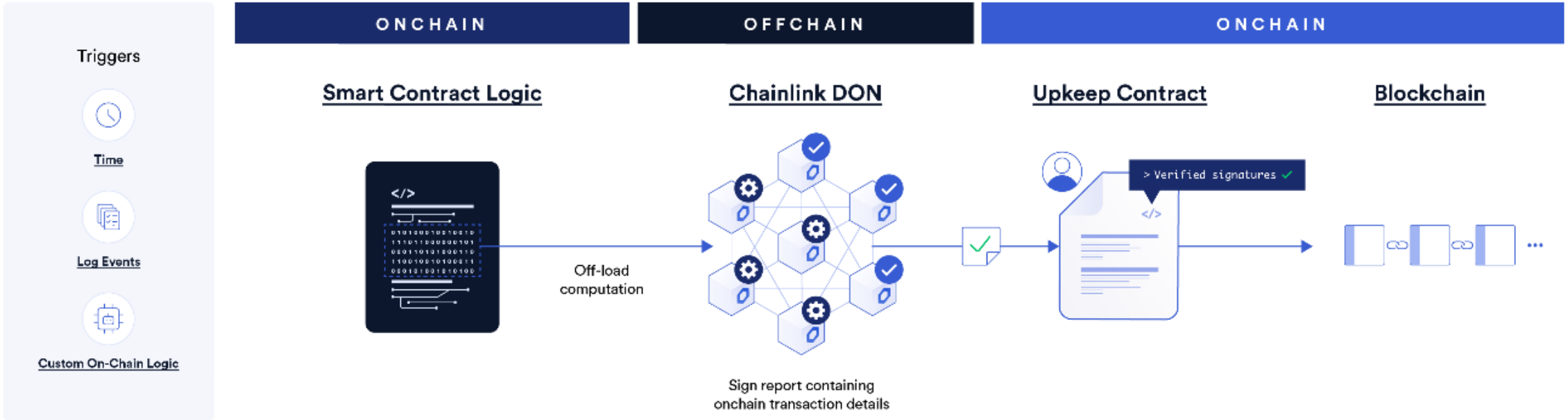

Développer et déployer des
contrats intelligents sur Ethereum
Automatisation avec Chainlink
Contrats Évolutifs - UUPS

Automatisation avec Chainlink

Automatisation Chainlink

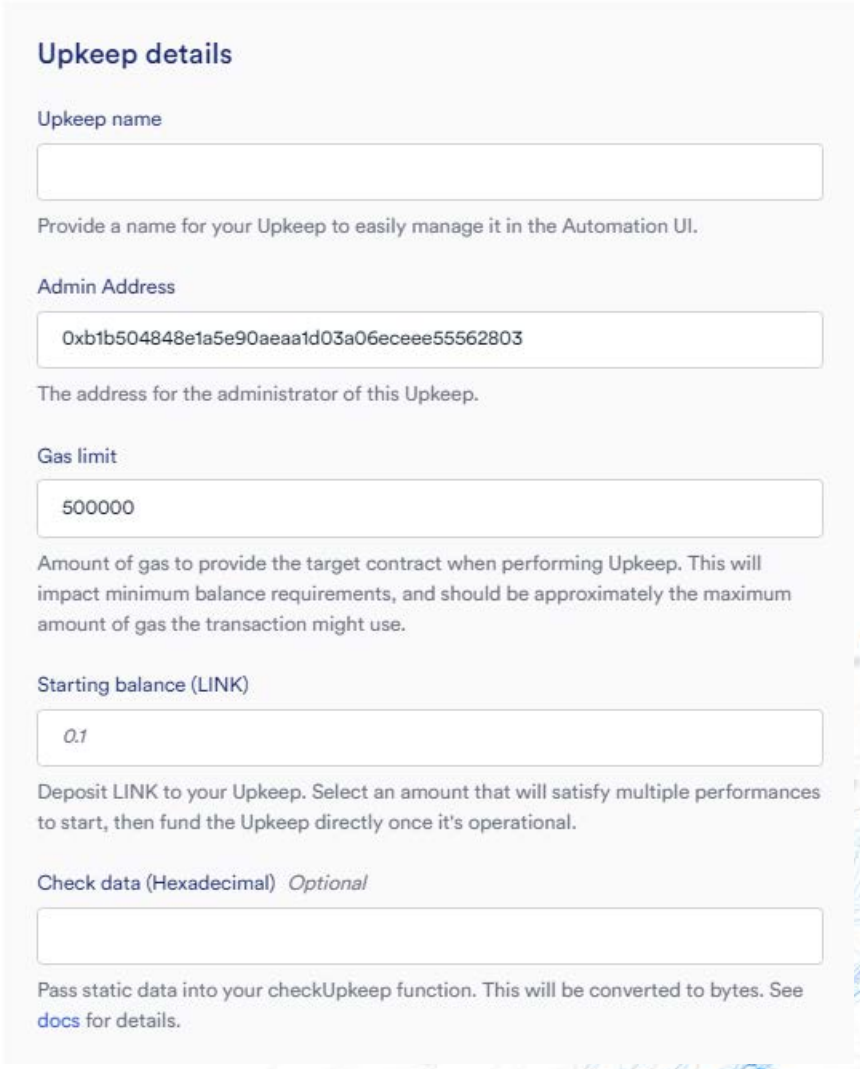
- L'automatisation Chainlink exécute des fonctions de contrat intelligent en utilisant une variété de déclencheurs : basés sur le temps, la logique personnalisée (évaluée hors chaîne), journal des transactions
- Les nœuds d'automatisation simulent la fonction "***checkUpkeep***" => déterminent quand un "***Upkeep***" doit être effectuées
- Le rapport signé contient les "***performData***" qui seront exécutés sur la chaîne



