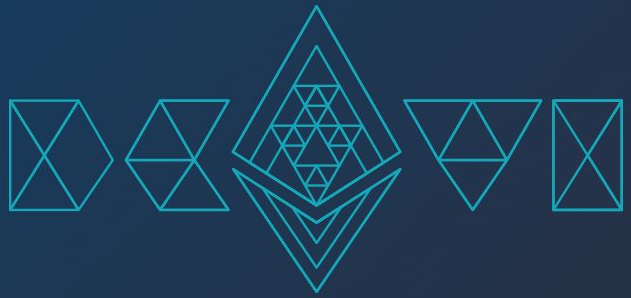


Alice in Proxyland

Playing with upgradeabilty and the Router Proxy

Alejandro Santander a.k.a. Ethernaut

Core Contributor @ Synthetix

(ex Aragon, ex OpenZeppelin)

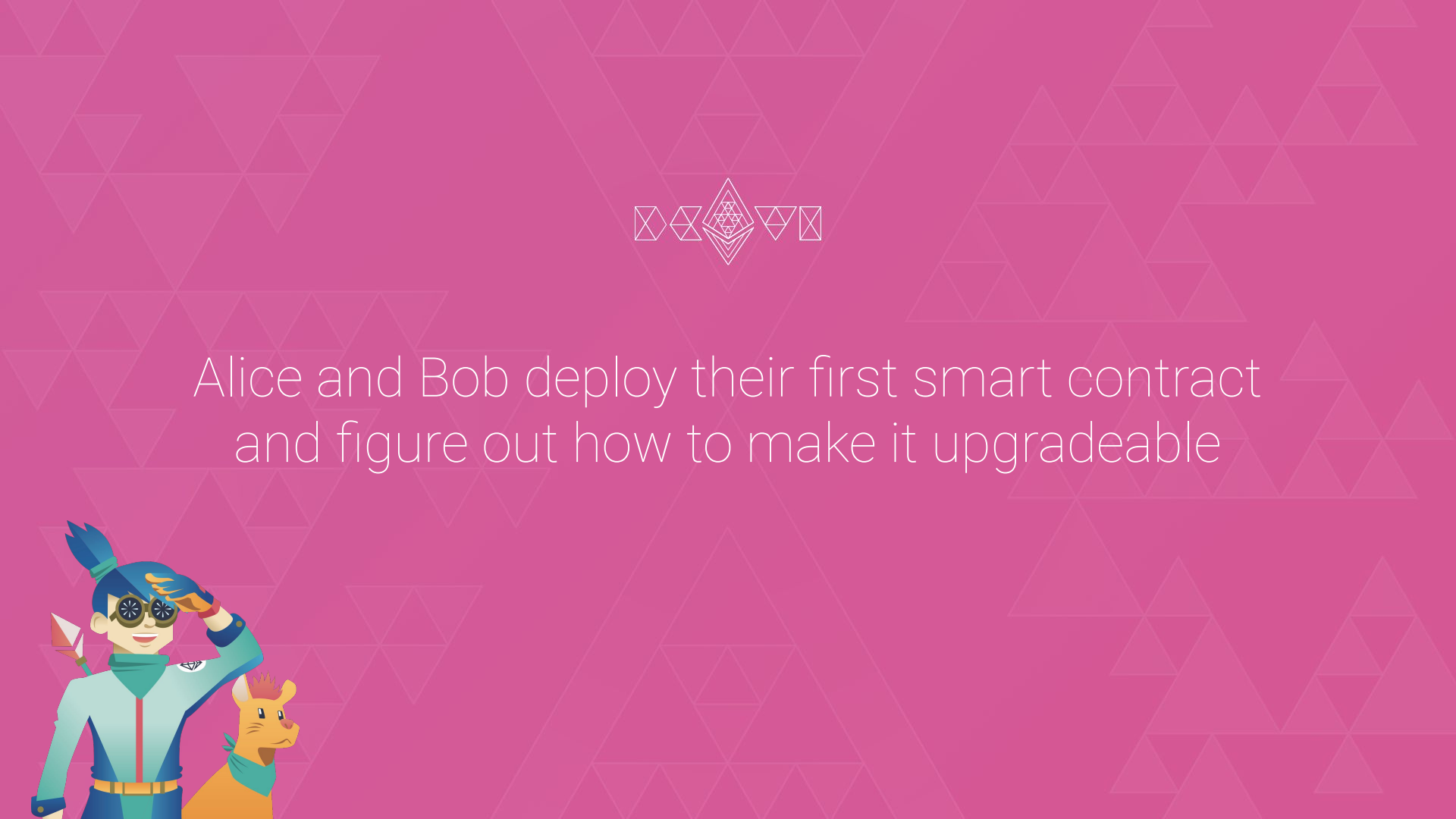
Understanding how proxies work under the hood is important

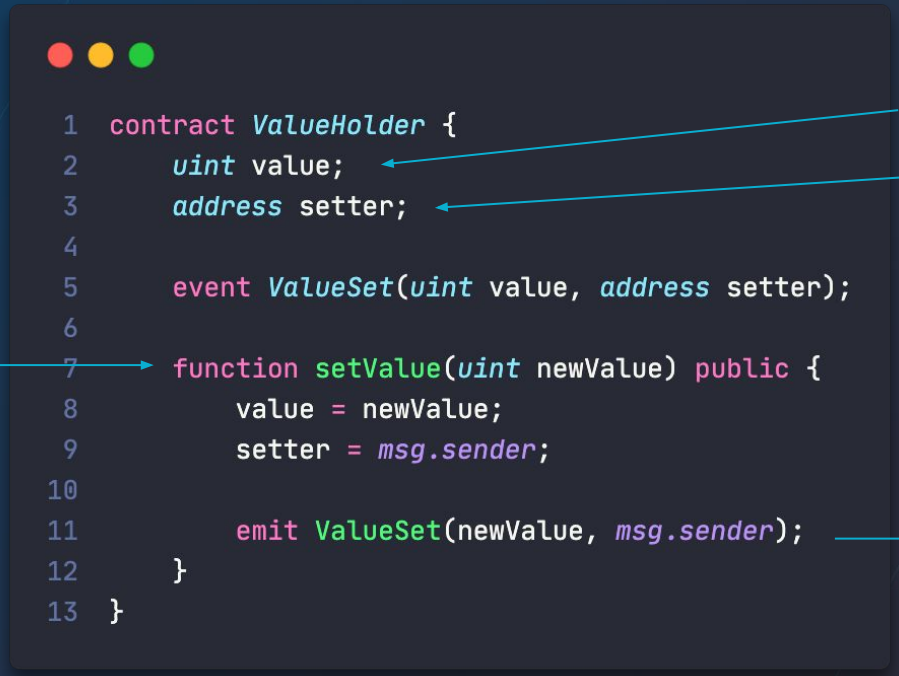
Everyone is using proxies, but few really understand them, and their dangers

