The Starknet Foundry

Starknet Foundry is a toolchain for developing Starknet smart contracts. It helps with writing, deploying, and testing your smart contracts. It is inspired by Foundry.

Installation

Starknet Foundry is easy to install on Linux, macOS and Windows. In this section, we will walk through the process of installing Starknet Foundry.

Contents

Requirements

📝 **Note**

Ensure all requirements are installed and follow the required minimum versions. Starknet Foundry will not run if not following these requirements.

To use Starknet Foundry, you need:

- Scarb version >= 2.7.0
- Universal-Sierra-Compiler
- *(Optional for Scarb >= 2.10.0)*[1] Rust version >= 1.80.1

all installed and added to your `PATH` environment variable.

[1] Additionally, your platform must be one of the supported:

- aarch64-apple-darwin
- aarch64-unknown-linux-gnu
- x86_64-apple-darwin
- x86_64-pc-windows-msvc
- x86_64-unknown-linux-gnu

📝 **Note** `Universal-Sierra-Compiler` will be automatically installed if you use `asdf` or `snfoundryup`. You can also create `UNIVERSAL_SIERRA_COMPILER` env var to make it visible for `snforge`.

Linux and macOS

Info

If you already have installed Rust, Scarb and asdf simply run `asdf plugin add starknet-foundry`

Install asdf

Follow the instructions from asdf docs.

To verify that asdf was installed, run

```
asdf --version
```

Install Scarb version >= 2.7.0

First, add Scarb plugin to asdf

```
asdf plugin add scarb
```

Install Scarb

```
asdf install scarb latest
```

Set a version globally (in your ~/.tool-versions file):

```
asdf global scarb latest
```

To verify that Scarb was installed, run

```
scarb --version
```

and verify that version is >= 2.7.0

(Optional for Scarb >= 2.10.0)1 Rust Installation

Info

Rust installation is only required if **ANY** of the following is true:

- You are using Scarb version <= 2.10.0
- Your platform is not one of the following supported platforms:
  - aarch64-apple-darwin
  - aarch64-unknown-linux-gnu
  - x86_64-apple-darwin
  - x86_64-pc-windows-msvc
  - x86_64-unknown-linux-gnu

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

To verify that correct Rust version was installed, run

```
rustc --version
```

and verify that version is >= 1.80.1

See Rust docs for more details.

Install Starknet Foundry

First, add Starknet Foundry plugin to asdf

```
asdf plugin add starknet-foundry
```

Install Starknet Foundry

```
asdf install starknet-foundry latest
```

Set a version globally (in your ~/.tool-versions file):

```
asdf global starknet-foundry latest
```

To verify that Starknet Foundry was installed, run

```
snforge --version
```

or

```
sncast --version
```

Windows

> **Info** - WSL (Windows Subsystem for Linux)
>
> Starknet Foundry can be installed natively on Windows, but currently, for smoother experience, it is recommended to use WSL.
>
> If you are using WSL, please follow the Linux and macOS guide.

Install Scarb version >= 2.7.0

Follow the instructions from Scarb docs.

1. Download the release archive matching your CPU architecture from https://docs.swmansion.com/scarb/download.html#precompiled-packages.
2. Extract it to a location where you would like to have Scarb installed. We recommend `%LOCALAPPDATA%\Programs\scarb`.
3. From this directory, get the full path to `scarb\bin` and add it to PATH. See this article for instructions on Windows 10 and 11.

To verify that Scarb was installed, run

```
scarb --version
```

and verify that version is >= 2.7.0

(Optional for Scarb >= 2.10.0)1 Rust Installation

Info

Rust installation is only required if:

You are using Scarb version <= 2.10.0, *OR*

- Your platform is not one of the following supported platforms:
  - x86_64-pc-windows-msvc

Go to https://www.rust-lang.org/tools/install and follow the installation instructions.

To verify that correct Rust version was installed, run

```
rustc --version
```

and verify that version is >= 1.80.1

See Rust docs for more details.

Install Universal Sierra Compiler

1. Download the release archive matching your CPU architecture from https://github.com/software-mansion/universal-sierra-compiler/releases/latest. Look for package with `windows` in the name e.g. `universal-sierra-compiler-v2.3.0-x86_64-pc-windows-msvc.zip`.
2. Extract it to a location where you would like to have Starknet Foundry installed. We recommend `%LOCALAPPDATA%\Programs\universal-sierra-compiler`.
3. From this directory, get the full path to `universal-sierra-compiler\bin` and add it to PATH. See this article for instructions on Windows 10 and 11.

To verify that Starknet Foundry was installed, run

```
universal-sierra-compiler --version
```

Install Starknet Foundry

1. Download the release archive matching your CPU architecture from https://github.com/foundry-rs/starknet-foundry/releases/latest. Look for package with `windows` in the name e.g. `starknet-foundry-v0.34.0-x86_64-pc-windows-msvc.zip`.
2. Extract it to a location where you would like to have Starknet Foundry installed. We recommend `%LOCALAPPDATA%\Programs\snfoundry`.
3. From this directory, get the full path to `snfoundry\bin` and add it to PATH. See this article for instructions on Windows 10 and 11.

To verify that Starknet Foundry was installed, run

```
snforge --version
```

or

```
sncast --version
```

Common Errors

No Version Set (Linux and macOS Only)

Users may encounter this error when trying to use `snforge` or `sncast` without setting a version:

```
No version is set for command snforge
Consider adding one of the following versions in your config file at
$HOME/.tool_versions
starknet-foundry 0.32.0
```

This error indicates that `Starknet Foundry` version is unset. To resolve it, set the version globally using asdf:

```
asdf global starknet-foundry <version>
```

For additional information on asdf version management, see the asdf

Invalid Rust Version

When running any `snforge` command, error similar to this is displayed

```
Compiling snforge_scarb_plugin v0.34.0
error: package snforge_scarb_plugin v0.34.0 cannot be built because it requires
rustc 1.80.1 or newer, while the currently active rustc version is 1.76.0
```

This indicates incorrect Rust version is installed or set.

Verify if rust version >= 1.80.1 is installed

```
rustc --version
1.80.1
```

To fix, follow the platform specific instructions:

Linux and macOS

If the version is incorrect or the error persists, try changing the global version of Rust

```
rustup default stable
```

and local version of Rust

```
rustup override set stable
```

Windows

Follow Rust installation and ensure correct version of rust was added to PATH.

`scarb test` Isn't Running `snforge`

By default, `scarb test` doesn't use `snforge` to run tests, and it needs to be configured. Make sure to include this section in `Scarb.toml`

```
[scripts]
test = "snforge test"
```

Universal-Sierra-Compiler update

If you would like to bump the USC manually (e.g. when the new Sierra version is released) you can do it by running:

Linux and macOS

```
curl -L https://raw.githubusercontent.com/software-mansion/universal-sierra-compiler/master/scripts/install.sh | sh
```

Windows

Follow Universal Sierra Compiler installation for Windows.

How to build Starknet Foundry from source code

If you are unable to install Starknet Foundry using the instructions above, you can try building it from the source code as follows:

1. Set up a development environment.
2. Run `cd starknet-foundry && cargo build --release`. This will create a `target` directory.
3. Move the `target` directory to the desired location (e.g. `~/.starknet-foundry`).
4. Add `DESIRED_LOCATION/target/release/` to your `PATH`.

First Steps With Starknet Foundry

In this section we provide an overview of Starknet Foundry `snforge` command line tool. We demonstrate how to create a new project, compile, and test it.

To start a new project with Starknet Foundry, run `snforge new`

```
$ snforge new hello_starknet
```

Let's check out the project structure

```
$ cd hello_starknet
$ tree . -L 1
```

▶ Output:

- `src/` contains source code of all your contracts.
- `tests/` contains tests.
- `Scarb.toml` contains configuration of the project as well as of `snforge`
- `Scarb.lock` a locking mechanism to achieve reproducible dependencies when installing the project locally

And run tests with `snforge test`

```
$ snforge test
```

▶ Output:

Using `snforge` With Existing Scarb Projects

To use `snforge` with existing Scarb projects, make sure you have declared the `snforge_std` package as your project development dependency.

Add the following line under `[dev-dependencies]` section in the `Scarb.toml` file.

```
# ...

[dev-dependencies]
snforge_std = "0.33.0"
```

Make sure that the above version matches the installed `snforge` version. You can check the currently installed version with

```
$ snforge --version
```

▶ Output:

It is also possible to add this dependency using `scarb add` command.

```
$ scarb add snforge_std@0.33.0 --dev
```

Additionally, ensure that starknet-contract target is enabled in the `Scarb.toml` file.

```
# ...
[[target.starknet-contract]]
```

Note

You can additionally specify `scarb` settings to avoid compiling Cairo plugin which `snforge_std` depends on. The plugin is written in Rust and, by default, is compiled locally on the user's side.

```
[tool.scarb]
allow-prebuilt-plugins = ["snforge_std"]
```

This configuration requires Scarb version >= 2.10.0 .

Scarb

Scarb is the package manager and build toolchain for Starknet ecosystem. Those coming from Rust ecosystem will find Scarb very similar to Cargo.

Starknet Foundry uses Scarb to:

- manage dependencies
- build contracts

One of the core concepts of Scarb is its manifest file - `Scarb.toml`. It can be also used to provide configuration for Starknet Foundry Forge. Moreover, you can modify behaviour of `scarb test` to run `snforge test` as described here.

> **Note**
>
> `Scarb.toml` is specifically designed for configuring scarb packages and, by extension, is suitable for `snforge` configurations, which are package-specific. On the other hand, `sncast` can operate independently of scarb workspaces/packages and therefore utilizes a different configuration file, `snfoundry.toml`. This distinction ensures that configurations are appropriately aligned with their respective tools' operational contexts.

Last but not least, remember that in order to use Starknet Foundry, you must have Scarb installed and added to the `PATH` environment variable.

Project Configuration

`snforge`

Configuring `snforge` Settings in `Scarb.toml`

It is possible to configure `snforge` for all test runs through `Scarb.toml`. Instead of passing arguments in the command line, set them directly in the file.

```
# ...
[tool.snforge]
exit_first = true
# ...
```

`snforge` automatically looks for `Scarb.toml` in the directory you are running the tests in or in any of its parents.

`sncast`

Defining Profiles in `snfoundry.toml`

To be able to work with the network, you need to supply `sncast` with a few parameters — namely the rpc node url and an account name that should be used to interact with it. This can be done by either supplying `sncast` with those parameters directly see more detailed CLI description, or you can put them into `snfoundry.toml` file:

```
# ...
[sncast.myprofile]
account = "user"
accounts-file = "~/my_accounts.json"
url = "http://127.0.0.1:5050/rpc"
# ...
```

With `snfoundry.toml` configured this way, we can just pass `--profile myprofile` argument to make sure `sncast` uses parameters defined in the profile.

> 📝 **Note** `snfoundry.toml` file has to be present in current or any of the parent directories.

> 📝 **Note** If there is a profile with the same name in Scarb.toml, scarb will use this profile. If not, scarb will default to using the dev profile. (This applies only to subcommands using scarb - namely `declare` and `script`).

> 💡 **Info** Not all parameters have to be present in the configuration - you can choose to include only some of them and supply the rest of them using CLI flags. You can also override parameters from the configuration using CLI flags.

```
$ sncast --profile myprofile \
    call \
    --contract-address
0x0589a8b8bf819b7820cb699ea1f6c409bc012c9b9160106ddc3dacd6a89653cf \
    --function get_balance \
    --block-id latest
```

▶ Output:

Multiple Profiles

You can have multiple profiles defined in the `snfoundry.toml`.

Default Profile

There is also an option to set up a default profile, which can be utilized without the need to specify a `--profile`. Here's an example:

```
# ...
[sncast.default]
account = "user123"
accounts-file = "~/my_accounts.json"
url = "http://127.0.0.1:5050/rpc"
# ...
```

With this, there's no need to include the `--profile` argument when using `sncast`.

```
$ sncast call \
    --contract-address
0x0589a8b8bf819b7820cb699ea1f6c409bc012c9b9160106ddc3dacd6a89653cf \
    --function get_balance \
    --block-id latest
```

▶ Output:

Global Configuration

Global configuration file is a `snfoundry.toml`, which is a common storage for configurations to apply to multiple projects across various directories. This file is stored in a predefined location and is used to store profiles that can be used from any location on your computer.

Interaction Between Local and Global Profiles

Global config can be overridden by a local config.

If both local and global profiles with the same name are present, local profile will be combined with global profile. For any setting defined in both profiles, the local setting will take precedence. For settings not defined in the local profile, values from the corresponding global profile will be used, or if not defined, values from the global default profile will be used instead.

This same behavior applies for default profiles as well. A local default profile will override a global default profile.

 **Note** Remember that arguments passed in the CLI have the highest priority and will always override the configuration file settings.

Global Configuration File Location

The global configuration is stored in a specific location depending on the operating system:

- macOS/Linux : The global configuration file is located at `$HOME/.config/starknet-foundry/snfoundry.toml`
- Windows : The file can be found at `C:\Users\<user>\AppData\Roaming\starknet-foundry\snfoundry.toml`

> 📝 **Note** If missing, global configuration file will be created automatically on running any `sncast` command for the first time.

Config Interaction Example

```
root/
├── .config/
│   └── starknet-foundry/
│       └── snfoundry.toml -> A
└── /../../
        └── projects/
            ├── snfoundry.toml -> B
            └── cairo-projects/
                └── opus-magnum/
```

Glossary:

- **A:** Global configuration file containing the profiles `default` and `testnet`.
- **B:** Local configuration file containing the profiles `default` and `mainnet`.

