# Getting Started

A simple use case for Chainlink CCIP is sending data between smart contracts on different blockchains. This guide shows you how to deploy a CCIP sender contract and a CCIP receiver contract to two different blockchains and send data from the sender contract to the receiver contract. You pay the CCIP fees using LINK.

Fees can also be paid in alternative assets, which currently include the native gas tokens of the source blockchain and their ERC20 wrapped version. For example, you can pay ETH or WETH when you send transactions to the CCIP router on Ethereum and MATIC or WMATIC when you send transactions to the CCIP router on Polygon.

## Before you begin

- If you are new to smart contract development, learn how toDeploy Your First Smart Contract so you are familiar with the tools that are necessary for this guide:* TheSolidity programming language
- TheMetaMask wallet
- TheRemix development environment
- Acquire testnet funds. This guide requires a testnet ETH and LINK onEthereum Sepolia. It also requires testnet MATIC onPolygon Mumbai. If you need to use different networks, you can find more faucets on theLINK Token Contracts page.* Go tofaucets.chain.link and get testnet ETH and LINK onEthereum Sepolia.
- Go tofaucet.polygon.technology to acquire testnet MATIC.
- Learn how toFund your contract with LINK .

## Deploy the sender contract

Deploy theSender.solcontract onEthereum Sepolia. To see a detailed explanation of this contract, read theCode Explanation section.

1. Open the Sender.sol contract in Remix.

Open in Remix What is Remix? 2. Compile the contract. 3. Deploy the sender contract onEthereum Sepolia:

1. Open MetaMask and select theEthereum Sepolianetwork.
2. In Remix under theDeploy & Run Transactionstab, selectInjected Provider - MetaMaskin theEnvironmentlist. Remix will use the MetaMask wallet to communicate withEthereum Sepolia.
3. Under theDeploysection, fill in the router address and the LINK token contract addresses for your specific blockchain. You can find both of these addresses on theSupported Networks page. The LINK token contract address is also listed on theLINK Token Contracts page. ForEthereum Sepolia, the router address is0x0BF3dE8c5D3e8A2B34D2BEeB17ABfCeBaf363A59and the LINK address is0x779877A7B0D9E8603169DdbD7836e478b4624789.
4. Click theDeploybutton to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract toEthereum Sepolia.
5. After you confirm the transaction, the contract address appears in theDeployed Contractslist. Copy your contract address.
6. Open MetaMask and send0.1LINK to the contract address that you copied. Your contract will pay CCIP fees in LINK.

## Deploy the receiver contract

Deploy the receiver contract onPolygon Mumbai. You will use this contract to receive data from the sender that you deployed onEthereum Sepolia. To see a detailed explanation of this contract, read theCode Explanation section.

1. Open the Receiver.sol contract in Remix.

Open in Remix What is Remix? 2. Compile the contract. 3. Deploy the receiver contract onPolygon Mumbai:

1. Open MetaMask and select thePolygon Mumbainetwork.
2. In Remix under theDeploy & Run Transactionstab, make sure theEnvironmentis still set toInjected Provider - MetaMask.
3. Under theDeploysection, fill in the router address field. ForPolygon Mumbai, the Router address is0x1035CabC275068e0F4b745A29CEDf38E13aF41b1. You can find the addresses for each network on theSupported Networks page.
4. Click theDeploybutton to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract toPolygon Mumbai.
5. After you confirm the transaction, the contract address appears as the second item in theDeployed Contractslist. Copy this contract address.

You now have onesendercontract onEthereum Sepoliaand onereceivercontract onPolygon Mumbai. You sent0.1LINK to thesendercontract to pay the CCIP fees. Next, send data from the sender contract to the receiver contract.

## Send data

Send aHello World!string from your contract onEthereum Sepoliato the contract you deployed onPolygon Mumbai:

1. Open MetaMask and select theEthereum Sepolianetwork.
2. In Remix under theDeploy & Run Transactionstab, expand the first contract in theDeployed Contractssection.
3. Expand thesendMessagefunction and fill in the following arguments:

ArgumentDescriptionValue (Polygon Mumbai)destinationChainSelectorCCIP Chain identifier of the target blockchain. You can find each network's chain selector on thesupported networks page12532609583862916517receiverThe destination smart contract addressYour deployed contract addresstextAnystringHello World! 4. Click thetransactbutton to run the function. MetaMask prompts you to confirm the transaction.

note

During gas price spikes, your transaction might fail, requiring more than0.1 LINKto proceed. If your transaction fails, fund your contract with moreLINKtokens and try again. 5. After the transaction is successful, note the transaction hash. Here is anexample of a successful transaction onEthereum Sepolia.

