Le langage Solidity

Smart-Contracts

Les **smart contracts** sont des programmes ou des contrats auto-exécutants codés sur une blockchain.

Ce sont des règles numériques qui s'exécutent automatiquement lorsqu'une condition spécifiée est remplie, sans nécessiter l'intervention d'un tiers de confiance.

Contrairement à un contrat traditionnel (papier ou numérique), un smart contract est **immuable**, ce qui signifie qu'une fois écrit et déployé sur la blockchain, il ne peut plus être modifié.

Un smart contract permet d'automatiser des actions comme des

- transferts d'actifs
- accords commerciaux
- systèmes de vote
- des jeux

tout cela de manière transparente et fiable.

Exemple : Un smart contract peut être utilisé pour une **transaction d'échange** où une partie envoie un actif à une autre uniquement si certaines conditions sont remplies, comme l'atteinte d'une date limite ou la vérification de l'identité d'une personne.

Introduction à Solidity

- **Langage de programmation** utilisé pour écrire des smart contracts sur Ethereum.
- **Langage statiquement typé** inspiré de JavaScript, Python et C++.
- **Déployé sur Ethereum** pour automatiser des transactions et interactions sur la blockchain.

La documentation est disponible à l'adresse suivante :

https://docs.soliditylang.org/fr/latest/

Principes de base de Solidity

- **Smart contracts** : Programmes auto-exécutants, sans besoin d'autorisation d'un tiers.

- **Ethereum Virtual Machine (EVM)** : Environnement d'exécution des smart contracts sur la blockchain Ethereum.

- **Gas** : Coût des opérations sur la blockchain, mesuré en « gas ».

- **Variables d'état** : Stockage permanent de données sur la blockchain.

- **Fonctions** : Actions définies dans le contrat, publiques ou privées.

- **Modificateurs** : Logique d'accès et de contrôle avant/pendant/après l'exécution des fonctions.

Premier contrat en Solidity

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >= 0.8.0;

contract Registre{
    uint v;
    event ChangedValue(uint value);

    constructor (uint _v){    infinite gas 75600 gas
        v = _v;
    }
    function get() public view returns (uint) {    2432 gas
        return v;
    }

    function set(uint _v) public {    infinite gas
        v = _v;
        emit ChangedValue(_v);
    }
```

Déclaration du compilateur

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >= 0.8.0;
```

- Le commentaire // SPDX-License-Identifier: GPL-3.0

- La commande pragma solidity >= 0.8.0;spécifie que le code est sous la licence **GPL version 3.0**, une licence open-source qui permet la modification et la redistribution sous certaines conditions. Cela permet d'indiquer clairement la licence du code, assurant ainsi la conformité juridique et facilitant la gestion des droits d'utilisation, de modification et de redistribution.

Définition du contrat

```
contract Registre{
```

- Un smart contract est défini à l'aide du mot-clé contract, suivi du nom du contrat.
- Le contrat contient toutes les **variables** et **fonctions** qui le composent.

Variables d’état

```
uint v;
```

Ces variables sont stockées de manière permanente sur la blockchain. Leur état persiste entre les appels de fonction.

Exemples de types:

- **uint**, **int** : Entiers, positifs ou négatifs.
- **address** : Adresses Ethereum.
- **bool** : Valeur booléenne (true/false).
- **string** : Chaînes de caractères.
- **bytes** : Données binaires.
- **mapping** : Association clé-valeur.
- **enum** : Types personnalisés avec un ensemble de valeurs.
- **struct** : Types complexes regroupant plusieurs variables.

Constructeurs

```
constructor (uint _v){
    v = _v;
}
```

Le **constructeur** est une fonction spéciale qui est exécutée une seule fois lors du déploiement du contrat.

Elle permet d'initialiser les variables d'état du contrat avec des valeurs définies au moment du déploiement.

Ici, **_v** est un paramètre d'entrée passé lors du déploiement, et **v** est initialisée avec cette valeur.

