Easy! Web3

Contents

Overview

What This Package Is

Easy! Web3 is a suite of tools built for Unity3D developers making it easier than ever to access data on the blockchain via a C# application built with Unity. Easy! Web3 is built on top of Nethereum, primarily building on top of the Nethereum.Web3.dll to massively improve usability for ERC20 calls, ERC721 calls, custom contract calls with support for complex structures, transaction management, wallet management, and blockchain scanning.

By using this asset you will easily be able to access data across three major blockchains; Ethereum, Binance Smart Chain, and Polygon. Everything will work out of the box as we provide test nodes for you to use. While this may impact testing performance as more people use this asset, it provides a working proof-of-concept for your immediate convenience. It is recommended that you replace our testing nodes with your own as you launch your application to a production state.

What This Package Is Not

This is not meant to replace a wallet provider, such as MetaMask. The intent is not to allow game developers to manage private keys or keystore json files for newly created or imported wallets. For security, this problem must be solved through a trusted proxy protocol, and for that reason it is not a feature that is supported in this package at this time.

Features

Supported Chains

With Easy! Web3 you will have access to Ethereum, Binance Smart Chain, and Polygon. Access is easily determined by passing a ChainId into various class constructors, which we will explore later. Here is the ChainId enumeration:

```
public enum ChainId {
  ETH_MAINNET,
  ETH_ROPSTEN,
  BSC_MAINNET,
  BSC_TESTNET,
  MATIC_MAINNET,
  MATIC_TESTNET

}
```

Free Nodes For Mainnets and Testnets

You will be granted free access to use the nodes provided in this package. You can replace the node URLs by using overloaded URLs, explained in the API section. You will be provided, out of the box, with the following blockchain access:

- Ethereum Mainnet
- Ethereum Ropsten Testnet
- Binance Smart Chain Mainnet
- Binance Smart Chain Testnet
- Polygon Mainnet
- Polygon Testnet

The Ethereum nodes provided are running on the servers at NullRef Labs, so you can count on using those with zero throttling. However, the BSC and MATIC nodes are provided courtesy of Moralis speedy-nodes. If you run into throttling issues when getting blockchain data from these networks we recommend simply grabbing your own speedy-node. We tried it and it was completely free and easy to set up. https://admin.moralis.io/login

Feature-Rich Demo Scene

The demo scene will help you explore the possibilities of what you can do with this package. By studying it, you will understand how to test your blockchain functions, make custom smart contract calls, interact with standards like ERC-20 and ERC-721 (NFT), check user balances, enforce asset-based zones in-game, and scan the blockchain for any data you might possibly need.

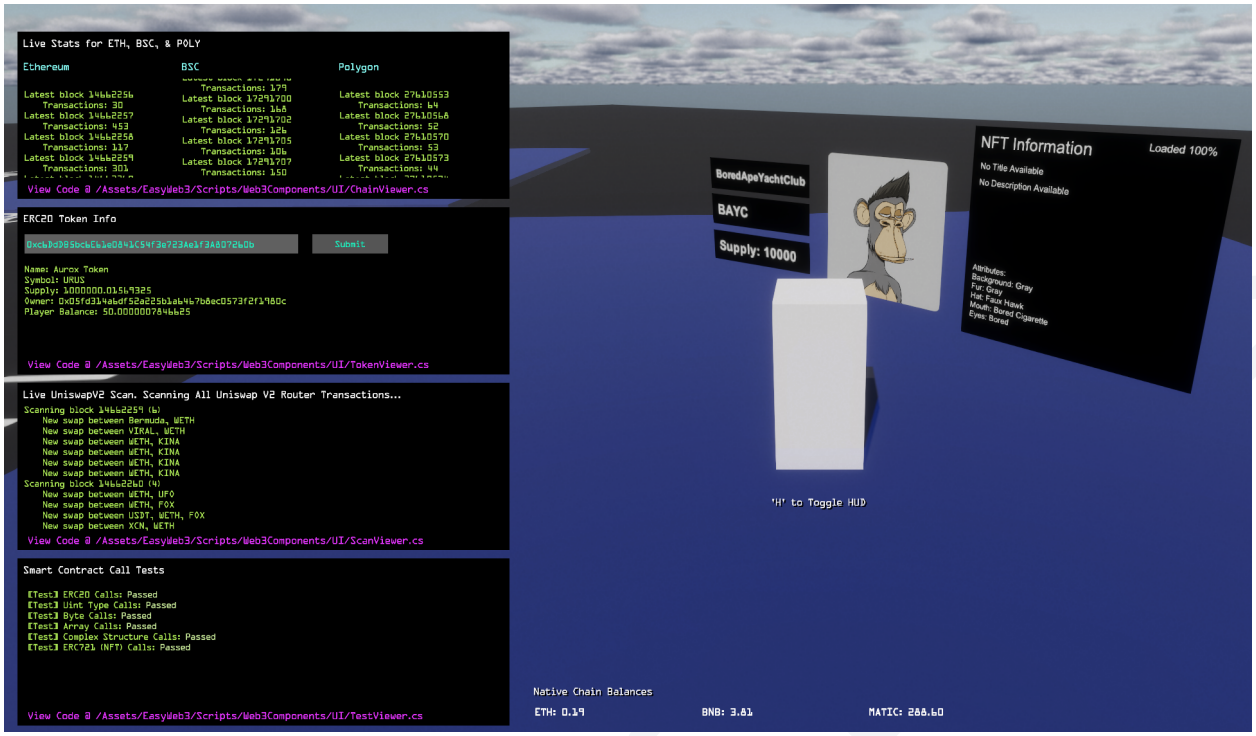

Figure 1. Demo Scene

Video Tutorials

If you prefer videos over text please see our tutorial for this asset. In the video we cover every detail of the asset. https://youtu.be/g4wG8TkkFyQ

