



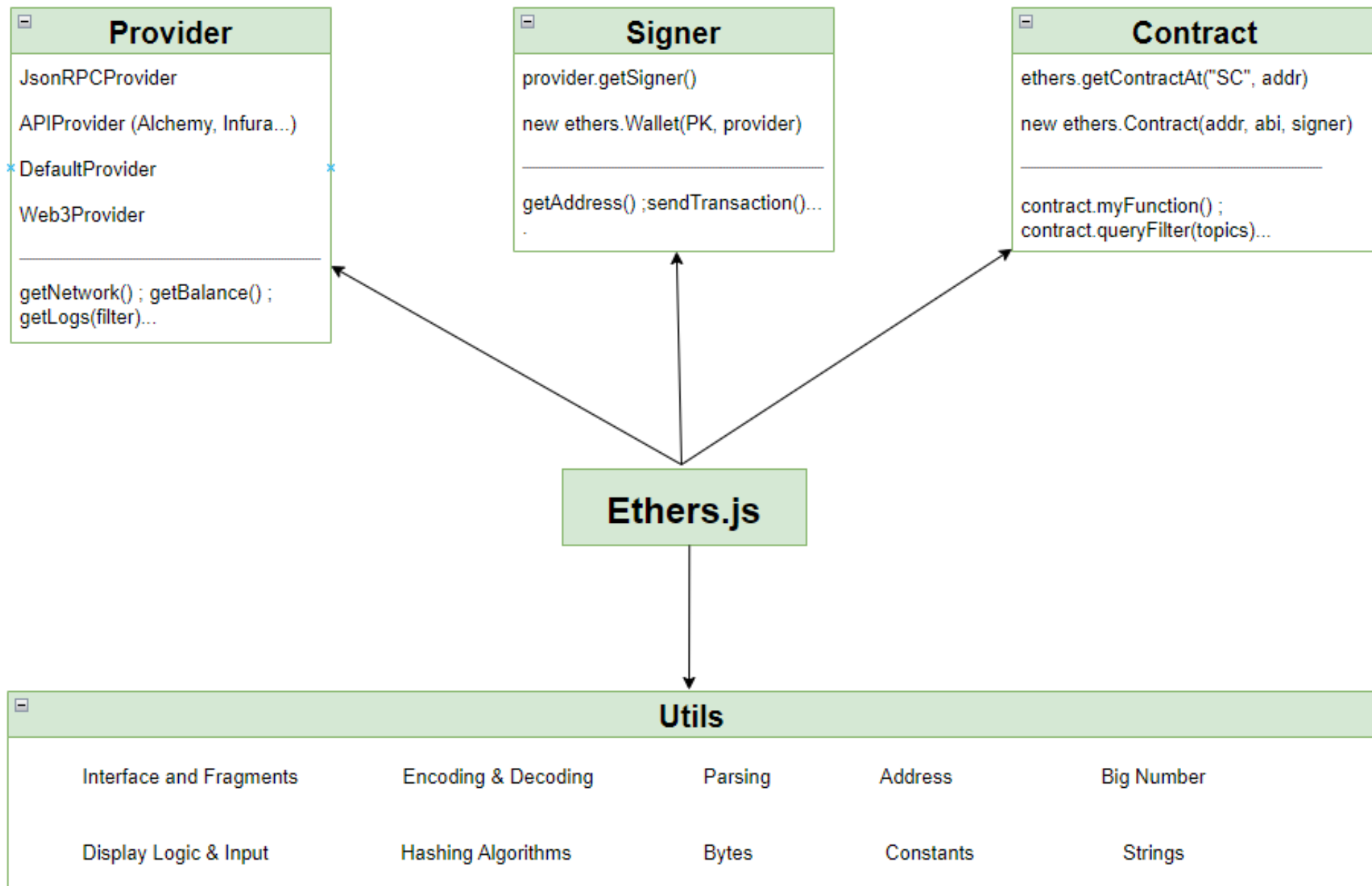
# ETHEREUM SMART CONTRACT DEVELOPMENT

Ethers.js – Testing – ERC20  
ERC721 – Reentrancy - DAO

# Ethers.js

- JavaScript library for interacting with the Ethereum blockchain
- Facilitates the process of calling smart contract functions
- Most important classes: **Provider, Signer and Contract**
- Various utility classes (**Utils**) that facilitates the interaction with the ABI, the treatment of byte, string, address and BigNumber types...
- **Provider:** provides a connection to the Ethereum network - enables read-only access to the Blockchain
- **Signer:** has access to a private key and can sign messages and transactions
- **Contract:** provides a connection to a specific contract on the Ethereum Network





