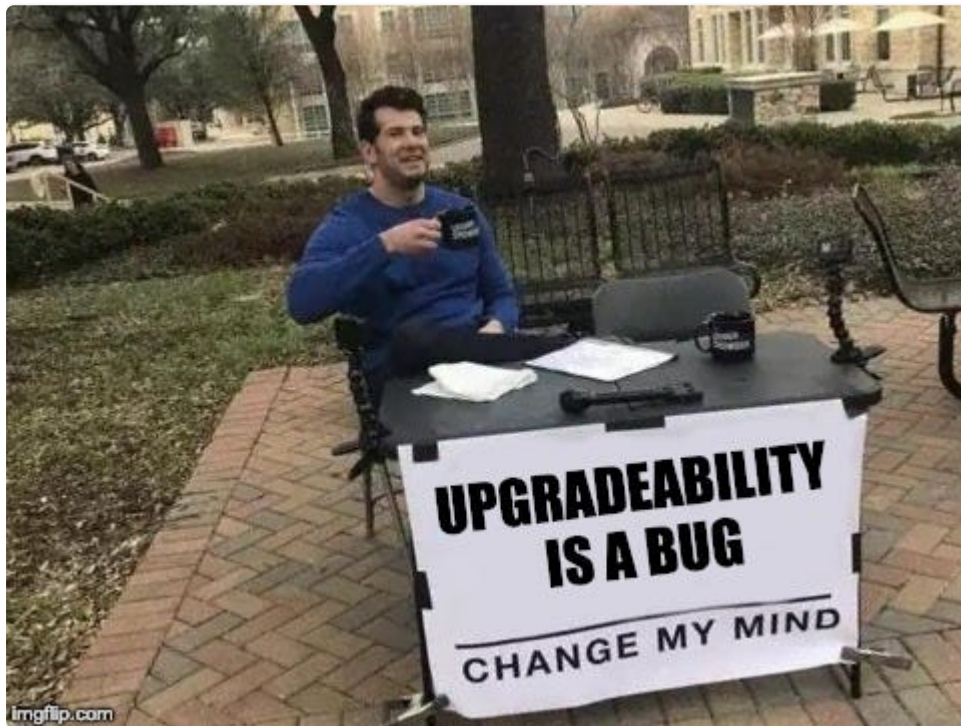# Lesson4

Upgradability Background

The great advantage to smart contract is that they're immutable, no one can hack them or change their terms once they are deployed

The great drawback to smart contracts is that they're immutable, you can't fix them once they're deployed.

The problems we need to solve are

1. How to change the functionality in the contract
2. How to migrate data if necessary

We will look at some of the patterns used to allow upgradability
I have taken examples from this guide, the guide applies to truffle or hardhat.
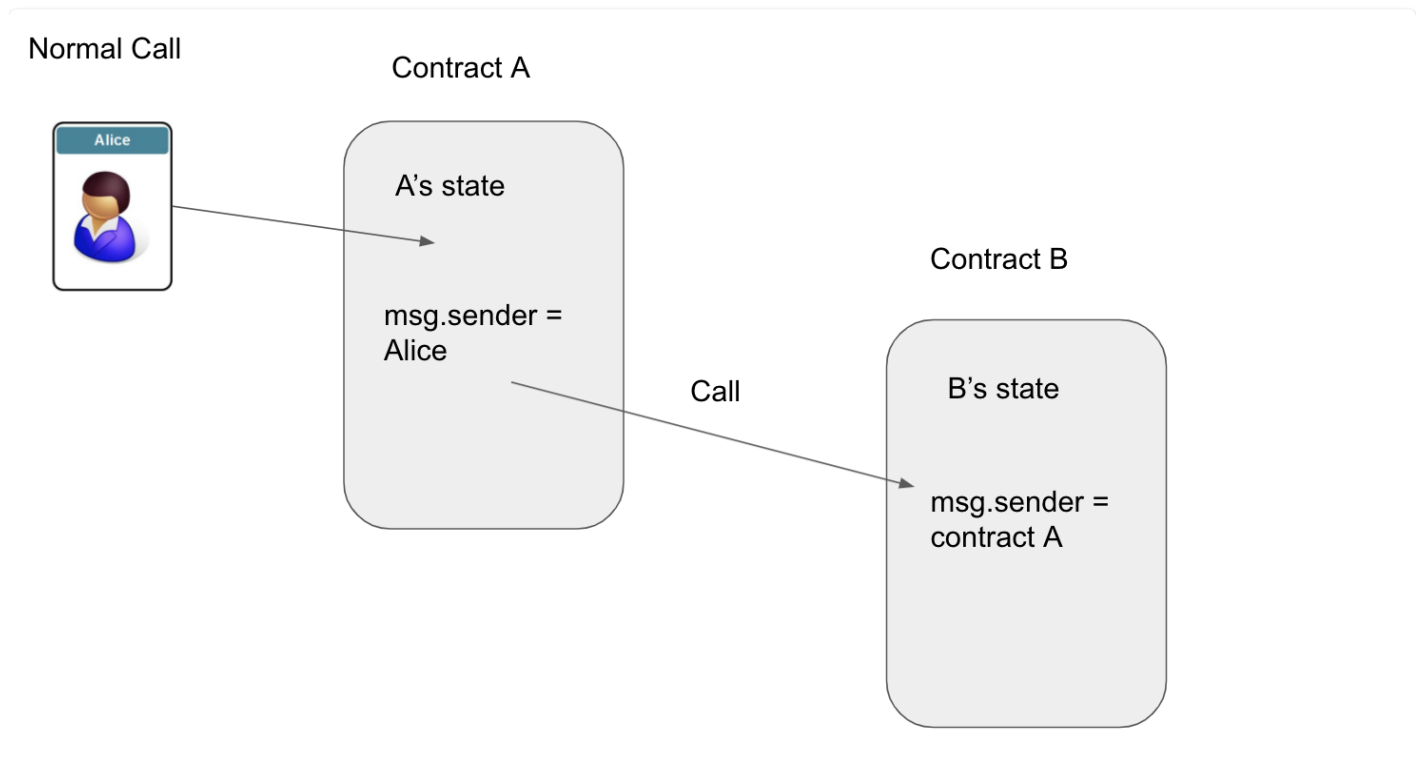See State of upgrades

Digression - Message Calls

There are a number of ways for contracts to call each other
See : [Read the Docs : Message Calls](#)

Message calls have a source (this contract), a target (the other contract), data payload, Ether, gas and return data.

The other contract gets a fresh context to work in, its own contract state as you would expect.

