

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 to [Deploy Your First Smart Contract](#) so you are familiar with the tools that are necessary for this guide.* The [Solidity](#) programming language
- The [MetaMask](#) wallet
- The [Remix](#) development environment
- Acquire testnet funds. This guide requires a testnet ETH and LINK on Ethereum Sepolia. It also requires testnet MATIC on Polygon Mumbai. If you need to use different networks, you can find more faucets on the [LINK Token Contracts](#) page.* Go to [faucets.chain.link](#) and get testnet ETH and LINK on Ethereum Sepolia.
- Go to [faucet.polygon.technology](#) to acquire testnet MATIC.
- Learn how to [Fund your contract with LINK](#).

Deploy the sender contract

Deploy the `Sender.sol` contract on Ethereum Sepolia. To see a detailed explanation of this contract, read the [Code 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 the Ethereum Sepolia network.
2. In Remix under the Deploy & Run Transaction tab, select Injected Provider - MetaMask in the Environment list. Remix will use the MetaMask wallet to communicate with Ethereum Sepolia.
3. Under the Deployment section, fill in the router address and the LINK token contract addresses for your specific blockchain. You can find both of these addresses on the [Supported Networks](#) page. The LINK token contract address is also listed on the [LINK Token Contracts](#) page. For Ethereum Sepolia, the router address is 0x0BF3d2E8c5D3e8A2B34D2BEb17ABfCeBaf363A59 and the LINK address is 0x779877A7B80D9E8603169DbD7836e478B4624789.
4. Click the Deploy button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract to Ethereum Sepolia.
5. After you confirm the transaction, the contract address appears in the Deployed Contracts list. Copy your contract address.
6. Open MetaMask and send 0.1 LINK to the contract address that you copied. Your contract will pay CCIP fees in LINK.

Deploy the receiver contract

Deploy the receiver contract on Polygon Mumbai. You will use this contract to receive data from the sender that you deployed on Ethereum Sepolia. To see a detailed explanation of this contract, read the [Code 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 on Polygon Mumbai:

1. Open MetaMask and select the Polygon Mumbai network.
2. In Remix under the Deploy & Run Transaction tab, make sure the Environment is still set to Injected Provider - MetaMask.
3. Under the Deployment section, fill in the router address field. For Polygon Mumbai, the Router address is `0x1035CabC275068e0F4b745A29CEDf38E13aF41b1`. You can find the addresses for each network on the [Supported Networks](#) page.
4. Click the Deploy button to deploy the contract. MetaMask prompts you to confirm the transaction. Check the transaction details to make sure you are deploying the contract to Polygon Mumbai.
5. After you confirm the transaction, the contract address appears as the second item in the Deployed Contracts list. Copy this contract address.

You now have `onesendercontract` on Ethereum Sepolia and `onereceivercontract` on Polygon Mumbai. You sent 0.1 LINK to the `sendercontract` to pay the CCIP fees. Next, send data from the `sendercontract` to the `receivercontract`.

Send data

Send a `Hello World!` string from your contract on Ethereum Sepolia to the contract you deployed on Polygon 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 the[supported networks page](#)1253260958386291651receiverThe 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 than 0.1 LINK to proceed. If your transaction fails, fund your contract with more LINK tokens and try again. 5. After the transaction is successful, note the transaction hash. Here is an [example](#) of a successful transaction on Ethereum Sepolia.

After the transaction is finalized on the source chain, it will take a few minutes for CCIP to deliver the data to Polygon Mumbai and call the `theccipReceive` function on your receiver contract. You can use the [CCIP explorer](#) to see the status of your CCIP transaction and then read data stored by your receiver contract.

1. Open the [CCIP 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 the Polygon Mumbai network.
2. In Remix under the Deploy & Run Transaction tab, open the list of contracts of your smart contract deployed on Polygon Mumbai.
3. Click the `getLastReceivedMessageDetails` function 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: MITpragma solidity 0.8.19;import({IRouterClient}from"@chainlink/contracts-ccip/src/v0.8/ccip/interfaces/IRouterClient.sol");import({OwnableCreator}from"@chainlink/contracts-ccip/src/v0.8/shared/access/OwnableCreator.sol");import({Client}from"@chainlink/contracts-ccip/src/v0.8/ccip/libraries/Client.sol");import({LinkTokenInterface}from"@chainlink/contracts-ccip/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.contract Senderis OwnableCreator{// 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, uint) The unique ID of the CCIP message,uint64indexeddestinationChainSelector, uint) The chain selector of the destination chain.addressreceiver, uint) The address of the receiver on the destination chain.stringtext, uint) The text being sent.addressfeeToken, uint) the token address used to pay CCIP fees.uint256fees, uint) 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)externallyOwnedreturns(bytes32messageId){ Create an EVM2AnyMessage struct in memory with
```

necessary information for sending a cross-chain message

