LogRocket
Frontend Analytics

Feb 24, 2022 · 8 min read

Using the UUPS proxy pattern to upgrade smart contracts

Pranesh A. S.
Backend Engineer and Blockchain Developer. Keep learning, spread knowledge.

Table of contents

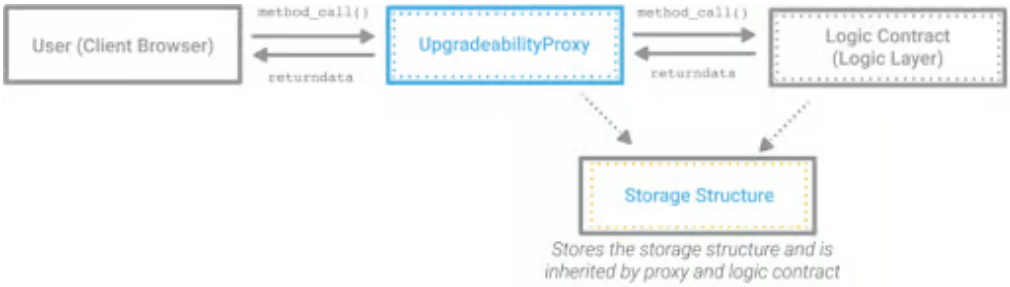
We all know that one of the most impressive features of the blockchain is its immutability property. But it is not advantageous in all cases.

Imagine a deployed smart contract that holds user funds having a vulnerability. Developers should fix the bug as early as possible, like what they can do in the Web2 applications. But in terms of Web3 development, it's not the same case.

Traditional smart contract patterns don't allow such hot fixes. Instead, the developers need to deploy a new contract every time they want to add a feature or fix a bug. While it doesn't seem like a big issue in the beginning, it'll be a huge overhead when the codebase grows. And every time, the data needs to be migrated from the old contract to a new contract to reflect the current state of the protocol.

To solve this problem, various upgradability patterns have been introduced. Among them, the proxy pattern is considered the truest form of upgradability.

When we speak about upgradability, it means that the client always interacts with the same contract (proxy), but the underlying logic can be changed (upgraded) whenever needed without losing any previous data.



N.B., *one can argue that via upgradable proxies, a protocol can even change the underlying logic for their needs (even without the knowledge of their community). There are various methods to prevent that, as DAOs follow timelocks. But, it's beyond the scope of the article and that itself is a topic for another day.*

Here's a summary of what we'll cover in this article:

Types of proxy patterns

Currently, there are three types of proxy patterns:

- Diamond pattern: EIP-2532
- Transparent proxy pattern
- Universal upgradeable proxy standard (UUPS): EIP-1822

What is a diamond pattern?

