

NFT Minter DAPP



**A complete Web3 project using Hardhat,
Solidity, Ethers.js, OpenZeppelin and React**

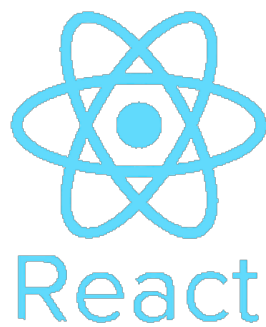


Table of contents

Our NFT DAPP - What are we actually building?.....	5
The Smart Contract Project:	6
The Frontend Project:	7
Tools and Technologies used for the Project	9
Tools for your development environment:	10
Reference documents:	10
Blockchain related services:	11
Other Ethereum related tools and services (optional):	11
And, here are some additional resources I provide on blockchain development:	11
Setting up your development environment	12
Installing Node.js:	12
Installing VSCode	13
Installing Ganache:	14
Installing Git - optional:	15
Setting up Metamask	16
Configuration of Metamask:	17
Creating a new account in Metamask:	18
Adding local test networks to Metamask:	18
Exporting your Private Key from Metamask:	19
Importing accounts into Metamask:	21
Creating your smart contract project using Hardhat	23
Get the project code from GitHub:	25
The Hardhat config file:	26
Importing packages - dotenv:	26
Configuring the Solidity compiler:	27
Adding additional networks:	27
Adding an Etherscan API key:	27
The .env file, keys, secrets and addresses:	27
1: REACT_APP_PRIVATE_KEY and REACT_APP_PRIVATE_KEY2 :	28
2: REACT_APP_CONTRACT_ADDRESS and REACT_APP_CONTRACT_ADDRESS_LOCAL:	29
3: REACT_APP_ALCHEMY_API_URL_SEPOLIA:	29
4: REACT_APP_PINATA_KEY and REACT_APP_PINATA_SECRET:	31

5: REACT_APP_ETHERSCAN_API_KEY:	33
Writing our NFT smart contract	34
A very high-level overview of smart contracts and Solidity:	34
Types:	34
Special variables:	35
Functions:	35
Special Functions:	36
Modifiers:	36
Events:	37
Error Handling:	37
Inheritance:	38
NFT's and ERC721 Tokens:	42
What are NFT's?	42
ERC721 Tokens:	42
Creating our NFT smart contract - MyNFT.sol	44
Compiling and deploying our smart contract	47
Deploying the smart contract:	47
Deploying the contract to a local Hardhat network:	48
Deploying to Ganache:	48
Deploying to the Sepolia test network:	49
Verifying your smart contract on Etherscan:	50
Using Ethers.js to communicate with our smart contract	53
Ethers.js:	53
Provider:	53
Signer:	54
Contract:	54
Utilities:	55
The NFT minting script:	57
Executing the script on a local node:	60
Executing the script on Sepolia:	60
Testing our smart contract	61
Writing the tests for our MyNFT contract:	62
Creating several test groups:	63
Deploying our contract before running each test:	63
Asserting basic properties after contract deployment:	64

Minting and transferring tokens:	64
Asserting token balances:	65
Asserting Events:.....	66
Asserting reverted functions:	67
Running our tests:.....	68
Running your tests on a standalone Hardhat node:	68
Running your tests on an in-memory Hardhat node:	68
Creating the DAPP with React	69
The main files of our React project:	69
Running our web application:	70
Testing the app on a local Hardhat node:	72
Testing the app on Sepolia:	72
Importing your NFT into Metamask:	73
Bootstrapping a React project:	74
The blockchain related code of our React application:	75
blockchainInteraction.js	75
pinata.js	78
NFTDAPP.js	78
Adding your project to Git and GitHub	81
Final thoughts & tips	84


Our NFT DAPP - What are we actually building?


The NFT DAPP (Non Fungible Token Decentralized Application) we are going to build is a simple yet powerful Web3 project that contains all important building blocks required to create sophisticated blockchain-based applications.


For this little project we will be using all the top tools, techniques and frameworks that are currently used by pretty much all of the top EVM-based (Ethereum Virtual Machine) projects.

NFT DAPP


[Connect Wallet](#)

 NFT Image URL:

 NFT Name:

 NFT Description:

[Mint NFT](#)

 [Connect to Metamask using the top right button.](#)

Our DAPP integrates seamlessly with Metamask (in order to sign transaction) and it allows us to mint NFT's with specific user-defined parameters (image, name, description). The smart contract used by the DAPP is written in Solidity and uses various OpenZeppelin smart contracts and libraries.

After deploying our smart contract to the Sepolia testnet, we will mint a couple of NFT's, display them to our Metamask wallet and we will also check out the associated images and attributes on Opensea (the biggest NFT marketplace).

Ok, this was a very high-level introduction of our project. Now, let's spend a few minutes to take a closer look and get a better understanding of what we are going to learn and what tools and technologies we are going to use.

Our project consists of two parts:

The Smart Contract Project:

The smart contract which will be written in Solidity and tested, compiled, deployed... using the Hardhat development environment. We will also be using the ethers.js library to interact with our smart contract from JavaScript.



`ethers.js`

When writing smart contracts it is very important to write as little code as possible and to re-use as much battle-tested code as possible in order to avoid those nasty little bugs that could cost your project literally millions in lost funds. The top library for that purpose is OpenZeppelin and of course, we will also use it for our project. Of those tens of thousands Web3 projects out there, there probably is not a single one that does not use OpenZeppelin!



OpenZeppelin

During development, we will be deploying and testing our smart contract on a lightweight local blockchain (actually, it is rather an in-memory blockchain simulation). This allows us to quickly test our code without having to wait those 10+ seconds it takes to integrate our transactions (required whenever we want to modify anything on the blockchain). For that purpose we will be using the Hardhat Network (which comes integrated with the Hardhat development environment) and we also take a quick look at Ganache, which comes with a graphical user interface.



And yes, a smart contract needs rigorous testing - more than any other software project! Luckily, Hardhat comes with a solid test framework integrated that uses Mocha and Chai (probably the most used test frameworks in the JavaScript world). However, in order to properly test our smart contracts, we have a few very specific requirements. But, don't worry, Hardhat (once again) has our back covered.

Afterwards, of course we will also deploy and test our smart contract on a "real" blockchain - the Sepolia test network. We check our transactions on Etherscan and we will also learn how to get our smart contract verified by Etherscan.

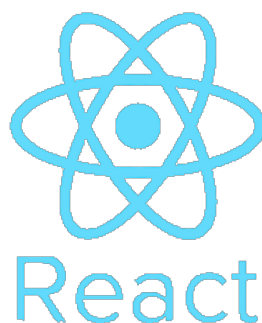


Verifying a smart contract means that its Solidity code will get published to Etherscan so that everyone can easily verify what is going on in your smart contract. That is very important, because a project that doesn't make its code available on Etherscan has zero credibility and probably no one will want to use a DAPP that depends on that smart contract.

Ok, there is so much more to talk about Solidity, Hardhat, blockchain... but please be patient, we'll get there soon. For now, we only take a very high-level view on what we are going to build

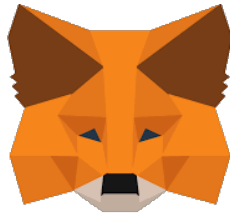
The Frontend Project:

Yes, of course, a DAPP needs a user interface - we should not expect our not so tech savvy users to get excited about using little black windowed command line tools. No, we need a proper, responsive, good looking user interface. And for that purpose we will be using React. Currently, about 80% of all Web3 frontends are built using React.



Still, by far the most used wallet application is Metamask and we will also integrate our DAPP with Metamask. That way, our DAPP user can easily sign transaction with an easy to use tool he is already familiar with.

And, sure, our DAPP will also need to communicate with our smart contract (reading data from the blockchain, minting NFT's...) and, once again, we will be using the powers of ethers.js to make our life as simple as possible.



To make our application truly decentralized, we will deploy our assets (images for our NFT's and JSON metadata files) to IPFS (Inter Planetary File System). To keep our assets readily available we use a service called Pinata in order to pin our assets to an IPFS node.



Don't worry about the jargon and the how-to for now, we'll take a closer look at all this stuff in the coming chapters. And, as difficult as all of this may sound right now, you will see, it is really easy.

Tools and Technologies used for the Project

This is not a full-blown Solidity, blockchain, Web3 development... course that explains everything in detail. There is already a lot of material out there for that purpose and such a course/book would fill easily 500+ pages.

The official Solidity documentation is already 360 pages and that's just one of many building blocks required to build a complete DAPP. But, don't worry, you don't need to know every little detail about all the tools and technologies we are going to use.

When I got started with blockchain development, I was really confused with the wealth of different tools, libraries and technologies out there. And, honestly, I didn't know how and where to get started.

All I wanted was a simple yet complete project that touches on all the different aspects of blockchain / Web3 development. A collection of techniques and recipes that allow me to get up to speed quickly while having fun working on a real project.

That's what I'm trying to achieve with this course and I hope it will help you to get started and up to speed quickly in the world of Web3 development for EVM compatible blockchains. And, of course, having a lot of fun on your way.

Here are some of the things you will learn in this course:

- How to get a proper, professional development environment setup
- How to use fast in-memory blockchains during development
- How to work with one of the Ethereum testnets (like Sepolia)
- How to manage environment variables and deployments for different blockchains
- Which ones to use from the hundreds of features and methods provided by ethers.js
- How to setup your tests and use some of the amazing features provided by Hardhat
- How to get your smart contract verified on etherscan
- How to re-use code from libraries like OpenZeppelin
- How to integrate Metamask with my DAPP
- How to call smart contract functions and issue transactions for an in-memory blockchain as well as for a proper blockchain
- How to deploy digital assets to IPFS using Pinata

As I already mentioned, we won't be taking a deep dive into the various technologies (which should not be your first step anyway). But, below you can find a list of useful resources you should consult whenever you need specific details on the various technologies we are using for this project:

Tools for your development environment:

- **Node.js & NPM** (JavaScript runtime and node package manager): <https://nodejs.org/en/>
- **VS Code** (source code editor): <https://code.visualstudio.com/>
- **Metamask** (wallet): <https://metamask.io/download/>
- **Git** (local source code management - optional): <https://git-scm.com/downloads>
- **Github** (source code management / collaboration - optional): <https://github.com/>
- **Remix** (online smart contract development environment - we are not using Remix for our project, but it's a fantastic tool to quickly prototype and test your smart contracts): <https://remix.ethereum.org/>

Reference documents:

- **Solidity** documentation (smart contract development language reference): <https://docs.soliditylang.org/en/v0.8.17/>
- **Hardhat** documentation (local smart contract development environment for testing, compiling, deploying... smart contracts): <https://hardhat.org/docs>
- **Ethers.js** documentation (interaction with your smart contract from JavaScript): <https://docs.ethers.org/v5/>
- **OpenZeppelin** (re-usable smart contracts and libraries): <https://docs.openzeppelin.com/contracts/4.x/>
- **OpenZeppelin Github**: <https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts>

Blockchain related services:

- **Alchemy** (third party node provider): <https://www.alchemy.com/>
- **Pinata** (pinning digital assets to an IPFS node): <https://www.pinata.cloud/>
- **Pinata API** (PinJsonToIPFS function):
<https://api.pinata.cloud/pinning/pinJSONToIPFS>
- **Sepolia Faucets** (get free test Ether - some of them may not be working):
<https://sepolia-faucet.pk910.de/> && <https://sepoliafaucet.com/>

Other Ethereum related tools and services (optional):

- Etherscan Blockchain Explorer: <https://etherscan.io/>
- Ethereum Gas Tracker: <https://cointool.app/gasPrice/eth>
- Ethereum Network Status: <https://ethstats.net/>
- Chainlist (settings for EVM compatible networks): <https://chainlist.org/>
- Keccak-256 hash: https://emn178.github.io/online-tools/keccak_256.html
- EVM Opcodes: <https://ethereum.org/en/developers/docs/evm/opcodes/>
- String to Byte Converter: <https://onlinestringtools.com/convert-string-to-bytes>

And, here are some additional resources I provide on blockchain development:

- Articles on blockchain development: <https://blockchaindevtips.com/>
- Videos on smart contract development and related subjects:
<https://www.youtube.com/@BlockchainDevTips>
- Stay up to date with the latest developments in the blockchain space:
<https://twitter.com/bchaindevtips>

Setting up your development environment

Ok, let's get started working on our project. But before diving into smart contract and Web3 coding, we first have to setup your development environment. Fortunately, that's quite easy and it should not take more than 10 minutes.

First, we will install Node.js and NPM. Node.js is a JavaScript runtime engine that allows us to execute JavaScript code outside a web browser. NPM is a Node Package Manager that comes bundled with Node.js.

NPMJS.com is the world's largest software registry where you can find JavaScript code for pretty much any task you can imagine. You need some code that accomplishes a specific task - someone has probably already coded and uploaded a software package to npmjs.com

So, whenever you are looking for a specific software library, head over to <https://www.npmjs.com/> and see what you can find.

We will also include various npm packages into our project and therefore we need NPM.

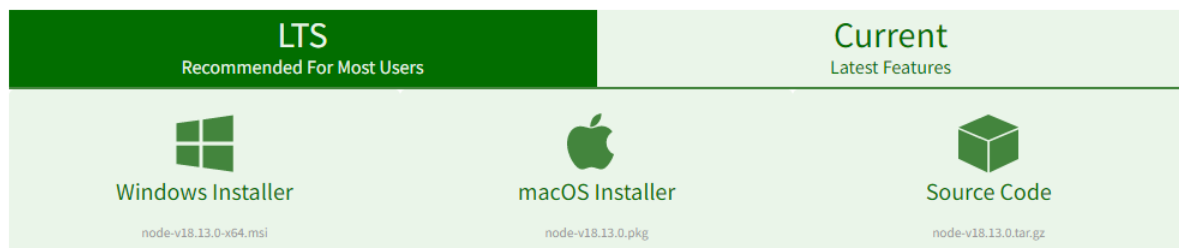
Installing Node.js:

Head over to <https://nodejs.org/en/download/>, download the latest LTS version and install it on your machine. The installation takes only a couple of minutes and you can accept all default parameters.

Downloads

Latest LTS Version: **18.13.0** (includes npm 8.19.3)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

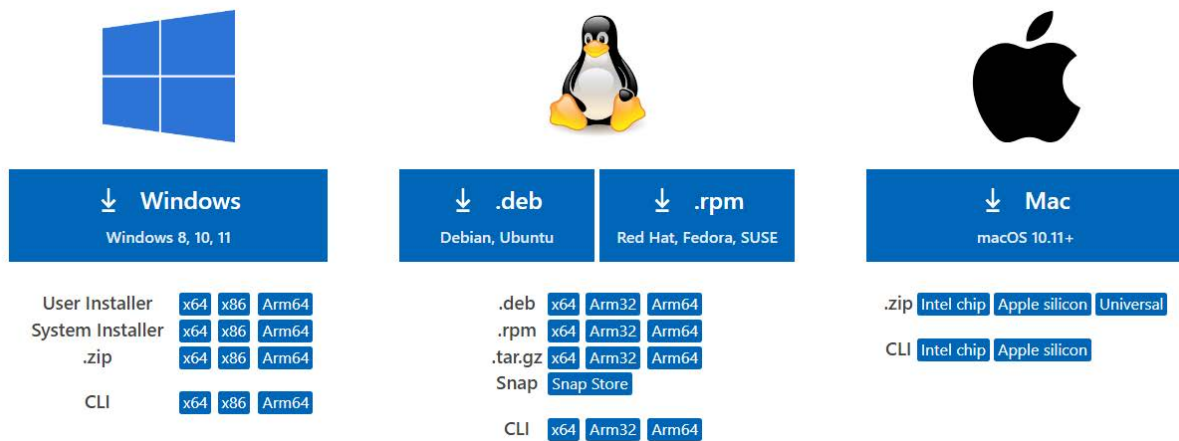


When you install Node.js, the latest version of NPM is installed automatically - so, there is nothing else you need to do.

Installing VSCode

You also need an editor. There are numerous great products available. The one I'm using and recommending is Visual Studio Code (VSCode).