Proper proxy usage requires tooling, and a fundamental knowledge of how they work. It's not ok to not understand their core principles and caveats.

It's easy tho. We're now going to understand them by playing dumb with a silly example…

After the example, we're going to use the core principles we learnt to talk about a new type of proxy.


Alice and Bob deploy their first smart contract
and figure out how to make it upgradeable


contract ValueHolder {
uint value;
address setter;
event ValueSet(uint value, address setter);
function setValue(uint newValue) public {
value = newValue;
setter = msg.sender;
emit ValueSet(newValue, msg.sender);
}
}
Storage @ 0x1
0: 42
1: 0xb0b
0xb0b calls
setValue(42)
Event from 0x1
deployed by 0xb0b @ 0x1

Alice and Bob decide to make their contract
upgradeable.

0xb0b calls
setImplementation(0x1)

```
1  contract CallProxy {
2      address implementation;
3
4      function setImplementation(
5          address newImplementation
6      ) public {
7          implementation = newImplementation;
8      }
9
10     fallback() external {
11         // Forwards any incoming call
12         // to the implementation
13         // using CALL
14     }
15 }
```

deployed @ 0x2

Storage @ 0x2
0: **0x1**


0xb0b calls value()
contract CallProxy {
address implementation;
function setImplementation(
address newImplementation
) public {
implementation = newImplementation;
}
fallback() external {
// Forwards any incoming call
// to the implementation
// using CALL
}
}
0x2
Storage @ 0x2
0: 0x1
contract ValueHolder {
uint value;
address setter;
event ValueSet(uint value, address setter);
function setValue(uint newValue) public {
value = newValue;
setter = msg.sender;
emit ValueSet(newValue, msg.sender);
}
}
0x1
Storage @ 0x1
0: 42
1: 0xb0b
42 is returned

0xb0b calls
setValue(1337)

```
1  contract CallProxy {
2      address implementation;
3
4      function setImplementation(
5          address newImplementation
6      ) public {
7          implementation = newImplementation;
8      }
9
10     fallback() external {
11         // Forwards any incoming call
12         // to the implementation
13         // using CALL
14     }
15 }
```

0x2

Storage @ 0x2
0: 0x1

CALL

```
1  contract ValueHolder {
2      uint value;
3      address setter;
4
5      event ValueSet(uint value, address setter);
6
7      function setValue(uint newValue) public {
8          value = newValue;
9          setter = msg.sender;
10
11         emit ValueSet(newValue, msg.sender);
12     }
13 }
```

0x1

Execution context

Storage @ 0x1
0: **1337**
1: **0x2**

Event from **0x1**

Execution context?

When running EVM code, the execution context determines:

- Which contract storage to use
- Who msg.sender is: whoever made the call
- Which contract to emit events from

We want the execution context to be the **proxy**, not the implementation.

But how?

CALL vs DELEGATECALL

- **CALL** runs code in the **current** context
- **DELEGATECALL** runs code in the context of the **caller**

0xb0b calls
setImplementation(0x1)

```
1  contract DelegateCallProxy {
2      address implementation;
3
4      function setImplementation(
5          address newImplementation
6      ) public {
7          implementation = newImplementation;
8      }
9
10     fallback() external {
11         // Forwards any incoming call
12         // to the implementation
13         // using DELEGATECALL
14     }
15 }
```

deployed @ 0x3

Storage @ 0x3
0: **0x1**

Event from **0x3**

Storage @ 0x3
0: **1337**
1: **0xb0b**

0xb0b calls
setValue(1337)

```sol
contract DelegateCallProxy {
    address implementation;

    function setImplementation(
        address newImplementation
    ) public {
        implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
```

0x3

Execution context

DELEGATECALL

```sol
contract ValueHolder {
    uint value;
    address setter;

    event ValueSet(uint value, address setter);

    function setValue(uint newValue) public {
        value = newValue;
        setter = msg.sender;

        emit ValueSet(newValue, msg.sender);
    }
}
```

0x1

Storage @ 0x1
0: 1337
1: 0x2


contract DelegateCallProxy {
address implementation;
function setImplementation(
address newImplementation
) public {
implementation = newImplementation;
}
fallback() external {
// Forwards any incoming call
// to the implementation
// using DELEGATECALL
}
}
0xb0b calls value()
Storage @ 0x3
0: 1337
1: 0xb0b
Storage collision between proxy & implementation
DELEGATECALL
1337?
0x3
Execution context

DELEGATECALL is great, but it is dangerous

How to avoid the storage collision?

Bob **“destructures”** the proxy’s storage.

Destructuring?

Destructuring is simply choosing where in storage to put a variable

- Solidity uses slots consecutively
- First variable is given slot 0
- Second variable slot 1, etc
- A contract has infinite potential slots
- Destructuring is just picking a custom slot

Storage @ 0x4

0: 0

...

