

Getting Started with Chainlink VRF

Requirements

This guide assumes that you have basic knowledge about writing and deploying smart contracts. If you are new to smart contract development, learn how [Deploy Your First Smart Contract](#) before you begin.

VRF v2 - Developer Walkthrough

In this guide, you will learn about generating randomness on blockchains. This includes learning how to implement a Request and Receive cycle with Chainlink oracles and how to consume random numbers with Chainlink VRF in smart contracts.

How is randomness generated on blockchains? What is Chainlink VRF?

Randomness is very difficult to generate on blockchains. This is because every node on the blockchain must come to the same conclusion and form a consensus. Even though random numbers are versatile and useful in a variety of blockchain applications, they cannot be generated natively in smart contracts. The solution to this issue is [Chainlink VRF](#), also known as Chainlink Verifiable Random Function.

What is the Request and Receive cycle?

The [Data Feeds Getting Started](#) guide explains how to consume Chainlink Data Feeds, which consist of reference data posted onchain by oracles. This data is stored in a contract and can be referenced by consumers until the oracle updates the data again.

Randomness, on the other hand, cannot be reference data. If the result of randomness is stored onchain, any actor could retrieve the value and predict the outcome. Instead, randomness must be requested from an oracle, which generates a number and a cryptographic proof. Then, the oracle returns that result to the contract that requested it. This sequence is known as the [Request and Receive cycle](#).

What is the payment process for generating a random number?

VRF requests receive funding from subscription accounts. The [Subscription Manager](#) lets you create an account and pre-pay for VRF requests, so that funding of all your application requests are managed in a single location. To learn more about VRF requests funding, see [Subscriptions limits](#).

How can I use Chainlink VRF?

To see a basic implementation of Chainlink VRF, see [Get a Random Number](#). In this section, you will create an application that uses Chainlink VRF to generate randomness. The contract used in this application has a [Game of Thrones](#) theme.

After the contract requests randomness from Chainlink VRF, the result of the randomness will transform into a number between 1 and 20, mimicking the rolling of a 20 sided die. Each number represents a Game of Thrones house. If the dice land on the value 1, the user is assigned house Targaryan, 2 for Lannister, and so on. A full list of houses can be found [here](#).

When rolling the dice, it uses an address variable to track which address is assigned to each house.

The contract has the following functions:

- rollDice: This submits a randomness request to Chainlink VRF
- fulfillRandomWords: The function that the Oracle uses to send the result back
- house: To see the assigned house of an address

Note: to jump straight to the entire implementation, you can [open the VRFD20.sol contract](#) in remix.

Create and fund a subscription

Chainlink VRF requests receive funding from subscription accounts. The [Subscription Manager](#) lets you create an account and pre-pay your use of Chainlink VRF requests. For this example, create a new subscription on the Sepolia testnet as explained [here](#).

Importing VRFConsumerBaseV2 and VRFCoordinatorV2Interface

Chainlink maintains a [library of contracts](#) that make consuming data from oracles easier. For Chainlink VRF, you will use:

- [VRFConsumerBaseV2](#) that must be imported and extended from the contract that you create.
- [VRFCoordinatorV2Interface](#) that must be imported to communicate with the VRF coordinator.

// SPDX-License-Identifier:

MITpragmasolidity"0.8.7;import"@chainlink/contracts/src/v0.8/interfaces/VRFCoordinatorV2Interface.sol";import"@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";contractVRFD20isVRFCons

Contract variables

This example is adapted for [Sepolia testnet](#) but you can change the configuration and make it run for any [supported network](#).

```
uint64s_subscriptionId;addresss_owner;VRFCoordinatorV2Interface
COORDINATOR;addressvrfCoordinator=0x8103B0A8A00be2DDC778e6e7eaa21791Cd364625;bytes32s_keyHash=0x474e34a077df58807dbe9c96d3c009b23b3c6d0cce433e59bbf5b34f823bc56c;uint
* uint64 s_subscriptionId: The subscription ID that this contract uses for funding requests. Initialized in the constructor. * address s_owner: The address of the owner of the contract that you will deploy.
This is initialized in the constructor, and it will be the address you use when deploying the contract. * VRFCoordinatorV2Interface COORDINATOR: The address of the Chainlink VRF Coordinator
contract that this contract will use. Initialized in the constructor. * address vrfCoordinator: The address of the Chainlink VRF Coordinator contract. * bytes32 s_keyHash: The gas lane key hash value,
which is the maximum gas price you are willing to pay for a request in wei. It functions as an ID of the offchain VRF job that runs in response to requests. * uint32 callbackGasLimit: The limit for how
much gas to use for the callback request to your contract's fulfillRandomWords function. It must be less than the maxGasLimit on the coordinator contract. Adjust this value for larger requests depending
on how your fulfillRandomWords function processes and stores the received random values. If your callbackGasLimit is not sufficient, the callback will fail and your subscription is still charged for the work
done to generate your requested random values. * uint16 requestConfirmations: How many confirmations the Chainlink node should wait before responding. The longer the node waits, the more
secure the random value is. It must be greater than the minimumRequestBlockConfirmations limit on the coordinator contract. * uint32 numWords: How many random values to request. If you can use
several random values in a single callback, you can reduce the amount of gas that you spend per random value. In this example, each transaction requests one random value.
```

