Learn ink! - Order Food on Blockchain

Why learn ink!

There are many blockchain smart contracts: Solidity, Vyper, Yul, Cairo, Move, and ink!, etc., supporting different virtual machines running on different blockchains, each with pros and cons,

As compared to Ethereum and its EVM, the Polkadot blockchain, with its unique architecture, WASM (Web Assembly) virtual machine, and ink! Smart contract language, is a growing ecosystem attracting increasing developer adoption.

Parity Technologies, the company behind Polkadot led by Kevin Woods, has ambition to build the Web 3.0 foundation, from the Substrate blockchain framework to Polkadot to ink!, its sharded protocol and tools enabling parachains to operate seamlessly at scale. In this article, Why we believe in Wasm as the base layer of decentralized application development, Parity shared its belief and offered ink! as a solution.

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on all devices, from server to browser, with great portability. Already endorsed by industry heavy weights and popular in the web2 world, there's increasing indication that Wasm VM is gaining traction as an EVM alternative. For example, Abitrum adopted Wasm for fraud proof, in parallel to the same smart contract code compiled to run on EVM.

ink! Is a Rust-based domain specific language (DSL) for writing Wasm based smart contracts for Substrate blockchains. Compared to Solidity, Ink! Is a relatively new kid yet with cool features and great potential. Being a community focused project, it needs developers to share valuable learning experiences and input for continuous improvements. This article is meant to do exactly that, with objective learning and development experiences, evaluating and climbing the ink! learning curve.

Learning by doing in 3 stages

The project's plan is to come up with an intuitive example, implement in ink!, and share learnings with the community. Think it is a technical showcase with simple yet decent complexity to demonstrate 3 chosen aspects, as compared to Solidity as a contrast.

1. ink! Smart contract and macros for code simplicity, reusability and readability
2. Upgradability of Ink! based smart contracts
3. Performance benchmark and gas cost savings

There's a lot of details to go over, so we'll progress in corresponding 3 stages, with takeaways documented in a series of 3 articles, each covering respective areas. This is the first of the series to focus on macros in ink! smart contract development. .

The series could potentially benefit several types of audiences:
- Developers wanting to check out alternative smart contract languages
- Rust developers wanting to do blockchain based smart contract development
- Ink! novice wanting to learn more about declarative and procedure macros
- Solidity / Vyper developers comparing with other smart contract development paradigms
- Network operators wanting to know more about smart contract upgrade practices
- Web2 developers wanting to get a taste of coding smart contracts using their familiar programming language such as Rust
- EVM or compatible VM developers concerning performance and gas fees
- Blockchain researchers surveying on vs off-chain computing and storage

The use case: Order Food on Blockchain

Current ink! document uses a very simplistic example called Flipper, which switches a boolean on and off. It lacks the sophistication to showcase the full potential of the ink! language. In this example, we came up with a contrived workflow example, where multiple parties: restaurants, couriers, consumers and dApp admin interact to register, order, prepare and deliver food on chain.

We chose to define and implement this use case for multiple reasons:

1. A more sophisticated business workflow with multiple parties leveraging blockchain to conduct transactions
2. There are reusable business logic to be encapsulated and implemented in macros
3. Demonstrate the end-to-end process of Wasm based smart contract upgrades, and best practices to best preserve states and to avoid hard fork
4. Implementation best practices demonstrating techniques to improve performance, reduce binary footprint, and lower gas fees

Here's the main flow of this Order Food on Blockchain example, with specific code open sourced and available in Github.

Personas

- Consumers- can submit order to buy food, pay the food and delivery service fees, confirm delivery, using units of tokens supported by the underlying deployment network
- Restaurants - add / update / delete menu including dish name, description, price and eta (estimated time to prepare the dish), confirm order, prepare food and request delivery, get paid
- Couriers - delivery food to customer upon food delivery request, get paid service fees, which is a percentage of order total, upon successfully confirmed food delivery,
- dApp manager - administrative tasks: access control, list restaurants, couriers and customers, approve service fee rate, change smart contract ownership, etc.
- dApp users - anyone can self register in dApp with wallet and info, list all restaurants, all food, all couriers, register to be service providers (restaurants, couriers), or submit customers to order food