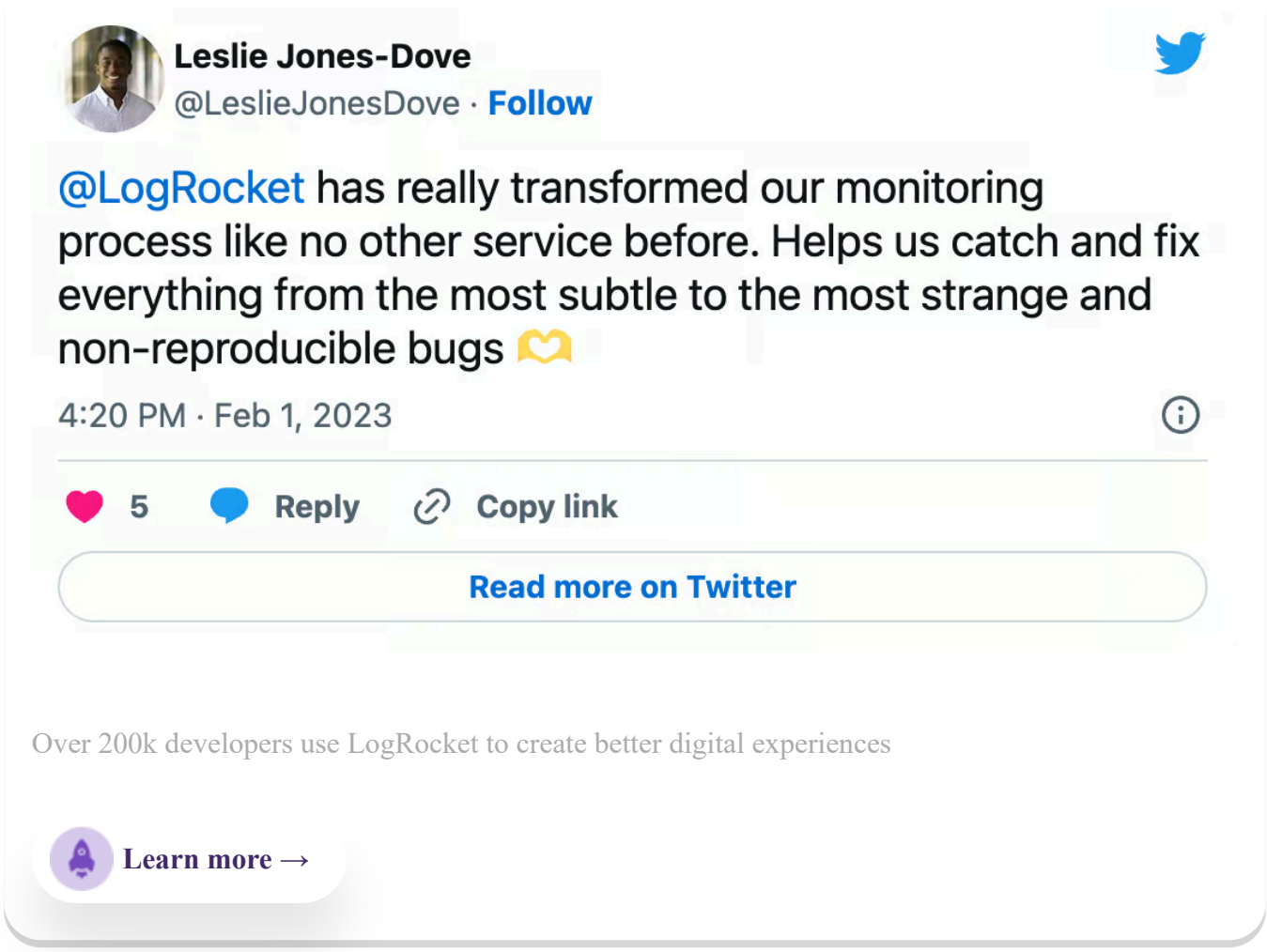
In a nutshell, a diamond pattern is an upgradeable proxy pattern in which there are multiple logic contracts (facets) instead of one. Whenever a function is called in the diamond proxy contract, it checks the hash table (mapping) to see which facet has the function defined and then delegates the call to that facet. This `delegatecall` occurs in the proxy's `fallback()` method. We can add or remove any number of methods from any facet in a single transaction using a method called `diamondCut()`. In order to avoid storage collisions, this pattern uses the `DiamondStorage` technique. It also allows the developers to implement logic in facets, independent of other facets.

What is a Transparent proxy pattern?

As mentioned previously, to implement an upgradeable smart contract, the logic layer (i.e., the implementation contract) is separated from the storage layer (i.e., the proxy contract) and all calls to the proxy contract are delegated to the logic contract.

This method worked just fine until It was fine until a malicious backdoor, proxy selector clashing, was identified and addressed. Proxy selector clashing occurs when two or more methods have identical function signatures in the proxy and logic contract. This can lead to smart contract exploits.

To resolve the clashing, OpenZeppelin introduced the transparent proxy pattern. This pattern allows identical function signatures to exist in the proxy and logic contract, but the `delegatecall` to the logic contract only occurs if the caller is not a contract admin. Otherwise, the function is invoked in the proxy contract itself if it exists or reverts if not.



What is a UUPS proxy pattern?

The UUPS proxy pattern is similar to the transparent proxy pattern, except the upgrade is triggered via the logic contract rather than from the proxy contract.

There is a unique storage slot in the proxy contract to store the address of the logic contract that it points to. Whenever the logic contract is upgraded, that storage slot is updated with the new logic contract address. The function to upgrade the contracts should be a protected function to avoid unauthorized access. Also, this provides the ability to go completely non-upgradeable gradually as the logic contract can completely remove the `upgradeTo()` method in the new implementation if needed.

Comparing proxy patterns

The below table compares the pros and cons of the diamond, transparent, and UUPS proxy patterns:

Proxy pattern	Pros	Cons
Transparent proxy pattern	**Comparatively easy and simpler to implement; widely used**	**Requires more gas for deployment, comparatively**
Diamond proxy pattern	**Helps to battle the 24KB size limit via modularity; incremental upgradeability**	**More complex to implement and maintain; uses new terminologies that can be harder for newcomers to understand; as of this writing, not supported by tools like Etherscan**
UUPS proxy pattern	**Gas efficient; Flexibility to remove upgradeability**	**Not as commonly used as it is fairly new; extra care is required for the upgrade logic (access control) as it resides in the implementation contract**

When should we use UUPS?