The installation process is quick and easy. Go to <https://code.visualstudio.com/download> and download the correct version for your environment (Windows, Linux or Mac)

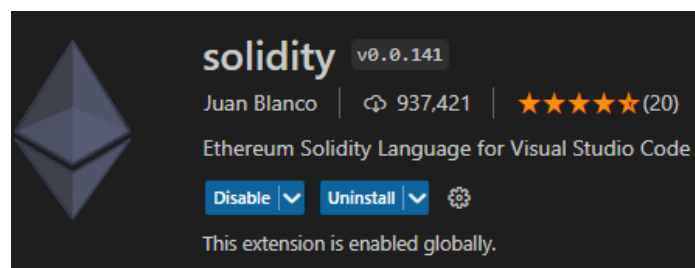


The screenshot shows the VS Code download page with three main sections: Windows, Linux, and Mac. Each section has a download button and a list of available installers for different architectures.

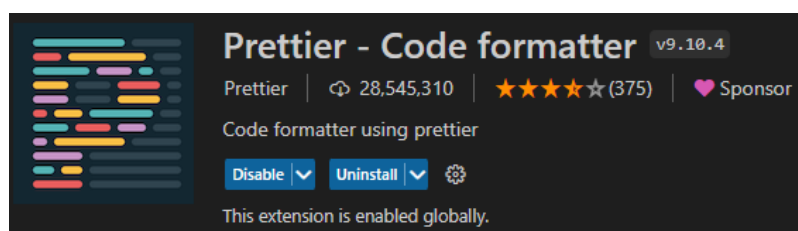
Platform	OS/Version	Installer Type	Architectures
Windows	Windows 8, 10, 11	User Installer	x64, x86, Arm64
		System Installer	x64, x86, Arm64
		.zip	x64, x86, Arm64
		CLI	x64, x86, Arm64
Linux	Debian, Ubuntu	.deb	x64, Arm32, Arm64
		.rpm	x64, Arm32, Arm64
	Red Hat, Fedora, SUSE	.tar.gz	x64, Arm32, Arm64
		Snap	Snap Store
	CLI		x64, Arm32, Arm64
Mac	macOS 10.11+	.zip	Intel chip, Apple silicon, Universal
		CLI	Intel chip, Apple silicon

If you decide to use VSCode, I recommend you also install a few plugins once you're your editor is installed. Open VSCode and you will see several icons on the left side of the editor. Click on the "Extensions" icon (this should be the sixth icon from the top) and search for the following plugins and install each of them:

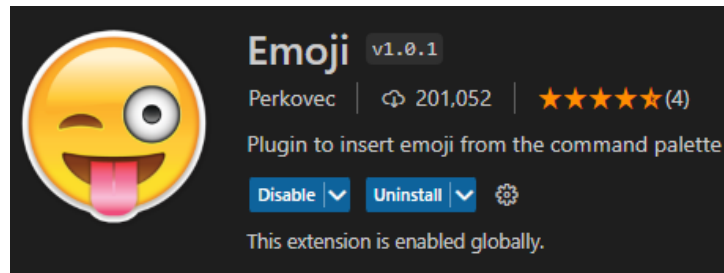
- **Solidity by Juan Blanco:** This is a very useful plugin that provides Solidity support for VSCode - syntax highlighting, code snippets, code completion, linting...



- **Prettier:** This is a powerful code formatter that makes you code look, well - prettier.



- **Emoji:** An easy way to add Emoji's to your frontend - this is optional and not really important.



Actually, there are hundreds of interesting and useful plugins and if you dig around a bit you will probably install many other extensions.

Installing Ganache:

Ganache is an in-memory development blockchain that comes with a graphical UI. This is optional and not really required because we will be using the Hardhat Network instead.

However, I will show you later on how to use Ganache as a local development network and how to integrate it with Metamask to test your DAPP locally before deploying your smart contract to the Sepolia testnet.

You can download Ganache from: <https://trufflesuite.com/ganache/>



After installing Ganache, open it and click the "Quickstart Ethereum" button - this provides you with a preconfigured local blockchain on port 8545 (<http://127.0.0.1:8545>) with 1337 as network Id and 10 test accounts that can be imported into Metamask for local testing. We'll learn more about the various parameters and settings in later chapters.



Installing Git - optional:

Git is a source code version control system that is used by pretty much any developer. You don't need it for this project, but if you never used Git or Github, I highly recommend you check it out and start using it.

Github is a web-based source control and collaborative tool. You can find most big blockchain / Web3 projects on Github (for example: Uniswap - the biggest decentralized exchange)

Here is the download link for Git: <https://git-scm.com/downloads>

Github is entirely web-based and you need to create an account if you want to host your source code on Github: <https://github.com/>

The code for this project is also on GitHub, but you don't need to create an account to access the code - you simply download or clone the project source code to your PC.

Later on, I'll give you a short introduction on the most important Git/Github commands, so you can start hosting your own projects on Github as well.

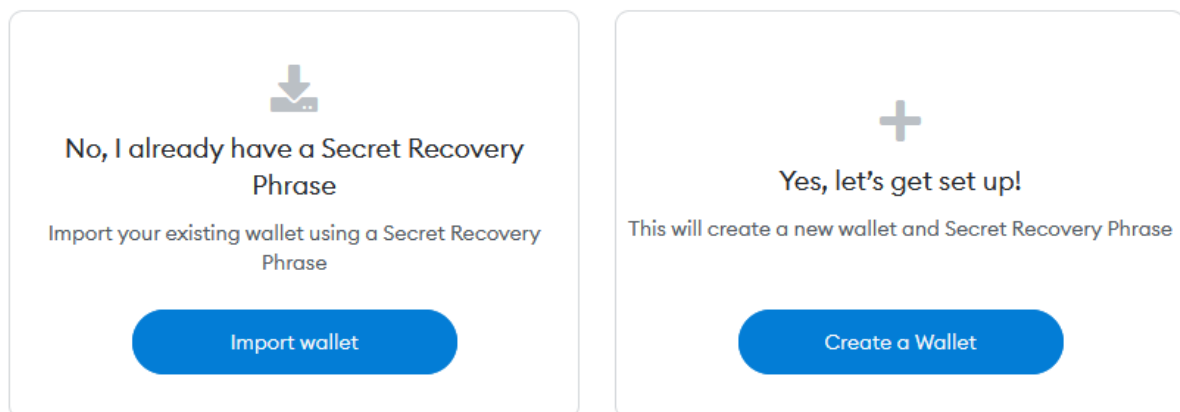
Setting up Metamask

Ok, one final step before we get started on our Web3 project - we have to install and configure Metamask. Metamask is the leading wallet application for Web3 DAPP's and every serious Web3 app that uses an EVM compatible blockchain absolutely needs to integrate Metamask.

Metamask is a browser plugin (it is also available for iOS and Android) and it is supported by all major web browsers.

You can download Metamask from: <https://metamask.io/download/>

The installation is quick and easy. Click "Create a Wallet" to setup a new wallet and make sure to save your Recovery Phrase somewhere safe.



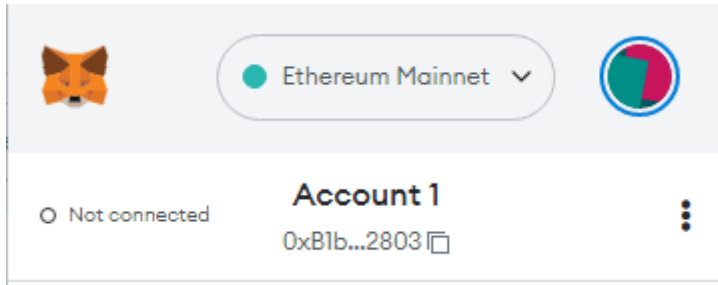
I don't keep real Ether (or other crypto currencies) on Metamask, I use those wallets only for testing and development. That's why I'm not particularly careful about my recovery phrases - after all, there is no real value on it.

If, however you choose to keep real Ether on your Metamask wallet, you should not store your recovery phrase in digital form. For real Ether, I recommend you use only cold storage like Trezor, Ledger...

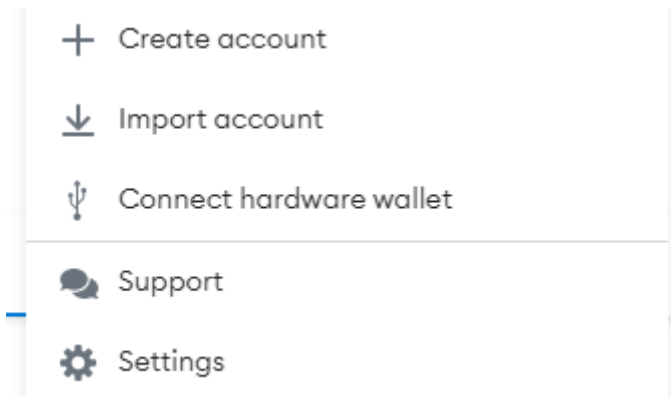
Once you are finished installing Metamask, you are automatically connected to the Ethereum Mainnet and you have one account.

Configuration of Metamask:

Click on the Metamask icon in your browser and on the popup window, *click the colored circle icon on the top right.*



On the menu that opens, *click "Settings"*



When you click the *"General" menu* you can change various general parameters, like: the primary currency to display, the language, the Metamask theme...

- Next, click the "Advanced" menu and do the following:
- Switch "Advanced gas controls" on
- Switch "Show test networks" on
- Switch "Customize transaction nonce" on
- You can also increase the value of the "Auto-lock timer"

Advanced gas controls
Select this to show gas price and limit controls directly on the send and confirm screens.

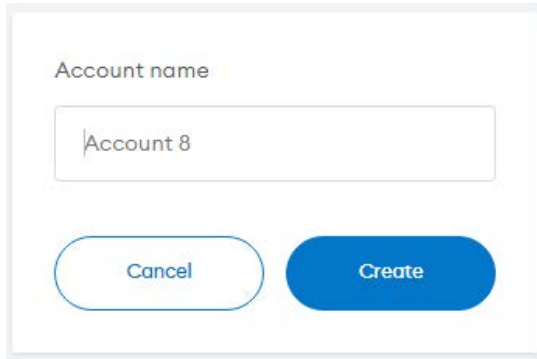


Show test networks
Select this to show test networks in network list



Creating a new account in Metamask:

Click on the Metamask icon in your browser, on the popup window, *click the colored circle icon on the top right* and on the menu that opens, *click "Create account"*



A screenshot of the Metamask account creation form. It features a text input field labeled "Account name" containing the text "Account 8". Below the input field are two buttons: a light blue "Cancel" button and a dark blue "Create" button.

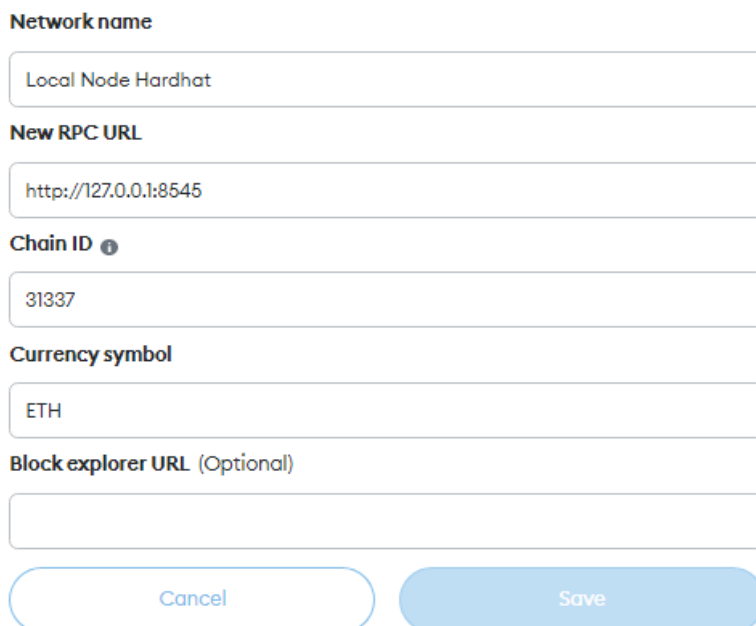
Provide a name for the account and click "Create"

Adding local test networks to Metamask:

Once again, click on the Metamask icon in your browser and on the popup window, *click the colored circle icon on the top right*. On the menu that opens, *click "Networks"*

Now, click "Add Network" and then, "Add a network manually"

First, we will be adding our local Hardhat network. Add the following parameters and click the save button:



A screenshot of the Metamask "Add Network" form. It contains several input fields: "Network name" with "Local Node Hardhat", "New RPC URL" with "http://127.0.0.1:8545", "Chain ID" with "31337", and "Currency symbol" with "ETH". There is also an empty "Block explorer URL (Optional)" field. At the bottom are "Cancel" and "Save" buttons.

Optionally, you can also add the Ganache network to Metamask. For Ganache, add the following parameters and click the save button:

Network name

Local Node Ganache

New RPC URL

http://localhost:8545

Chain ID ⓘ

1337

Currency symbol

ETH

Block explorer URL (Optional)

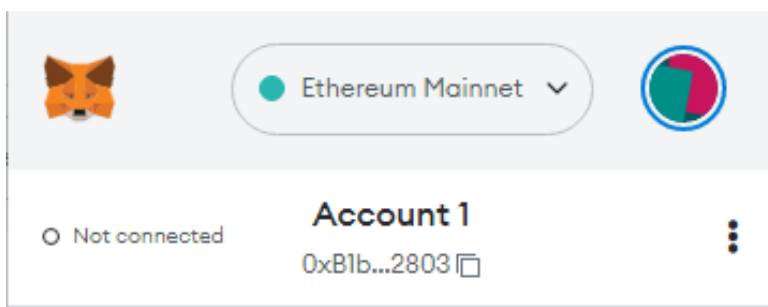
If you decide to change any of the parameters of your local test networks - for example, the port number (which is 8545 by default) or the chain Id, you will also need to update the network settings in Metamask accordingly.

Exporting your Private Key from Metamask:

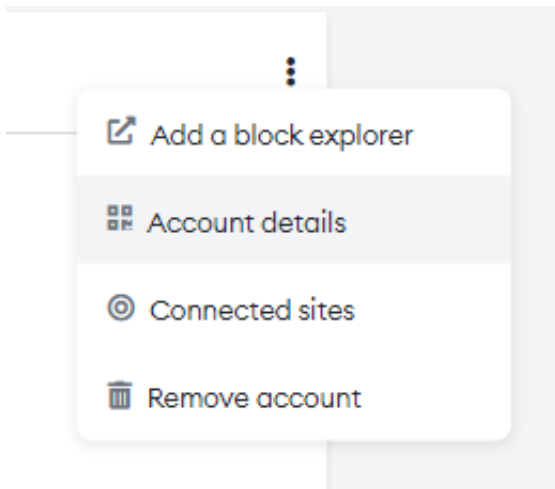
Later on, we will also need the private key of one of your test accounts for our project.

Once again, DON'T use any private keys that provide access to real money for that purpose!

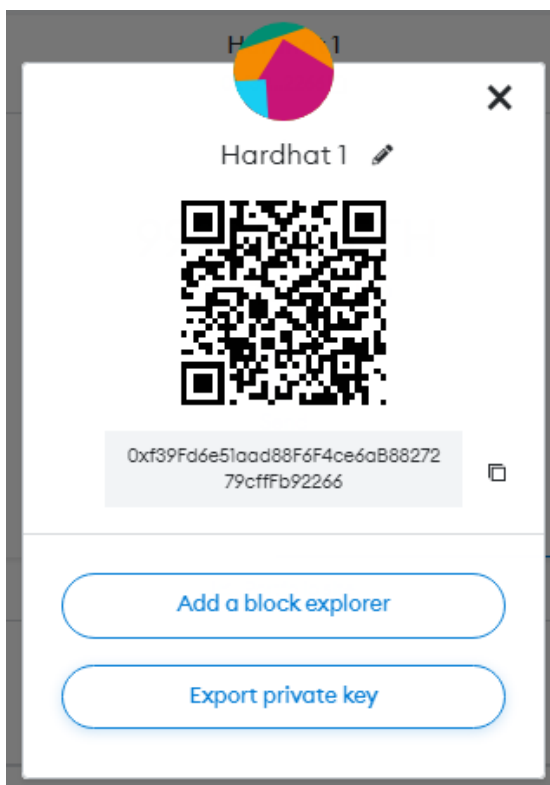
Click on the Metamask icon in your browser and on the popup window and on the popup window, *click the three vertical dots next to your account.*



Click "Account details"



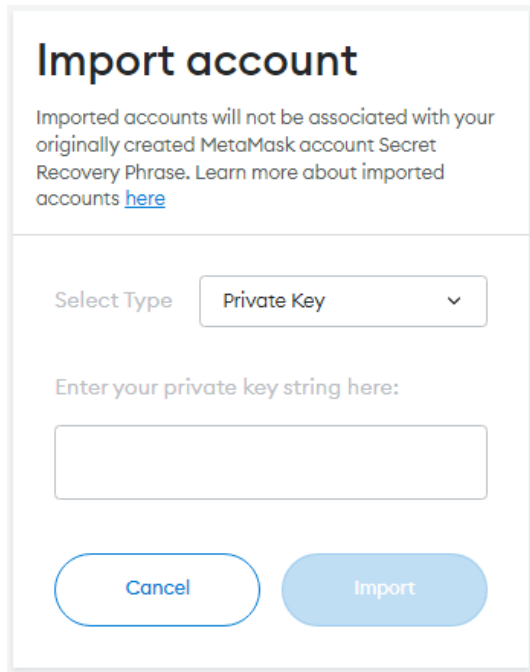
Provide your Metamask password *click "Export private key"* and store the private key for later use.



