Smart Contract With Patract Labs

Bonan Yuan

Patract

Who We Are?

Patract Labs provides full-stack support for smart contract development based on Substrate.

Jupiter
Contract Testnets

Ask!
Assembly Script eDSL for WASM

Redspot
Contract Dev Scaffold

Metis
Standard Contract Library

Europa
Contract Sandbox Env

Elara
High Availability Substrate API

Parascan
Blockchain Explorer

Himalia
Multi Contract SDKs

Patract

Current Problems With Contract Development

Our Solution

- **Lack of Multi Language Support.**
  Developers can only program in Rust even WASM supports multi-language compilation.

→ **Ask!**

- **Lack of Development Suites.**
  Developers have to deploy and test contract manually.

→ **Redspot**

- **Lack of Standard Contract Library similar to Openzeppelin.**
  Developers have to manually copy and paste the code, which is time-consuming and error-proning.

→ **Metis**

- **Lack of Test Sandbox.**
  There is no out-of-box contract test sandbox designed specifically for contract development and debugging.

→ **Europa**

Ask!—Bring AssemblyScript to Substrate

eDSL	• Similar to ink!, ask! Is eDSL (embedded domain-specific language).
Decorators	• Using Decorators to interact with `pallet-contracts` • High level abstraction, developers can focus on the contract logics.
Storage	• More storages types: `SpreadStorableMap` `SpreadStorableArray` `PackedStorableMap` `PackedStorableArray`
Interface and Inheritance	• Solidity developers can quick get used to ask! • Support Standard Contract Library similar to Openzeppelin.
Under development	• We are at version v0.2 and will publish v0.3 soon.

Redspot—Development Suites

- **Template**
  - Based on contract standard, automatically generate sample contract, deploy script, test script and Redspot Config.
  - `npx redspot-new erc20`
- **Compile**
  - Wraps around `cargo-contract`
  - `npx redspot compile`
- **Deploy**
  - Upload the contract and instantiate it using script.
  - `npx redspot run scripts/deploy.ts --no-compile`
- **Test**
  - Test contract methods using script.
  - `npx redspot test --no-compile`

Redspot—Flexible and Plugable

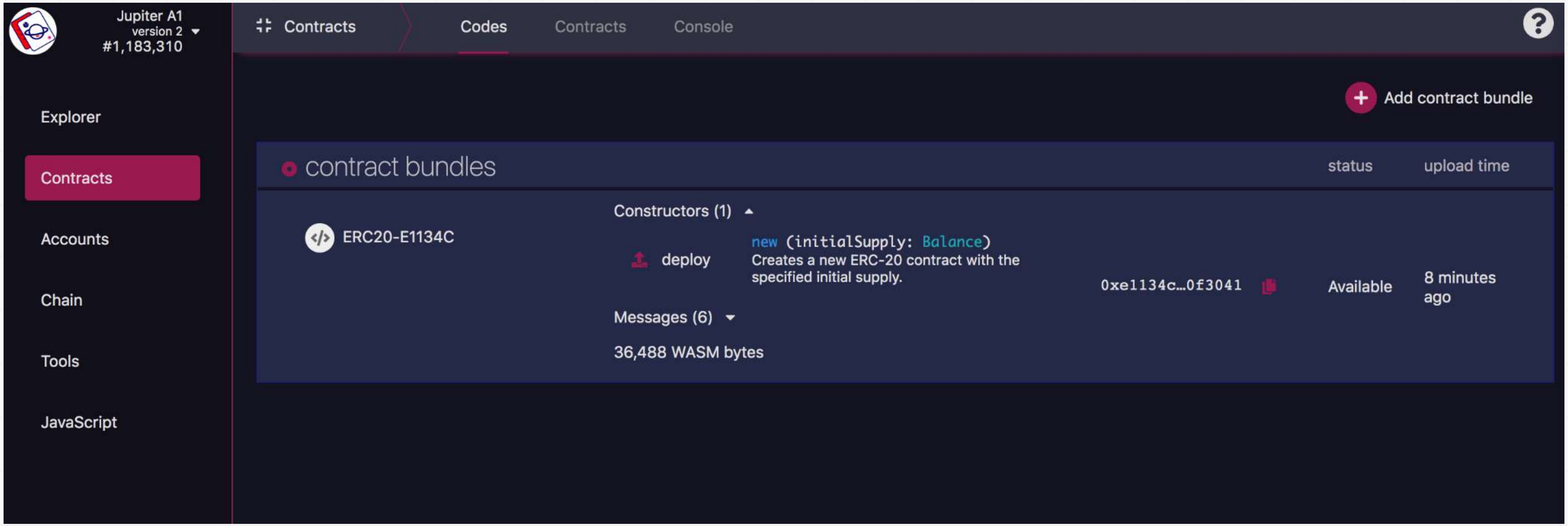
- **Console**
  - Powerful JavaScript Interactive Console for unit testing.
  - `npx redspot console --no-compile`

