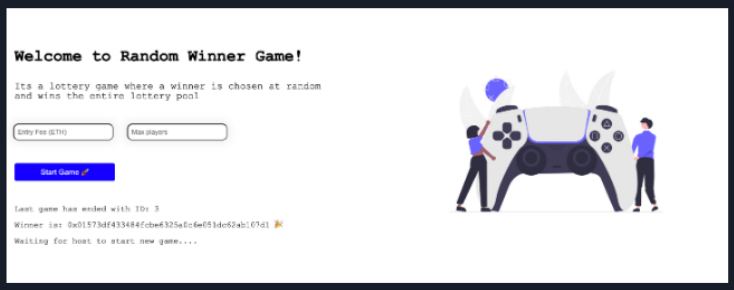
Lesson Type: Practical | Estimated Time: 4-8 hours | Current Score: 0%

The Graph

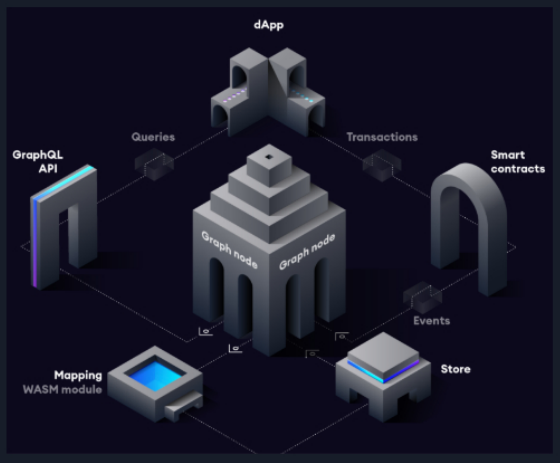


The Graph is a decentralized query protocol and indexing service for the blockchain. It allows developers to easily track events being emitted from smart contracts on various networks, and write custom data transformation scripts, which are run in real time. The data is also made available through a simple GraphQL API which developers can then use to display things on their frontends.

Prerequisites

- We will be using yarn which is a package manager just like npm.
- Please install yarn from here If your computer doesn't have yarn already installed
- Please watch this 40 minute tutorial on GraphQL
- If you dont know what axios is, Watch this short tutorial
- You should have completed the Chainlink VRF tutorial

How it Works



1. A dApp sends a transaction and some data gets stored in the smart contract. 2/ This smart contract then emits one or more events.

2. Graph's node keeps scanning Ethereum for new blocks and the data for your subgraph that these blocks may contain.

3. If the node finds an event you were looking for and defined in your subgraph, it runs the data transformation scripts (mappings) you defined. The mapping is a WASM (Web assembly) module that creates or updates data `Entities` on the Graph Nodes in response to the event.

4. We can query the Graph's node for this data using the GraphQL Endpoint

(Referenced From The Graph's website)

Build

We will be using the folder named `RandomWinnerGame` that you created in the `Chainlink VRF` tutorial

Create an abi.json file inside your `RandomWinnerGame` folder (You will need this file to intilialize your graph) and copy the following **content**.

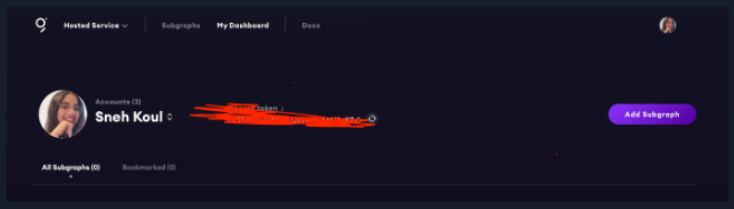
Note this is the abi of the `RandomWinnerGame` contract you created in the Chainlink VRF tutorial

So your final folder structure should look like this:

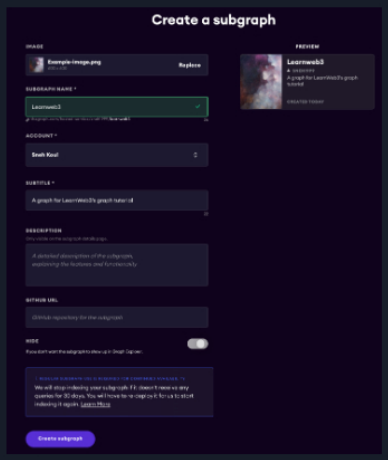
```
RandomWinnerGame
    - hardhat-tutorial
    - abi.json
```

To create your subgraph you will need to go to **The Graph's Hosted Service**

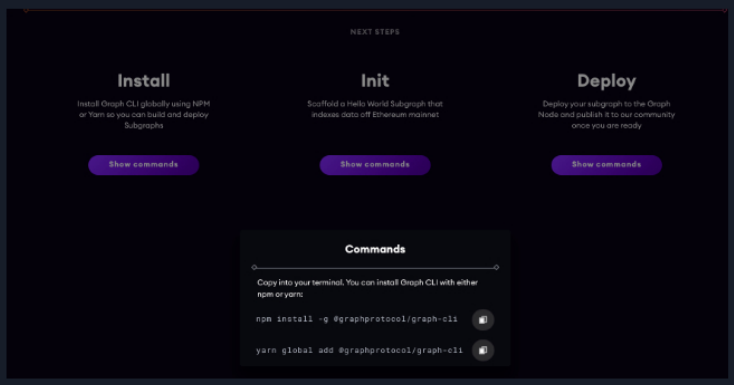
Login using your GitHub account and visit `My Dashboard` tab