Inscription d'un Upkeep

- Sélectionner un réseau et enregistrer un Upkeep: https://automation.chain.link

- Choisir un déclencheur, fournir une adresse de contrat cible et les détails de l'entretien : nom de l'entretien, adresse de l'administrateur, limite de gaz, solde LINK, données fournies pour la fonction "***checkUpkeep***" (optionnel)



LINK pour le réseau Sepolia

- Obtenir des jetons LINK de test : https://faucets.chain.link/sepolia
- Contrats pour le mainnet, Sepolia, BNB, Avalanche, Polygon, Arbitrum, Optimism... : https://docs.chain.link/resources/link-token-contracts
- Adresse LINK de Sepolia : 0x779877A7B0D9E8603169DdbD7836e478b4624789

Les contrats compatibles avec l'automatisation

L'interface à implémenter dépend du déclencheur utilisé :

- Pour un déclencheur basé sur le temps, aucune interface n'est requise
- Pour un déclencheur de logique personnalisée, l'interface ***AutomationCompatibleInterface*** est requise
- Pour un déclencheur de journal, l'interface ***ILogAutomation*** est requise

Les contrats compatibles avec une logique personnalisée

- Importer ***AutomationCompatibleInterface*** depuis ***AutomationCompatibleInterface.so***l

- Inclure une fonction ***checkUpkeep*** => contient la logique qui sera exécutée hors chaîne pour déterminer si ***performUpkeep*** doit être exécuté. checkUpkeep peut utiliser des données onchain et un paramètre checkData (bytes) pour effectuer des calculs complexes hors chaîne, puis envoyer le résultat à performUpkeep

  function checkUpkeep(bytes calldata checkData) external view override returns (bool upkeepNeeded, bytes memory performData)

- Inclure une fonction ***performUpkeep*** qui sera exécutée onchain lorsque ***checkUpkeep*** renvoie true

