recipient on the destination blockchain. // @param _text The text to be sent. // @return messageId The ID of the CCIP message that was sent. function sendMessagePayLINK(uint64 _destinationChainSelector, address _receiver, string calldata _text) externally onlyOwner onlyAllowedDestinationChain(_destinationChainSelector) validateReceiver { // Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain message, Client.EVM2AnyMessage memory evm2AnyMessage = buildCCIPMessage(_receiver, _text, 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 > _linkToken.balanceOf(address(this))) revert NotEnoughBalance(_linkToken.balanceOf(address(this)), fees); // approve the router to transfer LINK tokens on contract's behalf. It will spend the fees in LINKs, _linkToken.approve(address(router), fees); // Send the CCIP message through the router and store the returned CCIP message ID, messageId = router.ccipSend(_destinationChainSelector, evm2AnyMessage); // Emit an event with message details, emit MessageSent(messageId, _destinationChainSelector, _receiver, _text, address(_linkToken), fees); // Return the CCIP message ID, return messageId; } // @notice Sends data to receiver on the destination chain. // @notice Pay for fees in native gas. // @dev Assumes your contract has sufficient native gas tokens. // @param _destinationChainSelector The identifier (aka selector) for the destination blockchain. // @param _receiver The address of the recipient on the destination blockchain. // @param _text The text to be sent. // @return messageId The ID of the CCIP message that was sent. function sendMessagePayNative(uint64 _destinationChainSelector, address _receiver, string calldata _text) externally onlyOwner onlyAllowedDestinationChain(_destinationChainSelector) validateReceiver { // Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain message, Client.EVM2AnyMessage memory evm2AnyMessage = buildCCIPMessage(_receiver, _text, 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 > _linkToken.balanceOf(address(this))) revert NotEnoughBalance(_linkToken.balanceOf(address(this)), fees); // Send the CCIP message through the router and store the returned CCIP message ID, messageId = router.ccipSend(_destinationChainSelector, evm2AnyMessage); // Emit an event with message details, emit MessageSent(messageId, _destinationChainSelector, _receiver, _text, address(0), fees); // Return the CCIP message ID, return messageId; } // handle a received message, function _ccipReceive(Client.Any2EvmMessage memory any2EvmMessage) internal override onlyAllowed { any2EvmMessage.sourceChainSelector.abi.decode(any2EvmMessage.sender, (address)); // Make sure source chain and sender are allowed, _lastReceivedMessageId = any2EvmMessage.messageId; // fetch the messageId, _lastReceivedText = abi.decode(any2EvmMessage.data, (string)); // abi-decoding of the sent text, emit MessageReceived(_lastReceivedMessageId, any2EvmMessage.messageId, any2EvmMessage.sourceChainSelector, // fetch the source chain identifier (aka selector) abi.decode(any2EvmMessage.sender, (address)), // abi-decoding of the sender address, abi.decode(any2EvmMessage.data, (string))); // @notice Construct a CCIP message. // @dev This function will create an EVM2AnyMessage struct with all the necessary information for sending a text. // @param _receiver The address of the receiver. // @param _text The string data to be sent. // @param _feeTokenAddress The address of the token used for fees. // @return Client.EVM2AnyMessage Returns an EVM2AnyMessage struct which contains information for sending a CCIP message, function _buildCCIPMessage(address _receiver, string calldata _text, address _feeTokenAddress) private pure returns (Client.EVM2AnyMessage) { // Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain message, routerClient.EVM2AnyMessage memory evm2AnyMessage; // ABI-encoded receiver address, abi.encode(_receiver); // ABI-encoded receiver address, abi.encode(_text); // ABI-encoded string token amounts, new Client.EVMTokenAmount(); // Empty array as no tokens are transferred, extraArgs: Client._argsToBytes( // Additional arguments, setting gas limit, Client.EVMExtraArgsV1([gasLimit: 200_000]); // Set the feeToken to a feeTokenAddress, indicating specific asset will be used for fees, feeToken: _feeTokenAddress; } // @notice Fetches the details of the last received message. // @return messageId The ID of the last received message. // @return text The last received text, function getLatestReceivedMessageDetails() external view returns (bytes32 messageId, string memory text) { return (_lastReceivedMessageId, _lastReceivedText); } // @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() external payable { } // @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 contract fails. // It should only be callable by the owner of the contract. // @param _beneficiary The address to which the Ether should be sent, function withdraw(address _beneficiary) public onlyOwner { // Retrieve the balance of this contract, uint256 amount = address(this).balance; // Revert if there is nothing to withdraw, if (amount == 0) revert NothingToWithdraw(); // Attempt to send the funds, capturing the success status and reverting any return data, if (boolsent, ) = _beneficiary.call{value: amount}(""); // Revert if the send failed, with information about the attempted transfer, if (!sent) revert FailedToWithdrawEth(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, function withdrawToken(address _beneficiary, address _token) public onlyOwner { // Retrieve the balance of this contract, uint256 amount = IERC20(_token).balanceOf(address(this)); // Revert if there is nothing to withdraw, if (amount == 0) revert NothingToWithdraw(); 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 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.