Workflow

- Consider this example implements a mini workflow with multiple parties forming a marketplace with service providers (restaurants, couriers) and service consumers interacting with each other in this blockchain based decentralized application.
- Service providers can register / update services such as restaurant info, menu including food, estimated wait time, prices; or couriers on delivery service rates.
- Consumers can browse available restaurants, its offered food and price, place order and pick a courier service to deliver food
- All marketplace participants has wallet addresses registered to send / receive payments
- dApp provides brief escrow from receiving payment on submitted order, to split the proceeds sending food subtotal to restaurant upon food picked up event, as well as sending delivery charges to courier upon food delivered event.
- The mini workflow implements a state machine that's part of the smart contract storage with each state transition captured on-chain. For example an order has multiple states in its lifecycle: OrderSubmitted, OrderConfirmed, FoodPrepared, FoodDelivered, DeliveryAcceptted.

- The smart contract emits an event for every state transition or error, which triggers the next step in the state machine.

Obviously this contrived example meant for learning and sharing with a moderate level of complexity to demonstrate ink! Based implementation. In the real world, marketplace participants can leverage blockchain to trustlessly transact among themselves handling enterprise grade workflows.

Open source community can take it further, by adopting a workflow spec, written by non technical users, or auto generated via workflow DSL to smart contract compiler, to conduct business on blockchain. Connected with an intuitive frontend and tools, enterprises across the globe, regardless of large or small, can leverage the open ledger as a trust layer, interact and transact transparently on a leveled playing field, achieving the next level of efficiency, scalability and cost effectiveness.

ink! Implementation

The entire example is implemented in ink! smart contract language, leverage on selected features from openBrush, in the Substrate framework interacting with runtime pallets. Developers familiar with Rust should be able to read the open source code with ease as ink! simply a Rust based DSL.

For non-Rust based smart contract developers, it might still be a worthwhile exercise to toy with the code / sandbox to get a sense of how ink! based smart contract development's like as compared to for example Solidity experiences. More data points will be shared on this smart contract's performance benchmark, binary size, gas costs via various optimization techniques, in part three of this series.

Some highlights of the data, services, security and access control in this ink! Implementation:

- Top level data types, implemented via openBrush's Mapping, each type has its own corresponding fields / sub-structures.
    - Food
    - Order
    - Delivery
    - Customer
    - Restaurant
    - Courier
    - Manager
- Services - the smart contract defines and implements multiple service categories:
    - Restaurant services
    - Courier service
    - Customer services
    - Payment services

    - Manager services
- Data is persisted in the smart contract storage, services are transactions going into blocks. How to preserve and minimize smart contract states from version to version during smart contract upgrades, is what's the part2 of this series will be focus on
- The smart contract uses OpenBrush's ownable to set / get owners with corresponding access control to crud (create, read, update, delete) data. For example restaurants and couriers are owners and only owners that can admin their respective services.
- Read the section below for an openBrush vs openZeppelin comparison on ownable and access control features.
- The entire codebase is available in this open source Github repo under Apache 2 license..

Security & Access Control

OpenBrush is a smart contract development framework for ink! that provides a number of features to help developers build secure and reliable contracts. It is an open source project aims to provide similar capabilities as OpenZeppelin for Solidity smart contract developers.

OpenBrush attempts to provide similar security and access control features like OpenZeppelin for ink! smart contract development: Ownable extensions and AccessControl with AccessControlEnumerable extension,

Ownable

Ownable is for implementing ownership in smart contracts. This is the most common and basic form of access control, where there's an account that is the owner of a contract and can administer it. OpenBrush's Ownable contract provides the following security features:

- A public modifier called onlyOwner that can be used to restrict access to certain functions.
- A public function called transferOwnership that can be used to transfer ownership of the contract to another address.
- A public function called isOwner that can be used to check whether the current account is the owner of the contract.
- A public function called renounceOwnership that can be used to renounce ownership of the contract.

The renounceOwnership function is a unique feature of OpenBrush's Ownable contract. This function allows the owner of the contract to permanently give up ownership of the contract. This can be useful in situations where the owner no longer wants to be responsible for the contract.