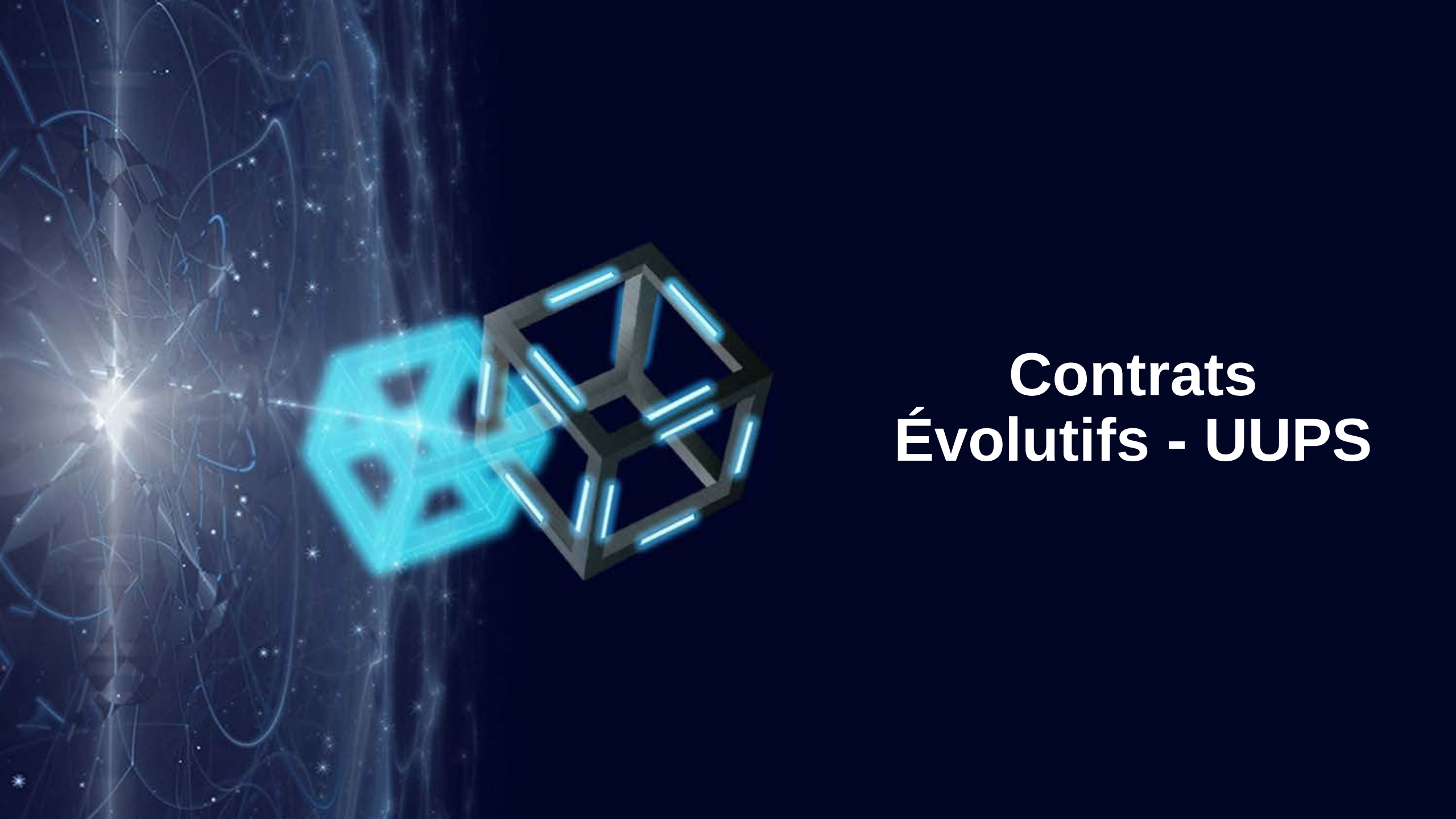
  function performUpkeep(bytes calldata performData) external

- Une fois que le Upkeep est enregistré, le réseau d'automatisation Chainlink simule fréquemment la fonction checkUpkeep hors chaîne pour déterminer si performUpkeep doit être appelée

- Lorsque ***checkUpkeep*** renvoie true, Chainlink appellee la fontion ***performUpkeep*** onchain

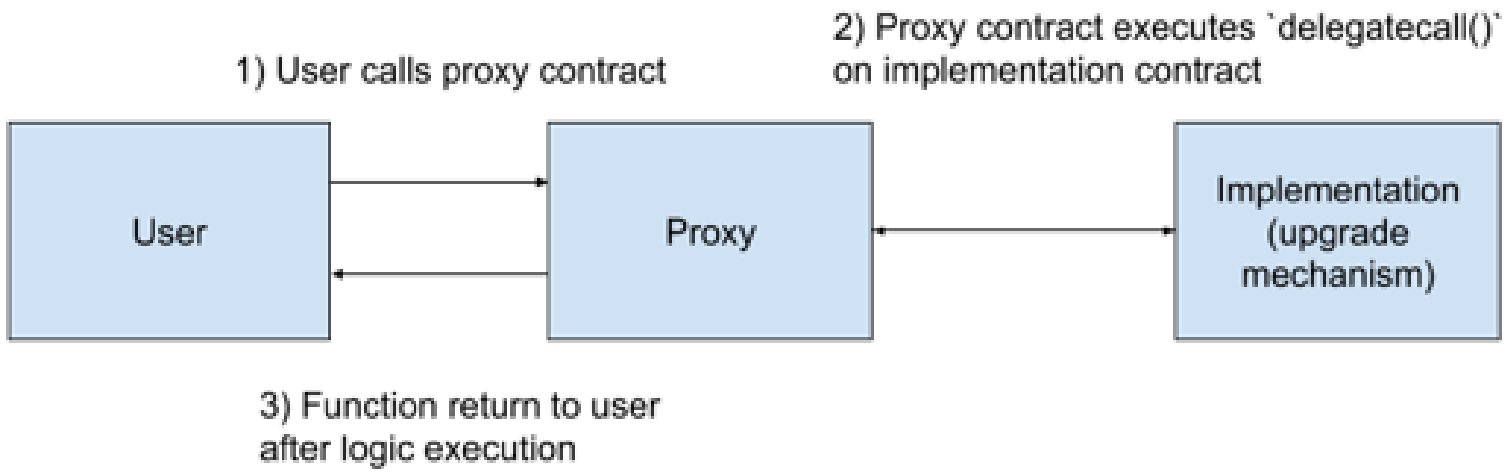
Effectuer des calculs complexes sans frais de gaz

