

# Développer et déployer des contrats intelligents sur Ethereum

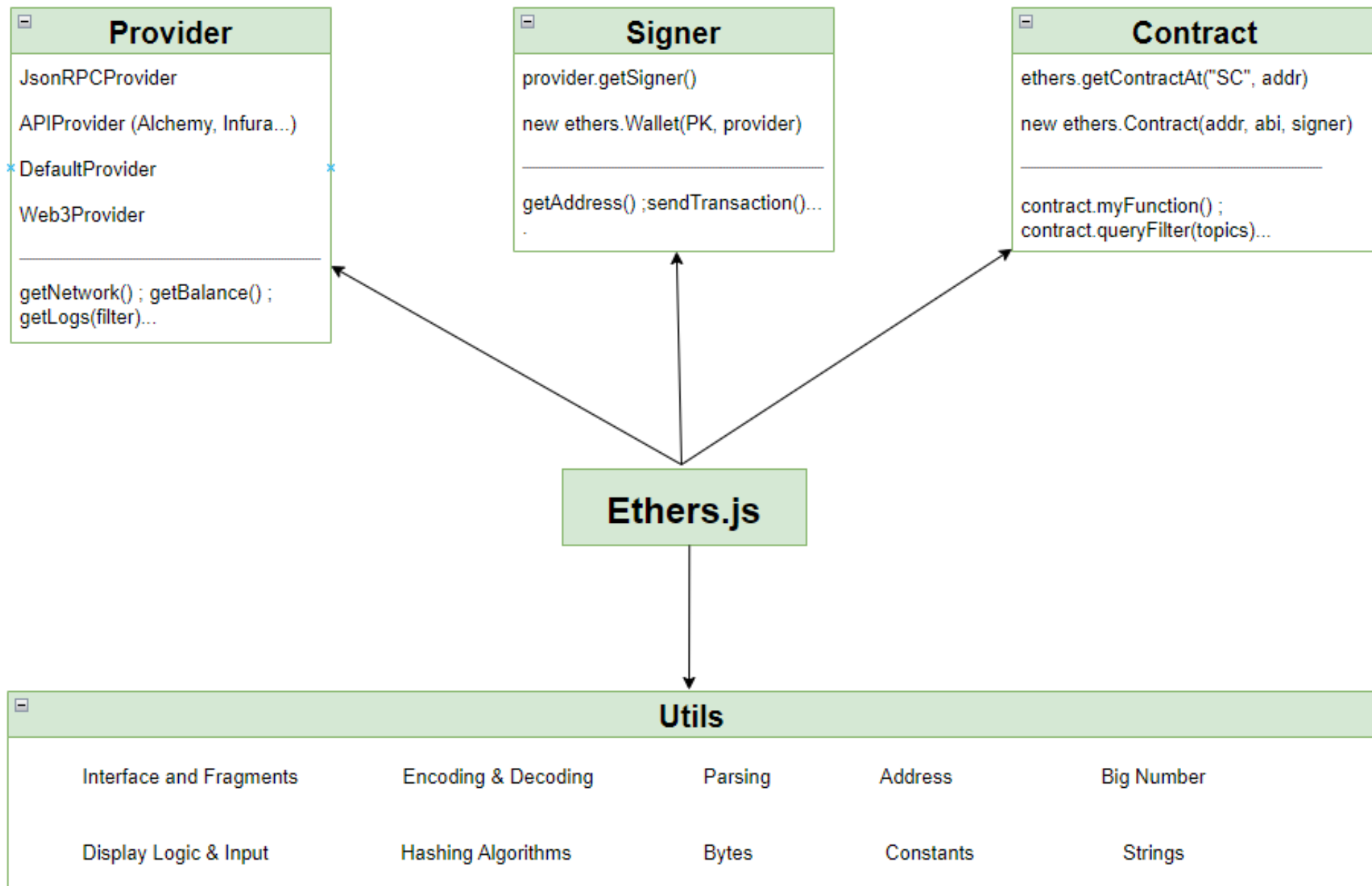


Ethers.js – test des contrats – ERC20  
ERC721 – attaques de réentrée – DAO

# Ethers.js

- Bibliothèque JavaScript pour communiquer avec la blockchain Ethereum
- Facilite les appels des fonctions des contrats intelligents
- Classes les plus importantes : **Provider, Signer et Contract**
- Diverses classes d'utilité (**Utils**) qui facilite l'interaction avec l'ABI, le traitement des types différents : byte, string, adresse et BigNumber...
- **Provider** : Connexion au réseau Ethereum en lecture seule
- **Signer** : Accès au clé privée => peut signer des messages et des transactions
- **Contract** : Abstraction d'un contrat spécifique sur le réseau Ethereum