1000: **0x1**

```
1  contract UnstructuredProxy {
2      struct ProxyStore {
3          address implementation;
4      }
5
6      function _getStore()
7          private
8          pure
9          returns (ProxyStore storage store)
10     {
11         assembly {
12             store.slot := 1000
13         }
14     }
15
16     function setImplementation(
17         address newImplementation
18     ) public {
19         _getStore().implementation = newImplementation;
20     }
21
22     fallback() external {
23         // Forwards any incoming call
24         // to the implementation
25         // using DELEGATECALL
26     }
27 }
```

0xb0b calls setImplementation(0x1)

deployed @ 0x4

Event from **0x4**

Storage @ 0x4

0: **1337**

1: **0xb0b**

...

1000: 0x1

0xb0b calls setValue(1337)

```
contract UnstructuredProxy {
    struct ProxyStore {
        address implementation;
    }

    function _getStore()
        private
        pure
        returns (ProxyStore storage store)
    {
        assembly {
            store.slot := 1000
        }
    }

    function setImplementation(
        address newImplementation
    ) public {
        _getStore().implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
```

0x4

Execution context

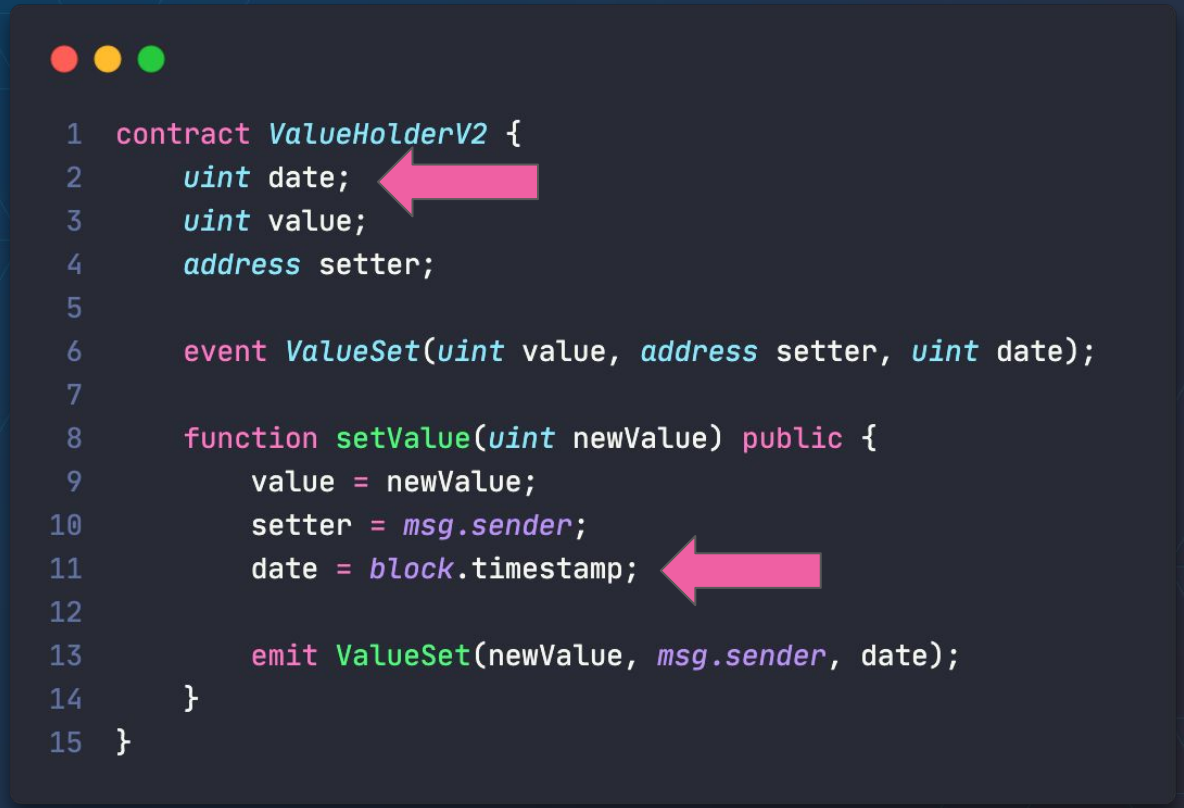
DELEGATECALL

```
contract ValueHolder {
    uint value;
    address setter;

    event ValueSet(uint value, address setter);

    function setValue(uint newValue) public {
        value = newValue;
        setter = msg.sender;

        emit ValueSet(newValue, msg.sender);
    }
}
```

0x1

Ok, we've finally found a proxy that works.

It's time to actually use it and upgrade the implementation.



deployed @ 0x5

```
1  contract UnstructuredProxy {
2      struct ProxyStore {
3          address implementation;
4      }
5
6      function _getStore()
7          private
8          pure
9          returns (ProxyStore storage store)
10     {
11         assembly {
12             store.slot := 1000
13         }
14     }
15
16     function setImplementation(
17         address newImplementation
18     ) public {
19         _getStore().implementation = newImplementation;
20     }
21
22     fallback() external {
23         // Forwards any incoming call
24         // to the implementation
25         // using DELEGATECALL
26     }
27 }
```

Storage @ 0x4

0: 1337

1: 0xb0b

...

