Non-Fungible Token

We are going to learn today what are Non-fungible Tokens (NFTs) , what are Fungible Tokens, what is ERC-721 and REC-20 tokens. We will learn how to build NFTs and deploy them. No pre-requisite is required for this content and let's jump into it.

What are NFTs?

NFTs or Non-Fungible Tokens are tools used on Blockchain world. You would not really find NFTs outside of Blockchain space. NFTs allows for range of possibilities and applications. NFTs can be anything - a cat, a dog, a color token, anything that you want. THe difference between NFT and fungible Tokens is that every Fungible Token has the same value - like a dollar is well, a dollar. And hence people can exchange two fungible tokens without worrying about the value as long as the number of tokens exchanged were fine. NFTs on the other hand do not have same value. Let's say if you made dogs as your NFTs, then a great dane might have more value than a golden retriever. Trading NFTs is not easy because one has to ensure that the value remains the same. Every NFT has a unique id which decides its worth. Anothier example of NFT would be baseball carss - not every baseball cars have the same worth, some are mint condition and some can be foun in every store. One example of a fungible token is OMG cryptocurrency - OMG is an ERC-20 token so every OMG can be traded with another OMG token of same value because every OMG token has the same worth as any other OMG.

Then What are ERC 721 and ERC 20 tokens? ERC-721 Token is an example of NFT and ERC-20 token is an example of fungible token. ERC-721 standard is basicallly just a specification that says how the smart contract is supposed to work. It describes the type of functions that the smart contract implements like what the arguments for these functions are and the return values.

That is an overview of what NFTs, fungible tokens, ERC-721, ERC-20 are. Now let's build our own NFTs.

https://www.youtube.com/watch?v=YPbgjPPC1d0

Dependencies Required for building NFTs

1. We need to have **Node.js** already installed. To check if you already have Node.js installed, simple go to the command terminal and type node.js. If it does not show any error, then you already have node.js installed. If it shows errors, then you can install it by going to the website.

   Or go to terminal and type -> *npm install node.js* ->This is also an easy way.

2. We will be using Ganche as our personal blockchain. You can download Ganache from this link. Once downloaded, installed and launched, you should be able to see two options - "Quickstart" and "New Workspace". Click on the first option, Quickstart - and you will see 10 test accounts with 100 ethers in each of them. We will be using these public addresses for our project - these are test ethers and that is all.

   What is a personal blockchain and why will be using Ganache? A personal blockchain is like a real blockchain and anyone in public can run it but it runs on our own local computer on a closed Network. Ganache is basically a process that runs on a computer that spins up this blockchain and runs on a server so we can use this to develop smart contracts that run tests against it and we can run scripts against the network develop applications and actually talk to this blockchain.

3. Another tool we will be needing is Truffle. You can get truffle by running the command - npm install -g truffle@5.0 on the terminal.

   We need truffle framework to develop Ethereum smart contracts with the solidity programming language. It is important to use the exact version of the truffle as mentioned above because we will be writing our contracts in 0.5.0 versions.

4. Metamask - Metamask, is a digital wallet that connects with blockchain network and has test networks where ewe can run our smart contracts. If you still don't have your metamask account then you should install it from going to google store and clicking install and then add it as an extension.

5. Finally, we are using a GitHub repository project because building it from scratch will take a lot of time. We will also be using open zeppelin libraries. Open Zeppelin has designed this go to codes for tokens and ownership and many different blockchain tools. So, we import them in our ocde and use them to make our project easier and smooth to run. You need to clone the repository into your local computer. The repository contains truffle-config files which will automatically connect to ganache and run easily.

   To clone, go to terminal and write :

   git clone https://github.com/PunyajaMish/nft NFTs

   We have named the folder NFTs. Make sure every library being downloaded is in the same folder in your computer so that the code is able to find it when you are importing.

   Next, also download the open zeppelin library. This link is the link to their repository. And to install go to terminal (in the same directory where you have been) and type -

   npm install @openzeppelin/contracts

   You will also find this in the README file. README files are a great place to look at if you get stuck because they have information about the project in detail for people to understand and work with it.

6. Text editor : VS Code. We need a text editor where we can code our smart contract. For this project, we are using VS Code. You can also use Sublime Text if you like.

Let's start

1. Open the text editor and open the folder NFTs. Go to File > Open folder and then go to te directory where you have been installing everything.

2. Make surge Ganache is running on localhost for it to be connected to truffle so that we are talking to blockchain.

3. Now let's look at the smart contract : Color.sol

   We are using ERC721 token file's help from open zeppelin because it has the standard code and all functionalities for an NFT to run.

4. I have also used truffle-flattener. What is truffle flattener? It is a tool that makes sure that the code we are using from open zeppelin remains the same. Open Zeppelin keeps updating their code and after a few weeks you might see that the compiler version has changed to 0.8.6 or the files we were using are not there anymore. For example, we are using ERC721Full.sol because it also has link to ERCMetadata, ERC721Enumerable and ERC165. By using truffle flattener, all the files are compressed and added into the same file ERCFull.sol which we have added in out src > contracts for using. It has already been done but to do so first this is how we do : type this in terminal :

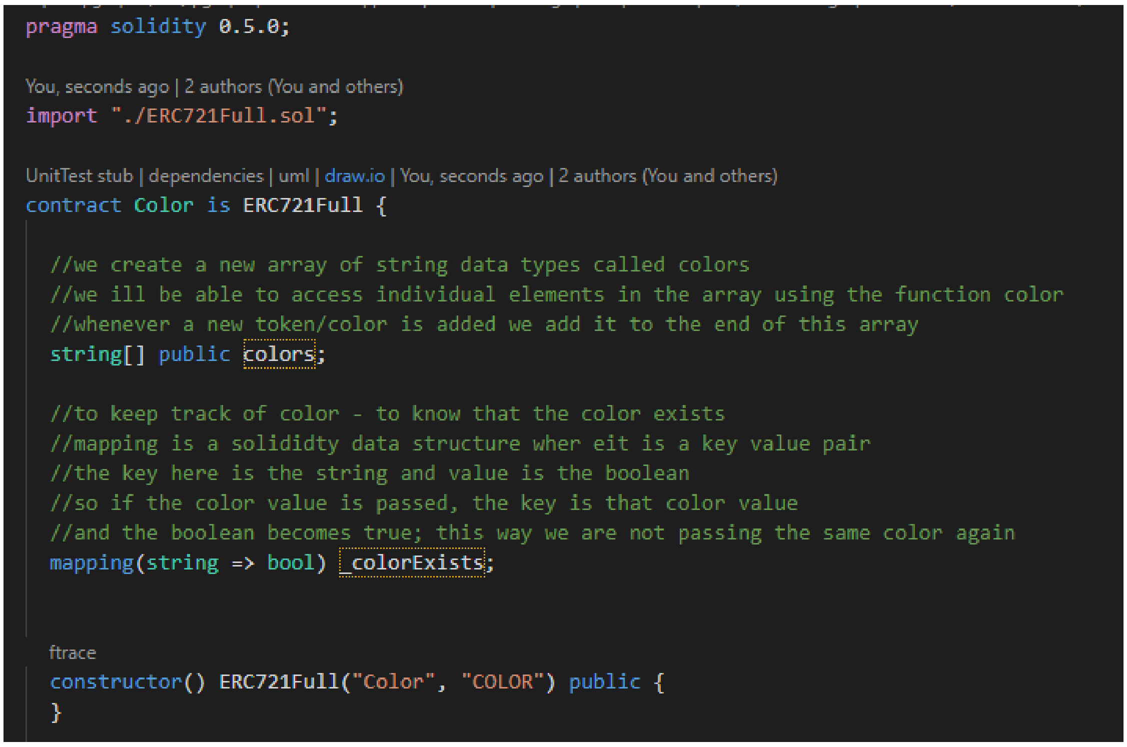
   ./node_modules/.bin/truffle-flattener file to be flattened path > where to be flattened folder path and name

Smart contract

The NFT we are building is Colors - as the name suggests. Let's understand the smart contract we have :