Discord Support

Running into bugs is inevitable. So when it happens to you, find us on our Discord support channel. We are happy to help you from there. https://discord.gg/3NV829ch76

Iterative Improvements

As standards evolve, the blockchain industry changes, and our users request new features, we will continue iterating and improving this asset for the support of innovation in the Web3 space. We are firm believers in the value and prospects Web3 and decentralization has to offer!

Code Usage

ERC-20 and ERC-721 Contracts

To easily use functions from the ERC-20 and ERC-721 standards, use the respective class to construct your object.

```
string EasyWeb3UnitTestContract = "0x70396512216dbf32C42EB798C51a9616FdDb683a";
ERC20 _token = new ERC20(EasyWeb3UnitTestContract, ChainId.ETH_ROPSTEN);
BigInteger _decimals = await _token.GetDecimals();
BigInteger _totalSupply = await _token.GetTotalSupply();
string _name = await _token.GetName();
string _symbol = await _token.GetSymbol();
string _owner = await _token.GetOwner();
BigInteger _balance = await _token.GetBalanceOf("0x34221445c2dd9fd3f41a8a8bfa7d49ec898e0ef4");
BigInteger _allowance = await _token.GetAllowance("0xbd0dbb9fddc73b6ebffc7c09cfae1b19d6dece40",
Constants.ADDR_UNISWAPV2);
```

The ERC-721 contract object provides similar functions for total supply, name, and symbol, but there are a few other functions you can call to manage NFTs.

```
private async Task<bool> LoadOwnerNFTS(string _nftContract, string _nftOwner) {
  ERC721 _nft = new ERC721(_nftContract, ChainId.ETH_MAINNET);
  await _nft.Load();
  Debug.Log("\tname: "+_nft.Name);
  Debug.Log("\tsymbol: "+_nft.Symbol);
  Debug.Log("\ttotal supply: "+_nft.ValueFromDecimals(_nft.TotalSupply));
  Debug.Log("Getting NFTs from an owner...");
  List<NFT> _nfts = await _nft.GetOwnerNFTs(_nftOwner,
    (_nft,_progress) => { // called when an nft is found
      Debug.Log("\t"+(_progress*100)+"% | Loaded NFT: "+_nft.Data.image);
    },
    (_nftId, _error) => { // called when an nft fails to load
      Debug.LogWarning("\tFailed to load tokenId "+_nftId+": "+_error);
    });
  return true;
```

You can use a function like this to load every NFT for a given owner provided the NFT contract address. You will await a call to 'GetOwnerNFTs', which will return a 'List<NFT>' while allowing you to hook into a callback that invokes for each individual NFT that loads.

Custom Contracts

The last two code snippets showed you how to utilize the ERC-20 and ERC-721 classes. In this section we will look into creating a custom class for your contract. It is worth noting that everything shown in this section can also be utilized by ERC20 and ERC721 as these classes simply inherit from a base class, shown below.

```
Contract _cronos = new Contract("0xa0b73e1ff0b80914ab6fe0444e65848c4c34450b", ChainId.ETH_MAINNET);
var _out = await _cronos.CallFunction("getUpgradeState()", new string[]{"uint"});
Debug.Log("CRONOS upgrade state: "+(BigInteger)_out[0]);
_out = await _cronos.CallFunction("canUpgrade()", new string[]{"bool"});
Debug.Log("CRONOS upgradeable: "+(bool)_out[0]);
_out = await _cronos.CallFunction("airdropReserveWallet()", new string[]{"address"});
Debug.Log("CRONOS airdrop reserve wallet: "+(string)_out[0]);
```

You can call any contract function you find (almost). There are some restrictions on struct and byte input types, but here is a full list of compatible input and output types when calling custom functions:

Inputs	**Outputs**
- uint256/uint128/uint64/uint32/uint16/uint8/uint - uint256[]/uint128[]/uint64[]/uint32[]/uint16[]/uint8[]/uint[] - string, string[] - bytes, bytes[] - address, address[] - bool, bool[]	- bool, bool[] - string, string[] - address, address[] - bytes, bytes[] - uint, uint[] - struct

There are a couple things to point out based on this spec;

1. All input types must be specified using the literal Solidity type definition. For instance, if the solidity function uses uint128, you must also. This is simply because we must be able to find the correct function hash for blockchain encoding.
2. Outputs can be specified more generally than inputs. For instance, if a solidity return value specifies any uint type, you can just specify ‘uint’.
3. Structs are only supported as singular outputs, arrays are not yet supported.