Normal Call

Contract A

Alice

A's state

msg.sender = Alice

Call

Contract B

B's state

msg.sender = contract A

Delegate Call

There is a special type of call, Delegate Call which behaves differently in that it executes in the context of the calling contract (and msg.sender and msg.value do not change)

Delegate Call

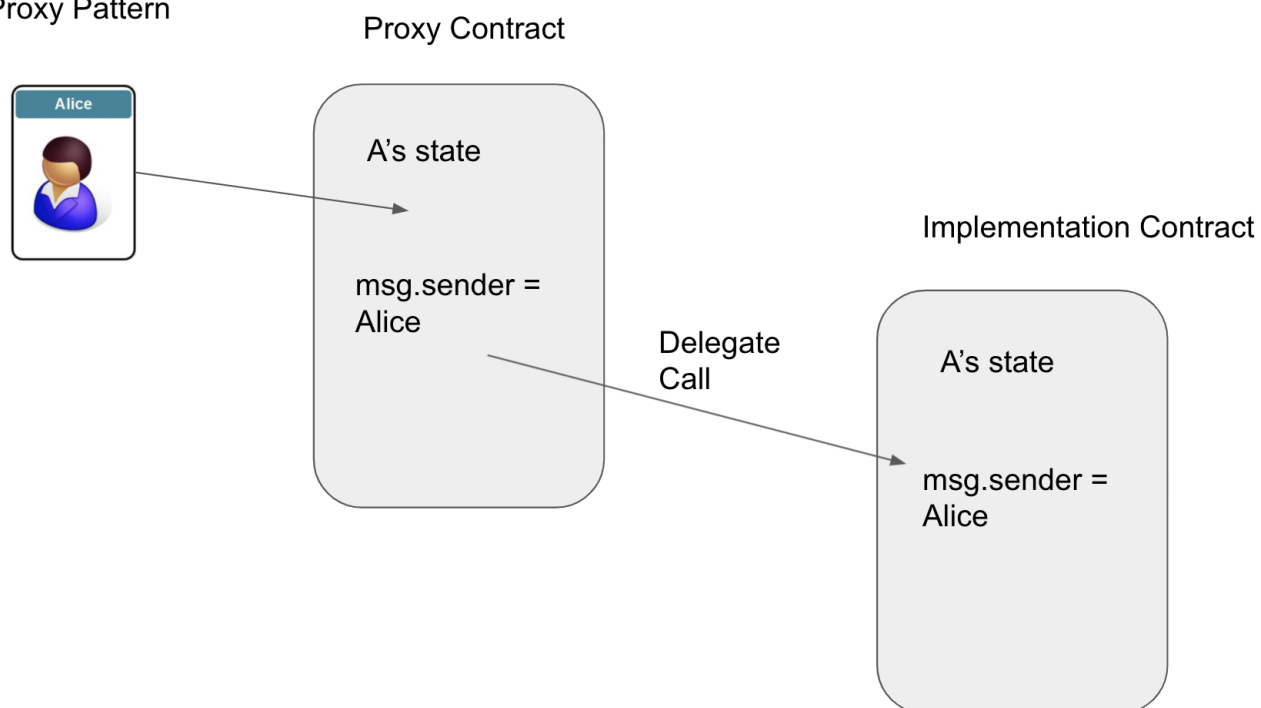Alice → Contract A

Contract A

A's state

msg.sender = Alice

Delegate Call →

Contract B

A's state

msg.sender = Alice
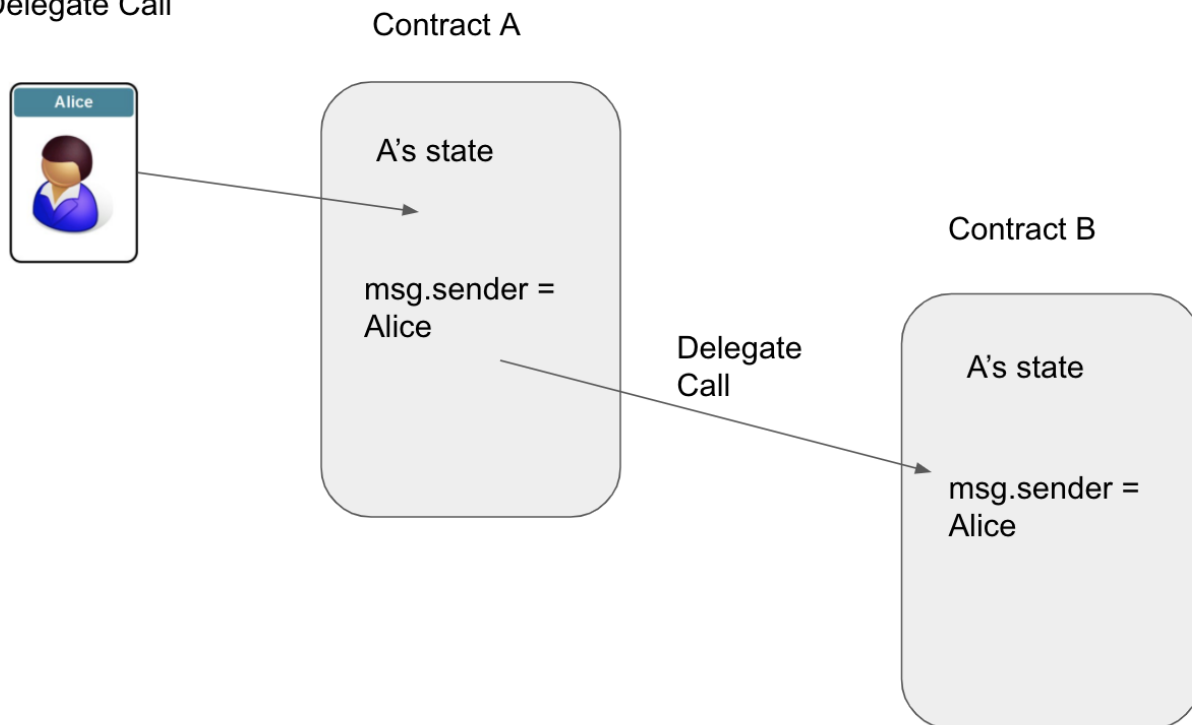
Another way to look at this is to think of contract A loading and executing contract B's code.

Proxy patterns

See EIP 897

Proxy Pattern

Alice → Proxy Contract

Proxy Contract

A's state

msg.sender = Alice

Delegate Call →

Implementation Contract

A's state

msg.sender = Alice

We have a proxy contract and an implementation contract
Users always interact with the Proxy contract and need not be aware of the

implementation contract.

It is also possible for multiple proxy contracts to use the same implementation contract (This can be a way of deploying multiple instances of a contract cheaply, see EIP1167).

Open Zeppelin Clones Library

We are cheaply creating a proxy for another contract
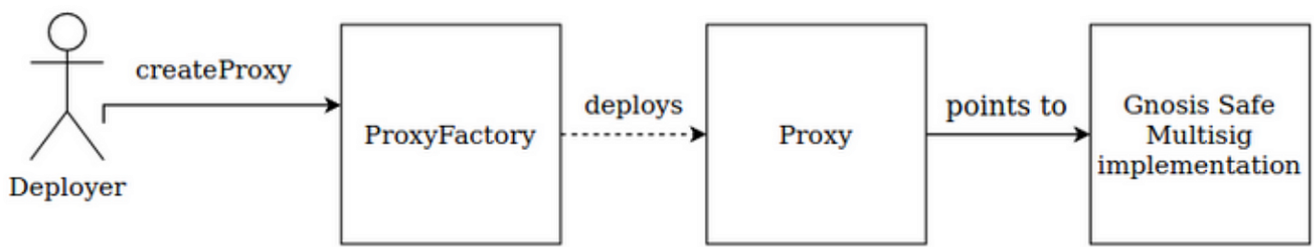
Deploying a Gnosis Safe Multisig wallet with the ProxyFactory contract

This function uses the create2 opcode and a `salt` to deterministically deploy the clone. Using the same `implementation` and `salt` multiple time will revert, since the clones cannot be deployed twice at the same address.

