J Jeffrey Scholz Dec 28, 2023 2 min read

The Fallback Extension Pattern

The fallback-extension pattern is a simple way to circumvent the 24kb smart contract size limit.

Suppose we have functions foo() and bar() in our primary contract and wish to add baz() but cannot due to a lack of space.

We add a fallback function to our primary smart contract that delegates unknown function calls to an *extension* contract similar to how a proxy works.

We put baz() in the *extension* contract. When we call "baz()" on the main contract, it will not match any of the function selectors in the *primary* contract, and the thus trigger the fallback function. Then, baz() will be delegatecalled in the *extension* contract.

Ensuring an identical storage layout

For this pattern to work, both the primary contract and the extension contract need an identical storage layout. A simple way to accomplish this is to put *all* the storage variables (no exceptions!) into a single contract. Then both the primary contract and the extension contract inherit from that. Here is an example:

```
contract StorageContract {
    uint256 internal a;
    uint256 internal b;
}

contract Primary is StorageContract {
    address immutable extensionDelegate; // NOT A STORAGE VARIABLE

    constructor(address extensionDelegate_) {
        extension = extensionDelegate_;
    }

    function foo() external returns (uint256) {
        return a + b;
    }

    function bar(uint256 a_, uint256 b_) external {
        a = a_;
        b = b_;
    }

    fallback(bytes calldata data) external payable returns (bytes memory retdata) {
        (bool ok, bytes memory retdata) = extensionDelegate.delegatecall(data);
        if (ok) {
            return retdata;
        }
        assembly {
            revert(retdata)
        }
    }
}

contract Extension is StorageContract {
    function baz() external returns (uint256) {
        return a * b;
    }
}
```

Changing the extension

In the example above, the address of the extension contract is stored in an immutable variable. We could add an additional storage variable to the storage contract which holds the address to the extension, then update the extension address when we want to change some of the functionality of the contract.

This approach is not recommended — if upgradeability is needed, then it's better to use an established proxy pattern. Additionally, reading that extension address variable from storage will cost an extra 2,100 gas.

Care must be taken to avoid function selector collisions

There is an approximately 1 in 4 million chance two random functions will have the same selector. However, due to the birthday paradox, this probability increases rapidly when we have *n* function selectors and only one collision is needed to cause unwanted behavior. There are no tools for this, the developer must manually check the function selectors.

Gas Considerations

The extension itself can follow this pattern and send it to another delegate. In fact, there is no in-principal limit to how many times we do this.
However, each "hop" adds an extra 2,600 gas (the minimum gas required to issue a CALL or DELEGATECALL to a new address), so the cost can be substantial if the chain is long.

Because functions in the extension cost an extra 2,600 gas, we want to put rarely-used functions in the extension, or functions that are intended to be mostly called from off-chain where gas doesn't matter.

Use EIP 2930 in combination with this pattern

Using access list transactions with this pattern will save 100 gas when calling functions in the extension. In general, if an Ethereum transaction contains a cross contract call or delegatecall, an access list transaction should be used.

Using a fallback-extension as an implementation contract for upgradeable proxies

This pattern can be used with regular proxies. That is, a proxy contract can delegate to an implementation contract which then delegates to an extension. Keep in mind that tooling for upgrades (like Openzeppelin upgrade tools) are not designed to work with the extension fallback pattern and might not catch upgrade related issues.

Learn More with RareSkills

Please see our Solidity Bootcamp to learn more.