Access Control

Role-Based Access Control (RBAC) offers flexibility in defining multiple roles. OpenZeppelin's AccessControl library provides a number of features to help developers control access to their contracts.

- Roles: Developers can define roles for different users and groups of users.
- Permissions: Developers can assign permissions to roles, such as the ability to transfer tokens, create new contracts, or call specific functions.
- Policies: Developers can define policies that control how permissions are applied. For example, a policy might require that two approvers sign a transaction before it can be executed.

OpenBrush's AccessControl library provides similar features, but it also includes a number of additional features:

- Timed permissions: Developers can specify that permissions expire after a certain amount of time.
- Multi-factor authentication: Developers can require users to provide multiple forms of authentication, such as a password and a one-time code, before they can access a contract.
- Audit trails: Developers can track who has accessed their contracts and what they have done.

ink! macros

Macros is a great way to encapsulate reusable logic to improve smart contract code simplicity, reusability and readability. ink! macros are a way to define custom syntax or code that can be reused within a smart contract. Macros in ink! are defined using the Rust programming language's macro system, which provides a powerful set of tools for code generation and metaprogramming.

Ink! macros get translated into actual rust code during compilation time. The smart contract code has better readability, maintainability and is simplified. To add another menu operation, for example, adding food pictures for each menu item, a developer can expand the same macro to support additional functionalities, making it easier to share among dApp developers to invoke the macro with increasing operations as a reusable component similar to code library.

The code reusability is important for adoption as it provides a way to modularize a set of related functionalities, reuse shared macros across open source projects, and accelerate ink! smart contract development aided by easy cut and paste of mature code snippets.

An advantage worth highlighting is that ink! macro can handle a variable number of parameters, vs Rust functions with a fixed number of arguments. A classic example is print!. Ink! macros allow a developer Iterating through any number of items to be processed inside the macro depending on predefined patterns.

This is why this exercise comes up with specific macro implementations to illustrate its potentials not just at system level, but also encapsulating business logic in dApps, with multiple benefits:

- ink! macros, similar to “libraries”, can help to improve smart contract readability, reusability and maintainability reducing repetition and inconsistency
- Optimally implemented macros can reduce gas fees and improve code quality
- Lower ink! learning barrier and increase community adoption
- Flexibility to handle variable number of parameters, which functions can’t
- Extensibility to incrementally add more functionalities to the macro
- Separation of smart contract dev layers to divide and conquer complexities
- Abstraction of business logic in dApps to facilitate functional invocation among smart contracts, even cross chains, following web3 design patterns.

Payment_macro

This is a declarative macro to abstract payment functions. The code use to call separate functions to pay restaurants and couriers respectively, both now being consolidated into the same transfer_from_contract_to_account macro defined in helpers.rs:

```
#[macro_export]
macro_rules! transfer_from_contract_to_account {
    ($account:expr, $amount:expr) => {
        if T::env().transfer($account, $amount).is_err() {
            return core::prelude::v1::Err(FoodOrderError::NotTransfered);
        }
    };
}
```

crud_macro

This is a procedure macro to demonstrate how to abstract multiple business functions into one straightforward macro. Unlike the declarative macro, which matches expression with predefined patterns and route subsequent actions accordingly, the procedure macro parses the AST tree from the compiler, rewrites the code by manipulating the AST tree as desired. The code gets auto generated during compilation time. There are many potential uses of macros. This example simply demonstrates how a procedure macro can encapsulate business logic to simplify the smart contract code at application level.

Previously there were separate functions to create, read, update and delete food menu items by a restaurant, now they are all folded under the same ink! macro to invoke any C(reate) R(ead) U(pdate) D(elete) function, a level of abstraction to encapsulate food operations.

This procedure macro is defined in crud-macro/src/lib.rs, for example this update_food header below with specific data quality checks, identifier matches and error handlings, all encapsulated in the macro.

```
#[proc_macro_attribute]
pub fn update_food(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let input = parse_macro_input!(item as ItemFn);
```

Correspondingly, in restaurant_service.rs, it gets invoked without much low level details, increasing code simplicity and readability.

```
#[update_food]
    #[modifiers(is_restaurant_user)]
    fn update_food(
        &mut self,
        food_id: FoodId,
        food_name: String,
        description: String,
        price: Balance,
        eta: u64,
    ) -> FoodResult {
        // Emit `UpdateFoodEvent`
        self.emit_update_food_event(food_id, food_name.clone(),
                                             description.clone(), price, eta);
    }
```

The same macro also includes create_food to insert food items into the restaurant menu, and delete_food to purge deprecated menu items. More functions can be incrementally added as needed. Further, if there's multiple invocations, the developer only needs to change the macro code without needing to change otherwise all scattered function calls, making it less error prone and easier to maintenance.