OpenZeppelin suggests using the UUPS pattern as it is more gas efficient. But the decision of when to use UUPS is really based on several factors like the business requirements of the projects, and so on.

The original motivation for UUPS was for deploying many smart contract wallets on the mainnet. The logic could be deployed once. The proxy could be deployed hundreds of times for each new wallet, without spending much gas.

As the upgrade method resides in the logic contract, the developer can choose UUPS if the protocol wants to remove upgradeability completely in the future.

It’s demo time: Smart contract upgrade using UUPS proxy pattern

Enough of the introduction and theory. Let’s set up and deploy an upgradable Pizza contract using the UUPS proxy pattern, leveraging Hardhat and OpenZeppelin’s UUPS

library contracts.

Setup with Hardhat and OpenZeppelin

We'll deploy a simple smart contract called `Pizza` and upgrade it to `PizzaV2` using the UUPS proxy pattern.

As we'll be using Hardhat for development purposes, you'll need to have NodeJS and NPM installed in your machine.

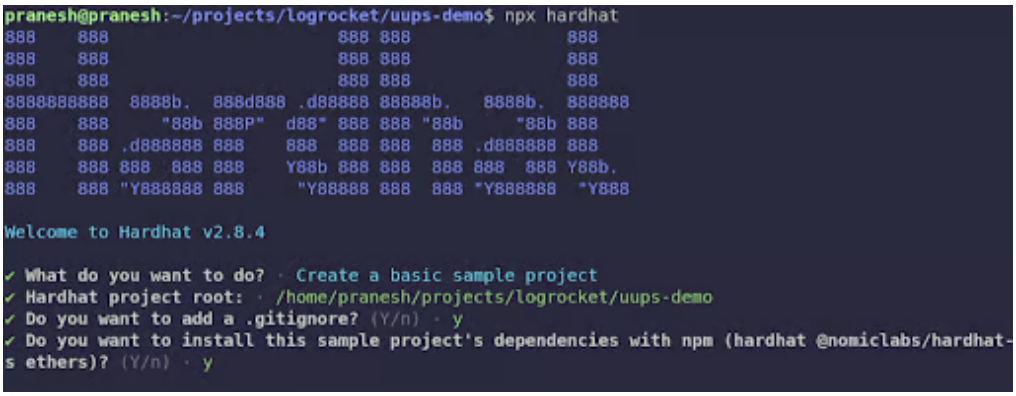
Once Node.js is installed and set up, you can install Hardhat globally in your machine from your command line by running the command `npm install hardhat -g`.

Once Hardhat is installed, you can create new Hardhat projects easily!

Let's create a fresh new directory for our project:

```
mkdir uups-demo && cd uups-demo
```

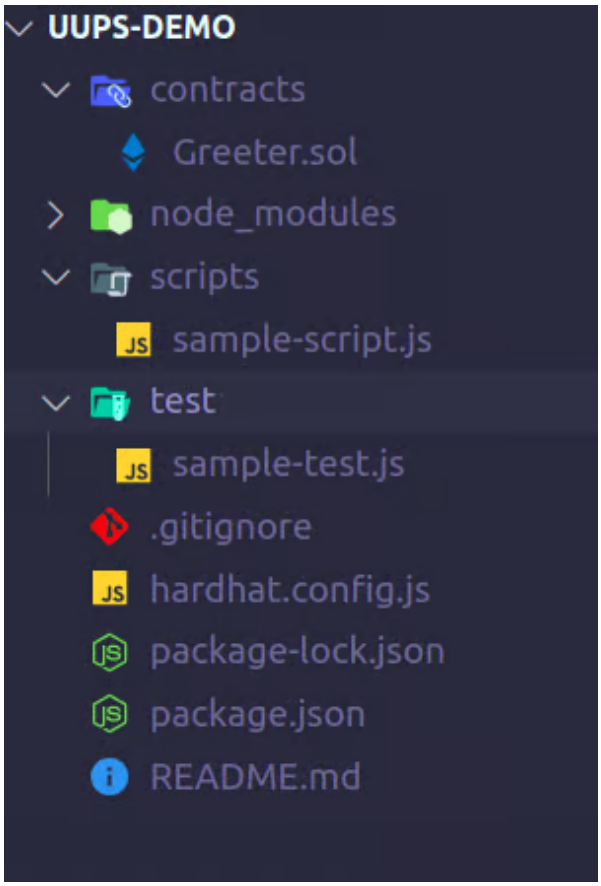
Initialize a new Hardhat project by running `npx hardhat` and choosing the initial config for the project.