Importing accounts into Metamask:

We also need to import one of our Hardhat accounts into Metamask in order to test our DAPP locally before deploying our smart contract to the Sepolia test network.

Click on the Metamask icon in your browser, on the popup window, *click the colored circle icon on the top right* and on the menu that opens, *click "Import account"*



Make sure, "Private Key" is selected in the dropdown box, copy/paste one of your Hardhat private keys into the textbox below and click the "Import" button

We have not installed Hardhat yet, but when you launch the Hardhat network in a command window, 20 account addresses with their corresponding private keys are displayed. Those accounts and private keys do not change.

Here are the first 2 accounts with their private keys - simply copy/paste the first private key into Metamask as explained above:

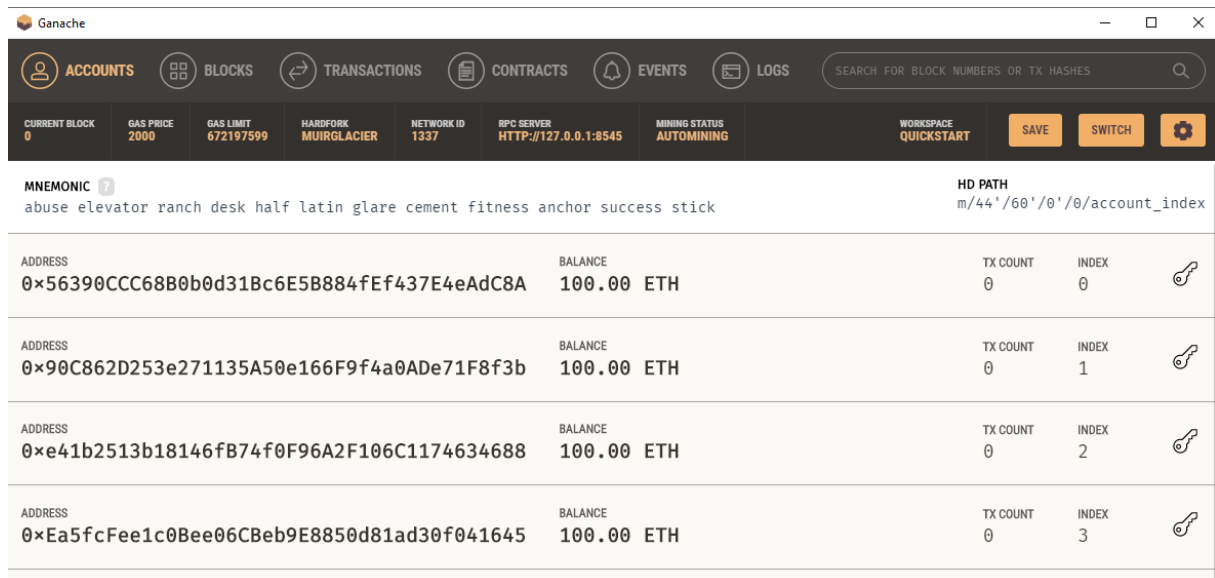
Account #0: 0xf39Fd6e51aad88F6F4ce6aB8827279cFfB92266 (10000 ETH)

Private Key: 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80

Account #1: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8 (10000 ETH)

Private Key: 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d

Optionally, if you prefer to use Ganache, you also have to import one of the accounts from Ganache into Metamask.



The screenshot shows the Ganache application interface. At the top, there is a navigation bar with icons for ACCOUNTS, BLOCKS, TRANSACTIONS, CONTRACTS, EVENTS, and LOGS. Below this is a status bar with various network parameters like CURRENT BLOCK, GAS PRICE, GAS LIMIT, HARDFORK, NETWORK ID, RPC SERVER, MINING STATUS, and WORKSPACE QUICKSTART. The main area displays account information, including a mnemonic phrase and an HD path. Below this, there is a table listing four accounts with their addresses, balances, transaction counts, and indices.

ADDRESS	BALANCE	TX COUNT	INDEX	
0x56390CCC68B0b0d31Bc6E5B884fEf437E4eAdC8A	100.00 ETH	0	0	
0x90C862D253e271135A50e166F9f4a0Ade71F8f3b	100.00 ETH	0	1	
0xe41b2513b18146fB74f0F96A2F106C1174634688	100.00 ETH	0	2	
0xEa5fcFee1c0Bee06CBeb9E8850d81ad30f041645	100.00 ETH	0	3	

Open Ganache, click the key icon on the right, next to account number 1 and copy that private key into Metamask.

Creating your smart contract project using Hardhat

Hardhat is a development environment to compile, deploy, test, and debug EVM compatible smart contracts. It comes built-in with the Hardhat Network (a local Ethereum network for development) and the Hardhat Runner - the CLI to interact with Hardhat.

For all the details, please check out the official Hardhat docs: <https://hardhat.org/docs>

To setup a basic Hardhat project, open a command window, navigate to the folder where you want to create your project and execute the following commands:

1: npm init

You need to provide various parameters for your project, like: package name, version, description... you can keep all default parameters - just hit Enter several times and confirm your final choice. This will create a package.json file in your project folder - this is the file that contains all the important metadata for your Node project.

2: npm install --save-dev hardhat

This command will install Hardhat into your project folder. The flag "--save-dev" indicates that Hardhat will be installed as a development dependency. If you create an NPM package and someone else installs your package, development dependencies will not be installed (they are only required during development, but not for actually using your package).

3: npx hardhat

That's the command we use to create a new Hardhat project. The default choice is "Create a JavaScript project" - that's exactly what we want. Hit Enter several times to confirm the proposed choices.

Now, a basic Hardhat project has been create for you with the following files and folders:

- **hardhat.config.js:** This is the Hardhat configuration file. We'll take a closer look at it in a few minutes.
- **contracts:** That's the folder where all your smart contracts will be located. There is already a test contract "Lock.sol" - we won't need any of the provided sample files.
- **scripts:** Here, all your scripts are located, like deployment or smart contract test scripts.
- **test:** Here are your Mocha test files. Rigorous testing is very important in smart contract development and we'll take a closer look at it in a later chapter.

4: *npm install --save-dev @nomicfoundation/hardhat-toolbox*

This installs various Hardhat packages that are required testing, deploying, interacting... with our smart contracts

5: *npm install @openzeppelin/contracts axios dotenv ethers*

Those are additional packages we'll need for our project. We'll discuss them a bit later when we start taking a look at the relevant code sections. Here we are not using the "--save-dev" flag - this means, those packages will be installed as core dependencies and they will show up in the "" section of our package.json file.

Here are once again all the commands to bootstrap a Hardhat project:

1: *npm init*

2: *npm install --save-dev hardhat*

3: *npx hardhat*

4: *npm install --save-dev @nomicfoundation/hardhat-toolbox*

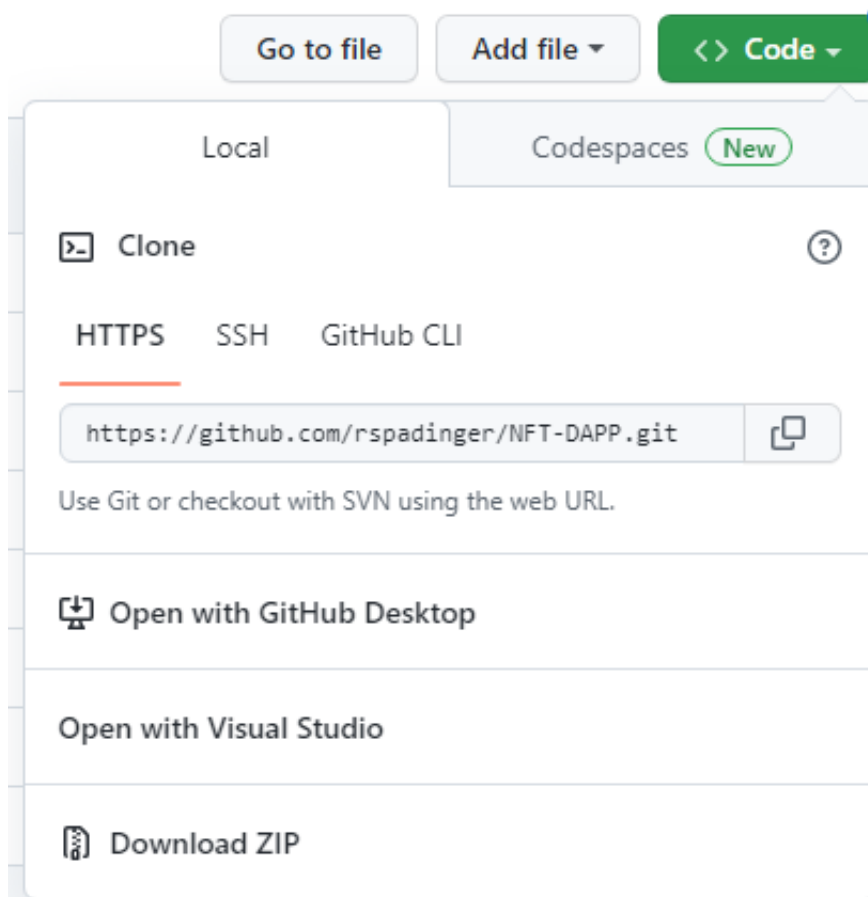
5: *npm install @openzeppelin/contracts ethers dotenv axios*

Command #5 is optional, but you will probably need at least the *@openzeppelin/contracts* and *ethers* packages

Get the project code from GitHub:

Now would probably be a good time to download the NFT DAPP code from GitHub so you can copy/paste various code sections into your own project or simple start playing around with the original project code.

Go to: <https://github.com/rspadinger/NFT-DAPP> and either clone the Repository or download the entire project as .zip file.



Next, you need to install all required packages: Open a command window, navigate to your project folder and execute the following command:

```
npm install
```

To run the web application, execute the following command:

```
npm run start
```

The Hardhat config file:

Hardhat provides you with a basic `hardhat.config.js` configuration file and we'll add a few additional settings. The original file only specifies the Solidity version, but we also want to specify additional optimization settings for the compiler, add a few networks and an API key for etherscan in order to automatically verify our smart contract.

Open the file: `hardhat.config.js` from the project you just downloaded from GitHub - it contains the following code:

```
1  require("@nomicfoundation/hardhat-toolbox")
2  require("dotenv").config()
3
4  const { REACT_APP_ALCHEMY_API_URL_SEPOLIA, REACT_APP_PRIVATE_KEY, REACT_APP_ETHERSCAN_API_KEY } =
5  |   process.env
6
7  module.exports = {
8  |   solidity: {
9  |     |   version: "0.8.17",
10 |     |   settings: {
11 |     |     |   optimizer: { enabled: true, runs: 200 },
12 |     |     |   },
13 |     |   },
14 |   defaultNetwork: "localhost",
15 |   networks: {
16 |     |   sepolia: {
17 |     |     |   url: REACT_APP_ALCHEMY_API_URL_SEPOLIA,
18 |     |     |   accounts: [`0x${REACT_APP_PRIVATE_KEY}`],
19 |     |     |   },
20 |     |   },
21 |   etherscan: {
22 |     |   apiKey: REACT_APP_ETHERSCAN_API_KEY,
23 |     |   },
24 |   }
```

Importing packages - dotenv:

The first 2 code lines import the required packages. `dotenv` is a package that allows you to access environment variables that are stored in the file: `.env` (located in the root of the original project). The `.env` file contains sensitive information like API keys that should not be stored with anyone else.

In the `.gitignore` file you should find an entry for: `.env` - this means, the `.env` file won't be considered by git and it won't be uploaded to GitHub. Only you should have access to that file.

Line #4 loads the values for all the required environment variables from the `.env` file using the `process.env` command

We'll take a closer look at the `.env` file in the next chapter.

Configuring the Solidity compiler:

On line #7-12 we specify the version of Solidity we want to use for our smart contract code and the optimizer settings for the compiler. The "runs" parameter allows to either optimize the generated bytecode for code size (deploy cost) or for code execution (cost after deployment). 200 is a standard value that optimizes the code for execution cost.

Adding additional networks:

Later on we want to deploy our smart contract and execute our scripts on 2 different networks: our local Hardhat network (or Ganache) and on the Sepolia test network.

On line #13-19 we specify our default network: "localhost" is either our Hardhat network or the network provided by Ganache and we also specify the Sepolia network. To access a remote network, we are using a third party node provider. There are various node providers available, the one we will be using is Alchemy (<https://www.alchemy.com/>)

Adding an Etherscan API key:

In order for our project to be credible, we need to verify and publish our smart contract code on Etherscan. Therefore, we need to create an account and an API key on Etherscan.

On line #20-22 we specify our Etherscan API key - not directly, but via an environment variable.

The .env file, keys, secrets and addresses:

I added the file: .ENV-EXAMPLE to the project that lists all the required environment variables. Rename that file to .env and provide your own keys.

Here are the required keys:

```
1 REACT_APP_ALCHEMY_API_URL_SEPOLIA = "https://eth-sepolia.g.alchemy.com/v2/..."
2 REACT_APP_PRIVATE_KEY = ""
3 REACT_APP_PRIVATE_KEY2 = ""
4
5 REACT_APP_CONTRACT_ADDRESS = 0x...
6 REACT_APP_CONTRACT_ADDRESS_LOCAL = 0x5FbDB2315678afecb367f032d93F642f64180aa3
7
8 REACT_APP_PINATA_KEY = ""
9 REACT_APP_PINATA_SECRET = ""
10
11 REACT_APP_ETHERSCAN_API_KEY = ""
```

Just in case you wonder, why I prefixed all variable names with "REACT_APP_" - later on, we'll be adding a React frontend project which will be created with create-react-app. And that project can read environment variables from .env without the need to install the dotenv package, however in order for this to work, all variables need to be prefixed with "REACT_APP_".

So, long story short, in order to use our variables from our JavaScript files as well as from our React app we add the above mentioned prefix.

Ok, but where do you get all those API keys from?

We need to create several account: On Alchemy, Pinata and Etherscan

But, lets start with the easy ones first:

1: REACT_APP_PRIVATE_KEY and REACT_APP_PRIVATE_KEY2 :

Those are 2 account private keys. In the previous chapter (Setting up Metamask) I provided you with the first 2 private keys from the Hardhat network. To list all accounts and private keys, open a command window, navigate to your project folder and type the following command:

npx hardhat node

This starts a local Hardhat network and lists all accounts and private keys. You'll get something like:

```
Started HTTP and WebSocket JSON-RPC server at http://127.0.0.1:8545/

Accounts
=====

WARNING: These accounts, and their private keys, are publicly known.
Any funds sent to them on Mainnet or any other live network WILL BE LOST.

Account #0: 0xf39Fd6e51aad88F6F4ce6aB8827279cFFfB92266 (10000 ETH)
Private Key: 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80

Account #1: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8 (10000 ETH)
Private Key: 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d

Account #2: 0x3C44CdDdB6a900fa2b585dd299e03d12FA4293BC (10000 ETH)
Private Key: 0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a

Account #3: 0x90f79bf6EB2c4f870365E785982E1f101E93b906 (10000 ETH)
Private Key: 0x7c852118294e51e653712a81e05800f419141751be58f605c371e15141b007a6
```

For local development, you can provide the first 2 private keys for the above mentioned variables.

If you prefer to use Ganache, you need to copy/paste the private keys from Ganache as mentioned in the "Setting up Metamask" chapter.

Later, when you deploy your contract to the Sepolia test network, you will need to replace those 2 private keys with the private keys from your Metamask accounts.

Take a look at the previous chapter: *Setting up Metamask => Exporting your Private Key from Metamask* for the instructions on how to export a private key from Metamask.

