**Using with Upgrades**

If your contract is going to be deployed with upgradeability, such as using the [OpenZeppelin Upgrades Plugins](https://docs.openzeppelin.com/upgrades-plugins/1.x/), you will need to use the Upgradeable variant of OpenZeppelin Contracts.

This variant is available as a separate package called @openzeppelin/contracts-upgradeable, which is hosted in the repository [OpenZeppelin/openzeppelin-contracts-upgradeable](https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable).

It follows all of the rules for [Writing Upgradeable Contracts](https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable): constructors are replaced by initializer functions, state variables are initialized in initializer functions, and we additionally check for storage incompatibilities across minor versions.

|  |  |
| --- | --- |
|  | OpenZeppelin provides a full suite of tools for deploying and securing upgradeable smart contracts. [Check out the full list of resources](https://docs.openzeppelin.com/contracts/4.x/upgradeable#openzeppelin::upgrades.adoc). |

**Overview**

**Installation**

$ npm install @openzeppelin/contracts-upgradeable

**Usage**

The package replicates the structure of the main OpenZeppelin Contracts package, but every file and contract has the suffix Upgradeable.

-import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

+import "@openzeppelin/contracts-upgradeable/token/ERC721/ERC721Upgradeable.sol";

-contract MyCollectible is ERC721 {

+contract MyCollectible is ERC721Upgradeable {

Constructors are replaced by internal initializer functions following the naming convention \_\_{ContractName}\_init. Since these are internal, you must always define your own public initializer function and call the parent initializer of the contract you extend.

- constructor() ERC721("MyCollectible", "MCO") public {

+ function initialize() initializer public {

+ \_\_ERC721\_init("MyCollectible", "MCO");

}

|  |  |
| --- | --- |
|  | Use with multiple inheritance requires special attention. See the section below titled [Multiple Inheritance](https://docs.openzeppelin.com/contracts/4.x/upgradeable#multiple-inheritance). |

Once this contract is set up and compiled, you can deploy it using the [Upgrades Plugins](https://docs.openzeppelin.com/upgrades-plugins/1.x/). The following snippet shows an example deployment script using Hardhat.

// scripts/deploy-my-collectible.js

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

async function main() {

const MyCollectible = await ethers.getContractFactory("MyCollectible");

const mc = await upgrades.deployProxy(MyCollectible);

await mc.deployed();

console.log("MyCollectible deployed to:", mc.address);

}

main();

**Further Notes**

**Multiple Inheritance**

Initializer functions are not linearized by the compiler like constructors. Because of this, each \_\_{ContractName}\_init function embeds the linearized calls to all parent initializers. As a consequence, calling two of these init functions can potentially initialize the same contract twice.

The function \_\_{ContractName}\_init\_unchained found in every contract is the initializer function minus the calls to parent initializers, and can be used to avoid the double initialization problem, but doing this manually is not recommended. We hope to be able to implement safety checks for this in future versions of the Upgrades Plugins.

**Storage Gaps**

You may notice that every contract includes a state variable named \_\_gap. This is empty reserved space in storage that is put in place in Upgradeable contracts. It allows us to freely add new state variables in the future without compromising the storage compatibility with existing deployments.

It isn’t safe to simply add a state variable because it "shifts down" all of the state variables below in the inheritance chain. This makes the storage layouts incompatible, as explained in [Writing Upgradeable Contracts](https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable#modifying-your-contracts). The size of the \_\_gap array is calculated so that the amount of storage used by a contract always adds up to the same number (in this case 50 storage slots).

# Upgrades Plugins

**Integrate upgrades into your existing workflow.** Plugins for [Hardhat](https://hardhat.org/) and [Truffle](https://www.trufflesuite.com/truffle) to deploy and manage upgradeable contracts on Ethereum.

* Deploy upgradeable contracts.
* Upgrade deployed contracts.
* Manage proxy admin rights.
* Easily use in tests.

|  |  |
| --- | --- |
|  | Upgrades Plugins are only a part of a comprehensive set of OpenZeppelin tools for deploying and securing upgradeable smart contracts. [Check out the full list of resources](https://docs.openzeppelin.com/upgrades). |

## Overview

### Installation

#### Hardhat install

$ npm install --save-dev @openzeppelin/hardhat-upgrades @nomiclabs/hardhat-ethers ethers

This installs our Hardhat plugin along with the necessary peer dependencies.

You also need to load it in your Hardhat config file:

// hardhat.config.js

require('@openzeppelin/hardhat-upgrades');

// hardhat.config.ts

import '@openzeppelin/hardhat-upgrades';

#### Truffle install

$ npm install --save-dev @openzeppelin/truffle-upgrades

### Usage

See the documentation for using [Truffle Upgrades](https://docs.openzeppelin.com/upgrades-plugins/1.x/truffle-upgrades) and [Hardhat Upgrades](https://docs.openzeppelin.com/upgrades-plugins/1.x/hardhat-upgrades), or take a look at the sample code snippets below.