- Ajouter des calculs intensifs en gaz à la fonction ***checkUpkeep()***
- Cette computation ne consomme aucun gaz - les nœuds d'automatisation effectuent la computation hors chaîne
- Plusieurs Upkeeps peuvent être utilisées pour le même contrat afin d'effectuer le travail en parallèle



Contrats Évolutifs - UUPS

Les contrats évolutifs



- Le client interagit toujours avec le contrat proxy
- Le proxy effectue un ***delegatecall()*** sur le contrat d'implémentation
- Le contrat d'implémentation modifie le stockage du contrat proxy

UUPS - universal upgradeable proxy standard

- L'upgrade est déclenchée via le contrat de logique
- Il existe un emplacement de stockage unique dans le contrat proxy pour stocker l'adresse du contrat de logique auquel il pointe
- Chaque fois, un upgrade est effectué, cet emplacement de stockage est mis à jour avec l'adresse du nouveau contrat de logique

Initialisation d'un contrat d'implémentation

- Le code à l'intérieur du constructeur d'un contrat de logique ne fait pas partie du bytecode et il ne sera jamais exécuté dans le contexte de l'état du proxy.

- Solution : déplacer le code du constructeur vers une fonction "***initialize***" et appeler-la lorsque le proxy est lié à la logique.

Paquets requis :

- npm i @openzeppelin/***contracts-upgradeable*** @openzeppelin/***hardhat-upgrades*** --save-dev

Méthodes clés :

- ***initialize()*** : Les contrats évolutifs devraient avoir une méthode initialize à la place des constructeurs, ainsi que le modificateur ***initializer*** pour s'assurer que le contrat est initialisé une seule fois

- ***_authorizeUpgrade()*** : Protéger les mises à jour non autorisées en utilisant le modificateur ***onlyOwner***

Exemple : Créer et déployer un contrat évolutif

- Création d'un contrat évolutif (Pizza) en utilisant le modèle de proxy UUPS
- Déploiement du contrat en utilisant la fonction ***upgrades.deployProxy()***

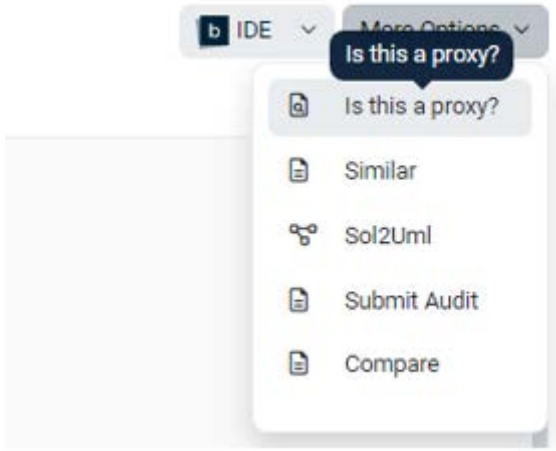
```
3   const PizzaFactory = await ethers.getContractFactory("Pizza")
4
5   const pizza = await upgrades.deployProxy(PizzaFactory, [SLICES], {
6       initializer: "initialize",
7       kind: "uups",
8   })
9
10  await pizza.waitForDeployment()
```

Deux nouveaux contrats sont déployés via deux transactions :

- Le premier est le contrat Pizza (le contrat d'implémentation)
- Le deuxième est le contrat proxy

Vérification du contrat

- **npx hardhat verify --network sepolia CONTRACT_ADDRESS**
- Nous interagissons avec le contrat Pizza via le contrat proxy => informer Etherscan que le contrat est un proxy
- Dans l'onglet Contrat, choisir "***Is this a proxy ?***" dans le menu déroulant et cliquer sur Vérifiee



- Pour interagir avec le contrat, utiliser les options "Read as Proxy" et "Write as Proxy" dans l'onglet du contrat proxy