In any directory in the file system, a user can run the `sncast` command using the `default` and `testnet` profiles, because they are defined in global config (file A).

If no profiles are explicitly specified, the `default` profile from the global configuration file will be used.

When running `sncast` from the `opus-magnum` directory, there is a configuration file in the parent directory (file B). This setup allows for the use of the following profiles: `default`, `testnet`, and `mainnet`. If the `mainnet` profile is specified, the configuration from the local file will be used to override the global `default` profile, as the `mainnet` profile does not exist in the global configuration.

Environmental Variables

Programmers can use environmental variables in both `Scarb.toml::tool::snforge` and in `snfoundry.toml`. To use an environmental variable as a value, use its name either with or without curly braces, prefixed with `$` (e.g. `${MY_ENV}` or `$MY_ENV`). This might be useful, for example, to hide node urls in the public repositories. As an example:

```
# ...
[sncast.default]
account = "my_account"
accounts-file = "~/my_accounts.json"
url = "$NODE_URL"
# ...
```

Variable values are automatically resolved to numbers and booleans (strings `true`, `false`) where possible.

Running Tests

To run tests with `snforge`, simply run the `snforge test` command from the package directory.

```
$ snforge test
```

▶ Output:

Filtering Tests

You can pass a filter string after the `snforge test` command to filter tests. By default, any test with an absolute module tree path matching the filter will be run.

```
$ snforge test calling
```

▶ Output:

Running a Specific Test

To run a specific test, you can pass a filter string along with an `--exact` flag. Note, you have to use a fully qualified test name, including a module name.

Note

Running a specific test results in optimized compilation. `snforge` will try to compile only the desired test, unlike the case of running all tests where all of them are compiled.

```
$ snforge test hello_snforge_integrationtest::test_contract::test_calling --exact
```

▶ Output:

Stopping Test Execution After First Failed Test

To stop the test execution after first failed test, you can pass an `--exit-first` flag along with `snforge test` command.

```
$ snforge test --exit-first
```

▶ Output:

Displaying Resources Used During Tests

To track resources like `builtins` / `syscalls` that are used when running tests, use `snforge test --detailed-resources`.

```
$ snforge test --detailed-resources
```

▶ Output:

For more information about how starknet-foundry calculates those, see gas and resource estimation section.

Writing Tests

`snforge` lets you test standalone functions from your smart contracts. This technique is referred to as unit testing. You should write as many unit tests as possible as these are faster than integration tests.

Writing Your First Test

First, add the following code to the `src/lib.cairo` file:

```
fn sum(a: felt252, b: felt252) -> felt252 {
    return a + b;
}

#[cfg(test)]
mod tests {
    use super::sum;

    #[test]
    fn test_sum() {
        assert(sum(2, 3) == 5, 'sum incorrect');
    }
}
```

It is a common practice to keep your unit tests in the same file as the tested code. Keep in mind that all tests in `src` folder have to be in a module annotated with `#[cfg(test)]`. When it comes to integration tests, you can keep them in separate files in the `tests` directory. You can find a detailed explanation of how `snforge` collects tests here.

Now run `snforge` using a command:

```
$ snforge test
```

▶ Output:

Failing Tests

If your code panics, the test is considered failed. Here's an example of a failing test.

```
fn panicking_function() {
    let mut data = array![];
    data.append('panic message');
    panic(data)
}

#[cfg(test)]
mod tests {
    use super::panicking_function;

    #[test]
    fn failing() {
        panicking_function();
        assert(2 == 2, '2 == 2');
    }
}
```

```
$ snforge test
```

▶ Output:

When contract fails, you can get backtrace information by setting the `SNFORGE_BACKTRACE=1` environment variable. Read more about it here.

Expected Failures

Sometimes you want to mark a test as expected to fail. This is useful when you want to verify that an action fails as expected.

To mark a test as expected to fail, use the `#[should_panic]` attribute.

You can specify the expected failure message in three ways:

1. **With ByteArray**:

```
    #[test]
    #[should_panic(expected: "This will panic")]
    fn should_panic_exact() {
        panic!("This will panic");
    }

    // here the expected message is a substring of the actual message
    #[test]
    #[should_panic(expected: "will panic")]
    fn should_panic_expected_is_substring() {
        panic!("This will panic");
    }
```

With this format, the expected error message needs to be a substring of the actual error message. This is particularly useful when the error message includes dynamic data such as a hash or address.

2. **With felt**

```
    #[test]
    #[should_panic(expected: 'panic message')]
    fn should_panic_felt_matching() {
        assert(1 != 1, 'panic message');
    }
```

3. **With tuple of felts**:

```
    use core::panic_with_felt252;

    #[test]
    #[should_panic(expected: ('panic message',))]
    fn should_panic_check_data() {
        panic_with_felt252('panic message');
    }

    // works for multiple messages
    #[test]
    #[should_panic(expected: ('panic message', 'second message',))]
    fn should_panic_multiple_messages() {
        let mut arr = ArrayTrait::new();
        arr.append('panic message');
        arr.append('second message');
        panic(arr);
    }
```

```
$ snforge test
```

▶ Output:

Ignoring Tests

Sometimes you may have tests that you want to exclude during most runs of `snforge test`. You can achieve it using `#[ignore]` - tests marked with this attribute will be skipped by default.

```
#[cfg(test)]
mod tests {
    #[test]
    #[ignore]
    fn ignored_test() { // test code
    }
}
```

```
$ snforge test
```

▶ Output:

To run only tests marked with the `#[ignore]` attribute use `snforge test --ignored`. To run all tests regardless of the `#[ignore]` attribute use `snforge test --include-ignored`.

Writing Assertions and `assert_macros` Package

> ⚠️ **Recommended only for development** ⚠️
>
> Assert macros package provides a set of macros that can be used to write assertions such as `assert_eq!`. In order to use it, your project must have the `assert_macros` dependency added to the `Scarb.toml` file. These macros are very expensive to run on Starknet, as they result a huge amount of steps and are not recommended for production use. They are only meant to be used in tests. For snforge `v0.31.0` and later, this dependency is added automatically when creating a project using `snforge init`. But for earlier versions, you need to add it manually.

```
[dev-dependencies]
snforge_std = ...
assert_macros = "<scarb-version>"
```

Available assert macros are

- `assert_eq!`
- `assert_ne!`
- `assert_lt!`
- `assert_le!`
- `assert_gt!`
- `assert_ge!`

Test Attributes

`snforge` allows setting test attributes for test cases in order to modify their behavior.

Currently, those attributes are supported:

- `#[test]`
- `#[ignore]`
- `#[should_panic]`
- `#[available_gas]`
- `#[fork]`
- `#[fuzzer]`

`#[test]`

Marks the function as a test case, to be visible for test collector. Read more about test collection here.

`#[ignore]`

Marks the function as ignored, it will be skipped after collecting. Use this if you don't want the test to be run (the runner will display how many tests were ignored in the summary).

Read more about the behavior and how to override this here.

`#[should_panic]`

A test function can be marked with this attribute, in order to assert that the test function itself will panic. If the test panics when marked with this attribute, it's considered as "passed".

Moreover, it can be used with either a tuple of shortstrings or a string for assessment of the exit panic data (depending on what your contract throws).

Usage

Asserting the panic data can be done with multiple types of inputs:

ByteArray:

```
#[should_panic(expected: "No such file or directory (os error 2)")]
```

Shortstring:

```
#[should_panic(expected: 'panic message')]
```

Tuple of shortstrings:

```
#[should_panic(expected: ('panic message', 'second message', )]
```

Asserting that the function panics (any with any panic data):

```
#[should_panic]
```

`#[available_gas]`

Sets a gas limit for the test. If the test exceeds the limit, it fails with an appropriate error.

Usage

Asserts that the test does not use more than 5 units of gas.

```
#[available_gas(5)]
```

`#[fork]`

Enables state forking for the given test case.

Read more about fork testing here.

Usage

Configures the fork endpoint with a given URL and a reference point for forking, which can be a block number, block hash, or a named tag (only "latest" is supported).

Reference Type	Example Usage
`block_number`	`#[fork(url: "http://example.com", block_number: 123)]`
`block_hash`	`#[fork(url: "http://example.com", block_hash: 0x123deadbeef)]`
`block_tag`	`#[fork(url: "http://example.com", block_tag: latest)]`

You can also define your frequently used fork configs in your `Scarb.toml`:

```
[[tool.snforge.fork]]
name = "TESTNET"
url = "http://your.rpc.url"
block_id.tag = "latest"
```

Then, instead of repeating them inside the attribute, you can reference them by the given name of the config declared in `Scarb.toml`:

```
#[fork("TESTNET")]
```

`#[fuzzer]`

Enables fuzzing for a given test case.

Read more about test case fuzzing here.

Usage

Mark the test as fuzzed test, and configure the fuzzer itself. Configures how many runs will be performed, and the starting seed (for repeatability).

```
#[fuzzer(runs: 10, seed: 123)]
```

Any parameter of `fuzzer` attribute can be omitted:

```
#[fuzzer]
#[fuzzer(runs: 10)]
#[fuzzer(seed: 123)]
```

And will be filled in with default values in that case (default `runs` value is 256).

> ⚠️ **Warning**
>
> Please note, that the test function needs to have some parameters in order for fuzzer to have something to fuzz. Otherwise it will fail to execute and crash the runner.

Testing Smart Contracts

> **Info**
>
> To use the library functions designed for testing smart contracts, you need to add `snforge_std` package as a dependency in your `Scarb.toml` using the appropriate version.

```
[dev-dependencies]
snforge_std = "0.33.0"
```

Using unit testing as much as possible is a good practice, as it makes your test suites run faster. However, when writing smart contracts, you often want to test their interactions with the blockchain state and with other contracts.

The Test Contract

Let's consider a simple smart contract with two methods.

```
#[starknet::interface]
pub trait ISimpleContract<TContractState> {
    fn increase_balance(ref self: TContractState, amount: felt252);
    fn get_balance(self: @TContractState) -> felt252;
}

#[starknet::contract]
pub mod SimpleContract {
    #[storage]
    struct Storage {
        balance: felt252,
    }

    #[abi(embed_v0)]
    pub impl SimpleContractImpl of super::ISimpleContract<ContractState> {
        // Increases the balance by the given amount
        fn increase_balance(ref self: ContractState, amount: felt252) {
            self.balance.write(self.balance.read() + amount);
        }

        // Gets the balance.
        fn get_balance(self: @ContractState) -> felt252 {
            self.balance.read()
        }
    }
}
```

Note that the name after `mod` will be used as the contract name for testing purposes.

Writing Tests

Let's write a test that will deploy the `SimpleContract` contract and call some functions.

```
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait};

use testing_smart_contracts_writing_tests::{
    ISimpleContractDispatcher, ISimpleContractDispatcherTrait
};

#[test]
fn call_and_invoke() {
    // First declare and deploy a contract
    let contract = declare("SimpleContract").unwrap().contract_class();
    // Alternatively we could use `deploy_syscall` here
    let (contract_address, _) = contract.deploy(@array![]).unwrap();

    // Create a Dispatcher object that will allow interacting with the deployed
contract
    let dispatcher = ISimpleContractDispatcher { contract_address };

    // Call a view function of the contract
    let balance = dispatcher.get_balance();
    assert(balance == 0, 'balance == 0');

    // Call a function of the contract
    // Here we mutate the state of the storage
    dispatcher.increase_balance(100);

    // Check that transaction took effect
    let balance = dispatcher.get_balance();
    assert(balance == 100, 'balance == 100');
}
```

📝 Note

Notice that the arguments to the contract's constructor (the `deploy`'s `calldata` argument) need to be serialized with `Serde`.

`SimpleContract` contract has no constructor, so the calldata remains empty in the example above.

```
$ snforge test
```

▶ Output:

Handling Errors

Sometimes we want to test contracts functions that can panic, like testing that function that verifies caller address panics on invalid address. For that purpose Starknet also provides a `SafeDispatcher`, that returns a `Result` instead of panicking.

First, let's add a new, panicking function to our contract.

```
#[starknet::interface]
pub trait IPanicContract<TContractState> {
    fn do_a_panic(self: @TContractState);
    fn do_a_string_panic(self: @TContractState);
}

#[starknet::contract]
pub mod PanicContract {
    use core::array::ArrayTrait;

    #[storage]
    struct Storage {}

    #[abi(embed_v0)]
    pub impl PanicContractImpl of super::IPanicContract<ContractState> {
        // Panics
        fn do_a_panic(self: @ContractState) {
            panic(array!['PANIC', 'DAYTAH']);
        }

        fn do_a_string_panic(self: @ContractState) {
            // A macro which allows panicking with a ByteArray (string) instance
            panic!("This is panicking with a string, which can be longer than 31
characters");
        }
    }
}
```

If we called this function in a test, it would result in a failure.

```
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait};

use testing_smart_contracts_handling_errors::{
    IPanicContractDispatcher, IPanicContractDispatcherTrait
};

#[test]
fn failing() {
    let contract = declare("PanicContract").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();
    let dispatcher = IPanicContractDispatcher { contract_address };

    dispatcher.do_a_panic();
}
```

```
$ snforge test
```

▶ Output:

SafeDispatcher

Using `SafeDispatcher` we can test that the function in fact panics with an expected message. Safe dispatcher is a special kind of dispatcher, which are not allowed in contracts themselves, but are available for testing purposes.