Approaches to Upgradability

Upgrading is an anti pattern - don't do it

There **is** an argument for this approach, it favours decentralisation
See [Upgradability is a bug](#)

- Smart contracts are useful because they're trustless.
- Immutability is a critical feature to achieve trustlessness.
- Upgradeability undermines a contract's immutability.
- Therefore, upgradeability is a bug.

From the article :
"We strongly advise against the use of these patterns for upgradable smart contracts. Both strategies have the potential for flaws, significantly increase complexity, and introduce bugs, and ultimately decrease trust in your smart contract. Strive for simple, immutable, and secure contracts rather than importing a significant amount of code to postpone feature and security issues."

It may be sufficient to parameterise your contract and adjust those parameters instead of upgrading
For example Maker DAO's stability fee, or a farming reward rate that can be adjusted by the an administrator (or a DAO, or some governance mechanism)

Migrate the data manually

Deploy your V2 contract, and migrate manually any existing data

Advantages

Conceptually simple.
No reliance on libraries.

Disadvantages

Can be difficult and costly (gas and time) in practice, and if the amount of data to migrate is large, it may hit gas limits when migrating.

Use a Registry contract

A registry (similar to ENS) holds the address of the latest version of the contract. DApps should read this registry to get the correct address

Advantages

Simple to implement

Disadvantages

We rely on the DApp code to choose the correct contract
There is trust involved in the developers, not to switch out a contract with reasonable terms for an unfavourable one.
"Keep in mind that users of your smart contract don't trust you, and that's why you wrote a smart contract in the first place."

This doesn't solve the data migration problem

Separate code into function and data contracts

Maybe simple to implement
Partially solves data migration

It is difficult to get the security implemented correctly
Fails if your data contract needs to change.

Choose an function at runtime in other contracts or libraries

This is moving towards the Proxy patterns and the Diamond pattern
Essentially the Strategy Pattern
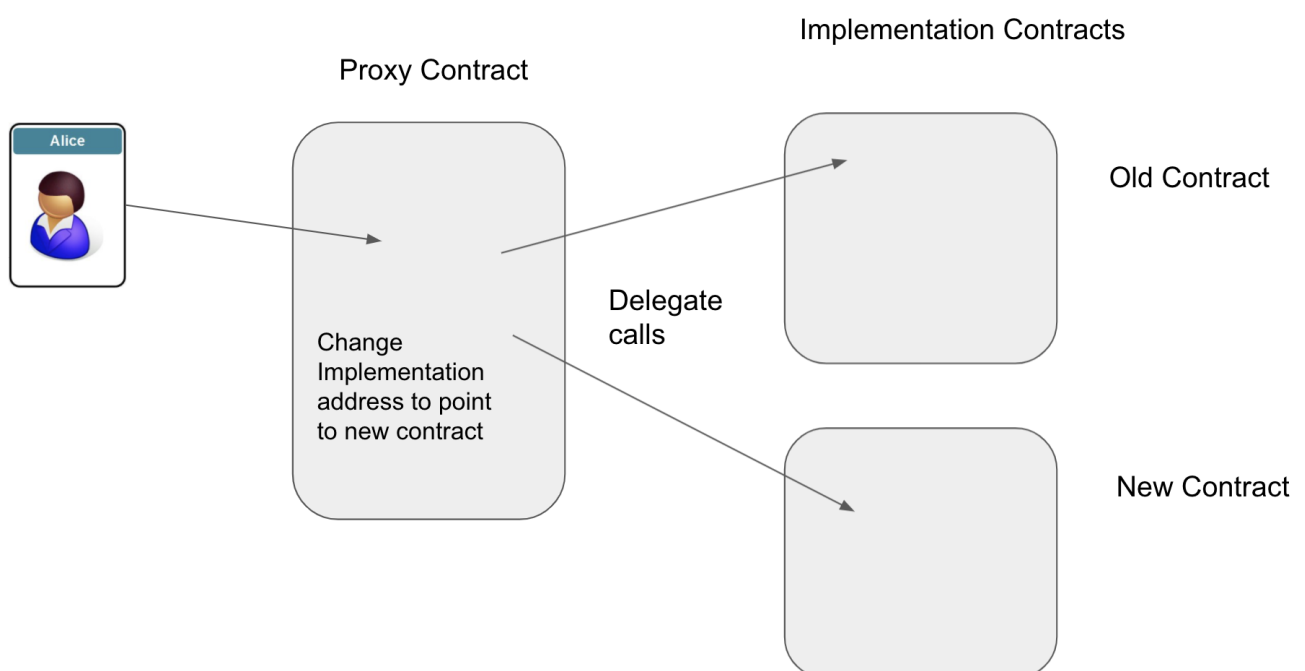Compound use this approach with their interest rate model
A variant of this is the use of pluggable modules
such as in Gnosis Safe
The module approach is additive, if there is a bug in the core code this approach won't fix it.

Using Proxy contracts to upgrade

```
contract AdminUpgradeableProxy {                        language—none
    address implementation;
    address admin;

    fallback() external payable {
        // delegate here
    }

    function upgrade(address newImplementation) external {
        require(msg.sender == admin);
        implementation = newImplementation;
    }
}
```

This can be open to vulnerabilities, instead the
Transparent Proxy Contract can be used

```
contract TransparentAdminUpgradeableProxy {             language—none
    address implementation;
    address admin;

    fallback() external payable {
        require(msg.sender != admin);
        // delegate here
    }

    function upgrade(address newImplementation) external {
        if (msg.sender != admin) fallback();
        implementation = newImplementation;
    }
}
```

This pattern is widely used, but comes at a cost because of the additional lookup
of the implementation address and admin address

A cheaper and more recent alternative is the
universal upgradeable proxy standard (UUPS)

In this pattern, the upgrade logic is placed in the implementation contract.

```
contract UUPSProxy {                                    language-none
    address implementation;

    fallback() external payable {
       // delegate here
    }
}

abstract contract UUPSProxiable {
    address implementation;
    address admin;

    function upgrade(address newImplementation) external {
        require(msg.sender == admin);
        implementation = newImplementation;
    }
}
```

Cost Comparison

	Transparent	UUPS
Proxy Deployment	740k + 480k ProxyAdmin	390k
Implementation Deployment	+ 0	+ 320k
Runtime Overhead	7.3k	4.9k

Overwriting data

But what about the data is there a possibility of overwriting data in our proxy contract unintentionally ?

Layout of variables in storage

From [Solidity Documentation](#)

State variables of contracts are stored in storage in a compact way such that multiple values sometimes use the same storage slot. Except for dynamically-sized arrays and mappings, data is stored contiguously item after item starting with the first state variable, which is stored in slot 0. For each variable, a size in bytes is determined according to its type.