# Ethers.js - Provider

# Ethers.js - Provider

- Permet un accès à la blockchain en lecture seule, le "**Signer**" permet un accès en lecture/écriture
- Pour un réseau locale (comme Ganache ou Hardhat) on peut utiliser le **JsonRpcProvider**  
*provider = new ethers.JsonRpcProvider("http://localhost:8545")*
- Pour se connecter à une blockchain à distance, il est recommandé d'utiliser un fournisseur tiers :  
*provider = new ethers.AlchemyProvider((network = "sepolia"), MYAPIKEY)*
- Au lieu d'utiliser un fournisseur tiers, on peut aussi utiliser le **defaultProvider**. Des clés API gratuit sont offert, cependant, le nombre de requêtes réseau est très limité :  
*provider = ethers.getDefaultProvider((network = "sepolia"))*
- Pour les applications Web qui utilisent Metamask (utilise Infura en interne), il faut utiliser le **BrowserProvider** : *provider = new ethers.BrowserProvider(window.ethereum)*
- Avec Hardhat, on peut configurer une liste de réseaux dans le fichier hardhat.config.js – selon le réseau utilisé, Hardhat injecte le bon fournisseur dans la classe **ethers**, qui peut être consulté via : *provider = ethers.provider*



# Ethers.js - Provider

## Propriétés et méthodes :

- `provider.getNetwork()` //détails sur le réseau utilisé
- `provider.getBalance("0x123...")`
- `provider.getTransactionCount("0x123...")`
- `provider.getBlockNumber ()`
- `provider.getGasPrice()` // estimation du prix du gaz en wei
- `provider.getFeeData()` // estimations pour `maxFeePerGas` et `maxPriorityFeePerGas`
- Diverses méthodes pour écouter les événements et retourner les logs pour un filtre fourni...





**Ethers.js - Signer**

# Ethers.js - Signer

- Abstraction d'un compte Ethereum. Peut être utilisé pour signer et envoyer des transactions
- **JsonRpcSigner** : connecté à un **JsonRpcProvider** (nœud local) - acquis via :  
*signer = provider.getSigner()*
- **Wallet** : connaît sa clé privée et peut signer des transactions :  
*signer = new ethers.Wallet( privateKey , provider )*

## Exemple envoi d'une transaction :

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





# Ethers.js – reçu d'une Transaction

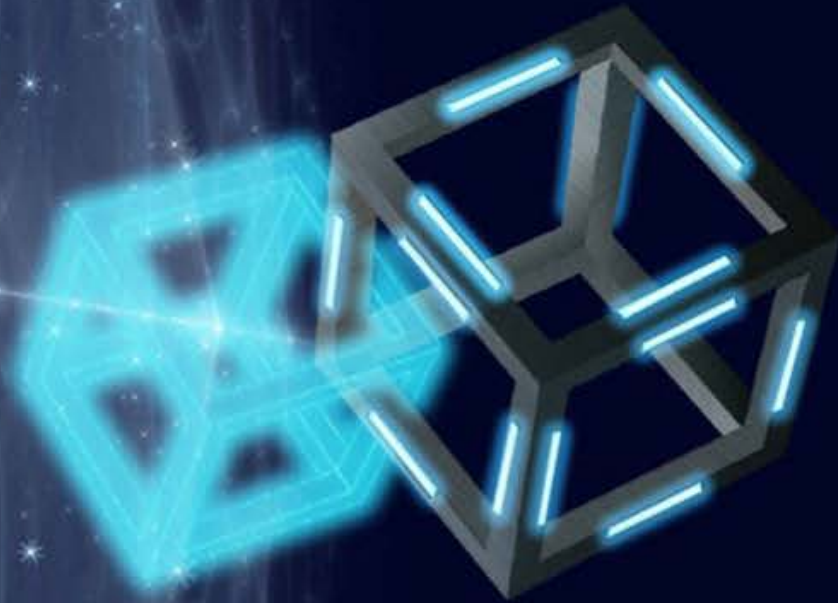
[illegible]

# Ethers.js - Signer

## Propriétés et méthodes :

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





**Ethers.js - contrat,  
logs, événements  
& filtres**

# Ethers.js - Contract

Pour communiquer avec un contrat déployé. Cette classe doit connaître les méthodes disponibles et elle obtient cette information de l'ABI :

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

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

## Propriétés et méthodes :

- `contract.target`
- `contract.provider`
- `contract.signer`
- `contract.connect(signerOrProvider)` // nouvelle instance du contrat connecté  
// avec le signer ou le provider spécifié
- *`(returnValue1, returnValue22...) = contract.myReadMethod(arg1, arg2...)`*
- *`txn = contract.myWriteMethod(arg1, arg2...)`*





# Ethers.js – événements, logs & filtres

- Les logs permettent de stocker des données pas cher
- Jusqu'à 3 paramètres peuvent être indexées (**Topics**) pour un filtrage efficace lorsqu'un contrat émet un événement

## Accès aux logs depuis le "*provider*" :

- *provider.getLogs(filter)* // retourne un tableau de logs : data, blockNumber, txnHash...
- *provider.on(filter, (log) => { console.log(log)...})* // fromBlock & toBlock ne sont pas utilisés
- Propriétés de l'objet filtre : **fromBlock** (numéro ou "latest"), **toBlock**, **address**, **topics**



# Ethers.js – événements, logs & filtres

## Accès aux logs depuis le contrat :

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

// retourne une liste de sujets pour un événement spécifique – null : correspondant à tous  
***filter** = contract.filters.UpdateMessage(someAddress, null, null)*

```
Event topics: {
  address: '0x5FbDB2315678afecb367f032d93F642f64180aa3',
  topics: [
    '0x393bbe2c5115b2370579ad2f520ee8319935cfff3b04145a3246b8c8c7730dc73',
    '0x00000000000000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cffffb92266'
  ]
}
```