Let’s look at the anatomy of a custom contract call next.

```
Debug.Log("TEST: getComplex2(string[],uint256[],bool[])");
List<object> _out = await _contract.CallFunction(
  "getComplex2(string[],uint256[],bool[])", // the function signature, without input var names
  new string[]{"struct(uint,string,bool,address)","uint[]","string"}, // the output types
  new string[]{"string[](str1,str2)","uint[](1,2,3)","bool[](1,0,1,0)"} // the input values
);
_structint = (BigInteger)_out[0];
_structstring = (string)_out[1];
_structbool = (bool)_out[2];
_structaddr = (string)_out[3];
BigInteger[] _intarr = (BigInteger[])_out[4];
string _str = (string)_out[5];
```

```
13    Debug.Log("\t_struct:");
14    Debug.Log("\t\tint: "+_structint);
15    Debug.Log("\t\tstring: "+_structstring);
16    Debug.Log("\t\tbool: "+_structbool);
17    Debug.Log("\t\taddress: "+_structaddr);
18    Debug.Log("\t_intarr: ");
19    for(int i = 0; i < _intarr.Length; i++) {
20      Debug.Log("\t\t"+i+": "+_intarr[i]);
21    }
22    Debug.Log("\tstring: "+_str);
```

This complex function is calling the same ropsten testing contract that all of the tests hit from the TestViewer.cs script. You can view that contract here: https://ropsten.etherscan.io/address/0x70396512216dbf32C42EB798C51a9616FdDb683a#code

By studying this block of code you will understand how to make any custom smart contract call. Here are the high-level points you should be aware of:

1. The 'CallFunction' function returns a list of objects. These objects will map to solidity types like so:
    a. address/address[] => string/string[]
    b. string/string[] => string/string[]
    c. bytes/bytes[] => string/string[]
    d. uint256/uint256[] => BigInteger/BigInteger[]
    e. bool/bool[] => bool/bool[]
    f. struct => object[]
2. The first parameter is a string Solidity function signature excluding the variable names. For instance, function getComplex2(string[] memory _str, uint[] memory _num, bool[] memory _bool) external pure returns (DataStruct memory, uint[] memory, string memory) => "getComplex2(string[],uint256[],bool[])". This value allows us to derive the function hash.
3. The second parameter represents the output types, this is a required param, and is an array of strings, 'string[]'. This should also match the order of the function signature pasted above. If the returns are (DataStruct memory, uint[] memory, string memory) we can reliably enter "new string[]{"struct(uint,string,bool,address)", "uint[]", "string"}". You can simply nest the struct types into the struct using parentheses.
4. The third parameter is optional, and describes any inputs you might require for the function. This parameter is also entered as an array of strings, 'string[]'. Array value inputs are specified accordingly: "arraytype[](arrval1,arrval2,...)". All other input types can just be stringified and included in the string[] input array.

5. When you get a successful response you will likely want to access the outputs. You can do this by iterating over a list of objects from the response, 'List<object>'. All outputs will be accessible in order. Struct returns are spread out into the return list. Array returns will be accessed as arrays. All return types need to be explicitly cast into the correct C# data type.

To close this topic out, let's look at a more common, more simple example.

```
1    public async Task<BigInteger> GetAllowance(string _owner, string _spender) {
2      var _out = await CallFunction("allowance(address,address)", new string[]{"uint"}, new
3    string[]{_owner,_spender});
4      return (BigInteger)_out[0];
5    }
```

NFTs

NFTs are a hot topic today, so we should revisit the functionality that comes with Easy! Web3 out of the box for NFTs. Let's start by getting an understanding for the data we can get natively from the ERC-721 standard.

When you request an NFT from a smart contract you get a link to some JSON data describing the NFT. You might find a title, description, and some attributes attached, but there should always be an image field where we can find the image, video, gif, or other media. Here is what that might look like:

{"name":"The Polygon Punk #5","description":"The Polygon Punks finally arrived on the Polygon Network. The Polygon Punks is a collection of 9999 unique Polygon Punk NFTs living on the Polygon blockchain.","image":"ipfs://QmR77ooBgaAuuWphYQiyQdfLrjuBJWP7FJHYRzPyKfRDw9/5.png","dna":"ca4db5357fc6bf1ebbc171398707665394ca7827","edition":5,"date":1643291333156,"attributes":[{"trait_type":"Background","value":"Purple"},{"trait_type":"Ethnicity","value":"African American"},{"trait_type":"Eyes","value":"Closed"},{"trait_type":"Mouth","value":"White Open"},{"trait_type":"Clothing","value":"Sleevless Green"},{"trait_type":"Beard","value":"Thick Brown"},{"trait_type":"Hair","value":"Fringes"},{"trait_type":"Personality","value":"Crooked"},{"trait_type":"Skill","value":"Explosive expert"}],"compiler":"HashLips Art Engine"}

By using the ERC721 class, you can easily access this data via predefined class structures. Let's take another look at how we can get owner NFTs, and then talk about what is inside the NFT object.

```
1   private async Task<bool> LoadOwnerNFTS(string _nftContract, string _nftOwner) {
2     ERC721 _nft = new ERC721(_nftContract, ChainId.ETH_MAINNET);
3     await _nft.Load();
4     Debug.Log("\tname: "+_nft.Name);
5     Debug.Log("\tsymbol: "+_nft.Symbol);
6     Debug.Log("\ttotal supply: "+_nft.ValueFromDecimals(_nft.TotalSupply));
7     Debug.Log("Getting NFTs from an owner...");
8     List<NFT> _nfts = await _nft.GetOwnerNFTs(_nftOwner,
9       (_nft,_progress) => { // called when an nft is found
10        Debug.Log("\t"+(_progress*100)+"% | Loaded NFT: "+_nft.Data.image);
11      },
12      (_nftId, _error) => { // called when an nft fails to load
13        Debug.LogWarning("\tFailed to load tokenId "+_nftId+": "+_error);
14      });
15    return true;
```