#### Hardhat usage

Hardhat users will be able to write [scripts](https://hardhat.org/guides/scripts.html) that use the plugin to deploy or upgrade a contract, and manage proxy admin rights.

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

async function main() {

// Deploying

const Box = await ethers.getContractFactory("Box");

const instance = await upgrades.deployProxy(Box, [42]);

await instance.deployed();

// Upgrading

const BoxV2 = await ethers.getContractFactory("BoxV2");

const upgraded = await upgrades.upgradeProxy(instance.address, BoxV2);

}

main();

#### Truffle usage

Truffle users will be able to write [migrations](https://www.trufflesuite.com/docs/truffle/getting-started/running-migrations) that use the plugin to deploy or upgrade a contract, or manage proxy admin rights.

const { deployProxy, upgradeProxy } = require('@openzeppelin/truffle-upgrades');

const Box = artifacts.require('Box');

const BoxV2 = artifacts.require('BoxV2');

module.exports = async function (deployer) {

const instance = await deployProxy(Box, [42], { deployer });

const upgraded = await upgradeProxy(instance.address, BoxV2, { deployer });

}

#### Test usage

Whether you’re using Hardhat or Truffle, you can use the plugin in your tests to ensure everything works as expected.

it('works before and after upgrading', async function () {

const instance = await upgrades.deployProxy(Box, [42]);

assert.strictEqual(await instance.retrieve(), 42);

await upgrades.upgradeProxy(instance.address, BoxV2);

assert.strictEqual(await instance.retrieve(), 42);

});

## How the plugins work

Both plugins provide functions which take care of managing upgradeable deployments of your contracts.

For example, deployProxy does the following:

1. Validate that the implementation is [upgrade safe](https://docs.openzeppelin.com/upgrades-plugins/1.x/faq#what-does-it-mean-for-a-contract-to-be-upgrade-safe).
2. Deploy a [proxy admin](https://docs.openzeppelin.com/upgrades-plugins/1.x/faq#what-is-a-proxy-admin) for your project (if needed).
3. Deploy the [implementation contract](https://docs.openzeppelin.com/upgrades-plugins/1.x/faq#what-is-an-implementation-contract).
4. Create and initialize the proxy contract.

And when you call upgradeProxy:

1. Validate that the new implementation is [upgrade safe](https://docs.openzeppelin.com/upgrades-plugins/1.x/faq#what-does-it-mean-for-a-contract-to-be-upgrade-safe) and is [compatible](https://docs.openzeppelin.com/upgrades-plugins/1.x/faq#what-does-it-mean-for-an-implementation-to-be-compatible) with the previous one.
2. Check if there is an [implementation contract](https://docs.openzeppelin.com/upgrades-plugins/1.x/faq#what-is-an-implementation-contract) deployed with the same bytecode, and deploy one if not.
3. Upgrade the proxy to use the new implementation contract.

The plugins will keep track of all the implementation contracts you have deployed in an .openzeppelin folder in the project root, as well as the proxy admin. You will find one file per network there. It is advised that you commit to source control the files for all networks except the development ones (you may see them as .openzeppelin/unknown-\*.json).

Note: the format of the files within the .openzeppelin folder is not compatible with those of the [OpenZeppelin CLI](https://docs.openzeppelin.com/cli/2.8/). If you want to use the Upgrades Plugins for an existing OpenZeppelin CLI project, you can [migrate using the guide](https://docs.openzeppelin.com/upgrades-plugins/1.x/migrate-from-cli).

## Proxy patterns

The plugins support the UUPS, transparent, and beacon proxy patterns. UUPS and transparent proxies are upgraded individually, whereas any number of beacon proxies can be upgraded atomically at the same time by upgrading the beacon that they point to. For more details on the different proxy patterns available, see the documentation for [Proxies](https://docs.openzeppelin.com/contracts/4.x/api/proxy).

For UUPS and transparent proxies, use deployProxy and upgradeProxy as shown above. For beacon proxies, use deployBeacon, deployBeaconProxy, and upgradeBeacon. See the documentation for [Hardhat Upgrades](https://docs.openzeppelin.com/upgrades-plugins/1.x/hardhat-upgrades) and [Truffle Upgrades](https://docs.openzeppelin.com/upgrades-plugins/1.x/truffle-upgrades) for examples.

## Managing ownership

Transparent proxies define an admin address which has the rights to upgrade them. By default, the admin is a [proxy admin contract](https://docs.openzeppelin.com/upgrades-plugins/1.x/faq#what-is-a-proxy-admin) deployed behind the scenes. You can change the admin of a proxy by calling the admin.changeProxyAdmin function in the plugin. Keep in mind that the admin of a proxy can only upgrade it, but not interact with the implementation contract. Read [Transparent Proxies and Function Clashes](https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies#transparent-proxies-and-function-clashes) for more info on this restriction.