# Ethers.js - Utils



# Ethers.js – Interface

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

## Encodage et décodage :

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

```
// décoder les arguments fournies à par txn.data => retourne : [ 'test', newMessage: 'test' ]
```

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

```
// Obtenir le sélecteur de fonction (4 octets) => calldata pour appeler la fonction
```

```
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 – manipulation des adresses et des octets

## Adresse :

- *ethers.getAddress("0x8ba1f...")* // renvoie l'adresse sous forme de somme de contrôle
- *ethers.computeAddress( publicOrPrivateKey )*

## Octets :

// convertir DataHexString en Uint8Array

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

// convertir un nombre en Hex

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





# Ethers.js – BigInt

Permet des opérations mathématiques sur des nombres de toute 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) // 0x10*



# Ethers.js – affichage et données d'entrée

**formatUnits** : formater BigInt en string, utile par exemple pour affiche le solde :

**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** : convertir une chaîne représentant de l'Ether en wei (BigInt) :

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

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

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

*ethers.parseEther("12") // 120000000000000000000n*



# Ethers.js – chaînes et algorithmes de hachage

L'utilisateur fournit des données (chaîne) sur le site web => convertir en bytes32 => envoyer au contrat => beaucoup moins cher que de travailler avec des chaînes dans des contrats intelligents

- *ethers.encodeBytes32String("hello")* // convertir chaîne en bytes32
- *ethers.decodeBytes32String("0x68656c6c6f000...")* // convertir bytes32 en chaîne
- *ethers.toUtf8Bytes("hello")* // convertir chaîne en tableau d'octets (UTF8) [104, 101, ...]
- *ethers.toUtf8String(new Uint8Array([104, 101, 108, 108, 111]))* // convertir tableau d'octets (UTF8) en chaîne

## Algorithmes de hachage :

- *ethers.keccak256("0x1234")* // Keccak256 hex string : HexString32
- *ethers.id("UpdateMessage(address,string,string)")* // topic de l'événement : HexString32



# Ethers.js – transactions & constants

**Paramètres de transaction :** hash, to, from, nonce, data, value, gasLimit ,  
maxFeePerGas, maxPriorityFeePerGas, v, r, s

```
txnParams = {  to: accountAddress,  
               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 utilise Infura (fournisseur de nœuds tiers) pour se connecter à la blockchain
- Metamask communique avec la blockchain (nœud complet) via JSON RPC
- Metamask fonctionne avec toutes les blockchain qui expose un interface JSON RPC API (interface entre client et nœud complet) qui est compatible avec Ethereum
- Par exemple, pour récupérer le solde du compte, MetaMask peut appeler la fonction RPC : ***eth\_getBalance*** avec l'adresse du compte comme paramètre (requête HTTP à un nœud complet)
- Une application web peut communiquer avec Metamask via l'objet ***window.ethereum*** => permet aux sites Web de demander des comptes d'utilisateurs, lire les données sur la blockchain, signer des transactions...



# Metamask RPC API

**Un DAPP doit effectuer les opérations suivantes :**

- Détecter le fournisseur (provider) injecté par Metamask (**window.ethereum**)
- Vérifier si le DAPP est déjà connecté à un ou plusieurs comptes gérés par MetaMask et retourner ces comptes
- Afficher tous les comptes gérés par MetaMask et retourner les comptes sélectionnés
- Détecter les changements de réseau et de compte



# Metamask RPC API

Pour communiquer avec MetaMask depuis une application web :

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

Obtenir une liste de tous les comptes MetaMask déjà connectés au DAPP :

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

Autoriser une DAPP sur MM (sélectionner le ou les comptes dans la fenêtre contextuelle) :

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

Réagir si l'utilisateur change le réseau ou le compte :

```
window.ethereum.on('accountsChanged', (newAccount) => { // reload page... })
```

```
window.ethereum.on('chainChanged', (newChainId) => { // reload page... })
```



# Metamask RPC API – envoyer une transaction

```
const txnParameters = {
  gasPrice: '0x09184e72a000', // personnalisable par l'utilisateur sur Metamask
  to: '0x123...', // requis sauf pour le déploiement d'un contrat
  from: myAddress, // doit correspondre à l'adresse de l'utilisateur
  value: '0x0', // seulement requis pour envoyer de l'Ether
  data: '0x7f746573743200000000000000000000...'
};
```

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







# **Projet DAPP (en React)**

# Application web avec React & 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



# Application web avec React & Ethers.js

## Besoins de projet :

- Créer une application React (javascript-swc) : **`npm create vite@latest app-name`**  
=> npm install => npm run dev
- Quand l'application démarre, vérifiez s'il y a déjà des comptes connectés à l'application : **`request({method: "eth_accounts"...`**
- Récupérer les comptes depuis Metamask et permettre à l'application d'être connecté à un ou plusieurs comptes : **`request({method: "eth_requestAccounts"...`**
- Récupérer la valeur du "message" (depuis le contrat) et l'afficher dans l'interface utilisateur
- Permettre la mise à jour du message depuis l'interface en envoyant une transaction à Metamask : **`request({method: "eth_sendTransaction"...`**
- Mettre à jour l'interface lorsque la transaction a été intégrée dans un nouveau bloc et la valeur du message a été mise à jour sur la blockchain : **`contract.on("UpdateMessage"...`**