However, don't forget: This is only for testing purposes and you should never use a private key that provides access to real Ether. Anyway, the final DAPP, doesn't need any private keys, because the UI uses Metamask and it is the end users who will sign transactions using their own Metamask private keys.

2: REACT_APP_CONTRACT_ADDRESS and REACT_APP_CONTRACT_ADDRESS_LOCAL:

When you deploy your smart contract (using a deployment script) you receive the address of your contract. First, we'll deploy locally (to the Hardhat network) and you assign the resulting address to REACT_APP_CONTRACT_ADDRESS_LOCAL.

Something like (there are no quotes around the address):

```
REACT_APP_CONTRACT_ADDRESS_LOCAL = 0x5FbDB2315678afecb367f032d93F642f64180aa3
```

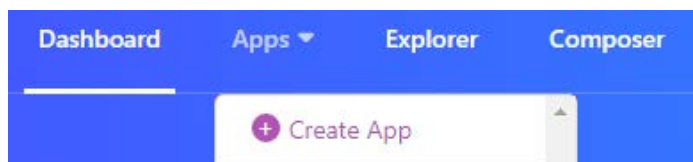
Later, we'll deploy our contract to Sepolia and we assign that address to REACT_APP_CONTRACT_ADDRESS

3: REACT_APP_ALCHEMY_API_URL_SEPOLIA:

Here we need to specify the API URL from our third party node provider (Alchemy). You first need to create an account (which is free), then you create an App...

Go to <https://auth.alchemy.com/signup> and create a free account

Log into your account and click "*Apps => Create App*"



Provide a name and a description, select "**Ethereum**" for the chain and "**Sepolia**" for the network and click "Create app"

Create App X

NAME ⓘ
NFT DAPP

DESCRIPTION ⓘ
My NFT Web3 App...

CHAIN NETWORK

Ethereum Sepolia

ADVANCED FEATURES

Reinforce Transactions \$999/mo intro offer

Opt in to get transactions on chain 7.9x faster with 100% success rate.

[More Info](#)

CREATE APP

Navigate to your App, by clicking Apps => MyAppName and click on "View Key" on the top right.



Copy the second value listed in the "HTTPS" field and paste it into your .env file, next to the REACT_APP_ALCHEMY_API_URL_SEPOLIA variable:

You should now have something similar in your .env file:

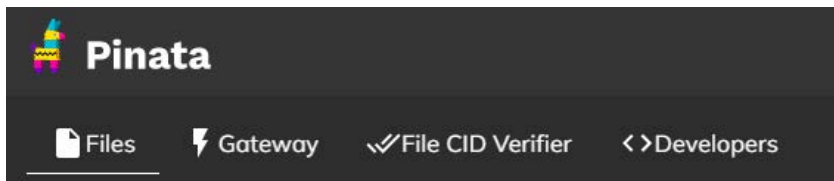
```
REACT_APP_ALCHEMY_API_URL_SEPOLIA = "https://eth-sepolia.g.alchemyapi.com/v2/YOURKEY"
```

4: REACT_APP_PINATA_KEY and REACT_APP_PINATA_SECRET:

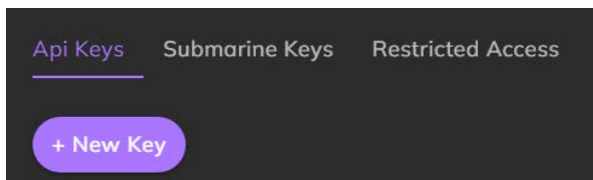
We use Pinata to pin our digital assets (images and metadata) to an IPFS node (more on this a bit later). But first, we need to create an account.

Go to <https://app.pinata.cloud/register> , create an account and log in.

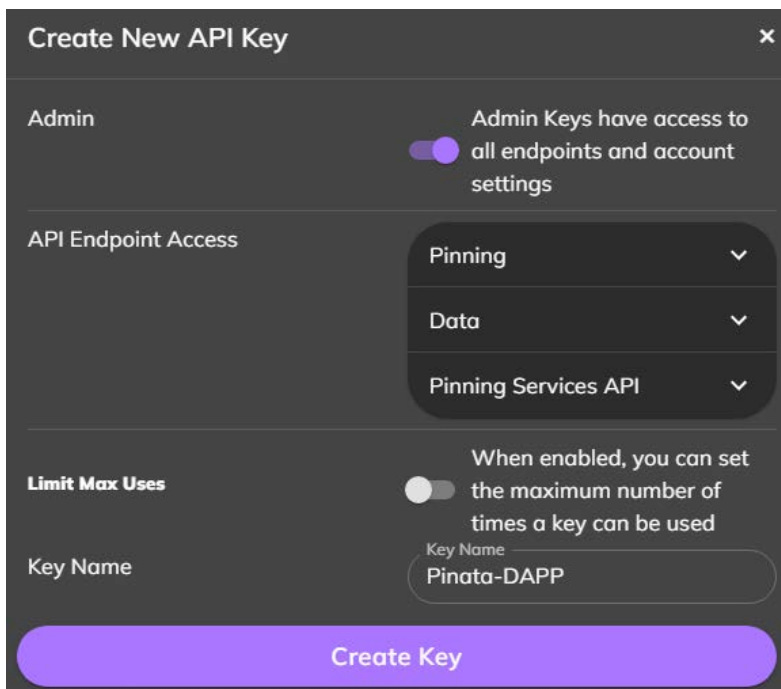
Click on "< > *Developers*":



And, then on "*Api Keys*" and "+ *New Key*":



Select the "*Admin*" option, provide a name for your key and click "*Create Key*":



On the window that opens, copy/paste the API Key and the API Secret into your .env file next to their corresponding variables.

Let's upload a couple of files to IPFS via Pinata. There are 2 digital assets in the project you downloaded from GitHub, both are located in the project root folder:

- `cat.jpg` - that's just an image file of a cat
- `nft-metadata.json` - a metadata file with various attributes and properties, like: name, description, image URL...

We'll need both files later on for our NFT project.

Login to your Pinata account, click the big "Upload +" button, select "File" and upload the `cat.jpg` file.



Make a copy of the CID (the value that starts with Qmc...)

Now, open the "*nft-metadata.json*" file, replace the CID part of the image URL with your own CID and upload the modified `.json` using Pinata.

Keep those 2 asset URL's around, you will need them soon for your NFT project. The URL's have the following format:

<https://gateway.pinata.cloud/ipfs/QmcSM8rxpnmknRu6HX9KWmqG4QJaBfijF3sZeuvSeDfNB7>

Of course, you will need to replace the CID (starting with Qmc...) with your own


5: REACT_APP_ETHERSCAN_API_KEY:

As I already mentioned, we need an Etherscan API key to verify and publish our smart contract code. That way, everyone can have a look at our smart contract code and be assured that nothing shady is going on.


Go to <https://etherscan.io/register> , create an account and log in.

On the left menu bar, click on "**API keys**":

OTHERS

 API Keys

 Verified Addresses

 Custom ABIs

Click the "+ **Add**" button on the top of the page, provide an App Name and click "**Create New API Key**":

Create API Key ×

App Name

Cancel Create New API Key

Copy/paste the API key next to the REACT_APP_ETHERSCAN_API_KEY variable in your .env file.

Ok, this was quite a bit of work, but finally we can get started writing our smart contract...

Writing our NFT smart contract

As I already mentioned, the goal of this course is not to teach you all the details about Solidity, ethers.js, Hardhat... and other tools and technologies we are using. So, we won't spend much time learning the intrinsics of Solidity. I will give you a very high level overview of Solidity and of course, I will explain you the code of our NFT smart contract.

A very high-level overview of smart contracts and Solidity:

A smart contract is a piece of code that is running on the blockchain. It can be viewed as a state machine. In order to change state, a transaction needs to be executed and mined.

Most EVM (Ethereum Virtual Machine) compatible smart contracts are written in Solidity which is a Turing-complete programming language somewhat similar to JavaScript. Turing completeness means that the language is capable to solve any computational problem.

After compiling a Solidity smart contract we get 2 important pieces of information: The smart contract byte code and the Application Binary Interface (ABI). The byte code is required for the deployment of the smart contract to the Ethereum blockchain - this bytecode is stored in the contract account.

The ABI is required to access specific contract functions from a client application. The ABI contains the specification of all public contract functions including the types of the function arguments and the types of the return values.

In Solidity, we have the following important building blocks:

Types:

There are 2 categories of types: Value types and reference types. Unlike a value type, a reference type does not store its value directly. Instead, it stores the address where the value is stored. In other words, a reference type contains a pointer to another memory location that holds the data.

We have the following value types: Booleans, integers (int and uint), addresses, fixed-size byte arrays and enums

And, the following reference types: Arrays (fixed-size and dynamic), structs and mappings, string and bytes (both are considered as special arrays)

When we use reference types, we also have to specify the data location where the type is stored (except for state variables - they are storage by default).

There are 3 different data locations: storage (permanent), memory and calldata (both are temporary and calldata is only for function arguments)

Special variables:

We also have a few special variables/objects that are always available in the global namespace.

msg: This object provides information about the current call. There are 3 member functions: msg.sender (provides the address of the caller), msg.value (provides the amount that was transferred) and msg.data

block: This object provides information about the current block. The most important members are: block.timestamp (in seconds since unix epoch), block.hash and block.number

Functions:

There are basically 2 types of function. Those that modify the state of a smart contract and those that do not. State modifying functions require a transactions and a gas fee needs to be paid. The amount of the fee depends on the complexity of the function to execute.

Functions that do not modify the state do not require a transaction (a simple function call is sufficient) and no gas fees need to be paid. Those functions can be labeled with the "view" keyword if they only read from state, but don't modify it. If a function does not even read from state (for example, a helper function that performs some calculation) it can be labeled with the "pure" keyword.

Below are some examples for functions that modify state, read from state (view) and don't read from state at all (pure):

```
uint someValue = 5;

function funcWrite(uint val) public {
    someValue = val;
}

function funcView() public view returns(uint) {
    return someValue;
}

function funcPure(uint val) internal pure returns(uint) {
    return val * 2;
}
```

Special Functions:

In Solidity we also have a few functions and constructs that can be considered as special functions.

getter functions: The compiler automatically creates a getter function for all public state variables. So, for example if we declare a state variable:

```
uint public counter = 0;
```

We don't need to write a function like: `function readCounterValue() ...` to access the value of our variable. A getter function with the same name as the variable is created automatically.

constructor: The constructor allows us to initialize state variables of our smart contract and it is called only once during the deployment of the contract.

selfdestruct: This is not really a function, but a keyword that allows us to render our smart contract unusable. The history of all transactions, of course remains on the blockchain.

receive: The receive function is executed when the smart contract receives Ether. It cannot have any arguments, it cannot return anything and it must have external visibility and be labeled as payable:

```
receive() external payable {...
```

fallback: The fallback function is executed on a call to the contract if none of the other functions match the given function signature. If a receive function does not exist, but a payable fallback function exists, the fallback function will be called on a plain Ether transfer.

```
fallback() external payable { counter = 1; }
```

Modifiers:

Modifiers are used to add reusable functionality to functions in a declarative manner. They are typically used to check specific conditions before executing the function

Multiple modifiers can be applied to a function - separated by whitespace and executed in the order presented. The symbol `_` in the modifier is replaced with the function body.

Here is an example of a simple modifier (onlyBefore) and how to use it on a function (bid):

```

uint public biddingEnd; //set in constructor
error TooLate(uint time);

modifier onlyBefore(uint time) {
    if (block.timestamp >= time) revert TooLate(time);
    _;
}

function bid() external payable onlyBefore(biddingEnd)
{
    //do something...
}

```

Events:

We cannot return anything from functions that modify state to the outside world. In order to provide data from a state changing function outside the contract, we have to use events.

Events allow to log specific information - they are also used as cheap data storage (storing data on the blockchain is very expensive)

Client applications can subscribe and listen to events and react accordingly. Up to 3 parameters can be indexed and searched for later.

Below is an example on how to declare an event and how to emit that event in a function:

```

event Transfer(address indexed from, address indexed to, uint256 value);

function _transfer(address sender, address recipient, uint256 amount) internal
{
    // transfer tokens...
    emit Transfer(sender, recipient, amount);
}

```

Error Handling:

Transactions are atomic - they either fail or succeed as a whole. All errors and exceptions revert the state of the contract to its initial values (as it was, before calling the function that generated an exception).

In Solidity we have the following 3 statements that allow us to generate an exception and to revert the state to its initial values:

require(bool condition, [string memory message]) : typically used to validate input parameters, reverts if the condition is not met.

assert(bool condition) : used for internal errors, if such an error occurs, the code is probably flawed and needs to be fixed. Assert is also triggered, if: division or modulo by zero, out of bounds index is accessed...

revert(string memory reason) or revert MyCustomError() : aborts execution and reverts all state changes. Using custom errors is cheaper than providing information in a string - a detailed explanation of the error can be provided via NatSpec comments.

Below are a few examples on how to use error handling in Solidity:

```
/// the provided amount for the purchase is incorrect
/// the product costs 1 ETH
/// @param provided The amount provided by the user
error WrongAmount(uint provided);

function buySomethingFor1ETH() public payable {
    if (msg.value != 1 ether)
        revert("Incorrect amount.");

    // Better use a custom error
    if (msg.value != 1 ether)
        revert WrongAmount(msg.value);

    // Alternative way to do it:
    require(msg.value != 1 ether, "Incorrect amount.");

    // Perform the purchase...
}
```

Inheritance:

Solidity supports multiple inheritance and you can derive from one or more contracts using the "is" keyword, for example:

contract C is A, B { ...

The order of the contracts we inherit from is important. In the example above, contract B is the most derived contract and if we call a function in contract C that is defined in contract A and B, then the function defined in contract B will be called

A function in a lesser derived contract can be called by specifying the contract name, for example, in contract C we could call: *A.functionName()* in order to execute the function defined in contract A.

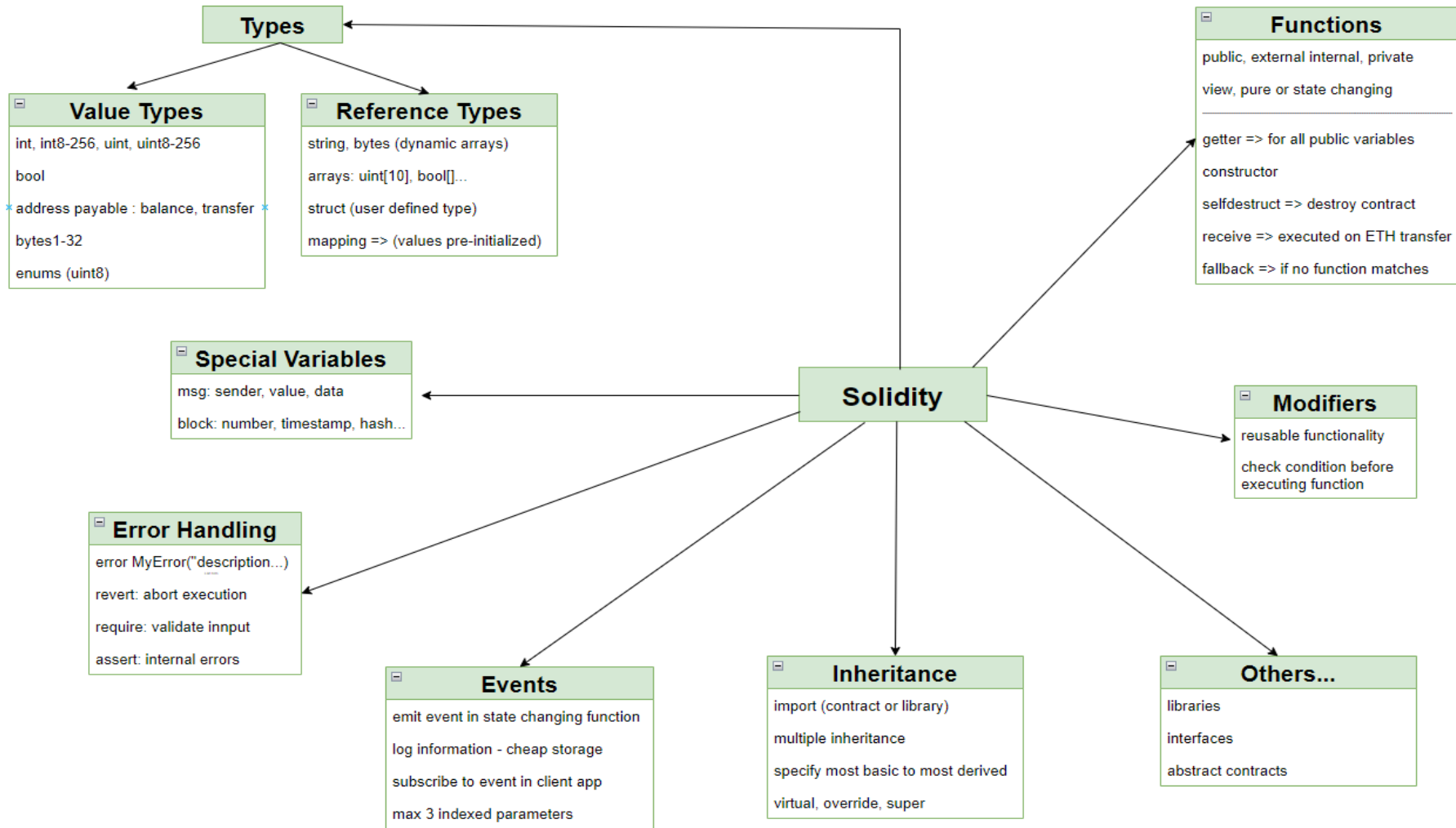
A derived contract can access all non-private members of the contracts it inherits from. If we want to allow a function to be overridden in an inheriting contract, we need to use the *virtual* keyword on the function and in the inheriting contract we need to specify the *override* keyword in order to change the implementation of the function.