Now Hardhat will install some of the required libraries. Other than that, we'll require some additional npm modules as well for the UUPS pattern. Run the following command to install the modules.

```
npm i @openzeppelin/contracts-upgradeable @openzeppelin/hardhat-upgra
```

Once everything has been installed, the initial directory structure will look something like this:



N.B., *the file names and the contents will be modified as we proceed.*

Making the implementation and proxy contracts

Create a new file called `Pizza.sol` inside the `contracts` directory and add the following code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

// Open Zeppelin libraries for controlling upgradability and access.
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable
import "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeab
import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable

contract Pizza is Initializable, UUPSUpgradeable, OwnableUpgradeable
    uint256 public slices;

        ///@dev no constructor in upgradable contracts. Instead we have
      ///@param _sliceCount initial number of slices for the pizza
```

```
    function initialize(uint256 _sliceCount) public initializer {
        slices = _sliceCount;

       ///@dev as there is no constructor, we need to initialise the O
        __Ownable_init();
    }

    ///@dev required by the OZ UUPS module
    function _authorizeUpgrade(address) internal override onlyOwner {}

    ///@dev decrements the slices when called
    function eatSlice() external {
        require(slices > 1, "no slices left");
        slices -= 1;
    }
}
```

This is a simple Pizza contract that has three methods:

- `initialize()` : Upgradable contracts should have an `initialize` method in place of constructors, and also the `initializer` keyword makes sure that the contract is initialized only once
- `_authorizeUpgrade()` : This method is required to safeguard from unauthorized upgrades because in the UUPS pattern the upgrade is done from the implementation contract, whereas in the transparent proxy pattern, the upgrade is done via the proxy contract
- `_eatSlice()` : A simple function to reduce the slice count whenever called

Now let's compile and deploy the Pizza contract.

Before doing that, we have to update the `hardhat.config.js` file with the following contents:

```
require("@nomiclabs/hardhat-ethers");
require("@openzeppelin/hardhat-upgrades");
```

```
require("@nomiclabs/hardhat-etherscan");

require("dotenv").config();

module.exports = {
 solidity: "0.8.10",
 networks: {
   kovan: {
     url: `https://kovan.infura.io/v3/${process.env.INFURA_API_KEY}`,
     accounts: [process.env.PRIVATE_KEY],
   },
 },
 etherscan: {
   apiKey: process.env.ETHERSCAN_API_KEY,
 },
};
```

Create a new file called `.env` and add the following contents:

```
PRIVATE_KEY = <<DEPLOYER_PRIVATE_KEY>>
ETHERSCAN_API_KEY = <<ETHERSCAN_API_KEY>>
INFURA_API_KEY=  <<INFURA_API_KEY>>
```

The `PRIVATE_KEY` is the private key of the deployer wallet. You can grab the `INFURA_API_KEY from here` and `ETHERSCAN_API_KEY from here`.

Once the `.env` file is created, you can compile the contracts by running `npx hardhat compile` in your terminal.

Now let's deploy our contract. Inside the `scripts` directory, create a new file called `deploy_pizza_v1.js` and add the following contents:

```
const { ethers, upgrades } = require("hardhat");

const SLICES = 8;
```

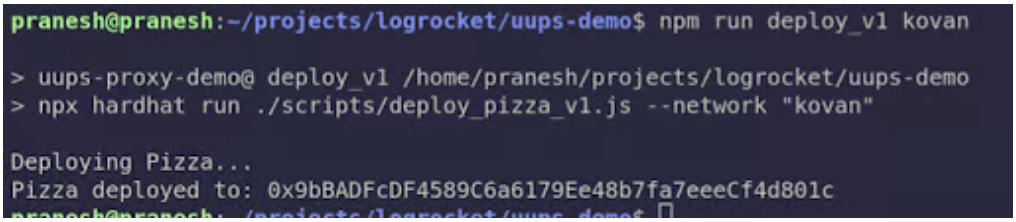
```
async function main() {
 const Pizza = await ethers.getContractFactory("Pizza");

 console.log("Deploying Pizza...");

 const pizza = await upgrades.deployProxy(Pizza, [SLICES], {
   initializer: "initialize",
 });
 await pizza.deployed();

 console.log("Pizza deployed to:", pizza.address);
}

main();
```

Save the file.

Now you can deploy the contracts by running the following command in the terminal:

```
npx hardhat run ./scripts/deploy_pizza_v1.js -network kovan
```

You should see something like this. The address will be different!



The address displayed in the console is the address of the proxy contract. If you visit Etherscan and search the deployer address, you'll see two new contracts created via two transactions. The first one is the actual Pizza contract (the implementation contract), and the second one is the proxy contract.

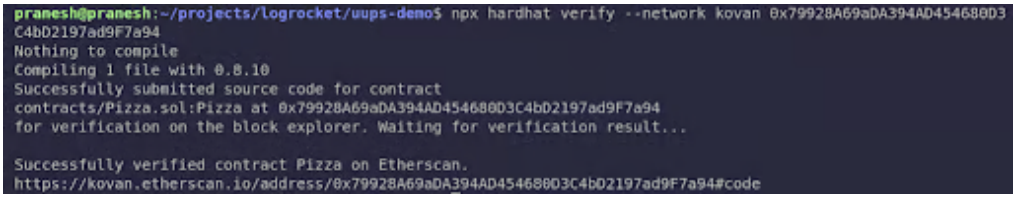
In my case, the Pizza contract address is 0x79928a69ada394ad454680d3c4bd2197ad9f7a94.

The proxy contract address is 0x9bBADFcDF4589C6a6179Ee48b7fa7eeeCf4d801c.

You can copy the address of the Pizza contract from Etherscan and verify it by running the command below:

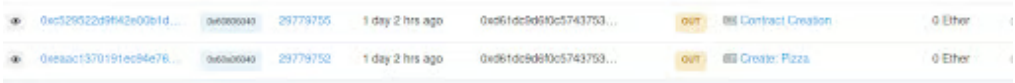
```
npx hardhat verify –network kovan <<CONTRACT_ADDRESS>>
```

The output should be something like this:



N.B., *after deployment, if you're confused which contract is the proxy or implementation, the proxy contract source code will be already verified on Etherscan (in most cases). The unverified will be the implementation contract!*

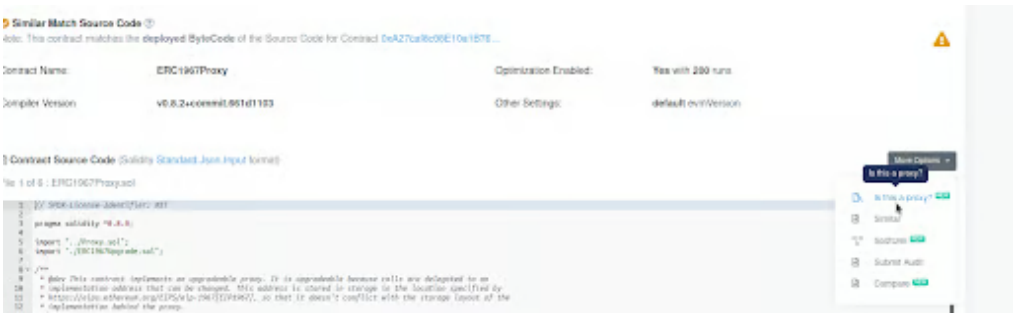
Once verified, your Etherscan transactions will look like this:



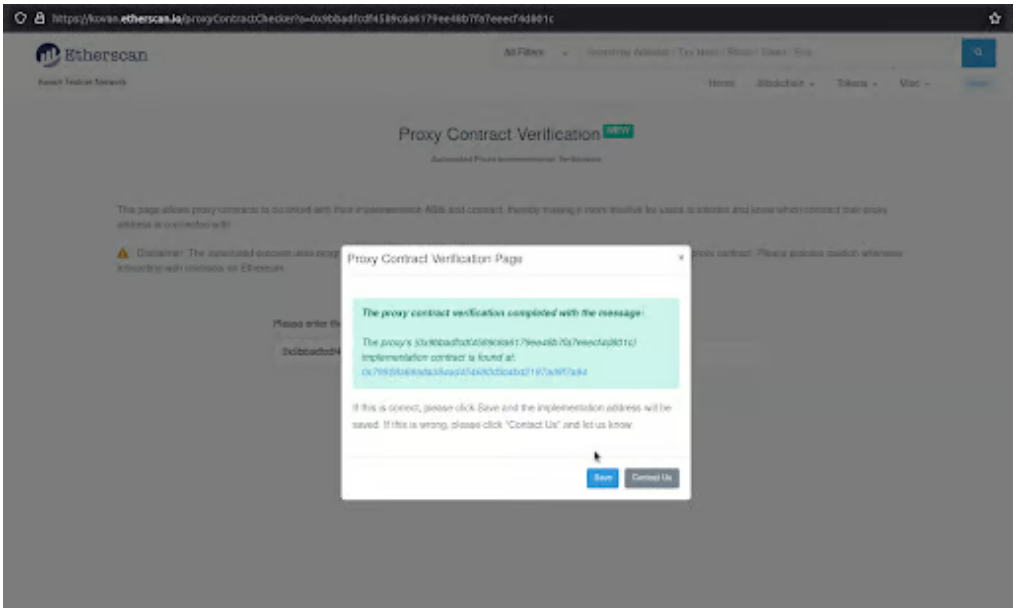
If you check the Pizza contract in Etherscan, the values like `owner`, `slices`, etc. will not be set or initialized because in the proxy pattern, everything is stored and executed in the context of the proxy contract.