They allow using the contract without automatically unwrapping the result, which allows to catch the error like shown below.

```
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait};

use testing_smart_contracts_safe_dispatcher::{
    IPanicContractSafeDispatcher, IPanicContractSafeDispatcherTrait
};

#[test]
#[feature("safe_dispatcher")]
fn handling_errors() {
    let contract = declare("PanicContract").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();
    let safe_dispatcher = IPanicContractSafeDispatcher { contract_address };

    match safe_dispatcher.do_a_panic() {
        Result::Ok(_) => panic!("Entrypoint did not panic"),
        Result::Err(panic_data) => {
            assert(*panic_data.at(0) == 'PANIC', *panic_data.at(0));
            assert(*panic_data.at(1) == 'DAYTAH', *panic_data.at(1));
        }
    };
}
```

Now the test passes as expected.

```
$ snforge test
```

▶ Output:

Similarly, you can handle the panics which use `ByteArray` as an argument (like an `assert!` or `panic!` macro)

```
// Necessary utility function import
use snforge_std::byte_array::try_deserialize_bytearray_error;
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait};

use testing_smart_contracts_handling_errors::{
    IPanicContractSafeDispatcher, IPanicContractSafeDispatcherTrait
};

#[test]
#[feature("safe_dispatcher")]
fn handling_string_errors() {
    let contract = declare("PanicContract").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();
    let safe_dispatcher = IPanicContractSafeDispatcher { contract_address };

    match safe_dispatcher.do_a_string_panic() {
        Result::Ok(_) => panic!("Entrypoint did not panic"),
        Result::Err(panic_data) => {
            let str_err =
try_deserialize_bytearray_error(panic_data.span()).expect('wrong format');

            assert(
                str_err == "This is panicking with a string, which can be longer
than 31 characters",
                'wrong string received'
            );
        }
    };
}
```

You also could skip the de-serialization of the `panic_data`, and not use `try_deserialize_bytearray_error`, but this way you can actually use assertions on the `ByteArray` that was used to panic.

📝 Note

To operate with `SafeDispatcher` it's required to annotate its usage with `#[feature("safe_dispatcher")]`.

There are 3 options:

- module-level declaration

```
#[feature("safe_dispatcher")]
mod my_module;
```

- function-level declaration

```
#[feature("safe_dispatcher")]
fn my_function() { ... }
```

- directly before the usage

```
#[feature("safe_dispatcher")]
let result = safe_dispatcher.some_function();
```

Expecting Test Failure

Sometimes the test code failing can be a desired behavior. Instead of manually handling it, you can simply mark your test as `#[should_panic(...)]`. See here for more details.

Testing Contracts' Internals

Sometimes, you want to test a function which uses Starknet context (like block number, timestamp, storage access) without deploying the actual contract.

Since every test is treated like a contract, using the aforementioned pattern you can test:

- functions which are not available through the interface (but your contract uses them)
- functions which are internal
- functions performing specific operations on the contracts' storage or context data
- library calls directly in the tests

Utilities For Testing Internals

To facilitate such use cases, we have a handful of utilities which make a test behave like a contract.

`contract_state_for_testing()` - State of Test Contract

This is a function generated by the `#[starknet::contract]` macro. It can be used to test some functions which accept the state as an argument, see the example below:

```
#[starknet::interface]
trait IContract<TContractState> {}

#[starknet::contract]
pub mod Contract {
    #[storage]
    pub struct Storage {
        pub balance: felt252,
    }

    #[generate_trait]
    pub impl InternalImpl of InternalTrait {
        fn internal_function(self: @ContractState) -> felt252 {
            self.balance.read()
        }
    }

    pub fn other_internal_function(self: @ContractState) -> felt252 {
        self.balance.read() + 5
    }
}

#[cfg(test)]
mod tests {
    use core::starknet::storage::{
        StoragePointerReadAccess, StoragePointerWriteAccess
    }; // <--- Ad. 1
    use super::Contract;
    use super::Contract::{InternalTrait, other_internal_function}; // <--- Ad. 2

    #[test]
    fn test_internal() {
        let mut state = Contract::contract_state_for_testing(); // <--- Ad. 3
        state.balance.write(10);

        let value = state.internal_function();
        assert(value == 10, 'Incorrect storage value');

        let other_value = other_internal_function(@state);
        assert(other_value == 15, 'Incorrect return value');
    }
}
```

This code contains some caveats:

1. To access `read/write` methods of the state fields (in this case it's `balance`) you need to also import `StoragePointerReadAccess` and `StoragePointerWriteAccess` for reading and writing respectively.
2. To access functions implemented directly on the state you need to also import an appropriate trait or function.

3. This function will always return the struct keeping track of the state of the test. It means that within one test every result of `contract_state_for_testing` actually points to the same state.

`snforge_std::test_address()` - Address of Test Contract

That function returns the contract address of the test. It is useful, when you want to:

- Mock the context (`cheat_caller_address` , `cheat_block_timestamp` , `cheat_block_number` , ...)
- Spy for events emitted in the test

Example usages:

1. Mocking the context info

Example for `cheat_block_number` , same can be implemented for `cheat_caller_address` / `cheat_block_timestamp` / `elect` etc.

```
use core::result::ResultTrait;
use core::box::BoxTrait;
use starknet::ContractAddress;
use snforge_std::{start_cheat_block_number, stop_cheat_block_number,
test_address};

#[test]
fn test_cheat_block_number_test_state() {
    let test_address: ContractAddress = test_address();
    let old_block_number = starknet::get_block_info().unbox().block_number;

    start_cheat_block_number(test_address, 234);
    let new_block_number = starknet::get_block_info().unbox().block_number;
    assert(new_block_number == 234, 'Wrong block number');

    stop_cheat_block_number(test_address);
    let new_block_number = starknet::get_block_info().unbox().block_number;
    assert(new_block_number == old_block_number, 'Block num did not change back');
}
```

2. Spying for events

You can use both `starknet::emit_event_syscall` , and the spies will capture the events, emitted in a `#[test]` function, if you pass the `test_address()` as a spy parameter (or spy on all events).

Given the emitting contract implementation:

```
#[starknet::interface]
pub trait IEmitter<TContractState> {
    fn emit_event(ref self: TContractState);
}

#[starknet::contract]
pub mod Emitter {
    use core::result::ResultTrait;
    use starknet::ClassHash;

    #[event]
    #[derive(Drop, starknet::Event)]
    pub enum Event {
        ThingEmitted: ThingEmitted
    }

    #[derive(Drop, starknet::Event)]
    pub struct ThingEmitted {
        pub thing: felt252
    }

    #[storage]
    struct Storage {}

    #[external(v0)]
    pub fn emit_event(ref self: ContractState) {
        self.emit(Event::ThingEmitted(ThingEmitted { thing: 420 }));
    }
}
```

You can implement this test:

```
use core::array::ArrayTrait;
use snforge_std::{
    declare, ContractClassTrait, spy_events, EventSpy, EventSpyTrait,
EventSpyAssertionsTrait,
    Event, test_address
};
use testing_contract_internals::spying_for_events::Emitter;

#[test]
fn test_expect_event() {
    let contract_address = test_address();
    let mut spy = spy_events();

    let mut testing_state = Emitter::contract_state_for_testing();
    Emitter::emit_event(ref testing_state);

    spy
        .assert_emitted(
            @array![
                (
                    contract_address,
                    Emitter::Event::ThingEmitted(Emitter::ThingEmitted { thing:
420 })
                )
            ]
        )
}
```

You can also use the `starknet::emit_event_syscall` directly in the tests:

```
use core::array::ArrayTrait;
use core::result::ResultTrait;
use starknet::{ContractAddress, SyscallResultTrait, syscalls::emit_event_syscall};
use snforge_std::{
    declare, ContractClassTrait, spy_events, EventSpy, EventSpyTrait,
EventSpyAssertionsTrait,
    Event, test_address
};

#[test]
fn test_expect_event() {
    let contract_address = test_address();
    let mut spy = spy_events();

    emit_event_syscall(array![1234].span(), array![2345].span()).unwrap_syscall();

    spy
        .assert_emitted(
            @array![(contract_address, Event { keys: array![1234], data: array!
[2345] })]
        );

    assert(spy.get_events().events.len() == 1, 'There should no more events');
}
```

Using Library Calls With the Test State Context

Using the above utilities, you can avoid deploying a mock contract, to test a `library_call` with a `LibraryCallDispatcher`.

For contract implementation:

```
#[starknet::interface]
pub trait ILibraryContract<TContractState> {
    fn get_value(self: @TContractState,) -> felt252;
    fn set_value(ref self: TContractState, number: felt252);
}

#[starknet::contract]
pub mod LibraryContract {
    #[storage]
    struct Storage {
        value: felt252
    }

    #[external(v0)]
    pub fn get_value(self: @ContractState,) -> felt252 {
        self.value.read()
    }

    #[external(v0)]
    pub fn set_value(ref self: ContractState, number: felt252) {
        self.value.write(number);
    }
}
```

We use the `SafeLibraryDispatcher` like this:

```
use testing_contract_internals::using_library_calls::{
    ILibraryContractSafeLibraryDispatcher, ILibraryContractSafeDispatcherTrait
};
use starknet::{ClassHash, ContractAddress, syscalls::library_call_syscall};
use snforge_std::{declare, DeclareResultTrait};

#[test]
fn test_library_calls() {
    let class_hash =
declare("LibraryContract").unwrap().contract_class().class_hash.clone();
    let lib_dispatcher = ILibraryContractSafeLibraryDispatcher { class_hash };

    let value = lib_dispatcher.get_value().unwrap();
    assert(value == 0, 'Incorrect state');

    lib_dispatcher.set_value(10).unwrap();

    let value = lib_dispatcher.get_value().unwrap();
    assert(value == 10, 'Incorrect state');
}
```

⚠ Warning

This library call will write to the `test_address` memory segment, so it can potentially **overwrite** the changes you make to the memory through `contract_state_for_testing` object and vice-versa.

Using Cheatcodes

ℹ️ **Info** To use cheatcodes you need to add `snforge_std` package as a dependency in your `Scarb.toml` using the appropriate version.

```
[dev-dependencies]
snforge_std = "0.33.0"
```

When testing smart contracts, often there are parts of code that are dependent on a specific blockchain state. Instead of trying to replicate these conditions in tests, you can emulate them using cheatcodes.

> ⚠️ **Warning**
>
> These examples make use of `assert_macros`, so it's recommended to get familiar with them first. Learn more about `assert_macros`

The Test Contract

In this tutorial, we will be using the following Starknet contract:

```
#[starknet::interface]
pub trait ICheatcodeChecker<TContractState> {
    fn increase_balance(ref self: TContractState, amount: felt252);
    fn get_balance(self: @TContractState) -> felt252;
    fn get_block_number_at_construction(self: @TContractState) -> u64;
    fn get_block_timestamp_at_construction(self: @TContractState) -> u64;
}

#[starknet::contract]
pub mod CheatcodeChecker {
    use core::box::BoxTrait;
    use starknet::get_caller_address;

    #[storage]
    struct Storage {
        balance: felt252,
        blk_nb: u64,
        blk_timestamp: u64,
    }

    #[constructor]
    fn constructor(ref self: ContractState) {
        // store the current block number
        self.blk_nb.write(starknet::get_block_info().unbox().block_number);
        // store the current block timestamp

self.blk_timestamp.write(starknet::get_block_info().unbox().block_timestamp);
    }

    #[abi(embed_v0)]
    impl ICheatcodeCheckerImpl of super::ICheatcodeChecker<ContractState> {
        // Increases the balance by the given amount
        fn increase_balance(ref self: ContractState, amount: felt252) {
            assert_is_allowed_user();
            self.balance.write(self.balance.read() + amount);
        }
        // Gets the balance.
        fn get_balance(self: @ContractState) -> felt252 {
            self.balance.read()
        }
        // Gets the block number
        fn get_block_number_at_construction(self: @ContractState) -> u64 {
            self.blk_nb.read()
        }
        // Gets the block timestamp
        fn get_block_timestamp_at_construction(self: @ContractState) -> u64 {
            self.blk_timestamp.read()
        }
    }

    fn assert_is_allowed_user() {
        // checks if caller is '123'
        let address = get_caller_address();
        assert(address.into() == 123, 'user is not allowed');
```

```
    }
}
```

Writing Tests

We can try to create a test that will increase and verify the balance.

```
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait};
use using_cheatcodes::{ICheatcodeCheckerDispatcher,
ICheatcodeCheckerDispatcherTrait};

#[test]
fn call_and_invoke() {
    let contract = declare("CheatcodeChecker").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();
    let dispatcher = ICheatcodeCheckerDispatcher { contract_address };

    let balance = dispatcher.get_balance();
    assert(balance == 0, 'balance == 0');

    dispatcher.increase_balance(100);

    let balance = dispatcher.get_balance();
    assert(balance == 100, 'balance == 100');
}
```

This test fails, which means that `increase_balance` method panics as we expected.

```
$ snforge test
```

▶ Output:

Our user validation is not letting us call the contract, because the default caller address is not `123`.

Using Cheatcodes in Tests

By using cheatcodes, we can change various properties of transaction info, block info, etc. For example, we can use the `start_cheat_caller_address` cheatcode to change the caller address, so it passes our validation.

