Local testing using a Mock contract

You are viewing the VRF v2 guide - Subscription method

To learn how to request random numbers without a subscription, see the direct funding method guide.

Security Considerations

Be sure to review your contracts with the security considerations in mind.

This guide explains how to test Chainlink VRF v2 on a Remix IDE sandbox blockchain environment.Note: You can reuse the same logic on another development environment, such as Hardhat or Truffle. For example, read the Hardhat Starter Kit RandomNumberConsumer unit tests .

Test on public testnets thoroughly

Even though local testing has several benefits, testing with a VRF mock covers the bare minimum of use cases. Make sure to test your consumer contract throughly on public testnets.

Benefits of local testing

Testing locally using mock contracts saves you time and resources during development. Some of the key benefits include:

- Faster feedback loop: Immediate feedback on the functionality and correctness of your smart contracts. This helps you quickly identify and fix issues without waiting for transactions to be mined/validated on a testnet.
- Saving your native testnet gas: Deploying and interacting with contracts requires paying gas fees. Although native testnet gas does not have any associated value, supply is limited by public faucets. Using mock contracts locally allows you to test your contracts freely without incurring any expenses.
- Controlled environment: Local testing allows you to create a controlled environment where you can manipulate various parameters, such as block time and gas prices, to test your smart contracts' function as expected under different conditions.
- Isolated testing: You can focus on testing individual parts of your contract, ensuring they work as intended before integrating them with other components.
- Easier debugging: Because local tests run on your machine, you have better control over the debugging process. You can set breakpoints, inspect variables, and step through your code to identify and fix issues.
- Comprehensive test coverage: You can create test cases to cover all possible scenarios and edge cases.

Testing logic

Complete the following tasks to test your VRF v2 consumer locally:

1. Deploy the VRFCoordinatorV2Mock . This contract is a mock of the VRFCoordinatorV2 contract.
2. Call the VRFCoordinatorV2Mock createSubscription function to create a new subscription.
3. Call the VRFCoordinatorV2Mock fundSubscription function to fund your newly created subscription.Note: You can fund with an arbitrary amount.
4. Deploy your VRF consumer contract.
5. Call the the VRFCoordinatorV2Mock addConsumer function to add your consumer contract to your subscription.
6. Request random words from your consumer contract.
7. Call the VRFCoordinatorV2Mock fulfillRandomWords function to fulfill your consumer contract request.

Testing

Open the contracts on RemixIDE

OpenVRFCoordinatorV2Mockand compile in Remix:

```
// SPDX-License-Identifier: MITpragmasolidity^0.8.7;import"@chainlink/contracts/src/v0.8/mocks/VRFCoordinatorV2Mock.sol";
```

Open in Remix What is Remix? OpenVRFv2Consumerand compile in Remix:

```
// SPDX-License-Identifier: MIT// An example of a consumer contract that relies on a subscription for
funding.pragmasolidity^0.8.7;import{VRFCoordinatorV2Interface}from"@chainlink/contracts/src/v0.8/interfaces/VRFCoordinatorV2Interface.sol";import{VRFConsumerBaseV2}from"@chainlink/contracts/s
* 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 The RandomNumberConsumerV2 contract * @notice A contract that gets random values from Chainlink VRF V2
/contractRandomNumberConsumerV2isVRFConsumerBaseV2{VRFCoordinatorV2Interface immutable COORDINATOR;// Your subscription ID.uint64immutable s_subscriptionId;// The gas lane to use, which specifies the maximum gas price to bump to.// For a list of available gas lanes on each network,// see https://docs.chain.link/docs/vrf-contracts/#configurationsbytes32immutable s_keyHash;// Depends on the number of requested values that you want sent to the// fulfillRandomWords() function. Storing each word costs about 20,000 gas,// so 100,000 is a safe default for this example contract. Test and adjust// this limit based on the network that you select, the size of the request,// and the processing of the callback request in the fulfillRandomWords()// function.uint32constantCALLBACK_GAS_LIMIT=100000;// The default is 3, but you can set this higher.uint16constantREQUEST_CONFIRMATIONS=3;// For this example, retrieve 2 random values in one request.// Cannot exceed
VRFCoordinatorV2.MAX_NUM_WORDS.uint32constantNUM_WORDS=2;uint256[]publics_randomWords;uint256publics_requestId;addresss_owner;eventReturnedRandomness(uint256[]randomWords)
* @notice Constructor inherits VRFConsumerBaseV2 * * @param subscriptionId - the subscription ID that this contract uses for funding requests * @param vrfCoordinator - coordinator, check https://docs.chain.link/docs/vrf-contracts/#configurations * @param keyHash - the gas lane to use, which specifies the maximum gas price to bump to
/constructor(uint64subscriptionId,addressvrfCoordinator,bytes32keyHash)VRFConsumerBaseV2(vrfCoordinator)
{COORDINATOR=VRFCoordinatorV2Interface(vrfCoordinator);s_keyHash=keyHash;s_owner=msg.sender;s_subscriptionId=subscriptionId;}/ * @notice Requests randomness * Assumes the subscription is funded sufficiently; "Words" refers to unit of data in Computer Science */functionrequestRandomWords()externalonlyOwner{// Will revert if subscription is not set and funded.s_requestId=COORDINATOR.requestRandomWords(s_keyHash,s_subscriptionId,REQUEST_CONFIRMATIONS,CALLBACK_GAS_LIMIT,NUM_WORDS);}/ * @notice Callback function used by VRF Coordinator * * @param - id of the request * @param randomWords - array of random results from VRF Coordinator /functionfulfillRandomWords(uint256/ requestId
*/,uint256[]memoryrandomWords)internaloverride{s_randomWords=randomWords;emitReturnedRandomness(randomWords);}modifieronlyOwner(){require(msg.sender==s_owner);_;}}
```