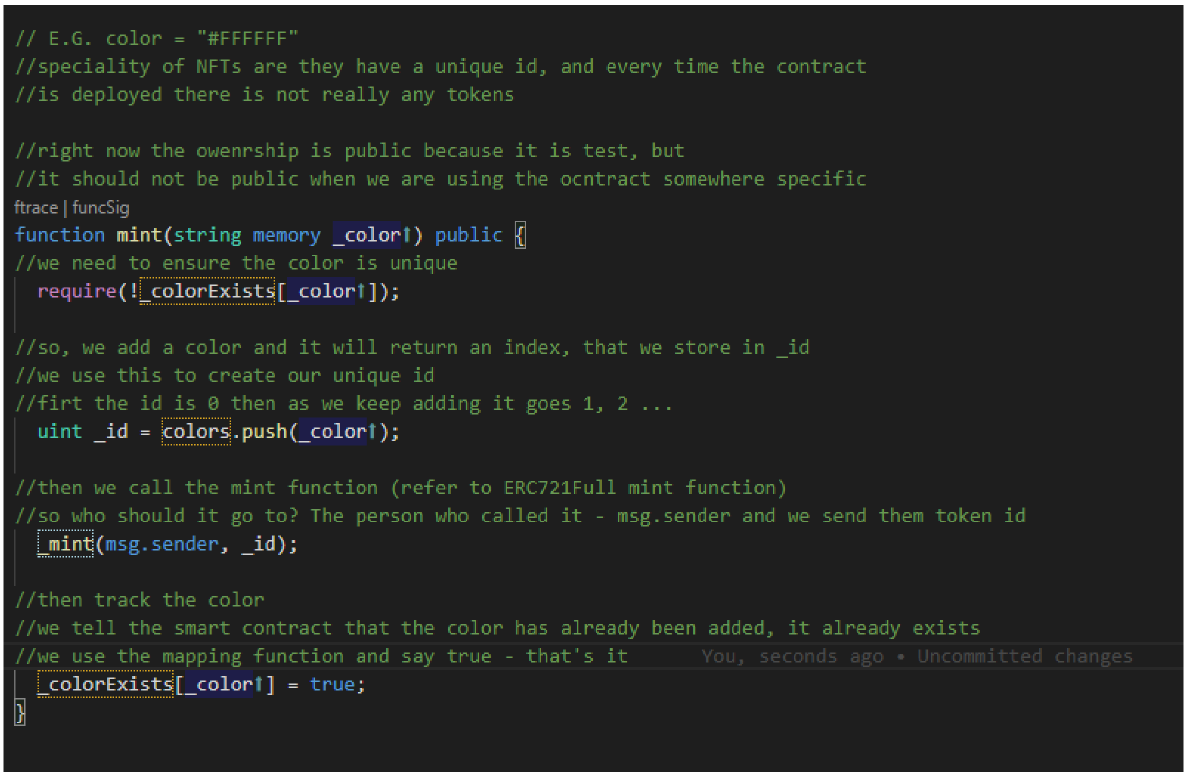
contract Color is ERC721Full : This means that our contract will be using ERC721Full.sol

I have added comments to the code to understand each line of the code.



```
pragma solidity 0.5.0;

You, seconds ago | 2 authors (You and others)
import "./ERC721Full.sol";

UnitTest stub | dependencies | uml | draw.io | You, seconds ago | 2 authors (You and others)
contract Color is ERC721Full {

  //we create a new array of string data types called colors
  //we ill be able to access individual elements in the array using the function color
  //whenever a new token/color is added we add it to the end of this array
  string[] public colors;

  //to keep track of color - to know that the color exists
  //mapping is a solididty data structure wher eit is a key value pair
  //the key here is the string and value is the boolean
  //so if the color value is passed, the key is that color value
  //and the boolean becomes true; this way we are not passing the same color again
  mapping(string => bool) _colorExists;


  ftrace
  constructor() ERC721Full("Color", "COLOR") public {
  }
```

Now let's look at the code. We start with constructor. ERC721 has a constructor that takes 2 parameters, so we need to pass 2 paramteters as well. The first parameter - name will be "Color" because that is what we are making and symbol will be "COLOR". We also have to set the visibility of the function so all the functions of smart contracts have visibility there either public or private. So we will make the constructor public.



```
// E.G. color = "#FFFFFF"
//speciality of NFTs are they have a unique id, and every time the contract
//is deployed there is not really any tokens

//right now the owenrship is public because it is test, but
//it should not be public when we are using the ocntract somewhere specific
ftrace | funcSig
function mint(string memory _color) public {
//we need to ensure the color is unique
  require(!_colorExists[_color]);

//so, we add a color and it will return an index, that we store in _id
//we use this to create our unique id
//firt the id is 0 then as we keep adding it goes 1, 2 ...
  uint _id = colors.push(_color);

//then we call the mint function (refer to ERC721Full mint function)
//so who should it go to? The person who called it - msg.sender and we send them token id
  _mint(msg.sender, _id);

//then track the color
//we tell the smart contract that the color has already been added, it already exists
//we use the mapping function and say true - that's it       You, seconds ago • Uncommitted changes
  _colorExists[_color] = true;
}
```

Tests Folder

In order to check if the code is working and connecting to blockchain, we need a test.js file which is in the test folder. The test was written using *Mocha Framework* - (website). This helps write tests for the smart contracts and also the chai assertion library and this allows us to write successful test examples.

First we import the file smart contract by - **const Color = artifacts.require('./Color.sol')**

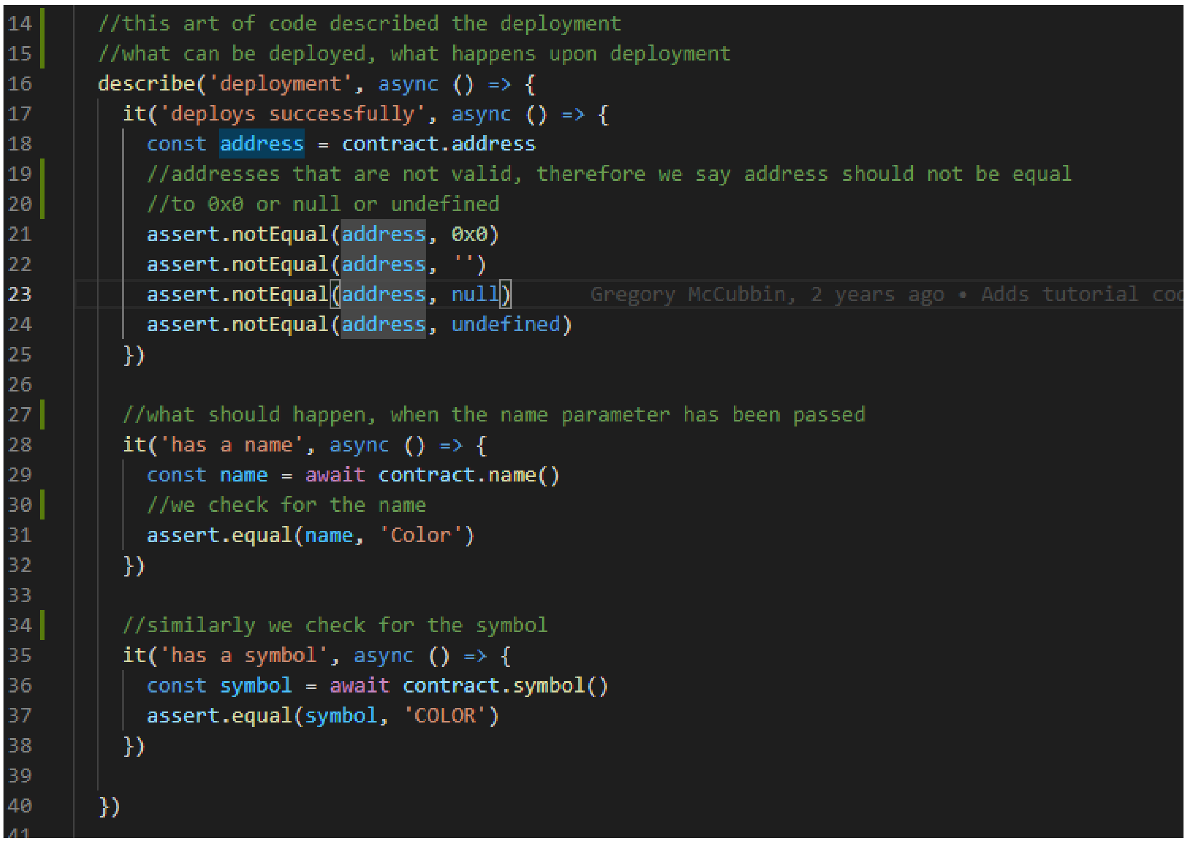
We also have abis folders which is the abstract binary interface of the file - basically a JSON description of what the smart contract looks like, the functions, information like what the address wants to put on the network and so on.