So in order to interact with the Pizza contract, you should do it via the proxy contract. To do that, first we need to inform Etherscan that the deployed contract is actually a proxy.

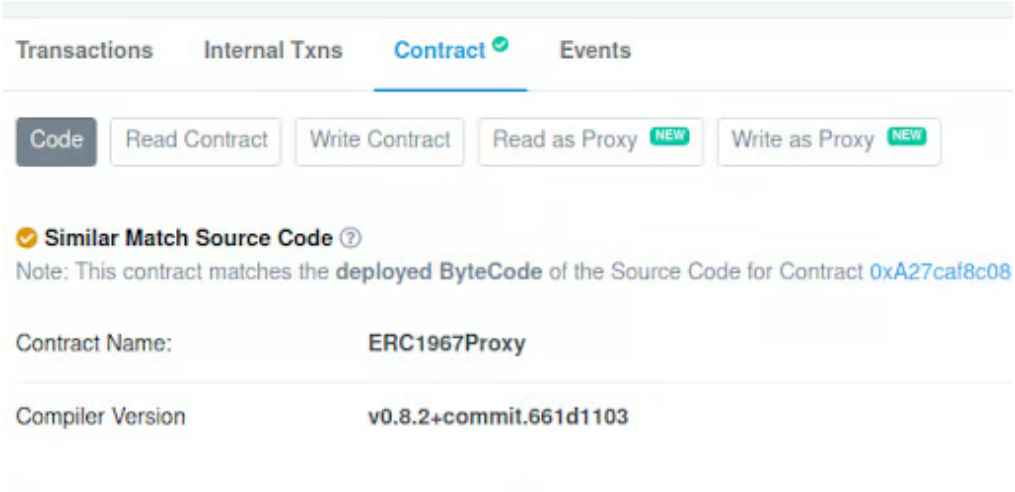
In the `Contract` tab of the proxy contract, there'll be a small dropdown above the source code section (on the right side).



Choose "**Is this a proxy?**" option from the dropdown and then **Verify**.



You can see **Read as Proxy** and **Write as Proxy** options in the **Contract** tab of the proxy contract.



Now you can interact with the Pizza contract using those options!

Upgrading the contract

After some time passes, let's say we have to include additional functionality to our Pizza contract. For example, let's make a simple function to refill slices and a function to return the current contract version.

We can create our PizzaV2 contract. Inside the contracts folder, create a new file called `PizzaV2.sol`, add the following contents, and save the file:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.10;

import "./Pizza.sol";

contract PizzaV2 is Pizza {
   ///@dev increments the slices when called
   function refillSlice() external {
       slices += 1;
   }

   ///@dev returns the contract version
   function pizzaVersion() external pure returns (uint256) {
       return 2;
   }
}
```