Looking at line 9 we can see a callback function is feeding us the NFT data via the '_nft' variable. Now we can explore the internals of this object. The following code defines the NFT..

```
1   public class NFTAttribute {
2     public string trait_type;
3     public string value;
4   }
5   public class NFTData {
6     public string title;
7     public string image;
8     public string name;
9     public string description;
10    public NFTAttribute[] attributes;
11  }
12  public class NFT {
13    public int Id {get; private set;}
14    public string Uri {get; private set;}
15    public NFTData Data {get; private set;}
16    public NFT(int _id, string _uri, NFTData _data) {
17      Id = _id;
18      Uri = _uri;
19      Data = _data;
20    }
21  }
```

This is the way we are able to access NFT images, attributes, title, and descriptions from the demo. All of the json is parsed for your convenience. Later we will look into some more details of the TriggerNFTLoader to understand how we might use this data.

Transactions

To start, please refer to the Nethereum Transactions classes;
https://github.com/Nethereum/Nethereum/blob/abd05e8a3b936419f9f278bbfe57f816ee1182b0/src/Nethereum.RPC/Eth/DTOs/Transaction.cs
https://github.com/Nethereum/Nethereum/blob/abd05e8a3b936419f9f278bbfe57f816ee1182b0/src/Nethereum.RPC/Eth/DTOs/TransactionReceipt.cs

We leverage this class to assist with accessing transactions that are output from the Easy! Web3 scanners. The main reason we have a wrapper for the Transaction class is so that we can easily decode the inputs of the transaction into a list of objects 'List<object>'. There are a couple reasons you might interact with web3 at the transaction level. Perhaps you are processing transactions from a scanner, or maybe you just want to track the progress of a single transaction. Let's first look at how we might track the progress of a single transaction. Looking at the TransactionProgressTracker.cs file, we find the following code:

```
1   private async void CheckReceipt() {
2     Web3ify m_Web3 = new Web3ify(ChainId.ETH_MAINNET);
3     var _receipt = await m_Web3.GetTransactionReceipt(Hash);
4     if (_receipt == null) {
5       Debug.Log("[TransactionProgressTracker] Pending");
6       return;
7     }
8     BigInteger _status = (BigInteger)_receipt.Status;
9     if (_status == 0) {
10      Debug.Log("[TransactionProgressTracker] Failed");
11      if (OnFail != null) {
12        OnFail.Invoke();
13      }
14    } else {
15      Debug.Log("[TransactionProgressTracker] Success");
16      if (OnSuccess != null) {
17        OnSuccess.Invoke();
18      }
19    }
20    m_IsComplete = true;
21  }
```

This function runs in a coroutine until 'm_IsComplete' evaluates true. We can access some general chain functions by initializing a 'Web3ify' object. After grabbing the receipt via transaction hash, we can check the status of the transaction. Another example of accessing transaction data is through a scanner. Let's take a look at an example of a basic blockchain scanner next by opening ChainViewer.cs

```
1   private IEnumerator Scan() {
2     var _web3 = new Web3ify(chainId);
3     while (true) {
4       _web3.ScanAll(async (_txs, _blockNum, _isNew)=>{
5         if (_isNew) {
6           Feed.text += "\nLatest block "+_blockNum+"\n\tTransactions: "+_txs.Count;
7           Scroller.verticalScrollbar.value = 0;
8           Canvas.ForceUpdateCanvases();
9         }
10        foreach(Nethereum.RPC.Eth.DTOs.Transaction _nethTx in _txs) {
11          Transaction _tx = new Transaction(_nethTx); // wrapper
12          var _inputHex = _tx.Data.Input; // Access all Nethereum properties through the Data property
13          var _methodId = _tx.MethodId;
14          // process the hex...
15        }
16      });
17      yield return new WaitForSeconds(ScanRate);
18    }
19  }
```

To utilize the Transaction wrapper you can look toward line 11 where we simply pass the Nethereum object into the constructor. Now the '_tx' variable has some extra functionality for handling the transaction. We can still access the inherent transaction properties through the 'Data' field, but we can also process the input more easily as well. We will explore this in depth in the scanner sections below.

Wallets

There are three core functions you get out of the Wallet class.