First we import chai - **require('chai')**

.use(require('chai-as-promised'))

.should()

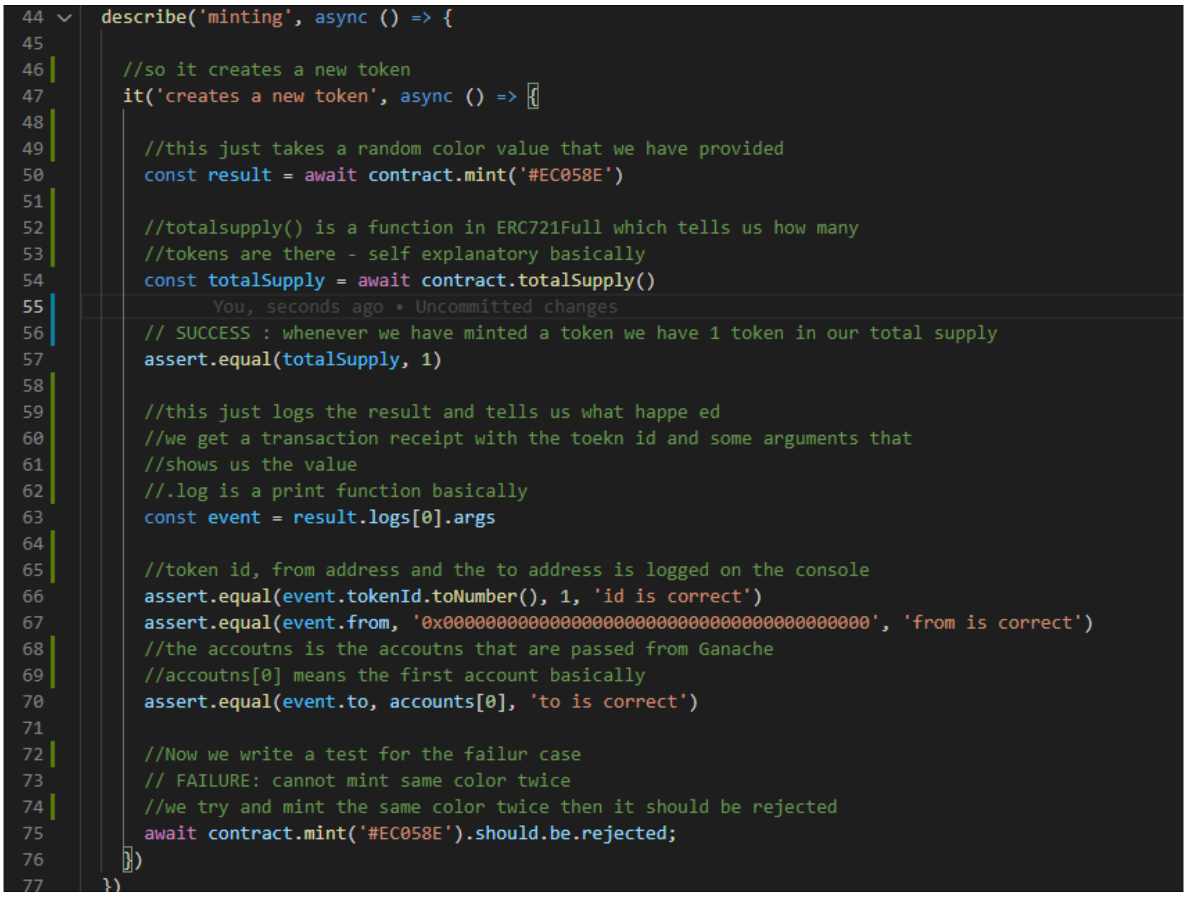
Describe Deployment



```javascript
//this art of code described the deployment
//what can be deployed, what happens upon deployment
describe('deployment', async () => {
  it('deploys successfully', async () => {
    const address = contract.address
    //addresses that are not valid, therefore we say address should not be equal
    //to 0x0 or null or undefined
    assert.notEqual(address, 0x0)
    assert.notEqual(address, '')
    assert.notEqual(address, null)       Gregory McCubbin, 2 years ago • Adds tutorial coc
    assert.notEqual(address, undefined)
  })

  //what should happen, when the name parameter has been passed
  it('has a name', async () => {
    const name = await contract.name()
    //we check for the name
    assert.equal(name, 'Color')
  })

  //similarly we check for the symbol
  it('has a symbol', async () => {
    const symbol = await contract.symbol()
    assert.equal(symbol, 'COLOR')
  })

})
```

Describe Minting

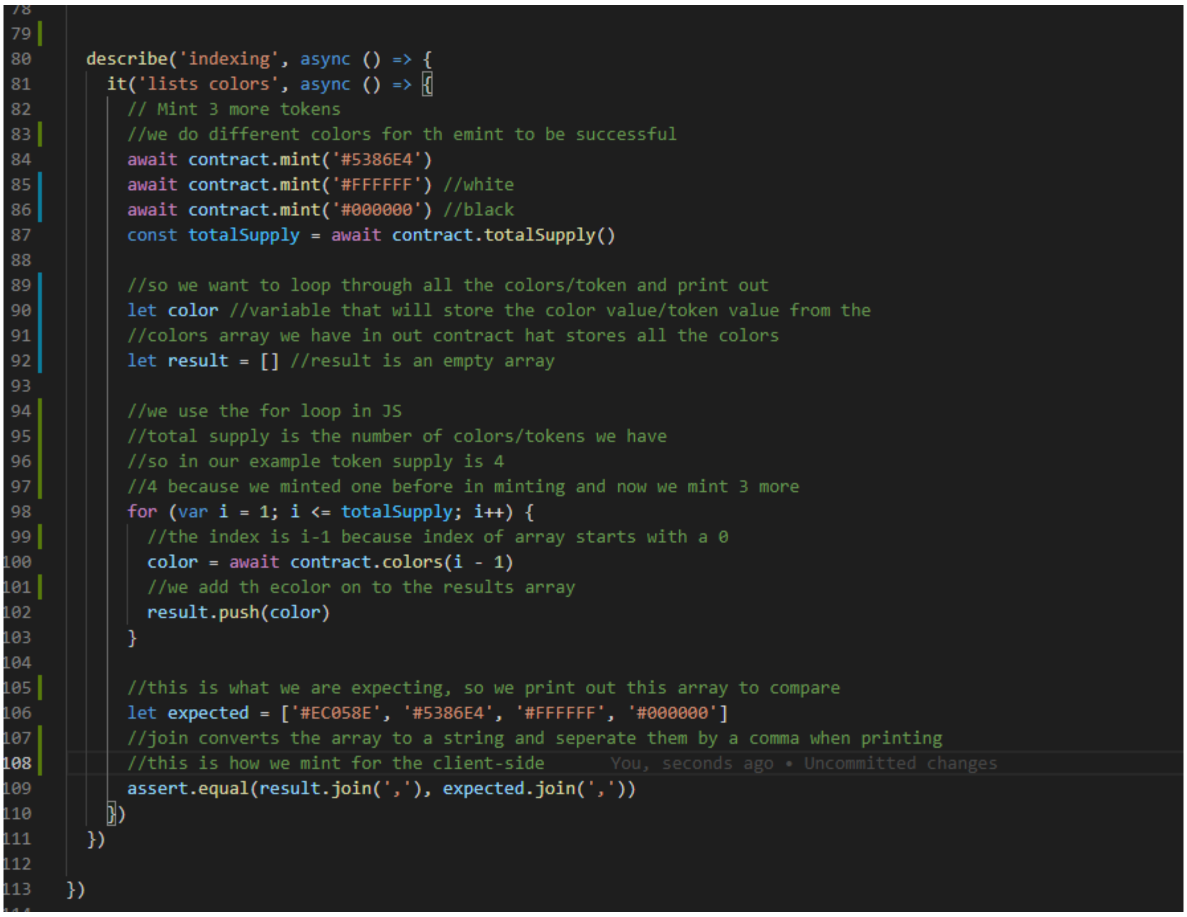
The function mint we are using in our smart contract is from ERC721Full and it uses the function ***transfer*** also described in ERC721 - so basically it tells whom the token was minted to and the token id as well.



```javascript
describe('minting', async () => {

  //so it creates a new token
  it('creates a new token', async () => {

    //this just takes a random color value that we have provided
    const result = await contract.mint('#EC058E')

    //totalsupply() is a function in ERC721Full which tells us how many
    //tokens are there - self explanatory basically
    const totalSupply = await contract.totalSupply()
          You, seconds ago • Uncommitted changes
    // SUCCESS : whenever we have minted a token we have 1 token in our total supply
    assert.equal(totalSupply, 1)

    //this just logs the result and tells us what happe ed
    //we get a transaction receipt with the toekn id and some arguments that
    //shows us the value
    //.log is a print function basically
    const event = result.logs[0].args

    //token id, from address and the to address is logged on the console
    assert.equal(event.tokenId.toNumber(), 1, 'id is correct')
    assert.equal(event.from, '0x0000000000000000000000000000000000000000', 'from is correct')
    //the accoutns is the accoutns that are passed from Ganache
    //accoutns[0] means the first account basically
    assert.equal(event.to, accounts[0], 'to is correct')

    //Now we write a test for the failur case
    // FAILURE: cannot mint same color twice
    //we try and mint the same color twice then it should be rejected
    await contract.mint('#EC058E').should.be.rejected;
  })
})
```

Describing Indexing

If we want to see all the colors that exist, we need this part of the code



```
  describe('indexing', async () => {
    it('lists colors', async () => {
      // Mint 3 more tokens
      //we do different colors for th emint to be successful
      await contract.mint('#5386E4')
      await contract.mint('#FFFFFF') //white
      await contract.mint('#000000') //black
      const totalSupply = await contract.totalSupply()

      //so we want to loop through all the colors/token and print out
      let color //variable that will store the color value/token value from the
      //colors array we have in out contract hat stores all the colors
      let result = [] //result is an empty array

      //we use the for loop in JS
      //total supply is the number of colors/tokens we have
      //so in our example token supply is 4
      //4 because we minted one before in minting and now we mint 3 more
      for (var i = 1; i <= totalSupply; i++) {
        //the index is i-1 because index of array starts with a 0
        color = await contract.colors(i - 1)
        //we add th ecolor on to the results array
        result.push(color)
      }

      //this is what we are expecting, so we print out this array to compare
      let expected = ['#EC058E', '#5386E4', '#FFFFFF', '#000000']
      //join converts the array to a string and seperate them by a comma when printing
      //this is how we mint for the client-side       You, seconds ago • Uncommitted changes
      assert.equal(result.join(','), expected.join(','))
    })
  })

})
```

Migrations folder

We need migrations file like the initial_migration file to put the smart contracts on the blockchain and this is just like migrating a database in some other context where you're adding new tables or maybe you're moving data from one database to another. That's kind of the idea with smart contract migrations - you're moving some our contracts one place to another, you're migrating them or you're changing the state from one state to another.