N.B., *the PizzaV2 contract inherits from the Pizza contract. So all the functions including the two newer functions will be present in the V2 contract.*

Once the file is saved, now we can upgrade our Pizza contract to PizzaV2. Inside the `scripts` directory, create a new file `upgrade_pizza_v2.js` , add the following contents, and save the file. It is responsible for upgrading the deployed contract:

```
const { ethers, upgrades } = require("hardhat");

const PROXY = <<REPLACE_WITH_YOUR_PROXY_ADDRESS>>;

async function main() {
 const PizzaV2 = await ethers.getContractFactory("PizzaV2");
 console.log("Upgrading Pizza...");
 await upgrades.upgradeProxy(PROXY, PizzaV2);
 console.log("Pizza upgraded successfully");
}
```

```
main();
```

You can run the following command to execute the upgrade:

```
npx hardhat run ./scripts/upgrade_pizza_v2.js –network kovan
```

```
pranesh@pranesh:~/projects/logrocket/uups-demo$ npm run upgrade_v2 kovan

> uups-proxy-demo@ upgrade_v2 /home/pranesh/projects/logrocket/uups-demo
> npx hardhat run ./scripts/upgrade_pizza_v2.js --network "kovan"

Upgrading Pizza...
Pizza upgraded successfully
```

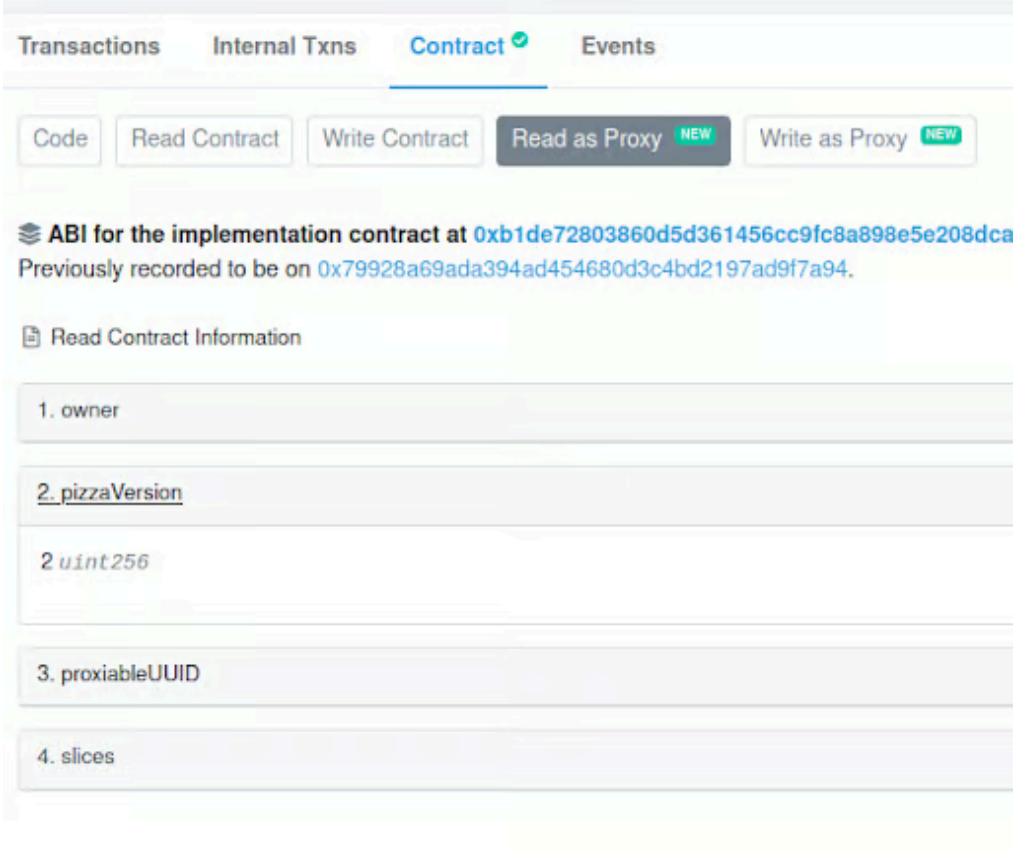
N.B., *if you face any errors when running the above command, retry it two or three times. It should work.*

If you check Etherscan, you can see there'll be another two transactions from the deployer wallet. The first one is the deployment of the PizzaV2 contract and the second transaction will be the `upgradeTo` call in the Pizza contract to perform an upgrade. This makes sure that the proxy contract points to the newly deployed PizzaV2 contract.

You can verify the PizzaV2 contract from the terminal by running:

```
npx hardhat verify –network kovan <<PIZZA_V2_ADDRESS>>
```

If you check the **Write as Proxy** tab inside the **Contract** tab of the proxy contract in Etherscan, you can see the newly created view method — `refillSlice()` — along the older methods. Also, there will be a `pizzaVersion()` method in the **Read as Proxy** tab, which confirms that the upgrade is successful!