Mappings and dynamic arrays

Due to their unpredictable size, mappings and dynamically-sized array types cannot be stored "in between" the state variables preceding and following them. Instead, they are considered to occupy only 32 bytes with regards to the rules above and the elements they contain are stored starting at a different storage slot that is computed using a Keccak-256 hash.

slot	\<inherited contract\>.variable (bytes)		
	FiatTokenV2_1 \<\<Contract\>\> 0xa2327a938febf5fec13bacfb16ae10ecbc4cbdcf		
	type: \<inherited contract\>.variable (bytes)		
0	unallocated (12)	address: Ownable._owner (20)	
1	unallocated (11)	bool: Pausable.paused (1)	address: Pausable.pauser (20)
2	unallocated (12)	address: Blacklistable.blacklister (20)	
3	mapping(address=>bool): Blacklistable.blacklisted (32)		
4	string: FiatTokenV1.name (32)		
5	string: FiatTokenV1.symbol (32)		
6	unallocated (31)	uint8: FiatTokenV1.decimals (1)	
7	string: FiatTokenV1.currency (32)		
8	unallocated (11)	bool: FiatTokenV1.initialized (1)	address: FiatTokenV1.masterMinter (20)
9	mapping(address=>uint256): FiatTokenV1.balances (32)		
10	mapping(address=>mapping(address=>uint256)): FiatTokenV1.allowed (32)		
11	uint256: FiatTokenV1.totalSupply_ (32)		
12	mapping(address=>bool): FiatTokenV1.minters (32)		
13	mapping(address=>uint256): FiatTokenV1.minterAllowed (32)		
14	unallocated (12)	address: Rescuable._rescuer (20)	
15	bytes32: EIP712Domain.DOMAIN_SEPARATOR (32)		
16	mapping(address=>mapping(bytes32=>bool)): EIP3009._authorizationStates (32)		
17	mapping(address=>uint256): EIP2612._permitNonces (32)		
18	unallocated (31)	uint8: FiatTokenV2._initializedVersion (1)	

If we have our proxy and implementation like this

```
contract UUPSProxy {
    address implementation;

    fallback() external payable {
        // delegate here
    }
```

```solidity
}

abstract contract UUPSProxiable {
    uint256 counter;
    address implementation;
    address admin;

function foo() public {
    counter ++;
}

    function upgrade(address newImplementation) external {
        require(msg.sender == admin);
        implementation = newImplementation;
    }
}
```

If our implementation contract writes to the slot that it sees as counter, then it will overwrite the implementation variable.

To prevent this we use Unstructured storage
Open Zeppelin puts the implementation address at a 'random' address in storage

```
bytes32 private constant implementationPosition =
bytes32(uint256(
  keccak256('eip1967.proxy.implementation')) - 1
));
```

This solves the problem for the implementation variable, but what about our other values in storage ?

Say our versions of the implementation contracts looks like this

Implementation_v0	Implementation_v1
address owner	address lastContributor
mapping balances	address owner
uint256 supply	mapping balances
...	uint256 supply
	...

Then we will have a storage collision when writing to lastContributor

The correct approach is only to append to the storage when we upgrade

Implementation_v0	Implementation_v1
address owner	address owner
mapping balances	mapping balances
uint256 supply	uint256 supply
...	address lastContributor
	...

Question – What about the constructor in the implementation contract ?

For upgradeable contracts we use an initialiser function rather than the constructor.

Eternal Storage

An alternative to the above is to split up our data types and store them in mappings

```
// Sample code, do not use in production!                    language—none
contract EternalStorage {
    mapping(bytes32 => uint256) internal uintStorage;
    mapping(bytes32 => string) internal stringStorage;
    mapping(bytes32 => address) internal addressStorage;
    mapping(bytes32 => bytes) internal bytesStorage;
    mapping(bytes32 => bool) internal boolStorage;
    mapping(bytes32 => int256) internal intStorage;
}

contract Box is EternalStorage {
    function setValue(uint256 newValue) public {
        uintStorage['value'] = newValue;
    }
}
```

I include this for completeness but do not recommend it.

Using the UUPS plugin

What the plugins do

Both plugins provide two main functions, deployProxy and upgradeProxy, which take care of managing upgradeable deployments of your contracts. In the case of deployProxy, this means:

- Validate that the implementation is upgrade safe.
- Deploy a proxy admin for your project.
- Deploy the implementation contract.
- Create and initialize the proxy contract.

And when you call upgradeProxy:

- Validate that the new implementation is upgrade safe and is compatible with the previous one.
- Check if there is an implementation contract deployed with the same bytecode, and deploy one if not.
- Upgrade the proxy to use the new implementation contract.

Writing your contract

1. You need to include an initialising function and ensure it is called only once. Open Zeppelin provide a base contract to do this for you, you just need to inherit from it.

```
// contracts/MyContract.sol
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.0;

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

contract MyContract is Initializable,UUPSUpgradeable {
    uint256 public x;

    function initialize(uint256 _x) public initializer {
        x = _x;
```

```
        }
    }
```

Since this is not a constructor, the constructors of parent contracts will not be called, you will need to do this manually.

This also applies to initial values applied to variables (but constant is ok)
e.g.

```
contract MyContract {                                    language-none
    uint256 public hasInitialValue = 42;
    // equivalent to setting in the constructor
}
```

2. If you are using standard Open Zeppelin libraries, you need to switch to their upgradeable versions. It is recommended that your new version of the implementation contract inherits from your previous version
3. Use the plugins to deploy and upgrade your contracts for you
   Instead of the usual migration scripts in hardhat / truffle you will need something like this

Hardhat

```
// In Hardhat config                                    language-none
require('@openzeppelin/hardhat-upgrades');


// 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();
```

Truffle

```
const { deployProxy } = require('@openzeppelin/trufflelanguage-none
upgrades');

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

module.exports = async function (deployer) {
  const instance = await deployProxy(Box, [42],{ kind: 'uups'
});
  console.log('Deployed', instance.address);
};
```

The deployProxy function has a number of [options](#) :

```
async function deployProxy(
  Contract: ContractClass,
  args: unknown[] = [],
  opts: {
    deployer: Deployer,
    initializer: string | false,
    unsafeAllow: ValidationError[],
    kind: 'uups' | 'transparent',
  } = {},
```

```
): Promise<ContractInstance>
```

Performing the upgrade

See [documentation](#)

```
const { upgradeProxy } = require('@openzeppelin/truffle    language-none
upgrades');

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

module.exports = async function (deployer) {
  const existing = await Box.deployed();
  const instance = await upgradeProxy(existing.address, BoxV2,
{ kind: 'uups' });
  console.log("Upgraded", instance.address);
};
```

Testing the upgrade process

You can add the upgrade process to a unit test

```
const { deployProxy, upgradeProxy } =                      language-none
require('@openzeppelin/truffle-upgrades');

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

describe('upgrades', () => {
  it('works', async () => {
    const box = await deployProxy(Box, [42] { kind: 'uups' });
    const box2 = await upgradeProxy(box.address, BoxV2);

    const value = await box2.value();
    assert.equal(value.toString(), '42');
  });
});
```

Other functions

prepareUpgrade

Use this to allow the plugin to check that your contracts are upgrade safe and deploy a new implementation contract `

admin.changeAdminForProxy

Use this to change the administrator of the proxy contract.

Security Considerations

1. Always initialise your contract
2. Do not allow self destruct or delegatecall in your implementation contracts.

The closest I have found for Foundry is this project

Diamond pattern

Based on [EIP 2535](EIP 2535)

From Aavegotchi (https://docs.aavegotchi.com/overview/diamond-standard)
The diamond pattern is a contract that uses a fallback function to delegate function calls to multiple other contracts called facets. Conceptually a diamond can be thought of as a contract that gets its external functions from other contracts. A diamond has four standard functions (called the loupe) that report what functions and facets a diamond has. A diamond has a `DiamondCut` event that reports all functions/facets that are added/replaced/removed on a diamond, making upgrades on diamonds transparent.