Ces variables sont stockées de manière permanente sur la blockchain. Leur état persiste entre les appels de fonction.

Fonctions

```
function set(uint x) public {
    v = x;
}
```

Les **fonctions** dans un smart contract définissent les actions que le contrat peut effectuer.

La visibilité d'une fonction détermine qui peut l'appeler :

- **public** : accessible de l'extérieur du contrat.
- **internal** : accessible uniquement à l'intérieur du contrat et par les contrats qui en héritent.
- **private** : accessible uniquement à l'intérieur du contrat.
- **external** : accessible uniquement de l'extérieur du contrat.

Par exemple, la fonction **set** est publique et elle permet de modifier la valeur de la variable d'état **v**

Évènements

```
event ChangedValue(uint value);
```

Les **événements** permettent aux smart contracts d'**émettre des logs** qui peuvent être écoutés par des applications externes ou des interfaces utilisateurs.

Les événements sont utilisés pour **notifier** des actions importantes, comme des modifications de données ou l'exécution de certaines fonctions.

```
emit ChangedValue(_v);
```

Lorsqu’un événement est déclenché avec la commande **emit** dans une fonction, il envoie un log dans la blockchain qui peut être récupéré via des interfaces comme web3.js.

Variables locales et globales

Local Variables : Variables temporaires utilisées uniquement dans une fonction et effacées après l'exécution.

Global Variables : Des variables intégrées qui donnent des informations sur l'état de la blockchain, comme msg.sender (adresse de l'appelant) ou block.timestamp (timestamp du bloc).

Fonctions view

```
function get() public view returns (uint)
    return v;
}
```

Une **fonction view** permet de **lire** des données sans modifier l'état du contrat.

- **Sans gas** : Pas de frais lors de l'appel de la fonction (si elle est appelée de l'extérieur)
- N'engendre **pas de transaction** dans la blockchain

Utilisation typique : Lire des variables d'état, retourner des informations sans effets sur la blockchain.

Exemple : La fonction **get** permet de lire la variable v mais ne modifie pas l'état de la blockchain.Elle est marquée comme view pour indiquer qu'elle n'interagit pas avec l'état du contrat.

Gas

Le **gas** est l'unité de mesure du travail effectué pour exécuter des opérations sur la blockchain. Il permet de **protéger la blockchain** contre les abus et garantit que les ressources nécessaires à l'exécution des contrats sont utilisées de manière équitable

- Chaque instruction dans un smart contract consomme une quantité de gas, et le gas doit être payé en Ether (ETH) pour effectuer des transactions.

- Le coût en gas dépend de la complexité de l'opération. Les **fonctions simples** consommeront moins de gas, tandis que des calculs ou des boucles complexes en consommeront plus.

Gas Fees (Frais de Gas) : Ce sont des frais payés pour exécuter une transaction ou une fonction sur la blockchain. Ces frais sont payés en **Ether (ETH)**.

Gas Limit : C'est le nombre maximal de gas qu'un utilisateur est prêt à payer pour une transaction. Si le gas nécessaire dépasse le gas limit, la transaction échoue.

Gas Price : C'est le prix que l'utilisateur est prêt à payer par unité de gas, exprimé en **gwei** (1 gwei = 10^{-9} ETH).