So, we create another migrations file names 2_deploy_contratcs.js : 2 is for the compiler to know that it should be ran second. We have the same code as the initial_migration, except we just change the Migrations to Color to tell that Color.sol smart contract should run.

To finally migrate it we go to terminal and type -

truffle migrate

If it does not show any errors that means it was migrated successfully.

Next we type in

truffle console

First we write the smart contracts in solidity, then we interact with JavaScript to test them. JavaScript has the actual functions that we need to ensue the smart contract is running. We do this iin console to have lightweight runtime environment. To get the smart contract we type in :

Color.deployed()

This will get a deployed copy of the smart contract and return it. Now since we have *async await* method, therefore we need to type in :

color=await Color.deployed()

This is what we have in our test file as well.

Client-side portion of the Application

Now we will look into the Components folder - App.js and App.css. Mostly App.js because it has all the functionalities. App.css is just how the designing needs to be. App.js is written in React.js. React implements ***render()*** method that takes input data and returns what to display. The syntax type is XML-like syntax called JSX. Input data passed into component can be accessed by ***render()*** via ***this.props***. In short we mix and match HTML and JS code. Check out the React.org website to know more about React if you do not know.

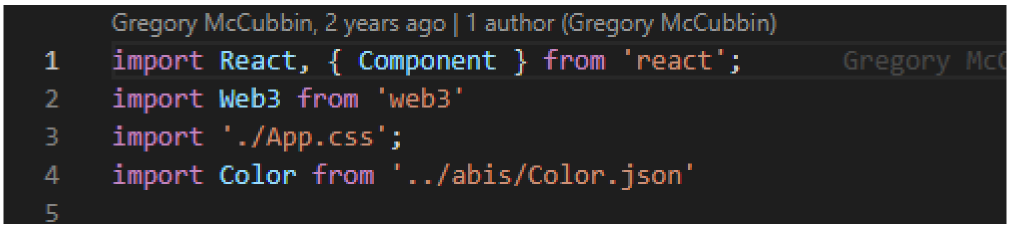
We can just run react project by typing in terminal (new window) :

npm run start

Then we type ***Yes*** and we get the starter kit of the DApp University since they have one that wires up with React and truffle and ganache. This allows us to avoid writing to much design code - CSS.

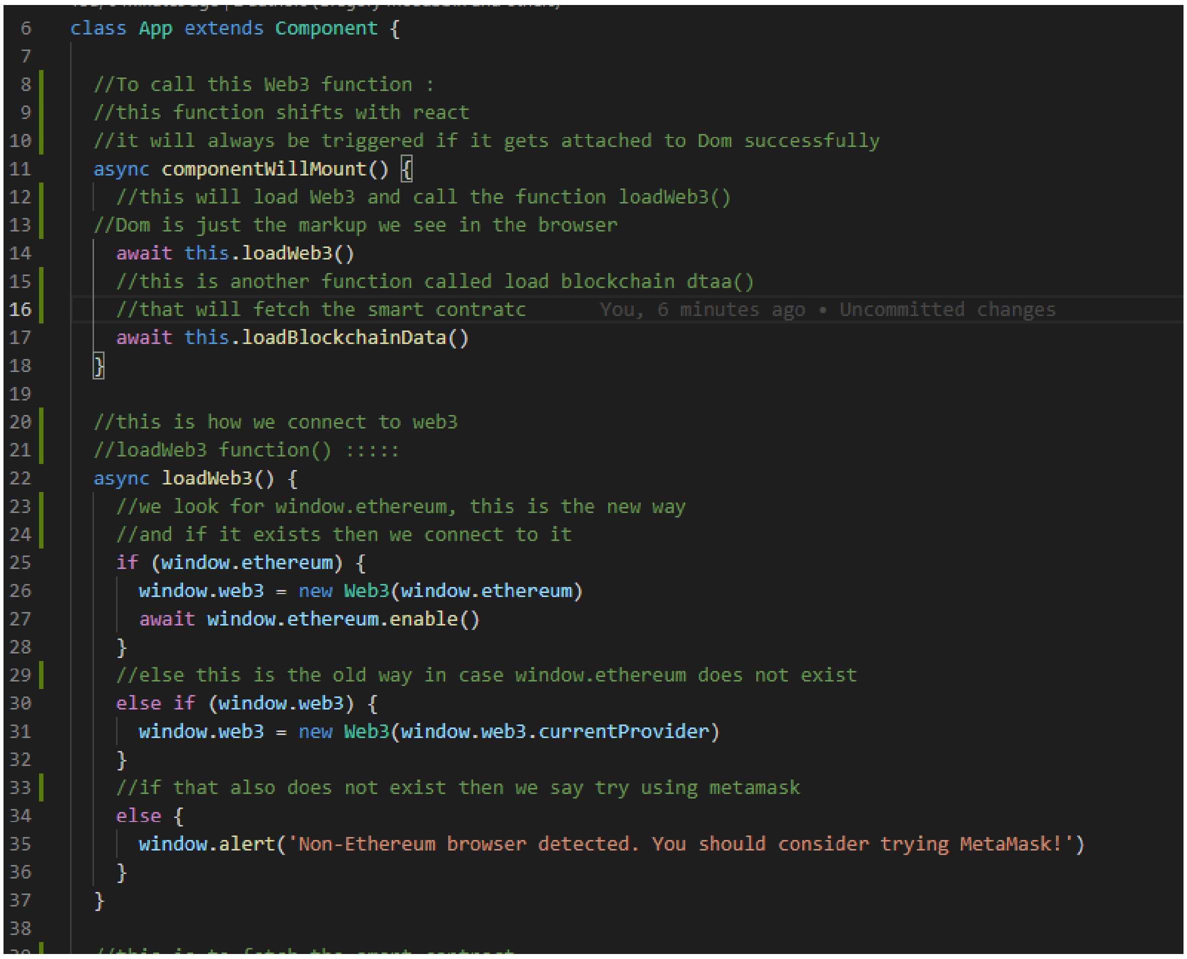
Required import - Web3

We are using React.js, App.css and Color.json file, but we need one more important thing and that is ***Web3***. Web3 is a JS library that basically converts our web browser into a blockchain. on the web3.js site, we can get instance of web3.js - it gives an Ethereum provider. Out Ethereum provider is Metamask in this case. Make sure we are running on localhost port 7545.



```
import React, { Component } from 'react';
import Web3 from 'web3'
import './App.css';
import Color from '../abis/Color.json'
```