- **Docker**
  - Test your contract on multiple platforms
  - Start Docker: `$ npx redspot testnet`
  - Compile on Docker: `$ npx redspot compile --docker true`

- **Explorer**
  - Highly integrated into Respot with no more configurations
  - `npx redspot explorer`

- **Plugin**
  - Extend Redspot with custom plugins
  - Stay compatible with Custom Substrate Chain by extending types

Redspot—Explore



Metis—Rusty Solution For Standard Contract Library

Current problems when developing in ink!

- **Lack of standard library makes the smart contract unsafe.**
  - The DAO attack in 2016 stole 3.6m Ether due to a bug in the smart contract. This attack enforced hardfork for Ethereum Mainnet.
- **Developers have to manually copy/paste existing implementation.**
  - The source code could be unaudited and unsafe to use.
  - Time consuming and error-proning during the process.

Metis

- **Standard Contract library based on reusable components.**
  - Unlike Openzeppelin's Inheritance model, Metis is based on Components, which means users do not directly inherit the standard implementation. Instead, Metis provides a set of reusable components for users to assemble.

Metis—Composition, not inheritance

Inheritance — *Used by Openzeppelin*

- **Pros:**
  - **Simplicity :** Minimize the code to write for developers.
- **Cons:**
  - **Ambiguity :** Conceal method definitions; Uncertain Inheritance tree with multiple Inheritance.