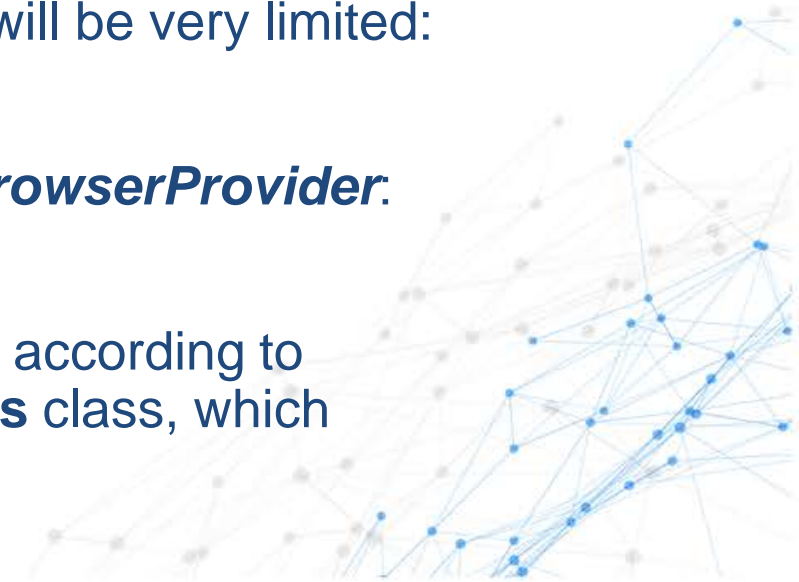




# Ethers.js - Provider

# Ethers.js - Provider

- Enables read-only access the blockchain, a signer provides read/write access
- For a local blockchain like Ganache or Hardhat, use the **JsonRpcProvider**:  
*provider = new ethers.JsonRpcProvider("http://localhost:8545")*
- To connect to a remote blockchain, it is recommended to use a third party provider:  
*provider = new ethers.AlchemyProvider((network = "sepolia"), MYAPIKEY)*
- Instead of using a third party provider, you can also use the **defaultProvider**. Ethers.js offers default API keys for free, however, the number of network requests will be very limited:  
*provider = ethers.getDefaultProvider((network = "sepolia"))*
- For web apps that use Metamask (uses Infura internally), use the **BrowserProvider**:  
*provider = new ethers.BrowserProvider(window.ethereum)*
- In Hardhat, we can configure a list of networks in hardhat.config.js – according to the used network, Hardhat injects the correct provider into the **ethers** class, which can be accessed via: *provider = ethers.provider*



# Ethers.js - Provider

## Properties & methods:

- `provider.getNetwork()` //details about the currently used network
- `provider.getBalance("0x123...")`
- `provider.getTransactionCount("0x123...")`
- `provider.getBlockNumber ()`
- `provider.getGasPrice()` //estimation of gas price in wei
- `provider.getFeeData()` //estimation for `maxFeePerGas`, `maxPriorityFeePerGas`
- Various methods to listen to events and to return logs for a provided filter...





# Ethers.js - Signer

# Ethers.js - Signer

- Abstraction of an Ethereum Account. Can be used to sign and send transactions
- **JsonRpcSigner**: connected to a **JsonRpcProvider** (local node) - acquired via:  
*signer = provider.getSigner()*
- **Wallet**: knows its private key and can sign transactions:  
*signer = new ethers.Wallet( privateKey , provider )*

## Example transaction:

```
txnParams = {  
  to: accountAddress,  
  value: ethers.parseEther("0.1"),  
}  
  
txn = await signer.sendTransaction(txnParams)  
txnReceipt = await txn.wait()
```





# Ethers.js – Example Transaction

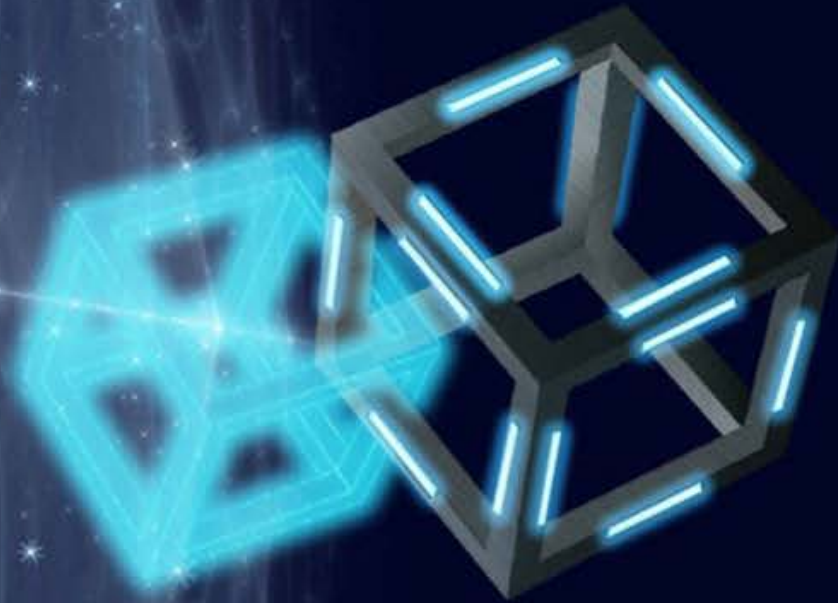
[illegible]

# Ethers.js - Signer

## Properties & methods:

- `signer.getAddress()`
- `signer.getNonce()`
- `signer.estimateGas( transactionRequest )`
- `signer.signTransaction(tx)`
- `signer.sendTransaction(tx)`





# **Ethers.js**

## **Contract, Logs, Events & Filters**

# Ethers.js - Contract

Used to communicate with a deployed contract. This class needs to know what methods are available and it gets this information from the ABI :

```
contract = await ethers.getContractAt("HelloWorld", contractAddress)
```

```
contract = new ethers.Contract(contractAddress, abi, signerOrProvider)
```

## Properties & methods:

- `contract.target`
- `contract.provider`
- `contract.signer`
- `contract.connect(signerOrProvider)` //new contract instance connected to signer or provider
- *`(returnValue1, returnValue22...) = contract.myReadMethod(arg1, arg2...)`*
- *`txn = contract.myWriteMethod(arg1, arg2...)`*





# Ethers.js – Events, Logs & Filters

- Logs and filters allow for efficient queries of indexed data and provide low-cost data storage if the data is not required to be accessed on-chain (cannot be accessed by contract)
- Up to 3 event arguments can be indexed (**Topics**) for efficient filtering when a contract emits an event.

