**Chainlink**

Chainlink expands the capabilities of smart contracts by enabling access to **real-world data and off-chain computation** while maintaining the **security and reliability** guarantees inherent to blockchain technology.

**Overview**

In general, you **create and deploy your smart contracts** operate using the following process:

1. **Write:** Write a contract to define *how the contract functions, what data it can store, what other contracts it interacts with, and what external APIs it might call.*
2. **Compile:** Pass your smart contract code through a compiler to *translate the contract into bytecode that the blockchain can understand*.
3. **Deploy:** Send the compiled smart contract to the blockchain. From that point forward, the *contract cannot be altered.*
4. **Run functions:** When you run the functions that you defined for the contract, the *network processes those functions and modifies the state of your contract.*

**Install and Fund MetaMask**

1. [**Install and configure** the MetaMask extension](https://metamask.io/download) in the browser.
2. Open your browser extension list and click MetaMask to **open MetaMask.**
3. Set MetaMask to use the **Kovan test network.**
4. Go to the [**Kovan Chainlink Faucet**](https://faucets.chain.link/kovan/?_ga=2.120209992.1123431018.1652161415-147459993.1652161415) and follow the steps to send 0.1 test ETH.([Faucets | Chainlink](https://faucets.chain.link/))

## **Write, compile, and deploy your first smart contract**

1. [Open the contract](https://remix.ethereum.org/#url=https://docs.chain.link/samples/Tutorials/HelloWorld.sol) in the **Remix IDE.**
2. On the left side of Remix, click the **Solidity Compiler tab** to view the compiler settings.
3. Click the **Compile**. (converts the contract from Solidity into bytecode)
4. On the left side of Remix, click the **Deploy and Run** tab to view the deployment settings.
5. In the deployment settings, select the **Injected Web3 environment.** This tells Remix that you want to deploy your contract to the blockchain that you configured in **MetaMask.**
6. Next to the **Deploy button**, enter a **message that you want to send** with the smart contract when you deploy it.
7. Click the **Deploy button** to deploy the **contract and its initial message** to the blockchain network. **MetaMask** opens and asks you to **confirm payment** to deploy the contract.
8. In the MetaMask prompt, click **Confirm to approve the transaction** and spend your testnet ETH.
9. After a few seconds, the transaction completes and your contract appears under the **Deployed Contracts list in Remix.** Click the contract dropdown to **view its variables and functions.**
10. To see details about your deployed contract, copy the **contract address** from the list in Remix and search for it in the [**Etherscan Kovan Testnet Explorer**](https://kovan.etherscan.io/)**.**

## **Run functions in your contract**

The **contract, its variables, and its functions** remain in the blockchain permanently. To **change the message variable** that is stored with your contract, run the **updateMessage function.**

1. In your deployed contract, **enter a new message** next to the **updateMessage function.**
2. **Click the updateMessage button** to set the new message in the contract data. MetaMask opens and asks you to **confirm payment to update the state** of your contract.
3. In the new MetaMask prompt, **click Confirm to approve the transaction.**
4. **Click the message variable** again to see the updated value. It might take a few seconds before the transaction updates the variable.

# **Consuming Data Feeds**

When you connect **a smart contract to real-world services or off-chain data**, you create a **hybrid smart contract.**

For example, you can use **Chainlink Data Feeds** to connect your smart contracts to asset pricing data like the [ETH / USD feed](https://feeds.chain.link/eth-usd).

These **data feeds use the data aggregated** from many independent Chainlink node operators.

Each **price feed has an on-chain address and functions** that enable contracts to read pricing data from that address.

## 

## **The sample contract**

## import "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";

* The **import line imports** an interface named **AggregatorV3Interface**.
  + Interfaces **define functions without their implementation**, which leaves **inheriting contracts** to define the actual implementation themselves.
  + In this case, **AggregatorV3Interface** defines that all **v3 Aggregators** have the **function latestRoundData.**

constructor() {

priceFeed = AggregatorV3Interface(0x9326BFA02ADD2366b30bacB125260Af641031331);

}

* The **constructor() {}**
  + initializes an **interface object named priceFeed**
  + **priceFeed** that uses **AggregatorV3Interface** and connects specifically to a **proxy aggregator contract**
  + **proxy aggregator contract** that is already deployed at 0x9326BFA02ADD2366b30bacB125260Af641031331.
  + The interface allows your **contract to run functions** on that deployed aggregator contract.

