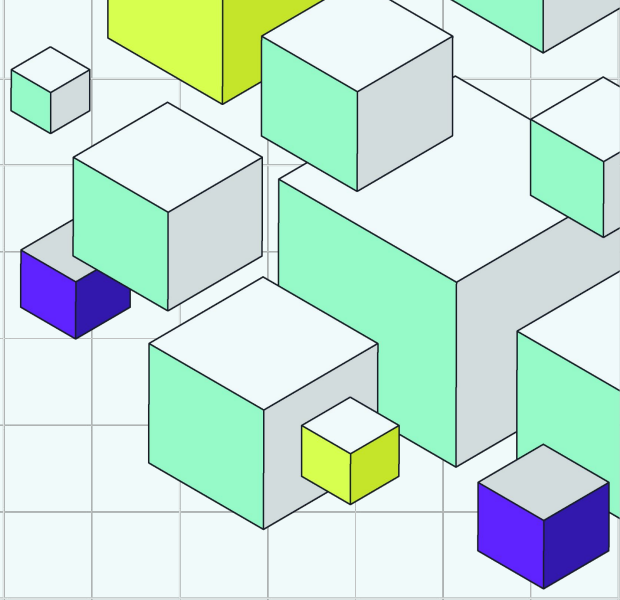
LimeChain



Introduction to web3 development

Solana Workflow and Developer Tooling Overview

19.01.25

Ognyan Chikov

Head of Engineering

E: ognyan@limechain.tech

LinkedIn: https://www.linkedin.com/in/ognyan-chikov/

@OGI_CHI

Let's connect on Telegram

We Design, Build, and Improve Web3 Solutions and Infrastructure

Technical Excellence

We are a leading blockchain R&D and software development company with over 150 experts committed to delivering technical excellence and robust solutions that help set industry standards and contribute to the development and maturation of the blockchain infrastructure.

Universal Blockchain Support

Our experience and wide range of technical expertise allows us a flexible approach not limited to any one blockchain technology. This enables us to select the most suitable solution for your specific needs.

Multidisciplinary Approach and Expertise

We've created, cultivated, and tested the right conditions and expertise that enable us to approach blockchain technology from a strategic multidisciplinary perspective. This allows us to provide you with robust and tailored solutions that address the unique challenges and requirements of your project while fostering collaboration between our teams.

Agenda

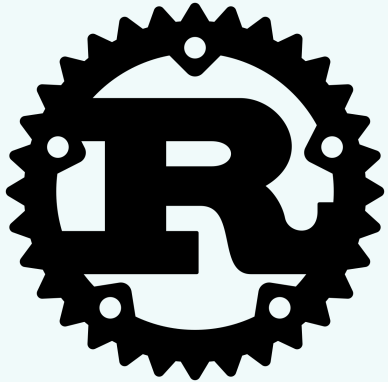
The agenda:

- Overview of Solana dApp Development
- Developer Tools and Their Use Cases
- Hands-On Demos
- What is zest?

Overview of Solana dApp Development

- Where to start from
- What is the workflow of Solana dApp development
- Useful websites and tools

Where to start from...



- https://report.helius.dev/
- https://solana.com/docs
- https://solana.stackexchange.com/

What is the workflow of Solana dApp development

Install prerequisites
Rust, Solana CLI, Anchor

Create project
Manually or using npx

Define your dApp architecture and use case
Outline your dApp`s purpose and features

Design your solana programs (contracts)
Using rust and anchor framework

Tests
Test your Solana programs

Audits
Perform audits

Starts with the FE (dApp)
Prepare the FE framework

Integrate with wallet adaptor
Solana wallet adaptor + Solana web3.js

Define appropriate 3rd party libraries
What my project needs as features

Deployments
Solana blockchain deployments
FE deployments

Useful websites and tools

- https://explorer.solana.com/
- https://faucet.solana.com/
- https://solana.com/solana-wallets
- https://beta.solpg.io/
- https://www.solconverter.com/
- https://solana-labs.github.io/solana-web3.js/

Developer Tools and Their Use Cases

zest

code coverage CLI tool for Solana programs

Entry Point

```rust
// src/main.rs
fn main() -> eyre::Result<()> {
    let Config { command } = Config::parse();
    match command.unwrap_or(Subcommands::Coverage(Default::default())) {
        Subcommands::Coverage(config) => {
            let config = coverage::Config::parse_with_config_file(Some(config))?;
            coverage::run(config)
        }
    }
}
```

What happens here:

- Command-line arguments are parsed
- If no command is specified, defaults to Coverage
- Configuration is loaded from either: Command line arguments or Configuration file (zest-coverage.toml)
- Default values

Project Analysis

```rust
// Native Program (examples/counter/native/src/lib.rs)
pub fn process_instruction(program_id: &Pubkey, accounts: &[AccountInfo], ...) -> R
    // Native implementation
}

// Anchor Program (examples/counter/anchor/programs/counter_anchor/src/lib.rs)
#[program]
pub mod counter_anchor {
    pub fn increment(ctx: Context<CounterContext>) -> Result<()> {
        // Anchor implementation
    }
}
```

What happens here:

- Project type detection
- Native Solana Program: Looks for process_instruction entry point -
- Anchor Program: Looks for #[program] attribute -
- CPI Program: Detects cross-program invocations

Key points:

- Different program types require different analysis approaches
- Test structure varies between Native and Anchor programs
- CPI programs need special handling for cross-program coverage

Coverage Setup

```rust
// src/coverage/mod.rs
pub fn run(config: Config) -> eyre::Result<()> {
    // Environment setup
    env::set_current_dir(&path)?;
    install_llvm_tools(compiler_version.as_ref())?;
```

What happens here:

- Changes to project directory
- Installs necessary LLVM tools for coverage
- Sets up coverage environment variables
- Creates necessary directories

Key points:

- Sets up build environment based on program type
- Configures coverage instrumentation
- Prepares test environment
- Different build process for Native vs Anchor
- Coverage instrumentation is added during build
- Test framework is configured based on project type

How Instrumentation Works

```rust
// src/coverage/mod.rs
pub fn run(config: Config) -> eyre::Result<()> {
    // Set up LLVM profiling environment
    let mut env_vars: HashMap<&str, String> = HashMap::new();
    env_vars.insert(
        "LLVM_PROFILE_FILE",
        format!("{}/zest-%p-%m.profraw", coverage_dir)
    );

    // Configure coverage instrumentation
    match coverage_strategy {
        CoverageStrategy::InstrumentCoverage => {
            // Add instrumentation flags
            env_vars.insert("RUSTFLAGS", "-C instrument-coverage");
            if branch {
                // Add branch coverage if enabled
                env_vars.push_str(" -Z coverage-options=mcdc");
            }
        }
    }
```

What happens here:

- LLVM profiling is configured to generate .profraw files
- Each program function is instrumented with coverage tracking
- Branch points are marked for tracking (if enabled)

Key points:

- Sets up build environment based on program type
- Configures coverage instrumentation
- Prepares test environment
- Different build process for Native vs Anchor
- Coverage instrumentation is added during build
- Test framework is configured based on project type

Instrumentation Example

```rust
// examples/counter/anchor/programs/counter_anchor/src/lib.rs
#[program]
pub mod counter_anchor {
    pub fn increment(ctx: Context<CounterContext>) -> Result<()> {
        ctx.accounts.counter_pda.count += 1;
        Ok(())
    }
}
```

What Gets Instrumented:

- Function Entry Points:
  - Tracks when `increment` is called
  - Records number of invocations
- Line Coverage:
  - Tracks execution of counter_pda.count += 1
  - Records how many times each line runs
- Branch Points (if enabled):
  - Tracks error conditions
  - Records which paths are taken

Test Execution & Coverage Collection

```rust
// src/coverage/mod.rs
if tests.is_empty() {
    cargo_test(None)?;
} else {
    tests.iter().map(Some).try_for_each(cargo_test)?;
}
```

What happens:

- Tests are identified and filtered
- Each test is run with coverage instrumentation
- Raw coverage data is stored in `profraw` files

Coverage Data Collection

- Raw coverage data is stored in `profraw` files
- LLVM generates coverage data during test execution
- Data includes:
  - Line execution counts
  - Branch decisions (if enabled)
  - Function entry/exit points

Coverage Processing

```rust
// src/coverage/mod.rs
if tests.is_empty() {
    cargo_test(None)?;
} else {
    tests.iter().map(Some).try_for_each(cargo_test)?;
}
```

What happens:

- Raw coverage data is collected from `profraw` files
- Data is processed and analyzed
- Coverage metrics are calculated
- Source mapping is applied

Report generation

- Generates reports in specified formats (HTML, LCOV)
- Maps coverage data back to source code
- Creates visual representation of coverage

Solana mucho

Command line tool to simplify development and testing

Book a 1-on-1 consultation with the LimeChain team

Q&A