## Accessing / listening to logs from the provider class:

- *provider.getLogs(filter)* //returns an array of logs: data, blockNumber, txnHash...
- *provider.on(filter, (log) => { console.log(log)...})* //fromBlock & toBlock are not used
- Properties of the filter object: **fromBlock** (number or “latest”), **toBlock**, **address**, **topics**



# Ethers.js – Events, Logs & Filters

## Accessing / listening to logs from the contract class:

- *myEvents = await contract.**queryFilter**(filter, "latest", "latest")*
- *contract.on(**filter**, (from, oldMessage, newMessage, event) => { console.log(event)...})*

// returns a list of topics for a specific Event – null means any match  
***filter** = contract.filters.UpdateMessage(someAddress, null, null)*

```
Event topics: {  
  address: '0x5FbDB2315678afecb367f032d93F642f64180aa3',  
  topics: [  
    '0x393bbe2c5115b2370579ad2f520ee8319935cff3b04145a3246b8c8c7730dc73',  
    '0x000000000000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfff92266'  
  ]  
}
```



[illegible][illegible]



**Ethers.js - Utils**



# Ethers.js – Interface

```
iface = new ethers.Interface(["function updateMessage(string newMessage)"])  
iface = new ethers.Interface(contractJson.abi)
```

## Encoding & Decoding:

```
encodedFunction = iface.encodeFunctionData("updateMessage", ["test"]) // hex string
```

```
// decode provided argument values from txn.data => returns: [ 'test', newMessage: 'test' ]
```

```
argValues = iface.decodeFunctionData("updateMessage", txn.data)
```

```
// get the function selector (4 bytes) => calldata for method call
```

```
sigHash = iface.getFunction("updateMessage").selector
```



# Ethers.js – ParseTransaction

*parsedTxn = iface.parseTransaction({ data: txn.data })*

```
Parsed txn: TransactionDescription {  
  args: [ '6051', newMessage: '6051' ],  
  functionFragment: {  
    type: 'function',  
    name: 'updateMessage',  
    constant: false,  
    inputs: [ [ParamType] ],  
    outputs: [],  
    payable: false,  
    stateMutability: 'nonpayable',  
    gas: null,  
    _isFragment: true,  
    constructor: [Function: FunctionFragment] {  
      from: [Function (anonymous)],  
      fromObject: [Function (anonymous)],  
      fromString: [Function (anonymous)],  
      isFunctionFragment: [Function (anonymous)]  
    },  
    format: [Function (anonymous)]  
  },  
  name: 'updateMessage',  
  signature: 'updateMessage(string)',  
  sighash: '0x1923be24',  
  value: BigNumber { value: "0" }  
}
```



# Ethers.js – ParseLog

```
topics = ["0x393...", "0x749..."]
```

```
log = iface.parseLog({ data: txnRec.logs[0].data, topics })
```

```
Parsed log: LogDescription {
  eventFragment: {
    name: 'UpdateMessage',
    anonymous: false,
    inputs: [ [ParamType], [ParamType], [ParamType] ],
    type: 'event',
    _isFragment: true,
    constructor: [Function: EventFragment] {
      from: [Function (anonymous)],
      fromObject: [Function (anonymous)],
      fromString: [Function (anonymous)],
      isEventFragment: [Function (anonymous)]
    },
    format: [Function (anonymous)]
  },
  name: 'UpdateMessage',
  signature: 'UpdateMessage(address,string,string)',
  topic: '0x393bbe2c5115b2370579ad2f520ee8319935cff3b0414',
  args: [
    '0xf39Fd6e51aad88F6F4ce6aB8827279cFfFb92266',
    '5266',
    '1638',
    from: '0xf39Fd6e51aad88F6F4ce6aB8827279cFfFb92266',
    oldStr: '5266',
    newStr: '1638'
  ]
}
```



# Ethers.js – Address & Byte Manipulation

## Address:

- *ethers.getAddress("0x8ba1f...")* // return checksum address
- *ethers.computeAddress( publicOrPrivateKey )*

## Byte manipulation:

// converts DataHexString to a Uint8Array

*ethers.getBytes("0x1234")* // Uint8Array [ 18, 52 ]

// converts a number to Hex

*ethers.toBeHex(1)* // 0x01





# Ethers.js – BigInt

**Allows mathematical operations on numbers of any magnitude.**

- *let n1 = 10n*
- *let n2 = BigInt("0x32") // 50n*
- *let n3 = n1 + 20n // 30n*
- *let n4 = Number(n1) + 20*