# Application web avec React & Ethers.js

## Exercice – interact.js:

- Ajouter le fournisseur (provider)
- Exporter le contrat
- Obtenir des comptes déjà connectés => **eth\_accounts** et enregistrer un message (console.log) chaque fois le réseau ou un compte change => **accountsChanged & chainChanged** dans **getCurrentWalletConnected()**
- Récupérer toutes les adresses de Metamask => **eth\_requestAccounts** dans **connectWallet()**
- Récupérer le message du contrat dans : **loadCurrentMessage()**
- Dans **updateMessage()** : encoder la fonction que nous voulons appeler ; configurer les paramètres de la transaction => transactionParameters : to, from, data ; envoyer la transaction via Metamask => **eth\_sendTransaction** , return txHash



# Application web avec React & Ethers.js

## Exercice – HelloWorld.js:

- Dans ***useEffect()*** : récupérer le message ; vérifier si un compte Metamask est déjà connectée avec l'application => récupérer l'adresse du compte
- Dans ***addSmartContractListener()*** : écouter l'événement : ***UpdateMessage*** => setMessage, setNewMessage & setStatus
- Dans ***connectWalletPressed()*** : connecter à Metamask et mettre à jour le "status"
- Dans ***onUpdatePressed()*** : appeler ***updateMessage*** et mettre à jour le "status"







# Token ERC20 & OpenZeppelin

# Les jetons ERC20

- Un jeton peut représenter pratiquement tout dans Ethereum : points de réputation, compétences d'un personnage dans un jeu, des actifs financiers, une monnaie fiduciaire comme l'USD, une once d'or, de l'immobilier...
- ERC20 est la norme technique pour les jetons fongibles sur la blockchain Ethereum. Fongible signifie interchangeable ou égal
- Cas d'utilisation de jetons ERC20 : moyen de change, devise, droit de vote, jalonnement...
- Un jeton ERC20 est créé en déployant un contrat intelligent qui respecte la norme ERC20
- Exemples des jetons ERC20 : USDC, USDT, BNB, DAI, MAKER



# Norme technique ERC20 1/2

## Fonctions & événements requises pour la norme ERC20 :

- ***totalSupply()*** → ***uint256*** : le nombre total de jetons
- ***balanceOf(address account)*** → ***uint256*** : solde du compte
- ***transfer(address recipient, uint256 amount)*** → ***bool*** : transfert d'un nombre de jetons à l'adresse spécifié
- ***transferFrom(address sender, address recipient, uint256 amount)*** → ***bool*** : transfert d'un nombre de jetons depuis l'adresse de l'émetteur à l'adresse du destinataire



# Norme technique ERC20 2/2

## Fonctions & événements requises pour la norme ERC20 :

- ***approve(address spender, uint256 amount)*** → ***bool*** : permet à un dépensier (spender) de retirer un nombre spécifié de jetons du compte de l'appelant
- ***allowance(address owner, address spender)*** → ***uint256*** : retourne les dépenses (en nombre de jetons) qui son autorisées au nom du propriétaire (owner)
- ***Transfer(from, to, value)*** : événement déclenché après un transfert réussi
- ***Approval(owner, spender, value)*** : événement déclenché lorsque l'allocation d'un dépensier (spender) est modifié par un appel à ***approve(...)***



# OpenZeppelin

- Une bibliothèque de contrats modulaires, réutilisables et sécurisés pour le réseau Ethereum
- Minimiser les risques en utilisant des contrats et bibliothèques bien testés
- Docs: <https://docs.openzeppelin.com/contracts/4.x>
- Code: <https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts>

## Contrats fournis :

- Tokens : ERC20, ERC721, ERC1155
- Contrôle d'accès : propriété, contrôle d'accès basé sur les rôles
- Gouvernance, cryptographie, maths, sécurité, multical...





# OpenZeppelin

## Installation :

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

## Utilistion :

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

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





# Token ERC20 Projet

# Projet jeton ERC20

- Créer un jeton ERC20 avec le nom : *MyToken* et le symbole : *MT*
- Pendant le déploiement, crée (*\_mint*) 100.000 MT's pour le propriétaire du contrat
- Le token doit fournir les propriétés, fonctions et événements suivants :
  - `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
- Déployer le contrat sur le testnet Sepolia
- Importer le jeton dans Metamask et envoyer quelques jetons à une autre adresse