function getLatestPrice() public view returns (int) {

(

/\*uint80 roundID\*/,

int price,

/\*uint startedAt\*/,

/\*uint timeStamp\*/,

/\*uint80 answeredInRound\*/

) = priceFeed.latestRoundData();

return price;

* The **getLatestPrice() function** calls **priceFeed** object and runs the **latestRoundData()** function.
  + When you **deploy the contract**, it **initializes** the **priceFeed object** to point to the aggregator at 0x9326BFA02ADD2366b30bacB125260Af641031331, which is the **proxy address for the Kovan ETH / USD data feed.**
  + The contract connects to **that address and executes the function.**
  + The aggregator connects with several **oracle nodes and aggregates the pricing data from those nodes.**
  + The **response from the aggregator includes several variables**, but **getLatestPrice() returns only the price variable.**

## **Compile, deploy, and run the contract**

1. [Open the example contract](https://remix.ethereum.org/#url=https://docs.chain.link/samples/PriceFeeds/PriceConsumerV3.sol) in Remix
2. click the **Solidity Compiler** tab
3. Click the **Compile**
4. On the **Deploy** tab, select the **Injected Web3** environment.
5. In the **Contract** section, select the PriceConsumerV3 contract to make sure that Remix deploys the correct contract.
6. Click **Deploy** to deploy the contract to the Kovan testnet.
7. In the MetaMask prompt, click **Confirm** to approve the transaction and spend your testnet ETH required to deploy the contract.
8. After a few seconds, the transaction completes and your contract appears under the **Deployed Contracts** list in Remix.
9. Click **getLatestPrice** to show the latest price from the aggregator contract.

# **Random Numbers: Using Chainlink VRF**

# **1. 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**](https://docs.chain.link/docs/chainlink-vrf/)**, also known as Chainlink Verifiable Random Function.**

## **2. What is the Request and Receive cycle?**

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**](https://docs.chain.link/docs/architecture-request-model/)**.**

## **3. What is the payment process for generating a random number?**

VRF requests **receive funding from subscription accounts.**

The [**Subscription Manager**](https://vrf.chain.link/) 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.

## **4. How can I use Chainlink VRF?**

The **contract** will request **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.

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

The contract will have 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**

Chainlink VRF requests **receive funding from subscription accounts.**

The [Subscription Manager](https://vrf.chain.link/?_ga=2.47227687.1201123188.1652335176-80528070.1652335176) lets you **create an account and pre-pay your use of Chainlink VRF requests.**

### **Importing VRFConsumerBaseV2 and VRFCoordinatorV2Interface**

Chainlink maintains **a** [**library of contracts**](https://github.com/smartcontractkit/chainlink/tree/master/contracts) that make consuming data from oracles easier. For Chainlink VRF, you will use:

* [**VRFConsumerBaseV2**](https://github.com/smartcontractkit/chainlink/blob/master/contracts/src/v0.8/VRFConsumerBaseV2.sol)that must be **imported** and extended from the contract that you create.
* [**VRFCoordinatorV2Interface**](https://github.com/smartcontractkit/chainlink/blob/master/contracts/src/v0.8/interfaces/VRFCoordinatorV2Interface.sol) that must be **imported to communicate** with the VRF coordinator.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

**import "@chainlink/contracts/src/v0.8/interfaces/VRFCoordinatorV2Interface.sol";**

**import "@chainlink/contracts/src/v0.8/VRFConsumerBaseV2.sol";**

contract VRFD20 is VRFConsumerBaseV2 {

}

**Contract Variables**

uint64 s\_subscriptionId;

The subscription ID that this contract uses for **funding requests.** Initialized in the constructor.

address vrfCoordinator = 0x6168499c0cFfCaCD319c818142124B7A15E857ab;

The address of the **Chainlink VRF Coordinator** contract.

bytes32 s\_keyHash = 0xd89b2bf150e3b9e13446986e571fb9cab24b13cea0a43ea20a6049a85cc807cc;

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 off-chain VRF job** that runs in response to requests.

uint32 callbackGasLimit = 40000;

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.