After the transaction is finalized on the source chain, it will take a few minutes for CCIP to deliver the data toPolygon Mumbaiand call theccipReceivefunction on your receiver contract. You can use theCCIP explorer to see the status of your CCIP transaction and then read data stored by your receiver contract.

1. Open theCCIP explorer and use the transaction hash that you copied to search for your cross-chain transaction. The explorer provides several details about your request.
2. When the status of the transaction is marked with a "Success" status, the CCIP transaction and the destination transaction are complete.

## Read data

Read data stored by the receiver contract onPolygon Mumbai:

1. Open MetaMask and select thePolygon Mumbainetwork.
2. In Remix under theDeploy & Run Transactionstab, open the list of contracts of your smart contract deployed onPolygon Mumbai.
3. Click thegetLastReceivedMessageDetailsfunction button to read the stored data. In this example, it is "Hello World!".

Congratulations! You just sent your first cross-chain data using CCIP. Next, examine the example code to learn how this contract works.

## Examine the example code

### Sender code

The smart contract in this tutorial is designed to interact with CCIP to send data. The contract code includes comments to clarify the various functions, events, and underlying logic. However, this section explains the key elements. You can see the full contract code below.

// SPDX-License-Identifier: MITpragmasolidity0.8.19;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{LinkTokenInterface}from"@chainlink/contracts/src/v0.8/shared/interfaces/LinkTokenInterface.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 contract for sending string data across chains.contractSenderisOwnerIsCreator{// Custom errors to provide more descriptive revert messages.errorNotEnoughBalance(uint256currentBalance,uint256calculatedFees);// Used to make sure contract has enough balance.// Event emitted when a message is sent to another chain.eventMessageSent(bytes32indexedmessageId,// The unique ID of the CCIP message.uint64indexeddestinationChainSelector,// The chain selector of the destination chain.addressreceiver,// The address of the receiver on the destination chain.stringtext,// The text being sent.addressfeeToken,// the token address used to pay CCIP fees.uint256fees// The fees paid for sending the CCIP message.);IRouterClientprivates_router;LinkTokenInterfaceprivates_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){s_router=IRouterClient(_router);s_linkToken=LinkTokenInterface(_link);}/// @notice Sends data to receiver on the destination chain./// @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 string text to be sent./// @return messageId The ID of the message that was sent.functionsendMessage(uint64destinationChainSelector,addressreceiver,stringcalldatatext)externalonlyOwnerreturns(bytes32messageId){// Create an EVM2AnyMessage struct in memory with necessary information for sending a cross-chain messageClient.EVM2AnyMessagememoryevm2AnyMessage=Client.EVM2AnyMessage({receiver:abi.encode(receiver),// ABI-encoded receiver addressdata:abi.encode(text),// ABI-encoded stringtokenAmounts:newClient.EVMTokenAmount,// Empty array indicating no tokens are being sentextraArgs:Client._argsToBytes(// Additional arguments, setting gas limitClient.EVMExtraArgsV1({gasLimit:200_000})),// Set the feeToken address, indicating LINK will be used for feesfeeToken:address(s_linkToken)});// Get the fee required to send the messageuint256fees=s_router.getFee(destinationChainSelector,evm2AnyMessage);if(fees>s_linkToken.balanceOf(address(this)))revertNotEnoughBalance(s_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(s_router),fees);// Send the message through the router and store the returned message IDmessageId=s_router.ccipSend(destinationChainSelector,evm2AnyMessage);// Emit an event with message detailsemitMessageSent(messageId,destinationChainSelector,receiver,text,address(s_linkToken),fees);// Return the message IDreturnmessageId;}} Open in Remix What is Remix?

#### Initializing the contract

When deploying the contract, you define the router address and the LINK contract address of the blockchain where you choose to deploy the contract.

The router address provides functions that are required for this example:

- ThegetFeefunction to estimate the CCIP fees.
- TheccipSendfunction to send CCIP messages.

