Quickstart: write a smart contract in Rust using Stylus

This guide will get you started withStylus ' basics. We'll cover the following steps:

1. Setting up your development environment
2. Creating a Stylus project with cargo stylus
3. Checking the validity of your contract
4. Deploying your contract
5. Exporting your contract's ABIs
6. Calling your contract
7. Sending a transaction to your contract

Setting up your development environment

Prerequisites

Rust toolchain Follow the instructions onRust Lang's installation page to install a complete Rust toolchain (v1.81 or newer) on your system. After installation, ensure you can access the programsrustup ,rustc , andcargo from your preferred terminal application. VS Code We recommendVSCode as the IDE of choice for its excellent Rust support, but feel free to use another text editor or IDE if you're comfortable with those.

Some helpful VS Code extensions for Rust development:

- rust-analyzer
- : Provides advanced features like smart code completion and on-the-fly error checks
- Error Lens
- : Immediately highlights errors and warnings in your code
- Even Better TOML
- : Improves syntax highlighting and other features for TOML files, often used in Rust projects
- Dependi
- : Helps manage Rust crate versions directly from the editor Docker The testnode we will use as well as somecargo stylus commands require Docker to operate.

You can download Docker fromDocker's website . Foundry's Cast Foundry's Cast is a command-line tool that allows you to interact with your EVM contracts. You need toinstall the Foundry CLI to use Cast. Nitro devnode Stylus is available on Arbitrum Sepolia, but we'll use nitro devnode which has a pre-funded wallet saving us the effort of wallet provisioning or running out of tokens to send transactions.

Install your devnode git clone https://github.com/OffchainLabs/nitro-devnode.git cd nitro-devnode Launch your devnode ./run-dev-node.sh

Creating a Stylus project with cargo stylus

cargo stylus is a CLI toolkit built to facilitate the development of Stylus contracts.

It is available as a plugin to the standard cargo tool used for developing Rust programs.

Installing cargo stylus

In your terminal, run:

cargo

install

--force cargo-stylus Add WASM (WebAssembly) as a build target for the specific Rust toolchain you are using. The below example sets your default Rust toolchain to 1.80 as well as adding the WASM build target:

rustup default 1.80 rustup target add wasm32-unknown-unknown --toolchain

1.80 You can verify that cargo stylus is installed by runningcargo stylus --help in your terminal, which will return a list of helpful commands, we will use some of them in this guide:

cargo stylus --help returns: Cargo command

for developing Stylus projects

Usage: cargo stylus < COMMAND

Commands: new Create a new Stylus project init Initializes a Stylus project in the current directory export-abi Export a Solidity ABI activate Activate an already deployed contract [aliases: a] cache Cache a contract using the Stylus CacheManager for Arbitrum chains check Check a contract [aliases: c] deploy Deploy a contract [aliases: d] verify Verify the deployment of a Stylus contract [aliases: v] cgen Generate c code bindings for a Stylus contract replay Replay a transaction in gdb [aliases: r] trace Trace a transaction [aliases: t] help Print this message or the help of the given command (s)

Options: -h, --help Print help -V, --version Print version

Creating a project

Let's create our first Stylus project by running:

cargo stylus new < YOUR_PROJECT_NAME

> cargo stylus new generates a starter template that implements a Rust version of theSolidityCounter smart contract example .

At this point, you can move on to the next step of this guide or develop your first Rust smart contract. Feel free to use theStylus Rust SDK reference section as a starting point; it offers many examples to help you quickly familiarize yourself with Stylus.

Checking if your Stylus project is valid

By runningcargo stylus check against your first contract, you can check if your program can be successfullydeployed and activated onchain.

Important: Ensure your Docker service runs so this command works correctly.

cargo stylus check cargo stylus check executes a dry run on your project by compiling your contract to WASM and verifying if it can be deployed and activated onchain.

If the command above fails, you'll see detailed information about why your contract would be rejected:

Reading WASM file at bad-export.wat Compressed WASM size: 55 B Stylus checks failed: program pre-deployment check failed when checking against ARB_WASM_ADDRESS 0x0000…0071: (code: -32000, message: program activation failed: failed to parse program)

Caused by: binary exports reserved symbol stylus_ink_left

Location: prover/src/binary.rs:493:9, data: None The contract can fail the check for various reasons (on compile, deployment, etc...). Reading theInvalid Stylus WASM Contracts explainer can help you understand what makes a WASM contract valid or not.

If your contract succeeds, you'll see something like this:

Finished release [optimized] target (s)

in

1 .88s Reading WASM file at hello-stylus/target/wasm32-unknown-unknown/release/hello-stylus.wasm Compressed WASM size: 3 KB Program succeeded Stylus onchain activation checks with Stylus version: 1 Note that runningcargo stylus check may take a few minutes, especially if you're verifying a contract for the first time .

Seecargo stylus check --help for more options.

Deploying your contract

Once you're ready to deploy your contract onchain,cargo stylus deploy will help you with the deployment and its gas estimation.

Estimating gas

Note: For every transaction, we'll use the testnode pre-funded wallet, you can use0xb6b15c8cb491557369f3c7d2c287b053eb229daa9c22138887752191c9520659 as your private key.