La plateforme Remix IDE

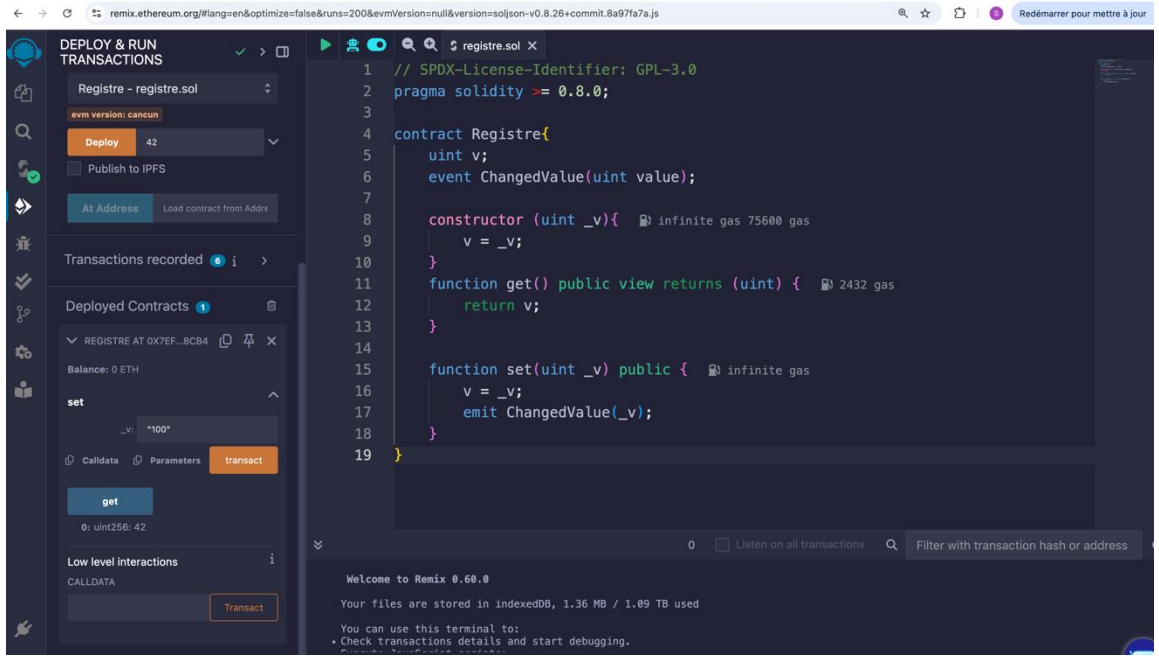
Introduction à Remix IDE

Remix IDE est une plateforme de développement en ligne pour écrire, tester et déployer des smart contracts Solidity.

Il fournit un environnement complet pour interagir avec la blockchain Ethereum sans avoir à installer des outils locaux complexes.

Accès via navigateur : Aucun téléchargement nécessaire, tout se fait directement depuis le navigateur web.

https://remix.ethereum.org



Fonctionnalités de Remix IDE

- **Éditeur Solidity** : Interface pour écrire du code Solidity avec colorisation syntaxique et autocomplétion.
- **Compilation** : Compile le code Solidity directement dans le navigateur.
- **Déploiement** : Permet de déployer des contrats sur différentes chaînes Ethereum (ex : réseau local, testnets, mainnet).
- **Interaction avec les contrats** : Interface graphique pour interagir avec les smart contracts déployés (appels de fonctions, gestion des transactions).
- **Débogage et tests** : Outils de débogage pour inspecter les erreurs et tester les contrats avec différentes données.

La bibliothèque web3.py

Introduction à web3.py

La bibliothèque **web3.py** est une bibliothèque Python permettant de **communiquer avec la blockchain Ethereum**.

Elle permet d’interagir avec des contrats intelligents, de gérer des transactions, de récupérer des informations sur la blockchain (blocs, transactions, etc.), et d'utiliser des nœuds Ethereum.

Web3.py est très utile pour développer des applications décentralisées (DApps) avec un back-end Python.

https://web3py.readthedocs.io/en/stable/

Principales fonctionnalités de web3.py

- **Connexion aux nœuds Ethereum** : Se connecter à un nœud Ethereum via HTTP, WebSocket ou IPC.
- **Gestion des comptes** : Créer des comptes, gérer des clés privées et publiques.
- **Interaction avec les smart contracts** : Déployer, appeler des fonctions, envoyer des transactions à des contrats Ethereum.
- **Récupération d'informations** : Obtenir des données sur des blocs, transactions, et événements de la blockchain.