#### Sending data

ThesendMessagefunction completes several operations:

1. Construct a CCIP-compatible message using theEVM2AnyMessagestruct :
2. Thereceiveraddress is encoded in bytes format to accommodate non-EVM destination blockchains with distinct address formats. The encoding is achieved throughabi.encode .
3. Thedatais encoded from a string text to bytes usingabi.encode .
4. ThetokenAmountsis an array. Each element comprises astruct that contains the token address and amount. In this example, the array is empty because no tokens are sent.
5. TheextraArgsspecify thegasLimitfor relaying the CCIP message to the recipient contract on the destination blockchain. In this example, thegasLimitis set to200000.
6. ThefeeTokendesignates the token address used for CCIP fees. Here,address(linkToken)signifies payment in LINK.
7. Compute the CCIP message fees by invoking the router'sgetFeefunction .
8. Ensure that your contract balance in LINK is enough to cover the fees.
9. Grant the router contract permission to deduct the fees from the contract's LINK balance.
10. Dispatch the CCIP message to the destination chain by executing the router'sccipSendfunction .

Sender contract best practices

This example is simplified for learning purposes. For production code, use the following best practices:

- Do not hardcodeextraArgs: To simplify the example,extraArgsare hardcoded in the contract. The recommendation is to make sureextraArgsis mutable. For example, you can buildextraArgsoffchain and pass it in your functions call or store it in a storage variable that you can update on demand. Thus, you can make sureextraArgsremains backward compatible for future CCIP upgrades.
- Validate the destination chain.

### Receiver code

The smart contract in this tutorial is designed to interact with CCIP to receive data. The contract code includes comments to clarify the various functions, events, and underlying logic. However, this section explains the key elements. You can see the full contract code below.

// SPDX-License-Identifier: MITpragmasolidity0.8.19;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";/* ** 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 contract for receiving string data across chains.contractReceiverisCCIPReceiver{// Event emitted when a message is received from another chain.eventMessageReceived(bytes32indexedmessageId,// The unique ID of the message.uint64indexedsourceChainSelector,// The chain selector of the source chain.addresssender,// The address of the sender from the source chain.stringtext// The text that was received.);bytes32privates_lastReceivedMessageId;// Store the last received messageId.stringprivates_lastReceivedText;// Store the last received text./// @notice Constructor initializes the contract with the router address./// @param router The address of the router contract.constructor(addressrouter)CCIPReceiver(router){}/// handle a received messagefunction_ccipReceive(Client.Any2EVMMessagememoryany2EvmMessage)internaloverride{s_lastReceivedMessageId=any2EvmMessage.messageId;// fetch the messageIds_lastReceivedText=abi.decode(any2EvmMessage.data,(string));// abi-decoding of the sent textemitMessageReceived(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 Fetches the details of the last received message./// @return messageId The ID of the last received message./// @return text The last received text.functiongetLastReceivedMessageDetails()externalviewreturns(bytes32messageId,stringmemorytext){return(s_lastReceivedMessageId,s_lastReceivedText);}} Open in Remix What is Remix?

#### Initializing the contract

When you deploy the contract, you define the router address. The receiver contract inherits from theCCIPReceiver.sol contract, which uses the router address.

#### Receiving data

On the destination blockchain:

1. The CCIP Router invokes theccipReceivefunction .Note: This function is protected by theonlyRoutermodifier , which ensures that only the router can call the receiver contract.
2. TheccipReceivefunction calls an internal function_ccipReceivefunction . The receiver contract implements this function.
3. This_ccipReceivefunction expects anAny2EVMMessagestruct that contains the following values:
4. The CCIPmessageId.
5. ThesourceChainSelector.
6. Thesenderaddress in bytes format. The sender is a contract deployed on an EVM-compatible blockchain, so the address is decoded from bytes to an Ethereum address using theABI specification .
7. Thedatais also in bytes format. Astringis expected, so the data is decoded from bytes to a string using theABI specification .

Recommendations Receiver contract

The example was simplified for learning purposes. For production code, use the following best practices:

- Validate the source chain.
- Depending on your use case, analyze whether you should validate the sender address.

Note that the receiver contract in this example inherits from the base contractCCIPReceiver.sol , which uses theonlyRoutermodifier to ensure that only the router can call theccipReceivefunction .