Cheating an Address

```
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait,
start_cheat_caller_address};
use using_cheatcodes_cheat_address::{ICheatcodeCheckerDispatcher,
ICheatcodeCheckerDispatcherTrait};

#[test]
fn call_and_invoke() {
    let contract = declare("CheatcodeChecker").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();
    let dispatcher = ICheatcodeCheckerDispatcher { contract_address };

    let balance = dispatcher.get_balance();
    assert(balance == 0, 'balance == 0');

    // Change the caller address to 123 when calling the contract at the
`contract_address` address
    start_cheat_caller_address(contract_address, 123.try_into().unwrap());

    dispatcher.increase_balance(100);

    let balance = dispatcher.get_balance();
    assert(balance == 100, 'balance == 100');
}
```

The test will now pass without an error

```
$ snforge test
```

▶ Output:

Cancelling the Cheat

Most cheatcodes come with corresponding `start_` and `stop_` functions that can be used to start and stop the state change. In case of the `start_cheat_caller_address`, we can cancel the address change using `stop_cheat_caller_address`. We will demonstrate its behavior using `SafeDispatcher` to show when exactly the fail occurs:

```
use snforge_std::{
    declare, ContractClassTrait, DeclareResultTrait, start_cheat_caller_address,
    stop_cheat_caller_address
};

use using_cheatcodes_cancelling_cheat::{
    ICheatcodeCheckerSafeDispatcher, ICheatcodeCheckerSafeDispatcherTrait
};

#[test]
#[feature("safe_dispatcher")]
fn call_and_invoke() {
    let contract = declare("CheatcodeChecker").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();
    let dispatcher = ICheatcodeCheckerSafeDispatcher { contract_address };

    let balance = dispatcher.get_balance().unwrap();
    assert(balance == 0, 'balance == 0');

    // Change the caller address to 123 when calling the contract at the
`contract_address` address
    start_cheat_caller_address(contract_address, 123.try_into().unwrap());

    // Call to method with caller restriction succeeds
    dispatcher.increase_balance(100).expect('First call failed!');

    let balance = dispatcher.get_balance();
    assert_eq!(balance, Result::Ok(100));

    // Cancel the cheat
    stop_cheat_caller_address(contract_address);

    // The call fails now
    dispatcher.increase_balance(100).expect('Second call failed!');

    let balance = dispatcher.get_balance();
    assert_eq!(balance, Result::Ok(100));
}
```

```
$ snforge test
```

▶ Output:

We see that the second `increase_balance` fails since we cancelled the cheatcode.

Cheating Addresses Globally

In case you want to cheat the caller address for all contracts, you can use the global cheatcode which has the `_global` suffix. Note, that we don't specify target, nor the span, because this cheatcode type works globally and indefinitely. For more see Cheating Globally.

```
use snforge_std::{
    declare, ContractClassTrait, DeclareResultTrait,
start_cheat_caller_address_global,
    stop_cheat_caller_address_global
};
use using_cheatcodes_others::{ICheatcodeCheckerDispatcher,
ICheatcodeCheckerDispatcherTrait};

#[test]
fn call_and_invoke_global() {
    let contract = declare("CheatcodeChecker").unwrap().contract_class();
    let (contract_address_a, _) = contract.deploy(@array![]).unwrap();
    let (contract_address_b, _) = contract.deploy(@array![]).unwrap();
    let dispatcher_a = ICheatcodeCheckerDispatcher { contract_address:
contract_address_a };
    let dispatcher_b = ICheatcodeCheckerDispatcher { contract_address:
contract_address_b };

    let balance_a = dispatcher_a.get_balance();
    let balance_b = dispatcher_b.get_balance();
    assert_eq!(balance_a, 0);
    assert_eq!(balance_b, 0);

    // Change the caller address to 123, both targets a and b will be affected
    // global cheatcodes work indefinitely until stopped
    start_cheat_caller_address_global(123.try_into().unwrap());

    dispatcher_a.increase_balance(100);
    dispatcher_b.increase_balance(100);

    let balance_a = dispatcher_a.get_balance();
    let balance_b = dispatcher_b.get_balance();
    assert_eq!(balance_a, 100);
    assert_eq!(balance_b, 100);

    // Cancel the cheat
    stop_cheat_caller_address_global();
}
```

Cheating the Constructor

Most of the cheatcodes like `cheat_caller_address`, `mock_call`, `cheat_block_timestamp`, `cheat_block_number`, `elect` do work in the constructor of the contracts.

Let's say, that you have a contract that saves the caller address (deployer) in the constructor, and you want it to be pre-set to a certain value.

To `cheat_caller_address` the constructor, you need to `start_cheat_caller_address` before it is invoked, with the right address. To achieve this, you need to precalculate the address of the contract by using the `precalculate_address` function of `ContractClassTrait` on the declared contract, and then use it in `start_cheat_caller_address` as an argument:

```
use snforge_std::{
    declare, ContractClassTrait, DeclareResultTrait, start_cheat_block_number,
    start_cheat_block_timestamp
};

use using_cheatcodes_others::{ICheatcodeCheckerDispatcher,
ICheatcodeCheckerDispatcherTrait};

#[test]
fn call_and_invoke() {
    let contract = declare("CheatcodeChecker").unwrap().contract_class();

    // Precalculate the address to obtain the contract address before the
constructor call (deploy)
    // itself
    let contract_address = contract.precalculate_address(@array![]);

    // Change the block number and timestamp before the call to contract.deploy
    start_cheat_block_number(contract_address, 0x420_u64);
    start_cheat_block_timestamp(contract_address, 0x2137_u64);

    // Deploy as normally
    contract.deploy(@array![]).unwrap();

    // Construct a dispatcher with the precalculated address
    let dispatcher = ICheatcodeCheckerDispatcher { contract_address };

    let block_number = dispatcher.get_block_number_at_construction();
    let block_timestamp = dispatcher.get_block_timestamp_at_construction();

    assert_eq!(block_number, 0x420_u64);
    assert_eq!(block_timestamp, 0x2137_u64);
}
```

Setting Cheatcode Span

Sometimes it's useful to have a cheatcode work only for a certain number of target calls.

That's where `CheatSpan` comes in handy.

```
enum CheatSpan {
    Indefinite: (),
    TargetCalls: usize,
}
```

To set span for a cheatcode, use `cheat_caller_address` / `cheat_block_timestamp` / `cheat_block_number` / etc.

```
cheat_caller_address(contract_address, new_caller_address,
CheatSpan::TargetCalls(1))
```

Calling a cheatcode with `CheatSpan::TargetCalls(N)` is going to activate the cheatcode for `N` calls to a specified contract address, after which it's going to be automatically canceled.

Of course the cheatcode can still be canceled before its `CheatSpan` goes down to 0 - simply call `stop_cheat_caller_address` on the target manually.

> ℹ️ **Info**
>
> Using `start_cheat_caller_address` is **equivalent** to using `cheat_caller_address` with `CheatSpan::Indefinite`.

To better understand the functionality of `CheatSpan`, here's a full example:

```
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait,
cheat_caller_address, CheatSpan};
use starknet::ContractAddress;

use using_cheatcodes_others::{
    ICheatcodeCheckerSafeDispatcher, ICheatcodeCheckerSafeDispatcherTrait
};

#[test]
#[feature("safe_dispatcher")]
fn call_and_invoke() {
    let contract = declare("CheatcodeChecker").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();
    let safe_dispatcher = ICheatcodeCheckerSafeDispatcher { contract_address };

    let balance = safe_dispatcher.get_balance().unwrap();
    assert_eq!(balance, 0);

    // Function `increase_balance` from HelloStarknet contract
    // requires the caller_address to be 123
    let spoofed_caller: ContractAddress = 123.try_into().unwrap();

    // Change the caller address for the contract_address for a span of 2 target
calls (here, calls
    // to contract_address)
    cheat_caller_address(contract_address, spoofed_caller,
CheatSpan::TargetCalls(2));

    // Call #1 should succeed
    let call_1_result = safe_dispatcher.increase_balance(100);
    assert!(call_1_result.is_ok());

    // Call #2 should succeed
    let call_2_result = safe_dispatcher.increase_balance(100);
    assert!(call_2_result.is_ok());

    // Call #3 should fail, as the cheat_caller_address cheatcode has been
canceled
    let call_3_result = safe_dispatcher.increase_balance(100);
    assert_eq!(call_3_result, Result::Err(array!['user is not allowed']));

    let balance = safe_dispatcher.get_balance().unwrap();
    assert_eq!(balance, 200);
}
```

Testing events

Examples are based on the following `SpyEventsChecker` contract implementation:

```
#[starknet::interface]
pub trait ISpyEventsChecker<TContractState> {
    fn emit_one_event(ref self: TContractState, some_data: felt252);
}

#[starknet::contract]
pub mod SpyEventsChecker {
    #[storage]
    struct Storage {}

    #[event]
    #[derive(Drop, starknet::Event)]
    pub enum Event {
        FirstEvent: FirstEvent
    }

    #[derive(Drop, starknet::Event)]
    pub struct FirstEvent {
        pub some_data: felt252
    }

    #[external(v0)]
    pub fn emit_one_event(ref self: ContractState, some_data: felt252) {
        self.emit(FirstEvent { some_data });
    }
}
```

Asserting emission with `assert_emitted` method

This is the simpler way, in which you don't have to fetch the events explicitly. See the below code for reference:

```
use snforge_std::{
    declare, ContractClassTrait, DeclareResultTrait, spy_events,
    EventSpyAssertionsTrait, // Add for assertions on the EventSpy
};

use testing_events::contract::{
    SpyEventsChecker, ISpyEventsCheckerDispatcher,
ISpyEventsCheckerDispatcherTrait
};

#[test]
fn test_simple_assertions() {
    let contract = declare("SpyEventsChecker").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();
    let dispatcher = ISpyEventsCheckerDispatcher { contract_address };

    let mut spy = spy_events(); // Ad. 1

    dispatcher.emit_one_event(123);

    spy
        .assert_emitted(
            @array![ // Ad. 2
                (
                    contract_address,
                    SpyEventsChecker::Event::FirstEvent(
                        SpyEventsChecker::FirstEvent { some_data: 123 }
                    )
                )
            ]
        );
}
```

Let's go through the code:

1. After contract deployment, we created the spy using `spy_events` cheatcode. From this moment all emitted events will be spied.
2. Asserting is done using the `assert_emitted` method. It takes an array snapshot of `(ContractAddress, event)` tuples we expect that were emitted.

📝 **Note** We can pass events defined in the contract and construct them like in the `self.emit` method!

Asserting lack of event emission with `assert_not_emitted`

In cases where you want to test an event was *not* emitted, use the `assert_not_emitted` function. It works similarly as `assert_emitted` with the only difference that it panics if an event was emitted during the execution.

Given the example above, we can check that a different `FirstEvent` was not emitted:

```
spy.assert_not_emitted(@array![
    (
        contract_address,
        SpyEventsChecker::Event::FirstEvent(
            SpyEventsChecker::FirstEvent { some_data: 456 }
        )
    )
]);
```

Note that both the event name and event data are checked. If a function emitted an event with the same name but a different payload, the `assert_not_emitted` function will pass.

Asserting the events manually

If you wish to assert the data manually, you can do that on the `Events` structure. Simply call `get_events()` on your `EventSpy` and access `events` field on the returned `Events` value. Then, you can access the events and assert data by yourself.

```
use snforge_std::{
    declare, ContractClassTrait, DeclareResultTrait, spy_events,
EventSpyAssertionsTrait,
    EventSpyTrait, // Add for fetching events directly
    Event, // A structure describing a raw `Event`
};

use testing_events::contract::{
    SpyEventsChecker, ISpyEventsCheckerDispatcher,
ISpyEventsCheckerDispatcherTrait
};

#[test]
fn test_complex_assertions() {
    let contract = declare("SpyEventsChecker").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();
    let dispatcher = ISpyEventsCheckerDispatcher { contract_address };

    let mut spy = spy_events(); // Ad 1.

    dispatcher.emit_one_event(123);

    let events = spy.get_events(); // Ad 2.

    assert(events.events.len() == 1, 'There should be one event');

    let (from, event) = events.events.at(0); // Ad 3.
    assert(from == @contract_address, 'Emitted from wrong address');
    assert(event.keys.len() == 1, 'There should be one key');
    assert(event.keys.at(0) == @selector!("FirstEvent"), 'Wrong event name'); //
Ad 4.
    assert(event.data.len() == 1, 'There should be one data');
}
```

Let's go through important parts of the provided code:

1. After contract deployment we created the spy with `spy_events` cheatcode. From this moment all events emitted by the `SpyEventsChecker` contract will be spied.
2. We have to call `get_events` method on the created spy to fetch our events and get the `Events` structure.
3. To get our particular event, we need to access the `events` property and get the event under an index. Since `events` is an array holding a tuple of `ContractAddress` and `Event`, we unpack it using `let (from, event)`.
4. If the event is emitted by calling `self.emit` method, its hashed name is saved under the `keys.at(0)` (this way Starknet handles events)

Note To assert the `name` property we have to hash a string with the `selector!`

macro.