Connexion à un nœud Ethereum

Connexion à un nœud via HTTP : La façon la plus simple de se connecter à un nœud Ethereum est via une URL HTTP d'un fournisseur de nœuds, comme ankr, Infura ou Alchemy.

```
from web3 import Web3

url = 'https://rpc.ankr.com/eth_sepolia'
w3 = Web3(Web3.HTTPProvider(url))
```

On peut vérifier que la connexion a été établie de la manière suivante :

```
print(w3.is_connected())
```

Comptes Ethereum

On peut manipuler directement un compte Ethereum (signer des transactions, envoyer des transactions, etc.) à l'aide de ses clés (publique et privée)

```
mywallet ='0x75a60008F3235aBD91a6D269d1b98a22d1CfcbdE'
bal = w3.eth.get_balance(mywallet)
myprivkey = '19e6f40fe49f31c8205a83faabec03ca05cfe16a9ab952ac370e31b9323f935f'
```

Transactions

Pour envoyer des ETH à un autre compte, il faut créer une transaction de la manière suivante:

```
copain = '0x8BC69634fdFC3cd76bB4Dd433698370A0359449D'

tx = {
    'to': copain,
    'value': w3.to_wei(0.001, 'ether'),
    'gas': 21000,
    'gasPrice': w3.to_wei('20', 'gwei'),
    'nonce': w3.eth.get_transaction_count(mywallet),
    'chainId': w3.eth.chain_id
}
```

Pour il faut signer la transaction et l'envoyer.

```
stx = w3.eth.account.sign_transaction(tx, myprivkey)
htx = w3.eth.send_raw_transaction(stx.raw_transaction)
w3.eth.wait_for_transaction_receipt(htx)
```

Instance de smart-contract

Pour interagir avec un smart contract, il faut tout d'abord disposer de l'adresse du contrat et de son ABI (Application Binary Interface). Cette dernière est disponible sous Remix IDE.

```
contract_address = '0x0082C17C8A9449aa6D0f8DC7C55342401b15BDe1'
contract_abi = [
    {
        'name' : 'get',
        'type': 'function',
        'inputs' : [],
        'outputs' : [{'type':'uint256'}],
        'stateMutability': 'view' }
]
```

Il faut ensuite créer une instance du contrat à l'aide de la commande suivante

```
sepolia_contract = w3.eth.contract(address=contract_address, abi=contract_abi)
```

Interagir avec smart-contract : fonctions de lecture

Pour interagir avec une fonction de lecture (**view**) d'un smart contract, il faut simplement appeler la méthode **call** associée à l'objet correspondant à cette fonction.

```
sepolia_contract.functions.get().call()
```

Comme la fonction **get** est une fonction view, elle **ne modifie pas l'état** du contrat et **ne nécessite pas de frais de gas**.

Interagir avec smart-contract : fonctions d'écriture (1/2)

L'interaction avec une fonction d'écriture nécessite une **transaction**.

Il faut commencer par étendre l'ABI avec la signature de la fonction d'écriture

```
contract_abi = [
    ...
    { 'name' : 'set',
      'type': 'function',
      'inputs' : [{'name':'_v','type':'uint256'}],
      'outputs' : [],
      'stateMutability': 'nonpayable'}
]
```

Sans oublier de re-déclarer une instance du contrat

```
sepolia_contract = w3.eth.contract(address=contract_address, abi=contract_abi)
```

Interagir avec smart-contract : fonctions d'écriture (2/2)

Il faut ensuite construire une transaction de la manière suivante

```
tx = sepolia_contract.functions.set(100).build_transaction({
    'chainId': w3.eth.chain_id,
    'gas': 500000,
    'nonce': w3.eth.get_transaction_count(mywallet),
    'maxPriorityFeePerGas': w3.to_wei('3', 'gwei'),
    'maxFeePerGas': w3.to_wei('20', 'gwei')
})
```

Il faut ensuite signer la transaction puis l'envoyer

```
stx = w3.eth.account.sign_transaction(tx, myprivkey)
htx = w3.eth.send_raw_transaction(stx.raw_transaction)
```

Sites Web utiles

Faucet Sepolia de Google

https://cloud.google.com/application/web3/faucet/ethereum/sepolia

Sepolia **Etherscan**

https://sepolia.etherscan.io/

Sepolia transaction explorer

https://sepolia.ethernow.xyz/mempool/all

Projet

Le jeu Devine Nombre

Le jeu Devine Nombre est un jeu à deux joueurs A et B : le joueur A choisit un nombre et le joueur B essaie de le deviner. À chaque tour, B propose un nombre *n* et A lui répond si *n* est égal, plus grand ou plus petit que son nombre mystère. Le nombre de tour est bien sûr limité.