The proxy admin contract also defines an owner address which has the rights to operate it. By default, this address is the externally owned account used during deployment. You can change the proxy admin owner by calling the admin.transferProxyAdminOwnership function in the plugin. Note that changing the proxy admin owner effectively transfers the power to upgrade any proxy in your whole project to the new owner, so use with care. Refer to each plugin documentation for more details on the admin functions.

UUPS and beacon proxies do not use admin addresses. UUPS proxies rely on an [\_authorizeUpgrade](https://docs.openzeppelin.com/contracts/4.x/api/proxy#UUPSUpgradeable-_authorizeUpgrade-address-) function to be overridden to include access restriction to the upgrade mechanism, whereas beacon proxies are upgradable only by the owner of their corresponding beacon.

Once you have transferred the rights to upgrade a proxy or beacon to another address, you can still use your local setup to validate and deploy the implementation contract. The plugins include a prepareUpgrade function that will validate that the new implementation is upgrade-safe and compatible with the previous one, and deploy it using your local Ethereum account. You can then execute the upgrade itself from the admin or owner address. You can also use the proposeUpgrade function to automatically set up the upgrade in [Defender Admin](https://docs.openzeppelin.com/defender/admin).

**Using with Hardhat**

This package adds functions to your Hardhat scripts so you can deploy and upgrade proxies for your contracts. Depends on ethers.js.

|  |  |
| --- | --- |
|  | Check out the [step by step tutorial](https://forum.openzeppelin.com/t/openzeppelin-buidler-upgrades-step-by-step-tutorial/3580), showing from creating, testing and deploying, all the way through to upgrading with Gnosis Safe. |

**Installation**

$ npm install --save-dev @openzeppelin/hardhat-upgrades

$ npm install --save-dev @nomiclabs/hardhat-ethers ethers # peer dependencies

And register the plugin in your [hardhat.config.js](https://hardhat.org/config):

require('@openzeppelin/hardhat-upgrades');

**Usage in scripts**

**Proxies**

You can use this plugin in a [Hardhat script](https://hardhat.org/guides/scripts.html) to deploy an upgradeable instance of one of your contracts via the deployProxy function:

// scripts/create-box.js

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

async function main() {

const Box = await ethers.getContractFactory("Box");

const box = await upgrades.deployProxy(Box, [42]);

await box.deployed();

console.log("Box deployed to:", box.address);

}

main();

This will automatically check that the Box contract is upgrade-safe, set up a proxy admin (if needed), deploy an implementation contract for the Box contract (unless there is one already from a previous deployment), create a proxy, and initialize it by calling initialize(42).

Then, in another script, you can use the upgradeProxy function to upgrade the deployed instance to a new version. The new version can be a different contract (such as BoxV2), or you can just modify the existing Box contract and recompile it - the plugin will note it changed.

// scripts/upgrade-box.js

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

async function main() {

const BoxV2 = await ethers.getContractFactory("BoxV2");

const box = await upgrades.upgradeProxy(BOX\_ADDRESS, BoxV2);

console.log("Box upgraded");

}

main();

Note: While this plugin keeps track of all the implementation contracts you have deployed per network, in order to reuse them and validate storage compatibilities, it does *not* keep track of the proxies you have deployed. This means that you will need to manually keep track of each deployment address, to supply those to the upgrade function when needed.

The plugin will take care of comparing BoxV2 to the previous one to ensure they are compatible for the upgrade, deploy the new BoxV2 implementation contract (unless there is one already from a previous deployment), and upgrade the existing proxy to the new implementation.

**Beacon proxies**

You can also use this plugin to deploy an upgradeable beacon for your contract with the deployBeacon function, then deploy one or more beacon proxies that point to it by using the deployBeaconProxy function.

// scripts/create-box.js

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

async function main() {

const Box = await ethers.getContractFactory("Box");

const beacon = await upgrades.deployBeacon(Box);

await beacon.deployed();

console.log("Beacon deployed to:", beacon.address);

const box = await upgrades.deployBeaconProxy(beacon, Box, [42]);

await box.deployed();

console.log("Box deployed to:", box.address);

}

main();

Then, in another script, you can use the upgradeBeacon function to upgrade the beacon to a new version. When the beacon is upgraded, all of the beacon proxies that point to it will use the new contract implementation.

// scripts/upgrade-box.js

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

async function main() {

const BoxV2 = await ethers.getContractFactory("BoxV2");

await upgrades.upgradeBeacon(BEACON\_ADDRESS, BoxV2);

console.log("Beacon upgraded");

const box = BoxV2.attach(BOX\_ADDRESS);

}

main();

**Usage in tests**

You can also use the plugin’s functions from your Hardhat tests, in case you want to add tests for upgrading your contracts (which you should!). The API is the same as in scripts.