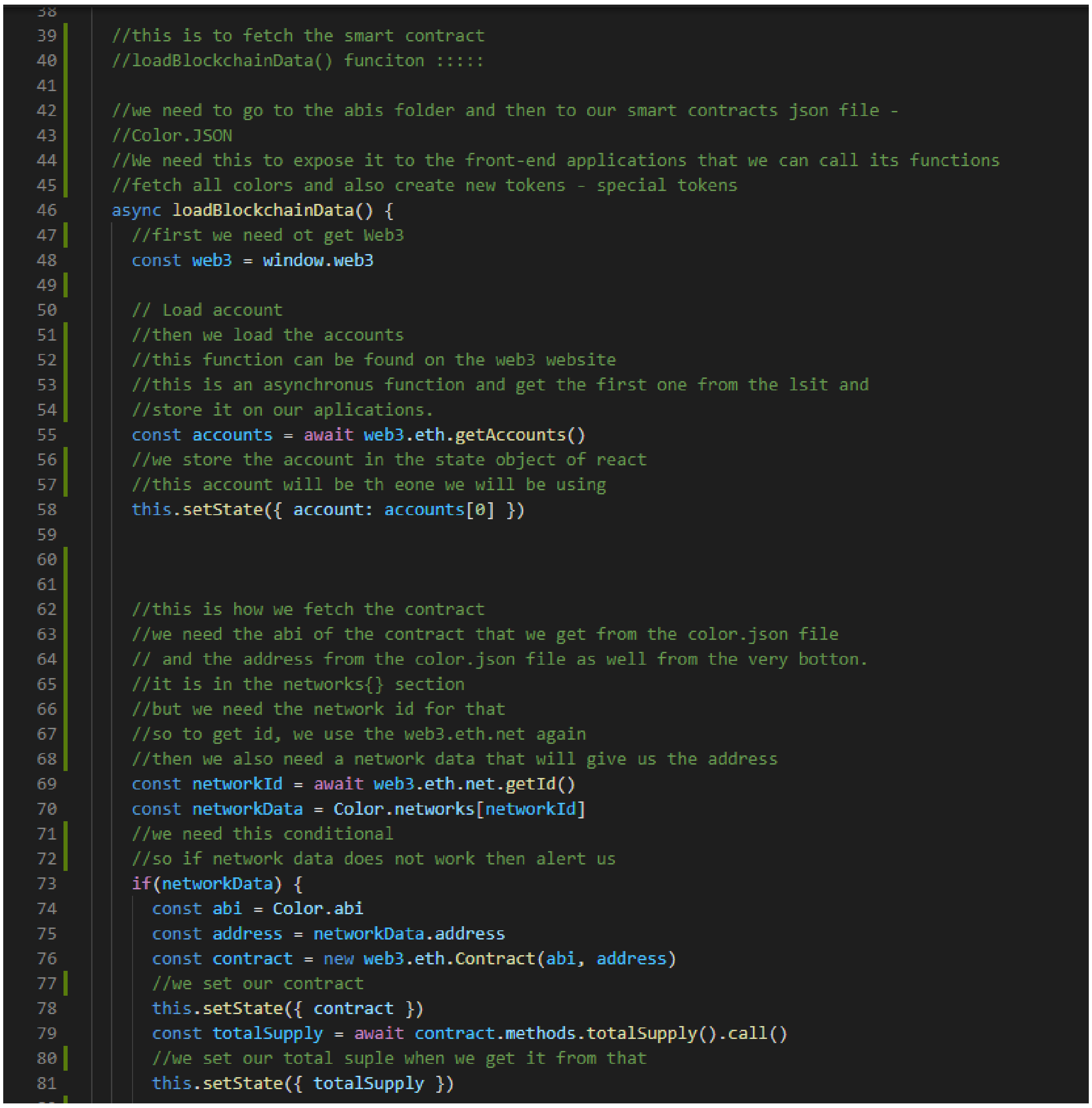
We need two functions - one that will connect to the Web3 and one that will connect and fetch the smart contract finally to run. Go to [this link about Web3](#) and you will find all the functions we have used in here.



```
class App extends Component {

  //To call this Web3 function :
  //this function shifts with react
  //it will always be triggered if it gets attached to Dom successfully
  async componentWillMount() {
    //this will load Web3 and call the function loadWeb3()
  //Dom is just the markup we see in the browser
    await this.loadWeb3()
    //this is another function called load blockchain dtaa()
    //that will fetch the smart contratc      You, 6 minutes ago • Uncommitted changes
    await this.loadBlockchainData()
  }

  //this is how we connect to web3
  //loadWeb3 function() :::::
  async loadWeb3() {
    //we look for window.ethereum, this is the new way
    //and if it exists then we connect to it
    if (window.ethereum) {
      window.web3 = new Web3(window.ethereum)
      await window.ethereum.enable()
    }
    //else this is the old way in case window.ethereum does not exist
    else if (window.web3) {
      window.web3 = new Web3(window.web3.currentProvider)
    }
    //if that also does not exist then we say try using metamask
    else {
      window.alert('Non-Ethereum browser detected. You should consider trying MetaMask!')
    }
  }

```

Fetch our smart contract

After we are connected with Web3, we still need to fetch the smart contract to connect it to the web browser of the client-side. We do this by using network id, ABI of the contract and web3.

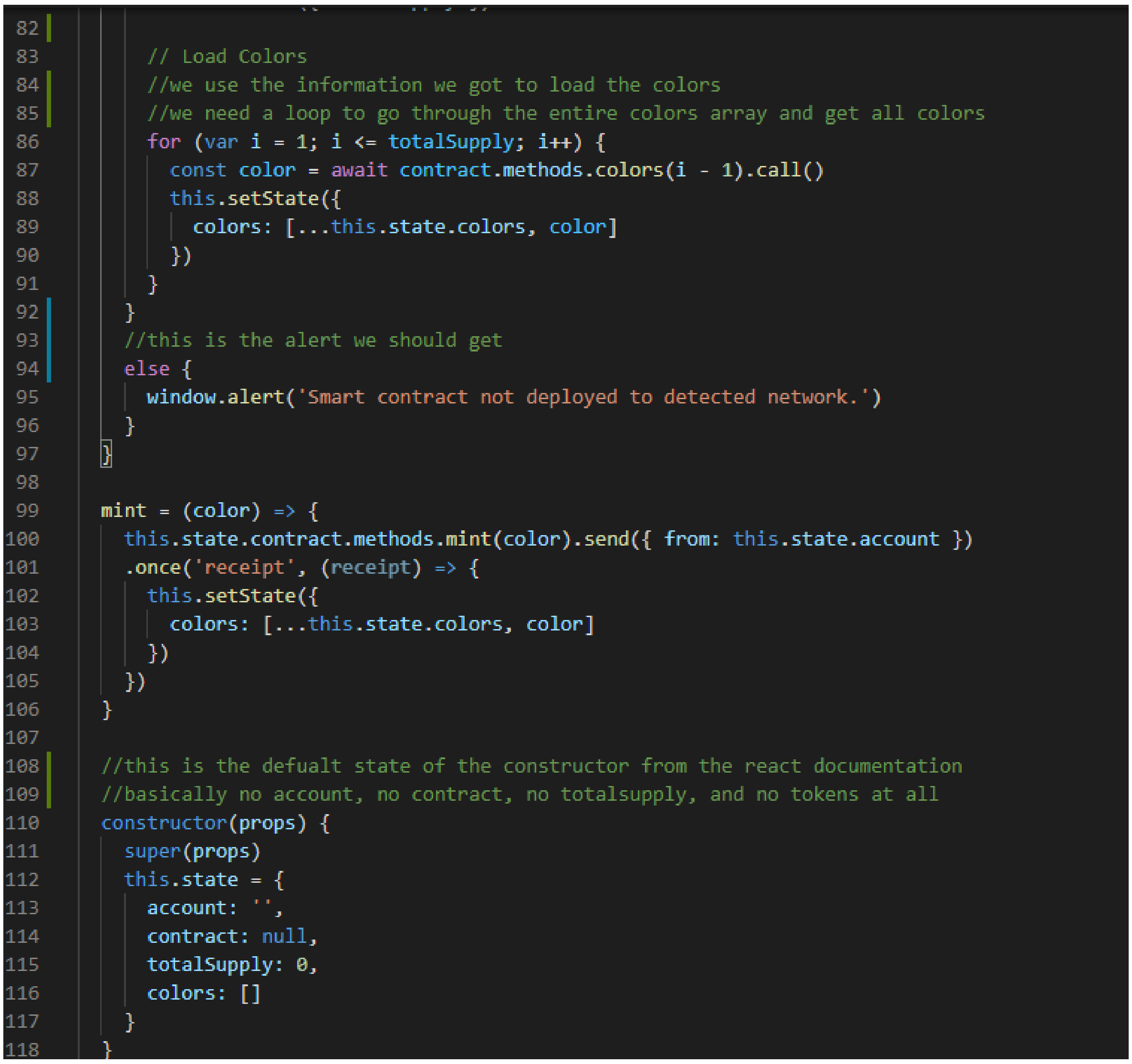


```
  //this is to fetch the smart contract
  //loadBlockchainData() funciton :::::

  //we need to go to the abis folder and then to our smart contracts json file -
  //Color.JSON
  //We need this to expose it to the front-end applications that we can call its functions
  //fetch all colors and also create new tokens - special tokens
  async loadBlockchainData() {
    //first we need ot get Web3
    const web3 = window.web3

    // Load account
    //then we load the accounts
    //this function can be found on the web3 website
    //this is an asynchronus function and get the first one from the lsit and
    //store it on our aplications.
    const accounts = await web3.eth.getAccounts()
    //we store the account in the state object of react
    //this account will be th eone we will be using
    this.setState({ account: accounts[0] })



    //this is how we fetch the contract
    //we need the abi of the contract that we get from the color.json file
    // and the address from the color.json file as well from the very botton.
    //it is in the networks{} section
    //but we need the network id for that
    //so to get id, we use the web3.eth.net again
    //then we also need a network data that will give us the address
    const networkId = await web3.eth.net.getId()
    const networkData = Color.networks[networkId]
    //we need this conditional
    //so if network data does not work then alert us
    if(networkData) {
      const abi = Color.abi
      const address = networkData.address
      const contract = new web3.eth.Contract(abi, address)
      //we set our contract
      this.setState({ contract })
      const totalSupply = await contract.methods.totalSupply().call()
      //we set our total suple when we get it from that
      this.setState({ totalSupply })
```

Load the NFT tokens onto the array

Once we are all connected, we should be able to access and load the NFT tokens onto an array so if we want to print them off, we should be able to do so.