## **Conversion:**

- *let myHexString = n1.toString(16) // 0x0a*



# Ethers.js – Display Logic & Input

**formatUnits** formats a **BigInt** to a string, which is useful when displaying a balance:

**ethers.formatUnits**( value [ , unit = "ether" ] ) ⇒ string (in wei)

*ethers.formatUnits(oneGweiBigInt, "gwei") // '1'*

*ethers.formatUnits(oneGweiBigInt, 9) // '1'*

*ethers.formatUnits(oneGweiBigInt, 0) // '10000000000'*

*ethers.formatEther(oneGweiBigInt) // '0.000000001'*

**parseUnits** parses a string representing ether, into a **BigInt** in wei:

**ethers.parseUnits**( value [ , unit = "ether" ] ) ⇒ BigInt (in wei)

*ethers.parseUnits("12", "gwei") // 12000000000n*

*ethers.parseUnits("12", 9) // 12000000000n*

*ethers.parseEther("12") // 12000000000000000000n*



# Ethers.js – Strings & Hashing Algorithms

User provides string data in frontend => convert to bytes32 => send to smart contract  
this is much cheaper than working with strings in smart contracts

- *ethers.encodeBytes32String("hello")* // convert string to a bytes32
- *ethers.decodeBytes32String("0x68656c6c6f000...")* // convert bytes32 to string
- *ethers.toUtf8Bytes("hello")* // convert string to UTF8 byte array [104, 101, ...]
- *ethers..toUtf8String(new UintArray([104, 101, 108, 108, 111]))* // convert UTF8 byte array to string

## Hashing Algorithms:

- *ethers.keccak256("0x1234")* // Keccak256 of hex string : HexString32
- *ethers.id("UpdateMessage(address,string,string)")* // event topic : HexString32



# Ethers.js – Transactions & Constants

**Transaction parameters:** hash, to, from, nonce, data, value, gasLimit ,  
maxFeePerGas, maxPriorityFeePerGas, v, r, s

```
txnParams = {  to: account2Address,  
              value: ethers.utils.parseEther("1.0"),  
              data: encodedFunction }
```

*ethers.Transaction.from(txnParams).unsignedSerialized* // hexString : 0xf88c0a...

*ethers.Transaction.from(txnSerialized)* // { to:... , value:... , data:... }

## Constants:

ethers.ZeroAddress ⇒ Address (20 bytes)

ethersss.WeiPerEther ⇒ BigInt







# Metamask RPC API

# Metamask RPC API

- Metamask uses Infura (third party node provider) to connect to the blockchain
- Metamask communicates with the blockchain (full node) via JSON RPC
- Metamask is compatible with any blockchain that exposes an Ethereum-compatible JSON RPC API (interface between client and full node)
- For example, to retrieve the account balance, MetaMask can make a HTTP request to a full node, invoking the ***eth\_getBalance*** RPC function with my address as a parameter
- A web app can interact with Metamask via the ***window.ethereum*** object. This API allows websites to request user accounts, read data from the blockchains, sign a transactions



# Metamask RPC API

## A DAPP needs to do the following:

- Detect the Ethereum provider injected by Metamask (**window.ethereum**)
- Verify if the DAPP is already connected with one or more accounts managed by MetaMask and return those accounts
- Display all accounts that are managed by MetaMask and return the selected ones
- Detect network and account changes



# Metamask RPC API

Submitting RPC requests to Ethereum via MetaMask:

```
window.ethereum.request(object)
```

Get a list of all MetaMask accounts that are already connected with the DAPP:

```
accounts = await window.ethereum.request({ method: 'eth_Accounts' })
```

Enable a DAPP on Metamask (select account(s) in popup):

```
accounts = await window.ethereum.request({ method: 'eth_requestAccounts' })
```

React if the user changes the network or account:

```
window.ethereum.on('accountsChanged', function (accounts) { // reload page... })
```

```
window.ethereum.on('chainChanged', function (chainId) { // reload page... })
```



# Metamask RPC API – Send Transaction

```
const txnParameters = {
  gasPrice: '0x09184e72a000', // customizable by user during MetaMask confirmation.
  to: '0x123...', // required except during contract publications.
  from: myAddress, // must match user's active address.
  value: '0x0', // only required to send ether
  data: '0x7f746573743200000000000000000000...'
};
```

```
const txHash = await window.ethereum.request({
  method: 'eth_sendTransaction',
  params: [txnParameters],
});
```







# React DAPP Project

# React Web App Using Ethers.js

## Requirements:

- Create a React application (javascript-swc) : `npm create vite@latest app-name`  
=> npm install => npm run dev
- Retrieve accounts from Metamask and allow the app to be connected to one or more accounts: `request({method: "eth_requestAccounts"...`
- Whenever the app starts, verify if there are already any connected accounts: `request({method: "eth_accounts"...`
- Retrieve the current message value and display it in the UI
- Allow the message to be updated from the UI by sending a transaction to Metamask: `request({method: "eth_sendTransaction"...`
- Update the UI when the transaction has been mined and the message value has been updated on the blockchain: `contract.on("UpdateMessage"...`



# React Web App Using Ethers.js

Hello World - React Metamask

Connected: 0xf39f...2266

Current Message:

test message

New Message:

Update the message in your smart contract.

🔔 Your message has been updated!