Libraries, interfaces, abstract contracts...

Solidity provides some additional features, we will just take a very quick look at libraries, interfaces and abstract contracts:

- A **library** provides re-usable code that allows us to extend the functionality of our smart contract without having to re-invent the wheel. In our NFT smart contract we will be using a couple of libraries from OpenZeppelin.
- An **interface** provides only the declaration of a set of functions without implementing the actual function code. To define an interface, we use the "*interface*" keyword. In an interface, all functions need to be external and they are virtual by default. Also, an interface cannot have state variables, modifiers or a constructor.
- An **abstract contract** is often used as a base contract for other contracts that inherit from it. A contract needs to be marked as abstract (by using the "*abstract*" keyword) when one or more functions are not implemented, but even if all functions are implemented, a contract can still be defined as abstract. Abstract contracts cannot be instantiated directly.

The image below provides a quick overview of the essential building blocks of the Solidity language:



Well, of course, this was a very quick and superficial overview of Solidity and in order to learn more about it, I recommend you take the following steps:

- Either go to: <https://docs.soliditylang.org/en/v0.8.18/> or download the latest version of the official Solidity documentation from:
<https://docs.soliditylang.org/ /downloads/en/latest/pdf/>
- Take a look at the chapters: "Introduction to Smart Contracts" until the chapter "Contracts". This corresponds with ~ page 10 to page 140 on the pdf download. You can skip the chapter: "Installing the Solidity compiler" and also wait with the chapter: "Solidity by Example"
- To get started, don't read all chapters line by line. Just get a quick overview and a basic understanding of what Solidity has to offer. At this stage you don't need all the details.
- Once you are finished with the previous step, take a look at the examples provided in the chapter "Solidity by Example", and try to understand what each line of code is doing.
- Go to: <https://remix.ethereum.org/> and start playing around with those examples. Also, try to code those smart contracts yourself without looking at the sample code. There are some intermediate and even advanced techniques in those code examples and they provide a great learning experience.
- As you are gaining more experience with Solidity, re-read some of the chapters in the official Solidity docs and take a look at the finer details to gain a deeper understanding of the capabilities of Solidity.

NFT's and ERC721 Tokens:

What are NFT's?

NFT stands for a non-fungible token, which means it is unique and not interchangeable. NFTs are used to represent ownership of unique items - they can only have one official owner at a time. Each minted token has a unique identifier that is directly linked to one Ethereum address. An NFT is a digital (tokenized) asset that can represent all kinds of things, like: collectibles, game items, art, real estate...

The creator of an NFT can define the scarcity of the asset. He may decide to create an NFT where only one item is minted as a special rare collectible or where 1000 identical items are minted that may be used as admission tickets to an event. In this case, each NFT would still have a unique identifier - like a bar code on a traditional ticket - with only one owner.

ERC721 Tokens:

Our goal is to write a smart contract that allows us to mint NFT's with a specific name and symbol. And, we would like to be able to mint each NFT to a specific recipient address and to specify a URL that contains specific NFT metadata, like: name, description, image URL and other NFT specific attributes.

The contract needs to adhere to the ERC721 token standard. ERC721 is a standard for NFT's that requires us to implement a various functions and events we can use to verify the token balance for a specific address, the owner of a specific token Id, transfer and approve tokens for specific users...

The ERC721 standard requires us to implement the following functions:

- balanceOf(owner)
- ownerOf(tokenId)
- transferFrom(from, to, tokenId)
- safeTransferFrom(from, to, tokenId)
- approve(to, tokenId)
- setApprovalForAll(operator, _approved)
- getApproved(tokenId)
- isApprovedForAll(owner, operator)

And, we also need to implement the following events:

- Transfer(from, to, tokenId)
- Approval(owner, approved, tokenId)
- ApprovalForAll(owner, operator, approved)

Now, that seems like quite a lot of work... Fortunately we don't have to write all that code ourselves. We will re-use the code provided by OpenZeppelin and create a contract that inherits from the OpenZeppelin ERC721 contract.

OpenZeppelin provides numerous contracts and libraries that can be re-used by other for free by any EVM-compatible smart contract project. Those contracts have been used by thousands of different projects, they are secure and battle-tested and you should not write any smart contract code that is already available on OpenZeppelin.

The paradigm in smart contract development is to write as little code as possible and to re-use as much of already tested and proven code as possible. Whenever you write code, there is always a possibility that a minor bug sneaks in - even on a very well tested project. And, as there is often a lot of money at stake those little bugs could prove to be very costly.

Here are the ERC721 API reference docs from OpenZeppelin:

<https://docs.openzeppelin.com/contracts/4.x/api/token/erc721>

And here is the code of all the OpenZeppelin contracts on Github:


<https://github.com/OpenZeppelin/openzeppelin-contracts>

I recommend, you study the code of some of the OpenZeppelin contracts and libraries. That's a great way to learn how to write top quality smart contract code.

Creating our NFT smart contract - MyNFT.sol

Ok, it's finally time to write our smart contract. Below is the code of the entire contract and we will take a look at it line by line.

You can find the contract file: *MyNFT.sol* in the *contracts* folder of the project you downloaded earlier on from my Github repository (<https://github.com/rspadinger/NFT-DAPP>)

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.17;
3
4 import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
5 import "@openzeppelin/contracts/utils/Counters.sol";
6 import "@openzeppelin/contracts/access/Ownable.sol";
7
8 contract MyNFT is ERC721URIStorage, Ownable {
9     using Counters for Counters.Counter;
10    Counters.Counter private _tokenIds;
11
12    constructor() ERC721("MyNFT", "MNFT") {}  infinite gas 2428000 gas
13
14    function mintNFT(address recipient, string memory tokenURI) external onlyOwner {
15        _tokenIds.increment();
16        uint256 newItemId = _tokenIds.current();
17        _mint(recipient, newItemId);
18        _setTokenURI(newItemId, tokenURI);
19    }
20 }
```

- **Line #1:** This is a licence identifier. It is not absolutely required, but the compiler will generate a warning if you don't include it in your code.
- **Line #2:** Here, you define the compiler version to be used for your .sol file. This file would not compile with a compiler version earlier than 0.8.17 or with a compiler version starting from 0.9.0 or later.
- **Line #4-6:** Here, we import various contracts and libraries from OpenZeppelin. We are actually not using the standard ERC721 contract, but the ERC721URIStorage contract which is also called an extension contract (there are several different ones). This extension contract inherits from the base ERC721 contract and implements some additional features that allow us to specify a token metadata file for the NFT.

The *Counters library* allows us to manage counters that can be increased, decreased or reset. We use it to increment the unique Id of each NFT we mint.

The *Ownable contract* provides access control features and it allows us to restrict access to specific functions.

- **Line #8:** Here, we create our contract, called: "*MyNFT*" and to make our life a bit easier, we inherit all the features provided by the OpenZeppelin contracts: *ERC721URIStorage* and *Ownable*, which we imported in lines #4 & 6
- **Line #9 & 10:** Here we make use of the Counters library. To make use of a library in Solidity, we use the statement:

use SOMELIBRARY for SOMETYPE

Our library is called "*Counters*" and we want to use all the defined functions (like; increment...) on a type that is also defined in our Counters library and which can be accessed at: "*Counters.Counter*"

We also define a private variable called "*_tokenIds*" which is of the type: "*Counters.counter*". Now, those 2 lines allow us to use all the functions defined in the Counters library on our variable "*_ tokenIds*"

- **Line #12:** We have to define a constructor function in order to call the constructor of the *ERC721URIStorage* contract we are inheriting from (which in turn inherits from the *ERC721* base contract). We do this by calling "*ERC721(name, symbol)*" - with the two required arguments for the name and symbol of the token - after our empty constructor.
- **Line #14:** Here, we create the only function we need for our smart contract and that allows us to mint an NFT to a specific address and with a specific URL attached. This URL could point to an image file or to a JSON file that specifies various metadata for our NFT.

This function can only be called externally and only by the owner of the contract - that's why we are using the *onlyOwner* modifier from the *Ownable* contract we inherit from.

- **Line #15 & 16:** Each time we mint an NFT, we increment the *tokenId* counter and we grab the current Id.

- **Line #17:** Here, we call the internal `_mint` function that is defined in the base ERC721 contract. We need to provide the recipient address and the unique token Id.
- **Line #18:** Finally, we call the `_setTokenURI` function that we inherit from our ERC721URIStorage contract. This function attaches the provided tokenURI to the NFT with the Id of "*newItemId*".

Compiling and deploying our smart contract

To compile the smart contract, open a command window, navigate to your project folder and execute the following command:

```
npx hardhat compile
```

If everything is ok, you should get the following response: Compiled 1 Solidity file successfully

Deploying the smart contract:

To deploy the smart contract, we use the deployment script provided by Hardhat and modify it slightly. You can find the deploy.js script in the "scripts" folder of the project you download from Github.

```
1  async function main() {
2      const MyNFTFactory = await ethers.getContractFactory("MyNFT")
3      const myNFT = await MyNFTFactory.deploy()
4
5      await myNFT.deployed()
6
7      console.log("myNFT deployed to:", myNFT.address)
8  }
9
10 main()
11   .then(() => process.exit(0))
12   .catch((error) => {
13       console.error(error)
14       process.exit(1)
15   })
16
```

To deploy the contract, we create a contract factory, by using the *getContractFactory* method on the *ethers* object and we specify the name of our contract, which is "MyNFT"

How come, we have access to the ethers object?

Because, in our hardhat.config.js file we imported the Hardhat Toolbox, which also imports the ethers.js package (amongst various other packages). Remember, the statement we used was:

```
require("@nomicfoundation/hardhat-toolbox")
```

Afterwards, we simply call the "**deploy**" method on our contract factory, which returns an instance of the contract. In order to get the address of the deployed contract, we have to wait until the contract has finally be deployed to our target blockchain. We do so by calling the "**deployed**" method on our contract instance (*myNFT*) and by awaiting the response.

To log the contract address, we simply call the "**address**" property on our contract instance.

Deploying the contract to a local Hardhat network:

1: Open a command window, navigate to your project folder and execute the following command:

```
npx hardhat node
```

This starts a new Hardhat network - an in-memory simulation of a blockchain.

2: Open another command window, navigate to your project folder and execute:

```
npx hardhat run scripts/deploy.js
```

This allows us to execute the deploy.js script that is located in our scripts folder. You should get the following response:

```
myNFT deployed to: 0x5FbDB2315678afecb367f032d93F642f64180aa3
```

You should also see the contract address and various other statements in the other command window where your Hardhat node is running.

Deploying to Ganache:

If you prefer to use Ganache, you have to stop your Hardhat node and start Ganache. You can't have both running at the same time.

Then simply run the same command as above:

```
npx hardhat run scripts/deploy.js
```


Deploying to the Sepolia test network:

When we deploy a contract, we are modifying the state of the blockchain and therefore we have to pay a fee. The amount to pay depends on the size and complexity of the contract we want to deploy.

Sepolia is a test network and the fees need to be paid in Sepolia Ether, which can be obtained for free on various so-called faucets.

Here are two url's for Sepolia faucets:

- <https://sepolia-faucet.pk910.de/>
- <https://sepoliafaucet.com/>

Unfortunately, those faucets are not always working and you may be required to try another service. If none of the faucets mentioned above are working, just type "Sepolia faucet" into Google and try some other addresses.

Once you have some Sepolia Ether on your account, you are ready to deploy your contract to the [https://sepoliafaucet.com/ testnet...](https://sepoliafaucet.com/testnet...)

In our *hardhat.config.js* file we already configured the Sepolia network with an API URL from Alchemy, our third party node provider and a private key of one of our Metamask accounts.

However, this is not our default network and that's why we have to specify it when we are running our deployment script. We do so, simply by specifying the network flag followed by the name of the network: `--network sepolia`

So the command for deploying our contract to Sepolia is:

```
npx hardhat run scripts/deploy.js --network sepolia
```

Now, we are deploying to a real blockchain, where new blocks are added every 10+ seconds. This means, the deployment won't be instantaneous, you will have to wait at least 10 seconds.


After a short while the address of your deployed contract will be displayed in the command window and you can check out your contract on Etherscan.




Verifying your smart contract on Etherscan:

Go to: <https://sepolia.etherscan.io/> and copy/paste the address of your deployed contract into the textbox.

You should see the transaction hash for the contract creation. Click on it and you will see the details of your transaction:

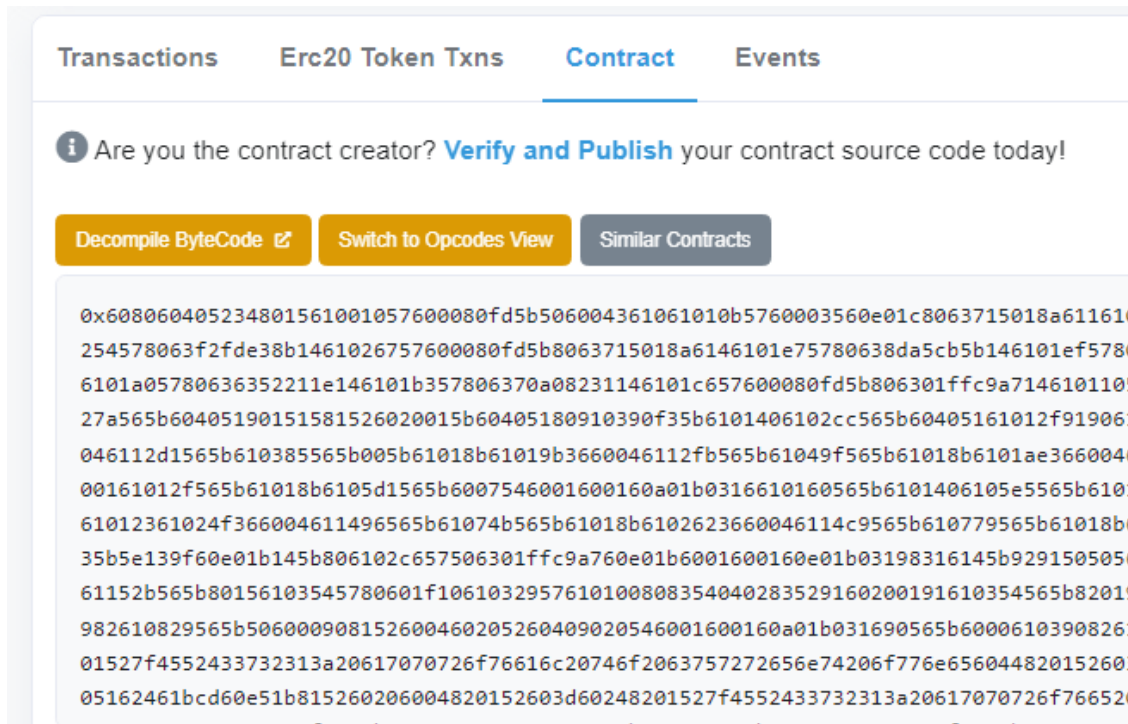
[This is a Sepolia Testnet transaction only]

Transaction Hash:	0xb5321b1e75b60237c6b5a4752136a0d7da66ab7339b6f73621bb094634fd9bd5 
Status:	Success
Block:	3155284 5 Block Confirmations
Timestamp:	1 min ago (Mar-24-2023 05:34:36 PM +UTC)

From:	0xB1b504848e1a5e90aEAA1D03A06ECEee55562803 
To:	[0x71f4708b58cc43d2324e24253bbfcee6b41ebf4d Created]  

Value:	0 ETH (\$0.00)
Transaction Fee:	0.001423061018499793 ETH (\$0.00)
Gas Price:	1.000000013 Gwei (0.000000001000000013 ETH)

When you click on the contract address (next to the "To" field) and then on the "Contract" tab, you can see the bytecode of the deployed contract. However, we can't see the actual Solidity code of our contract and that's a problem:



As I already mentioned, a project that does not publish its source code does not come across as very trustworthy and not many people will want to use a DAPP associated with such a smart contract.

Fortunately, there is an easy solution. We can use our Etherscan API key, we created earlier on and get our smart contract code published on Etherscan.

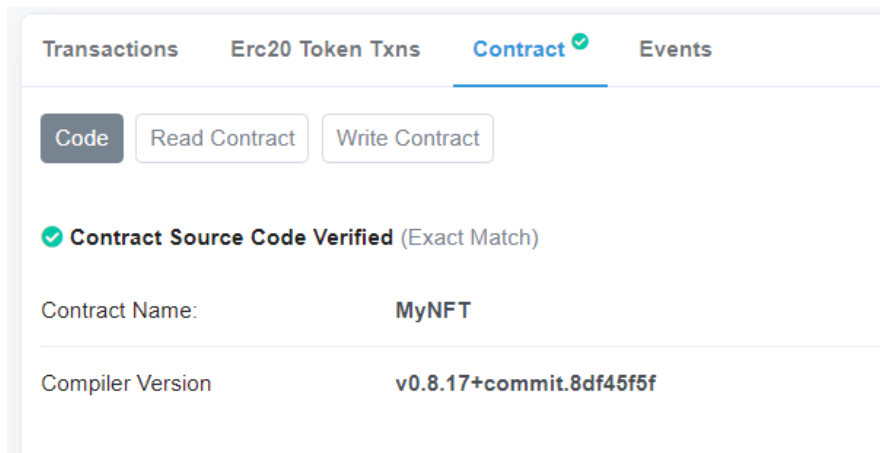
We already added the Etherscan API key to our hardhat.config.js file and all we need to do is execute the following command:

```
npx hardhat verify --network sepolia YOUR_CONTRACT_ADDRESS
```

You should get the following message in your command window:

Successfully verified contract MyNFT on Etherscan.
<https://sepolia.etherscan.io/address/0x04A1Df6c1c0fEcc72F42A2CcaCC720293eC63BdB#code>

Refresh the contract page on Etherscan and now you should see the Solidity code of your smart contract together with the code of all other contracts it inherits from:



Here is again a quick summary of all the commands required to deploy and verify your smart contract:

Compile the contract: `npx hardhat compile`

Run a Hardhat node: `npx hardhat node`

Deploy to Hardhat: `npx hardhat run scripts/deploy.js`

Deploy to Sepolia: `npx hardhat run scripts/deploy.js --network sepolia`

Verify on Etherscan: `npx hardhat verify --network sepolia CONTRACT_ADDRESS`

Using Ethers.js to communicate with our smart contract

Our smart contract is finally on the blockchain. Now, it's time to call some of its functions, like: mintNFT, balanceOf, ownerOf...

And, once again (as in our previous deployment script) we will make use of the ether.js library that will do all of the heavy lifting when it comes to communicating with smart contracts.

Ethers.js:

The official ethers.js docs can be found at: <https://docs.ethers.org/v5/>

Well, there is a lot of information and honestly, it can be a bit confusing when you take a look at the docs for the first time. There are tons of classes, methods and properties and it's not always clear which ones to use and when to use one class instead of another...

Here is a very quick and high-level introduction of Ethers.js:

Ethers.js is a library for interacting with the Ethereum Blockchain. Ethers.js greatly facilitates the process of calling smart contract functions, but it also provides various utility classes that facilitate the interaction with the ABI, the treatment of Byte, String, Address and BigNumber types and much more.

*The most important classes are: **Provider, Signer and Contract***

Provider:

A Provider is an abstraction of a connection to the Ethereum network. A Provider in ethers is a read-only abstraction to access the blockchain data, a signer provides read/write access.

The ethers library offers default API keys in order to access the various networks. However, those API keys don't work very well and I highly recommend to create a free account with one of the third party node providers (as we did earlier on with Alchemy).

Once you have your Alchemy API key, you can set up a connection to the Ethereum mainnet or a test network using the ethers.js AlchemyProvider. If you want to create a connection to a local blockchain, you would use the ethers.js JsonRpcProvider. And, there are various other providers...

However, as we are using Hardhat, we don't even have to worry about those details. Hardhat creates the most adapted provider for us and injects it into the global ethers object.

Remember, we added the following line of code to our hardhat.config.js file:

```
require("@nomicfoundation/hardhat-toolbox")
```

With the Hardhat toolbox, we get access to various software packages, amongst them, an extended version of ethers.js (with some additional methods and properties). And, as we also specified the Alchemy API URL for the Sepolia network, Hardhat has all the details to create a provider for us.

The provider can be accessed by calling:

```
let myProvider = ethers.provider
```

Signer:

A Signer is an abstraction of an Ethereum Account. It has access to a private key and it can be used to sign messages and transactions and to send transactions.

In the hardhat.config.js file, we also specified the private key of one of our accounts (for the setup of the Sepolia network) and this allows Hardhat to automatically create a signer for us.

We can access the signer by calling:

```
let mySigner = ethers.getSigners()
```

If we need additional signers for Sepolia, we can also define a Wallet signer, which requires our Alchemy API URL and the provider instance that Hardhat already created for us:

```
let additionalSigner = new ethers.Wallet(ALCHEMY_API_URL, myProvider)
```

Don't worry, this will become much clearer when we take a look at our code in a minute.

Contract:

The Contract class is used to communicate with our on-chain smart contract and it allows us to call external and public functions on that contract. But how does the contract instance know, which functions are available on our deployed smart contract? It gets that information from the Application Binary Interface (**ABI**) that is generated for us by the compiler together with the smart contract bytecode.

In your project folder, navigate to: artifacts\contracts - you should see a file called: **"MyNFT.json"**. That file was generated by the compiler and it contains the ABI and the contract bytecode.

Hardhat knows where to find the ABI and in order to create a contract instance, the only information you need to provide is the name of your contract (because, you could have several contracts in your project) and the address of your contract:

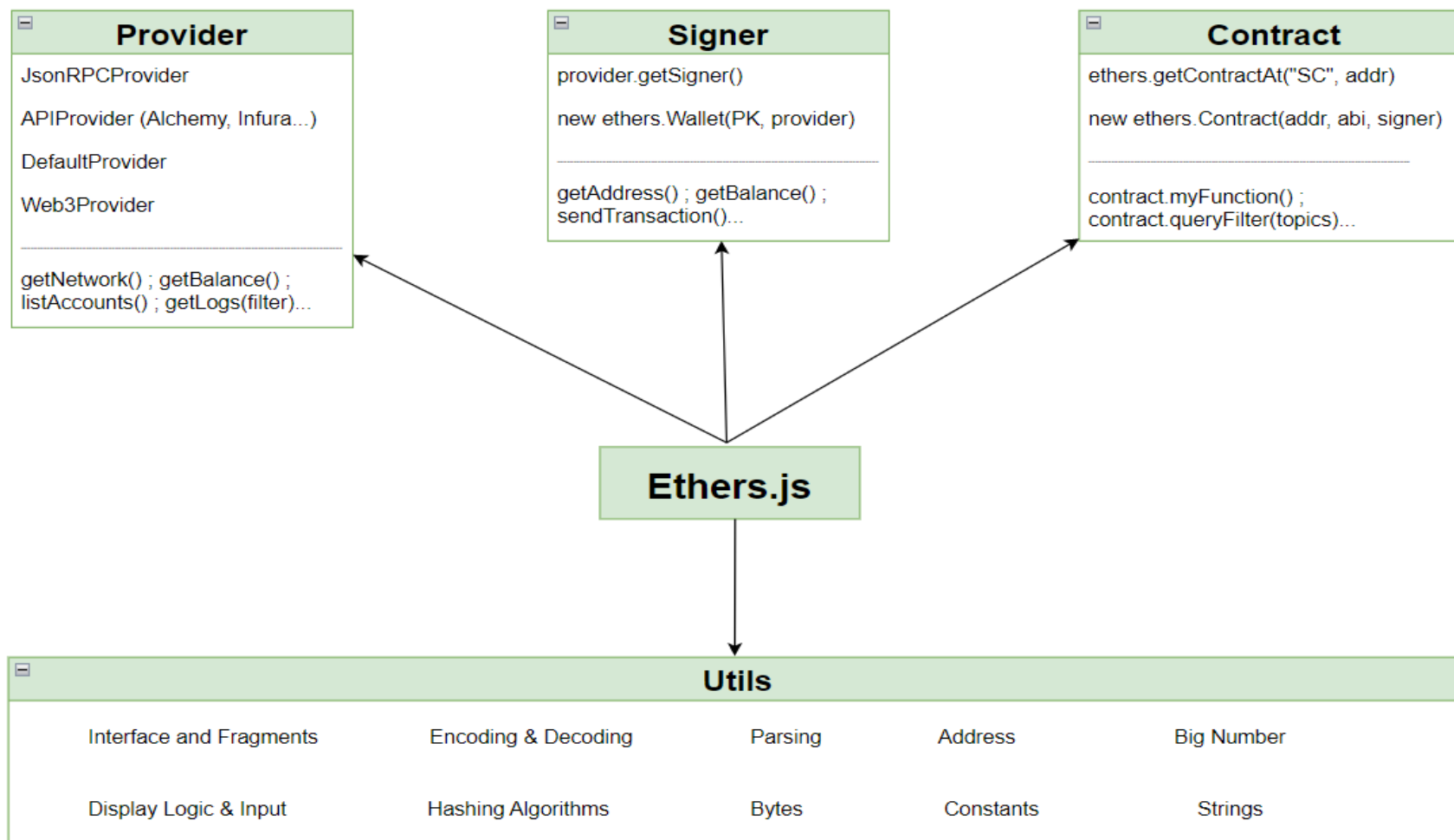
```
let myContract = await ethers.getContractAt("ContractName", "ContractAddress")
```

The method "***getContractAt***" is made available by Hardhat - it does not exist in the standard version of ethers.js

Utilities:

On top of that, ethers.js also provides various *Utility* classes for all kinds of blockchain related purposes, like: encoding and decoding data, big number utilities, hashing algorithm's... and much more.

The image below provides an overview of the various classes together with the most important methods and properties provided by ethers.js:



The NFT minting script:

Ok, it's time to take a look at the code of our NFT minting script. Open the following file from the project you downloaded from GitHub: *scripts/mint-nft.js*

Also, now you will need the URL of the *nft-metadata.json* file you uploaded earlier on to Pinata. Remember, it looks something like:

<https://gateway.pinata.cloud/ipfs/QmcSM8rx...>

```
1  const {
2    REACT_APP_PRIVATE_KEY,
3    REACT_APP_PRIVATE_KEY2,
4    REACT_APP_CONTRACT_ADDRESS,
5    REACT_APP_CONTRACT_ADDRESS_LOCAL,
6  } = process.env
7
8  // nft-metadata.json uploaded to Pinata => contains 2 proper
9  const tokenURI = "https://gateway.pinata.cloud/ipfs/QmPzekhp
10
11 let provider, signer, signer2, contract, txn, txnReceipt
12
```

First, we load the values of our environment variables. Just make sure, you provided the correct private keys (export from Metamask as explained earlier on). And, the contract addresses are correct - the first is from your remote deployment to Sepolia and the second one is from your local deployment to Hardhat or Ganache.

Don't forget to update the remote contract address if you re-compile and re-deploy your contract to Sepolia.

In the next line, you provide the Pinata URL to your *nft-metadata.json* file.

And, finally, we define a few variables we'll be using throughout the script.

```
13  async function main() {
14    provider = ethers.provider
15    const currentNetwork = await provider.getNetwork()
16
```

Our script uses only 1 function (main), which needs to be labeled as "*async*", because we'll be calling various asynchronous functions that need to be awaited.

We get the provider instance from ethers and we call the "getNetwork" function to access information about our currently used network (local Hardhat network or Sepolia).

```

17     if (currentNetwork.chainId.toString().includes(1337)) {
18         console.log("We are using a local network!")
19         contract = await ethers.getContractAt("MyNFT", REACT_APP_CONTRACT_ADDRESS_LOCAL)
20         ;[signer, signer2] = await ethers.getSigners()
21     } else {
22         console.log("We are using a remote network!")
23
24         signer = new ethers.Wallet(REACT_APP_PRIVATE_KEY, provider)
25         // we could also use: signer = await ethers.getSigners()
26         signer2 = new ethers.Wallet(REACT_APP_PRIVATE_KEY2, provider)
27         contract = await ethers.getContractAt("MyNFT", REACT_APP_CONTRACT_ADDRESS)
28     }

```

On the returned network instance, we check the "*chainId*" property. If it contains "1337", we know it is a local network, otherwise it's the Sepolia network. On Ganache, you can modify the chainId (under settings => Server). So, make sure, its value is 1337.

Then, we create a contract instance, by using the "*getContractAt*" function and by specifying the contract name and address (local or remote).

We also need 2 signer instances, because later on we want to transfer an NFT from account1 to account2 and then back from account2 to account1.

On a local network we can simply use the "*getSigners*" function, which returns 20 signer instances for Hardhat and 10 signer instances for Ganache. We use array destructuring and assign the first 2 signers to the variables *signer1* and *signer2*.

For our Sepolia network, we specified only one private key in the *hardhat.config.js* file. So, if we call the *getSigners()* function, of course, we will get only one signer.

To get a second signer, we use the "*Wallet*" function on the ethers object and we specify our second private key and our provider.

```

30     // mint an NFT to signer2
31     txn = await contract.mintNFT(signer2.address, tokenURI)
32     txnReceipt = await txn.wait()

```

Using our contract instance we just created, we call the "*mintNFT*" function and we specify the account address of *signer2* as the recipient and our token metadata file (on IPFS) as the associated *tokenURI*.