Testing

This project leverages Chai as the testing framework for end-to-end testing with testing scripts, assets and artifacts in the Github repo. A testing script on a happy path shows the following output:

```
foodorder test

    Main Functionality

      ✔ Platform is ready

      ✔ Restaurant A is added (...ms)

      ✔ Courier A is added (...ms)

      ✔ Customer A is added (...ms)

      ✔ Food A is added (...ms)

      ✔ Order is submitted (...ms)

      ✔ Order is Confirmed (...ms)

      ✔ Food is cooked and Payment is transferred to restaurant (...ms)

      ✔ Order is Delivered (...ms)

      ✔ Delivery is accepted and Payment is sent to courier (...ms)

    10 passing (..s)
```

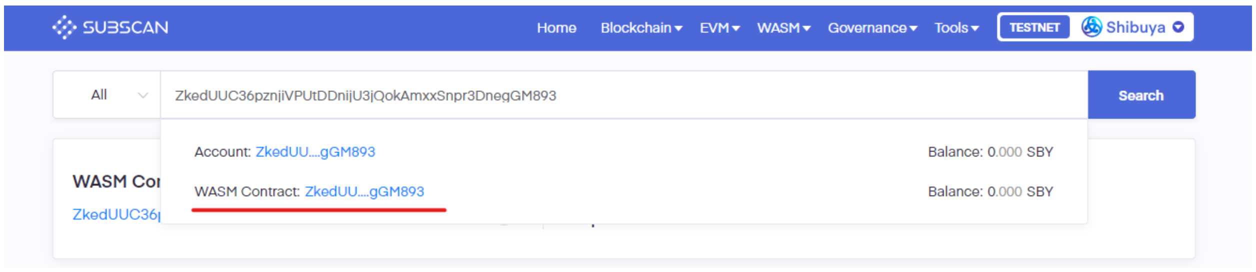
Deployment

Shiden Network is a multi-chain decentralized application layer on Kusama Network. Shibuya is the Shiden's parachain testnet with EVM functionalities. We choose it for deployment as Shiden supports EVM, Wasm, and Layer2 solutions. We prepared a shared document with screenshots to illustrate the flow of deploying the contract then step-by-step flow of ordering food on the blockchain. Here's the expected output when running test from the docker image:

```
✔ Initialising OK
✔ Getting WASM OK
⠇ Connecting to node2023-08-11 11:47:18        API/INIT: shibuya/105: Not decorating runtime apis without matching versions: EthereumRuntimeRPCApi/5 (4 known)
```

```
✔ Connecting to node OK
✔ Deploying OK
✔ Writing config OK
Contract deployed!
Contract address: Yn1dHJTbKuMhA6rLLsRXQtDu4mSFGC6xtvDTueNz1axJ5Dz
```

After successfully deployed, you can check the deployed contract on the shibuya blockexplorer https://shibuya.subscan.io/.