Click on `Add Subgraph`, fill in the information and create the subgraph



When your subgraph is created, it will show you some commands in `Install`, `Init` and `Deploy`



In your terminal execute this command pointing to `RandomWinnerGame` folder:

```
yarn global add @graphprotocol/graph-cli
```

If this doesn't work for you, you can try using `npm` `npm install -g @graphprotocol/graph-cli`

If this doesn't work for you, you can try using `npm`: `npm install -g @graphprotocol/graph-cli`

After that execute this command but replace `GITHUB_USERNAME` with your Github username and `YOUR_RANDOM_WINNER_GAME_CONTRACT_ADDRESS` with the address of the RandomWinnerGame contract that you deployed in your Chainlink VRF tutorial. Press enter for all the questions after that :)

```
graph init --contract-name RandomWinnerGame --product hosted-service GITHUB_USERNAME/Learnweb3  --from-contract YOUR_I
```

```
→  RandomWinnerGame git:(master) ✗ graph init --contract-name RandomWinnerGame --product hosted-service Sneh1999/Learnweb3  --from-contract 0
xB89EF69261272Caa27f0655D020BbAc7055EDAD5  --abi ./abi.json --network mumbai graph
✔ Protocol · ethereum
✔ Subgraph name · Sneh1999/Learnweb3
✔ Directory to create the subgraph in · graph
✔ Ethereum network · mumbai
✔ Contract address · 0xB89EF69261272Caa27f0655D020BbAc7055EDAD5
✔ ABI file (path) · ./abi.json
✔ Contract Name · RandomWinnerGame
```

For the deploy key, go to **The Graph's Hosted Service**, click on `My Dashboard`, copy the `Access Token` and paste it for the `Deploy Key`

```
graph auth
```

```
→  RandomWinnerGame git:(master) ✗ graph auth
✔ Product for which to initialize · hosted-service
✔ Deploy key · ********************************
Deploy key set for https://api.thegraph.com/deploy/
```

Now the last two commands to execute are:

```
cd graph
yarn deploy
# or `npm run deploy`
```

You can go back to **The Graph's Hosted Service** and click on `My Dashboard` you will be able to see your graph because its deployed now 🚀 👀

```
→  graph git:(master) ✗ yarn deploy
yarn run v1.22.10
$ graph deploy --node https://api.thegraph.com/deploy/ Sneh1999/Learnweb3
  Skip migration: Bump mapping apiVersion from 0.0.1 to 0.0.2
  Skip migration: Bump mapping apiVersion from 0.0.2 to 0.0.3
  Skip migration: Bump mapping apiVersion from 0.0.3 to 0.0.4
  Skip migration: Bump mapping apiVersion from 0.0.4 to 0.0.5
  Skip migration: Bump mapping specVersion from 0.0.1 to 0.0.2
✔ Apply migrations
✔ Load subgraph from subgraph.yaml
  Compile data source: RandomWinnerGame => build/RandomWinnerGame/RandomWinnerGame.wasm
✔ Compile subgraph
  Copy schema file build/schema.graphql
  Write subgraph file build/RandomWinnerGame/abis/RandomWinnerGame.json
  Write subgraph manifest build/subgraph.yaml
✔ Write compiled subgraph to build/
  Add file to IPFS build/schema.graphql
                .. QmbF2x1zKsaYxSVJXF1TWWpqAqG1iWQMdMP1xZvRqzbuty
  Add file to IPFS build/RandomWinnerGame/abis/RandomWinnerGame.json
                .. QmfTiaDrrrBD5gvTMG2n8kUUqppRFeybZQbJBDgFeMKQJh
  Add file to IPFS build/RandomWinnerGame/RandomWinnerGame.wasm
                .. QmXkvXqL435tm78LBvQBxn4aBGcwNUpY3DEhLqeK27pQSC
✔ Upload subgraph to IPFS

Build completed: QmeEPXJUzZUZDHzUjJVWLBdZUMjUxB4GZk7wikfaX9pcc7

Deployed to https://thegraph.com/explorer/subgraph/Sneh1999/Learnweb3

Subgraph endpoints:
Queries (HTTP):     https://api.thegraph.com/subgraphs/name/sneh1999/learnweb3
Subscriptions (WS): wss://api.thegraph.com/subgraphs/name/sneh1999/learnweb3

✨  Done in 11.93s.
```

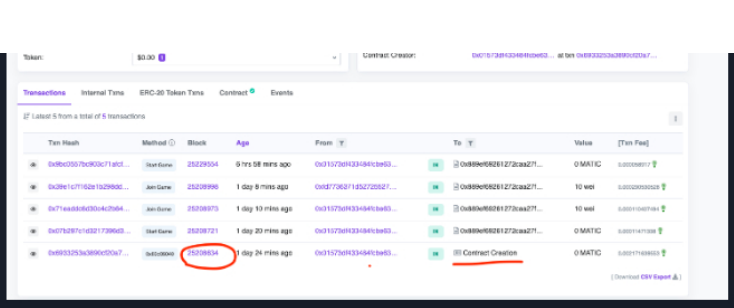
Boom you have deployed your first graph!!!