1. Wallet.ShortAddress;
2. Wallet.GetNativeBalance();
3. Wallet.GetERC20Balance();

Check out some examples of usage by looking at the Web3Player.cs script.

```
1   public class Web3Player : MonoBehaviour
2   {
3     public string ethAddress;
4     public string bscAddress;
5     public string maticAddress;
6
7     public async Task<double> GetNativeBalanceFromChain(EasyWeb3.ChainId _chainId) {
8       return await (new Wallet(GetAddressFromChain(_chainId), _chainId)).GetNativeBalance();
9     }
10
11    public async Task<double> GetERC20BalanceFromChain(string _contract, EasyWeb3.ChainId _chainId) {
12      return await (new Wallet(GetAddressFromChain(_chainId), _chainId)).GetERC20Balance(_contract);
13    }
14
15    public string GetAddressFromChain(EasyWeb3.ChainId _chainId) {
```

```
16      switch (_chainId) {
17        case EasyWeb3.ChainId.ETH_MAINNET:
18        case EasyWeb3.ChainId.ETH_ROPSTEN: return ethAddress;
19        case EasyWeb3.ChainId.BSC_MAINNET:
20        case EasyWeb3.ChainId.BSC_TESTNET: return bscAddress;
21        case EasyWeb3.ChainId.MATIC_MAINNET:
22        case EasyWeb3.ChainId.MATIC_TESTNET: return maticAddress;
23        default: return ethAddress;
24      }
25    }
26  }
```

Encoder & Decoder

The Encoder and Decoder classes are core components to enabling usability with the Easy! Web3 package. These utilities allow us to pass Solidity-ish code into our custom contract calls, and are responsible for converting relevant data to hex code for blockchains to process. For illustration, look at the code below:

```
1   _out = await _token.CallFunction(
2     "getComplex2(string[],uint256[],bool[])", // the function signature, without input var names
3     new string[]{"struct(uint,string,bool,address)","uint[]","string"}, // the output types
4     new string[]{"string[](str1,str2)","uint[](1,2,3)","bool[](1,0,1,0)"} // the input values
5   );
```

When Contract.CallFunction(string, string[], string[]) gets called, it uses the encoder and decoder objects to make hex conversions based on data types and input data. To get input hex data, you can use the encoder:

```
string _encodedInput = new Encoder().Encode(_functionSignature, _inputTypes, _inputValues);
```

…where the inputs are string, string[], and string[] respectively. If you ever need to decode hex values, like you will if you are processing scanned transactions, you can call the decoder like this:

```
int _l = 0; List<object> _out = new Decoder().Decode(_hex, _outputTypes, ref _l);
```

…where the inputs are string, string[], and ref int respectively. The final input type must be included, but is of no concern here. The output of the decoder is a list of objects.

Web3 Utils

The Web3Utils class provides static functions to assist with various hex, string, address, and integer conversions. In the API section we will list these functions out in detail. You will likely not need to use these utils unless you are modifying or extending the decoder and encoder functions. You will likely use the Web3Utils.GetNodeUrl(ChainId _chain); function to access your nodes, though.

Trigger-Based NFT Loader

This loader was included in the demo for demonstration purposes. While it is fun to use, it is not meant to set any sort of standard for showing NFTs to your users. The core purpose of this loader is to help you grasp the transfer of blockchain data into your application. By studying the TriggerNFTLoader.cs file you will learn how to access NFTs for your users.

```
1   private async Task<bool> LoadOwnerNFTS(string _nftContract, string _nftOwner) {
2     try {
3       m_IsLoading = true;
4       m_DidLoadNFTs = true;
5       Debug.Log("[TriggerNFTLoader] Getting NFTs from owner...");
6       List<NFT> _nfts = await m_NftContract.GetOwnerNFTs(_nftOwner,
7         (_nft,_progress) => { // called when an nft is found
8           Progress.text = "Loaded "+((int)(_progress*100))+"%";
9           LoadTexture(_nft);
10        },
11        (_nftId, _error) => { // called when an nft fails to load
12          Debug.LogWarning("\t[TriggerNFTLoader] Failed to load tokenId "+_nftId+": "+_error);
13        });
14    } catch (System.Exception) {
15      m_IsLoading = false;
16      m_DidLoadNFTs = false;
17    }
18    return true;
19  }
20
21  private async Task<bool> LoadTexture(NFT _nft) {
22    Debug.Log("\t[TriggerNFTLoader] Loaded NFT: "+_nft.Data.image);
23    string _url = _nft.Data.image.Contains("ipfs://") ? _nft.Data.image.Replace("ipfs://","https://ipfs.io/ipfs/") :
24  _nft.Data.image;
25    _nft.AssetUrl = _url;
26    if (_url.Contains("mp4")) {
27    } else {
28      UnityWebRequest _req = UnityWebRequestTexture.GetTexture(_url);
29      await _req.SendWebRequest();
30      Texture2D _tex = ((DownloadHandlerTexture)_req.downloadHandler).texture;
31      _nft.texture = _tex;
32      m_NFTs.Add(_nft);
33      if (m_NFTs.Count == 1) {
34        DrawNFTData(_nft);
35      }
36    }
37    return true;
38  }
```

These are the two core functions in the script. The function ‘LoadOwnerNFTs’ is sequentially requesting NFT data and passing the data to the function for ‘LoadTexture’. The NFT media is stored in the ‘_nft.Data.image’ property where you can filter for various types of media such as png, jpeg, mp4, etc… and handle those formats as you wish. Here we are taking everything that is not a video and using the ‘UnityWebRequest’ to access the online image.