**Proxies**

const { expect } = require("chai");

describe("Box", function() {

it('works', async () => {

const Box = await ethers.getContractFactory("Box");

const BoxV2 = await ethers.getContractFactory("BoxV2");

const instance = await upgrades.deployProxy(Box, [42]);

const upgraded = await upgrades.upgradeProxy(instance.address, BoxV2);

const value = await upgraded.value();

expect(value.toString()).to.equal('42');

});

});

**Beacon proxies**

const { expect } = require("chai");

describe("Box", function() {

it('works', async () => {

const Box = await ethers.getContractFactory("Box");

const BoxV2 = await ethers.getContractFactory("BoxV2");

const beacon = await upgrades.deployBeacon(Box);

const instance = await upgrades.deployBeaconProxy(beacon, Box, [42]);

await upgrades.upgradeBeacon(beacon, BoxV2);

const upgraded = BoxV2.attach(instance.address);

const value = await upgraded.value();

expect(value.toString()).to.equal('42');

});

});

# Writing Upgradeable Contracts

When working with upgradeable contracts using OpenZeppelin Upgrades, there are a few minor caveats to keep in mind when writing your Solidity code.

It’s worth mentioning that these restrictions have their roots in how the Ethereum VM works, and apply to all projects that work with upgradeable contracts, not just OpenZeppelin Upgrades.

## Initializers

You can use your Solidity contracts with OpenZeppelin Upgrades without any modifications, except for their constructors. Due to a requirement of the proxy-based upgradeability system, no constructors can be used in upgradeable contracts. To learn about the reasons behind this restriction, head to [Proxies](https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies#the-constructor-caveat).

This means that, when using a contract with the OpenZeppelin Upgrades, you need to change its constructor into a regular function, typically named initialize, where you run all the setup logic:

// NOTE: Do not use this code snippet, it's incomplete and has a critical vulnerability!

pragma solidity ^0.6.0;

contract MyContract {

uint256 public x;

function initialize(uint256 \_x) public {

x = \_x;

}

}

However, while Solidity ensures that a constructor is called only once in the lifetime of a contract, a regular function can be called many times. To prevent a contract from being initialized multiple times, you need to add a check to ensure the initialize function is called only once:

// contracts/MyContract.sol

// SPDX-License-Identifier: MIT

pragma solidity ^0.6.0;

contract MyContract {

uint256 public x;

bool private initialized;

function initialize(uint256 \_x) public {

require(!initialized, "Contract instance has already been initialized");

initialized = true;

x = \_x;

}

}

Since this pattern is very common when writing upgradeable contracts, OpenZeppelin Contracts provides an Initializable base contract that has an initializer modifier that takes care of this:

// contracts/MyContract.sol

// SPDX-License-Identifier: MIT

pragma solidity ^0.6.0;

import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";

contract MyContract is Initializable {

uint256 public x;

function initialize(uint256 \_x) public initializer {

x = \_x;

}

}

Another difference between a constructor and a regular function is that Solidity takes care of automatically invoking the constructors of all ancestors of a contract. When writing an initializer, you need to take special care to manually call the initializers of all parent contracts. Note that the initializer modifier can only be called once even when using inheritance, so parent contracts should use the onlyInitializing modifier:

// contracts/MyContract.sol

// SPDX-License-Identifier: MIT

pragma solidity ^0.6.0;

import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";

contract BaseContract is Initializable {

uint256 public y;

function initialize() public onlyInitializing {

y = 42;

}

}

contract MyContract is BaseContract {

uint256 public x;

function initialize(uint256 \_x) public initializer {

BaseContract.initialize(); // Do not forget this call!

x = \_x;

}

}

### Using Upgradeable Smart Contract Libraries

Keep in mind that this restriction affects not only your contracts, but also the contracts you import from a library. Consider for example [ERC20](https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.7.3/contracts/token/ERC20/ERC20.sol) from OpenZeppelin Contracts: the contract initializes the token’s name and symbol in its constructor.

// @openzeppelin/contracts/token/ERC20/ERC20.sol

pragma solidity ^0.8.0;

...

contract ERC20 is Context, IERC20 {

...

string private \_name;

string private \_symbol;

constructor(string memory name\_, string memory symbol\_) {

\_name = name\_;

\_symbol = symbol\_;

}

...