1000: **0x5**

0xb0b calls setImplementation(0x5)

0x4

Storage @ 0x4
0: 1337
1: 0xb0b
2: 0
...
1000: 0x5

Storage collision between implementations

0xb0b calls value()

DELEGATECALL

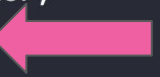
```
contract ValueHolderV2 {
    uint date;
    uint value;
    address setter;

    event ValueSet(uint value, address setter, uint date);

    function setValue(uint newValue) public {
        value = newValue;
        setter = msg.sender;
        date = block.timestamp;

        emit ValueSet(newValue, msg.sender, date);
    }
}
```

0x5

0xb0b is returned

Bob understands that implementation storage can only be **appended** to.

```
1  contract ValueHolderV3 {
2      uint value;
3      address setter;
4      uint date;
5
6      event ValueSet(uint value, address setter, uint date);
7
8      function setValue(uint newValue) public {
9          value = newValue;
10         setter = msg.sender;
11         date = block.timestamp;
12
13         emit ValueSet(newValue, msg.sender, date);
14     }
15 }
```



deployed @ 0x6

```
1  contract UnstructuredProxy {
2      struct ProxyStore {
3          address implementation;
4      }
5
6      function _getStore()
7          private
8          pure
9          returns (ProxyStore storage store)
10     {
11         assembly {
12             store.slot := 1000
13         }
14     }
15
16     function setImplementation(
17         address newImplementation
18     ) public {
19         _getStore().implementation = newImplementation;
20     }
21
22     fallback() external {
23         // Forwards any incoming call
24         // to the implementation
25         // using DELEGATECALL
26     }
27 }
```

Storage @ 0x4
0: 1337
1: 0xb0b
...
1000: **0x6**

0xb0b calls
setImplementation(0x6)

0x4

```
contract UnstructuredProxy {
    struct ProxyStore {
        address implementation;
    }

    function _getStore()
        private
        pure
        returns (ProxyStore storage store)
    {
        assembly {
            store.slot := 1000
        }
    }

    function setImplementation(
        address newImplementation
    ) public {
        _getStore().implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
```

0x4

Execution context

Storage @ 0x4

0: 1337

1: 0xb0b

2: 0

...

1000: 0x6

0xb0b calls value()

DELEGATECALL

```
contract ValueHolderV3 {
    uint value;
    address setter;
    uint date;

    event ValueSet(uint value, address setter, uint date);

    function setValue(uint newValue) public {
        value = newValue;
        setter = msg.sender;
        date = block.timestamp;

        emit ValueSet(newValue, msg.sender, date);
    }
}
```

0x6

1337 is returned

Storage collisions

It is critical to understand what they are:

- The execution context is the proxy; Everything is stored in the proxy
- Collisions can exist (1) between the proxy and the current implementation, and (2) between different versions of the implementation
- The first kind can be avoided with unstructuring the proxy's storage layout
- The rest of the storage layout is defined by the current implementation and it can be incompatible with a previous storage layout definition
- Always append to storage
- Multiple inheritance can produce unexpected storage layouts
- It becomes hard to detect invalid mutations so tooling is needed
- Even if detected, sometimes it can be hard to avoid an invalid mutation

Why not unstructure all the storage and avoid collisions everywhere, not just in the proxy?

```
contract ValueHolderV4 {
    struct ValueHolderStore {
        uint value;
        address setter;
        uint date;
    }

    event ValueSet(uint value, address setter, uint date);

    function _getStore() private pure returns (ValueHolderStore storage store) {
        assembly {
            store.slot := 5000
        }
    }

    function setValue(uint newValue) public {
        ValueHolderStore storage store = _getStore();

        store.value = newValue;
        store.setter = msg.sender;
        store.date = block.timestamp;

        emit ValueSet(newValue, msg.sender, block.timestamp);
    }

    function getValue() public view returns (uint) {
        return _getStore().value;
    }
}
```

deployed @ 0x7

0xb0b calls
setImplementation(0x7)

```
1  contract UnstructuredProxy {
2      struct ProxyStore {
3          address implementation;
4      }
5
6      function _getStore()
7          private
8          pure
9          returns (ProxyStore storage store)
10     {
11         assembly {
12             store.slot := 1000
13         }
14     }
15
16     function setImplementation(
17         address newImplementation
18     ) public {
19         _getStore().implementation = newImplementation;
20     }
21
22     fallback() external {
23         // Forwards any incoming call
24         // to the implementation
25         // using DELEGATECALL
26     }
27 }
```

Storage @ 0x4
0: 1337
1: 0xb0b
2: 0
...
1000: **0x7**

0x4

Event from **0x4**

```
contract UnstructuredProxy {
    struct ProxyStore {
        address implementation;
    }

    function _getStore()
        private
        pure
        returns (ProxyStore storage store)
    {
        assembly {
            store.slot := 1000
        }
    }

    function setImplementation(
        address newImplementation
    ) public {
        _getStore().implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
```

0x4

Execution context

0xb0b calls setValue(42)

DELEGATECALL

```
contract ValueHolderV4 {
    struct ValueHolderStore {
        uint value;
        address setter;
        uint date;
    }

    event ValueSet(uint value, address setter, uint date);

    function _getStore() private pure returns (ValueHolderStore storage store) {
        assembly {
            store.slot := 5000
        }
    }

    function setValue(uint newValue) public {
        ValueHolderStore storage store = _getStore();

        store.value = newValue;
        store.setter = msg.sender;
        store.date = block.timestamp;

        emit ValueSet(newValue, msg.sender, block.timestamp);
    }

    function getValue() public view returns (uint) {
        return _getStore().value;
    }
}
```

0x7

Storage @ 0x4

0: 1337

1: 0xb0b

2: 0

...

1000: 0x7

...

5000: **42**

5001: **0xb0b**

5002: **123456**

A proxy that works

- Execution context is kept at the proxy via DELEGATECALL
- Storage collisions are avoided by using unstructured storage everywhere
- Tooling should still be used to detect invalid storage mutations in the storage namespaces themselves
- This manual use of storage is much easier to control