Filtering Events

Sometimes, when you assert the events manually, you might not want to get all the events, but only ones from a particular address. You can address that by using the method `emitted_by` on the `Events` structure.

```
use snforge_std::{
    declare, ContractClassTrait, DeclareResultTrait, spy_events,
EventSpyAssertionsTrait,
    EventSpyTrait, Event,
    EventsFilterTrait, // Add for filtering the Events object (result of
`get_events`)
};

use testing_events::contract::{
    SpyEventsChecker, ISpyEventsCheckerDispatcher,
ISpyEventsCheckerDispatcherTrait
};

#[test]
fn test_assertions_with_filtering() {
    let contract = declare("SpyEventsChecker").unwrap().contract_class();
    let (first_address, _) = contract.deploy(@array![]).unwrap();
    let (second_address, _) = contract.deploy(@array![]).unwrap();

    let first_dispatcher = ISpyEventsCheckerDispatcher { contract_address:
first_address };
    let second_dispatcher = ISpyEventsCheckerDispatcher { contract_address:
second_address };

    let mut spy = spy_events();

    first_dispatcher.emit_one_event(123);
    second_dispatcher.emit_one_event(234);
    second_dispatcher.emit_one_event(345);

    let events_from_first_address = spy.get_events().emitted_by(first_address);
    let events_from_second_address = spy.get_events().emitted_by(second_address);

    let (from_first, event_from_first) = events_from_first_address.events.at(0);
    assert(from_first == @first_address, 'Emitted from wrong address');
    assert(event_from_first.data.at(0) == @123.into(), 'Data should be 123');

    let (from_second_one, event_from_second_one) =
events_from_second_address.events.at(0);
    assert(from_second_one == @second_address, 'Emitted from wrong address');
    assert(event_from_second_one.data.at(0) == @234.into(), 'Data should be 234');

    let (from_second_two, event_from_second_two) =
events_from_second_address.events.at(1);
    assert(from_second_two == @second_address, 'Emitted from wrong address');
    assert(event_from_second_two.data.at(0) == @345.into(), 'Data should be 345');
}
```

`events_from_first_address` has events emitted by the first contract only. Similarly, `events_from_second_address` has events emitted by the second contract.

Asserting Events Emitted With `emit_event_syscall`

Events emitted with `emit_event_syscall` could have nonstandard (not defined anywhere) keys and data. They can also be asserted with `spy.assert_emitted` method.

Let's extend our `SpyEventsChecker` with `emit_event_with_syscall` method:

```
#[starknet::interface]
pub trait ISpySyscallEventsChecker<TContractState> {
    fn emit_one_event(ref self: TContractState, some_data: felt252);
    fn emit_event_with_syscall(ref self: TContractState, some_key: felt252,
some_data: felt252);
}

#[starknet::contract]
pub mod SpySyscallEventsChecker {
    // ...
    // Rest of the implementation identical to `SpyEventsChecker`

    use core::starknet::{SyscallResultTrait, syscalls::emit_event_syscall};

    #[external(v0)]
    pub fn emit_event_with_syscall(ref self: ContractState, some_key: felt252,
some_data: felt252) {
        emit_event_syscall(array![some_key].span(), array!
[some_data].span()).unwrap_syscall();
    }
}
```

And add a test for it:

```
use snforge_std::{
    declare, ContractClassTrait, DeclareResultTrait, spy_events,
EventSpyAssertionsTrait,
    EventSpyTrait, Event, EventsFilterTrait,
};

use testing_events::syscall::{
    ISpySyscallEventsCheckerDispatcher, ISpySyscallEventsCheckerDispatcherTrait
};

#[test]
fn test_nonstandard_events() {
    let contract = declare("SpySyscallEventsChecker").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();
    let dispatcher = ISpySyscallEventsCheckerDispatcher { contract_address };

    let mut spy = spy_events();
    dispatcher.emit_event_with_syscall(123, 456);

    spy.assert_emitted(@array![(contract_address, Event { keys: array![123], data:
array![456] })]);
}
```

Using `Event` struct from the `snforge_std` library we can easily assert nonstandard events. This also allows for testing the events you don't have the code of, or you don't want to import those.

Testing messages to L1

There exists a functionality allowing you to spy on messages sent to L1, similar to spying events.

Check the appendix for an exact API, structures and traits reference

Asserting messages to L1 is much simpler, since they are not wrapped with any structures in Cairo code (they are a plain `felt252` array and an L1 address). In `snforge` they are expressed with a structure:

```
/// Raw message to L1 format (as seen via the RPC-API), can be used for asserting
the sent messages.
struct MessageToL1 {
    /// An ethereum address where the message is destined to go
    to_address: starknet::EthAddress,
    /// Actual payload which will be delivered to L1 contract
    payload: Array<felt252>
}
```

Similarly, you can use `snforge` library and call `spy_messages_to_l1()` to initiate a spy:

```
use snforge_std::{spy_messages_to_l1};

#[test]
fn test_spying_l1_messages() {
    let mut spy = spy_messages_to_l1();
    // ...
}
```

With the spy ready to use, you can execute some code, and make the assertions:

1. Either with the spy directly by using `assert_sent` / `assert_not_sent` methods from `MessageToL1SpyAssertionsTrait` trait:

```
use starknet::EthAddress;
use snforge_std::{
    declare, ContractClassTrait, DeclareResultTrait, spy_messages_to_l1,
    MessageToL1SpyAssertionsTrait, MessageToL1,
};

use testing_messages_to_l1::{IMessageSenderDispatcher,
IMessageSenderDispatcherTrait};

#[test]
fn test_spying_l1_messages() {
    let mut spy = spy_messages_to_l1();

    let contract = declare("MessageSender").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();

    let dispatcher = IMessageSenderDispatcher { contract_address };

    let receiver_address: felt252 = 0x2137;
    dispatcher.greet_ethereum(receiver_address);

    let expected_payload = array!['hello'];
    let receiver_l1_address: EthAddress = receiver_address.try_into().unwrap();

    spy
        .assert_sent(
            @array![
                (
                    contract_address, // Message sender
                    MessageToL1 { // Message content (receiver and payload)
                        to_address: receiver_l1_address, payload: expected_payload
                    }
                )
            ]
        );
}
```

2. Or use the messages' contents directly via `get_messages()` method of the `MessageToL1SpyTrait`:

```
use starknet::EthAddress;
use snforge_std::{
    declare, ContractClassTrait, DeclareResultTrait, spy_messages_to_l1,
MessageToL1SpyTrait,
    MessageToL1SpyAssertionsTrait, MessageToL1, MessageToL1FilterTrait,
};

use testing_messages_to_l1::{IMessageSenderDispatcher,
IMessageSenderDispatcherTrait};

#[test]
fn test_spying_l1_messages_details() {
    let mut spy = spy_messages_to_l1();

    let contract = declare("MessageSender").unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@array![]).unwrap();

    let dispatcher = IMessageSenderDispatcher { contract_address };

    let receiver_address: felt252 = 0x2137;
    let receiver_l1_address: EthAddress = receiver_address.try_into().unwrap();

    dispatcher.greet_ethereum(receiver_address);

    let messages = spy.get_messages();

    // Use filtering optionally on MessagesToL1 instance
    let messages_from_specific_address = messages.sent_by(contract_address);
    let messages_to_specific_address =
messages_from_specific_address.sent_to(receiver_l1_address);

    // Get the messages from the MessagesToL1 structure
    let (from, message) = messages_to_specific_address.messages.at(0);

    // Assert the sender
    assert!(*from == contract_address, "Sent from wrong address");

    // Assert the MessageToL1 fields
    assert!(*message.to_address == receiver_l1_address, "Wrong L1 address of the
receiver");
    assert!(message.payload.len() == 1, "There should be 3 items in the data");
    assert!(*message.payload.at(0) == 'hello', "Expected \"hello\" in payload");
}
```

Testing Scarb Workspaces

`snforge` supports Scarb Workspaces. To make sure you know how workspaces work, check Scarb documentation here.

Workspaces With Root Package

When running `snforge test` in a Scarb workspace with a root package, it will only run tests inside the root package.

For a project structure like this

```
$ tree . -L 3
```

▶ Output:

only the tests in `./src` and `./tests` folders will be executed.

```
$ snforge test
```

▶ Output:

To select the specific package to test, pass a `--package package_name` (or `-p package_name` for short) flag. You can also run `snforge test` from the package directory to achieve the same effect.

```
$ snforge test --package addition
```

▶ Output:

You can also pass `--workspace` flag to run tests for all packages in the workspace.

```
$ snforge test --workspace
```

▶ Output:

`--package` and `--workspace` flags are mutually exclusive, adding both of them to a `snforge test` command will result in an error.

Virtual Workspaces

Running `snforge test` command in a virtual workspace (a workspace without a root package) outside any package will by default run tests for all the packages. It is equivalent to running `snforge test` with the `--workspace` flag.

To select a specific package to test, you can use the `--package` flag the same way as in regular workspaces or run `snforge test` from the package directory.

How Tests Are Collected

Snforge executes tests, but it does not compile them directly. Instead, it compiles tests by internally running `scarb build --test` command.

The `snforge_scarb_plugin` dependency, which is included with `snforge_std` dependency makes all functions marked with `#[test]` executable and indicates to Scarb they should be compiled. Without the plugin, no snforge tests can be compiled, that's why `snforge_std` dependency is always required in all snforge projects.

Thanks to that, Scarb collects all functions marked with `#[test]` from valid locations and compiles them into tests that are executed by snforge.

`[[test]]` Target

Under the hood, Scarb utilizes the `[[test]]` target mechanism to compile the tests. More information about the `[[test]]` target is available in the Scarb documentation.

By default, `[[test]]]` target is implicitly configured and user does not have to define it. See Scarb documentation for more details about the mechanism.

Tests Organization

Test can be placed in both `src` and `test` directories. When adding tests to files in `src` you must wrap them in tests module.

You can read more about tests organization in Scarb documentation.

Unit Tests

Test placed in `src` directory are often called unit tests. For these test to function in snforge, they must be wrapped in a module marked with `#[cfg(test)]` attribute.

```
// src/example.rs
// ...

// This test is not in module marked with `#[cfg(test)]` so it won't work
#[test]
fn my_invalid_test() {
    // ...
}

#[cfg(test)]
mod tests {
    // This test is in module marked with `#[cfg(test)]` so it will work
    #[test]
    fn my_test() {
        // ..
    }
}
```

Integration Tests

Integration tests are placed in `tests` directory. This directory is a special directory in Scarb. Tests do not have to be wrapped in `#[cfg(test)]` and each file is treated as a separate module.

```
// tests/example.rs
// ...

// This test is in `tests` directory
// so it works without being in module with `#[cfg(test)]`
#[test]
fn my_test_1() {
    // ..
}
```

Modules and `lib.cairo`

As written above, each file in `tests` directory is treated as a separate module

```
$ tree
```

▼ Output:

```
tests/
├── module1.cairo <-- is collected
├── module2.cairo <-- is collected
└── module3.cairo <-- is collected
```

Scarb will collect each file and compile it as a separate test target. Each of these targets will be run separately by `snforge`.

However, it is also possible to define `lib.cairo` file in `tests`. This stops files in `tests` from being treated as separate modules. Instead, Scarb will only create a single test target for that `lib.cairo` file. Only tests that are reachable from this file will be collected and compiled.

```
$ tree
```

▼ Output:

```
tests/
├── lib.cairo
├── module1.cairo  <-- is collected
├── module2.cairo  <-- is collected
└── module3.cairo  <-- is not collected
```

```
// tests/lib.cairo

mod module1;
mod module2;
```

How Contracts Are Collected

`snforge` supports two mechanisms for collecting contracts used in tests. The default one depends on Scarb version used and can be controlled with `--no-optimization` flag.

- If using Scarb version >= 2.8.3, optimized collection mechanism is used by default.
- If using Scarb version < 2.8.3 or running `snforge test` with `--no-optimization` flag, the old collection mechanism is used.

> 📝 **Note**
>
> Enabling new mechanism **requires** Scarb version >= 2.8.3.

Differences Between Collection Mechanisms

Feature	Old Mechanism	Optimised Mechanism
Using contracts from `/src`	✅	✅
Using contracts from `/tests`	❌	✅
Using contracts from modules marked with `#[cfg(test)]`	❌	✅
Using contracts from dependencies	✅	✅
Contracts more closely resemble ones from real network	✅	❌
Less compilation steps required (faster compilation)	❌	✅
Additional compilation step required (`scarb build`)	✅	❌

How Contracts Are Collected

For the `declare` to work, snforge must collect and call build on contracts in the package. By default, if using Scarb version >= 2.8.3, snforge will combine test collection and contract collection steps.

When running `snforge test`, snforge will, under the hood, call the `scarb build --test` command. This command builds all the test and contracts along them. Snforge collects these contracts and makes them available for declaring in tests.

Contracts are collected from both `src` and `tests` directory, including modules marked with `#[cfg(test)]`. Internally, snforge collects contracts from all `[[test]]` targets compiled by Scarb. You can read more about that in test collection documentation.

Collection Order

When multiple `[[test]]` targets are present, snforge will first try to collect contracts from `integration` `test-type` target. If `integration` is not present, snforge will first collect contracts from the first encountered `[[test]]` target.

After collecting from initial `[[test]]` target, snforge will collect contracts from any other encountered targets. No specific order of collection is guaranteed.

> **Note**
>
> If multiple contracts with the same name are present, snforge will use the first encountered implementation and will not collect others.

Using External Contracts in Tests

To use contract from dependencies in tests, `Scarb.toml` must be updated to include these contracts under `[[target.starknet-contract]]`.

```
[[target.starknet-contract]]
build-external-contracts = ["path::to::Contract1", "other::path::to::Contract2"]
```

For more information about `build-external-contracts`, see Scarb documentation.

How Contracts Are Collected

When you call `snforge test`, one of the things that `snforge` does is that it calls Scarb, particularly `scarb build`. It makes Scarb build all contracts from your package and save them to the `target/{current_profile}` directory (read more on Scarb website).