}

This means you should not be using these contracts in your OpenZeppelin Upgrades project. Instead, make sure to use @openzeppelin/contracts-upgradeable, which is an official fork of OpenZeppelin Contracts that has been modified to use initializers instead of constructors. Take a look at what [ERC20Upgradeable](https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v4.7.3/contracts/token/ERC20/ERC20Upgradeable.sol) looks like in @openzeppelin/contracts-upgradeable:

// @openzeppelin/contracts-upgradeable/contracts/token/ERC20/ERC20Upgradeable.sol

pragma solidity ^0.8.0;

...

contract ERC20Upgradeable is Initializable, ContextUpgradeable, IERC20Upgradeable {

...

string private \_name;

string private \_symbol;

function \_\_ERC20\_init(string memory name\_, string memory symbol\_) internal onlyInitializing {

\_\_ERC20\_init\_unchained(name\_, symbol\_);

}

function \_\_ERC20\_init\_unchained(string memory name\_, string memory symbol\_) internal onlyInitializing {

\_name = name\_;

\_symbol = symbol\_;

}

...

}

Whether using OpenZeppelin Contracts or another smart contract library, always make sure that the package is set up to handle upgradeable contracts.

Learn more about OpenZeppelin Contracts Upgradeable in [Contracts: Using with Upgrades](https://docs.openzeppelin.com/contracts/4.x/upgradeable).

### Avoiding Initial Values in Field Declarations

Solidity allows defining initial values for fields when declaring them in a contract.

contract MyContract {

uint256 public hasInitialValue = 42; // equivalent to setting in the constructor

}

This is equivalent to setting these values in the constructor, and as such, will not work for upgradeable contracts. Make sure that all initial values are set in an initializer function as shown below; otherwise, any upgradeable instances will not have these fields set.

contract MyContract is Initializable {

uint256 public hasInitialValue;

function initialize() public initializer {

hasInitialValue = 42; // set initial value in initializer

}

}

|  |  |
| --- | --- |
|  | It is still ok to define constant state variables, because the compiler [does not reserve a storage slot for these variables](https://solidity.readthedocs.io/en/latest/contracts.html#constant-state-variables), and every occurrence is replaced by the respective constant expression. So the following still works with OpenZeppelin Upgrades: |

contract MyContract {

uint256 public constant hasInitialValue = 42; // define as constant

}

### Initializing the Implementation Contract

Do not leave an implementation contract uninitialized. An uninitialized implementation contract can be taken over by an attacker, which may impact the proxy. To prevent the implementation contract from being used, you should invoke the \_disableInitializers function in the constructor to automatically lock it when it is deployed:

/// @custom:oz-upgrades-unsafe-allow constructor

constructor() {

\_disableInitializers();

}

## Creating New Instances From Your Contract Code

When creating a new instance of a contract from your contract’s code, these creations are handled directly by Solidity and not by OpenZeppelin Upgrades, which means that **these contracts will not be upgradeable**.

For instance, in the following example, even if MyContract is deployed as upgradeable, the token contract created is not:

// contracts/MyContract.sol

// SPDX-License-Identifier: MIT

pragma solidity ^0.6.0;

import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";

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

contract MyContract is Initializable {

ERC20 public token;

function initialize() public initializer {

token = new ERC20("Test", "TST"); // This contract will not be upgradeable

}

}

If you would like the ERC20 instance to be upgradeable, the easiest way to achieve that is to simply accept an instance of that contract as a parameter, and inject it after creating it:

// contracts/MyContract.sol

// SPDX-License-Identifier: MIT

pragma solidity ^0.6.0;

import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";

import "@openzeppelin/contracts-upgradeable/token/ERC20/IERC20Upgradeable.sol";

contract MyContract is Initializable {

IERC20Upgradeable public token;

function initialize(IERC20Upgradeable \_token) public initializer {

token = \_token;

}

}

## Potentially Unsafe Operations

When working with upgradeable smart contracts, you will always interact with the contract instance, and never with the underlying logic contract. However, nothing prevents a malicious actor from sending transactions to the logic contract directly. This does not pose a threat, since any changes to the state of the logic contracts do not affect your contract instances, as the storage of the logic contracts is never used in your project.

There is, however, an exception. If the direct call to the logic contract triggers a selfdestruct operation, then the logic contract will be destroyed, and all your contract instances will end up delegating all calls to an address without any code. This would effectively break all contract instances in your project.

A similar effect can be achieved if the logic contract contains a delegatecall operation. If the contract can be made to delegatecall into a malicious contract that contains a selfdestruct, then the calling contract will be destroyed.

As such, it is not allowed to use either selfdestruct or delegatecall in your contracts.