To keep track of addresses that roll the dice, the contract uses mappings. [Mappings](#) are unique key-value pair data structures similar to hash tables in Java.

mapping(uint256=>address)private s_rollers; mapping(address=>uint256)private s_results; * s_rollers stores a mapping between the requestID (returned when a request is made), and the address of the roller. This is so the contract can keep track of who to assign the result to when it comes back. * s_results stores the roller and the result of the dice roll.

Initializing the contract

The coordinator and subscription id must be initialized in the constructor of the contract. To use VRFConsumerBaseV2 properly, you must also pass the VRF coordinator address into its constructor. The address that creates the smart contract is the owner of the contract. the modifier onlyOwner() checks that only the owner is allowed to do some tasks.

// SPDX-License-Identifier:

```
MITpragmasolidity"0.8.7;import"@chainlink/contracts/src/v0.8/interfaces/VRFCoordinatorV2Interface.sol";import"@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";contractVRFD20isVRFCons
variables// ...// constructorconstructor(uint64subscriptionId)VRFConsumerBaseV2(vrfCoordinator)
{COORDINATOR=VRFCoordinatorV2Interface(vrfCoordinator);s_owner=msg.sender;s_subscriptionId=subscriptionId;}//...modifieronlyOwner(){require(msg.sender==s_owner);_;}}
```

rollDice function

The rollDice function will complete the following tasks:

- Check if the roller has already rolled since each roller can only ever be assigned to a single house.
- Request randomness by calling the VRF coordinator.
- Store the request id and roller address.
- Emit an event to signal that the dice is rolling.

You must add a ROLL_IN_PROGRESS constant to signify that the dice has been rolled but the result is not yet returned. Also add a DiceRolled event to the contract.

Only the owner of the contract can execute the rollDice function.

```
// SPDX-License-Identifier:
MIT
pragma solidity ^0.8.7;
import "@chainlink/contracts/src/v0.8/interfaces/VRFCoordinatorV2Interface.sol";
import "@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";
contract VRFD20 is VRFConsumerBaseV2 {
    uint256 private constant ROLL_IN_PROGRESS = 42;
    event DiceRolled(uint256 indexed requestId, address indexed roller);
    constructor() {}
    rollDice function() public onlyOwner returns (uint256 requestId) {
        require(s_results[roller] == 0, "Already rolled");
        // Will revert if subscription is not set and funded.
        requestId = COORDINATOR.requestRandomWords(s_keyHash, s_subscriptionId, requestConfirmations, callbackGasLimit, numWords);
        s_rollers[requestId] = roller;
        s_results[roller] = ROLL_IN_PROGRESS;
    }
}
```

fulfillRandomWords function

fulfillRandomWords is a special function defined within the VRFConsumerBaseV2 contract that our contract extends from. The coordinator sends the result of our generated randomWords back to fulfillRandomWords. You will implement some functionality here to deal with the result:

1. Change the result to a number between 1 and 20 inclusively. Note that randomWords is an array that could contain several random values. In this example, request 1 random value.
2. Assign the transformed value to the address in the s_results mapping variable.
3. Emit a DiceLanded event.

```
// SPDX-License-Identifier:
MIT
pragma solidity ^0.8.7;
import "@chainlink/contracts/src/v0.8/interfaces/VRFCoordinatorV2Interface.sol";
import "@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";
contract VRFD20 is VRFConsumerBaseV2 {
    // variables
    // events
    event DiceLanded(uint256 indexed requestId, uint256 indexed result);
    constructor() {}
    rollDice function() public {
        // fulfillRandomWords function
        // house
        function house(address player) public view returns (string memory) {
            // dice has not yet been rolled to this address
            require(s_results[player] != 0, "Dice not rolled");
            // not waiting for the result of a thrown dice
            require(s_results[player] == ROLL_IN_PROGRESS, "Roll in progress");
            // returns the house name from the name list
            function returnHouseName(uint256 d) private pure returns (string memory) {
                // array storing the list of house's names
                string[20] memory houseNames = ["Targaryen", "Lannister", "Stark", "Tyrell", "Baratheon", "Martell", "Tully", "Bolton", "Greyjoy", "Arryn", "Frey", "Mormont", "Tarley", "Dayne", "Umber", "Valeryon", "Manderly", "Clegane", "Glover", "Karstark"];
                // returns the house name given an index
                return houseNames[d-1];
            }
            // Open in Remix What is Remix? You have now completed all necessary functions to generate randomness and assign the user a Game of Thrones house. We've added a few helper functions in there to make using the contract easier and more flexible. You can deploy and interact with the complete contract in Remix.
        }
    }
}
```

house function

Finally, the house function returns the house of an address.

To have a list of the house's names, create the getHouseName function that is called in the house function.

```
// SPDX-License-Identifier:
MIT
pragma solidity ^0.8.7;
import "@chainlink/contracts/src/v0.8/interfaces/VRFCoordinatorV2Interface.sol";
import "@chainlink/contracts/src/v0.8/vrf/VRFConsumerBaseV2.sol";
contract VRFD20 is VRFConsumerBaseV2 {
    // variables
    // events
    event DiceLanded(uint256 indexed requestId, uint256 indexed result);
    constructor() {}
    rollDice function() public {
        // fulfillRandomWords function
        // house
        function house(address player) public view returns (string memory) {
            // dice has not yet been rolled to this address
            require(s_results[player] != 0, "Dice not rolled");
            // not waiting for the result of a thrown dice
            require(s_results[player] == ROLL_IN_PROGRESS, "Roll in progress");
            // returns the house name from the name list
            function returnHouseName(uint256 d) private pure returns (string memory) {
                // array storing the list of house's names
                string[20] memory houseNames = ["Targaryen", "Lannister", "Stark", "Tyrell", "Baratheon", "Martell", "Tully", "Bolton", "Greyjoy", "Arryn", "Frey", "Mormont", "Tarley", "Dayne", "Umber", "Valeryon", "Manderly", "Clegane", "Glover", "Karstark"];
                // returns the house name given an index
                return houseNames[d-1];
            }
            // Open in Remix What is Remix? You have now completed all necessary functions to generate randomness and assign the user a Game of Thrones house. We've added a few helper functions in there to make using the contract easier and more flexible. You can deploy and interact with the complete contract in Remix.
        }
    }
}
```

How do I deploy to testnet?

You will deploy this contract on the Sepolia test network. You must have some Sepolia testnet ETH in your MetaMask account to pay for the gas. Testnet ETH is also available from several [faucets](#).

This deployment is slightly different than the example in the [Deploy Your First Contract](#) guide. In this case, you pass in parameters to the constructor upon deployment.

Once compiled, you'll see a dropdown menu that looks like this in the deploy pane:

Select the VRFD20 contract or the name that you gave to your contract.

Click the caret arrow on the right hand side of Deploy to expand the parameter fields, and paste your subscription ID.

Then click the Deploy button and use your MetaMask account to confirm the transaction.

Address, Key Hashes and more

For a full reference of the addresses, key hashes and fees for each network, see [RF Supported Networks](#).

At this point, your contract should be successfully deployed. However, it can't request anything because it is not yet approved to use the LINK balance in your subscription. If you click rollDice, the transaction will revert.

How do I add my contract to my subscription account?

After you deploy your contract, you must add it as an approved consumer contract so it can use the subscription balance when requesting for randomness. Go to the [Subscription Manager](#) and add your deployed contract address to the list of consumers. Find your contract address in Remix under Deployed Contract on the bottom left.

How do I test rollDice?

After you open the deployed contract tab in the bottom left, the function buttons are available. Find rollDice and click the caret to expand the parameter fields. Enter an Ethereum address to specify a "dice roller", and click 'rollDice'.

It takes a few minutes for the transaction to confirm and the response to be sent back. You can get your house by clicking the house function button with the address passed in rollDice. After the response is sent back, you'll be assigned a Game of Thrones house!

Further Reading

To read more about generating random numbers in Solidity, read our blog posts:

- [35+ Blockchain RNG Use Cases Enabled by Chainlink VRF](#)
- [How to Build Dynamic NFTs on Polygon](#)
- [Chainlink VRF v2 Now Live on Ethereum Mainnet](#)