Then, `snforge` loads compiled contracts from the package your tests are located, allowing you to declare the contracts in tests.

Only contracts from `src/` directory are loaded. Contracts from `/tests` and modules marked with `#[cfg(test)]` are not build or collected. To create contracts to be specifically used in tests see conditional compilation.

> ⚠️ **Warning**
>
> Make sure to define `[[target.starknet-contract]]` section in your `Scarb.toml`, otherwise Scarb won't build your contracts.

Using External Contracts In Tests

If you wish to use contracts from your dependencies inside your tests (e.g. an ERC20 token, an account contract), you must first make Scarb build them. You can do that by using `build-external-contracts` key in `Scarb.toml`, e.g.:

```
[[target.starknet-contract]]
build-external-contracts = ["openzeppelin::account::account::Account"]
```

For more information about `build-external-contracts`, see Scarb documentation.

Gas and VM Resources Estimation

`snforge` supports gas and other VM resources estimation for each individual test case.

It does not calculate the final transaction fee, for details on how fees are calculated, please refer to fee mechanism in Starknet documentation.

Gas Estimation

Single Test

When the test passes with no errors, estimated gas is displayed this way:

```
[PASS] tests::simple_test (gas: ~1)
```

This gas calculation is based on the estimated VM resources (that you can display additionally on demand), deployed contracts, storage updates, events and l1 <> l2 messages.

Fuzzed Tests

While using the fuzzing feature additional gas statistics will be displayed:

```
[PASS] tests::fuzzing_test (runs: 256, gas: {max: ~126, min: ~1, mean: ~65.00, std deviation: ~37.31})
```

> 📝 **Note**
>
> Starknet-Foundry uses blob-based gas calculation formula in order to calculate gas usage. For details on the exact formula, see the docs.

VM Resources estimation

It is possible to enable more detailed breakdown of resources, on which the gas calculations are based on.

Usage

In order to run tests with this feature, run the `test` command with the appropriate flag:

```
$ snforge test --detailed-resources
```

▶ Output:

This displays the resources used by the VM during the test execution.

Analyzing the results

Normally in transaction receipt (or block explorer transaction details), you would see some additional OS resources that starknet-foundry does not include for a test (since it's not a normal transaction per-se):

Not included in the gas/resource estimations

- Fee transfer costs
- Transaction type related resources - in real Starknet additional cost depending on the transaction type (e.g., `Invoke` / `Declare` / `DeployAccount`) is added
- Declaration gas costs (CASM/Sierra bytecode or ABIs)
- Call validation gas costs (if you did not call `__validate__` endpoint explicitly)

Included in the gas/resource estimations

- Cost of syscalls (additional steps or builtins needed for syscalls execution)

Coverage

Coverage reporting allows developers to gain comprehensive insights into how their code is executed. With `cairo-coverage`, you can generate a coverage report that can later be analyzed for detailed coverage statistics.

Prerequisites

`cairo-coverage` relies on debug information provided by Scarb. To generate the necessary debug information, you need to have:

1. Scarb version `2.8.0` or higher
2. `Scarb.toml` file with the following Cairo compiler configuration:

```
[profile.dev.cairo]
unstable-add-statements-code-locations-debug-info = true
unstable-add-statements-functions-debug-info = true
inlining-strategy = "avoid"
```

> **Note**
>
> That `unstable-add-statements-code-locations-debug-info = true` and `unstable-add-statements-functions-debug-info = true` will slow down the compilation and cause it to use more system memory. It will also make the compilation artifacts larger. So it is only recommended to add these flags when you need their functionality.

For more information about these sections, please refer to the Scarb documentation.

Installation and usage

In order to run coverage report with `cairo-coverage` you need to install it first. Please refer to the instructions provided in the README for guidance: https://github.com/software-mansion/cairo-coverage#installation

Usage details and limitations are also described there - make sure to check it out as well.

Integration with cairo-coverage

`snforge` is able to produce a file with a trace for each passing test (excluding fuzz tests). All you have to do is use the `--save-trace-data` flag:

```
$ snforge test --save-trace-data
```

The files with traces will be saved to `snfoundry_trace` directory. Each one of these files can then be used as an input for the cairo-coverage.

If you want `snforge` to call `cairo-coverage` on generated files automatically, use `--coverage` flag:

```
$ snforge test --coverage
```

This will generate a coverage report in the `coverage` directory named `coverage.lcov`.

Passing arguments to `cairo-coverage`

You can pass additional arguments to `cairo-coverage` by using the `--` separator. Everything after `--` will be passed to `cairo-coverage`:

```
$ snforge test --coverage -- --include macros
```

> 📝 **Note**
>
> Running `snforge test --help` won't show info about `cairo-coverage` flags. To see them, run `snforge test --coverage -- --help`.

Coverage report

`cairo-coverage` generates coverage data as an `.lcov` file. A summary report with aggregated data can be produced by one of many tools that accept the `lcov` format. In this example we will use the `genhtml` tool from the lcov package to generate an HTML report.

Run the following command in the directory containing your `coverage.lcov` file:

```
$ genhtml -o coverage_report coverage.lcov
```

You can now open the `index.html` file in the `coverage_report` directory to see the generated coverage report.

Fork Testing

`snforge` supports testing in a forked environment. Each test can fork the state of a specified real network and perform actions on top of it.

> **Note**
>
> Actions are performed on top of the `forked` state which means real network is not affected.

Test Contract

We will demonstrate fork testing on an example of the `Pokemons` contract deployed on Sepolia network. We are going to use a free, open RPC endpoint - Blast.

We first need to define the contract's interface along with all the structures used by its externals:

```
#[derive(Clone, Debug, PartialEq, Drop, Serde, starknet::Store)]
pub struct Pokemon {
    pub name: ByteArray,
    pub element: Element,
    pub likes: felt252,
    pub owner: starknet::ContractAddress
}

#[derive(Copy, Debug, PartialEq, Drop, Serde, starknet::Store)]
pub enum Element {
    Fire,
    Water,
    Grass
}

#[starknet::interface]
pub trait IPokemonGallery<TContractState> {
    fn like(ref self: TContractState, name: ByteArray);
    fn all(self: @TContractState) -> Array<Pokemon>;
    fn pokemon(self: @TContractState, name: ByteArray) -> Option<Pokemon>;
    fn liked(self: @TContractState) -> Array<Pokemon>;
}
```

Fork Configuration

There are two ways of configuring a fork:

- by specifying `url` and block-related parameters in the `#[fork(...)]` attribute
- or by passing a fork name defined in your `Scarb.toml` to the `#[fork(...)]` attribute

Configure a Fork in the Attribute

It is possible to pass `url` and only one of `block_number`, `block_hash`, `block_tag` arguments to the `fork` attribute:

- `url` (string literal) - RPC URL
- `block_number` (hexadecimal number) - number of block which fork will be pinned to
- `block_hash` (hexadecimal number) - hash of block which fork will be pinned to
- `block_tag` (identifier) - tag of block which fork will be pinned to. Currently only `latest` is supported

Once such a configuration is passed, it is possible to use state and contracts defined on the specified network.

We are going to test a very simple scenario:

1. Obtain a dispatcher generated by the `#[starknet::interface]`
2. Instantiate it with a real contract address present in the forked state
3. Call a method modifying the contract's state
4. Make some assertion about the changed state

Example uses of all methods:

1. `block_number`:

```
// import dispatcher generated by the interface we wrote
use fork_testing::{IPokemonGalleryDispatcher, IPokemonGalleryDispatcherTrait};

// take an address of a real network contract
const CONTRACT_ADDRESS: felt252 =
    0x0522dc7cbe288037382a02569af5a4169531053d284193623948eac8dd051716;

#[test]
#[fork(url: "https://starknet-sepolia.public.blastapi.io/rpc/v0_7", block_number:
77864)]
fn test_using_forked_state() {
    // instantiate the dispatcher
    let dispatcher = IPokemonGalleryDispatcher {
        contract_address: CONTRACT_ADDRESS.try_into().unwrap()
    };

    // call the mutating method
    dispatcher.like("Charizard");

    // check if the contract's state has changed
    let pokemon = dispatcher.pokemon("Charizard");

    assert!(pokemon.is_some());
    assert_eq!(pokemon.unwrap().likes, 1);
}
```

2. `block_hash`:

```
use fork_testing::{IPokemonGalleryDispatcher, IPokemonGalleryDispatcherTrait};

const CONTRACT_ADDRESS: felt252 =
    0x0522dc7cbe288037382a02569af5a4169531053d284193623948eac8dd051716;

#[test]
#[fork(
    url: "https://starknet-sepolia.public.blastapi.io/rpc/v0_7",
    block_hash: 0x0690f8d584b52c2798d76b3346217a516778abee9b1bd8e400beb4f05dd9a4e7
)]
fn test_using_forked_state() {
    let dispatcher = IPokemonGalleryDispatcher {
        contract_address: CONTRACT_ADDRESS.try_into().unwrap()
    };

    dispatcher.like("Charizard");
    let pokemon = dispatcher.pokemon("Charizard");

    assert!(pokemon.is_some());
    assert_eq!(pokemon.unwrap().likes, 1);
}
```

3. `block_tag`:

```
use fork_testing::{IPokemonGalleryDispatcher, IPokemonGalleryDispatcherTrait};

const CONTRACT_ADDRESS: felt252 =
    0x0522dc7cbe288037382a02569af5a4169531053d284193623948eac8dd051716;

#[test]
#[fork(url: "https://starknet-sepolia.public.blastapi.io/rpc/v0_7", block_tag:
latest)]
fn test_using_forked_state() {
    let dispatcher = IPokemonGalleryDispatcher {
        contract_address: CONTRACT_ADDRESS.try_into().unwrap()
    };

    dispatcher.like("Charizard");
    let pokemon = dispatcher.pokemon("Charizard");

    assert!(pokemon.is_some());
    assert_eq!(pokemon.unwrap().likes, 1);
}
```

Configure Fork in `Scarb.toml`

Although passing named arguments works fine, you have to copy-paste it each time you want to use the same fork in tests.

`snforge` solves this issue by allowing fork configuration inside the `Scarb.toml` file.

```
[[tool.snforge.fork]]
name = "SEPOLIA_LATEST"
url = "https://starknet-sepolia.public.blastapi.io/rpc/v0_7"
block_id.tag = "latest"
```

From this moment forks can be set using their name in the `fork` attribute.

```
use fork_testing::{IPokemonGalleryDispatcher, IPokemonGalleryDispatcherTrait};

const CONTRACT_ADDRESS: felt252 =
    0x0522dc7cbe288037382a02569af5a4169531053d284193623948eac8dd051716;

#[test]
#[fork("SEPOLIA_LATEST")]
fn test_using_forked_state() {
    let dispatcher = IPokemonGalleryDispatcher {
        contract_address: CONTRACT_ADDRESS.try_into().unwrap()
    };

    dispatcher.like("Charizard");
    let pokemon = dispatcher.pokemon("Charizard");

    assert!(pokemon.is_some());
    assert_eq!(pokemon.unwrap().likes, 1);
}
```

In some cases you may want to override `block_id` defined in the `Scarb.toml` file. You can do it by passing `block_number`, `block_hash`, `block_tag` arguments to the `fork` attribute.

```
use fork_testing::{IPokemonGalleryDispatcher, IPokemonGalleryDispatcherTrait};

const CONTRACT_ADDRESS: felt252 =
    0x0522dc7cbe288037382a02569af5a4169531053d284193623948eac8dd051716;

#[test]
#[fork("SEPOLIA_LATEST", block_number: 200000)]
fn test_using_forked_state() {
    let dispatcher = IPokemonGalleryDispatcher {
        contract_address: CONTRACT_ADDRESS.try_into().unwrap()
    };

    dispatcher.like("Charizard");
    let pokemon = dispatcher.pokemon("Charizard");

    assert!(pokemon.is_some());
    assert_eq!(pokemon.unwrap().likes, 1);
}
```

Testing Forked Contracts

Once the fork is configured, the test will run on top of the forked state, meaning that it will have access to every contract deployed on the real network.

With that, you can now interact with any contract from the chain the same way you would in a standard test.

> ⚠️ **Warning**
>
> Some cheats aren't supported and won't work for forked contracts written in **Cairo 0**.
> Only those cheats are going to have an effect:
>
> - `caller_address`
> - `block_number`
> - `block_timestamp`
> - `sequencer_address`
> - `spy_events`
> - `spy_messages_to_l1`

Fuzz Testing

In many cases, a test needs to verify function behavior for multiple possible values. While it is possible to come up with these cases on your own, it is often impractical, especially when you want to test against a large number of possible arguments.

ℹ️ **Info** Currently, `snforge` fuzzer only supports using randomly generated values. This way of fuzzing doesn't support any kind of value generation based on code analysis, test coverage or results of other fuzzer runs. In the future, more advanced fuzzing execution modes will be added.

Random Fuzzing

To convert a test to a random fuzz test, simply add arguments to the test function. These arguments can then be used in the test body. The test will be run many times against different randomly generated values.

```
fn sum(a: felt252, b: felt252) -> felt252 {
    return a + b;
}

#[cfg(test)]
mod tests {
    use super::sum;

    #[test]
    fn test_sum(x: felt252, y: felt252) {
        assert_eq!(sum(x, y), x + y);
    }
}
```

Then run `snforge test` like usual.

```
$ snforge test
```

▶ Output:

Types Supported by the Fuzzer

Fuzzer currently supports generating values of these types

- `u8`
- `u16`
- `u32`
- `u64`
- `u128`
- `u256`
- `felt252`

Trying to use arguments of different type in test definition will result in an error.

