

Unit Testing

Unit tests allow you to test the contract methods individually. They are suitable to check the storage is updated correctly, and that methods return their expected values. They are written in the contract's language and execute locally.

If you used one of our [examples](#) as template, then you simply need to navigate to the contract's folder, and use `yarn test`. In case you didn't, then we recommend you copy the necessary node files (e.g. `package.json`) from one of our templates.

tip You can run `yarn test` from the root folder of each project to run both unit and [integration](#) tests.

Snippet I: Testing a Counter

The tests in the [Counter Example](#) rely on basic functions to check that the `increment`, `decrement`, and `reset` methods work properly.

- Rust

contract-rs/src/lib.rs loading ... [See full example on GitHub](#)

Snippet II: Modifying the Context

While doing unit testing you can modify the [Environment variables](#) through the `VMContextBuilder`. This will enable you to, for example, simulate calls from different users, with specific attached deposit and GAS. Here we present a snippet on how we test the `donate` method from our [Donation Example](#) by manipulating the `predecessor` and `attached_deposit`.

- Rust

contract-rs/src/lib.rs loading ... [See full example on GitHub](#)

⚠ Limitations

Unit tests are useful to check for code integrity, and detect basic errors on isolated methods. However, since unit tests do not run on a blockchain, there are many things which they cannot detect. Unit tests are not suitable for:

- Testing [gas](#)
- and [storage](#)
- usage
- Testing [transfers](#)
- Testing [cross-contract calls](#)
- Testing complex interactions, i.e. multiple users depositing money on the contract

For all these cases it is necessary to complement unit tests with [integration tests](#). [Edit this page](#) Last updated on Mar 25, 2024 by [matiasbenary](#) Was this page helpful? Yes No

[Previous Introduction](#) [Next Integration Tests](#)