Open in Remix What is Remix? Your RemixIDE file explorer should displayVRFCoordinatorV2Mock.solandVRFv2Consumer.sol:

Deploy VRFCoordinatorV2Mock

1. OpenVRFCoordinatorV2Mock.sol.
2. UnderDEPLOY & RUN TRANSACTIONS, selectVRFCoordinatorV2Mock.
3. UnderDEPLOY, fill in the_BASEFEEand_GASPRICELINK. These variables are used in theVRFCoordinatorV2Mockcontract to represent the base fee and the gas price (in LINK tokens) for the VRF requests. You can set:_BASEFEE=100000000000000000and_GASPRICELINK=1000000000.
4. Click ontransactto deploy theVRFCoordinatorV2Mockcontract.
5. Once deployed, you should see theVRFCoordinatorV2Mockcontract underDeployed Contracts.
6. Note the address of the deployed contract.

Create and fund a subscription

1. Click oncreateSubscriptionto create a new subscription.
2. In the RemixIDE console, read your transaction decoded output to find the subscription ID. In this example, the subscription ID is1.
3. Click onfundSubscriptionto fund your subscription. In this example, you can set the_subidto1(which is your newly created subscription ID) and the_amountto10000000000000000000.

Deploy the VRF consumer contract

1. In the file explorer, openVRFv2Consumer.sol.
2. UnderDEPLOY & RUN TRANSACTIONS, selectRandomNumberConsumerV2.
3. UnderDEPLOY, fill inSUBSCRIPTIONIDwith your subscription ID,vrfCoordinatorwith the deployedVRFCoordinatorV2Mockaddress and,KEYHASHwith an arbitrarybytes32(In this example, you can set theKEYHASHto0xd89b2bf150e3b9e13446986e571fb9cab24b13cea0a43ea20a6049a85cc807cc).
4. Click ontransactto deploy theRandomNumberConsumerV2contract.
5. After the consumer contract is deployed, you should see theRandomNumberConsumerV2contract underDeployed Contracts.
6. Note the address of the deployed contract.

Add the consumer contract to your subscription

1. UnderDeployed Contracts, open the functions list of your deployedVRFCoordinatorV2Mockcontract.
2. Click onaddConsumerand fill in the_subidwith your subscription ID and_consumerwith your deployed consumer contract address.
3. Click ontransact.

Request random words

1. UnderDeployed Contracts, open the functions list of your deployedRandomNumberConsumerV2contract.
2. Click onrequestRandomWords.
3. In the RemixIDE console, read your transaction logs to find the VRF request ID. In this example, the request ID is1.
4. Note your request ID.

Fulfill the VRF request

Because you are testing on a local blockchain environment, you must fulfill the VRF request yourself.

1. UnderDeployed Contracts, open the functions list of your deployedVRFCoordinatorV2Mockcontract.
2. ClickfulfillRandomWordsand fill in_requestIdwith your VRF request ID and_consumerwith your consumer contract address.
3. Click ontransact.

Check the results

1. UnderDeployed Contracts, open the functions list of your deployedRandomNumberConsumerV2contract.
2. Click ons_requestIdto display the last request ID. In this example, the output is1.
3. Each time you make a VRF request, your consumer contract requests two random words. After the request is fulfilled, the two random words are stored in thes_randomWordsarray. You can check the stored random words by reading the two first indexes of thes_randomWordsarray. To do so, click on thes_randomWordsfunction and:
4. Fill in the index with0then click oncallto read the first random word.
5. Fill in the index with1then click oncallto read the second random word.

Next steps

This guide demonstrated how to test a VRF v2 consumer contract on your local blockchain. We made the guide on RemixIDE for learning purposes, but you can reuse the same testing logic on another development environment, such as Truffle or Hardhat. For example, read the Hardhat Starter KitRandomNumberConsumer unit tests .