## Modifying Your Contracts

When writing new versions of your contracts, either due to new features or bug fixing, there is an additional restriction to observe: you cannot change the order in which the contract state variables are declared, nor their type. You can read more about the reasons behind this restriction by learning about our [Proxies](https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies).

|  |  |
| --- | --- |
|  | Violating any of these storage layout restrictions will cause the upgraded version of the contract to have its storage values mixed up, and can lead to critical errors in your application. |

This means that if you have an initial contract that looks like this:

contract MyContract {

uint256 private x;

string private y;

}

Then you cannot change the type of a variable:

contract MyContract {

string private x;

string private y;

}

Or change the order in which they are declared:

contract MyContract {

string private y;

uint256 private x;

}

Or introduce a new variable before existing ones:

contract MyContract {

bytes private a;

uint256 private x;

string private y;

}

Or remove an existing variable:

contract MyContract {

string private y;

}

If you need to introduce a new variable, make sure you always do so at the end:

contract MyContract {

uint256 private x;

string private y;

bytes private z;

}

Keep in mind that if you rename a variable, then it will keep the same value as before after upgrading. This may be the desired behavior if the new variable is semantically the same as the old one:

contract MyContract {

uint256 private x;

string private z; // starts with the value from `y`

}

And if you remove a variable from the end of the contract, note that the storage will not be cleared. A subsequent update that adds a new variable will cause that variable to read the leftover value from the deleted one.

contract MyContract {

uint256 private x;

}

Then upgraded to:

contract MyContract {

uint256 private x;

string private z; // starts with the value from `y`

}

Note that you may also be inadvertently changing the storage variables of your contract by changing its parent contracts. For instance, if you have the following contracts:

contract A {

uint256 a;

}

contract B {

uint256 b;

}

contract MyContract is A, B {}

Then modifying MyContract by swapping the order in which the base contracts are declared, or introducing new base contracts, will change how the variables are actually stored:

contract MyContract is B, A {}

You also cannot add new variables to base contracts, if the child has any variables of its own. Given the following scenario:

contract Base {

uint256 base1;

}

contract Child is Base {

uint256 child;

}

If Base is modified to add an extra variable:

contract Base {

uint256 base1;

uint256 base2;

}

Then the variable base2 would be assigned the slot that child had in the previous version. A workaround for this is to declare unused variables or storage gaps in base contracts that you may want to extend in the future, as a means of "reserving" those slots. Note that this trick does not involve increased gas usage.

### Storage Gaps

Storage gaps are a convention for reserving storage slots in a base contract, allowing future versions of that contract to use up those slots without affecting the storage layout of child contracts.

To create a storage gap, declare a fixed-size array in the base contract with an initial number of slots. This can be an array of uint256 so that each element reserves a 32 byte slot. Use the name gap or a name starting with gap\_ for the array so that OpenZeppelin Upgrades will recognize the gap:

contract Base {

uint256 base1;

uint256[49] \_\_gap;

}

contract Child is Base {

uint256 child;

}

If Base is later modified to add extra variable(s), reduce the appropriate number of slots from the storage gap, keeping in mind [Solidity’s rules on how contiguous items are packed](https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html#layout-of-state-variables-in-storage). For example:

contract Base {

uint256 base1;

uint256 base2; // 32 bytes

uint256[48] \_\_gap;

}

Or:

contract Base {

uint256 base1;

address base2; // 20 bytes

uint256[48] \_\_gap; // array always starts at a new slot

}

Or:

contract Base {

uint256 base1;

uint128 base2a; // 16 bytes

uint128 base2b; // 16 bytes - continues from the same slot as above

uint256[48] \_\_gap;

}

To help determine the proper storage gap size in the new version of your contract, you can simply attempt an upgrade using upgradeProxy or just run the validations with validateUpgrade (see docs for [Hardhat](https://docs.openzeppelin.com/upgrades-plugins/1.x/api-hardhat-upgrades) or [Truffle](https://docs.openzeppelin.com/upgrades-plugins/1.x/api-truffle-upgrades)). If a storage gap is not being reduced properly, you will see an error message indicating the expected size of the storage gap.

# Proxy Upgrade Pattern

This article describes the "unstructured storage" proxy pattern, the fundamental building block of OpenZeppelin Upgrades.

|  |  |
| --- | --- |
|  | For a more in depth read, please see [our proxy-patterns blog post](https://blog.openzeppelin.com/proxy-patterns/), which discusses the need for proxies, goes into more technical detail on the subject, elaborates on other possible proxy patterns that were considered for OpenZeppelin Upgrades, and more. |

## Why Upgrade a Contract?

By design, smart contracts are immutable. On the other hand, software quality heavily depends on the ability to upgrade and patch source code in order to produce iterative releases. Even though blockchain based software profits significantly from the technology’s immutability, still a certain degree of mutability is needed for bug fixing and potential product improvements. OpenZeppelin Upgrades solves this apparent contradiction by providing an easy to use, simple, robust, and opt-in upgrade mechanism for smart contracts that can be controlled by any type of governance, be it a multi-sig wallet, a simple address or a complex DAO.

## Upgrading via the Proxy Pattern

The basic idea is using a proxy for upgrades. The first contract is a simple wrapper or "proxy" which users interact with directly and is in charge of forwarding transactions to and from the second contract, which contains the logic. The key concept to understand is that the logic contract can be replaced while the proxy, or the access point is never changed. Both contracts are still immutable in the sense that their code cannot be changed, but the logic contract can simply be swapped by another contract. The wrapper can thus point to a different logic implementation and in doing so, the software is "upgraded".