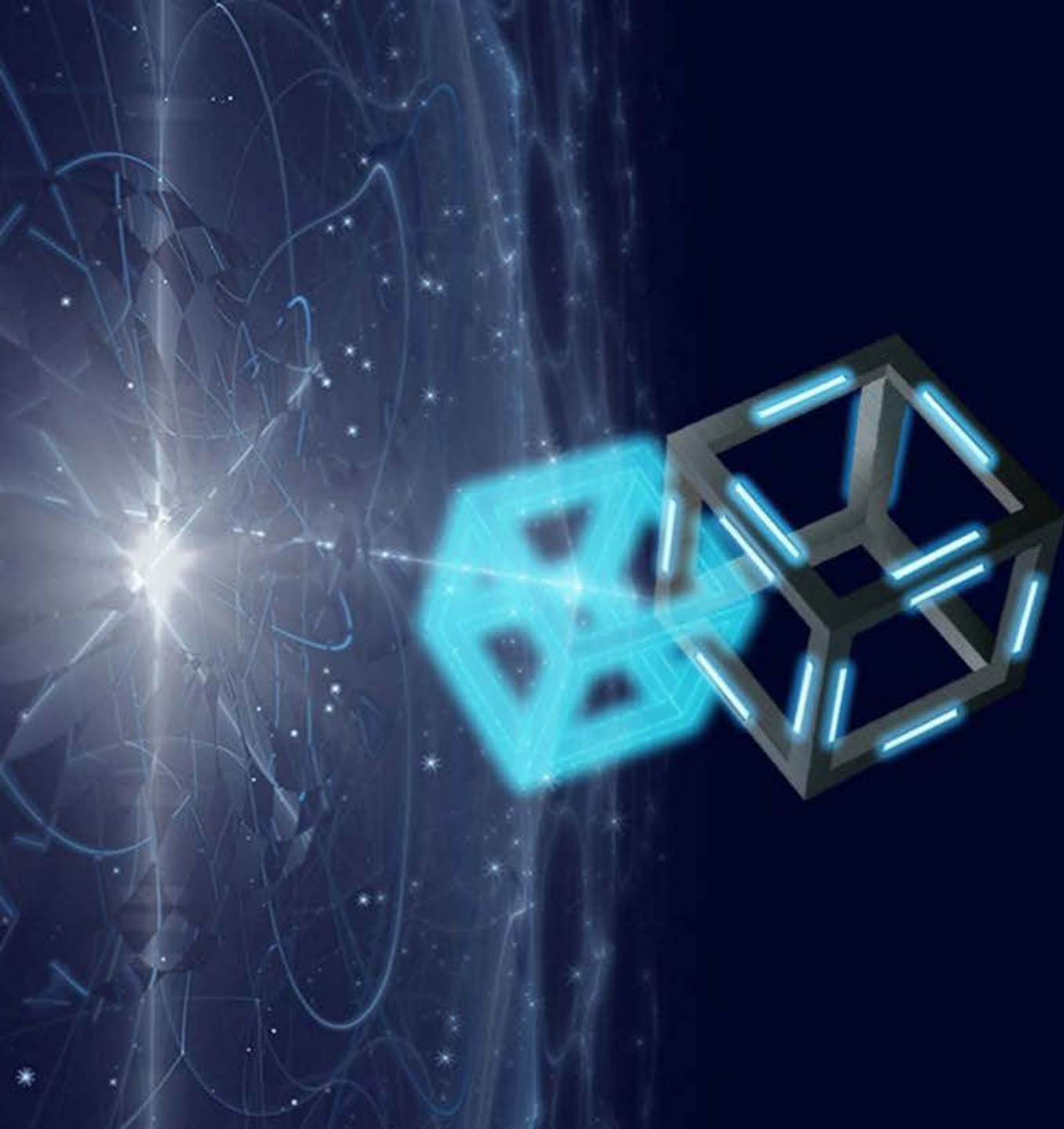


# Projet ERC20 - Exercice

**Ecrire un script qui effectue les tâches suivantes (exécuter en local sur Hardhat) :**

- Afficher le solde des jetons du compte de déploiement
- Transférer 1000 MT's du compte de déploiement à un autre compte Hardhat
- Afficher le solde des jetons de l'autre compte
- Permettre au deuxième compte de dépenser jusqu'à 50 MT à partir du compte de déploiement
- Afficher le nombre de jetons que le deuxième compte peut dépenser au nom du compte de déploiement





**Tester les  
contrats  
intelligents**

# Tester les contrats intelligents

## Les composants utilisée :

- Mocha - framework de test JavaScript
- Chai - bibliothèque d'assertions
- Hardhat-Chai - bibliothèque d'assertions pour Ethereum
- Hardhat Network Helper - fonctions pour une interaction rapide et facile avec Hardhat
- Les tests sont exécutés sur le réseau Hardhat

## Exécuter les tests :

- `npx hardhat test test/my-tests.js`
- `npx hardhat test`
- `npx hardhat coverage`
- `REPORT_GAS=true` (GitBash) ; `$env:REPORT_GAS='true'` (Powershell) & `npx hardhat test`





# Structure d'un fichier de test

```
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

Ce plugin ajoute des capacités spécifiques à Ethereum à la bibliothèque d'assertion Chai :  
<https://www.chaijs.com/api/bdd/>

## Installation :

- `npm install --save-dev @nomicfoundation/hardhat-toolbox` ( déjà installé )

## Utilisation :

- Ajouter au fichier `hardhat.config.js` :
  - `require("@nomicfoundation/hardhat-toolbox")`
- Ajouter au fichier de test :
  - `const { expect } = require("chai")`
  - `const { anyValue } = require("@nomicfoundation/hardhat-chai-matchers/withArgs")`



# Hardhat-Chai

## Nombres :

- `expect(await myToken.totalSupply()).to.eq(1_000_000)`
- `expect(await myToken.totalSupply()).to.eq(ethers.parseEther("1")) //BigInt`
- `expect(await myToken.totalSupply()).to.eq(10n ** 18n)`

## Transactions reversées (reverted, revertedWith & revertedWithCustomError):

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

*Le premier argument doit être l'instance du contrat qui définit l'erreur.  
Si l'erreur a des arguments, **.withArgs** peut être ajouté*



# Hardhat-Chai – événements & soldes

## Événements :

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

## Vérifier le changement du solde d'Ether pour une adresse spécifique :

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

## Vérifier le changement du solde d'un token ERC20 pour une adresse spécifique :

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



# Hardhat Network Helpers

- Interaction rapide et facile avec le réseau Hardhat
- Facilite la manipulation des attributs de compte (solde, nonce...)

## Utilisation :

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

## Modifier le solde pour l'adresse donnée :

- *await setBalance(someAddress, 100)*

## Obtenir le timestamp du dernier bloc :

- *await time.latest()*



# Hardhat Network Helpers

Ajouter un nouveau bloc avec un timestamp spécifique :

- *await time.increaseTo(newTimestamp)*

Prenez un snapshot de l'état de la blockchain au bloc actuel :

- *const snapshot = await takeSnapshot()*

Rétablir l'état de la blockchain a partir d'un snapshot :

- *await snapshot.restore()*





# Fixtures - loadFixture

- La première fois que loadFixture est appelé, le code est exécuté (et les contrats sont déployés)
- Dans les appels suivants, l'état du réseau (après le déploiement du contrat) est restauré à partir d'un snapshot
- C'est beaucoup plus rapide que de déployer les contrats chaque fois quand la fixture est exécutée