Now comes the fun part where we will modify the default code provided to us by The Graph into the code which can help us track events for our contract.

Lets get started

Open up you `subgraph.yaml` inside the `graph` folder and add a `startBlock` to the yaml file after the `abi: RamdomWinnerGame` line, to get the startBlock you will need to go to **Mumbai PolygonScan** and search up your contract address, after that you will need to copy the block number of the block in which your contract was deployed

The start block doesn't come with the default settings but because we know that we only need to track the events from the block the contract was deployed, we will not need to sync the entire blockchain but only the part after the contract was deployed for tracking the events



```
source:
  address: "0x889Ef69261272Caa27f0655D0208bAc7055EDAD5"
  abi: RandomWinnerGame
  startBlock: BLOCK_NUMBER
```

Your final file should look something like **this**

Okay now its time to create some `Entities`. `Entities` are objects which define the structure for how your data will be stored on `The Graph's nodes`. If you want to read more about them, click on this **link**

We will need an `Entity` which can cover all the variables we have in our events so that we can keep track of all of them. Open up for `schema.graphql` file and replace the already existing lines of code with the following lines of code:

```
type Game @entity {
  id: ID!
  maxPlayers: Int!
  entryFee: BigInt!
  winner: Bytes
  requestId: Bytes
  players: [Bytes!]!
}
```

- `ID` is the unique identifier for a game and will be equivalent to the `gameId` variable we have in our contract.
- `maxPlayers` will keep track of how many max players are allowed in this game.
- `entryFee` is the fees to enter into the game and it is a BigInt because we have `entryFee` as a `uint256` in our contract which is a BigNumber
- `winner` is the address of the winner in the game and is defined as Bytes because address is a hexadecimal string
- `requestId` is also a hexadecimal string and thus is defined as `Bytes`
- `players` is the list of addresses for the players in the game and because each address is a hexadecimal string, we have signified players as a Bytes array
- Also note that `!` indicates required variables, we have `maxPlayers`, `entryFee`, `players` and `id` marked as required because when the `Game` initially starts it will emit the `GameStarted` event which will emit these three variables(`maxPlayers`, `entryFee` and `id`) so a `Game` entity can never be created without these three variables and for the players array, it will be initialized to an empty array by us.
- `winner` and `requestId` will come with `GameEnded` event and `players` will keep track of every `player address` which is emitted by the `PlayerJoined` event

If you want to learn more about the types you can visit this **link**

Okay, so now we have let the graph know what kind of data we will be tracking and what will it contain 🥳 🥳 🥳

Now its time to query this data 🎉

Graph has an amazing functionality that given the `Entity` it can auto generate large chunk of code for you!!!

Isn't that amazing? Lets use that functionality. In your terminal pointing to the graph directory execute this command

```
yarn codegen
```

After this `The Graph` would have created most of the code for you expect of the mappings. If you look at the file inside `src` named `random-winner-game.ts`, the Graph CLI would have created for you some functions each pointing to one of the events that you created in your contract. These functions get called everytime the Graph finds an event relating to these functions. We will add some code to these functions so that we can store the data when an event comes in.

Replace the `random-winner-game.ts` file content with following lines

```ts
import { BigInt } from "@graphprotocol/graph-ts";
import {
  PlayerJoined,
  GameEnded,
  GameStarted,
  OwnershipTransferred,
} from "../generated/RandomWinnerGame/RandomWinnerGame";
import { Game } from "../generated/schema";

export function handleGameEnded(event: GameEnded): void {
  // Entities can be loaded from the store using a string ID; this ID
  // needs to be unique across all entities of the same type
  let entity = Game.load(event.params.gameId.toString());
  // Entities only exist after they have been saved to the store;
  // `null` checks allow to create entities on demand
  if (!entity) {
    return;
  }
  // Entity fields can be set based on event parameters
  entity.winner = event.params.winner;
  entity.requestId = event.params.requestId;
  // Entities can be written to the store with `.save()`
  entity.save();
}

export function handlePlayerJoined(event: PlayerJoined): void {
  // Entities can be loaded from the store using a string ID; this ID
  // needs to be unique across all entities of the same type
  let entity = Game.load(event.params.gameId.toString());
  // Entities only exist after they have been saved to the store;
  // `null` checks allow to create entities on demand
  if (!entity) {
    return;
  }
  // Entity fields can be set based on event parameters
  let newPlayers = entity.players;
  newPlayers.push(event.params.player);
  entity.players = newPlayers;
  // Entities can be written to the store with `.save()`
  entity.save();
}

export function handleGameStarted(event: GameStarted): void {
  // Entities can be loaded from the store using a string ID; this ID
  // needs to be unique across all entities of the same type
  let entity = Game.load(event.params.gameId.toString());
  // Entities only exist after they have been saved to the store;
  // `null` checks allow to create entities on demand
  if (!entity) {
    entity = new Game(event.params.gameId.toString());
    entity.players = [];
  }
  // Entity fields can be set based on event parameters
  entity.maxPlayers = event.params.maxPlayers;
  entity.entryFee = event.params.entryFee;
  // Entities can be written to the store with `.save()`
  entity.save();
}

export function handleOwnershipTransferred(event: OwnershipTransferred): void {}
```