Afin d'éviter la triche, on souhaite programmer ce jeu en utilisant un smart-contract déployé sur la blockchain Ethereum.

Implémentez ce jeu comme une Dapp en Python et Solidity, à l'aide de web3.py et Remix.

On pourra développer plusieurs versions successives du jeu, selon le niveau de sécurité/jouabilité.

Le jeu Devine Nombre

Version 1 : Jeu de base (version simple)

Dans cette version de base, le jeu consiste à deviner un nombre choisi par le joueur A. Le nombre de tentatives est limité, et le joueur B doit proposer des nombres successifs. Le smart-contract vérifie à chaque tour si la proposition de B est plus grande, plus petite ou égale au nombre mystère choisi par A.

Règles :

- **Joueur A :** Choisit un nombre secret entre 1 et 100 et déploie un smart contract dédié pour jouer.
- **Joueur B :** Devine le nombre mystère, en proposant un nombre à chaque tour.
- **Réponse du smart contract :** À chaque proposition de B, le smart contract renvoie si le nombre proposé est "plus petit", "plus grand", ou "correct".
- **Limitation des tentatives :** Le nombre de tentatives est limité à un maximum (par exemple, 10 essais).
- **Fin du jeu :** Le jeu se termine dès que le joueur B devine le bon nombre ou que le nombre de tentatives est épuisé.

Le jeu Devine Nombre

Version 2 : Récompenses

Dans cette version, les joueurs misent de l'ETH pour participer au jeu. Le joueur qui trouve le bon nombre gagne l'ETH misé par les deux joueurs. Cela ajoute une dimension économique et compétitive au jeu.

Règles :

- **Joueur A :** Choisit un nombre secret entre 1 et 100 et déploie un contract dédié. A met une mise d'ETH pour participer dans ce contrat.
- **Joueur B :** Devine le nombre mystère en proposant des nombres à chaque tour. B met également une mise d'ETH.
- **Réponse du smart contract :** À chaque proposition, le smart contract renvoie si le nombre est plus petit, plus grand ou égal. Si B trouve le bon nombre, il gagne la mise totale (mise de A et B).
- **Limitation des tentatives :** Le nombre de tentatives est limité (par exemple, 10 essais), comme dans la version simple.
- **Fin du jeu :** Le jeu se termine lorsque le joueur B devine le bon nombre ou que les tentatives sont épuisées. Si B trouve le bon nombre, A perd sa mise et B gagne. Si B ne trouve pas, A garde sa mise.

Le jeu Devine Nombre

Version 3 : Smart Contract générique

Reprendre les versions précédentes mais en déployant un unique contrat générique qui permet à des paires de joueurs (A,B) d'initier et de joueur à ce jeu.

Le jeu Devine Nombre

Version 4 : Jeu avec confidentialité du nombre

Dans cette version, nous ajoutons un mécanisme pour garantir que le nombre choisi par A reste secret pour le joueur B jusqu'à la fin du jeu.

Ce modèle protège la confidentialité du nombre pour éviter que le joueur B ne puisse le découvrir à partir des informations publiques sur la blockchain.

Règles :

- **Joueur A :** Choisit un nombre secret, mais ce nombre est « masqué » avant d'être soumis au smart contract.
- **Joueur B :** Devine le nombre mystère en proposant un nombre à chaque tour. Le smart contract compare la proposition de B avec le nombre « masqué » de A sans révéler directement ce nombre à B.