```
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)
```

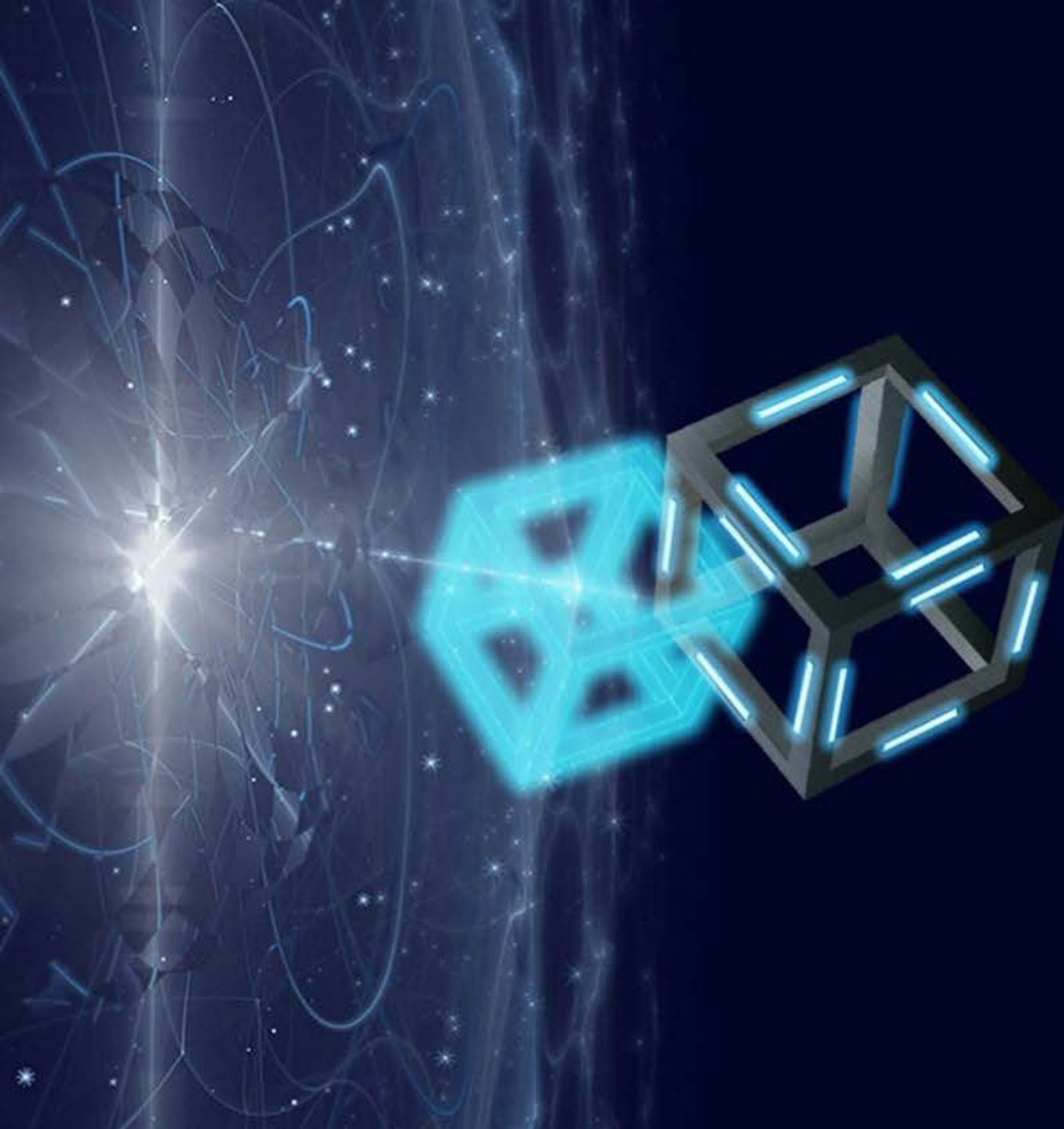


# Tester des contrats intelligents

## Besoins du projet :

- Créer une **fixture** pour le déploiement de contrat => retourner le contrat et les signataires (signer) pour les 2 premiers comptes
- Transférer 10 ETH au deuxième compte et vérifier le solde du jeton
- Appeler **setBalance** et vérifier le solde du jeton
- Transférer à une adresse zéro et vérifier si l'appel revient avec le message : "ERC20: transfer to the zero address" => **to.be.revertedWith(...)**
- Vérifier si l'événement "Transfer" est émis après un transfert de jeton - vérifier aussi les arguments d'événement => **to.emit(...).withArgs(...)**
- Vérifier si le solde du token change après un transfert de token  
=> **changeTokenBalance**





# NFT's & ERC721

# NFT – jeton non-fongible

- Jeton avec propriétés uniques
- Non interchangeables (contraire au jetons ERC20)
- Chaque jeton a un identifiant unique qui est directement lié à une adresse Ethereum
- Actif numérique qui peut représenter toutes sortes de choses : objets de collection, caractère d'un jeu, de l'art, immobilier...
- Le créateur d'un NFT peut définir la rareté de l'actif en créent un certain nombre de jetons depuis le contrat NFT
- On peut décider de créer un seul jeton depuis le contrat NFT - comme une collection rare spéciale
- Ou, on peut créer 1000 jetons "identiques", qui peuvent être utilisés comme billets d'entrée pour un événement => chaque NFT a toujours un identifiant unique avec un seul propriétaire





# NFT – CryptoPunks



# Norme technique ERC721

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

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

## Fonctions :

- `balanceOf(owner)`
- `ownerOf(tokenId)`
- `transferFrom(from, to, tokenId)`
- `safeTransferFrom(from, to, tokenId)` // pour éviter de perdre le jeton
- `approve(to, tokenId)` // permission pour transférer le jeton avec tokenId





# Norme technique ERC721

## Fonctions :

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

## *Evénements :*

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



# ERC721URIStorage & ERC1155

OpenZeppelin fournit plusieurs contrats ERC721 spécialisés (extensions) - l'un d'eux est le contrat **ERC721URIStorage** => permet de lier des propriétés spécifiques à chaque jeton :