Lets understand what is happening in `handleGameEnded` function

- it takes in the `GameEnded` event and expects `void` to be returned which means nothing gets returned from the function
- It loads a `Game` object from `Graph's` db which has an ID equal to the `gameId` that is present in the event that Graph detected
- If an entity with the given `id` doesn't exist return from the function and dont do anything
- If it exists, set the winner and the requestId from the event into our Game Object(Note `GameEnded` event has the winner and requestId)
- Then save this updated Game Object back to the `Graph's DB`
- For each game there will be a unique `Game` object which will have a unique `gameId`

Now lets see what's happening in the `handlePlayerJoined`

-

It loads a `Game` object from `Graph's` db which has an ID equal to the `gameId` that is present in the event that Graph detected

-

If an entity with the given `id` doesn't exist, return from the function and dont do anything

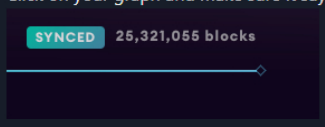
- To actually update the player's array, we need to reassign the property on the entity that contains the array (i.e. players) similar to how we assign values to other properties on the entity (e.g. maxPlayers). Therefore, we need to create a temporary array which contains all of the existing entity.players elements, push to that, and reassign entity.players to be equal to the new array.
- Then save this updated Game Object back to the `Graph's DB`
- Now lets see what's happening in the `handleGameStarted`
  - It loads a `Game` object from `Graph's` db which has an ID equal to the `gameId` that is present in the event that Graph detected
  - If an entity like that doesn't exist create a new one, also initialize the players array
  - Then set the maxPlayer and the entryFee from the event into our Game Object
  - Save this updated Game Object back to the `Graph's DB`

You can now go to your terminal pointing to the `graph` folder and execute the following command:

```
yarn deploy
# or `npm run deploy`
```

After deploying, **The Graph's Hosted Service** and click on `My Dashboard` you will be able to see your graph.

Click on your graph and make sure it says synced, if not wait for it to get synched before proceeding



Website

To develop the website we would be using **React** and **Next Js**. React is a javascript framework which is used to make websites and Next Js is built on top of React.

First, You would need to create a new `next` app. Your folder structure should look something like

```
- RandomWinnerGame
      - graph
      - hardhat-tutorial
      - next-app
      - abi.json
```

To create this `next-app`, in the terminal point to RandomWinnerGame folder and type

```
npx create-next-app@latest
```

and press `enter` for all the questions

Now to run the app, execute these commands in the terminal

```
cd my-app
npm run dev
```

Now lets install Web3Modal library(https://github.com/Web3Modal/web3modal). Web3Modal is an easy-to-use library to help developers add support for multiple providers in their apps with a simple customizable configuration. By default Web3Modal Library supports injected providers like (Metamask, Dapper, Gnosis Safe, Frame, Web3 Browsers, etc), You can also easily configure the library to support Portis, Fortmatic, Squarelink, Torus, Authereum, D'CENT Wallet and Arkane. Open up a terminal pointing at `my-app` directory and execute this command

```
npm install web3modal
```

In the same terminal also install `ethers.js`

```
npm i ethers
```

We will also be using `axios` to send requests to the graph, so lets install that

```
npm i axios
```

In your public folder, download this **image** . Make sure that the name of the downloaded image is `randomWinner.png`

Now go to `styles` folder and replace all the contents of `Home.modules.css` file with the following code, this would add some styling to your dapp:

```
.main {
  min-height: 90vh;
  display: flex;
  flex-direction: row;
  justify-content: center;
  align-items: center;
  font-family: "Courier New", Courier, monospace;
}

.footer {
  display: flex;
  padding: 2rem 0;
  border-top: 1px solid #eaeaea;
  justify-content: center;
  align-items: center;
}

.input {
  width: 200px;
  height: 100%;
  padding: 1%;
  margin: 2%;
  box-shadow: 0 0 15px 4px rgba(0, 0, 0, 0.06);
  border-radius: 10px;
}

.image {
  width: 70%;
  height: 50%;
  margin-left: 20%;
}

.title {
  font-size: 2rem;
  margin: 2rem 1rem;
}

.description {
  line-height: 1;
  margin: 2rem 1rem;
  font-size: 1.2rem;
}

.log {
  line-height: 1rem;
  margin: 1rem 1rem;
  font-size: 1rem;
}

.button {
  border-radius: 4px;
  background-color: blue;
  border: none;
  color: #ffffff;
  font-size: 15px;
  padding: 8px;
  width: 200px;
  cursor: pointer;
  margin: 2rem 1rem;
}
@media (max-width: 1000px) {
  .main {
    width: 100%;
```

```
    flex-direction: column;
    justify-content: center;
    align-items: center;
  }
}
```

Lets now write some code to query the graph, create a new folder inside your `my-app` folder and name it `queries`. Inside the `queries` folder create a new file named `index.js` and paste the following lines of code:

```
export function FETCH_CREATED_GAME() {
  return `query {
        games(orderBy:id, orderDirection:desc, first: 1) {
            id
            maxPlayers
            entryFee
            winner
            players
        }
    }`;
}
```

As you can see we created a `GraphQL` query where we said we want a `game` object where data is ordered by Id(which is the gameId) in decending order and we want the first game from this ordered data

Lets simplify this with an example: Suppose you have three game objects stored inside the `The Graph's` node.