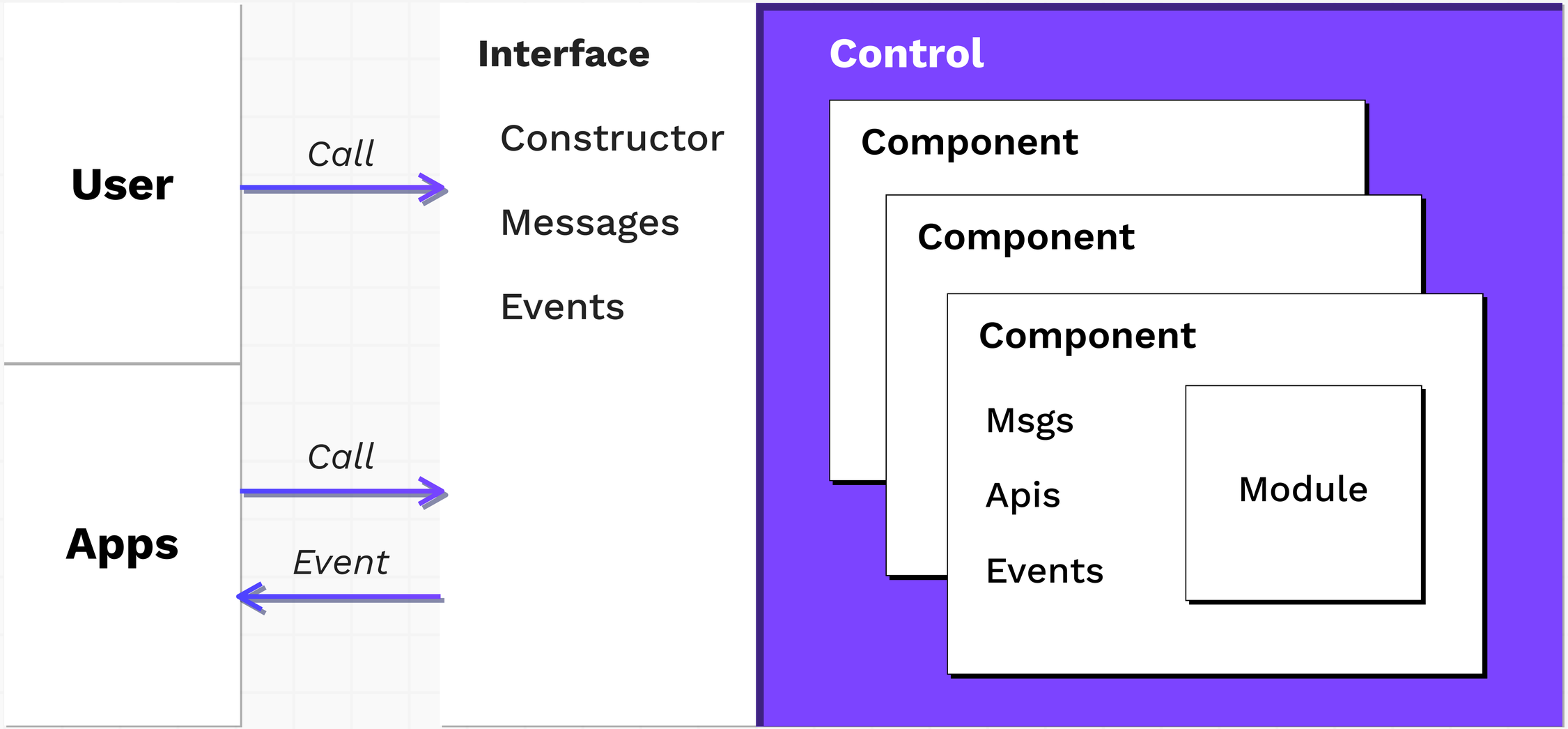
Composition — *Used by Metis*

- **Pros:**
  - **Clarity :** Improve code readability and audibility.
- **Cons:**
  - **Repetition :** Repeat writing the same code for existing implementation.

Metis—MCCI architecture

M *Data model*	• Contract State mapping. • One data model per component.
C *Component*	• Reusable. • Independent. • Encapsulation of data and methods. • Orthogonality with other components.
C *Controller*	• Coordinates the components and implements the contract interface.
I *Interface*	• The interface is the user interface of the contract. • The interface defines the behavior of the contract and, to some extent, defines metadata.

Metis—MCCI architecture



Metis—vs Native ink! (Storage)

Standardize Storage structure for ERC20 token

ink!

```
#[ink(storage)]
    pub struct Erc20 {
        /// Total token supply.
        total_supply: Lazy<Balance>,
        /// Mapping from owner to number of owned token.
        balances: StorageHashMap<AccountId, Balance>,
        /// Mapping of the token amount which an account is allowed to
withdraw
        /// from another account.
        allowances: StorageHashMap<(AccountId, AccountId), Balance>,
    }
```

Metis

```
#[ink(storage)]
    #[import(erc20)]
    pub struct Erc20 {
        erc20: erc20::Data<Erc20>,
    }
```

Metis—vs Native ink! (Constructor)

Skip the process of variable declaration; No need to manually emit event

Metis

```
#[ink(constructor)]
pub fn new(initial_supply: Balance) -> Self {
    let mut instance = Self {
        erc20: erc20::Data::new(),
    };

    erc20::Impl::init(
        &mut instance,
        String::from("MetisTestToken"),
        String::from("MET"),
        18_u8,
        initial_supply,
    );

    instance
}
```

ink!

```
#[ink(constructor)]
pub fn new(initial_supply: Balance) -> Self {
    let caller = Self::env().caller();
    let mut balances = StorageHashMap::new();
    balances.insert(caller, initial_supply);
    let instance = Self {
        total_supply: Lazy::new(initial_supply),
        balances,
        allowances: StorageHashMap::new(),
    };
    Self::env().emit_event(Transfer {
        from: None,
        to: Some(caller),
        value: initial_supply,
    });
    instance
}
```