Trigger-Based Balance Action

The TriggerBalanceAction.cs script is nice if you want to restrict areas of your game to users that hold a certain balance of a particular cryptocurrency.

```
1   public class TriggerBalanceAction : MonoBehaviour
2   {
3     public string Token;
4     public ChainId ChainId;
5     public int BalanceRequirement;
6     public UnityEvent OnRequirementMet;
7
8     private void OnTriggerEnter(Collider _col) {
9       Web3Player _player = _col.gameObject.GetComponent<Web3Player>();
10      if (_player != null) {
11        CheckBalance(_player);
12      }
13    }
14
15    private async void CheckBalance(Web3Player _player) {
16      string _addr = ChainId == ChainId.BSC_MAINNET ? _player.bscAddress : ChainId == ChainId.MATIC_MAINNET ?
17  _player.maticAddress : _player.ethAddress;
18      ERC20 _token = new ERC20(Token,ChainId);
19      BigInteger _bal = await _token.GetBalanceOf(_addr);
20      if (_bal >= BalanceRequirement) {
21        OnRequirementMet.Invoke();
22      }
23    }
24  }
```

On line 10 you can see we check for whether or not the player is entering the trigger zone. If so, we check the balance of the player by using the ChainId to determine the appropriate user address. Once we have the player's address we make a call, line 19, to get the balance. If the balance requirement is met we invoke a UnityEvent to cause some sort of action in-game. In the case of the demo, we open a door to a building.

UniswapV2 Scanner

To encourage creativity around data scanning we provided an example scanner for UniswapV2 transactions. It is important to realize that anything that processes transactions on the blockchain can be scanned and/or tracked. We will get into that in the next section.

To study the UniswapV2 scanner open up the ScanViewer.cs file.

```
1   private void Start() {
2     StartScan();
3   }
4
5   private void OnDisable() {
6     StopAllCoroutines();
```

```
7   }
8
9   private void StartScan() {
10    Contract _uniswapv2 = new Contract(Constants.ADDR_UNISWAPV2, m_ChainId);
11    StartCoroutine(Scan(_uniswapv2));
12  }
13
14  private IEnumerator Scan(Contract _contract) {
15    while (true) {
16      _contract.Scan(async (_txs, _blockNum, _isNew)=>{
17        if (_isNew) {
18          Feed.text += "\nScanning block "+_blockNum+" ("+_txs.Count+")";
19          ProcessTransactions(_txs);
20        }
21      });
22      Scroller.verticalScrollbar.value = 0;
23      yield return new WaitForSeconds(3);
24    }
25  }
```

The first step is to set up an IEnumerator so your scanner can run off of the main thread. This way we don't have to worry about a continuous loop interrupting the game loop. When you do this you want to make sure you have a way to stop the coroutine appropriately, see line 5. The coroutine will continuously loop and scan new blocks via the Contract.Scan() function. The function you pass in should accept three parameters. The first is a list of transactions for the block, the second is a block number, and the third will tell you if that block is new (has it been scanned yet).

```
1   private async void ProcessTransactions(List<Nethereum.RPC.Eth.DTOs.Transaction> _txs) {
2     foreach (var _tx in _txs) {
3       bool _success = await ProcessTx(new Transaction(_tx));
4     }
5   }
```

Because we want to display to our text feed sequentially we create an awaiter function that processes transactions one at a time. This is not a mandatory step, but it is useful if the order of transactions matters to you. It is important to note that we are using our Transaction wrapper in the argument of the ProcessTx() field. This will grant us some extra usability when decoding inputs. Next we will look into what 'ProcessTx' is doing.

```
1   string[] _trackedMethods = new string[]{
2     "swapExactTokensForTokens(uint256,uint256,address[],address,uint256)",
3     "swapExactTokensForETHSupportingFeeOnTransferTokens(uint256,uint256,address[],address,uint256)",
4    "swapExactTokensForTokensSupportingFeeOnTransferTokens(uint256,uint256,address[],address,uint256)",
5     "swapTokensForExactTokens(uint256,uint256,address[],address,uint256)",
6     "swapExactTokensForETH(uint256,uint256,address[],address,uint256)",
7     "swapTokensForExactETH(uint256,uint256,address[],address,uint256)",
8     "swapExactETHForTokens(uint256,address[],address,uint256)",
```

```
  "swapExactETHForTokensSupportingFeeOnTransferTokens(uint256,address[],address,uint256)",
  "swapETHForExactTokens(uint256,address[],address,uint256)",
  "addLiquidityETH(address,uint256,uint256,uint256,address,uint256)",
"removeLiquidityETHWithPermit(address,uint256,uint256,uint256,address,uint256,bool,uint8,bytes32,bytes
32)"
};
```