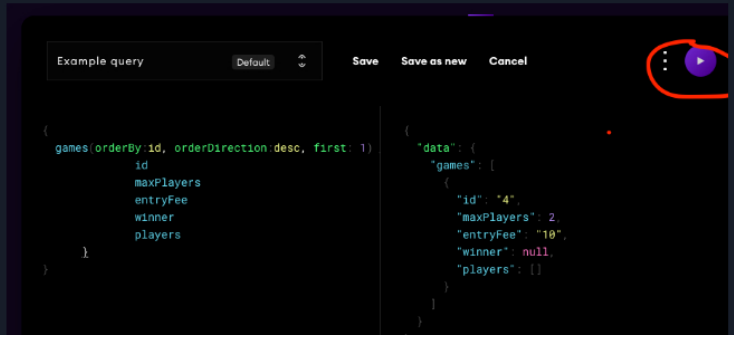
```
{
      "id": "1",
      "maxPlayers": 2,
      "entryFee": "10000000000000",
      "winner": "0xdb6eaffa95899b53b27086bd784f3bbfd58ff843"
    },
    {
      "id": "2",
      "maxPlayers": 2,
      "entryFee": "10000000000000",
      "winner": "0x01573df433484fcbe6325a0c6e051dc62ab107d1"
    },
    {
      "id": "3",
      "maxPlayers": 2,
      "entryFee": "100000000000000",
      "winner": "0x01573df433484fcbe6325a0c6e051dc62ab107d1"
    },
    {
      "id": "4",
      "maxPlayers": 2,
      "entryFee": "10",
      "winner": null
    }
```

Now you want the latest game every single time. To get the latest game, you will first have to order them by Id and then put this data in decending order so that gameId 4 can come at the top(It will be the current game) and then we say `first:1` because we only want the gameId 4 object, we dont care about the old games.

You can actually see this query working on the graph's hosted service. Lets try to do that:

I have already deployed a graph, lets look at my graph and use the query to query it, Go to **this link**

Replace the example query with our query and click on the purple play button



You can see some data coming up for my graph 👀 and the id of the game is 4

Simple right? Yes, indeed 😎

You can go to your graph through `My dashboard` and do the same things 🚀, it will be fun

Lets continue...

It was nice doing it through the already provided UI by `Graph` but to use this on our frontend we will need to write an axios post request so that we can get this data from the graph

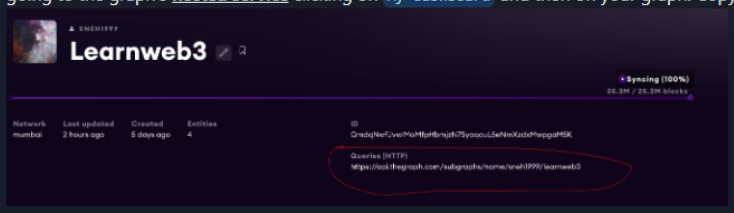
Create a new folder named `utils` and inside the folder create a new file named `index.js`. Copy the following lines of code inside the `index.js` file:

```
import axios from "axios";

export async function subgraphQuery(query) {
  try {
    // Replace YOUR-SUBGRAPH-URL with the url of your subgraph
    const SUBGRAPH_URL = "YOUR-SUBGRAPH-URL";
    const response = await axios.post(SUBGRAPH_URL, {
      query,
    });
    if (response.data.errors) {
      console.error(response.data.errors);
      throw new Error(`Error making subgraph query ${response.data.errors}`);
    }
    return response.data.data;
  } catch (error) {
    console.error(error);
    throw new Error(`Could not query the subgraph ${error.message}`);
  }
}
```

Lets try to understand what's happening in this function

It needs a `SUBGRAPH_URL`, you will need to replace "YOUR-SUBGRAPH-URL" with the url of your subgraph which you can get by going to the graph's **hosted service** clicking on `My dashboard` and then on your graph. Copy the url under `Queries(HTTP)`