```

34 // display how many NFT's (of this specific contract) are owned by the recipient
35 console.log(
36     "Number of NFT's owned by the recipient: ",
37     await contract.balanceOf(signer2.address)
38 )
39
40 // display the owner of NFT with Id = 1
41 console.log("Owner of NFT with Id 1: ", await contract.ownerOf(1))

```

Previously we minted an NFT to the address of signer2. Now, we are using the ERC721 function "*balanceOf*" to verify the balance of our "*MyNFT*" token, which should be 1

Then, we check the owner of the token *MyNFT* with the Id of 1, using the "*ownerOf*" function, which should be the address of signer2

```

43 // transfer NFT with Id = 1 to another account => swap signer and signer2 on
44 //the method call below. If we want to transfer it back, we also need to
45 //change the signer => safeTransfer requires :: from == owner &&
46 //msg.sender == owner
47 contract = await contract.connect(signer2)
48
49 //use this on methods with the same name
50 txn = await contract["safeTransferFrom(address,address,uint256)"](
51     signer2.address,
52     signer.address,
53     1
54 )
55 txnReceipt = await txn.wait()

```

Next, we want to transfer the token with the Id 1 from signer2 (the current owner) to signer1 using the ERC721 function "*safeTransferFrom*". Currently, when we call a function on our contract instance, we are calling it in the context of signer1, because this is our default signer. In order to change the caller, we can use the "*connect*" function on the contract instance and we provide the signer with which we want call/sign our contract functions/transaction.

Normally, we could simply call: `contract.safeTransferFrom(...)`

However, a function with same name is defined several times in the contracts we are inheriting our *MyNFT* contract from. The only way to call the correct function (the one that specifies a *from* and *to* address and the *tokenId* as uint) is by specifying the exact function signature with all the correct argument data types in angle brackets.

Calling this function provides us with a transaction response on which we need to call the "*wait*" function, which returns as soon as the transaction has been integrated in a new block.

Calling the *ownerOf* function again for tokenId 1, should now return the address of signer1

Executing the script on a local node:

1: Make sure, either Ganache or Hardhat is not (not both at the same time):

```
npx hardhat node
```

2: Deploy your smart contract to a local node (if it's not already done):

```
npx hardhat run scripts/deploy.js
```

3: Verify the contract address and if necessary, update the one you have in your .env file at: **REACT_APP_CONTRACT_ADDRESS_LOCAL**

4: Run the script we just created locally:

```
npx hardhat run scripts/mint-nft.js
```

You should see something similar in your terminal window:

```
$ npx hardhat run scripts/mint-nft.js
We are using a local network!
Number of NFT's owned by the recipient: BigNumber { value: "1" }
Owner of NFT with Id 1: 0x70997970C51812dc3A010C7d01b50e0d17dc79C8
New owner of NFT with Id 1: 0xf39Fd6e51aad88F6F4ce6aB8827279cFfFb92266
```

Executing the script on Sepolia:

1: Deploy the contract to Sepolia:

```
npx hardhat run scripts.deploy.js --network sepolia
```

2: Verify the contract address and if necessary, update the one you have in your .env file at: **REACT_APP_CONTRACT_ADDRESS**

3: Run the script we just created on Sepolia:

```
npx hardhat run scripts/mint-nft.js --network sepolia
```

The log in your terminal window will be similar as above, but now it will display: "We are using a remote network"

Testing our smart contract

Testing is crucially important for any smart contract project and you should never publish your smart contracts to the mainnet without extensive prior testing.

Hardhat comes with a fully integrated test framework that uses *Mocha* as test runner and *Chai* as assertion library. Additionally, custom *Hardhat-Chai matchers* (for various blockchain related features) and the *Hardhat Network Helpers* are provided. Tests are executed on the local Hardhat network.

Test files can be added to the "test" folder in the root of your Hardhat project. The basic structure of a test file looks something like the following:

```
1 describe("Contract...", function () {
2   async function myFixture() {
3     ...
4   }
5
6   describe("Test group 1", function () {
7     it("Some description...", async function () {
8       const { var1, var2... } = await loadFixture(myFixture)
9       ...
10    });
11    ...
12    it("other test description...", async function () {
13      ...
14    });
15  }
16
17  describe("Test group 2", function () {
18    it("Some description...", async function () {
19      ...
20    });
21    ...
22  }
23 });
```

You can organize related tests into groups, which are defined with the "*describe*" keyword. With the describe keyword, you need to provide a description of your test group and a callback function that lists the individual tests.

You can create as many describe blocks as you want and you can also nest several describe blocks inside of another describe block.

Individual tests are defined with the "*it*" keyword and you need to provide a description of your test and a callback function that contains the actual test code.

Of course, you can organize your tests into several individual files within the "test" folder.

Writing the tests for our MyNFT contract:

Open the file: `test/tests.js` in the project folder you downloaded from Github.

Those are typical smart contract tests that use some of the methods and features provided by the Hardhat test framework. However, we won't cover all the available features and functions provided by Hardhat and Chai.

For additional information, check out the following sources on Hardhat:

- <https://hardhat.org/tutorial/testing-contracts>
- <https://hardhat.org/hardhat-runner/docs/guides/test-contracts>

Also, our smart contract is very simple, there is basically only one function: `mintNFT`. However, to demonstrate various Hardhat test features, I will also test some the functions provided by OpenZeppelin contracts we are inheriting from - which isn't really necessary.

Ok, let's take a look at the code...

```
1  const { expect } = require("chai")
2  const { loadFixture } = require("@nomicfoundation/hardhat-network-helpers")
3  const { anyValue } = require("@nomicfoundation/hardhat-chai-matchers/withArgs")
4
5  const tokenURI = "https://gateway.pinata.cloud/ipfs/QmPzekhpuWN2j5yXome5dJYHy2KYH"
```

Here, we are loading all required packages. Chai allows us to perform assertions. They are used in the following way:

expect(result of contract call...).to.eq(the value that should be returned...)

The *hardhat-network-helpers* provide us with various test features. The one we are interested in, is the *loadFixture* component.

In a typical Mocha test, the duplication of code (e.g.: deploying a contract) is handled with a *beforeEach* hook that is executed before each individual test is run. Deploying our contract(s) each time before we run a test will considerably slow down the execution of our tests, which is not ideal.

The *loadFixture* helper fixes this problem. The first time *loadFixture* is called, the code in the fixture is executed (typically deploying one or several contracts). But, on subsequent calls, *loadFixture* will simply reset the network to the state where it was right after calling the fixture the first time. This is much faster than doing a complete deployment of all contracts for all of our test functions.

As you already know, Solidity events can contain several arguments and in our tests, we can assert the value of those arguments. Sometimes, you may want to assert the second argument of an event, but you don't really care about the first argument. It could have any value and you are fine with that. In that case, you can provide the *anyValue* helper for that first argument.

We'll take a look at a specific example in a minute...

Creating several test groups:

```
7 describe("MyNFT contract", function () {
8 >   async function deployContractFixture() {...
15   }
16
17 >   describe("Deployment", function () {...
29     })
30
31 >   describe("Features of the MyNFT Contract", function () {
88     })
89
90 >   describe("Events", function () {...
98     })
99
100 >   describe("Revert", function () {...
109     })
110 })
```

I created several test groups: Deployment, Features, Events... but, you could organize your tests in a complete different way. You could use one single describe block, or several test files...

Deploying our contract before running each test:

```
8   async function deployContractFixture() {
9     const [deployer, user] = await ethers.getSigners()
10
11     const MyNFTFactory = await ethers.getContractFactory("MyNFT", deployer)
12     const myNFTContract = await MyNFTFactory.deploy()
13
14     return { myNFTContract, deployer, user }
15   }
```

Here, we are creating a *fixture* that deploys our contract and will be used by all our tests. It's a basic function that uses the standard contract deployment code you already know. The function returns 2 signers and an instance of the deployed contract.

Asserting basic properties after contract deployment:

```
17 describe("Deployment", function () {
18     it("Should set the right owner", async function () {
19         const { myNFTContract, deployer } = await loadFixture(deployContractFixture)
20         expect(await myNFTContract.owner()).to.equal(deployer.address)
21     })
22
23     it("Should set the right token name", async function () {
24         const { myNFTContract } = await loadFixture(deployContractFixture)
25         expect(await myNFTContract.name()).to.equal("MyNFT")
26     })
27 })
```

Our first 2 tests. In all our tests, we'll be calling our deployment fixture, using the "*loadFixture*" helper. On the returned contract instance, we can now call the functions we want to test.

The syntax: *expect().to.eq()* allows us to assert the returned value. If it is different from the value we provide, the test will fail.

The first function tests if the deployer of the contract is also the owner. The owner function is provided by the *ownable* contract from OpenZeppelin. So, here we are basically testing an OpenZeppelin contract, which should not be really necessary. But, as I already mentioned, this is only for demonstration purposes.

The second test asserts the name of our contract - the name we provided in the ERC721 constructor.

Minting and transferring tokens:

```
30     it("Should mint a token to specified address", async function () {
31         const { myNFTContract, user } = await loadFixture(deployContractFixture)
32
33         await myNFTContract.mintNFT(user.address, tokenURI)
34
35         expect(await myNFTContract.ownerOf(1)).to.eq(user.address)
36     })
```

We are calling the mintNFT function and then we assert the owner of the token with Id = 1, which must be the recipient specified (user.address) in the mintNFT function.


```

38     it("Should transfer token ownership to user", async function () {
39         const { myNFTContract, deployer, user } = await loadFixture(deployContractFixture)
40
41         await myNFTContract.mintNFT(deployer.address, tokenURI)
42         await myNFTContract["safeTransferFrom(address,address,uint256)"](
43             deployer.address,
44             user.address,
45             1
46         )
47
48         expect(await myNFTContract.ownerOf(1)).to.eq(user.address)
49     })

```

In this test, we first call the `mintNFT` function and then we transfer the token from "*deployer*" to "*user*". We then assert the owner of the token.

```

51     it("Should return the correct tokenURI", async function () {
52         const { myNFTContract, deployer } = await loadFixture(deployContractFixture)
53
54         await myNFTContract.mintNFT(deployer.address, tokenURI)
55
56         await expect(await myNFTContract.tokenURI(1)).to.eq(tokenURI)
57     })

```

Here, we assert the *tokenURI* of our freshly minted token.

Asserting token balances:

```

59     it("Should change token balance of sender after transfer", async function () {
60         const { myNFTContract, deployer, user } = await loadFixture(deployContractFixture)
61
62         await myNFTContract.mintNFT(deployer.address, tokenURI)
63
64         await expect(
65             myNFTContract["safeTransferFrom(address,address,uint256)"](
66                 deployer.address,
67                 user.address,
68                 1
69             )
70         ).to.changeTokenBalance(myNFTContract, deployer.address, -1)
71     })

```

The "*expect().to.changeTokenBalance()*" function allows us to assert the change of token balance after a token transfer. The first argument of the "*changeTokenBalance*" needs to be the contract on which we would like to assert the change of balance, the second argument is the address for which we want to assert the change and the third argument is the number of

token the balance changed. In our example, the token balance of the deployer was reduced by 1, because we transferred exactly 1 token.

```
78     await expect(  
79         myNFTContract["safeTransferFrom(address,address,uint256)"](   
80             deployer.address,   
81             user.address,   
82             1   
83         )   
84     ).to.changeTokenBalances(myNFTContract, [deployer.address, user.address], [-1, 1])
```

Here, we use almost the same function as in the previous test: "**changeTokenBalances**" - this allows us to assert the change in balance for both accounts. For the second argument, we provide an array of both accounts and for the third argument, we provide an array with the balance changes for those 2 accounts.

Asserting Events:

```
88     describe("Events", function () {   
89         it("Should emit the Transfer event on token mint", async function () {   
90             const { myNFTContract, user } = await loadFixture(deployContractFixture)   
91   
92             await expect(myNFTContract.mintNFT(user.address, tokenURI))   
93                 .to.emit(myNFTContract, "Transfer")   
94                 .withArgs(ethers.constants.AddressZero, user.address, anyValue) // We   
95         })   
96     })
```

This test shows how we can verify if an event was triggered by the smart contract. When we call the *mintNFT* function, a "Transfer" event will be emitted. Our mintNFT function calls the internal *_mint* function in the **ERC721** contract from which our contract inherits from (via the **ERC721URIStorage** contract)

Check out the code at: <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol> => ~line 283 :

```
283         emit Transfer(address(0), to, tokenId);
```

We use the "*expect().to.emit()*" syntax. The first argument of emit needs to specify the contract on which we want to assert the event emission, the second argument specifies the name of the event that should be triggered.

And, we can also assert the arguments of the event, using "*withArgs()*". The first argument must be the zero address (ethers.js defines a constant for this: *AddressZero*). The second argument is the recipient of the token and we don't care about the third argument (the token id), so, we specify the "*anyValue*" helper we imported at the beginning of our script.

Asserting reverted functions:

```
98 describe("Revert", function () {
99     it("Should revert with invalid token ID", async function () {
100         const { myNFTContract, deployer, user } = await loadFixture(deployContractFixture)
101
102         await myNFTContract.mintNFT(deployer.address, tokenURI)
103
104         await expect(myNFTContract.tokenURI(10)).to.be.revertedWith("ERC721: invalid token ID")
105     })
106 })
```

Our final function allows us to assert if a function properly reverts if we specify an invalid parameter. We mint a token to the deployer address and then we call the tokenURI function with a non-existent token Id.

For the assert, we use: "*to.be.revertedWith()*" and we need to specify the correct error message. How do we know, the error message should be: "*ERC721: invalid token ID*" ?

We take a look at the "tokenURI" function defined in the OpenZeppelin contract, at: <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol> => ~ line 386 :

```
385     function _requireMinted(uint256 tokenId) internal view virtual {
386         require(_exists(tokenId), "ERC721: invalid token ID");
387     }
```

Running our tests:

Running your tests on a standalone Hardhat node:

- Launch a Hardhat node: `npx hardhat node`
- Run all tests: `npx hardhat test`
- You can also run an individual test file: `npx hardhat test test/my-tests.js`

Running your tests on an in-memory Hardhat node:

You could also run your tests on an in-memory instance of Hardhat without having to launch a standalone Hardhat node. In that case, you have to use the `--network` flag and specify `hardhat` as your target network:

- `npx hardhat test --network hardhat`

You should see something similar in your command window:

```
MyNFT contract
Deployment
  ✓ Should set the right owner (927ms)
  ✓ Should set the right token name
Features of the MyNFT Contract
  ✓ Should mint a token to specified address (177ms)
  ✓ Should transfer token ownership to user (362ms)
  ✓ Should return the correct tokenURI (110ms)
  ✓ Should change token balance of sender after transfer (317ms)
  ✓ Should change token balance of sender and receiver after transfer (377ms)
Events
  ✓ Should emit the Transfer event on token mint (126ms)
Revert
  ✓ Should revert with invalid token ID (205ms)

9 passing (3s)
```

Creating the DAPP with React

The final step of our project is the creation a web frontend. We'll be using React for that purpose. React is by far the most used framework for web user interfaces in the blockchain space.

React is a JavaScript library that allows to build reusable UI components and efficiently manage the state of the application. React uses a virtual DOM that greatly improves the performance of the application and only updates those components that have changed, rather than re-rendering the entire page on every small change.

React, of course is a huge subject and it is not the goal of this tutorial to teach you React, JavaScript, CSS or any other frontend technologies. I will show you how to quickly bootstrap a React project and I will explain those code sections that communicate with Metamask and our smart contract, but we won't be spending time on any other React related subjects.

If you want to learn more about React, I recommend you check out the official ReactJS website: <https://reactjs.org/>

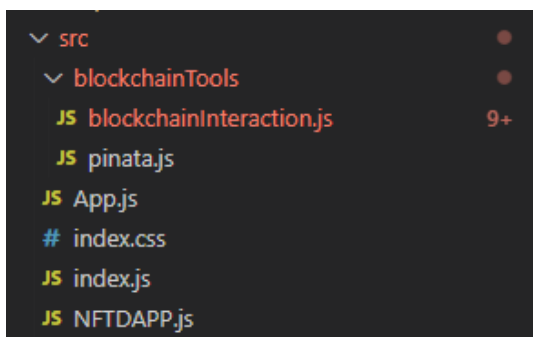
You can also find great beginner courses on YouTube. Here are 2 great resources:

- <https://www.youtube.com/watch?v=RVFAyFWO4go>
- <https://www.youtube.com/watch?v=bMknfKXIFA8>

The main files of our React project:

As usual, you can find all the code in the project you downloaded from Github: <https://github.com/rspadinger/NFT-DAPP>

The frontend code is in the "src" folder and contains the following files:



The entry point of the application is the *index.js* file that renders one single component (the "App" component) in a "div" element with the id = "root". This div element is defined in the *index.html* file that is located in the **public** folder:

```
7 root.render(  
8   <React.StrictMode>  
9     <App />  
10  </React.StrictMode>
```

The *index.css* file contains the styling for our application. This is basic CSS code.

The *App.js* file contains the App component we just mentioned and all it is doing is displaying our main component - the *NFTDAPP* component - in a div element:

```
4   return (  
5     <div className="App">  
6       <NFTDAPP></NFTDAPP>  
7     </div>
```

The following components contain blockchain relevant code and will be discussed a bit later:

- *blockchainTools/blockchaininteraction.js*: this file contains methods that allow us to connect to Metamask and to create and send a transaction via Metamask to our deployed smart contract.
- *blockchainTools/piñata.js*: the code in this file allows us to pin a json metadata file to an IPFS node using the Pinata API.
- *NFTDAPP.js*: The main (and only) React component that creates our user interface.

Running our web application:

Open a command window, navigate to your project folder and execute the following command:

```
npm run start
```


The application should be launched in your default browser at the following URL:


<http://localhost:3000/>


You should see the following web page:

NFT DAPP


[Connect Wallet](#)

 NFT Image URL:
jhjg

 NFT Name:
hjgjh

 NFT Description:
jhgjhg

[Mint NFT](#)

 [Connect to Metamask using the top right button.](#)

The DAPP is not yet connected with Metamask. Login to Metamask and click the "**Connect Wallet**" button on your DAPP. You should see a Metamask popup that asks you to connect your DAPP with one or more accounts. Select your account that contains Sepolia test Ether and also one of your Hardhat accounts you imported earlier on. Click "**Next**" and "**Connect**"

Your DAPP should now display the address of the wallet selected in Metamask:


NFT DAPP

[Connected: 0xb1b5...2803](#)

Testing the app on a local Hardhat node:

- Launch a Hardhat node: `npx hardhat node`
- Deploy your contract to Hardhat: `npx hardhat run scripts/deploy.js`
- Make sure you have the correct contract address (local) in the .env file
- Select your Hardhat account and the Hardhat network in Metamask
- Provide an IPFS image URL (something like: <https://gateway.pinata.cloud/ipfs/QmcSM8...>), a name and a description and click the "Mint NFT" button.
- You will see a Metamask popup asking you to sign and send a transaction. Click the "**Confirm**" button.

DETAILS DATA HEX

 Market >

Gas (estimated) ⓘ	\$0.13	0.00008214 ETH
Likely in < 30 seconds	Max fee:	0.00008819 ETH

Total	\$0.13	0.00008214 ETH
Amount + gas fee	Max amount:	0.00008819 ETH

CUSTOM NONCE

Testing the app on Sepolia:

- Deploy your contract to Sepolia: `npx hardhat run scripts/deploy.js --network sepolia`
- Make sure you have the correct contract address (remote) in the .env file
- Select your Sepolia account and the Sepolia network in Metamask
- Once again, provide an IPFS image URL, a name and a description and click the "Mint NFT" button in your DAPP.
- On the Metamask popup, click the "**Confirm**" button.
- On your DAPP, click the button: "**Check out your transaction on Etherscan**"

You should see the details of your mintNFT transaction:

Transaction Details < >

Overview Logs (1) State

[This is a Sepolia Testnet transaction only]

Transaction Hash:	0xb5321b1e75b60237c6b5a4752136a0d7da66ab7339b6f73621bb094634fd9bd5
Status:	Success
Block:	3155284 35 Block Confirmations
Timestamp:	7 mins ago (Mar-24-2023 05:34:36 PM +UTC)
From:	0xB1b504848e1a5e90aEAA1D03A06ECEee55562803
To:	[0x71f4708b58cc43d2324e24253bbfcee6b41ebf4d Created]
Value:	0 ETH (\$0.00)
Transaction Fee:	0.001423061018499793 ETH (\$0.00)
Gas Price:	1.000000013 Gwei (0.000000001000000013 ETH)

Importing your NFT into Metamask:

- Open Metamask, click the "Assets" tab, scroll down and click "**Import Tokens**"
- Provide your contract address, "**MNFT**" as the token symbol, 0 for decimals, click "**Add custom token**" and then "**Import tokens**".

Now, you should see your MNFT token in Metamask on the Assets page.

You can also check out your NFT with all its properties and attributes on the OpenSea testnet page: <https://testnets.opensea.io/account>

Bootstrapping a React project:

To create a React project from scratch, open a command window, navigate to the folder where you would like to create your project and execute the following command:

```
npx create-react-app your-app-name
```

To run this project, execute the following command:

```
npm run start
```

In the project you downloaded from GitHub, I put the React project in the same folder as the Hardhat project. To do so, perform the following steps:

- Copy/paste the *src* and the *public* folder of the React project into the root of the Hardhat project
- Rename the *package.json* file of the Hardhat project to *package2.json*
- Copy the package.json file of the React project into the root of the Hardhat project
- From the package2.json file, copy the entire "*devDependencies*" section into the package.json file
- From the "*dependencies*" section of the package2.json file, copy the following packages (including the version) into your new package.json file:
"*@openzeppelin/contracts*": ... and "*dotenv*": ...
- You can now delete the he package2.json file
- Open a command window, navigate to the root folder of your Hardhat project and execute the command: **npm install** => this will install all the packages required by React into the *node_modules* folder of your Hardhat project

Let's do some cleanup in our React project:

- Delete the following files from your React project:
 - public/logo192.png
 - public/logo512.png
 - public/robots.txt
 - src/App.css => we put all the css code into the index.css file
 - src/App.test.js
 - src/logo.svg
 - src/reportWebVitals.js
 - src/setupTest.js
- Replace the code in the following files with the code from the project you downloaded from GitHub:

- src/App.js
 - src/index.css
 - src/index.js
- Add the following files and folders to your project from the project you downloaded from GitHub:
 - blockchainTools folder
 - NFTDAPP.js file
 - Install the ethers and axios packages: `npm install axios ethers`

The blockchain related code of our React application:

blockchainInteraction.js

At the beginning of the file we import the ethers package and the piñata.js file. We also get the contract addresses from our .env file.

Then, we use a self-executing function to assign the correct value to the *contractAddress* variable depending on the network we are using.

```

8  if (window.ethereum) {
9      provider = new ethers.providers.Web3Provider(window.ethereum)
10     const currentNetwork = await provider.getNetwork()

```

Our React app is a web application and therefore it has access to the global *window* namespace. Metamask automatically injects the *ethereum* object into the window namespace.

Calling *window.ethereum* allow us to verify if Metamask is installed. Because we are running a web application, we can use the ethers *Web3Provider*. On the returned provider instance, we call the *getNetwork* function we already used in our test script.

In the *getCurrentWalletConnected* function, we verify once again if Metamask is installed. If it is not installed, we ask the user to install it (in the *else* block at the end of the function). We use this technique on all other functions as well.

```

23     const addressArray = await window.ethereum.request({
24         method: "eth_accounts",
25     })
26
27     if (addressArray.length > 0) {
28         selectedAddress = addressArray[0]

```

Then, we perform Metamask RPC call. The syntax for those calls is always the same, only the method changes and some additional parameters may be added:

```
const result = await window.ethereum.request({method: "eth_someRPCMethod", ...
```

Here, we are using the *eth_accounts* methods, which returns all Metamask accounts that are connected to our DAPP. If the call returns one or more accounts, we assign the first one to our *selectedAddress* variable.

This method will be called by our *NFTDAPP* React component and we will always return an object with 2 properties: *address and status*. Our React component will display that information to the user.

You may have realized that there are some emoji's in the code. To add emoji's to your code, add the "*Emoji*" plugin to VSCode and then type: *Ctrl+Shift+P => insert Emoji* and then simply click on the emoji you want to add to your code.

The *connectWallet* function is almost the same. The only difference is that we are making a call to the "*eth_requestAccounts*" Metamask RPC method. We call this function if our DAPP is not yet connected to Metamask - in other words, if the previous function does not return any accounts.

```
66     const addressArray = await window.ethereum.request({
67       method: "eth_requestAccounts",
68     })
```

The "*eth_requestAccounts*" method causes Metamask to display a popup with all Metamask accounts and it allows us to connect our DAPP to one or more of those accounts.

The "*mintNFT*" function creates a transaction that allows us to call the *mintNFT* function on our smart contract. At the beginning of the function, we perform some basic validation to make sure the user provides all required properties.

Then, we create a metadata file for our NFT that contains the properties provided by the user (imageURL, name and description) and some other hard-coded attributes (Fur and Eye color). The properties of the metadata object (*pinataMetadata and pinataContent*) are defined by Pinata.

Here are the Pinata docs: <https://docs.pinata.cloud/pinata-api/pinning/pin-json>

Then, we call the *pinJSONToIPFS* function in our *pinata.js* file. This function takes our metadata object, converts it into a JSON file, uploads it to your Pinata account, pins it to an IPFS node and returns the URL to that JSON metadata file.

Creating the transaction for Metamask:

```
150 let iface = new ethers.utils.Interface(["function mintNFT(address recipient, string memory tokenURI)"])
151 const myData = iface.encodeFunctionData("mintNFT", [selectedAddress, tokenURI])
152
153 const transactionParameters = {
154     to: contractAddress,
155     from: selectedAddress,
156     data: myData,
157 }
```

In our transaction we need to specify 3 parameters:

- **to**: this is the recipient address of the transaction. We want to call a method on our smart contract, so, obviously, this needs to be the address of our smart contract.
- **from**: this is the sender of the transaction, which is our currently selected Metamask address.
- **data**: this is the data we will forward to our smart contract. What do we want to do? We want to call the "*mintNFT*" function on our contract. So, that's the information we need to provide in our data field.

However, we can't just specify "mintNFT...". Our deployed smart contract contains only bytecode and can't understand the meaning of "mintNFT". That's why we need to translate (encode) the instructions for the method call.

To encode our function call, we first create an *interface* for the exact function signature with the correct data types for the required arguments. On that interface instance, we can then use the "*encodeFunctionData*" function to correctly encode our transaction calldata.

The required arguments for that function are the name of the function we want to call and an array of all the function arguments (the recipient of our minted NFT and the metadata URL).

Sending the transaction:

```
160 const txHash = await window.ethereum.request({
161     method: "eth_sendTransaction",
162     params: [transactionParameters],
163 })
```

To forward the transaction to Metamask, we use the "*eth_sendTransaction*" method and of course, we also need to specify our transaction parameters. Metamask will display a popup with the transaction details and ask you to confirm or reject the transaction.

pinata.js

This file contains the *pinJSONToIPFS* function that converts a metadata object into a JSON file, uploads it to Pinata, pins it to an IPFS node and returns the URL to the pinned JSON metadata file.

```
6     const data = JSON.stringify(metadata)
7     const config = {
8       method: "post",
9       url: "https://api.pinata.cloud/pinning/pinJSONToIPFS",
10      headers: {
11        "Content-Type": "application/json",
12        pinata_api_key: REACT_APP_PINATA_KEY,
13        pinata_secret_api_key: REACT_APP_PINATA_SECRET,
14      },
15      data: data,
```

We use the axios component to call the Pinata *pinJSONToIPFS* API method. We configure our API call with the correct URL and in the call header we specify our Pinata key and secret. We also need to provide the data for our API call, which is our metadata object converted into a string.

```
19     const res = await axios(config)
20     return {
21       success: true,
22       pinataUrl: "https://gateway.pinata.cloud/ipfs/" + res.data.IpfsHash,
23     }
```

Then, we simply perform the API call and we await the result. If everything works out fine, the API call will return the IPFS hash for the pinned JSON file and we simply prefix this value with the URL for the Pinata gateway and we return the URL

NFTDAPP.js

This is the only React component of our project. We are using 2 standard React hooks: *useState* and *useEffect*. They are commonly used in pretty much all React components.

The *useState* hook allows us to define state variables in a React component. The hook (or function) allows us to define the initial value of our state variable and it returns a variable that holds the current state value and a function we can use to update the value of our state variable.

The *useEffect* hook is typically used to initialize a component by fetching data, directly updating the DOM, using timer functions, like: `setTimeout()`...

```

15  useEffect(() => {
16    async function init() {
17      const { address, status } = await getCurrentWalletConnected()
18      setWallet(address)
19      setStatus(status)
20      addWalletListener()
21    }

```

In our component, we import those 2 React hooks and our functions from `blockchainInteraction.js`. Then, we setup various state variables and we initialize the component by calling the `getCurrentWalletConnected` function. We use the return values of that function to update a few state variables (wallet address and status message).

```

26    if (window.ethereum) {
27      window.ethereum.on("accountsChanged", (accounts) => {
28        if (accounts.length > 0) {
29          setWallet(accounts[0])
30          setStatus("👉 Provide an image url, a name and

```

Finally, we call the `addWalletListener` function that listens to the `accountsChanged` event from Metamask and updates our wallet address state variable whenever we select a different wallet in Metamask

```

49    const connectWalletPressed = async () => {
50      const { address, status } = await connectWallet()
51      setStatus(status)
52      setWallet(address)
53    }
54
55    const onMintPressed = async () => {
56      const { status } = await mintNFT(name, description, url)
57      setStatus(status)
58    }

```

Those are 2 simply click handlers that are called when the user clicks the "`Connect`" button respectively the "`mint NFT`" button. The handlers call the corresponding function that is defined in `blockchainInteraction.js` file and use the return value(s) to update the status and wallet state variables.

The return section of the component generates the HTML of our component. This is basic HTML and JavaScript code. We have to use curly braces to add JavaScript to our HTML code.

```
63     <button id="walletButton" onClick={connectWalletPressed}>
64         {walletAddress.length > 0 ? (
65             "Connected: " +
```

We define a button element and we use the *onClick* handler to call the *connectWalletPressed* function we defined above.

```
88     <input
89         type="text"
90         placeholder="Some details about your NFT..."
91         onChange={(e) => setDescription(e.target.value)}
92     />
```

That's what's called a *controlled component* and they are commonly used in React. The data of a controlled component (an input field in our case) is handled by the component's state (in this example: description). The component makes changes through callbacks like *onChange*. Whenever the *onChange* event fires, the corresponding state variable is updated using the *setDescription* function.

Adding your project to Git and GitHub

If you want, you can add your project to GitHub. But, first, you need to install Git if that's not already done.

Here is the download link for Git: <https://git-scm.com/downloads>

You also need to create an account on GitHub, which is a web-based source control and collaborative tool.

Github: <https://github.com/>

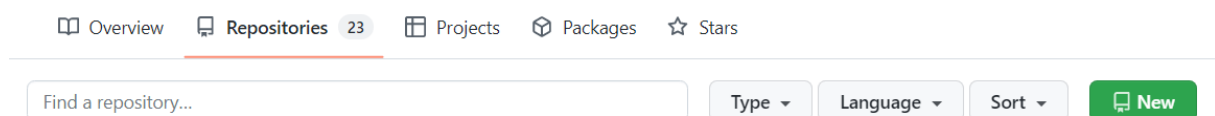
First, you need to add your source code to a local Git repository. Open a new command window, navigate to your project directory and execute the following commands:

```
git init                => this creates a new Git repository  
git add .              => this adds all changes to the staging area  
git commit -m "comment" => this will save all staged changes
```

Whenever you make changes to any of your project files, you need to re-execute the **git add .** and the **git commit -m "some comment..."** commands.

All your project files have now been committed to a local Git repository and you can now add your project to a remote GitHub repository.

First, let's create a new GitHub repository. Click the "**Repositories**" tab and then click the "**New**" button:





Provide a name for the repository - you can leave all other options as they are - and click the "**Create Repository**" button:

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)



Owner * Repository name *

 rspadinger ▾ / NFT-DAPP 

Great repository names are short and memorable. Need inspiration? How about [musical-octo-telegram?](#)

Description (optional)

A simple DAPP that allows to mint NFT's

-
-  **Public**
Anyone on the internet can see this repository. You choose who can commit.
-  **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

- Add a README file**
This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore

Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template: None ▾

Choose a license

A license tells others what they can and can't do with your code. [Learn more.](#)

License: None ▾

 You are creating a public repository in your personal account.

[Create repository](#)

In your command window, execute the following commands:

- ***git remote add origin*** <https://github.com/yourAccount/yourRepository.git> => use the link for your own repository provided by GitHub: this creates a new remote repository called "origin" that's located at the provided location
- ***git push -u origin master*** => this pushes your committed local changes (from the master branch) to your remote repository (origin)
- ***git pull*** => you can use this command to update your local repository with any changes from your remote GitHub repository

Final thoughts & tips

You should now have a good basic understanding of smart contract development for EVM-based blockchains, but there is still a lot to learn. Here are some actions you can take to improve your knowledge and get more experience in the field of smart contract / Web3 development:

- Read the official Solidity docs at: <https://docs.soliditylang.org/en/v0.8.18/> or download the latest version of the official Solidity documentation from: https://docs.soliditylang.org/_downloads/en/latest/pdf/
- On the official Solidity docs, check out the chapter: "Solidity by Example" and try to code all the listed examples yourself.
- Study the code of some of the OpenZeppelin contracts and libraries and apply what you learn to your own smart contracts: <https://github.com/OpenZeppelin/openzeppelin-contracts>
- Digg a bit deeper into ethers.js and start using the provided functions and features in your scripts and Web3 apps: <https://docs.ethers.org/v5/>
- It is also a good idea to learn more about React development. There are tons of resources on the official React website: <https://reactjs.org/> And, you can also find some very good and free tutorials on YouTube:
 - <https://www.youtube.com/watch?v=RVFAyFWO4go>
- Fork a real-world DAPP from GitHub, study the code and start applying the techniques and concepts you learn about on your own projects. For example, you could take a look at the Uniswap project. There are tons of repositories, but you could get started with the following ones:
 - V3-core smart contracts: <https://github.com/Uniswap/v3-core>
 - React UI (quite advanced): <https://github.com/Uniswap/interface>

Here are some additional resources I provide on blockchain development

- Articles on blockchain development: <https://blockchaindevtips.com/>
- Videos on smart contract development and related subjects: <https://www.youtube.com/@BlockchainDevTips>
- Stay up to date with the latest developments in the blockchain space: <https://twitter.com/bchaindevtips>