You can estimate the gas required to deploy your contract by running:

cargo stylus deploy \ --endpoint = 'http://localhost:8547'

\ --private-key = "0xb6b15c8cb491557369f3c7d2c287b053eb229daa9c22138887752191c9520659"

\ --estimate-gas The command should return something like this:

deployment tx gas: 7123737 gas price: "0.100000000" gwei deployment tx total cost: "0.000712373700000000" ETH

Deployment

Let's move on to the contract's actual deployment. Two transactions will be sent onchain: the contract deployment and its activation.

cargo stylus deploy \ --endpoint = 'http://localhost:8547'

\ --private-key = "0xb6b15c8cb491557369f3c7d2c287b053eb229daa9c22138887752191c9520659" Once the deployment and activations are successful, you'll see an output similar to this:

deployed code at address: 0x33f54de59419570a9442e788f5dd5cf635b3c7ac deployment tx hash: 0xa55efc05c45efc63647dff5cc37ad328a47ba5555009d92ad4e297bf4864de36 wasm already activated ! Make sure to save the contract's deployment address for future interactions!

More options are available for sending and outputting your transaction data. Seecargo stylus deploy --help for more details.

Exporting the Solidity ABI interface

The cargo stylus tool makes it easy to export your contract's ABI usingcargo stylus export-abi .

This command returns the Solidity ABI interface of your smart contract. If you have been runningcargo stylus new without modifying the output,cargo stylus export-abi will return:

/* *\ This file was automatically generated by Stylus and represents a Rust program. * For more information, please see [The Stylus SDK] (https://github.com/OffchainLabs/stylus-sdk-rs)*/ .

// SPDX-License-Identifier: MIT-OR-APACHE-2.0 pragma solidity ^0.8.23 ;

interface ICounter { function number ( ) external view returns (uint256) ;

function setNumber (uint256 new_number) external ;

function mulNumber (uint256 new_number) external ;

function addNumber (uint256 new_number) external ;

function increment ( ) external ; } Ensure you save the console output to a file that you'll be able to use with your[dApp](https://docs.arbitrum.io/welcome/get-started) .

Interacting with your Stylus contract

Stylus contracts are EVM-compatible, you can interact with them with your tool of choice, such asHardhat ,Foundry's Cast , or any other Ethereum-compatible tool.

In this example, we'll use Foundry's Cast to send a call and then a transaction to our contract.

Calling your contract

Our contract is a counter; in its initial state, it should store a counter value of0 . You can call your contract so it returns its current counter value by sending it the following command:

Call to the function: number()(uint256) cast call --rpc-url 'http://localhost:8547' --private-key 0xb6b15c8cb491557369f3c7d2c287b053eb229daa9c22138887752191c9520659 \ [deployed-contract-address]

"number()(uint256)" Let's break down the command:

- cast call
- command sends a call to your contract
- The--rpc-url
- option is theRPC URL
- endpoint of our testnode:http://localhost:8547
- The--private-key
- option is the private key of our pre-funded development account. It corresponds to the address0x3f1eae7d46d88f08fc2f8ed27fcb2ab183eb2d0e
- The [deployed-contract-address] is the address we want to interact with, it's the address that was returned bycargo stylus deploy
- number()(uint256)
- is the function we want to call in Solidity-style signature. The function returns the counter's current value

Calling 'number()(uint256)' returns: 0 Thenumber()(uint256) function returns a value of0 , the contract's initial state.

Sending a transaction to your contract

Let's increment the counter by sending a transaction to your contract'sincrement() function. We'll use Cast'ssend command to send our transaction.

Sending a transaction to the function: increment() cast send --rpc-url 'http://localhost:8547' --private-key 0xb6b15c8cb491557369f3c7d2c287b053eb229daa9c22138887752191c9520659 \ [deployed-contract-address]

"increment()" Transaction returns: blockHash 0xfaa2cce3b9995f3f2e2a2f192dc50829784da9ca4b7a1ad21665a25b3b161f7c blockNumber 20 contractAddress cumulativeGasUsed 97334 effectiveGasPrice 100000000 from 0x3f1Eae7D46d88F08fc2F8ed27FCb2AB183EB2d0E gasUsed 97334 logs [ ] logsBloom 0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
root status 1

(success) transactionHash 0x28c6ba8a0b9915ed3acc449cf6c645ecc406a4b19278ec1eb67f5a7091d18f6b transactionIndex 1 type

2 blobGasPrice blobGasUsed authorizationList to 0x11B57FE348584f042E436c6Bf7c3c3deF171de49 gasUsedForL1 "0x0" l1BlockNumber "0x1223" Our transactions returned a status of1 , indicating success, and the counter has been incremented (you can verify this by calling your contract'snumber()(uint256) function again).

Conclusion

Congratulations! You've successfully initialized, deployed, and interacted with your first contract using Stylus and Rust.

Feel free to explore theStylus Rust SDK reference for more information on using Stylus in your Arbitrum projects.Edit this page Last updatedonJan 28, 2025 Previous A gentle introduction Next Overview