User ---- tx ---> Proxy ----------> Implementation\_v0

|

------------> Implementation\_v1

|

------------> Implementation\_v2

## Proxy Forwarding

The most immediate problem that proxies need to solve is how the proxy exposes the entire interface of the logic contract without requiring a one to one mapping of the entire logic contract’s interface. That would be difficult to maintain, prone to errors, and would make the interface itself not upgradeable. Hence, a dynamic forwarding mechanism is required. The basics of such a mechanism are presented in the code below:

// This code is for "illustration" purposes. To implement this functionality in production it

// is recommended to use the `Proxy` contract from the `@openzeppelin/contracts` library.

// https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.8.2/contracts/proxy/Proxy.sol

assembly {

// (1) copy incoming call data

calldatacopy(0, 0, calldatasize())

// (2) forward call to logic contract

let result := delegatecall(gas(), implementation, 0, calldatasize(), 0, 0)

// (3) retrieve return data

returndatacopy(0, 0, returndatasize())

// (4) forward return data back to caller

switch result

case 0 {

revert(0, returndatasize())

}

default {

return(0, returndatasize())

}

}

This code can be put in the [fallback function](https://docs.soliditylang.org/en/latest/contracts.html#fallback-function) of a proxy, and will forward any call to any function with any set of parameters to the logic contract without it needing to know anything in particular of the logic contract’s interface. In essence, (1) the calldata is copied to memory, (2) the call is forwarded to the logic contract, (3) the return data from the call to the logic contract is retrieved, and (4) the returned data is forwarded back to the caller.

A very important thing to note is that the code makes use of the EVM’s delegatecall opcode which executes the callee’s code in the context of the caller’s state. That is, the logic contract controls the proxy’s state and the logic contract’s state is meaningless. Thus, the proxy doesn’t only forward transactions to and from the logic contract, but also represents the pair’s state. The state is in the proxy and the logic is in the particular implementation that the proxy points to.

## Unstructured Storage Proxies

A problem that quickly comes up when using proxies has to do with the way in which variables are stored in the proxy contract. Suppose that the proxy stores the logic contract’s address in its only variable address public \_implementation;. Now, suppose that the logic contract is a basic token whose first variable is address public \_owner. Both variables are 32 byte in size, and as far as the EVM knows, occupy the first slot of the resulting execution flow of a proxied call. When the logic contract writes to \_owner, it does so in the scope of the proxy’s state, and in reality writes to \_implementation. This problem can be referred to as a "storage collision".

|Proxy |Implementation |

|--------------------------|-------------------------|

|address \_implementation |address \_owner | <=== Storage collision!

|... |mapping \_balances |

| |uint256 \_supply |

| |... |

There are many ways to overcome this problem, and the "unstructured storage" approach which OpenZeppelin Upgrades implements works as follows. Instead of storing the \_implementation address at the proxy’s first storage slot, it chooses a pseudo random slot instead. This slot is sufficiently random, that the probability of a logic contract declaring a variable at the same slot is negligible. The same principle of randomizing slot positions in the proxy’s storage is used in any other variables the proxy may have, such as an admin address (that is allowed to update the value of \_implementation), etc.

|Proxy |Implementation |

|--------------------------|-------------------------|

|... |address \_owner |

|... |mapping \_balances |

|... |uint256 \_supply |

|... |... |

|... | |

|... | |

|... | |

|... | |

|address \_implementation | | <=== Randomized slot.

|... | |

|... | |

An example of how the randomized storage is achieved, following [EIP 1967](http://eips.ethereum.org/EIPS/eip-1967):

bytes32 private constant implementationPosition = bytes32(uint256(

keccak256('eip1967.proxy.implementation')) - 1

));

As a result, a logic contract doesn’t need to care about overwriting any of the proxy’s variables. Other proxy implementations that face this problem usually imply having the proxy know about the logic contract’s storage structure and adapt to it, or instead having the logic contract know about the proxy’s storage structure and adapt to it. This is why this approach is called "unstructured storage"; neither of the contracts needs to care about the structure of the other.

## Storage Collisions Between Implementation Versions

As discussed, the unstructured approach avoids storage collisions between the logic contract and the proxy. However, storage collisions between different versions of the logic contract can occur. In this case, imagine that the first implementation of the logic contract stores address public \_owner at the first storage slot and an upgraded logic contract stores address public \_lastContributor at the same first slot. When the updated logic contract attempts to write to the \_lastContributor variable, it will be using the same storage position where the previous value for \_owner was being stored, and overwrite it!