Multi-contract systems and the Router Proxy

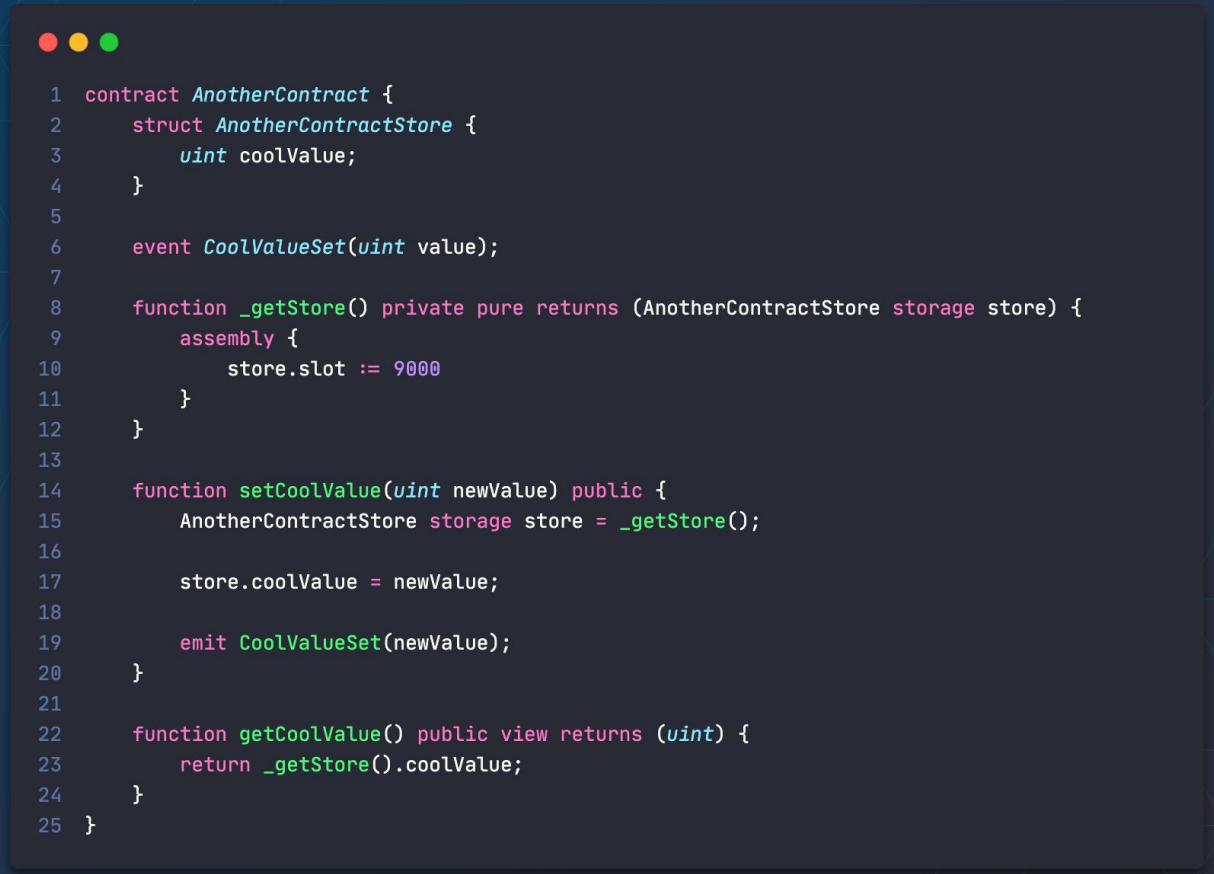
There is no ideal standard solution for multi-contract systems

Complex systems require many contracts.

These contracts need to talk to each other, and many times during these calls, they need to know if the call is coming from an official “system” contract.

People use registries, amongst other things, but in the end it all gets messy fast. And inefficient!

Let’s try a pretty crazy solution with the Router Proxy



```
contract AnotherContract {
    struct AnotherContractStore {
        uint coolValue;
    }

    event CoolValueSet(uint value);

    function _getStore() private pure returns (AnotherContractStore storage store) {
        assembly {
            store.slot := 9000
        }
    }

    function setCoolValue(uint newValue) public {
        AnotherContractStore storage store = _getStore();

        store.coolValue = newValue;

        emit CoolValueSet(newValue);
    }

    function getCoolValue() public view returns (uint) {
        return _getStore().coolValue;
    }
}
```

deployed @ 0x8

Bob builds and deploys a “Router”.

```
contract Router {
    address private constant _VALUE_HOLDER = 0x0000000000000000000000000000000000000007;
    address private constant _ANOTHER_CONTRACT = 0x0000000000000000000000000000000000000008;

    fallback() external payable {
        bytes4 sig4 = msg.sig;
        address implementation;

        assembly {
            let sig32 := shr(224, sig4)

            function findImplementation(sig) -> result {
                if lt(sig,0xbcfb658b) {
                    switch sig
                    case 0x20965255 { result := _VALUE_HOLDER } // ValueHolder.getValue()
                    case 0x55241077 { result := _VALUE_HOLDER } // ValueHolder.setValue(...)
                    leave
                }
                switch sig
                case 0xbcfb658b { result := _ANOTHER_CONTRACT } // AnotherContract.getCoolValue()
                case 0xced15014 { result := _ANOTHER_CONTRACT } // AnotherContract.setCoolValue()
                leave
            }

            implementation := findImplementation(sig32)
        }

        assembly {
            // Forwards any incoming call
            // to the implementation
            // using DELEGATECALL
        }
    }
}
```

deployed @ 0x9

Bob sets the router as the proxy’s implementation.

0xb0b calls
setImplementation(0x9)

```
1  contract UnstructuredProxy {
2      struct ProxyStore {
3          address implementation;
4      }
5
6      function _getStore()
7          private
8          pure
9          returns (ProxyStore storage store)
10     {
11         assembly {
12             store.slot := 1000
13         }
14     }
15
16     function setImplementation(
17         address newImplementation
18     ) public {
19         _getStore().implementation = newImplementation;
20     }
21
22     fallback() external {
23         // Forwards any incoming call
24         // to the implementation
25         // using DELEGATECALL
26     }
27 }
```

Storage @ 0x4
0: 1337
1: 0xb0b
2: 0
...
1000: **0x9**
...
5000: 42
5001: 0xb0b
5002: 123456

0x4

Event from **0x4**

```
contract UnstructuredProxy {
    struct ProxyStore {
        address implementation;
    }

    function _getStore()
        private
        pure
        returns (ProxyStore storage store)
    {
        assembly {
            store.slot := 1000
        }
    }

    function setImplementation(
        address newImplementation
    ) public {
        _getStore().implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
```