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 = 3;

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 = 1;

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.**

**mapping(uint256 => address) private s\_rollers;**

**mapping(address => uint256) private s\_results;**

To keep track of **addresses that roll the dice, the contract uses mappings**. [Mappings](https://medium.com/upstate-interactive/mappings-in-solidity-explained-in-under-two-minutes-ecba88aff96e) are **unique key-value pair data structures** similar to hash tables in Java.

* **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**

### **rollDice function**

The rollDice function will complete the following tasks:

1. Check if the **roller has already rolled since each roller can only ever be assigned to a single house.**
2. **Request randomness by calling the VRF coordinator.**
3. **Store the requestId and roller address.**
4. **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.

**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.**

### **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.

**API Calls: Using Any API**

## **1. How does the request and receive cycle work for API calls?**

The **request and receive cycle** describes how a smart contract **requests data from an oracle and receives the response in a separate transaction.**

For contracts that use [**Chainlink VRF**](https://docs.chain.link/docs/chainlink-vrf/)**, you request randomness from a VRF oracle** and then await the response. The **fulfillment function** is already given to us from the **VRFConsumerBase contract**, so oracles already know where to send the response to. However, with API calls, the **contract itself defines** which function it wants to receive the response to.

## **2. What are jobs?**

Chainlink nodes require [**Jobs**](https://docs.chain.link/docs/jobs/) to do anything useful.

In the case of a **Request and Receive job, the** [**Direct Request**](https://docs.chain.link/docs/jobs/types/direct-request/) **job** monitors the blockchain for a request from a smart contract.

Once it catches a request, it **runs the tasks** (both core and external adapters) that the **job is configured** to run and eventually **returns the response to the requesting contract.**

## **3. What are Tasks?**

Each **oracle job has a configured set of tasks** it must complete when it is run. These tasks are split into two subcategories:

* [**Tasks**](https://docs.chain.link/docs/tasks/) - These are tasks that come **built-in to each node.** (examples: http, ethabidecode, etc).
* [**External Adapters**](https://docs.chain.link/docs/external-adapters/) - These are **custom adapters built by node operators and community members**, which perform specific tasks like calling a particular endpoint with a specific set of parameters (like authentication secrets that shouldn't be publicly visible).

For example

If a job needs to make a **GET request to an API**, find a specific unsigned integer field in a JSON response, then submit that back to the requesting contract, it would need **a job containing the following Tasks:**

* [**HTTP**](https://docs.chain.link/docs/jobs/task-types/http/) **calls the API.** the method must be set to **GET**.
* [**JSON Parse**](https://docs.chain.link/docs/jobs/task-types/jsonparse/)parses the **JSON and extracts a value** at a given keypath.
* [**ETH ABI Encode**](https://docs.chain.link/docs/jobs/task-types/eth-abi-encode/) converts the **data to a bytes payload** according to ETH ABI encoding.
* [**ETH Tx**](https://docs.chain.link/docs/jobs/task-types/eth-tx/)submits the **transaction to the chain,** completing the cycle.

### **External Adapters**

Here are some examples of external adapters:

1. **Real-world events:** [Google Weather Data](https://market.link/adapters/654b3417-c381-4764-8cb7-4e8d552a9d95), [Associated Press](https://market.link/adapters/cee0cc56-43a2-40e1-9014-3905c2534caa)
2. **Social media proofs:** [MUBC Retweet Verifier](https://market.link/adapters/8fba59b1-f639-4d0a-9b01-7adff9b74442?network=1)
3. **Cryptocurrency aggregators:** [CoinAPI](https://market.link/adapters/f4d69a08-f3dd-46e7-b4b8-3d2fb620c033?network=1)

If all the **parameters are defined within the Oracle job,** the only things a **smart contract needs to define to consume are:**

* **JobId**
* **Oracle address**
* **LINK fee**
* **Fulfillment function**

## **4. How can I use Tasks in my own contract?**

1. To consume an API response, your **contract should inherit from ChainlinkClient.** This contract exposes **a struct called Chainlink.Request** that your contract should use to build the API request.
2. The request should include the **oracle address, the job id, the fee, adapter parameters, and the callback function signature.** Create variables for these items using the correct data types.
3. In the constructor, **set up the contract with the Oracle address, Job ID, and LINK fee** that the oracle charges for the job.
4. **Define the function**
5. The last component of your contract should be the **fulfill function.** This is where the sports data is sent upon the **Oracle Job's completion.**

# 

# 