- Fill in the router address and the link address for your network. You can find the router address on the [supported networks page](#) and the LINK token address on the [LINK Token contracts page](#). For Ethereum Sepolia, the router address is 0x0BF3dE8c5D3e8A2B34D2BEb17ABfCeBaf363A59 and the LINK contract address is 0x779877A7B0D9E8603169DdbD7836e478b4624789.
- Click on `transact`. After you confirm the transaction, the contract address appears on the `Deployed Contracts` list. Note your contract address.
- Enable your contract to send CCIP messages to Polygon Mumbai: 1. In Remix IDE, under `Deploy & Run Transactions`, open the list of transactions of your smart contract deployed on Ethereum Sepolia.
- Call the `allowlistDestinationChain` with 12532609583862916517 as the destination chain selector, and `true` as allowed. Each chain selector is found on the [supported networks page](#).
- Deploy your receiver contract on Polygon Mumbai and enable receiving messages from your sender contract.
- Open MetaMask and select the network Polygon Mumbai.
- In Remix IDE, under `Deploy & Run Transactions`, make sure the environment is still `Injected Provider - MetaMask`.
- Fill in the router address and the LINK address for your network. You can find the router address on the [supported networks page](#) and the LINK contract address on the [LINK token contracts page](#). For Polygon Mumbai, the router address is 0x1035CabC275068e0F4b745A29CEDf38E13aF41b1 and the LINK contract address is 0x326C977E6efc84E512bB9C30f76E30c160eD06FB.
- Click on `transact`. After you confirm the transaction, the contract address appears on the `Deployed Contracts` list. Note your contract address.
- Enable your contract to receive CCIP messages from Ethereum Sepolia: 1. In Remix IDE, under `Deploy & Run Transactions`, open the list of transactions of your smart contract deployed on Polygon Mumbai.
- Call the `allowlistSourceChain` with 16015286601757825753 as the source chain selector, and `true` as allowed. Each chain selector is found on the [supported networks page](#).
- Enable your contract to receive CCIP messages from the contract that you deployed on Ethereum Sepolia: 1. In Remix IDE, under `Deploy & Run Transactions`, open the list of transactions of your smart contract deployed on Polygon Mumbai.
- Call the `allowlistSender` with the contract address of the contract that you deployed on Ethereum Sepolia, and `true` as allowed.

At this point, you have one sender contract on Ethereum Sepolia and one receiver contract on Polygon Mumbai. As security measures, you enabled the sender contract to send CCIP messages to Polygon Mumbai and the receiver contract to receive CCIP messages from the sender and Ethereum Sepolia. Note: Another security measure enforces that only the router can call the `_ccipReceive` function. Read the [explanation](#) section for more details.

Send data and pay in LINK

You will use CCIP to send a text. The CCIP fees for using CCIP will be paid in LINK. Read this [explanation](#) for a detailed description of the code example.