Update



# React Web App Using Ethers.js - Exercise

## Exercise – interact.js:

- Add the provider
- Export the contract
- Get already connected Metamask accounts => **eth\_accounts** and log a message whenever the network or an account changes => **accountsChanged & chainChanged** in **getCurrentWalletConnected()**
- Get all addresses from metamask => **eth\_requestAccounts** in **connectWallet()**
- Return the initial message from the contract in **loadCurrentMessage()**
- In **updateMessage()** : encode the function we want to call ; set up transaction parameters => transactionParameters : to, from, data ; send the transaction using Metamask => **eth\_sendTransaction** , return txHash



# React Web App Using Ethers.js - Exercise

## Exercise – HelloWorld.js:

- In ***useEffect()*** : load current message ; verify if a Metamask address is already connected with the app and retrieve the address and status
- In ***addSmartContractListener()*** : listen to the ***UpdateMessage*** event => setMessage, setNewMessage & setStatus
- In ***connectWalletPressed()*** : connect to Metamask and update the states
- In ***onUpdatePressed()*** : call ***updateMessage*** and update the state







# ERC20 Tokens & OpenZeppelin

# ERC20 Tokens

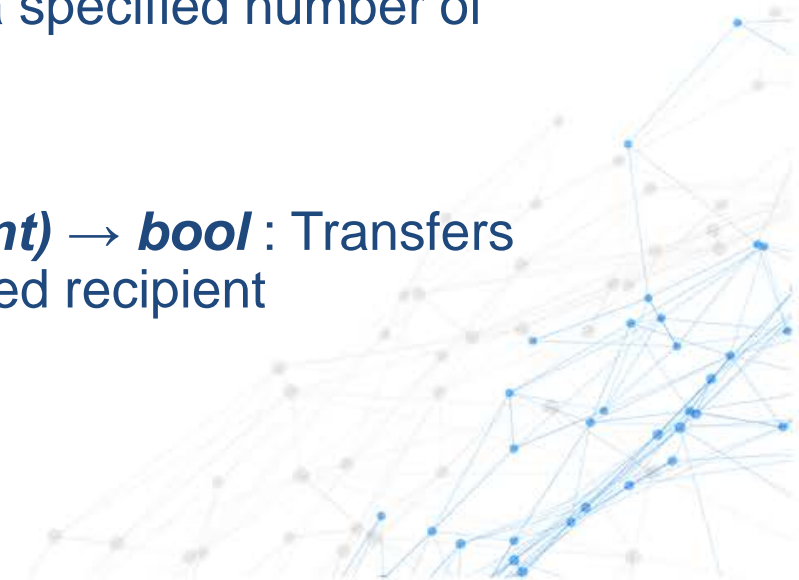
- A Tokens can represent virtually anything in Ethereum: reputation points, skills of a character in a game, financial assets, a fiat currency like USD, an ounce of gold, real estate...
- ERC20 is the technical standard for fungible tokens on the Ethereum blockchain. Fungible means interchangeable or equal
- Use cases of ERC20 tokens: medium of exchange, currency, voting rights, staking...
- An ERC20 tokens is created by deploying a smart contract that adheres to the ERC20 standard
- Examples for ERC20 Tokens: USDC, USDT, BNB, DAI, MAKER



# ERC20 Standard 1/2

## Required functions & events of the ERC20 standard:

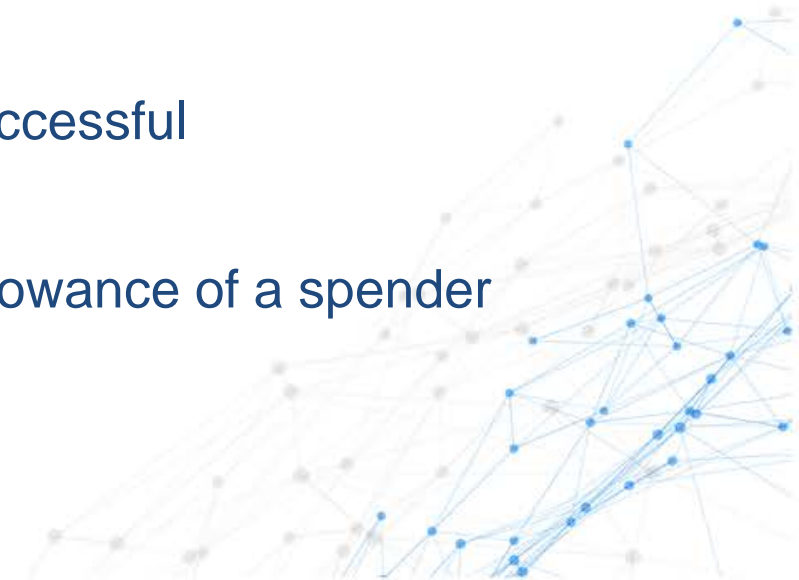
- ***totalSupply()*** → ***uint256*** : The total number of tokens that will ever be issued
- ***balanceOf(address account)*** → ***uint256*** : The account balance of a token owner's account
- ***transfer(address recipient, uint256 amount)*** → ***bool*** : Transfers a specified number of tokens to a specified address
- ***transferFrom(address sender, address recipient, uint256 amount)*** → ***bool*** : Transfers a specified number of tokens from a specified address to the specified recipient



# ERC20 Standard 2/2

## Required functions & events of the ERC20 standard:

- ***approve(address spender, uint256 amount) → bool*** : Allows a spender to withdraw a specified number of tokens from the caller's account
- ***allowance(address owner, address spender) → uint256*** : Returns the remaining number of tokens that spender will be allowed to spend on behalf of owner
- ***Transfer(from, to, value)*** : An event triggered when a transfer is successful
- ***Approval(owner, spender, value)*** : An event triggered when the allowance of a spender is modified by a call to **approve()** - value is the new allowance.