Incorrect storage preservation:

|Implementation\_v0 |Implementation\_v1 |

|--------------------|-------------------------|

|address \_owner |address \_lastContributor | <=== Storage collision!

|mapping \_balances |address \_owner |

|uint256 \_supply |mapping \_balances |

|... |uint256 \_supply |

| |... |

Correct storage preservation:

|Implementation\_v0 |Implementation\_v1 |

|--------------------|-------------------------|

|address \_owner |address \_owner |

|mapping \_balances |mapping \_balances |

|uint256 \_supply |uint256 \_supply |

|... |address \_lastContributor | <=== Storage extension.

| |... |

The unstructured storage proxy mechanism doesn’t safeguard against this situation. It is up to the user to have new versions of a logic contract extend previous versions, or otherwise guarantee that the storage hierarchy is always appended to but not modified. However, OpenZeppelin Upgrades detects such collisions and warns the developer appropriately.

## The Constructor Caveat

In Solidity, code that is inside a constructor or part of a global variable declaration is not part of a deployed contract’s runtime bytecode. This code is executed only once, when the contract instance is deployed. As a consequence of this, the code within a logic contract’s constructor will never be executed in the context of the proxy’s state. To rephrase, proxies are completely oblivious to the existence of constructors. It’s simply as if they weren’t there for the proxy.

The problem is easily solved though. Logic contracts should move the code within the constructor to a regular 'initializer' function, and have this function be called whenever the proxy links to this logic contract. Special care needs to be taken with this initializer function so that it can only be called once, which is one of the properties of constructors in general programming.

This is why when we create a proxy using OpenZeppelin Upgrades, you can provide the name of the initializer function and pass parameters.

To ensure that the initialize function can only be called once, a simple modifier is used. OpenZeppelin Upgrades provides this functionality via a contract that can be extended:

// contracts/MyContract.sol

// SPDX-License-Identifier: MIT

pragma solidity ^0.6.0;

import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";

contract MyContract is Initializable {

function initialize(

address arg1,

uint256 arg2,

bytes memory arg3

) public payable initializer {

// "constructor" code...

}

}

Notice how the contract extends Initializable and implements the initializer provided by it.

## Transparent Proxies and Function Clashes

As described in the previous sections, upgradeable contract instances (or proxies) work by delegating all calls to a logic contract. However, the proxies need some functions of their own, such as upgradeTo(address) to upgrade to a new implementation. This begs the question of how to proceed if the logic contract also has a function named upgradeTo(address): upon a call to that function, did the caller intend to call the proxy or the logic contract?

|  |  |
| --- | --- |
|  | Clashing can also happen among functions with different names. Every function that is part of a contract’s public ABI is identified, at the bytecode level, by a 4-byte identifier. This identifier depends on the name and arity of the function, but since it’s only 4 bytes, there is a possibility that two different functions with different names may end up having the same identifier. The Solidity compiler tracks when this happens within the same contract, but not when the collision happens across different ones, such as between a proxy and its logic contract. Read [this article](https://medium.com/nomic-labs-blog/malicious-backdoors-in-ethereum-proxies-62629adf3357) for more info on this. |

The way OpenZeppelin Upgrades deals with this problem is via the transparent proxy pattern. A transparent proxy will decide which calls are delegated to the underlying logic contract based on the caller address (i.e., the msg.sender):

* If the caller is the admin of the proxy (the address with rights to upgrade the proxy), then the proxy will **not** delegate any calls, and only answer any messages it understands.
* If the caller is any other address, the proxy will **always** delegate a call, no matter if it matches one of the proxy’s functions.

Assuming a proxy with an owner() and an upgradeTo() function, that delegates calls to an ERC20 contract with an owner() and a transfer() function, the following table covers all scenarios:

| **msg.sender** | **owner()** | **upgradeto()** | **transfer()** |
| --- | --- | --- | --- |
| Owner | returns proxy.owner() | returns proxy.upgradeTo() | fails |
| Other | returns erc20.owner() | fails | returns erc20.transfer() |

Fortunately, OpenZeppelin Upgrades accounts for this situation, and creates an intermediary ProxyAdmin contract that is in charge of all the proxies you create via the Upgrades plugins. Even if you call the deploy command from your node’s default account, the ProxyAdmin contract will be the actual admin of all your proxies. This means that you will be able to interact with the proxies from any of your node’s accounts, without having to worry about the nuances of the transparent proxy pattern. Only advanced users that create proxies from Solidity need to be aware of the transparent proxies pattern.

## Summary

Any developer using upgradeable contracts should be familiar with proxies in the ways that are described in this article. In the end, the concept is very simple, and OpenZeppelin Upgrades is designed to encapsulate all the proxy mechanics in a way that the amount of things you need to keep in mind when developing projects are reduced to an absolute minimum. It all comes down to the following list:

* Have a basic understanding of what a proxy is
* Always extend storage instead of modifying it
* Make sure your contracts use initializer functions instead of constructors

Furthermore, the OpenZeppelin Upgrades will let you know when something goes wrong with one of the items in this list.