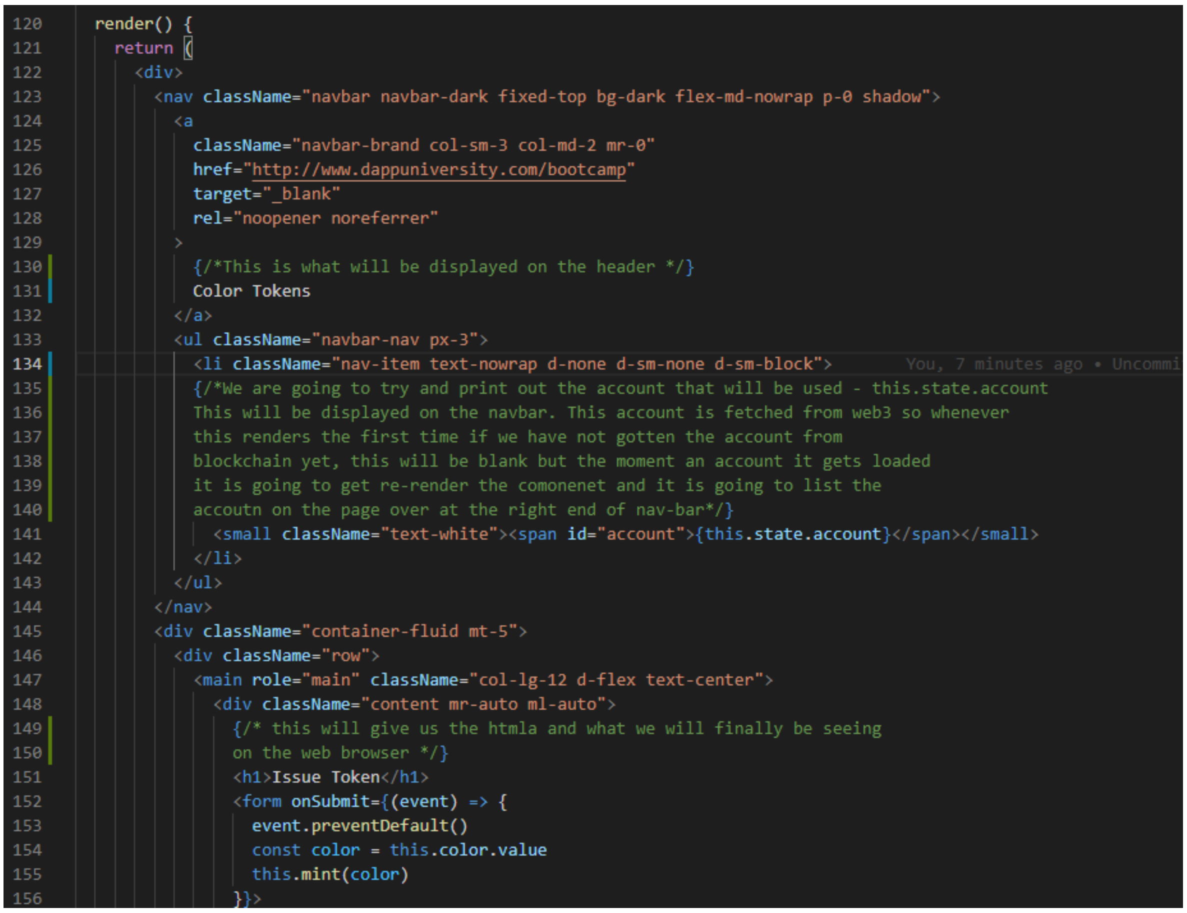
We also need a default constructor that sets the contract as null and total supply as 0 - basically the default values of everything at the beginning when no NFTs have been minted.



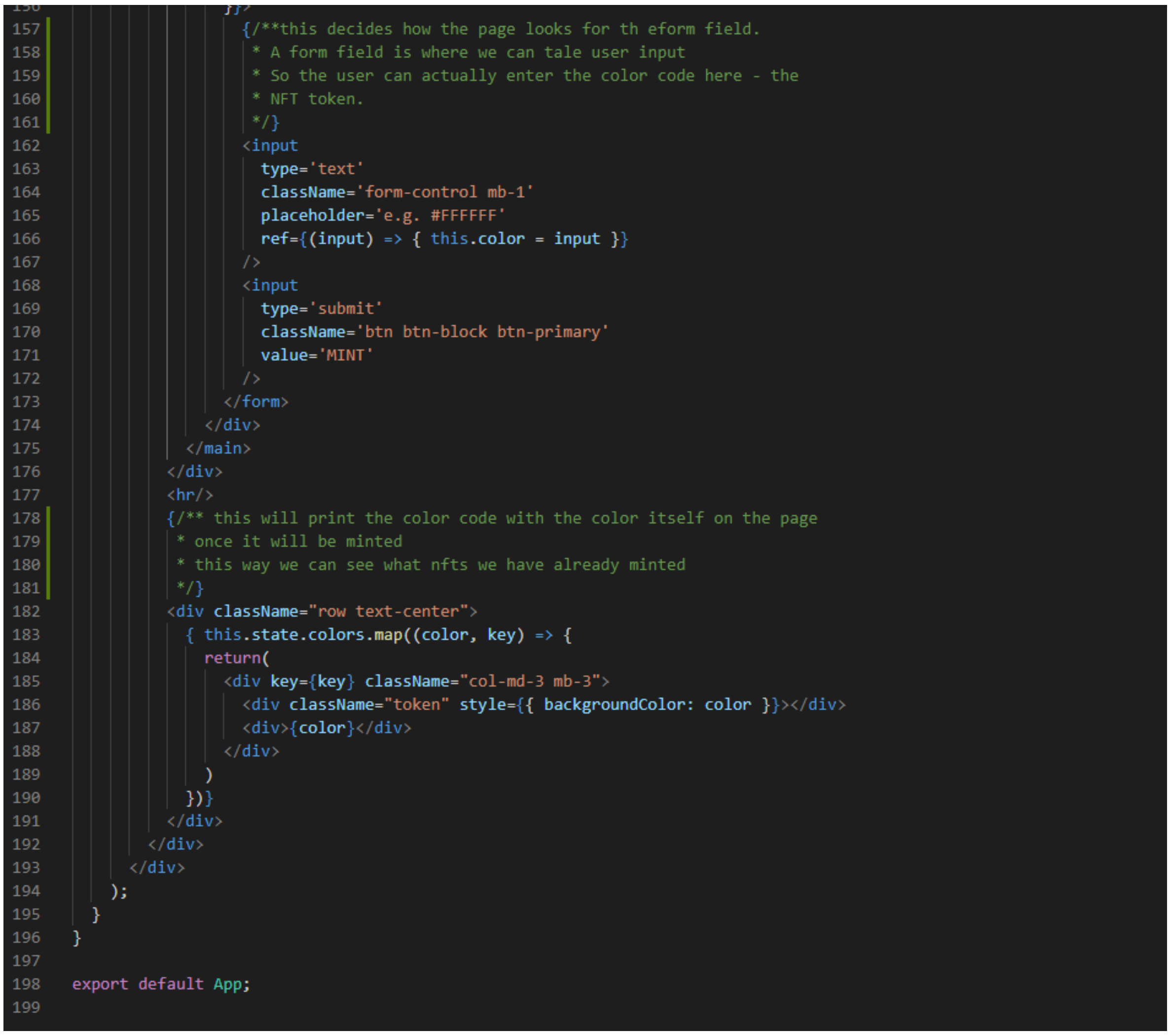
```
      // Load Colors
      //we use the information we got to load the colors
      //we need a loop to go through the entire colors array and get all colors
      for (var i = 1; i <= totalSupply; i++) {
        const color = await contract.methods.colors(i - 1).call()
        this.setState({
          colors: [...this.state.colors, color]
        })
      }
    }
    //this is the alert we should get
    else {
      window.alert('Smart contract not deployed to detected network.')
    }
  }

  mint = (color) => {
    this.state.contract.methods.mint(color).send({ from: this.state.account })
    .once('receipt', (receipt) => {
      this.setState({
        colors: [...this.state.colors, color]
      })
    })
  }

  //this is the defualt state of the constructor from the react documentation
  //basically no account, no contract, no totalsupply, and no tokens at all
  constructor(props) {
    super(props)
    this.state = {
      account: '',
      contract: null,
      totalSupply: 0,
      colors: []
    }
  }
```

React

React, as mentioned, i a mixture of HTML and JS. As soon as ***render()*** is called, we are running react.js. In this part of the code we are building the navigation bar where we print the header - ***Color Tokens*** because that is what our NFT is and then we also print the address of the owner who is trying to mind the token. This way we can see what account is being used.



```
  render() {
    return (
      <div>
        <nav className="navbar navbar-dark fixed-top bg-dark flex-md-nowrap p-0 shadow">
          <a
            className="navbar-brand col-sm-3 col-md-2 mr-0"
            href="http://www.dappuniversity.com/bootcamp"
            target="_blank"
            rel="noopener noreferrer"
          >
            {/*This is what will be displayed on the header */}
            Color Tokens
          </a>
          <ul className="navbar-nav px-3">
            <li className="nav-item text-nowrap d-none d-sm-none d-sm-block">        You, 7 minutes ago • Uncommi
            {/*We are going to try and print out the account that will be used - this.state.account
            This will be displayed on the navbar. This account is fetched from web3 so whenever
            this renders the first time if we have not gotten the account from
            blockchain yet, this will be blank but the moment an account it gets loaded
            it is going to get re-render the comonenet and it is going to list the
            accoutn on the page over at the right end of nav-bar*/}
              <small className="text-white"><span id="account">{this.state.account}</span></small>
            </li>
          </ul>
        </nav>
        <div className="container-fluid mt-5">
          <div className="row">
            <main role="main" className="col-lg-12 d-flex text-center">
              <div className="content mr-auto ml-auto">
                {/* this will give us the htmla and what we will finally be seeing
                on the web browser */}
                <h1>Issue Token</h1>
                <form onSubmit={(event) => {
                  event.preventDefault()
                  const color = this.color.value
                  this.mint(color)
                }}>
```



```
                  {/**this decides how the page looks for th eform field.
                   * A form field is where we can tale user input
                   * So the user can actually enter the color code here - the
                   * NFT token.
                   */}
                  <input
                    type='text'
                    className='form-control mb-1'
                    placeholder='e.g. #FFFFFF'
                    ref={(input) => { this.color = input }}
                  />
                  <input
                    type='submit'
                    className='btn btn-block btn-primary'
                    value='MINT'
                  />
                </form>
              </div>
            </main>
          </div>
          <hr/>
          {/** this will print the color code with the color itself on the page
           * once it will be minted
           * this way we can see what nfts we have already minted
           */}
          <div className="row text-center">
            { this.state.colors.map((color, key) => {
              return(
                <div key={key} className="col-md-3 mb-3">
                  <div className="token" style={{ backgroundColor: color }}></div>
                  <div>{color}</div>
                </div>
              )
            })}
          </div>
        </div>
      </div>
    );
  }
}

export default App;
```