```
Client.EVM2AnyMessage memory evm2AnyMessage = Client.EVM2AnyMessage({
  receiver: abi.encode(receiver), // ABI-encoded receiver address
  data: abi.encode(text), // ABI-encoded string
  tokenAmounts: new Client.EVMTokenAmount(), // Empty array indicating no tokens are being sent
  extraArgs: Client._argsToBytes(), // Additional arguments, setting gas limit
  Client.EVMExtraArgsV1({
    gasLimit: 200_000
  })
}); // Set the feeToken address, indicating LINK will be used for fees
feeToken: address(s_linkToken)); // Get the fee required to send the message
uint256 fees = s_router.getFee(destinationChainSelector, evm2AnyMessage);
if (fees > s_linkToken.balanceOf(address(this))) revert NotEnoughBalance(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 ID
messageId = s_router.ccipSend(destinationChainSelector, evm2AnyMessage); // Emit an event with message details
emit MessageSent(messageId, destinationChainSelector, receiver, text, address(s_linkToken), fees); // Return the message ID
return messageId;
}
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:

- The `getFee` function to estimate the CCIP fees.
- The `ccipSend` function to send CCIP messages.

Sending data

The `sendMessage` function completes several operations:

1. Construct a CCIP-compatible message using the `EVM2AnyMessage` struct :
2. The `receiverAddress` is encoded in bytes format 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 text to bytes using `abi.encode` .
4. The `tokenAmounts` is an array. Each element comprises a struct that contains the token address and amount. In this example, the array is empty because no tokens are sent.
5. The `extraArgs` specify the gas limit for relaying the CCIP message to the recipient contract on the destination blockchain. In this example, the gas limit is set to 200,000.
6. The `feeToken` designates the token address used for CCIP fees. Here, `address(linkToken)` signifies payment in LINK.
7. Compute the CCIP message fees by invoking the router's `getFee` function .
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's `ccipSend` function .

Sender contract best practices

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

- Do not hardcode `extraArgs`: To simplify the example, `extraArgs` are hardcoded in the contract. The recommendation is to make sure `extraArgs` is mutable. For example, you can build `extraArgs` offchain and pass it in your functions call or store it in a storage variable that you can update on demand. Thus, you can make sure `extraArgs` remains 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: MIT
pragma solidity 0.8.19;
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.
contract Receiver is CCIPReceiver {
  // Event emitted when a message is received from another chain.
  event MessageReceived(bytes32 indexed messageId, // The unique ID of the 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.

  /// @notice Constructor initializes the contract with the router address.
  /// @param router The address of the router
  constructor(address router) CCIPReceiver(router) {}

  // handle a received message
  function _ccipReceive(
    Client.Any2EVMMessage memory any2EvmMessage
  ) internal override {
    _lastReceivedMessageId = any2EvmMessage.messageId; // fetch the messageId
    _lastReceivedText = abi.decode(
      any2EvmMessage.data, (string)
    ); // abi-decoding of the sent text
    emit MessageReceived(
      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.
  function getLastReceivedMessageDetails() external view returns (bytes32 messageId, string memory text) {
    return (_lastReceivedMessageId, _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 the `CCIPReceiver` contract, which uses the router address.

Receiving data

On the destination blockchain:

1. The CCIP Router invokes the `ccipReceive` function . Note: This function is protected by the `onlyRouter` modifier , which ensures that only the router can call the receiver contract.
2. The `ccipReceive` function calls an internal function `_ccipReceive` . The receiver contract implements this function.
3. This `_ccipReceive` function expects an `Any2EVMMessage` struct that contains the following values:
4. The `CCIPMessageId`.
5. The `sourceChainSelector`.
6. The `senderAddress` 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 the `ABI specification` .
7. The `data` is also in bytes format. A string is expected, so the data is decoded from bytes to a string using the `ABI 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 contract `CCIPReceiver` , which uses the `onlyRouter` modifier to ensure that only the router can call the `ccipReceive` function .