Creating and deploying Ethereum Smart Contracts with Solidity

1. **Meta-mask Chrome Extension**

MetaMask acts both as an Ethereum browser and a wallet. It allows you to interact with smart contracts and dApps on the web without downloading the blockchain or installing any software. You only need to add MetaMask as a Chrome Extension, create a wallet and submit Ether.

Once it is downloaded and added as a Chrome extension, you can either import an already created wallet or create a new wallet. You must have some ethers in your Ethereum wallet to deploy Ethereum smart contract on the network.

2. **Develop an Ethereum Smart Contract**
   - 2.1. **Create a wallet at meta-mask**
   - 2.2. **Select any one test network**
   - 2.3. **Add some dummy Ethers to your wallet**
   - 2.4. **Use editor remix to write the smart contract in Solidity**
   - 2.5. **Create a .sol extension file**
     - 2.5.1. Choose a blockchain platform: You need to choose a blockchain platform that supports NFTs such as Ethereum or Binance Smart Chain.
     - 2.5.2. Smart Contract code: You can use a programming language like Solidity to write the smart contract code.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;


import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";


contract MyNFT is ERC721, ReentrancyGuard {

  // Variables

  address public owner;

  uint256 public maxNftCount = 5;

  uint256 public nftMintedCount = 0;

  uint256 public mintingStartTime = 1641532800; // 7 Jan 2023 00:00:00 GMT
```

```
uint256 public mintingEndTime = 1642152000; // 14 Jan 2023 00:00:00 GMT

// Mapping to keep track of whether a wallet has already minted an NFT or not
mapping(address => bool) private _hasMinted;

// Constructor
constructor(string memory name_, string memory symbol_)
    ERC721(name_, symbol_)
{
    owner = msg.sender;
}

// Function to mint NFTs
function mintNFT(string memory tokenURI) public nonReentrant {
    require(
        block.timestamp >= mintingStartTime && block.timestamp <= mintingEndTime,
        "Minting is not allowed at this time"
    );
    require(
        !_hasMinted[msg.sender],
        "You have already minted an NFT"
    );
    require(
        nftMintedCount < maxNftCount,
        "All NFTs have been minted"
    );
    uint256 tokenId = nftMintedCount + 1;
    _safeMint(msg.sender, tokenId);
    _setTokenURI(tokenId, tokenURI);
```

```
        nftMintedCount += 1;
        _hasMinted[msg.sender] = true;
    }
}
```

In the above code, we have defined a smart contract named MyNFT that extends ERC721 (the standard for NFTs) and ReentrancyGuard (a security feature to prevent reentrant calls).

We have defined several variables such as owner (the address that deploys the contract), maxNftCount (the maximum number of NFTs that can be minted), nftMintedCount (the number of NFTs that have already been minted), mintingStartTime and mintingEndTime (the duration when NFTs can be minted).

We have also defined a mapping named _hasMinted to keep track of whether a wallet has already minted an NFT or not.

The mintNFT function is the function that allows users to mint NFTs. It checks whether minting is allowed at the current time, whether the wallet has already minted an NFT or not, and whether all NFTs have been minted or not. If all conditions are satisfied, the function mints an NFT and sets its metadata.

2.5.3. Compile the Smart Contract code: Once you have written the smart contract code, you need to compile it using a Solidity compiler such as Remix or Truffle.

2.5.4. Deploy the Smart Contract: After compiling

2.6. Deploy contract

3. Test an Ethereum smart contract

3.1. Run all your smart contract methods like transfer, total supply, and balance. These methods are present at the right-hand side of the remix window and you can run all the processes from there itself.

3.2. Transfer some tokens to other ethereum wallet addresses and then check the balance of that address by calling the balance method.

3.3. Get total supply by running the total supply method.

4. **Deploy Ethereum Smart Contracts**

4.1. To make smart contract live, switch to the main ethereum network at metamask

4.2. Add some real ethers.

4.3. Now again, deploy your smart contract using remix as mentioned in the above steps.

4.4. When a smart contract is deployed successfully, visit http://www.etherscan.io and search your smart contract address there. Select your smart contract.

4.5. Now we need to verify your smart contract here, click “verify the contract.”

4.6. Copy smart contract code and paste it at Etherscan. Select the same compiler version that you selected at remix to compile your code.

4.7. Check “optimization” to Yes, if you had selected optimization at remix; otherwise, select No.

4.8. Click Verify.

4.9. It will take a few minutes and your smart contract will be live if no issue occurs.

4.10. We can now run your smart contract methods at Etherscan.

5. **Tools and Technologies required for implementing Ethereum Smart Contracts**
   - **5.1. Truffle**
   - **5.2. Web3.js**
   - **5.3. Visual Studio Code**
   - **5.4. Ganache CLI**
   - **5.5. Node.js**

6. **References :**

   - **6.1.** https://www.blockchainappfactory.com/nft-minting-platform-development
   - **6.2.** https://www.leewayhertz.com/ethereum-smart-contract-tutorial/
   - **6.3.** https://betterprogramming.pub/how-to-deploy-nft-smart-contracts-9271ce5e91c0
   - **6.4.** https://www.dappuniversity.com/articles/how-to-build-a-blockchain-app

NFT Portal – React js App

1. Set up the development environment: We need to set up the development environment to build the application. We can use the Create React App tool to generate a React project.
2. Install the necessary dependencies: We need to install the necessary dependencies such as web3.js, ether.js, and Bootstrap to build the application.
3. Connect to the blockchain network: We need to connect to the blockchain network using web3.js or ether.js to interact with the smart contract.
4. Create the smart contract interface: We need to create the interface for the smart contract that will enable us to claim (mint) NFT to the connected wallet. We can use Solidity to create the smart contract interface.
5. Display NFT metadata: We need to display the NFT image from NFT metadata. We can use IPFS to store the NFT metadata and fetch it from there.
6. Implement error handling: We need to implement error handling to ensure that the application works as expected and handle any errors that may occur.
7. Test the application: We need to test the application to ensure that it works as expected and is bug-free.
8. **Code : web3.js**

```
import Web3 from 'web3';

import { abi } from './MyNFT.json';


const contractAddress = '0x123...'; // Replace with contract address


async function mintNFT() {

  const web3 = new Web3(window.ethereum);

  await window.ethereum.enable(); // Request permission to connect to the Ethereum network

  const contract = new web3.eth.Contract(abi, contractAddress);

  const accounts = await web3.eth.getAccounts();

  const account = accounts[0];

  try {

    await contract.methods.mintNFT('https://example.com/nft-metadata.json').send({ from: account });

    const tokenId = await contract.methods.tokenOfOwnerByIndex(account, 0).call();
```

```
    const tokenURI = await contract.methods.tokenURI(tokenId).call();
    // Display the NFT image from the metadata using tokenURI
    console.log(tokenURI);
  } catch (error) {
    // Handle error
    console.error(error);
  }
}
```