Running the program finally

Now, that we are all set - let us run the contract on blockchain. There are a few steps we need to follow even here in order to ensure that it is connected and running properly. First of all we need to run the React and make sure that a new tab opens on the browser with the output we desire.

React

1. To render(), we need to ensure we have react installed. Open Terminal, and go the same directory where everything is. Since, we previously did install node_modules, and the repository that you cloned also has node_modules, there will be react. To check simple type in - ***npm run start***

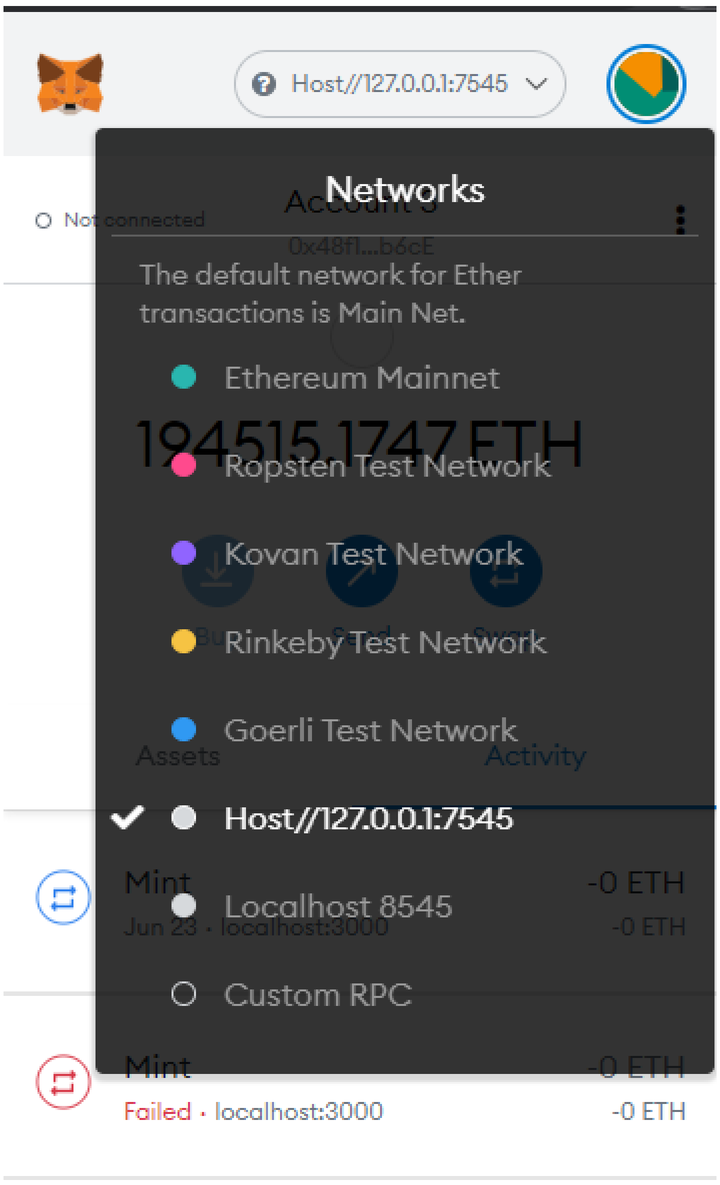
   If it shows error then that means, you do not have react or are in the wrong folder(not where we have our entire project in). Else simply run ***npm install --save react***.

   ### Metamask Network

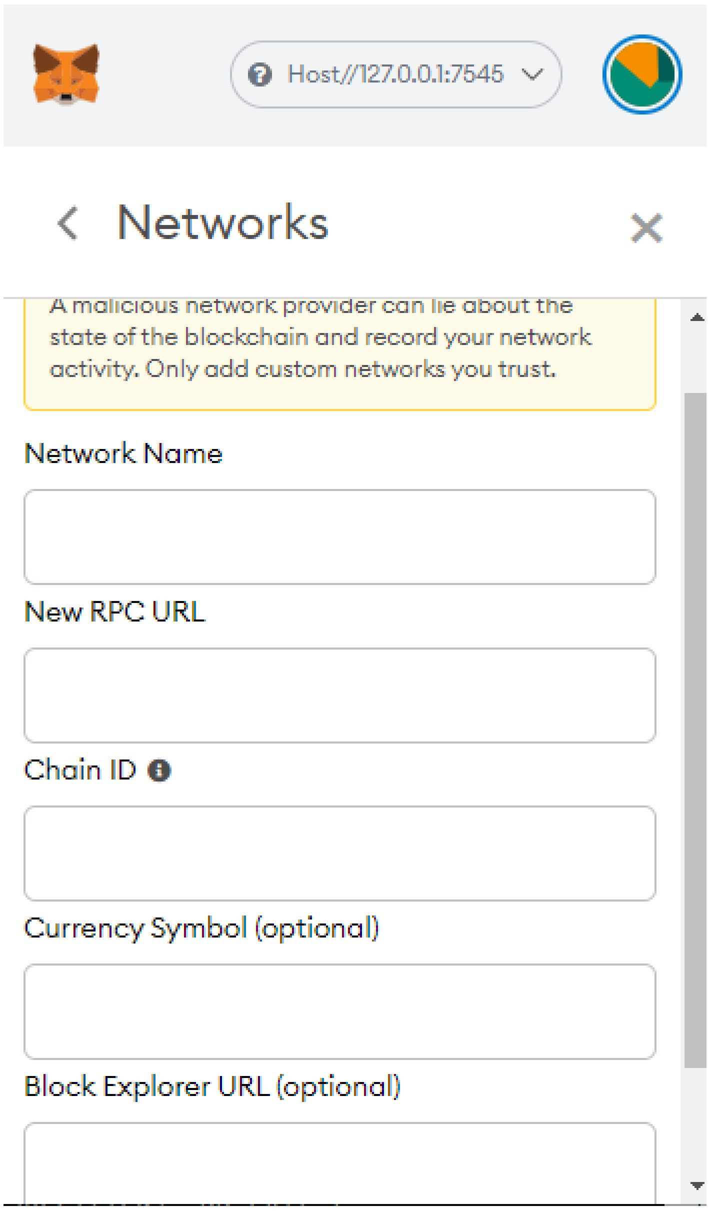
2. Next, once ***npm run start*** command runs, a new tab will load on to your web browser. Now we need to ensure that our Metamask is on the right network and also we are on the right account.

   The network we are in is **Host 127.0.0.1 : Port 7545**.

   To change the network click on the dropdown at the top of the metamask extension pop up.



There is a possibility that you might not have the option of this network. So, we can add the network. Click on ***Custom RPC*** - which is at the very end of the list above. You should see :


Host//127.0.0.1:7545

Networks

A malicious network provider can lie about the state of the blockchain and record your network activity. Only add custom networks you trust.