Whoa. 🎉 🎉 We've successfully deployed and upgraded contracts using the UUPS proxy pattern!

Closing thoughts

Though there are several advantages to the UUPS pattern, and the recommended proxy pattern is currently the UUPS pattern, there are some caveats that we should be aware of before implementing this into a real-world project.

One of the main caveats is that because the upgrades are done via the implementation contract with the help of `upgradeTo` method, there's a higher risk of newer implementations to exclude the `upgradeTo` method, which may permanently kill the ability to upgrade the smart contract. Also, this pattern is a bit complex to implement when compared to other proxy patterns.

Despite the warnings, UUPS is a very gas-efficient proxy pattern that has several advantages. The code for this project along with some unit tests can be found in the GitHub repo here. Feel free to play around with the code.

Happy coding! 🎉

Share this:

#blockchain

Recent posts:

Leader Spotlight: Riding the rocket ship of scale, with Justin Kitagawa

We sit down with Justin Kitagawa to learn more about his leadership style and approach for handling the complexities that come with scaling fast.

Jessica Srinivas

Mar 29, 2024 · 8 min read

Nx adoption guide: Overview, examples, and alternatives

Let’s explore Nx features, use cases, alternatives, and more to help you assess whether it’s the right tool for your needs.

Andrew Evans

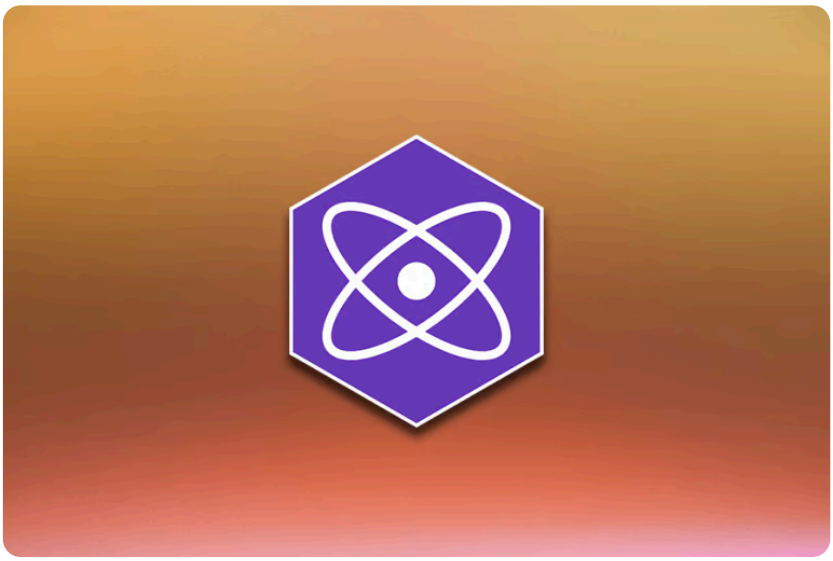
Mar 28, 2024 · 9 min read

Understanding security in React Native applications

Explore the various security threats facing React Native mobile applications and how to mitigate them.

Wisdom Ekpotu

Mar 27, 2024 · 10 min read



A guide to better state management with Preact Signals

Signals is a performant state management library with a set of reactive primitives for managing the application state.

Nefe James

Mar 26, 2024 · 8 min read

View all posts

1 Replies to "Using the UUPS proxy pattern to upgrade smart contracts"

SoPetey says: Reply

March 7, 2022 at 12:23 am

this is transparent proxy pattern

jonybgoode says: Reply

April 20, 2022 at 11:16 pm

https://kovan.etherscan.io/address/0x9bBADFcDF4589C6a6179Ee48b7fa7eeeCf4d801c#code
The proxy contract shows that it is ERC1967Proxy and the beginning of this blog
“Transparent proxy pattern : EIP-1967”

Mijo Grabovac says: Reply

November 24, 2022 at 6:37 am

Do we really need to import Initializer in our Pizza contract (Considering UUPSUpgradeable already has it). Or was it done here only to make tutorial more readable and understandable?

Akalanka Pathirage says: Reply

February 26, 2024 at 7:38 am

Even though the contract inherit from UUPSUpgradable, upgradable pattern has to be specified in deployProxy function. Otherwise it defaults to transparent proxy.
Should be something like this.

```
const Box = await ethers.getContractFactory("BoxV1UUPS");
console.log("Deploying Box...");
const box = await upgrades.deployProxy(Box, [42], {
initializer: "store",
kind: "uups",
});
```

Leave a Reply