# OpenZeppelin

- A library of modular, reusable, secure smart contracts for the Ethereum network
- Minimize risk by using battle-tested smart contracts and libraries for Ethereum
- Docs: <https://docs.openzeppelin.com/contracts/4.x>
- Code: <https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts>

## Provided contracts:

- Tokens: ERC20, ERC721, ERC1155
- Access control: ownership, role-based access control
- Governance, cryptography, math, security, multicall...



# OpenZeppelin

## Installation:

```
npm install @openzeppelin/contracts
```

## Usage:

```
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MyToken is ERC20 {
    constructor() ERC20("MyToken", "MT") {
        _mint(msg.sender, 100000 * 10**decimals());
    }
}
```







# ERC20 Token Project

# ERC20 Token Project - Requirements

- Create an ERC20 Token with the name: **MyToken** and the symbol: **MT**
- During deployment, mint 100.000 MT's to the contract deployer
- The token must provide the following properties, functions and events:
  - totalSupply()
  - balanceOf(address account)
  - transfer(address recipient, uint256 amount)
  - transferFrom(address sender, address recipient, uint256 amount)
  - approve(address spender, uint256 amount)
  - allowance(address owner, address spender)
  - Transfer(from, to, value) Event
  - Approval(owner, spender, value) Event
- Deploy the contract to the Sepolia testnet
- Import the token into Metamask and send some tokens to a different address



# ERC20 Token Project - Exercise

**Write a script that performs the following tasks on a local Hardhat node:**

- Display the MT balance of the deployer account
- Transfer 1000 MT's from the deployer account to another Hardhat account
- Display the MT balance of the second account
- Allow the second account to spend up to 50 MT's from the deployer account
- Display how many tokens the second account can spend on behalf of the deployer account





# Testing Smart Contracts

# Testing Smart Contracts

Tests are executed on the Hardhat network. Mocha is used as the test runner and Chai as assertion library. Additionally, custom Hardhat-Chai matchers and the Hardhat Network Helper plugins are used.

## Running the tests:

- `npx hardhat test test/my-tests.js ---` or:
- `npx hardhat test ---` or:
- `npx hardhat test --network hardhat`



# Basic Structure of a Test File

```
describe("Contract...", function () {  
  async function myFixture() {  
    ...  
  }  
  
  describe("Test group 1", function () {  
    it("Some description...", async function () {  
      const { var1, var2... } = await loadFixture(myFixture)  
      ...  
    });  
    ...  
    it("other test description...", async function () {  
      ...  
    });  
  }  
  
  describe("Test group 2", function () {  
    it("Some description...", async function () {  
      ...  
    });  
    ...  
  }  
});
```





# Hardhat Chai Matchers

This plugin adds Ethereum-specific capabilities to the Chai assertion library

## Installation:

- `npm install --save-dev @nomicfoundation/hardhat-toolbox`

## Usage:

- Add the following to `hardhat.config.js`:
  - `require("@nomicfoundation/hardhat-toolbox")`
- Add the following to the test file:
  - `const { expect } = require("chai")`
  - `const { anyValue } = require("@nomicfoundation/hardhat-chai-matchers/withArgs")`



# Hardhat Chai Matchers

## Numbers:

- *expect(**await** token1.totalSupply()).to.**eq**(1\_000\_000)*
- *expect(await token2.totalSupply()).to.eq(ethers.parseEther("1"))*
- *expect(await token2.totalSupply()).to.eq(10n \*\* 18n)*

## Reverted transactions (reverted, revertedWith & revertedWithCustomError):

- ***await** expect(token.transfer(address0, 10)).to.be.**reverted***
- *await expect(token.transfer(address0, 10)).to.be.**revertedWith**("some message...")*
- *await expect(myToken.transfer(address0, 10)).to.be.**revertedWithCustomError**(myToken, "InvalidTransferValue").withArgs(0)*

*The first argument must be the contract that defines the error. If the error has arguments, the .withArgs matcher can be added*



# Hardhat Chai Matchers – Events & Balances

## Events:

- *await expect(token.transfer(address, 100)).to.**emit**(myToken, "Transfer").**withArgs**(100)*

## Assert how a transaction affects the ETH balance of a specific address:

- *await expect(mySigner.sendTransaction({ to: receiver, value: 1000 })).  
to.**changeEtherBalance**(mySigner, -1000)*

## Assert how a transaction affects the ERC20 token balance of a specific address:

- *await expect(myToken.transfer(receiver, 1000))  
to.**changeTokenBalance**(myToken, sender, -1000)*



# Hardhat Network Helpers

- Functions for quick and easy interaction with the Hardhat Network
- Facilitates the manipulation of account attributes (balance, nonce...)