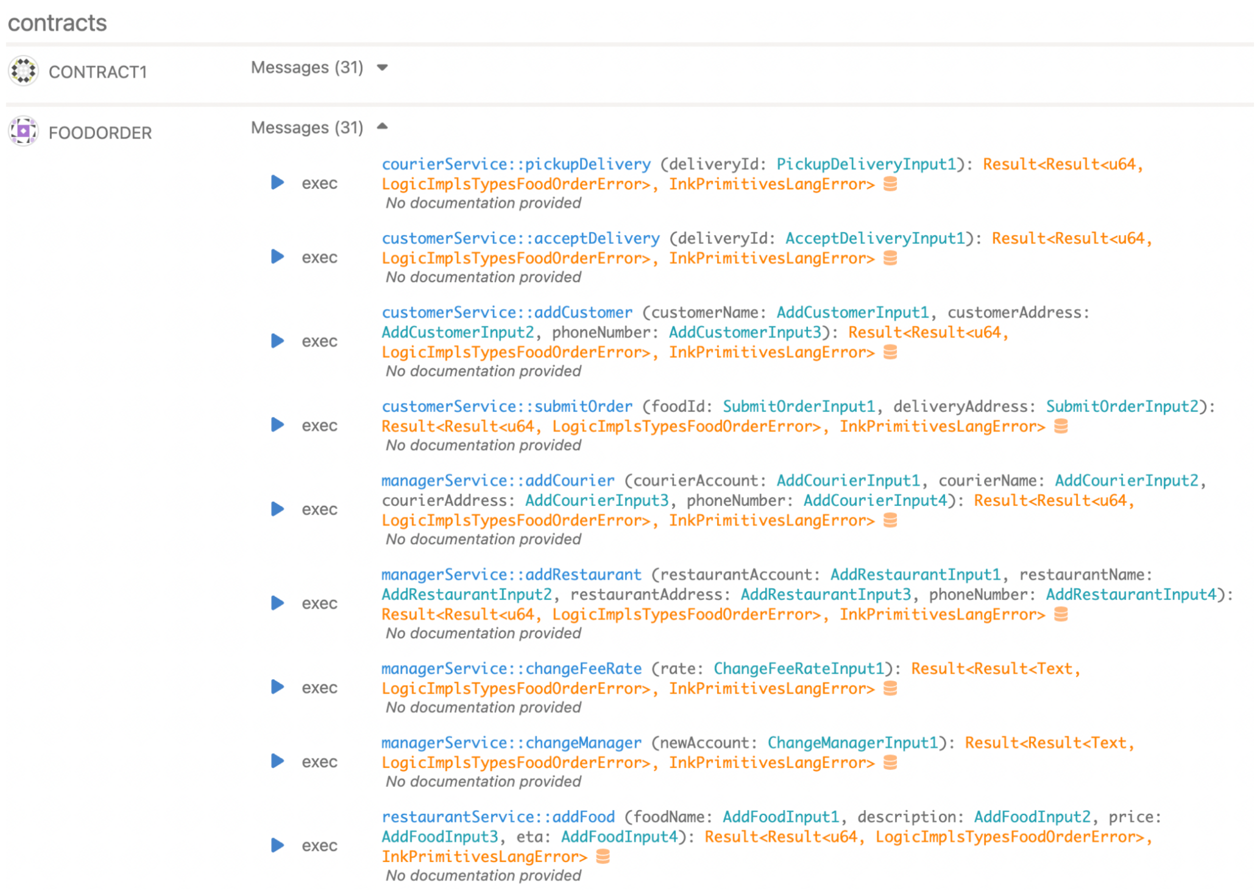
ZkedUUC36pznjiVPUtDDnijU3jQokAmxxSnpr3DnegGM893

Account: ZkedUU....gGM893 Balance: 0.000 SBY

WASM Contract: ZkedUU....gGM893 Balance: 0.000 SBY

ZkedUUC36

Once a contract is deployed, either one you just deployed at a new address, or an existing Wasm contract like this one already deployed, at address Yn1dHJTbKuMhA6rLLsRXQtDu4mSFGC6xtvDTueNz1axJ5Dz. It looks like the following with 31 messages listed under the FOODORDER Wasm contract.