0x4

Execution context

DELEGATECALL

```
contract Router {
    address private constant _VALUE_HOLDER = 0x0000000000000000000000000000000000000007;
    address private constant _ANOTHER_CONTRACT = 0x0000000000000000000000000000000000000008;

    fallback() external payable {
        bytes4 sig4 = msg.sig;
        address implementation;

        assembly {
            let sig32 := shr(224, sig4)

            function findImplementation(sig) -> result {
                if lt(sig,0xbcfb658b) {
                    switch sig
                    case 0x20965255 { result := _VALUE_HOLDER } // ValueHolder.getValue()
                    case 0x55241077 { result := _VALUE_HOLDER } // ValueHolder.setValue(...)
                    leave
                }
                switch sig
                case 0xbcfb658b { result := _ANOTHER_CONTRACT } // AnotherContract.getCoolValue()
                case 0xced15014 { result := _ANOTHER_CONTRACT } // AnotherContract.setCoolValue()
                leave
            }

            implementation := findImplementation(sig32)
        }

        assembly {
            // Forwards any incoming call
            // to the implementation
            // using DELEGATECALL
        }
    }
}
```

0x9

DELEGATECALL

```
contract AnotherContract {
    struct AnotherContractStore {
        uint coolValue;
    }

    event CoolValueSet(uint value);

    function _getStore() private pure returns (AnotherContractStore storage store) {
        assembly {
            store.slot := 9000
        }
    }

    function setCoolValue(uint newValue) public {
        AnotherContractStore storage store = _getStore();

        store.coolValue = newValue;

        emit CoolValueSet(newValue);
    }

    function getCoolValue() public view returns (uint) {
        return _getStore().coolValue;
    }
}
```

0x8

Storage @ 0x4

0: 1337

1: 0xb0b

2: 0

...

1000: 0x9

...

5000: 42

5001: 0xb0b

5002: 123456

...

9000: **7**

0xb0b calls setCoolValue(7)

Bob upgrades one of the system's modules.

```
1  contract AnotherContractV2 {
2      struct AnotherContractStore {
3          uint coolValue;
4      }
5
6      event CoolValueSet(uint value);
7
8      function _getStore() private pure returns (AnotherContractStore storage store) {
9          assembly {
10             store.slot := 9000
11         }
12     }
13
14     function setCoolValue(uint newValue) public {
15         AnotherContractStore storage store = _getStore();
16
17         store.coolValue = newValue * 7;
18
19         emit CoolValueSet(newValue);
20     }
21
22     function getCoolValue() public view returns (uint) {
23         return _getStore().coolValue;
24     }
25 }
```

deployed @ 0x10

```
contract RouterV2 {
    address private constant _VALUE_HOLDER = 0x0000000000000000000000000000000000000007;
    address private constant _ANOTHER_CONTRACT = 0x0000000000000000000000000000000000000010;

    fallback() external payable {
        bytes4 sig4 = msg.sig;
        address implementation;

        assembly {
            let sig32 := shr(224, sig4)

            function findImplementation(sig) -> result {
                if lt(sig,0xbcfb658b) {
                    switch sig
                    case 0x20965255 { result := _VALUE_HOLDER } // ValueHolder.getValue()
                    case 0x55241077 { result := _VALUE_HOLDER } // ValueHolder.setValue(...)
                    leave
                }
                switch sig
                case 0xbcfb658b { result := _ANOTHER_CONTRACT } // AnotherContract.getCoolValue()
                case 0xced15014 { result := _ANOTHER_CONTRACT } // AnotherContract.setCoolValue()
                leave
            }

            implementation := findImplementation(sig32)
        }

        assembly {
            // Forwards any incoming call
            // to the implementation
            // using DELEGATECALL
        }
    }
}
```

deployed @ 0x11

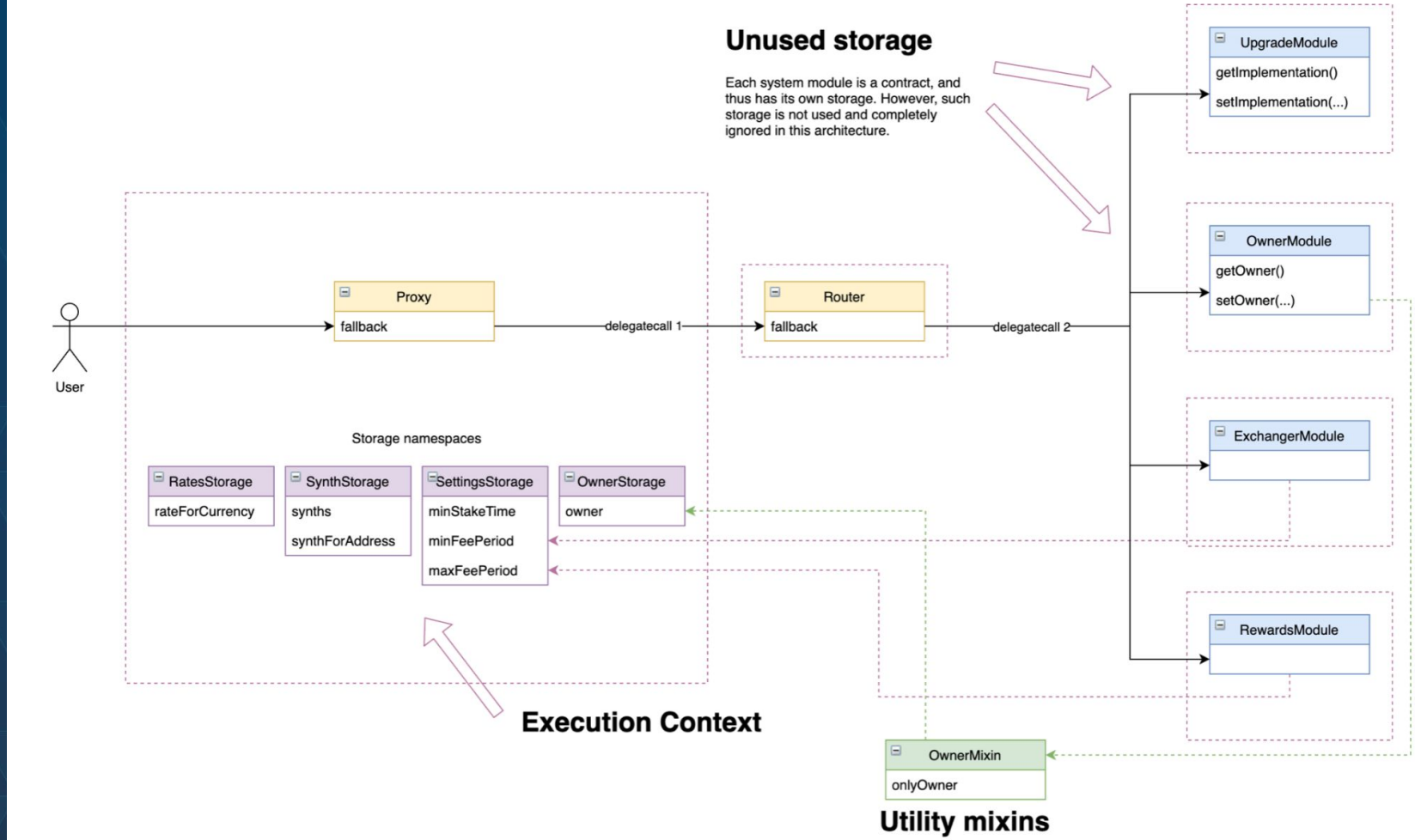
0xb0b calls
setImplementation(0x11)

```
contract UnstructuredProxy {
    struct ProxyStore {
        address implementation;
    }

    function _getStore()
        private
        pure
        returns (ProxyStore storage store)
    {
        assembly {
            store.slot := 1000
        }
    }

    function setImplementation(
        address newImplementation
    ) public {
        _getStore().implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
```