Fuzzer Configuration

It is possible to configure the number of runs of the random fuzzer as well as its seed for a specific test case:

```
fn sum(a: felt252, b: felt252) -> felt252 {
    return a + b;
}

#[cfg(test)]
mod tests {
    use super::sum;

    #[test]
    #[fuzzer(runs: 22, seed: 38)]
    fn test_sum(x: felt252, y: felt252) {
        assert_eq!(sum(x, y), x + y);
    }
}
```

It can also be configured globally, via command line arguments:

```
$ snforge test --fuzzer-runs 1234 --fuzzer-seed 1111
```

Or in `Scarb.toml` file:

```
# ...
[tool.snforge]
fuzzer_runs = 1234
fuzzer_seed = 1111
# ...
```

Conditional Compilation

> 📝 **Note** For more detailed guide on Scarb conditional compilation, please refer to Scarb documentation

It is possible to build some contracts solely for testing purposes. This can be achieved by leveraging Scarb features. Configuration in `Scarb.toml` is done in the same manner as described in the Scarb documentation. Additionally, for utilizing features the `snforge test` command exposes the following flags, aligned with `scarb` flags: --features, --all-features and --no-default-features.

Contracts

Firstly, define a contract in the `src` directory with a `#[cfg(feature: '<FEATURE_NAME>')]` attribute:

```
pub mod contract;
// pub mod function;  // TODO: include this module (issue #2515)

mod dummy {} // trick `scarb fmt -c`
```

> 📝 **Note** To declare mock contracts in tests, these contracts should be defined within the package and not in the `tests` directory. This requirement is due to the way snforge collects contracts.

Next, create a test that uses the above contract:

```
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait};

use conditional_compilation::contract::{IMockContractDispatcher,
IMockContractDispatcherTrait};

#[test]
fn test_mock_contract() {
    let (contract_address, _) = declare("MockContract")
        .unwrap()
        .contract_class()
        .deploy(@array![])
        .unwrap();

    let dispatcher = IMockContractDispatcher { contract_address };
    let response = dispatcher.response();

    assert_eq!(response, 1);
}
```

The `Scarb.toml` file needs to be updated so it includes the following lines:

```
[features]
enable_for_tests = []
```

Then, to use the contract in tests `snforge test` must be provided with a flag defined above:

```
$ snforge test --features enable_for_tests
```

Also, we can specify which features are going to be enabled by default:

```
[features]
default = ["enable_for_tests"]
enable_for_tests = []
```

📝 **Note** If `snforge test` is run without the above feature enabled, it won't build any artifacts for the `MockContract` and all tests that use this contract will fail.

Functions

Features are not limited to conditionally compiling contracts and can be used with other parts of the code, like functions:

```
#[cfg(feature: 'enable_for_tests')]
fn foo() -> u32 {
    2
}

#[cfg(feature: 'enable_for_tests')]
#[cfg(test)]
mod tests {
    #[test]
    fn test_using_conditionally_compiled_function() {
        assert_eq!(foo(), 2);
    }
}
```

Direct Storage Access

In some instances, it's not possible for contracts to expose API that we'd like to use in order to initialize the contracts before running some tests. For those cases `snforge` exposes storage-related cheatcodes, which allow manipulating the storage directly (reading and writing).

In order to obtain the variable address that you'd like to write to, or read from, you need to use either:

- `selector!` macro - if the variable is not a mapping
- `map_entry_address` function in tandem with `selector!` - for key-value pair of a map variable
- `starknet::storage_access::storage_address_from_base`

Example: Felt-only storage

This example uses only felts for simplicity.

1. Exact storage fields

```
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait, store, load,
map_entry_address};

#[test]
fn test_store_and_load_plain_felt() {
    let (contract_address, _) = declare("SimpleStorageContract")
        .unwrap()
        .contract_class()
        .deploy(@array![])
        .unwrap();

    // load existing value from storage
    let loaded = load(
        contract_address, // an existing contract which owns the storage
        selector!("plain_felt"), // field marking the start of the memory chunk
being read from
        1 // length of the memory chunk (seen as an array of felts) to read
    );

    assert_eq!(loaded, array![0x2137_felt252]);

    // overwrite it with a new value
    store(
        contract_address, // storage owner
        selector!("plain_felt"), // field marking the start of the memory chunk
being written to
        array![420].span() // array of felts to write
    );

    // load again and check if it changed
    let loaded = load(contract_address, selector!("plain_felt"), 1);
    assert_eq!(loaded, array![420]);
}
```

2. Map entries

```
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait, store, load,
map_entry_address};

#[test]
fn test_store_and_load_map_entries() {
    let (contract_address, _) = declare("SimpleStorageContract")
        .unwrap()
        .contract_class()
        .deploy(@array![])
        .unwrap();

    // load an existing map entry
    let loaded = load(
        contract_address,
        map_entry_address(
            selector!("mapping"), // start of the read memory chunk
            array!['some_key'].span(), // map key
        ),
        1, // length of the read memory chunk
    );

    assert_eq!(loaded, array!['some_value']);

    // write other value in place of the previous one
    store(
        contract_address,
        map_entry_address(
            selector!("mapping"), // storage variable name
             array!['some_key'].span(), // map key
        ),
        array!['some_other_value'].span()
    );

    let loaded = load(
        contract_address,
        map_entry_address(
            selector!("mapping"), // start of the read memory chunk
            array!['some_key'].span(), // map key
        ),
        1, // length of the read memory chunk
    );

    assert_eq!(loaded, array!['some_other_value']);

    // load value written under non-existing key
    let loaded = load(
        contract_address,
        map_entry_address(selector!("mapping"), array!
['non_existing_field'].span(),),
        1,
    );
```

```
    assert_eq!(loaded, array![0]);
}
```

Example: Complex structures in storage

This example uses a complex key and value, with default derived serialization methods (via `#[derive(starknet::Store)]`).

We use a contract along with helper structs:

```
use core::hash::LegacyHash;

// Required for lookup of complex_mapping values
// This is consistent with `map_entry_address`, which uses pedersen hashing of
keys
impl StructuredKeyHash of LegacyHash<MapKey> {
    fn hash(state: felt252, value: MapKey) -> felt252 {
        let state = LegacyHash::<felt252>::hash(state, value.a);
        LegacyHash::<felt252>::hash(state, value.b)
    }
}

#[derive(Copy, Drop, Serde)]
pub struct MapKey {
    pub a: felt252,
    pub b: felt252,
}

// Serialization of keys and values with `Serde` to make usage of
`map_entry_address` easier
impl MapKeyIntoSpan of Into<MapKey, Span<felt252>> {
    fn into(self: MapKey) -> Span<felt252> {
        let mut serialized_struct: Array<felt252> = array![];
        self.serialize(ref serialized_struct);
        serialized_struct.span()
    }
}

#[derive(Copy, Drop, Serde, starknet::Store)]
pub struct MapValue {
    pub a: felt252,
    pub b: felt252,
}

impl MapValueIntoSpan of Into<MapValue, Span<felt252>> {
    fn into(self: MapValue) -> Span<felt252> {
        let mut serialized_struct: Array<felt252> = array![];
        self.serialize(ref serialized_struct);
        serialized_struct.span()
    }
}

#[starknet::interface]
pub trait IComplexStorageContract<TContractState> {}

#[starknet::contract]
mod ComplexStorageContract {
    use starknet::storage::Map;
    use super::{MapKey, MapValue};

    #[storage]
    struct Storage {
        complex_mapping: Map<MapKey, MapValue>,
```

```
    }
}
```

And perform a test checking `load` and `store` behavior in context of those structs:

```
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait, store, load,
map_entry_address};

use direct_storage_access::complex_structures::{MapKey, MapValue};

#[test]
fn store_in_complex_mapping() {
    let (contract_address, _) = declare("ComplexStorageContract")
        .unwrap()
        .contract_class()
        .deploy(@array![])
        .unwrap();

    let k = MapKey { a: 111, b: 222 };
    let v = MapValue { a: 123, b: 456 };

    store(
        contract_address,
        map_entry_address( // uses Pedersen hashing under the hood for address
calculation
            selector!("mapping"), // storage variable name
             k.into() // map key
        ),
        v.into()
    );

    // complex_mapping = {
    //     hash(k): 123,
    //     hash(k) + 1: 456
    //     ...
    // }

    let loaded = load(contract_address, map_entry_address(selector!("mapping"),
k.into()), 2,);

    assert_eq!(loaded, array![123, 456]);
}
```

⚠ Warning

Complex data can often times be packed in a custom manner (see this pattern) to optimize costs. If that's the case for your contract, make sure to handle deserialization properly - standard methods might not work. **Use those cheatcode as a last-resort, for cases that cannot be handled via contract's API!**

Example: Using enums in storage

Enums use 0-based layout for serialization. For example, `FirstVariantOfSomeEnum(100)` will be serialized as `[0, 100]`. However, their Starknet storage layout is 1-based for most enums, especially for these with derived `Store` trait implementation. Therefore, `FirstVariantOfSomeEnum(100)` will be stored on Starknet as `[1, 100]`.

Remember that this rule may not hold for enums that with manual `Store` trait implementation. The most notable example is `Option`, e.g. `Option::None` will be stored as `[0]` and `Option::Some(100)` will be stored as `[1, 100]`.

Below is an example of a contract which can store `Option<u256>` values:

```
#[starknet::interface]
pub trait IEnumsStorageContract<TContractState> {
    fn read_value(self: @TContractState, key: u256) -> Option<u256>;
}

#[starknet::contract]
pub mod EnumsStorageContract {
    use starknet::{storage::{StoragePointerWriteAccess, StoragePathEntry, Map}};

    #[storage]
    struct Storage {
        example_storage: Map<u256, Option<u256>>,
    }

    #[abi(embed_v0)]
    impl EnumsStorageContractImpl of super::IEnumsStorageContract<ContractState> {
        fn read_value(self: @ContractState, key: u256) -> Option<u256> {
            self.example_storage.entry(key).read()
        }
    }
}
```

And a test which uses `store` and reads the value:

```
use direct_storage_access::using_enums::IEnumsStorageContractSafeDispatcherTrait;
use direct_storage_access::using_enums::IEnumsStorageContractSafeDispatcher;
use starknet::ContractAddress;
use snforge_std::{declare, ContractClassTrait, DeclareResultTrait, store,
map_entry_address, load};

fn deploy_contract(name: ByteArray) -> ContractAddress {
    let contract = declare(name).unwrap().contract_class();
    let (contract_address, _) = contract.deploy(@ArrayTrait::new()).unwrap();
    contract_address
}


#[test]
fn test_store_and_read() {
    let contract_address = deploy_contract("EnumsStorageContract");
    let safe_dispatcher = IEnumsStorageContractSafeDispatcher { contract_address
};

    let mut keys = ArrayTrait::new();
    let key: u256 = 1;
    key.serialize(ref keys);

    let value: Option<u256> = Option::Some((100));
    let felt_value: felt252 = value.unwrap().try_into().unwrap();

    // Serialize Option enum according to its 1-based storage layout
    let serialized_value = array![1, felt_value];

    let storage_address = map_entry_address(selector!("example_storage"),
keys.span());

    store(
        target: contract_address,
        storage_address: storage_address,
        serialized_value: serialized_value.span(),
    );

    let read_value = safe_dispatcher.read_value(key).expect('Failed to read
value');

    assert_eq!(read_value, value);
}
```

```
snforge test test_store_and_read
```

▶ Output:

📝 Note

The `load` cheatcode will return zeros for memory you haven't written into yet (it is a default storage value for Starknet contracts' storage).

Example with `storage_address_from_base`

This example uses `storage_address_from_base` with entry's of the storage variable.

```
use starknet::storage::StorageAsPointer;
use starknet::storage::StoragePathEntry;

use snforge_std::{declare, ContractClassTrait, DeclareResultTrait, store, load};
use starknet::storage_access::{storage_address_from_base};

use direct_storage_access::felts_only::{
    SimpleStorageContract, ISimpleStorageContractDispatcher,
ISimpleStorageContractDispatcherTrait
};

#[test]
fn update_mapping() {
    let key = 0;
    let data = 100;
    let (contract_address, _) = declare("SimpleStorageContract")
        .unwrap()
        .contract_class()
        .deploy(@array![])
        .unwrap();
    let dispatcher = ISimpleStorageContractDispatcher { contract_address };
    let mut state = SimpleStorageContract::contract_state_for_testing();

    let storage_address = storage_address_from_base(
        state.mapping.entry(key).as_ptr().__storage_pointer_address__.into()
    );
    let storage_value: Span<felt252> = array![data.into()].span();
    store(contract_address, storage_address.into(), storage_value);

    let read_data = dispatcher.get_value(key.into());
    assert_eq!(read_data, data, "Storage update failed")
}
```

Profiling

Profiling is what allows developers to get more insight into how the transaction is executed. You can inspect the call tree, see how many resources are used for different parts of the execution, and more!

Integration with cairo-profiler

`snforge` is able to produce a file with a trace for each passing test (excluding fuzz tests). All you have to do is use the `--save-trace-data` flag:

```
$ snforge test --save-trace-data
```

The files with traces will be saved to `snfoundry_trace` directory. Each one of these files can then be used as an input for the cairo-profiler.

If you want `snforge` to call `cairo-profiler` on generated files automatically, use `--build-profile` flag:

```
$ snforge test --build-profile
```

The files with profiling data will be saved to `profile` directory.