Metis—vs Native ink! (Event)

Stays pretty much the same due to current design of Ink!. Will mitigate in near future

Metis

```
#[ink(event)]
#[metis(erc20)]
pub struct Transfer {
    #[ink(topic)]
    from: Option<AccountId>,
    #[ink(topic)]
    to: Option<AccountId>,
    value: Balance,
}
```

ink!

```
#[ink(event)]
pub struct Transfer {
    #[ink(topic)]
    from: Option<AccountId>,
    #[ink(topic)]
    to: Option<AccountId>,
    value: Balance,
}
```

Metis—vs Native ink! (Message)

Patract

Contains default method implementation that ensures the security of balance transfer

Metis

```
#[ink(message)]
pub fn transfer_from(
    &mut self,
    from: AccountId,
    to: AccountId,
    value: Balance,
) -> Result<()> {
    erc20::Impl::transfer_from(self, from, to, value)
}
```

ink!

```
#[ink(message)]
pub fn transfer_from(
    &mut self,
    from: AccountId,
    to: AccountId,
    value: Balance,
) -> Result<()> {
    let caller = self.env().caller();
    let allowance = self.allowance(from, caller);
    if allowance < value {
        return Err(Error::InsufficientAllowance)
    }
    self.transfer_from_to(from, to, value)?;
    self.allowances.insert((from, caller), allowance - value);
    Ok(())
}
```

Metis—Design Principle

Why Can't we design the library similar to the right side?

- Instantiation means create a instance of smart contract on the blockchain.
- Calling method of another instance means calling method from another contract.
- Cross contract calling ends up with wrong result.

Desired Implementation (pseudocode):

```
contract MyErc20 {
    ERC20 myerc20 = erc20.new(some params);
    fn mint(int amount){
        self.myerc20.mint(amount)
    }
    fn balance(){
        self.myerc20.balance()
    }
}
```

Metis

Composition Of Multiple Components

ERC20 Pausable - Constructor

```
impl Erc20Pausable {
    #[ink(constructor)]
    pub fn new(initial_supply: Balance) -> Self {
        let mut instance = Self { erc20: erc20::Data::new(), ownable:
ownable::Data::new(), pausable: pausable::Data::new(),};
        erc20::Impl::init( &mut instance,
String::from("MetisTestToken"),
            String::from("MET"), 18_u8, initial_supply,);
        ownable::Impl::init(&mut instance);
        pausable::Impl::init(&mut instance);
        instance
    }
```

```
#[ink(storage)]
#[import(erc20, ownable, pausable)]
pub struct Erc20Pausable {
    erc20: erc20::Data<Erc20Pausable>,
    ownable: ownable::Data<Erc20Pausable>,
    pausable: pausable::Data,
}
```