The diamond pattern is a code implementation and organization strategy. The diamond pattern makes it possible to implement a lot of contract functionality that is compartmented into separate areas of functionality, but still using the same Ethereum address. The code is further simplified and saves gas because state variables are shared between facets.

Diamonds are not limited by the maximum contract size which is 24KB.

Facets can be deployed once and reused by any number of diamonds.

From Trail of Bits Audit
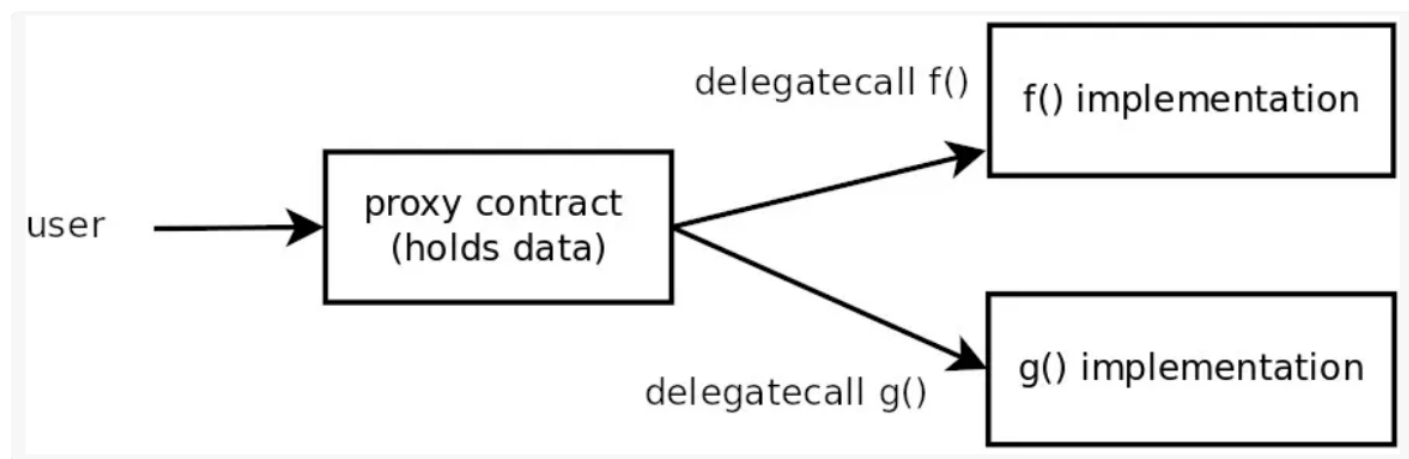

```solidity
bytes32 constant POSITION = keccak256(
    "some_string"
);

struct MyStruct {
    uint var1;
    uint var2;
}

function get_struct() internal pure returns(MyStruct storage ds)
{
    bytes32 position = POSITION;
    assembly { ds_slot := position }
}
```

(The _slot suffix gives the storage address)

Trail of Bits Audit - Good idea, bad design

"The code is over-engineered, with lots of unnecessary complexities, and we can't recommend it at this time."

But.. projects are using it
For example [Aavegotchi](#)

"AavegotchiDiamond provides a single Ethereum address for Aavegotchi functionality. All contract interaction with Aavegotchis is done with AavegotchiDiamond."

From Nick Mudge [article](#)
Diamonds solve these problems:

1. The maximum size a smart contract can be on Ethereum is 24kb. But sometimes larger smart contracts are needed or desired. Diamonds solve that problem.
2. Provides a structure to systematically and logically organize and extend larger smart contract systems so they don't turn into a spaghetti code mess.
3. Provides fine-grained upgrades. Other upgrade approaches require replacing functionality in bulk. With a diamond you can add, replace, or remove just the functionality that needs to be added, replaced or removed without affecting or touching other smart contract functionality.
4. Provides a single address for a lot of smart contract functionality. This makes integration with smart contracts and user interfaces and other software easier.

Question - How is this different to using a library ?

Metamorphic Contracts

CREATE and CREATE2 and selfdestruct

CREATE gives the address that a contract will be deployed to

```language-none
keccak256(rlp.encode(deployingAddress, nonce))[12:]
```

CREATE2 introduced in Feb 2019

```language-none
keccak256(0xff + deployingAddr + salt + keccak256(bytecode))
[12:]
```

Contracts can be deleted from the blockchain by calling selfdestruct.

selfdestruct sends all remaining Ether stored in the contract to a designated address.

This can be used to preserve the contract address among upgrades, but not its state. It relies on the contract calling selfdestruct then being re deployed via CREATE2

Seen as 'an abomination' by some
See [Metamorphic contracts](#)
and
[Abusing CREATE2 with Metamorphic Contracts](#)
and
[Efficient Storage](#)

IDE General Techniques

Importing from Github in Remix

See [Documentation](#)
You can import directly from github or npm

```
import "https://github.com/OpenZeppelin/openzeppelin-contracts
/contracts/access/Ownable.sol";
```

or

```
import "@openzeppelin/contracts@4.2.0/token/ERC20/ERC20.sol";
```

Logging in Remix

https://remix-ide.readthedocs.io/en/latest/hardhat_console.html

Remix IDE supports hardhat console library while using `JavaScript VM`. It can be used while making a transaction or running unit tests.
To try it out, you need to put an import statement and use `console.log` to print the value as shown in image.

```solidity
 7    * @dev Set & change owner
 8    */
 9
10   import "hardhat/console.sol";
11
12   contract Owner {
13
14       address private owner;
15
16       // event for EVM logging
17       event OwnerSet(address indexed oldOwner, address indexed newOwner);
18
19       // modifier to check if caller is owner
20       modifier isOwner() {
21           // If the first argument of 'require' evaluates to 'false', execution terminates and all
22           // changes to the state and to Ether balances are reverted.
23           // This used to consume all gas in old EVM versions, but not anymore.
24           // It is often a good idea to use 'require' to check if functions are called correctly.
25           // As a second argument, you can also provide an explanation about what went wrong.
26           require(msg.sender == owner, "Caller is not owner");
27           _;
28       }
29
30       /**
31        * @dev Set contract deployer as owner
32        */
33       constructor() {
34           owner = msg.sender; // 'msg.sender' is sender of current call, contract deployer for a constructor
35           emit OwnerSet(address(0), owner);
36       }
37
38       /**
39        * @dev Change owner
40        * @param newOwner address of new owner
41        */
42       function changeOwner(address newOwner) public isOwner {
43           console.log('msg.sender :', msg.sender);
44           emit OwnerSet(owner, newOwner);
45           owner = newOwner;
46       }
47
```

Deployed Contracts

OWNER AT 0XD91...39138 (MEMORY)

changeOwner   0xAb8483F64d9C6d1EcF9b

getOwner

Low level interactions   i

CALLDATA

Transact

---

0   listen on network   Search with transaction hash or address

[vm] from: 0x5B3...eddC4 to: Owner.(constructor) value: 0 wei data: 0x608...70033 logs: 1 hash: 0x7c5...7e0aa

transact to Owner.changeOwner pending ...