The first step is to define every function we want to track on the UniswapV2 contract. We create an array of function signatures that are found in the UniswapV2 Router contract: `0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D`.

```
for (int i = 0; i < _trackedMethods.Length; i++) {
  if (Web3Utils.FunctionHash(_trackedMethods[i]) == _tx.MethodId) {
    List<object> _inputs = null;
    string[] _path = null;
    string _symbol = null;
    string _token = null;
    switch (i) {
      case 0: //swapExactTokensForTokens
      case 1: //swapExactTokensForETHSupportingFeeOnTransferTokens
      case 2: //swapExactTokensForTokensSupportingFeeOnTransferTokens
      case 3: //swapTokensForExactTokens
      case 4: //swapExactTokensForETH
      case 5: //swapTokensForExactETH
        _inputs = _tx.GetInputs(new string[]{"uint","uint","address[]","address","uint"});
        _path = (string[])_inputs[2]; // path (address[]) is the 3rd input
        Feed.text += "\n\tNew swap between ";
        foreach (string _addr in _path) {
          _symbol = await (new ERC20(_addr, m_ChainId)).GetSymbol();
          Feed.text += _symbol + ", ";
        }
        Feed.text = Feed.text.Substring(0,Feed.text.Length-2);
        break;
      case 6: //swapExactETHForTokens
      case 7: //swapETHForExactTokens
      case 8: //swapExactETHForTokensSupportingFeeOnTransferTokens
        _inputs = _tx.GetInputs(new string[]{"uint","address[]","address","uint"});
        _path = (string[])_inputs[1]; // path (address[]) is the 2nd input
        Feed.text += "\n\tNew swap between ";
        foreach (string _addr in _path) {
          _symbol = await (new ERC20(_addr, m_ChainId)).GetSymbol();
          Feed.text += _symbol + ", ";
        }
        Feed.text = Feed.text.Substring(0,Feed.text.Length-2);
        break;
      case 9: //addLiquidityETH
        _inputs = _tx.GetInputs(new string[]{"address","uint","uint","uint","address","uint"});
        _token = (string)_inputs[0]; // token (address) is the 1st input
        _symbol = await (new ERC20(_token, m_ChainId)).GetName();
        Feed.text += "\n\tLiquidity was just added to "+_symbol;
        break;
      case 10: //removeLiquidityETHWithPermit

```

```
43              _inputs = _tx.GetInputs(new
44  string[]{"address","uint","uint","uint","address","uint","bool","uint","simplebytes","simplebytes"});
45              _token = (string)_inputs[0]; // token (address) is the 1st input
46              _symbol = await (new ERC20(_token, m_ChainId)).GetName();
47              Feed.text += "\n\tLiquidity was just removed from "+_symbol;
48              break;
49          }
50          Scroller.verticalScrollbar.value = 0;
51      }
    }
```

Finally, we loop through the function hash of each tracked method and compare it with the scanned transaction's MethodId field. If there is a match it means we found a function that we care about. Next we switch on the index of the function to determine how we want to process it. Thanks to our Transaction wrapper class we are able to convert the input hex into a C# list, 'List<object>'. For swaps we care about grabbing the involved tokens so we can display their symbols. The tokens are found in the path parameter, which is why you see on lines 15 and 27 we grab the 3rd and 2nd input parameter respectively. With the addresses we are able to construct the ERC20 object to access the symbol and display it to the feed.

Overloading Node URLs

You can use your own testnet and mainnet nodes if you want, and it is recommended before you go to a live production state to switch over to more reliable nodes. We provided a NodeURLOverloader.cs script for you to manage this. There is a prefab you can simply pull into your scene as well.

```
1   public class NodeURLOverloader : MonoBehaviour
2   {
3       public string ETHMainnetURL;
4       public string ETHRopstenURL;
5       public string BSCMainnetURL;
6       public string BSCTestnetURL;
7       public string MATICMainnetURL;
8       public string MATICTestnetURL;
9
10      // If Url is blank, the provided default url will be used for that chain
11      private void Awake() {
12          Web3ify.OVERLOADED_ETH_MAINNET_NODE_URL = ETHMainnetURL;
13          Web3ify.OVERLOADED_ETH_ROPSTEN_NODE_URL = ETHRopstenURL;
14          Web3ify.OVERLOADED_BSC_MAINNET_NODE_URL = BSCMainnetURL;
15          Web3ify.OVERLOADED_BSC_TESTNET_NODE_URL = BSCTestnetURL;
16          Web3ify.OVERLOADED_MATIC_MAINNET_NODE_URL = MATICMainnetURL;
17          Web3ify.OVERLOADED_MATIC_TESTNET_NODE_URL = MATICTestnetURL;
18      }
19  }
```

API

ChainId

A simple enumeration of supported chains.

ChainId.ETH_MAINNET
ChainId.ETH_ROPSTEN
ChainId.BSC_MAINNET
ChainId.BSC_TESTNET
ChainId.MATIC_MAINNET
ChainId.MATIC_TESTNET

Web3ify

Provides access to Nethereum's web3 object for accessing the blockchain. Manages the node url.

property ChainId chainId	Returns the chainId associated with this web3 object.
constructor Web3ify(ChainId)	Constructor accepting a ChainId object. The chainId is used to determine the node url.
constructor Web3ify(string)	Constructor accepting a string. The string should be a valid node url.
async Task<BigInteger> GetChainId()	Returns the chainId of the web3 object.
async Task<Transaction> GetTransaction(_hash)	Returns a Netheruem transaction object provided the string '_hash' transaction hash.
async Task<TransactionReceipt> GetTransactionReceipt(_hash))	Returns a Netheruem transaction receipt object provided the string '_hash' transaction hash.
async Task<HexBigInteger> GetBlockNumber()	Returns the block number of the current chain based on the current chainId.
async Task<BlockWithTransactions> GetTransactionsOnBlock(_blockNum)	Returns the Nethereum BlockWithTransactions object, which holds an array of Transaction, based on the provided HexBigInteger _blockNum.
async Task<List<Transaction>> ScanAll(_onScanComplete)	Returns a list of Nethereum Transaction objects for the current block. Each time this is called the scanner will use the current block, and ignore blocks that have already been scanned. The transaction list is also sent through a callback _onScanComplete.