ERC20 Pausable - Storage

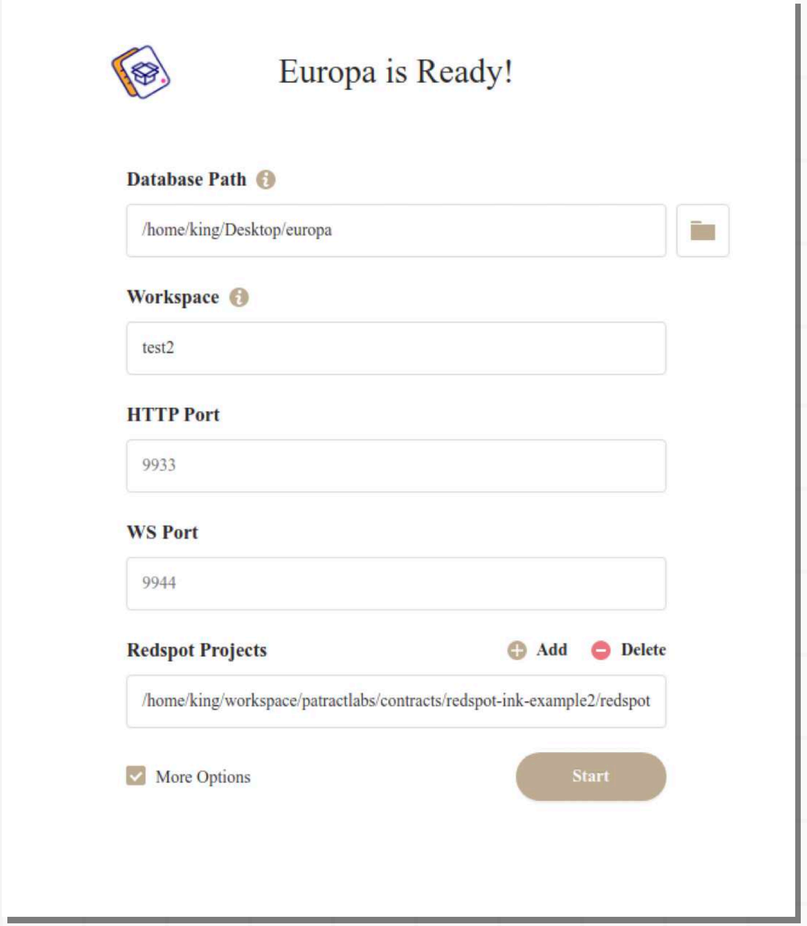
Patract

Europa—Out-of-box Local Test Net

- **Another Substrate Implementation** *Focus on smart contract development*
- **No consensus** *Only produce blocks when receiving extrinsics*
- **No WASM Runtime**
- **State KV Database** *Trace block state changes*
- **Contract Pallet Modification** *Reveal the blackbox of contract execution*

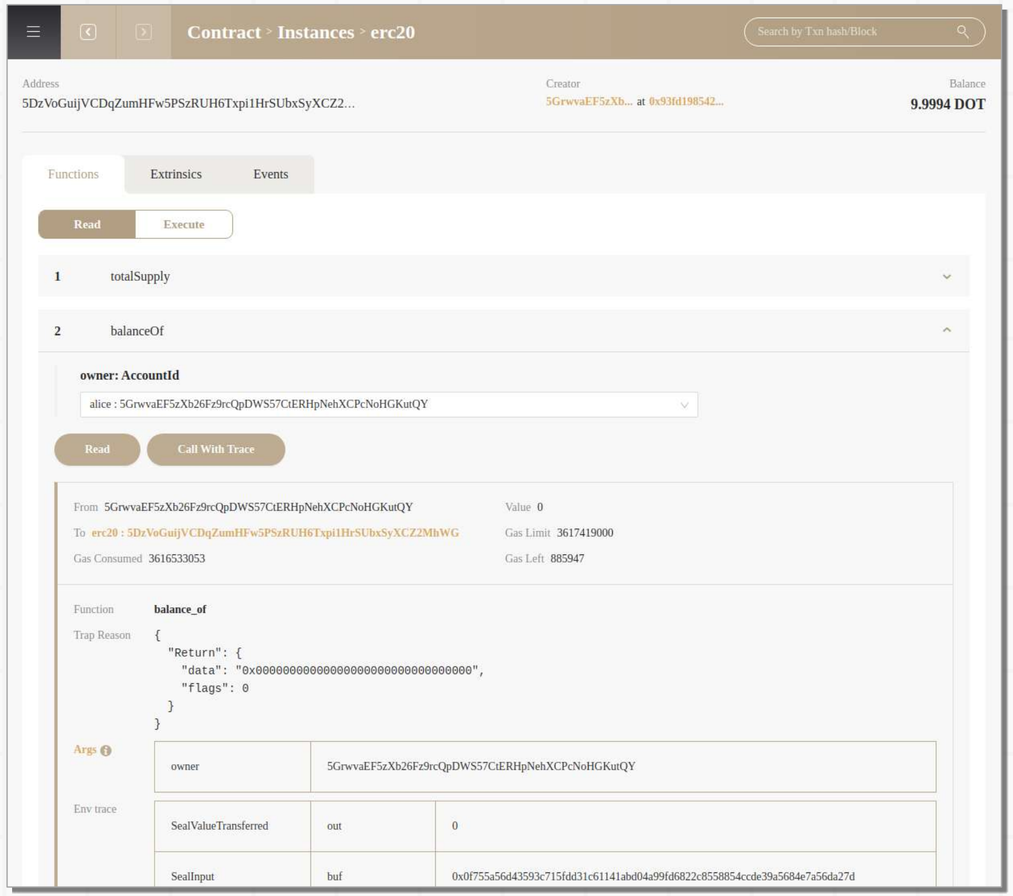
Europa UI—One Click Substrate node with GUI