## Usage:

```
const { loadFixture, setBalance } = require("@nomicfoundation/hardhat-network-helpers")
```

## Set the balance for the given address:

- *await setBalance(someAddress, 100)*

## Return the timestamp of the latest block:

- *await time.latest()*



# Hardhat Network Helpers

**Mine a new block whose timestamp is newTimestamp:**

- *await helpers.time.increaseTo(newTimestamp)*

**Take a snapshot of the state of the blockchain at the current block:**

- *const snapshot = await takeSnapshot()*

**After doing some changes, you can restore to the state of the snapshot:**

- *await snapshot.restore()*



# Fixtures - loadFixture

The first time ***loadFixture*** is called, the code is executed (and the contract(s) are deployed). In subsequent calls, the initial network state (after contract deployment) is restored from a snapshot. This is much faster than deploying the contracts each time the fixture is executed.

```
describe("MyToken contract", function () {
  async function deployContractFixture() {
    const [deployer, user] = await ethers.getSigners()

    const myTokenContract = await ethers.deployContract("MyToken")
    await myTokenContract.waitForDeployment()

    return { myTokenContract, deployer, user }
  }

  describe("Testing Hardhat Chai Matchers", function () {
    it("Should return correct token balance for user after transfer of 10 MT", async function () {
      const { myTokenContract, user } = await loadFixture(deployContractFixture)
```





# Testing Smart Contracts

## Requirements:

- Create a contract deployment fixture => return the contract and signers for the first 2 accounts
- Transfer 10 ETH to the second account and verify the token balance
- Call *setBalance* and verify the updated token balance
- Transfer to a zero address and verify if the call reverts with the message: ERC20: transfer to the zero address => *to.be.revertedWith(...)*
- Verify if the "transfer" event is emitted after a token transfer - verify also the event arguments => *to.emit(...).withArgs(...)*
- Verify if the token balance changes after a token transfer => *changeTokenBalance*





# **NFT's & ERC721 Tokens**

# What are NFT's - Non-Fungible Token?

- Token with unique properties
- Not interchangeable (contrary to ERC20 tokens)
- Used to represent ownership of unique items
- Can only have one owner at a time
- Each minted token has a unique identifier that is directly linked to one Ethereum address
- Digital asset that can represent all kinds of things: collectibles, game items, art, real estate...
- Creator of an NFT can define the scarcity of the asset by minting a certain number of tokens from the NFT smart contract
- We can create an NFT where only one token is minted as a special rare collectible
- Or, we can mint 1000 “identical” items - may be used as admission tickets to an event => each NFT still has a unique identifier with only one owner



# ERC721 Token Standard

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

Code: <https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts/token/ERC721>

## Methods:

- `balanceOf(owner)`
- `ownerOf(tokenId)`
- `transferFrom(from, to, tokenId)`
- `safeTransferFrom(from, to, tokenId)`
- `approve(to, tokenId)`



# ERC721 Token Standard

## Methods:

- `setApprovalForAll(operator, _approved)`
- `getApproved(tokenId)`
- `isApprovedForAll(owner, operator)`

## *Events:*

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



# ERC721URIStorage & ERC1155

OpenZeppelin provides several specialized ERC721 contracts (extensions) - one of them is the **ERC721URIStorage** contract => allows to provide token properties for each minted token:

- tokenURI(tokenId)
- \_setTokenURI(tokenId, \_tokenURI)

## ERC1155 Token Standard:

- Multi-Token Standard => a single smart contract can represent multiple tokens at once
- Massive gas savings for projects that require multiple tokens. Instead of deploying a new contract for each token type, a single ERC1155 contract can hold all required NFT's
- This can be useful for example for a smart contract that needs to hold various items with specific properties (NFT's) for a game: Sword, shield, knife...





# IPFS & Pinata

## IPFS:

- Decentralized, peer-to-peer file-sharing system that allows to store and access files, websites, applications, and other data.

## Pinata:

- Docs: <https://api.pinata.cloud/pinning/pinJSONToIPFS>
- Media management company for web3 developers (upload & manage data for web3 apps)
- API to interact with IPFS => “pin” different types of data to IPFS
- Data on an IPFS node is cached and periodically deleted (if it is not pinned)
- To prevent an IPFS node from deleting specific data, that data needs to be labeled (pinned)
- Advantages: Speed of data retrieval, uptime of IPFS nodes, almost unlimited space