contracts

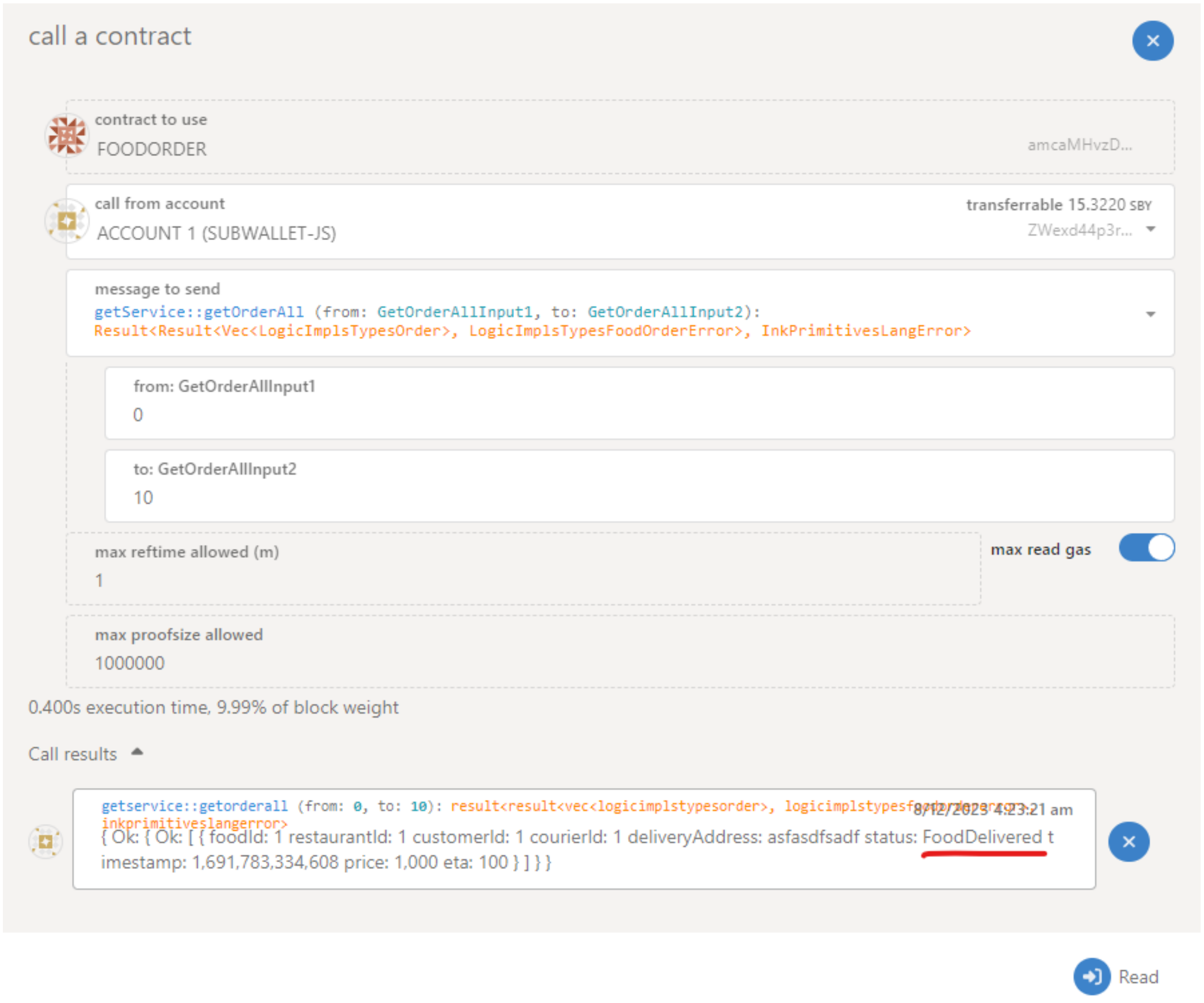
CONTRACT1 Messages (31)