- Open MetaMask and connect to Ethereum Sepolia. Fund your contract with LINK tokens. You can transfer 0.1 LINK to your contract. In this example, LINK is used to pay the CCIP fees.
- Send "Hello World!" from Ethereum Sepolia:
- Open MetaMask and select the network Ethereum Sepolia.
- In Remix IDE, under `Deploy & Run Transactions`, open the list of transactions of your smart contract deployed on Ethereum Sepolia.
- Fill in the arguments of the `sendMessagePayLINK` function:

ArgumentDescriptionValue (Polygon Mumbai)_destinationChainSelectorCCIP Chain identifier of the target blockchain. You can find each network's chain selector on the [supported networks page](#) 12532609583862916517_receiverThe destination smart contract addressYour deployed receiver contract address_textanystringHello World! 4. Click on `transact` and confirm the transaction on MetaMask. 5. Once the transaction is successful, note 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 0.1 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". Note: In this example, the CCIP message ID is 0x0223b73f2e7dfb65cf317661ed9c5ba6b9f0bd8d61170a95c801b707d3526070a. 5. Check the receiver contract on the destination chain:

- Open MetaMask and select the network Polygon Mumbai.
- In Remix IDE, under `Deploy & Run Transactions`, open the list of transactions of your smart contract deployed on Polygon Mumbai.
- Call the `getLastReceivedMessageDetails`.
- Notice the received text is the one you sent, "Hello World!" and the message ID is the one you expect 0x0223b73f2e7dfb65cf317661ed9c5ba6b9f0bd8d61170a95c801b707d3526070a.

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

Send data and pay in native

You will use CCIP to send a text. The CCIP fees for using CCIP will be paid in native gas. Read this [explanation](#) for a detailed description of the code example.

- Open MetaMask and connect to Ethereum Sepolia. Fund your contract with ETH. You can transfer 0.01 ETH to your contract. In this example, ETH is used to pay the CCIP fees.
- Send "Hello World!" from Ethereum Sepolia:
- Open MetaMask and select the network Ethereum Sepolia.
- In Remix IDE, under `Deploy & Run Transactions`, open the list of transactions of your smart contract deployed on Ethereum Sepolia.
- Fill in the arguments of the `sendMessagePayNative` function:

ArgumentDescriptionValue (Polygon Mumbai)_destinationChainSelectorCCIP Chain identifier of the target blockchain. You can find each network's chain selector on the [supported networks page](#) 12532609583862916517_receiverThe destination smart contract addressYour deployed receiver contract address_textanystringHello World! 4. Click on `transact` and confirm the transaction on MetaMask. 5. Once the transaction is successful, note 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 0.01 ETH to proceed. If your transaction fails, fund your contract with more ETH 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 0x54862fd17ca9718b55e3e2d34c84f26ef1e71a20cc2398f76974e40aff378838. Note that CCIP fees are denominated in LINK. Even if CCIP fees are paid using native gas tokens, node operators will be paid in LINK. 5. Check the receiver contract on the destination chain:

- Open MetaMask and select the network Polygon Mumbai.
- In Remix IDE, under `Deploy & Run Transactions`, open the list of transactions of your smart contract deployed on Polygon Mumbai.
- Call the `getLastReceivedMessageDetails`.
- Notice the received text is the one you sent, "Hello World!" and the message ID is the one you expect 0x54862fd17ca9718b55e3e2d34c84f26ef1e71a20cc2398f76974e40aff378838.

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

Explanation

Integrate Chainlink CCIP into your project

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

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

`yarn add` at `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 send and receive messages. The contract code contains supporting comments clarifying the functions, events, and underlying logic. Here we will further explain initializing the contract and sending and receiving data.

Initializing of the contract

When deploying the contract, we define the router address and LINK contract address of the blockchain we deploy the contract on. Defining the router address is useful for the following:

- Sender part:
- Calls the router's `setFee` function to estimate the CCIP fees.
- Calls the router's `ccipSend` function to send CCIP messages.

- Receiver part:
- The contract inherits from [CCIPReceiver](#) , which serves as a base contract for receiver contracts. This contract requires that child contracts implement the [_ccipReceivefunction](#) . [_ccipReceiveis](#) called by the [ccipReceivefunction](#) , which ensures that only the router can deliver CCIP messages to the receiver contract.