# Pinata – API Key & File Upload

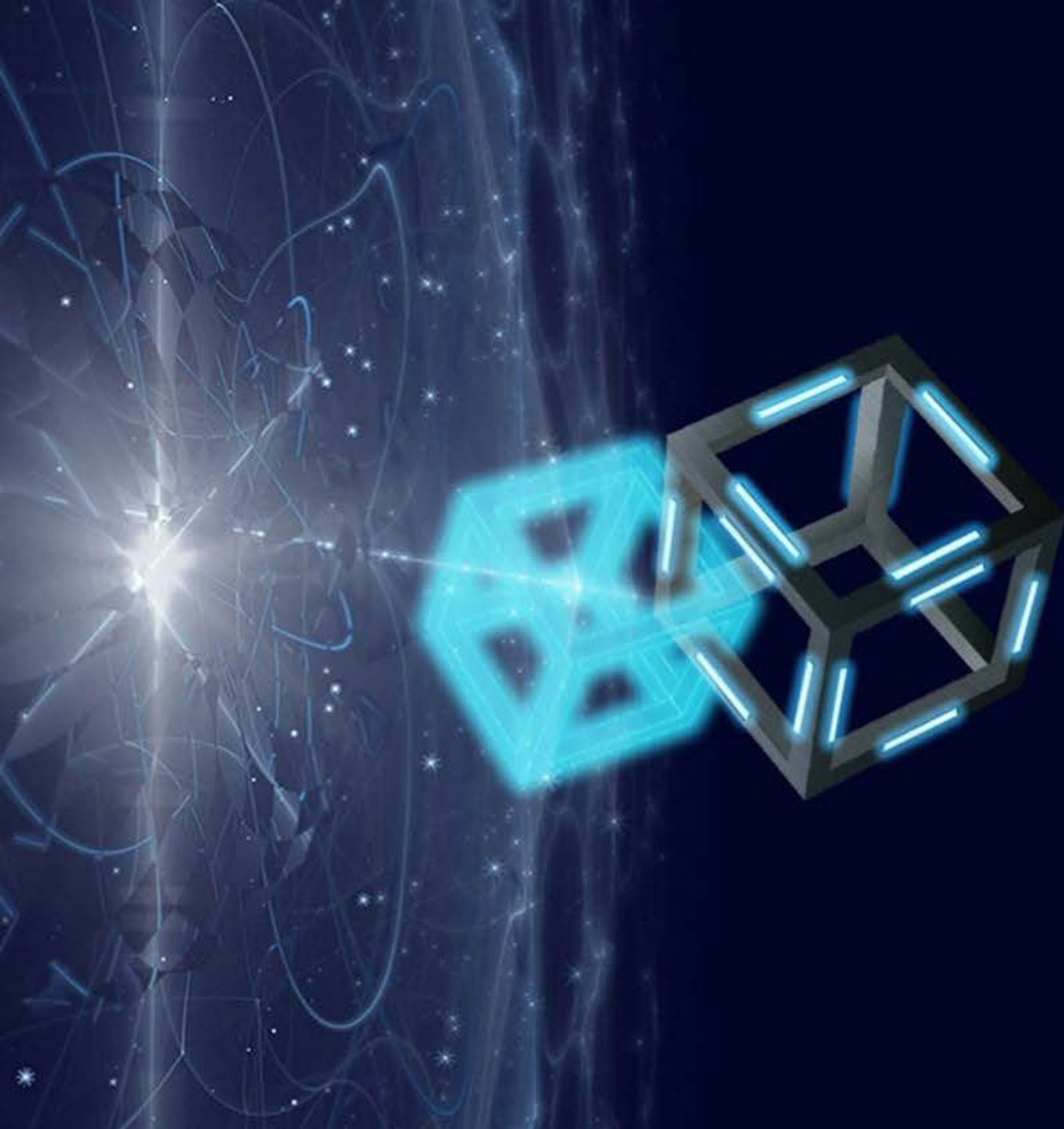
## Pinata API Key:

- Create a Pinata account: <https://app.pinata.cloud/register>
- Click on "**API Keys**" => "**New Key**" => select the "**Admin**" option => click "**Create Key**":
- Copy/paste the API Key and the API Secret into your .env file

## Upload files to Pinata:

- In your Pinata account, click the "**+ Add Files**" button, select "**File**" and upload the cat.jpg file
- Make a copy of the **CID** (the value that starts with Qmc...)
- 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






# NFT Minter Project


# NFT Minter Project - UI

## NFT Minter


Connect Wallet

 Link to asset:

e.g. <https://gateway.pinata.cloud/ipfs/<hash>>


 Name:

e.g. My first NFT!

 Description:

e.g. Even cooler than cryptokitties ;)

Mint NFT

 Connect to Metamask using the top right button.



# NFT Minter Project - Requirements

## ERC721 Token:

- Create an ERC721 token with the name: **MyNFT** and the symbol: **MNFT**
- The token should inherit from the **ERC721URIStorage** contract and only the contract owner should be able to mint NFT's => inherit from the **ownable** contract
- Implement a public **mintNFT** function that allows to associate a metadata file with the NFT and that manages the token Id => increment the Id each time a new NFT is minted and associate the Id with the provided **tokenURI**
- The token must provide the following properties, functions and events:
  - balanceOf(account)
  - ownerOf(tokenId)
  - safeTransferFrom(from, to, tokenId)
  - transferFrom(from, to, tokenId)
  - approve(to, tokenId)
  - Transfer(from, to, tokenId) => Event
  - Approval(owner, approved, tokenId) => Event



# NFT Minter Project - Requirements

## ERC721 Token:

- Deploy the contract to the Sepolia test network
- Import the token into Metamask and send some tokens to a different address
- Upload an image file to IPFS via Pinata
- Create a metadata file for the NFT with 2 properties, a name, a description and a link to the previously uploaded image file. Upload the metadata file to IPFS via Pinata





# NFT Minter Project - Requirements

## Web Application:

- Create a REACT web frontend with the following features:
  - A button to connect the DAPP to Metamask: request({method: "**eth\_requestAccounts**"...
  - Provide various NFT metadata: name, description and a URL to an image file that is stored on IPFS
  - A button to mint an NFT with the provided metadata properties
  - Send the transaction data (minting the NFT) to Metamask: request({method: "**eth\_sendTransaction**"...
- Use Pinata to pin the provided NFT metadata (.json file) on IPFS => will be used as the **tokenURI** in the **mintNFT** smart contract function



# NFT Minter Project - Exercise

**ERC721 Token => open scripts/mint-nft.js & provide the following features:**

- Mint a NFT for a recipient that is not the token owner and provide a link to a metadata file that is hosted on IPFS
- Display the number of NFT's (of type MNFT) owned by the recipient
- Display the owner of the NFT with the Id of 1
- Transfer the NFT from the recipient to the contract owner

**Web Application => open interact.js & provide the following features:**

- In the *mintNFT* function, encode the function data for the "mintNFT" smart contract function
- Create the txn parameters (from, to and encoded function data)
- Send the transaction using Metamask => *eth\_sendTransaction*