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

## Norme ERC1155 :

- Norme multi-jeton => un seul contrat peut représenter plusieurs jetons à la fois
- Grand économies de gaz pour les projets qui nécessitent plusieurs jetons. Au lieu de déployer un nouveau contrat pour chaque NFT, un seul contrat ERC1155 peut contenir tous les NFT requis
- Cela peut être utile pour un contrat qui doit contenir divers éléments avec des propriétés spécifiques - par exemple pour un jeu : épée, bouclier, couteau.....



# IPFS & Pinata

## IPFS :

- Système décentralisé de partage de fichiers P2P qui permet de stocker et d'accéder aux fichiers, sites Web et autres données...

## Pinata - <https://docs.pinata.cloud/introduction> :

- Société de gestion des médias pour les développeurs web3 qui permet de télécharger et gérer les données pour les applications web3)
- API pour communiquer avec IPFS => épingler (pin) différents types de données vers IPFS
- Les données d'un nœud IPFS sont mises en cache et supprimées périodiquement (si elles ne sont pas épinglées)
- Pour empêcher un nœud IPFS de supprimer des données, elles doivent être "épinglées"
- Avantages : rapidité de récupération des données, disponibilité des nœuds IPFS, espace presque illimité



# Pinata – clé API & transfert de fichiers

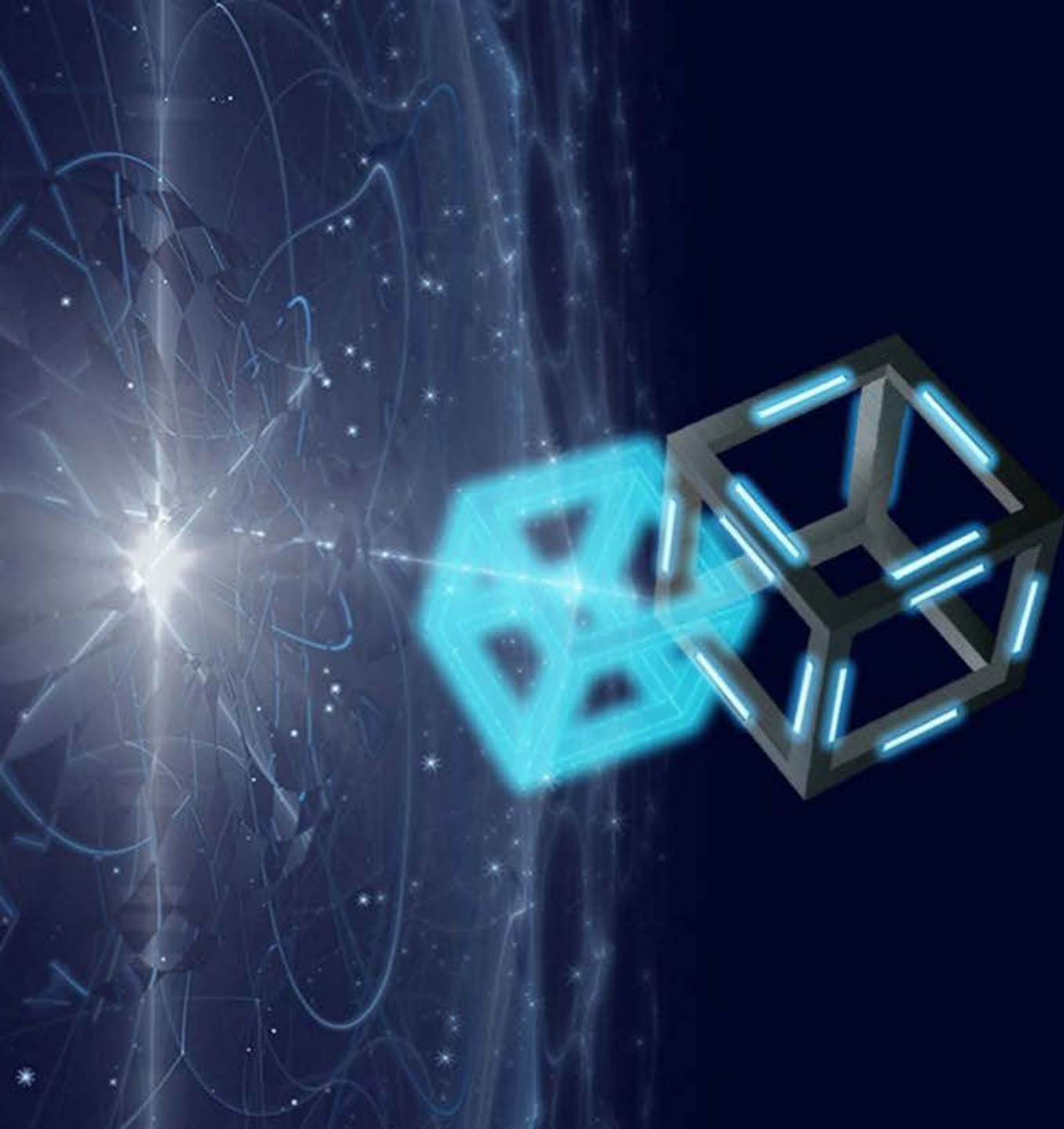
## Clé API Pinata :

- Créer un compte Pinata : <https://app.pinata.cloud/register>
- Cliquer sur "**API Keys**" => "**New Key**" => sélectionner l'option "**Admin**" => cliquer sur "**Create Key**"
- Copier/coller la clé API et le secret API dans le fichier env.

## Transferer les fichiers sur Pinata :

- Sur Pinata, cliquer sur le bouton "**+ Upload**", sélectionner "**File**" et transférer le fichier cat.jpg
- Faire une copie du **CID** (la valeur qui commence par Qmc...)
- Ouvrir le fichier "nft-metadata.json", remplacez la partie CID de l'URL de l'image par votre propre CID et transférer le .json modifié en utilisant Pinata






# Projet NFT


# Projet NFT - 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.





# Project NFT – besoins

## Jeton ERC721 :

- Créer un jeton ERC721 avec le nom : **MyNFT** et le symbole : **MNFT**