It then calls this url with the query that we created using axios post request

Then it handles any errors and sends back the reponse if there are no errors

Now open your `index.js` file under the `pages` folder and paste the following code, explanation of the code can be found in the comments.

```
import { BigNumber, Contract, ethers, providers, utils } from "ethers";
import Head from "next/head";
import React, { useEffect, useRef, useState } from "react";
import Web3Modal from "web3modal";
import { abi, RANDOM_GAME_NFT_CONTRACT_ADDRESS } from "../constants";
import { FETCH_CREATED_GAME } from "../queries";
import styles from "../styles/Home.module.css";
import { subgraphQuery } from "../utils";

export default function Home() {
  const zero = BigNumber.from("0");
  // walletConnected keep track of whether the user's wallet is connected or not
  const [walletConnected, setWalletConnected] = useState(false);
  // loading is set to true when we are waiting for a transaction to get mined
  const [loading, setLoading] = useState(false);
  // boolean to keep track of whether the current connected account is owner or not
  const [isOwner, setIsOwner] = useState(false);
  // entryFee is the ether required to enter a game
```

```
  const [entryFee, setEntryFee] = useState(zero);
  // maxPlayers is the max number of players that can play the game
  const [maxPlayers, setMaxPlayers] = useState(0);
  // Checks if a game started or not
  const [gameStarted, setGameStarted] = useState(false);
  // Players that joined the game
  const [players, setPlayers] = useState([]);
  // Winner of the game
  const [winner, setWinner] = useState();
  // Keep a track of all the logs for a given game
  const [logs, setLogs] = useState([]);
  // Create a reference to the Web3 Modal (used for connecting to Metamask) which persists as long as the page is ope
  const web3ModalRef = useRef();

  // This is used to force react to re render the page when we want to
  // in our case we will use force update to show new logs
  const forceUpdate = React.useReducer(() => ({}), {})[1];

  /*
    connectWallet: Connects the MetaMask wallet
  */
  const connectWallet = async () => {
    try {
      // Get the provider from web3Modal, which in our case is MetaMask
      // When used for the first time, it prompts the user to connect their wallet
      await getProviderOrSigner();
      setWalletConnected(true);
    } catch (err) {
      console.error(err);
    }
  };

  /**
   * Returns a Provider or Signer object representing the Ethereum RPC with or without the
   * signing capabilities of metamask attached
   *
   * A `Provider` is needed to interact with the blockchain - reading transactions, reading balances, reading state, 
   *
   * A `Signer` is a special type of Provider used in case a `write` transaction needs to be made to the blockchain, 
   * needing to make a digital signature to authorize the transaction being sent. Metamask exposes a Signer API to al
   * request signatures from the user using Signer functions.
   *
   * @param {*} needSigner - True if you need the signer, default false otherwise
   */
  const getProviderOrSigner = async (needSigner = false) => {
    // Connect to Metamask
    // Since we store `web3Modal` as a reference, we need to access the `current` value to get access to the underlyi
    const provider = await web3ModalRef.current.connect();
    const web3Provider = new providers.Web3Provider(provider);

    // If user is not connected to the Mumbai network, let them know and throw an error
    const { chainId } = await web3Provider.getNetwork();
    if (chainId !== 80001) {
      window.alert("Change the network to Mumbai");
      throw new Error("Change network to Mumbai");
    }

    if (needSigner) {
      const signer = web3Provider.getSigner();
      return signer;
    }
    return web3Provider;
  };

  /**
   * startGame: Is called by the owner to start the game
   */
  const startGame = async () => {
    try {
      // Get the signer from web3Modal, which in our case is MetaMask
      // No need for the Signer here, as we are only reading state from the blockchain
      const signer = await getProviderOrSigner(true);
      // We connect to the Contract using a signer because we want the owner to
      // sign the transaction
      const randomGameNFTContract = new Contract(
        RANDOM_GAME_NFT_CONTRACT_ADDRESS,
        abi,
        signer
      );
      setLoading(true);
      // call the startGame function from the contract
      const tx = await randomGameNFTContract.startGame(maxPlayers, entryFee);
      await tx.wait();
      setLoading(false);
    } catch (err) {
```

```
    console.error(err);
    setLoading(false);
  }
};

/**
 * startGame: Is called by a player to join the game
 */
const joinGame = async () => {
  try {
    // Get the signer from web3Modal, which in our case is MetaMask
    // No need for the Signer here, as we are only reading state from the blockchain
    const signer = await getProviderOrSigner(true);
    // We connect to the Contract using a signer because we want the owner to
    // sign the transaction
    const randomGameNFTContract = new Contract(
      RANDOM_GAME_NFT_CONTRACT_ADDRESS,
      abi,
      signer
    );
    setLoading(true);
    // call the startGame function from the contract
    const tx = await randomGameNFTContract.joinGame({
      value: entryFee,
    });
    await tx.wait();
    setLoading(false);
  } catch (error) {
    console.error(error);
    setLoading(false);
  }
};

/**
 * checkIfGameStarted checks if the game has started or not and intializes the logs
 * for the game
 */
const checkIfGameStarted = async () => {
  try {
    // Get the provider from web3Modal, which in our case is MetaMask
    // No need for the Signer here, as we are only reading state from the blockchain
    const provider = await getProviderOrSigner();
    // We connect to the Contract using a Provider, so we will only
    // have read-only access to the Contract
    const randomGameNFTContract = new Contract(