[vm] from: 0x5B3...eddC4 to: Owner.changeOwner(address) 0xd91...39138 value: 0 wei data: 0xa6f...35cb2 logs: 1 hash: 0xaf1...8ed3c

console.log:
  msg.sender : 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

>

Forking from mainnet

The easiest way to try this feature is to start a node from the command line:

```
npx hardhat node --fork https://eth-mainnet.alchemyapi.io/v2/<key>
```

You can also configure Hardhat Network to always do this:

```js
networks: {
  hardhat: {
    forking: {
      url: "https://eth-mainnet.alchemyapi.io/v2/<key>",
    }
  }
}
```

Solidity Templates

Paul Berg has created some useful templates

Solidity Template

[Repo](#)
This will give you

- [Hardhat](#): compile, run and test smart contracts
- [TypeChain](#): generate TypeScript bindings for smart contracts
- [Ethers](#): renowned Ethereum library and wallet implementation
- [Solhint](#): code linter
- [Solcover](#): code coverage
- [Prettier Plugin Solidity](#): code formatter

COVERAGE REPORTS

```
yarn coverage
```

all files contracts/

100% Statements `5/5` **100%** Branches `0/0` **75%** Functions `3/4` **83.33%** Lines `5/6`

File ▲		Statements		Branches		Functions		Lines	
Greeter.sol		100%	5/5	100%	0/0	75%	3/4	83.33%	5/6

all files / contracts/ Greeter.sol

100% Statements `5/5`   **100%** Branches `0/0`   **75%** Functions `3/4`   **83.33%** Lines `5/6`

```solidity
1     // SPDX-License-Identifier: MIT
2     pragma solidity >=0.8.4;
3
4     import "hardhat/console.sol";
5
6     error GreeterError();
7
8     contract Greeter {
9         string public greeting;
10
11        constructor(string memory _greeting) {
12 1×        console.log("Deploying a Greeter with greeting:", _greeting);
13 1×        greeting = _greeting;
14        }
15
16        function greet() public view returns (string memory) {
17 2×        return greeting;
18        }
19
20        function setGreeting(string memory _greeting) public {
21 12×       console.log("Changing greeting from '%s' to '%s'", greeting, _greeting);
22 12×       greeting = _greeting;
23        }
24
25        function throwError() external pure {
26            revert GreeterError();
27        }
28    }
29
```

Gas report

```
REPORT_GAS=true yarn test
```

```
  ✓ should return the new greeting once it's changed

·-----------------------------|---------------------------|-------------|----------------------------·
|    Solc version: 0.8.9      ·  Optimizer enabled: true  ·  Runs: 800  ·  Block limit: 30000000 gas  |
······························|···························|·············|·····························
|  Methods                                                                                            |
·············|················|·············|·············|·············|··············|···············
|  Contract  ·  Method        ·  Min        ·  Max        ·  Avg        ·  # calls     ·  usd (avg)   |
·············|················|·············|·············|·············|··············|···············
|  Greeter   ·  setGreeting   ·          -  ·          -  ·      34489  ·           1  ·           -  |
·············|················|·············|·············|·············|··············|···············
|  Deployments                ·                           ·             ·  % of limit  ·              |
·····························|·············|·············|·············|··············|···············
|  Greeter                    ·          -  ·          -  ·     428178  ·      1.4 %   ·           -  |
·-----------------------------|-------------|-------------|-------------|--------------|---------------·

  1 passing (2s)
```

Foundry Template

[Repo](#)
This gives you :

- [Forge](#): compile, test, fuzz, debug and deploy smart contracts
- [PRBTest](#): modern collection of testing assertions and logging utilities
- [Forge Std](#): collection of helpful contracts and cheatcodes for testing
- [Solhint](#): code linter
- [Prettier Plugin Solidity](#): code formatter

You can install from the github template button, or from the command line with

```
forge init my-project --template
https://github.com/PaulRBerg/foundry-template
cd my-project
yarn install # install solhint and prettier etc.
```

Foundry

There's a bit of a trend toward writing "clever" Solidity code and it's not a good one. Boring code is better code.

[Quote](#)

Foundry is a smart contract development toolchain written in Solidity. It can:

- manage dependencies
- compile your project
- run tests
- deploy contracts
- interact with the chain via scripts and CMD

Forge is the CLI tool to initialise, build and test Foundry projects.

Foundry allows fast (milliseconds) and sophisticated Solidity tests, e.g. reading and writing directly to storage slots.

Getting Started

1. Install Foundry
   - [Steps for various operation systems.](#)
2. Initialise a project called **lets_forge**
   - `forge init lets_forge`
3. Build the project
   - `forge build`
4. Verify install has worked by running the tests
   - `forge test`

You can also use this Foundry [template](#).

Project structure is as follows:

```cmd
$ cd hello_foundry
$ tree . -d -L 1
.
├── lib
├── script
├── src
└── test

4 directories
```

Build command will generate the folders `out` and `cache` to store the contract artifact (ABI) and cached data, respectively.

You configure your project in the `foundry.toml` file. See ref: https://book.getfoundry.sh/reference/config/

General documentation available in the Foundry [Book](#). **Highly recommended**.

Testing: Basics

> *Smart contracts are tested using smart contracts, which is the secret to Foundry's speed since there is no additional compilation being carried out.*

A smart contract e.g. MyContract.sol is tested using a file named MyContract.t.sol:

```cmd
├── src\
│   └── MyContract.sol\
└── test\
  └── MyContract.t.sol
```

MyContract.t.sol will import the contract under test in order to access it's functions.

First Contract

Create a contract called `A.sol` and save it in `src` with the following contents:

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

/// @title Encode smart contract A
/// @author Extropy.io
contract A {
    uint256 number;

    /**
     * @dev Store value in variable
     * @param num value to store
     */
    function store(uint256 num) public {
        number = num;
    }

    /**
     * @dev Return value
     * @return value of 'number'
     */
    function retrieve() public view returns (uint256) {
        return number;
```

```
        }
}
```

Then create a file to test the smart contract, for example `A.t.sol` in `test` with the following contents:

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

// Standard test libs
import "forge-std/Test.sol";
import "forge-std/Vm.sol";

// Contract under test
import {A} from "../src/A.sol";

contract ATest is Test {
    // Variable for contract instance
    A private a;

    function setUp() public {
        // Instantiate new contract instance
        a = new A();
    }

    function test_Log() public {
        // Various log examples
        emit log("here");
        emit log_address(address(this));
        // HEVM_ADDRESS is a special reserved address for the VM
        emit log_address(HEVM_ADDRESS);
    }

    function test_GetValue() public {
        assertTrue(a.retrieve() == 0);
    }

    function test_SetValue() public {
        uint256 x = 123;
        a.store(x);
        assertTrue(a.retrieve() == 123);
    }
}
```

```solidity
        // Define the value(s) being fuzzed as an input argument
        function test_FuzzValue(uint256 _value) public {
            // Define the boundaries for the fuzzing, in this case
0 and 99999
            _value = bound(_value, 0, 99999);
            // Call contract function with value
            a.store(_value);
            // Perform validation
            assertTrue(a.retrieve() == _value);
        }
    }
```

Finally run the test with `forge test -vv` to see the results and all logs.