Storage @ 0x4
0: 1337
1: 0xb0b
2: 0
...
1000: **0x11**
...
5000: 42
5001: 0xb0b
5002: 123456

0x4

What would a more complex system look like?


Unused storage
Each system module is a contract, and thus has its own storage. However, such storage is not used and completely ignored in this architecture.
UpgradeModule
getImplementation()
setImplementation(...)
OwnerModule
getOwner()
setOwner(...)
ExchangerModule
RewardsModule
User
Proxy
fallback
delegatecall 1
Router
fallback
delegatecall 2
Storage namespaces
RatesStorage
rateForCurrency
SynthStorage
synths
synthForAddress
SettingsStorage
minStakeTime
minFeePeriod
maxFeePeriod
OwnerStorage
owner
Execution Context
OwnerMixin
onlyOwner
Utility mixins

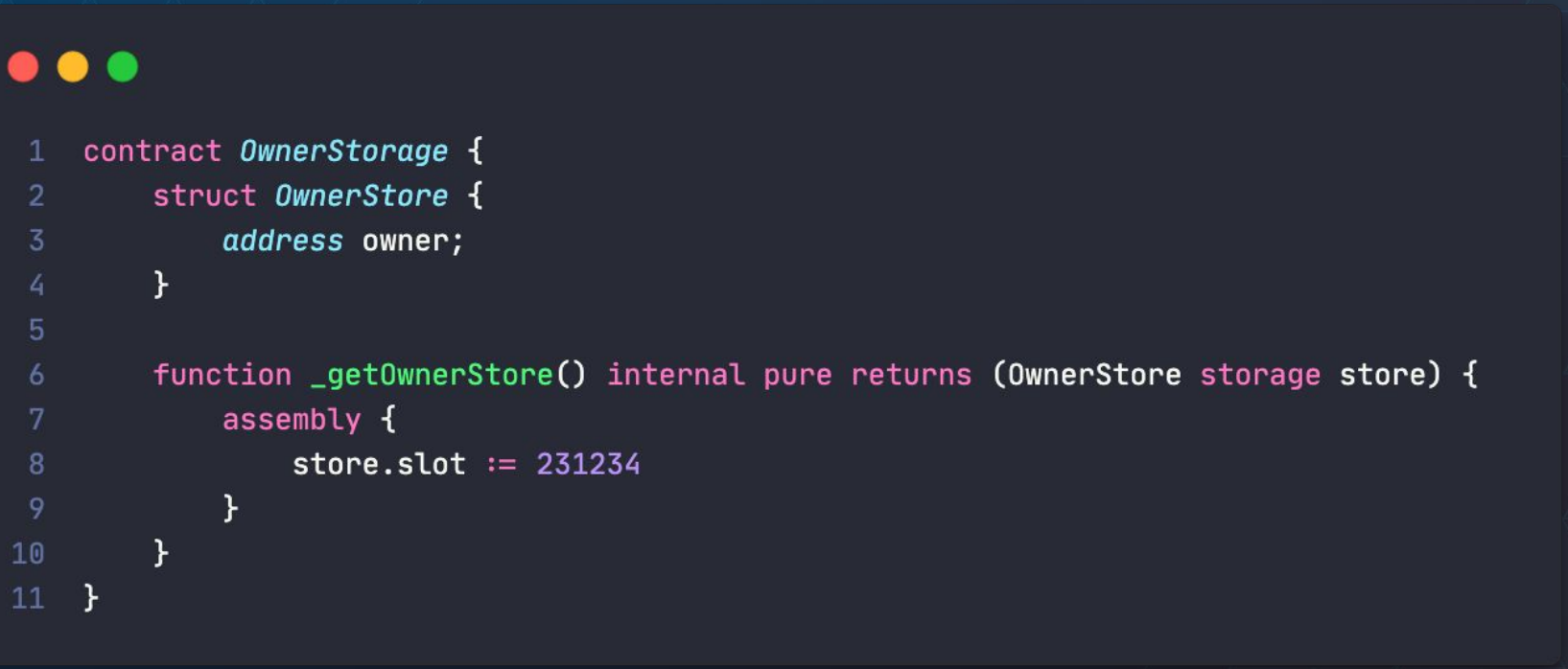
Gas efficiency

- A transparent proxy uses ~3000 gas
- A UUPS proxy uses ~1600 gas
- This proxy and router uses ~2600 gas