      RANDOM_GAME_NFT_CONTRACT_ADDRESS,
      abi,
      provider
    );
    // read the gameStarted boolean from the contract
    const _gameStarted = await randomGameNFTContract.gameStarted();

    const _gameArray = await subgraphQuery(FETCH_CREATED_GAME());
    const _game = _gameArray.games[0];
    let _logs = [];
    // Initialize the logs array and query the graph for current gameID
    if (_gameStarted) {
      _logs = [`Game has started with ID: ${_game.id}`];
      if (_game.players && _game.players.length > 0) {
        _logs.push(
          `${_game.players.length} / ${_game.maxPlayers} already joined 👀 `
        );
        _game.players.forEach((player) => {
          _logs.push(`${player} joined 🏃`);
        });
      }
      setEntryFee(BigNumber.from(_game.entryFee));
      setMaxPlayers(_game.maxPlayers);
    } else if (!gameStarted && _game.winner) {
      _logs = [
        `Last game has ended with ID: ${_game.id}`,
        `Winner is: ${_game.winner} 🎉 `,
        `Waiting for host to start new game....`,
      ];

      setWinner(_game.winner);
    }
    setLogs(_logs);
    setPlayers(_game.players);
    setGameStarted(_gameStarted);
    forceUpdate();
  } catch (error) {
    console.error(error);
  }
};
```

```
/**
 * getOwner: calls the contract to retrieve the owner
 */
const getOwner = async () => {
  try {
    // Get the provider from web3Modal, which in our case is MetaMask
    // No need for the Signer here, as we are only reading state from the blockchain
    const provider = await getProviderOrSigner();
    // We connect to the Contract using a Provider, so we will only
    // have read-only access to the Contract
    const randomGameNFTContract = new Contract(
      RANDOM_GAME_NFT_CONTRACT_ADDRESS,
      abi,
      provider
    );
    // call the owner function from the contract
    const _owner = await randomGameNFTContract.owner();
    // We will get the signer now to extract the address of the currently connected MetaMask account
    const signer = await getProviderOrSigner(true);
    // Get the address associated to the signer which is connected to  MetaMask
    const address = await signer.getAddress();
    if (address.toLowerCase() === _owner.toLowerCase()) {
      setIsOwner(true);
    }
  } catch (err) {
    console.error(err.message);
  }
};

// useEffects are used to react to changes in state of the website
// The array at the end of function call represents what state changes will trigger this effect
// In this case, whenever the value of `walletConnected` changes - this effect will be called
useEffect(() => {
  // if wallet is not connected, create a new instance of Web3Modal and connect the MetaMask wallet
  if (!walletConnected) {
    // Assign the Web3Modal class to the reference object by setting it's `current` value
    // The `current` value is persisted throughout as long as this page is open
    web3ModalRef.current = new Web3Modal({
      network: "mumbai",
      providerOptions: {},
      disableInjectedProvider: false,
    });
    connectWallet();
    getOwner();
    checkIfGameStarted();
    setInterval(() => {
      checkIfGameStarted();
    }, 2000);
  }
}, [walletConnected]);

/*
  renderButton: Returns a button based on the state of the dapp
*/
const renderButton = () => {
  // If wallet is not connected, return a button which allows them to connect their wllet
  if (!walletConnected) {
    return (
      <button onClick={connectWallet} className={styles.button}>
        Connect your wallet
      </button>
    );
  }

  // If we are currently waiting for something, return a loading button
  if (loading) {
    return <button className={styles.button}>Loading...</button>;
  }
  // Render when the game has started
  if (gameStarted) {
    if (players.length === maxPlayers) {
      return (
        <button className={styles.button} disabled>
          Choosing winner...
        </button>
      );
    }
    return (
      <div>
        <button className={styles.button} onClick={joinGame}>
          Join Game 🚀
        </button>
      </div>
    );
  }
```

```
    // Start the game
    if (isOwner && !gameStarted) {
      return (
        <div>
          <input
            type="number"
            className={styles.input}
            onChange={(e) => {
              // The user will enter the value in ether, we will need to convert
              // it to WEI using parseEther
              setEntryFee(
                e.target.value >= 0
                  ? utils.parseEther(e.target.value.toString())
                  : zero
              );
            }}
            placeholder="Entry Fee (ETH)"
          />
          <input
            type="number"
            className={styles.input}
            onChange={(e) => {
              // The user will enter the value in ether, we will need to convert
              // it to WEI using parseEther
              setMaxPlayers(e.target.value ?? 0);
            }}
            placeholder="Max players"
          />
          <button className={styles.button} onClick={startGame}>
            Start Game 🚀
          </button>
        </div>
      );
    }
  };

  return (
    <div>
      <Head>
        <title>LW3Punks</title>
        <meta name="description" content="LW3Punks-Dapp" />
        <link rel="icon" href="/favicon.ico" />
      </Head>
      <div className={styles.main}>
        <div>
          <h1 className={styles.title}>Welcome to Random Winner Game!</h1>
          <div className={styles.description}>
            Its a lottery game where a winner is chosen at random and wins the
            entire lottery pool
          </div>
          {renderButton()}
          {logs &&
            logs.map((log, index) => (
              <div className={styles.log} key={index}>
                {log}
              </div>
            ))}
        </div>
        <div>
          <img className={styles.image} src="./randomWinner.png" />
        </div>
      </div>

      <footer className={styles.footer}>Made with &#10084; by Your Name</footer>
    </div>
  );
}
```