Run tests

```
forge test
```

To print event logs:

```
forge test -vv
```

To print trace of any failed tests:

```
forge test -vvv
```

To print trace of all tests:

```
forge test -vvvv
```

To run a specific test:

```
forge test --match-test test_myTest
```

Dependencies

Forge uses [git submodules](#) so it works with any GitHub repo that contains smart contracts.

Adding Dependencies

To install e.g. OpenZeppelin contracts from the [repo](#) one would run `forge install OpenZeppelin/openzeppelin-contracts`. The repo will have been cloned into the `lib` folder.

Dependencies can be updated by running `forge update`.

Removing Dependencies

Dependencies can be removed by running `forge remove openzeppelin-contracts`, which is equivalent to `forge remove OpenZeppelin/openzeppelin-contracts`.

Integrating with Existing Hardhat project

Foundry can work with Hardhat-style projects where dependencies are npm packages (stored in `node_modules`) and contracts are stored in `contracts` as opposed to source.

1. Copy lib/forge-std from a newly-created empty Foundry project to this Hardhat project directory.
2. Copy foundry.toml configuration to this Hardhat project directory and change src, out, test, cache_path in it:

```
  [default]
src = 'contracts'
out = 'out'
libs = ['node_modules', 'lib']
test = 'test/foundry'
cache_path  = 'forge-cache'
```

3. Create a remappings.txt to make Foundry project work well with VS Code Solidity extension:

```
  ds-test/=lib/forge-std/lib/ds-test/src/
forge-std/=lib/forge-std/src/
```

4. Make a sub-directory test/foundry and write Foundry tests in it.

Foundry test works in this existing Hardhat project. As the Hardhat project is not touched and it can work as before.

See folder `hardhat-foundry`. Both test suites function:

```
npx hardhat test
```

```
forge test -vvv
```

Deploying Contracts

`forge create` allows you to deploy contracts to a blockchain.

To deploy the previous contract to e.g. Ganache using an account with private key `4e0...9b` you would type:

```
forge create A --legacy --contracts src/A.sol --private-key
4e0...9b --rpc-url http://127.0.0.1:8545
```

Result:

```
  Deployer: 0x536f8222c676b6566ff34e539022de6c1c22cc06
Deployed to: 0x79bb7a73d02b6d7e2e98848d26ad911720421df0
Transaction hash:
0xc5bb34ee82dc2f57bd7f7862eec440576a7cc7cfe4533392192704fd44653b68
```

The `--legacy` flag is used since Ganache doesn't support EIP-1559 transactions. Hardhat (`npx hardhat node`) circumvents this issue.

Solidity Scripting

> *Solidity scripting is a way to declaratively deploy contracts using Solidity, instead of using the more limiting and less user friendly forge create.*

Similar to Hardhat scripting but faster with dry-run capabilities (because it runs in Foundry EVM).

Scripts go in the `/scripts` folder and file extension is `.s.sol`.

Running a script. You should add the variables to the .env beforehand & run from root:

```
  # To load the variables in the .env file
source .env

# To deploy and verify our contract
forge script script/NFT.s.sol:MyScript --rpc-url $GOERLI_RPC_URL --broadcast --verify -vvvv
```

Example of a deployment script:

```js
// SPDX-License-Identifier: UNLICENSED          language-js
pragma solidity 0.8.18;

import "forge-std/Script.sol"; // Import script lib
import "../src/ContractA.sol";
import "../src/ContractB.sol";
import "../src/ContractC.sol";

contract Deployment is Script {
    function run() public {

        //Load key/unlock wallet. Use with caution if in
production
        uint256 deployerPrivateKey = vm.envUint("DEPLOY_KEY");
        vm.startBroadcast(deployerPrivateKey);

        //Deploy contracts
        ContractA contractA = new ContractA();
        ContractB contractB = new ContractB();
        ContractC contractC = new ContractC();

        //Do some complex deployment logic
        contractA.callSomeFunction();
        uint256 result =
contractB.anotherFunction(address(contractA));
        contractC.someOtherFunction(result);

        vm.stopBroadcast();
    }
}
```

Cast

> *Perform Ethereum RPC calls from the comfort of your command line.*

For example:

- `cast chain-id` Get the Ethereum chain ID.
- `cast publish` Publish a raw - transaction to the network.
- `cast receipt` Get the transaction receipt for a transaction.
- `cast send` Sign and publish a transaction.
- `cast call` Perform a call on an account without publishing a transaction.
- `cast block-number` Get the latest block number.
- `cast abi-encode` ABI encode the given function arguments, excluding the selector.

How to use Cast

```
cast [options] command [args] cast [options] --version cast [options] --help
```

eg:

```
cast call 0x79bb7a73d02b6d7e2e98848d26ad911720421df0 "retrieve()" --rpc-url http://127.0.0.1:8545
```

There are a lot of cast commands. Take a look at the reference:
https://book.getfoundry.sh/reference/cast/cast
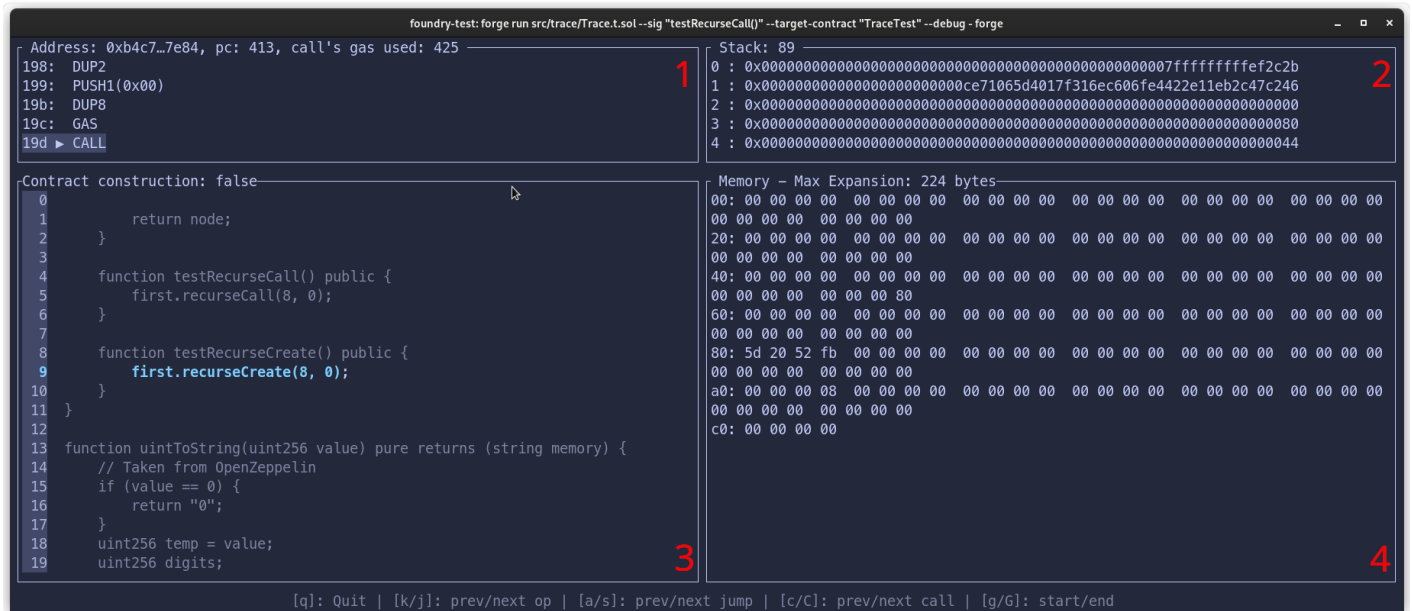
Debugging in Foundry