- Europa UI is specially designed for contract development.
- Download the binary release and start your own node with one click
- Available on all platforms

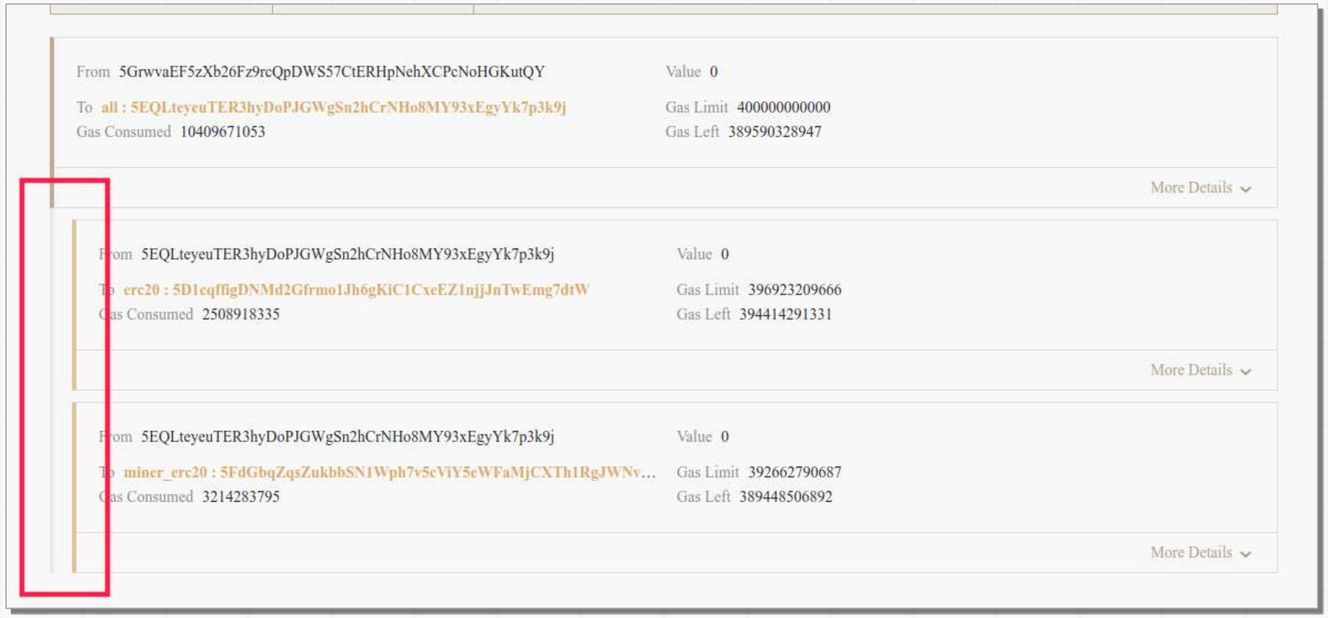


Europa UI

Contract Pages



Nested View of Cross Contract Call



Europa CLI—No More Blackbox For Contract Execution

Custom RPCs and Commands

- **RPCs**
  - europa_forwardToHeight
  - europa_backwardToHeight
  - europa_modifiedStateKvs
- **Commands**
  - State-kv
  - Workspace

Detailed Logging for Contract Execution

- **Trace all parameters and blockchain changes during contract execution**
- **WASM panic backtrace**