FOODORDER Messages (31)

```
exec  courierService::pickupDelivery (deliveryId: PickupDeliveryInput1): Result<Result<u64,
      LogicImplsTypesFoodOrderError>, InkPrimitivesLangError>
      No documentation provided

exec  customerService::acceptDelivery (deliveryId: AcceptDeliveryInput1): Result<Result<u64,
      LogicImplsTypesFoodOrderError>, InkPrimitivesLangError>
      No documentation provided

exec  customerService::addCustomer (customerName: AddCustomerInput1, customerAddress:
      AddCustomerInput2, phoneNumber: AddCustomerInput3): Result<Result<u64,
      LogicImplsTypesFoodOrderError>, InkPrimitivesLangError>
      No documentation provided

exec  customerService::submitOrder (foodId: SubmitOrderInput1, deliveryAddress: SubmitOrderInput2):
      Result<Result<u64, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError>
      No documentation provided

exec  managerService::addCourier (courierAccount: AddCourierInput1, courierName: AddCourierInput2,
      courierAddress: AddCourierInput3, phoneNumber: AddCourierInput4): Result<Result<u64,
      LogicImplsTypesFoodOrderError>, InkPrimitivesLangError>
      No documentation provided

exec  managerService::addRestaurant (restaurantAccount: AddRestaurantInput1, restaurantName:
      AddRestaurantInput2, restaurantAddress: AddRestaurantInput3, phoneNumber: AddRestaurantInput4):
      Result<Result<u64, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError>
      No documentation provided

exec  managerService::changeFeeRate (rate: ChangeFeeRateInput1): Result<Result<Text,
      LogicImplsTypesFoodOrderError>, InkPrimitivesLangError>
      No documentation provided

exec  managerService::changeManager (newAccount: ChangeManagerInput1): Result<Result<Text,
      LogicImplsTypesFoodOrderError>, InkPrimitivesLangError>
      No documentation provided

exec  restaurantService::addFood (foodName: AddFoodInput1, description: AddFoodInput2, price:
      AddFoodInput3, eta: AddFoodInput4): Result<Result<u64, LogicImplsTypesFoodOrderError>,
      InkPrimitivesLangError>
      No documentation provided
```

One can start interacting with it by manually reading or executing messages with parameters and different accounts representing different roles, restaurants, couriers, consumers, etc.. Recommend to install the Sub Wallet as browser extension, and get some free SBY tokens (since we deployed on the Shibuya testnet which uses SBY tokens) from a faucet.

Now one can run this example's order food on blockchain workflow by invoking any one of the 31 messages playing different roles. Notice that reading messages does not require gas fees but executing messages does. We prepared a shared document with screenshots to illustrate the flow of deploying the contract then step-by-step flow of ordering food on the blockchain. Towards the end of the workflow, the order's status advanced to "FoodDelivered" as highlighted in the screenshot below.



```
getService::getOrderAll (from: GetOrderAllInput1, to: GetOrderAllInput2):
Result<Result<Vec<LogicImplsTypesOrder>, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError>
```

from: GetOrderAllInput1: 0

to: GetOrderAllInput2: 10

max reftime allowed (m): 1

max read gas

max proofsize allowed: 1000000

0.400s execution time, 9.99% of block weight

```
getservice::getorderall (from: 0, to: 10): result<result<vec<logicimplstypesorder>, logicimplstypesfoodordererror>,
inkprimitiveslangerror>
8/12/2023 4:23:21 am
{ Ok: { Ok: [ { foodId: 1 restaurantId: 1 customerId: 1 courierId: 1 deliveryAddress: asfasdfsadf status: FoodDelivered t
imestamp: 1,691,783,334,608 price: 1,000 eta: 100 } ] } }
```

Recommendations and Next Steps

It is a great learning experience to come up with this Order Food on Blockchain example, implement it via ink! smart contracts, with testing scripts and deployment, for the purpose of learning ink! language and related tools. The project aims to focus on three specific areas: ink! macros, Wasm smart contract upgradability, and binary footprint / gas costs / performance benchmarks, all using the same contrived Order Food on Blockchain use case.

ink! learning resources

A good starting point could be Guided Tutorial for Beginners, and What is Parity's ink!? Post by Michi Müller to get a grasp of high level understanding of ink!. Advanced Rust developers can check out this github markdown to learn ink! Architecture and internals. For ink! 4.0 specifically, see this announcement for what's in the release.

Parity Technologies has ink! docs in this Github repo including ink! basics and some examples. Develop a smart contract tutorial can be found on Substrate's docs site covering specific topics on how to use a smart contract template, store simple values using a smart contract, increment and retrieve stored values using a smart contract, add public and private functions to a smart contract.

A curated list of ink! projects can be found here in Github including production and testnets, dApps, libraries and standards, deFi, gaming and block explorers. For tools, OpenBrush documented its ink! smart contract libraries here including ownable and access control. Contracts UI links to a frontend for contract deployment and interaction. Stack Exchange for Substrate and Polkadot can be found here with 400+ questions tagged ink.