Web3Utils

Provides an interface for conversions to and from hex and strings, addresses, and integers.

static string GetNodeUrl(ChainId)	Returns the appropriate node url based on chainId.
static string FunctionHash(string)	Returns a keccak hash from the first 4 bytes of a function signature. This is used for custom contract call encoding.
static string HexToString(string)	Returns a standard format string from a hex format string.
static string HexAddressToString(string)	Returns a standard format address and from a hex format address.
static string AddressToHexString(string)	Returns a hex format address from a standard format address
static string StringToHexBigInteger(string)	Returns a hex format integer from a stringified standard format integer.
static string StringToHexString(string)	Returns a hex format string from a standard format string.

Contract

Extends Web3ify. Provides custom contract call and blockchain scanning functionality for a given address.

property BigInteger TotalSupply	Returns the contract's total supply.
property BigInteger Decimals	Returns the contract's decimals.
property string Name	Returns the contract's name.
property string Owner	Returns the contract's owner.
property string Symbol	Returns the contract's symbol.
constructor Contract(string, ChainId)	Constructor accepting a contract address and a chain id.
constructor Contract(string, string)	Constructor accepting a contract address and a node url, respectively.
double ValueFromDecimals(_value)	Converts a BigInteger '_value' into a relative value based on the decimals loaded in the contract.
async Task<List<object>> CallFunction(string, string[], string[])	Builds a custom call for a contract address. Using a function signature, an array of outputs, and an optional array of inputs, a list of objects are returned based on the output of the custom contract call.
async Task<List<Transaction>> Scan(_onScanComplete)	Returns a list of Nethereum Transaction objects for the current block. Each time this is called the

	scanner will use the current block, and ignore blocks that have already been scanned. The transaction list is also sent through a callback _onScanComplete. Only transactions to this contract will be returned.

ERC20

Extends Contract, which extends Web3ify. Provides a basic interface for calling ERC20 functions.

constructor ERC20(string, ChainId)	Constructor accepting a token address and a chain id.
constructor ERC20(string, string)	Constructor accepting a token address and a node url, respectively.
constructor ERC20(string)	Constructor accepting a token address where the default chain is ETH_ROPSTEN.
async Task<bool> Load()	Loads the total supply, decimals, name, symbol, and owner of the token.
async Task<BigInteger> GetTotalSupply()	Returns the total supply of the token.
async Task<BigInteger> GetDecimals()	Returns the decimals of the token.
async Task<string> GetName()	Returns the name of the token.
async Task<string> GetSymbol()	Returns the symbol of the token.
async Task<string> GetOwner()	Returns the address of the owner of the token.
async Task<BigInteger> GetBalanceOf(_addr)	Returns the balance of the address '_addr' for the token.
async Task<BigInteger> GetAllowance(_owner, _spender)	Returns the allowance an address '_spender' has to spend for the address's '_owner' tokens.

ERC721

Extends Contract, which extends Web3ify. Provides a basic interface for calling ERC721 functions.

constructor ERC721(string, ChainId)	Constructor accepting a token address and a chain id.
constructor ERC721(string, string)	Constructor accepting a token address and a node url, respectively.

constructor ERC721(string)	Constructor accepting a token address where the default chain is ETH_ROPSTEN.
async Task<bool> Load()	Loads the total supply, name, and symbol of the token.
async Task<BigInteger> GetTotalSupply()	Returns the total supply of the token.
async Task<string> GetName()	Returns the name of the token.
async Task<string> GetSymbol()	Returns the symbol of the token.
async Task<BigInteger> GetBalanceOf(_addr)	Returns the number of NFTs held by the owner address '_addr'.
async Task<BigInteger> GetTokenOfOwnerByIndex(_owner, _index)	Returns the token ID of the NFT found at the '_index' of the addresses's '_owner' balance.
async Task<string> GetToken(_tokenId)	Returns the URI that holds the metadata for the NFT at '_tokenId'.
async Task<List<NFT>> GetOwnerNFTs(_owner, _onProgress, _onFail)	Returns a list of NFTs the address '_owner' holds for this contract. A callback can optionally be included for '_onProgress' and '_onFail' to capture NFTs that successfully loaded or failed to load.

Transaction

Wraps around the Nethereum Transaction object. Provides more usability for decoding hex input data from blockchain transactions.

property string MethodId	Returns the hex method ID of the transaction.
property Transaction Data	Returns the wrapped Nethereum Transaction object.
constructor Transaction(Transaction)	Constructor accepting a Nethereum transaction object.
List<object> GetInputs(string[])	Returns a list of decoded objects provided an array of Solidity types expected in the transaction's Input field.

Wallet

Provides a quick interface for grabbing native and ERC-20 balances for a given address.

property string ShortAddress	Returns the short-form address for this wallet. (0x0000…0000).
constructor Wallet(string, ChainId)	Constructor accepting a wallet address and a chainId.

constructor Wallet(ChainId)	Constructor accepting a wallet address. The default chainId is ETH_ROPSTEN.
async Task<double> GetNativeBalance()	Returns the balance of the chain's native token which this wallet holds.
async Task<double> GetERC20Balance(_contract)	Returns the balance of an ERC20-compatible token which this wallet holds.
async Task<BigInteger> GetTransactionCount()	Returns the number of transactions this wallet has executed.