Patract

Demo

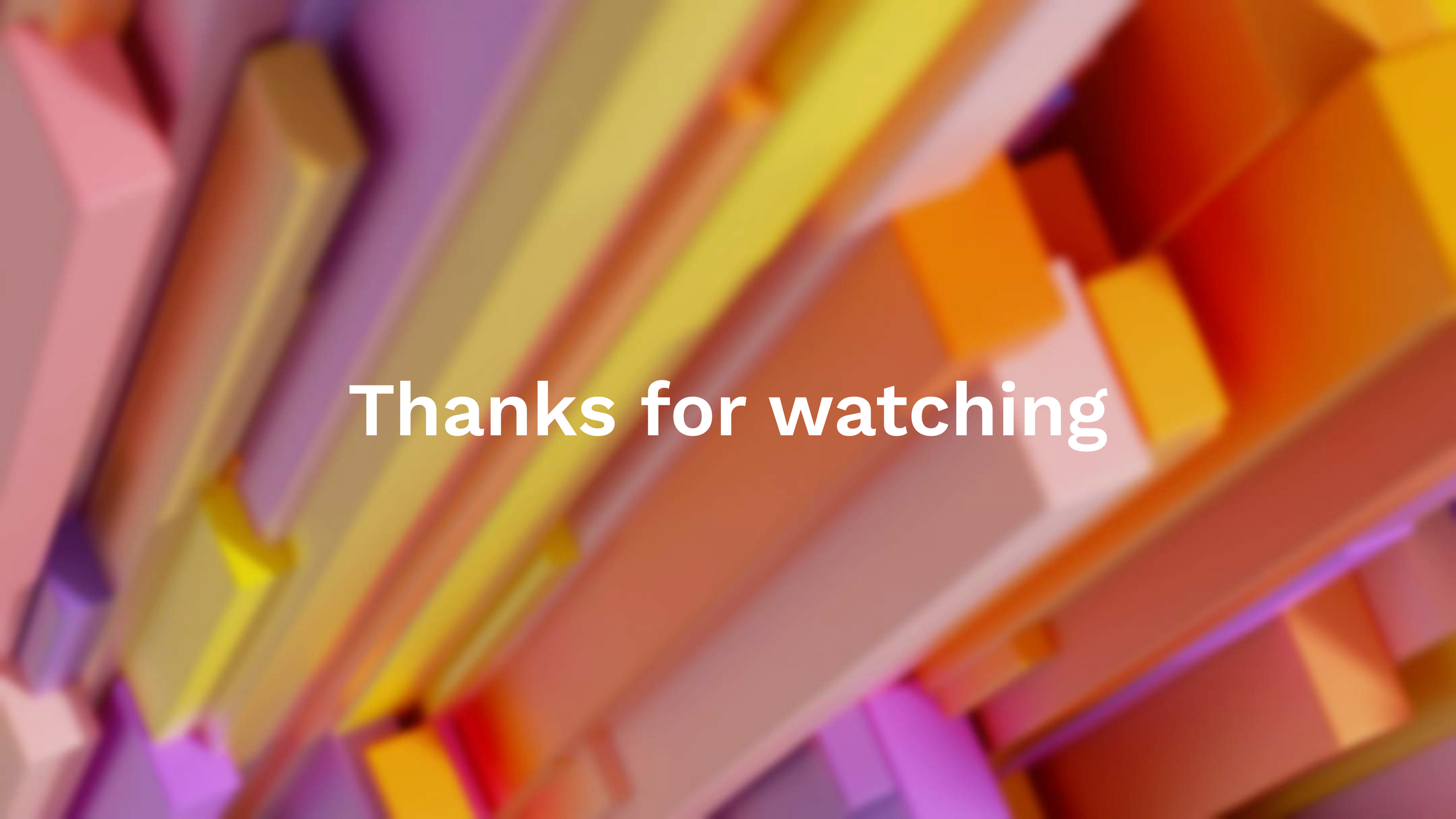
1 Use Redspot to quickly setup development environment with built-in templates

2 Start Europa UI

3 Showcase writing an ERC20 Pausable Smart contract with Access Control

4 Compile and deploy the smart contract using Redspot

5 Test the smart contract methods using Europa



Thanks for watching