Now create a new folder under the my-app folder and name it `constants`. In the constants folder create a file, `index.js` and paste the following code.

- Replace `"address of your RandomWinnerGame contract"` with the address of the RandomWinnerGame contract that you deployed and saved to your notepad.
- Replace `---your abi---` with the abi of your RandomWinnerGame Contract. To get the abi for your contract, go to your `hardhat-tutorial/artifacts/contracts/RandomWinnerGame.sol` folder and from your `RandomWinnerGame.json` file get the array marked under the `"abi"` key.

```
export const abi = "---your abi---";
export const RANDOM_GAME_NFT_CONTRACT_ADDRESS =
  "address of your RandomWinnerGame contract";
```

Now in your terminal which is pointing to `my-app` folder, execute

```
npm run dev
```

Your RandomWinnerGame dapp should now work without errors 🚀

Have fun and play the game!

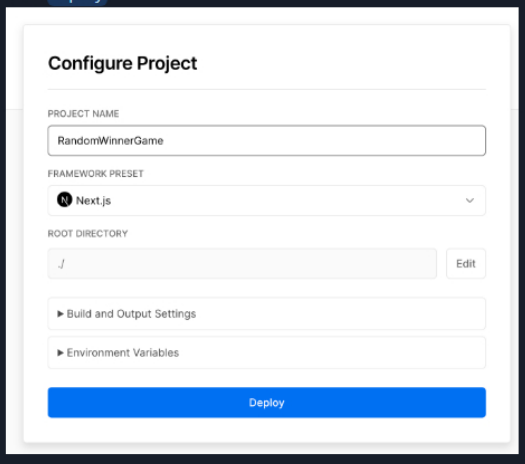
Push to github

Make sure before proceeding you have **pushed all your code to github** :)

Deploying your dApp

We will now deploy your dApp, so that everyone can see your website and you can share it with all of your LearnWeb3 DAO friends.

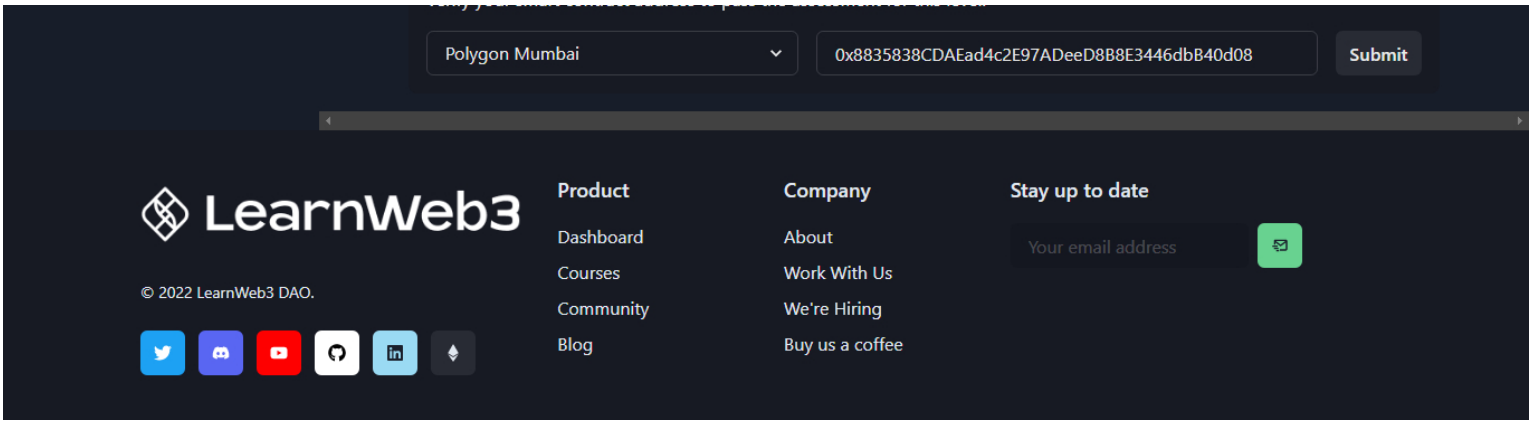
- Go to https://vercel.com/ and sign in with your GitHub
- Then click on `New Project` button and then select your RandomWinnerGame repo
- When configuring your new project, Vercel will allow you to customize your `Root Directory`
- Click `Edit` next to `Root Directory` and set it to `my-app`
- Select the Framework as `Next.js`
- Click `Deploy`



- Now you can see your deployed website by going to your dashboard, selecting your project, and copying the `domain` from there!
- Share this domain with everyone on discord and lets all play the game together 🚀 🚀 🚀 🚀

Submit Practical

Verify your smart contract address to pass the assessment for this level.



LearnWeb3

Product

- Dashboard
- Courses
- Community
- Blog

Company

- About
- Work With Us
- We're Hiring
- Buy us a coffee

Stay up to date

Your email address