Inter-modular communications

- All modules ***are*** the system
- Cast and **call** self?
  - from ModuleB: ModuleA(this).doSomething()
  - but msg.sender is lost
- Cast and **delegatecall** self?
  - works
- Mixins
  - Pieces of code that know how to interact with storage
  - Not deployed, inherited
  - Act as delegates of another module
  - Avoids calls altogether!
  - Communication becomes super cheap

Mixin example


contract OwnerStorage {
struct OwnerStore {
address owner;
}
function _getOwnerStore() internal pure returns (OwnerStore storage store) {
assembly {
store.slot := 231234
}
}
}

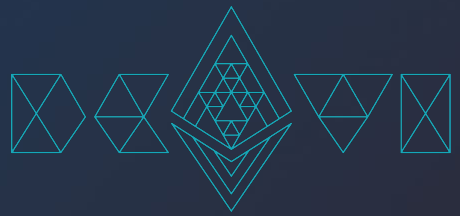
```
contract OwnerMixin is OwnerStorage {
    error Unauthorized(address who);

    modifier onlyOwner() {
        OwnerStore storage store = _getOwnerStore();

        if (msg.sender != store.owner) {
            revert Unauthorized(msg.sender);
        }

        _;
    }
}
```

```
contract OwnerModule is OwnerMixin {
    function setOwner(address newOwner) public onlyOwner {
        OwnerStore storage store = _getOwnerStore();

        store.owner = newOwner;
    }

    function getOwner() internal view returns (address) {
        return _getOwnerStore().owner;
    }
}
```

```
contract ValueHolderV5 is OwnerMixin {
    struct ValueHolderStore {
        uint value;
        address setter;
        uint date;
    }

    event ValueSet(uint value, address setter, uint date);

    function _getStore() private pure returns (ValueHolderStore storage store) {
        assembly {
            store.slot := 5000
        }
    }

    function setValue(uint newValue) public onlyOwner {
        ValueHolderStore storage store = _getStore();

        store.value = newValue;
        store.setter = msg.sender;
        store.date = block.timestamp;

        emit ValueSet(newValue, msg.sender, block.timestamp);
    }

    function getValue() public view returns (uint) {
        return _getStore().value;
    }
}
```

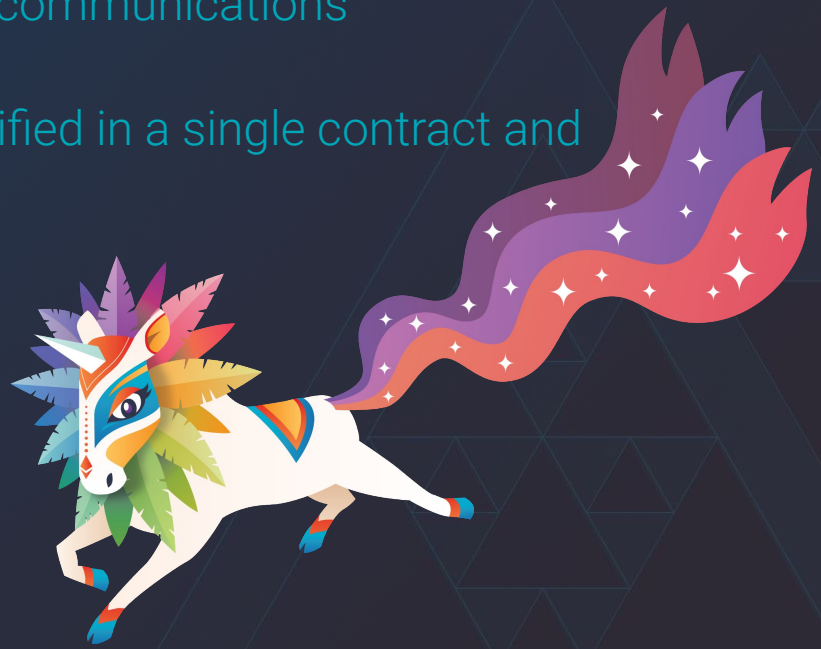
Slight code style change

- Can’t use “regular” Solidity variables
- Must use storage namespaces always
- It’s not as weird as it looks, and you get used to it fast
- Should Solidity do this under the hood?
  - Proposed by maxsam4: https://github.com/ethereum/solidity/issues/11102



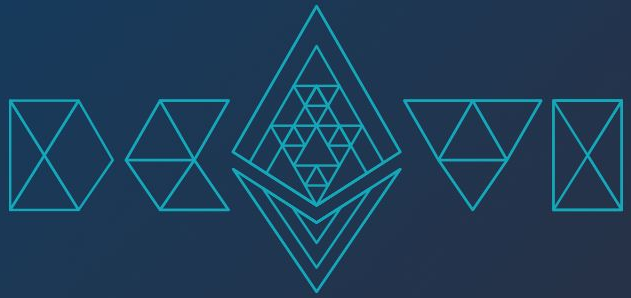
Why use the router?

- No more contract size limits!
  - All contracts are “merged” into one
- Simplified and efficient inter-modular communications
  - Ideal for complex systems
- System composition is explicitly specified in a single contract and upgraded with a single tx
  - Ideal for governance
- Core component of Synthetix v3



How to use the router?

- Hardhat plugin @ https://github.com/Synthetixio/synthetix-v3/tree/main/packages/hardhat-router
- Generates the router
- Manages storage namespaces
- Performs validations to ensure that there are no storage collisions



Thank you!

Alejandro Santander a.k.a.
Ethernaut

Core Contributor @ Synthetix

@the_ethernaut