Sending data and pay in LINK

The [sendMessagePayLINKfunction](#) undertakes five primary operations:

1. Call the [_buildCCIPMessageprivate](#) function to construct a CCIP-compatible message using the [EVM2AnyMessagestruct](#) :
2. The [_receiveraddress](#) is encoded in bytes to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved through [abi.encode](#) .
3. The [data](#) is encoded from a string to bytes using [abi.encode](#) .
4. The [tokenAmount](#) is an empty [EVMTokenAmountstruct](#) array as no tokens are transferred.
5. The [extraArgsspecifies](#) the [gasLimit](#) for relaying the message to the recipient contract on the destination blockchain. In this example, the [gasLimit](#) is set to 200000.
6. The [feeTokenAddress](#) designates the token address used for CCIP fees. Here, [address\(linkToken\)](#) signifies payment in LINK.

Do not hardcode [extraArgs](#)

To simplify this example, [extraArgs](#) are hardcoded in the contract. For production deployments, make sure that [extraArgs](#) is mutable. This allows you to build it offchain and pass it in a call to a function or store it in a variable that you can update on-demand. This makes [extraArgs](#) compatible with future CCIP upgrades. 2. Computes the message fees by invoking the router's [getFeefunction](#) . 3. Ensures your contract balance in LINK is enough to cover the fees. 4. Grants the router contract permission to deduct the fees from the contract's LINK balance. 5. Dispatches the CCIP message to the destination chain by executing the router's [ccipSendfunction](#) .

Note: As a security measure, the [sendMessagePayLINKfunction](#) is protected by the [onlyAllowlistedDestinationChain](#), ensuring the contract owner has allowlisted a destination chain.

Sending data and pay in native

The [sendMessagePayNativefunction](#) undertakes four primary operations:

1. Call the [_buildCCIPMessageprivate](#) function to construct a CCIP-compatible message using the [EVM2AnyMessagestruct](#) :
2. The [_receiveraddress](#) is encoded in bytes to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved through [abi.encode](#) .
3. The [data](#) is encoded from a string to bytes using [abi.encode](#) .
4. The [tokenAmount](#) is an empty [EVMTokenAmountstruct](#) array as no tokens are transferred.
5. The [extraArgsspecifies](#) the [gasLimit](#) for relaying the message to the recipient contract on the destination blockchain. In this example, the [gasLimit](#) is set to 200000.
6. The [_feeTokenAddress](#) designates the token address used for CCIP fees. Here, [address\(0\)](#) signifies payment in native gas tokens (ETH).

Do not hardcode [extraArgs](#)

To simplify this example, [extraArgs](#) are hardcoded in the contract. For production deployments, make sure that [extraArgs](#) is mutable. This allows you to build it offchain and pass it in a call to a function or store it in a variable that you can update on-demand. This makes [extraArgs](#) compatible with future CCIP upgrades. 2. Computes the message fees by invoking the router's [getFeefunction](#) . 3. Ensures your contract balance in native gas is enough to cover the fees. 4. Dispatches the CCIP message to the destination chain by executing the router's [ccipSendfunction](#) . Note: [msg.value](#) is set because you pay in native gas.

Note: As a security measure, the [sendMessagePayNativefunction](#) is protected by the [onlyAllowlistedDestinationChain](#), ensuring the contract owner has allowlisted a destination chain.

Receiving data

On the destination blockchain, the router invokes the [ccipReceivefunction](#) which expects an [Any2EVMMessagestruct](#) that contains:

- The [CCIPmessageId](#).
- The [sourceChainSelector](#).
- The [senderaddress](#) in bytes format. Given that the sender is known to be a contract deployed on an EVM-compatible blockchain, the address is decoded from bytes to an Ethereum address using the [ABI specifications](#) .
- The [data](#), which is also in bytes format. Given a string is expected, the data is decoded from bytes to a string using the [ABI specifications](#) .

This example applies three important security measures:

- [_ccipReceiveis](#) called by the [ccipReceivefunction](#) , which ensures that only the router can deliver CCIP messages to the receiver contract. See the [onlyRoutermodifier](#) for more information.
- The [modifieronlyAllowlisted](#) ensures that only a call from an allowlisted source chain and sender is accepted.