While ink!'s official document is a great source to learn ink! as a smart contract language however it seems lacking sufficient coverage of macros within the context of ink!, even though it started as a bunch of macros itself then expanded into a Rust based DSL. A useful document about Rust macros can be found in the Rust Programming Language where one can read what it is, the difference with functions and various types of macros. An excellent source specifically on Rust macros is the Little Book of Rust Macros, still to be fully completed, yet has in-depth coverage on the subject. However none of these talk about macros in the context of ink! smart contract language.

Recommendations

Compared to early smart contract languages such as Solidity, ink! Is a new kid on the block. It provides an excellent alternative option to write smart contracts for those who are familiar with Rust programming language. As more and more web2 developers come towards web3 paradigm, ink! has made great progress to bridge the gap with Rust programming skill sets and tools for easier onramp.

Parity has released ink! 4.0 and is actively developing the latest. We shared many great features of ink! showcased in this example. To provide a balanced view, we also share here improvement areas that both the ink! project and the ecosystem can collaborate and contribute. Some of our own developer experiences from this exercise:

- Learning - There's a steep learning curve when it comes to ink and macros for developers not familiar with Rust. One also needs to learn Substrate framework with relevant pallets and Polkadot to get the full context. There is quite some ink! learning materials but scattered around. Ideally a well organized learning center with more in-depth coverage on sophisticated examples and resources should serve the community well. It takes more than just tipping the toe in the water to have full immersion developer experiences.

- Tooling - while there are some tools available for ink!, there is room for improvement in this area. For example, tools to manage upgrades, interoperability, gas estimator, performance benchmarks, industry tailored smart contract macros and templates, to make it easier on ramping for developer adoption.

- Ecosystem - Solidity has a more mature ecosystem than ink!, with more libraries, frameworks, tools and even job postings available. Building a robust ecosystem around ink! would make it more attractive to developers as a viable option building dApps.

- Enhancements - There are an increasing number of smart contract languages such as Move with security, auditability and resource oriented design. There are also rollups such as Arbitrum Nitro stack developing Stylus that directly supports conventional languages such as C, C++, Rust, etc.compiled into Wasm code, running side-by-side with Solidity running on EVM sharing the same states. Ink! can get inspiration from the latest innovations beyond just the Substrate framework.

- Adoptions - Our perception is that ink! is well received by a niche Rust developer cohort implementing smart contracts mostly on Polkadot / Kusama. To expand the community, ink needs to do more heavy lifting reaching out to target segments beyond this small pod, and win incremental adoptions to grow the community. To that end, we can share some of our own experiences we believe that could potentially hamper adoption:

    - Community - The open source project is narrowly concentrated from Parity Technology and a handful of Dotsama parachain developer shops.

    - Communication - Issues such as Polkadot API is failing to connect to Substrate Contracts Node docker container without a clear documented solution and responses is not an effective communication. Developers stumbled on it could walk away less confident on the level of quality and support needed for adoption.

    - Support - Bugs especially those affecting important features should be written up and clearly communicated with the community on the issue, workaround solution if any, and expectation of proposed fixes and timeline. For example this issue and related issue affect smart contract upgradability. Lack of clarity on the solution and expectation of technical issues potentially discourages adoptions.

    - Branding - It seems there’s not enough publicity about ink! and related tech stack outside of the Dotsama (Polkadot & Kusama) community. The squid themed branding is really cute but one thing worth mentioning is that when searching for related information about the language, tools, jobs, discussion threads, the unique “ink!” name often messed up with other search results such as printer ink, inc, etc. The branding does not really stand out, as limited information is buried deep.

    - Developer experiences - Compared to say Solidity, the ink! community is much smaller with much less contents, discussion threads, tutorials, code repos, tools, etc. However it is growing quickly hopefully to catch up and bring dev experiences on par with more mature ones.

Despite the kinks on this ink! learning journey, we feel ink! has strong technical merits and comparative advantages for smart contract development, nutrients from the community to grow the ecosystem, and ingredients to fully cook a delicious meal, and ready for order in web3!

Next steps

The project is fully open sourced under Apache 2.0 license to share with both web2 and web3 developer communities. This is the first of a three parts post covering the use case, ink! implementation with macros, with upcoming part 2 on upgradability, and part3 to benchmark the performance, quantify binary footprints and optimize gas costs in various ink! optimization variations.

Stay tuned, community comments, suggestions and contributions welcome!