There are 2 ways to debug:

- With `forge test` passing a function from tests `forge test --debug $FUNC`

Example `forge test --debug "testSomething()"`

- With `forge debug` passing a file `forge debug --debug $FILE --sig $FUNC`

Example: `forge debug --debug src/SomeContract.sol --sig "myFunc(uint256,string)" 123 "hello"`

When the debugger is run, you are presented with a terminal divided into four quadrants:

- 1: The opcodes in the debugging session
- 2: The current stack, as well as the size of the stack
- 3: The source view
- 4: The current memory of the EVM

> *If you have multiple contracts with the same function name, you need to limit the matching functions down to only one case using --match-path and --match-contract.*

If the matching test is a fuzz test, the debugger will open the first failing fuzz scenario, or the last successful one, whichever comes first.

See guide for more info [book](#)

Forking

Forge supports testing in a forked environment with two different approaches:

- Forking Mode — use a single fork for all your tests via the forge test `--fork-url` flag.

```
forge test --fork-url <your_rpc_url>
```

- Forking Cheatcodes in tests — create, select, and manage multiple forks directly in Solidity test code via forking cheatcodes

```solidity
contract ForkTest is Test {
    // the identifiers of the forks
    uint256 mainnetFork;
    uint256 optimismFork;

    //Access variables from .env file via vm.envString("varname")
    //Replace ALCHEMY_KEY by your alchemy key or Etherscan key, change RPC url if need
    //inside your .env file e.g:
    //MAINNET_RPC_URL = 'https://eth-mainnet.g.alchemy.com//v2/ALCHEMY_KEY'
    //string MAINNET_RPC_URL = vm.envString("MAINNET_RPC_URL");
    //string OPTIMISM_RPC_URL = vm.envString("OPTIMISM_RPC_URL");

    // create two _different_ forks during setup
    function setUp() public {
        mainnetFork = vm.createFork(MAINNET_RPC_URL);
        optimismFork = vm.createFork(OPTIMISM_RPC_URL);
    }

    // demonstrate fork ids are unique
    function testForkIdDiffer() public {
        assert(mainnetFork != optimismFork);
    }

    // select a specific fork
    function testCanSelectFork() public {
```

```solidity
        // select the fork
        vm.selectFork(mainnetFork);
        assertEq(vm.activeFork(), mainnetFork);

        // from here on data is fetched from the `mainnetFork`
        // if the EVM requests it and written to the storage of
        // `mainnetFork`
    }

    // manage multiple forks in the same test
    function testCanSwitchForks() public {
        vm.selectFork(mainnetFork);
        assertEq(vm.activeFork(), mainnetFork);

        vm.selectFork(optimismFork);
        assertEq(vm.activeFork(), optimismFork);
    }

    // forks can be created at all times
    function testCanCreateAndSelectForkInOneStep() public {
        // creates a new fork and also selects it
        uint256 anotherFork =
            vm.createSelectFork(MAINNET_RPC_URL);
        assertEq(vm.activeFork(), anotherFork);
    }

    // set `block.timestamp` of a fork
    function testCanSetForkBlockTimestamp() public {
        vm.selectFork(mainnetFork);
        vm.rollFork(1_337_000);

        assertEq(block.number, 1_337_000);
    }
}
```

Cheatcodes

Cheatcodes allow you to change the block number, your identity, and more. They are invoked by calling specific functions on a specially designated address: `0x7109709ECfa91a80626fF3989D68f67F5b1DD12D` .

You can access cheatcodes easily via the 'vm' instance available in Forge Standard Library's Test contract. Forge Standard Library is explained in greater detail in the following section.

Below are some subsections for the different Forge cheatcodes.

- Environment: Cheatcodes that alter the state of the EVM.
- Assertions: Cheatcodes that are powerful assertions
- Fuzzer: Cheatcodes that configure the fuzzer
- External: Cheatcodes that interact with external state (files, commands, ...)
- Utilities: Smaller utility cheatcodes
- Forking: Forking mode cheatcodes
- Snapshots: Snapshot cheatcodes
- File: Cheatcodes for working with files

See cheatcodes reference in docs: https://book.getfoundry.sh/cheatcodes/

```solidity
// Set block.timestamp
function warp(uint256) external;

// Set block.number
function roll(uint256) external;

// Set block.basefee
function fee(uint256) external;

// Set block.difficulty
function difficulty(uint256) external;

// Set block.chainid
function chainId(uint256) external;

// Loads a storage slot from an address
function load(address account, bytes32 slot) external returns
```

```
    (bytes32);

    // Stores a value to an address' storage slot
    function store(address account, bytes32 slot, bytes32 value)
    external;

    // Signs data
    function sign(uint256 privateKey, bytes32 digest)
        external
        returns (uint8 v, bytes32 r, bytes32 s);

    // Computes address for a given private key
    function addr(uint256 privateKey) external returns (address);
```

Broadcast will emulate your deployment on chain and return a comprehensive trace of the execution.

```
→ forge-template git:(master) ✗ forge script script/Script.sol --rpc-url $ETH_RPC_URL -vvvv --broadcast --private-key $PRIVKEY
[··] Compiling...
[··] Compiling 6 files with 0.8.13
[··] Solc 0.8.13 finished in 666.82ms
Compiler run successful
Traces:
  [87942] MyScript::run()
    ├─ [0] VM::startBroadcast()
    │   └─ ← ()
    ├─ [51587] → new Contract@0xe2d5bb16874adb5de7c257919feff7610624865b
    │   └─ ← 147 bytes of code
    ├─ [238] Contract::test()
    │   └─ ← ()
    ├─ [261] Contract::x() [staticcall]
    │   └─ ← 12345
    └─ ← ()


Script ran successfully.
Gas used: 87942
==========================
Simulated On-chain Traces:

  [107343] → new Contract@0xe2d5bb16874adb5de7c257919feff7610624865b
    └─ ← 147 bytes of code

  [26202] Contract::test()
    └─ ← ()



==========================

Estimated total gas used for script: 133545
==========================

###
Finding wallets for all the necessary addresses...

#####
✅ Hash: 0xe1ba914f63259139a18561509730db39703a8c73dd81581da29943081447eb2e
Contract Address: 0xe2d5bb16874adb5de7c257919feff7610624865b
Block: 10762422
Nonce: 21
Paid: 0.00032862900109543 ETH (109543 gas * 3.00000001 gwei)

#####
✅ Hash: 0x99dbde7eec0977b26a8f667949eb43efffc6fbae8ea61e3610ff239500743145
Block: 10762422
Nonce: 22
Paid: 0.00007860600026202 ETH (26202 gas * 3.00000001 gwei)


==========================

ONCHAIN EXECUTION COMPLETE & SUCCESSFUL. Transaction receipts written to "broadcast/Script.sol/4/run-latest.json"

Transactions saved to: broadcast/Script.sol/4/run-latest.json

→ forge-template git:(master) ✗ █
```