Network Name

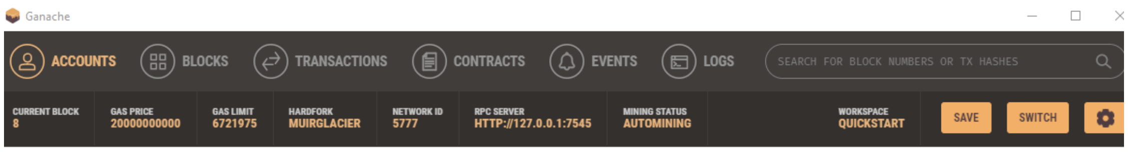
New RPC URL

Chain ID

Currency Symbol (optional)

Block Explorer URL (optional)

How to know what the host name and rpc url is? Open Ganache and you will find that at the top bar - it has all the information you need



CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE
8	20000000000	6721975	MUIRGLACIER	5777	HTTP://127.0.0.1:7545	AUTOMINING	QUICKSTART

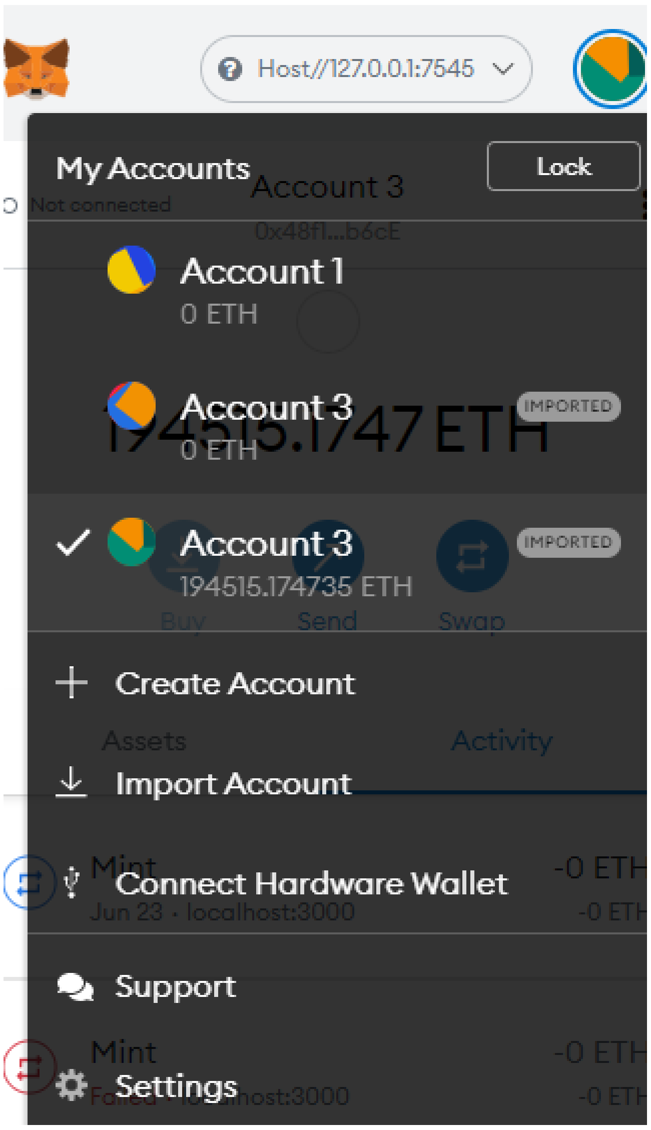
Network Name : (you can give whatever you want. I have given -) Host//127.0.0.1:7545

New RPC URL : HTPP://127.0.0.1:7545

Chain ID : 1337 (if not, once you hit Save, it will show an error and tell you the right Chain ID).

Account

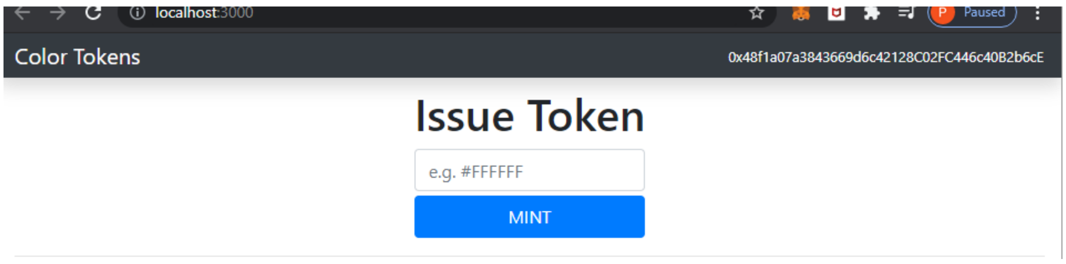
3. Next, we need to ensure we are using one of the test accounts in the Ganache because it has test ethers on there. Click on the small circle that is on the top right corner in Metamask and then hit ***import account***



It will ask you for the private key. Open Ganache, choose any account you want, and hit the key at the very right - that is your private key. These are test accounts so it does not matter, but when you are using the real account, never show you private key to anyone if you do not wish to lose any funds. Enter the Private key and you are good to go! Select that account - as you can see above, i have chosen Account 3 because that is the test account I have to use.

Minting NFTs

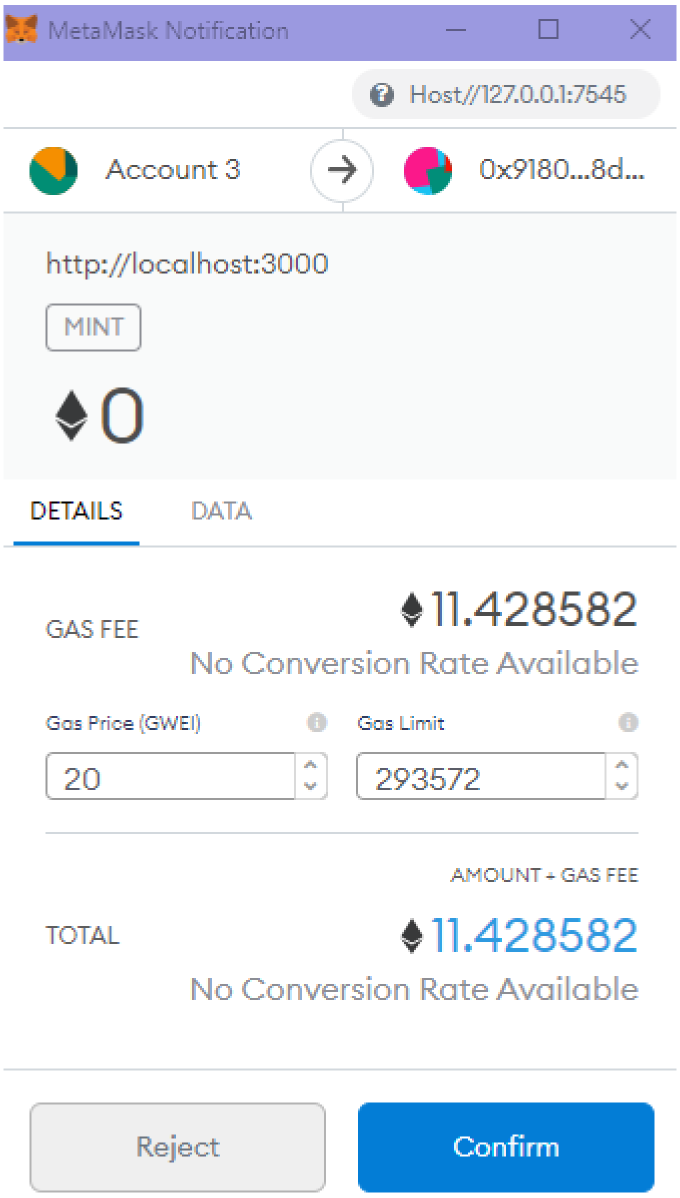
4. Finally, now we can mint the NFT tokens. You should be able to see your tab like this :



We have our header ***Color Tokens *** and the account we are using to mint on the navigation bar.

Go to [Color Hunt website]() and find yourself your favorite color and copy the code of that color. Come back to the Color Tokens page and enter the code with **#** > And hit Mint.

Let's say I am minting the color ***#A239EA*** Once I hit mint, my metamask should pop up with information like this : This is just showing me the gas fee required and asking me to confirm the transaction. Hit confirm and reload the page.


MetaMask Notification

Host//127.0.0.1:7545

Account 3 → 0x9180...8d...

http://localhost:3000

MINT

0

DETAILS DATA

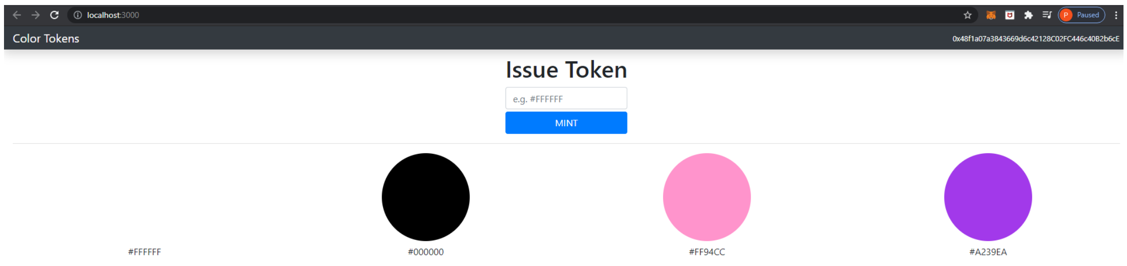
GAS FEE 11.428582
No Conversion Rate Available

Gas Price (GWEI): 20
Gas Limit: 293572

AMOUNT + GAS FEE
TOTAL 11.428582
No Conversion Rate Available

Reject | Confirm

Once you reload, you should be able to see the new NFT token color code, the color and also in your metamask account - ethers will have been deducted.


localhost:3000

Color Tokens 0x48f1a07a3843669d6c42128C02FC446c40B2b6cE

Issue Token

e.g. #FFFFFF

MINT

#FFFFFF #000000 #FF94CC #A239EA

As you can see, I have minted 4 NFTs until now and Ethers have also been deducted form my account and are no longer 100 Ethers.

And that is it! You have learned how to code, build and deploy your own NFTs!