Mise à jour du contrat

- Créer une nouvelle version du contrat Pizza en ajoutant ou modifiant des fonctions
- Mettre à niveau le contrat Pizza existant en utilisant la fonction ***upgrades.upgradeProxy()***

```
3   const PizzaV2Factory = await ethers.getContractFactory("PizzaV2")
4
5   await upgrades.upgradeProxy(PROXY_CONTRACT_ADDRESS, PizzaV2Factory)
6
```

La fonction upgradeProxy() crée 2 transactions :

- Le déploiement du contrat PizzaV2
- L'appel de "***upgradeTo***" sur PizzaV2 pour effectuer une mise à jour => le contrat proxy pointe vers le nouveau contrat PizzaV2

Bibliothèques de contrats intelligents évolutifs

- Utiliser la variante évolutif des Contrats OpenZeppelin
- Définir la fonction d'initialisation publique et appeler l'initialiseur du contrat parent
- Convention de nommage pour l'initialiseur parent : **__{*ContractName*}_init**

```
3   import {ERC721Upgradeable} from "@openzeppelin/contracts-upgradeable/token/ERC721/ERC721Upgradeable.sol";
4
5   contract MyUpgradeableNFT is ERC721Upgradeable {
6       function initialize() public initializer {
7           __ERC721_init("MyNFT", "MNFT");
8       }
9   }
```

Héritage et initialiseurs

- Solidity se charge d'appeler automatiquement les constructeurs de tous les ancêtres d'un contrat
- Dans les contrats évolutifs, les initialiseurs de tous les contrats parents doivent être appelés manuellement
- Les contrats parents doivent utiliser le modificateur "***onlyInitializing***"

```
3   contract BaseContract is Initializable {
4       uint256 public x;
5
6       function initialize() public onlyInitializing {
7           x = 42;
8       }
9   }
10
11  contract MyContract is BaseContract {
12      uint256 public y;
13
14      function initialize(uint256 _y) public initializer {
15          BaseContract.initialize();
16          y = _y;
17      }
18  }
```

Éviter les valeurs initiales dans les déclarations de champs

- L'utilisation de valeurs initiales est équivalente à définir ces valeurs dans le constructeur
- Les valeurs initiales doivent être définies dans une fonction d'initialisation, sinon elles ne seront pas définies
- Il est acceptable de définir des variables d'état constantes => le compilateur ne réserve pas un emplacement de stockage pour ces variables

```
3   contract MyContract is Initializable {
4       uint256 public hasInitialValue;
5
6       function initialize() public initializer {
7           hasInitialValue = 42;
8       }
9   }
```

Modification d'un contrat d'implémentation

Éviter les collisions de stockage entre les contrats d'implémentation :

- Ne changer pas l'ordre des variables d'état
- Ne changer pas le type des variables d'état
- Éviter d'introduire de nouvelle(s) variable(s) d'état avant les existantes
- Ajouter de nouvelles variables d'état à la fin

```
3   contract MyContractV1 {
4       uint256 private x;
5       string private y;
6   }
7
8   contract MyContractV2 {
9       string private y;
10      uint256 private x;
11  }
```

On ne peut pas ajouter de nouvelles variables aux contrats de base, si le contrat enfant a également des variables de stockage :

- Si une variable supplémentaire est ajoutée au contrat de base, elle se verra attribuer l'emplacement qu'avait le contrat enfant dans la version précédente

```
3   contract Base {
4       uint256 base1;
5   }
6
7
8   contract Child is Base {
9       uint256 child;
10  }
```

Storage Gaps

- Réserver des emplacements de stockage dans un contrat de base => les futures versions de ce contrat peuvent utiliser ces emplacements sans affecter la disposition du stockage des contrats enfants
- Pour créer un espacement de stockage, déclarer un tableau de taille fixe dans le contrat de base avec un nombre initial d'emplacements
- Déclarer un tableau de uint256 afin que chaque élément réserve un emplacement de 32 octets
- Utilisez le nom ***__gap*** ou un nom commençant par ***__gap_*** pour le tableau afin que les Upgrades de OpenZeppelin reconnaissent l'espacement
- Si le contrat de base doit ajouter ultérieurement des variable(s) supplémentaire(s), il faut réduire le nombre approprié d'emplacements de l'espacement de stockage

```
3   contract BaseV1 {
4       uint256 base1;
5       uint256[49] __gap;
6   }
7
8   contract Child is Base {
9       uint256 child;
10  }
11
12  contract BaseV2 {
13      uint256 base1;
14      uint256 base2; // 32 bytes
15      uint256[48] __gap;
16  }
```