- Le token doit hériter du contrat **ERC721URIStorage** et uniquement le propriétaire du contrat a les droit de créer des NFT => hériter du contrat **ownable**
- Créer une fonction publique : **mintNFT** qui permet d'associer un fichier de métadonnées au NFT et qui gère l'ID du jeton => incrémenter l'ID chaque fois qu'un nouveau NFT est crée et associer l'ID au **tokenURI** fourni
- Le token doit fournir les propriétés, fonctions et événements suivants :
  - 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



# Project NFT - besoins

## JettonERC721 :

- Déployer le contrat sur le testnet Sepolia
- Afficher les jetons dans Metamask et envoyer un jeton à une autre adresse
- Stocker un fichier image sur IPFS via Pinata
- Créez un fichier de métadonnées pour le NFT avec 2 propriétés, un nom, une description et un lien vers le fichier d'image. Stocker le fichier de métadonnées dans IPFS via Pinata



# Projet NFT - exercice

**Jeton ERC721 => ouvrir scripts/mint-nft.js & fournir les fonctionnalités suivantes :**

- Créer un NFT pour un destinataire qui n'est pas le titulaire du jeton et fournir un lien vers le fichier des métadonnées stocké sur IPFS
- Afficher le nombre de NFT (MNFT) appartenant au bénéficiaire
- Afficher le propriétaire du NFT avec l'ID de 1
- Transférer le NFT du bénéficiaire au titulaire du contrat

**Application Web => ouvrir interact.js & fournir les fonctionnalités suivantes :**

- Dans la fonction *mintNFT*, encoder les données de la fonction "mintNFT" du contrat
- Créer un objet avec les paramètres de la txn (from, to et les données de la fonction encodées)
- Envoyer la transaction en utilisant Metamask => *eth\_sendTransaction*



# Projet NFT - besoins

## Application Web:

- Créez une interface web en REACT avec les fonctionnalités suivantes :
  - Un bouton pour connecter le DAPP à Metamask : `request({method: "eth_requestAccounts"...`
  - Fournir les métadonnées du NFT : nom, description et adresse URL d'un fichier image sur IPFS
  - Un bouton pour créer un NFT avec les propriétés des métadonnées fournies
  - Envoyer les données de transaction pour créer le NFT à Metamask :  
`request({method: "eth_sendTransaction"...`
- Utilisez Pinata pour épingler les métadonnées (fichier .json) sur IPFS => sera utilisé comme **tokenURI** dans la fonction **mintNFT** du contrat