Passing arguments to `cairo-profiler`

You can pass additional arguments to `cairo-profiler` by using the `--` separator. Everything after `--` will be passed to `cairo-profiler`:

```
$ snforge test --build-profile -- --show-inlined-functions
```

Note

Running `snforge test --help` won't show info about `cairo-profiler` flags. To see them, run `snforge test --build-profile -- --help`.

Backtrace

Prerequisites

Backtrace feature relies on debug information provided by Scarb. To generate the necessary debug information, you need to have:

1. Scarb version `2.8.0` or higher
2. `Scarb.toml` file with the following Cairo compiler configuration:

```
[profile.dev.cairo]
unstable-add-statements-code-locations-debug-info = true
unstable-add-statements-functions-debug-info = true
```

> **Note**
>
> That `unstable-add-statements-code-locations-debug-info = true` and `unstable-add-statements-functions-debug-info = true` will slow down the compilation and cause it to use more system memory. It will also make the compilation artifacts larger. So it is only recommended to add these flags when you need their functionality.

Usage

> **Note**
>
> Currently, only the last line of failure in each contract is guaranteed to appear in the backtrace. The complete call tree is not fully supported yet; however, in most cases, it will be available. It internally relies on the inlining behavior of the compiler, and a full backtrace is available if all functions are inlined. To obtain a more detailed backtrace, ensure that your inlining strategy in `Scarb.toml` is set to `default`.

When a contract call fails, the error message alone may not always provide enough information to identify the root cause of the issue. To aid in debugging, `snforge` offers a feature that can generate a backtrace of the execution.

If your contract fails and a backtrace can be generated, `snforge` will prompt you to run the operation again with the `SNFORGE_BACKTRACE=1` environment variable (if it's not already configured). For example, you may see failure data like this:

```
$ snforge test
```

▶ Output:

To enable backtraces, simply set the `SNFORGE_BACKTRACE=1` environment variable and rerun the operation.

When enabled, the backtrace will display the call tree of the execution, including the specific line numbers in the contracts where the errors occurred. Here's an example of what you might see:

```
$ SNFORGE_BACKTRACE=1 snforge test
```

▶ Output:

`sncast` Overview

Starknet Foundry `sncast` is a command line tool for performing Starknet RPC calls. With it, you can easily interact with Starknet contracts!

> 💡 **Info** At the moment, `sncast` only supports contracts written in Cairo v1 and v2.

> ⚠️ **Warning** Currently, support is only provided for accounts that use the default signature based on the Stark curve.

How to Use `sncast`

To use `sncast`, run the `sncast` command followed by a subcommand (see available commands):

```
$ sncast <subcommand>
```

If `snfoundry.toml` is present and configured with `[sncast.default]`, `url`, `accounts-file` and `account` name will be taken from it. You can, however, overwrite their values by supplying them as flags directly to `sncast` cli.

> 💡 **Info** Some transactions (like declaring, deploying or invoking) require paying a fee, and they must be signed.

Examples

General Example

Let's use `sncast` to call a contract's function:

```
$ sncast call \
    --network sepolia \
    --contract-address
0x522dc7cbe288037382a02569af5a4169531053d284193623948eac8dd051716 \
    --function "pokemon" \
    --arguments '"Charizard"' \
    --block-id latest
```

▶ Output:

📝 **Note** In the above example we supply `sncast` with `--account` flag. If `snfoundry.toml` is present, and have this property set, value provided using this flags will override value from `snfoundry.toml`. Learn more about `snfoundry.toml` configuration here.

Network and RPC Providers

When providing `--network` flag, `sncast` will randomly select on of the free RPC providers. When using free provider you may experience rate limits and other unexpected behavior.

If using `sncast` extensively, we recommend getting access to a dedicated RPC node and providing its URL to sncast with `--url` flag.

Arguments

Some `sncast` commands (namely `call`, `deploy` and `invoke`) allow passing arguments to perform an action with on the blockchain.

Under the hood cast always send request with serialized form of arguments, but it can be passed in human-readable form thanks to the calldata transformation feature present in Cast.

In the example above we called a function with a deserialized argument: `'"Charizard"'`, passed using `--arguments` flag.

⚠️ **Warning** Cast will not verify the serialized calldata. Any errors caused by passing improper calldata in a serialized form will originate from the network. Basic static analysis is possible only when passing expressions - see calldata transformation.

Using Serialized Calldata

The same result can be achieved by passing serialized calldata, which is a list of hexadecimal-encoded field elements.

For example, this is equivalent to using the `--calldata` option with the following value: `0x0 0x43686172697a617264 0x9`.

To obtain the serialized form of the wished data, you can write a Cairo program that calls `Serde::serialize` on subsequent arguments and displays the results.

Read more about it in the Cairo documentation.

How to Use `--wait` Flag

Let's invoke a transaction and wait for it to be `ACCEPTED_ON_L2`.

```
$ sncast --account my_account \
    --wait \
    deploy \
    --network sepolia \
    --class-hash
0x0227f52a4d2138816edf8231980d5f9e6e0c8a3deab45b601a1fcee3d4427b02 \
```

▶ Output:

As you can see command waited for the transaction until it was `ACCEPTED_ON_L2`.

After setting up the `--wait` flag, command waits 60 seconds for a transaction to be received and (another not specified amount of time) to be included in the block.

Note By default, all commands don't wait for transactions.

Creating And Deploying Accounts

Account is required to perform interactions with Starknet (only calls can be done without it). Starknet Foundry `sncast` supports entire account management flow with the `sncast account create` and `sncast account deploy` commands.

Difference between those two commands is that the first one creates account information (private key, address and more) and the second one deploys it to the network. After deployment, account can be used to interact with Starknet.

To remove an account from the accounts file, you can use `sncast account delete`. Please note this only removes the account information stored locally - this will not remove the account from Starknet.

💡 **Info** Accounts creation and deployment is supported for

- OpenZeppelin
- Argent (with guardian set to 0)
- Braavos

Examples

Creating an Account

Do the following to start interacting with the Starknet:

Create account with the `sncast account create` command

```
$ sncast \
    account create \
    --network sepolia \
    --name new_account
```

▶ Output:

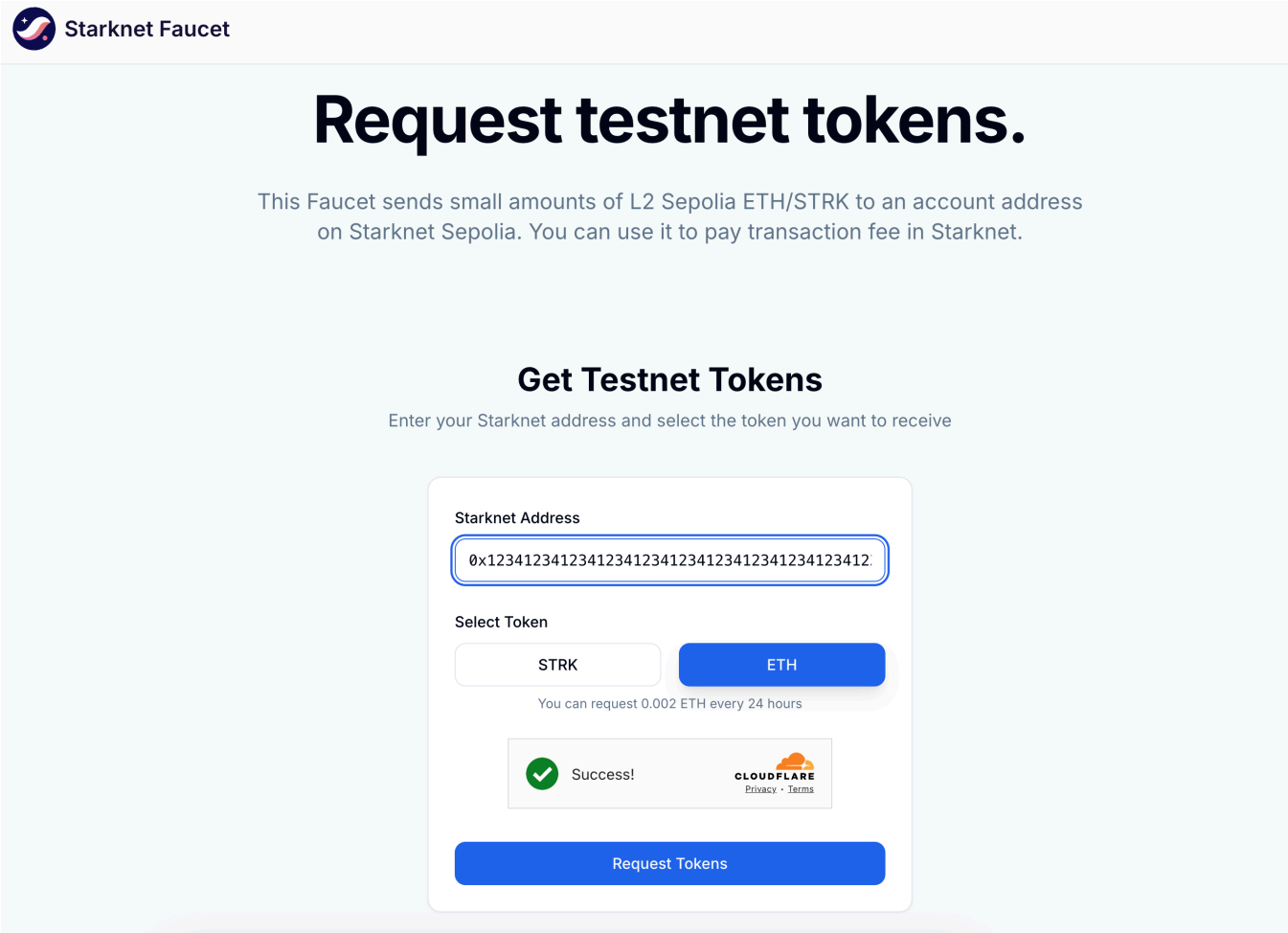
For a detailed CLI description, see account create command reference.

See more advanced use cases below or jump directly to the section here.

Prefund generated address with tokens

To deploy an account in the next step, you need to prefund it with STRK tokens (read more about them here). You can do it both by sending tokens from another starknet account or by bridging them with StarkGate.

💡 **Info** When deploying on a Sepolia test network, you can also fund your account with artificial tokens via the Starknet Faucet



Deploy account with the `sncast account deploy` command

```
$ sncast \
    account deploy \
    --network sepolia \
    --name new_account \
    --max-fee 9999999999999
```

▶ Output:

For a detailed CLI description, see account deploy command reference.

Managing Accounts

If you created an account with `sncast account create` it by default it will be saved in `~/.starknet_accounts/starknet_open_zeppelin_accounts.json` file which we call `default accounts file` in the following sections.

`account import`

To import an account to the `default accounts file`, use the `account import` command.

```
$ sncast \
    account import \
    --network sepolia \
    --name my_imported_account \
    --address 0x3a0bcb72428d8056cc7c2bbe5168ddfc844db2737dda3b4c67ff057691177e1 \
    --private-key 0x2 \
    --type oz
```

`account list`

List all accounts saved in `accounts file`, grouped based on the networks they are defined on.

```
$ sncast account list
```

▶ Output:

You can specify a custom location for the accounts file with the `--accounts-file` or `-f` flag. There is also possibility to show private keys with the `--display-private-keys` or `-p` flag.

`account delete`

Delete an account from `accounts-file` and its associated Scarb profile. If you pass this command, you will be asked to confirm the deletion.

```
$ sncast account delete \
    --name new_account \
    --network-name alpha-sepolia
```

Advanced Use Cases

Custom Account Contract

By default, `sncast` creates/deploys an account using OpenZeppelin's account contract class hash. It is possible to create an account using custom openzeppelin, argent or braavos contract declared to starknet. This can be achieved with `--class-hash` flag:

```
$ sncast \
    account create \
    --name new_account_2 \
    --network sepolia \
    --class-hash
0x00e2eb8f5672af4e6a4e8a8f1b44989685e668489b0a25437733756c5a34a1d6
    --type oz
```

`account create` With Salt Argument

Instead of random generation, salt can be specified with `--salt`.

```
$ sncast \
    account create \
    --network sepolia \
    --name another_account_3 \
    --salt 0x1
```

Additional features provided with `account import/create`

Specifying `--accounts-file`

Account information such as `private_key`, `class_hash`, `address` etc. will be saved to the file specified by `--accounts-file` argument, if not provided, the `default accounts file` will be used.

Specifying `--add-profile`

When the `--add-profile` flag is used, the profile is automatically created for the account. Simply use the `--profile` argument followed by the account name in subsequent requests.

Using Keystore and Starkli Account

Accounts created and deployed with starkli can be used by specifying the `--keystore` argument.

💡 **Info** When passing the `--keystore` argument, `--account` argument must be a path to the starkli account JSON file.

```
$ sncast \
    --keystore keystore.json \
    --account account.json  \
    declare \
    --network sepolia \
    --contract-name my_contract \
```

Creating an Account With Starkli-Style Keystore

It is possible to create an openzeppelin account with keystore in a similar way starkli does.

```
$ sncast \
    --keystore my_key.json \
    --account my_account.json \
    account create \
    --network sepolia
```

The command above will generate a keystore file containing the private key, as well as an account file containing the openzeppelin account info that can later be used with starkli.

Importing Accounts

You can export your private key from wallet (Argent, Braavos) and import it into the file holding the accounts info (`~/.starknet_accounts/starknet_open_zeppelin_accounts.json` by default).

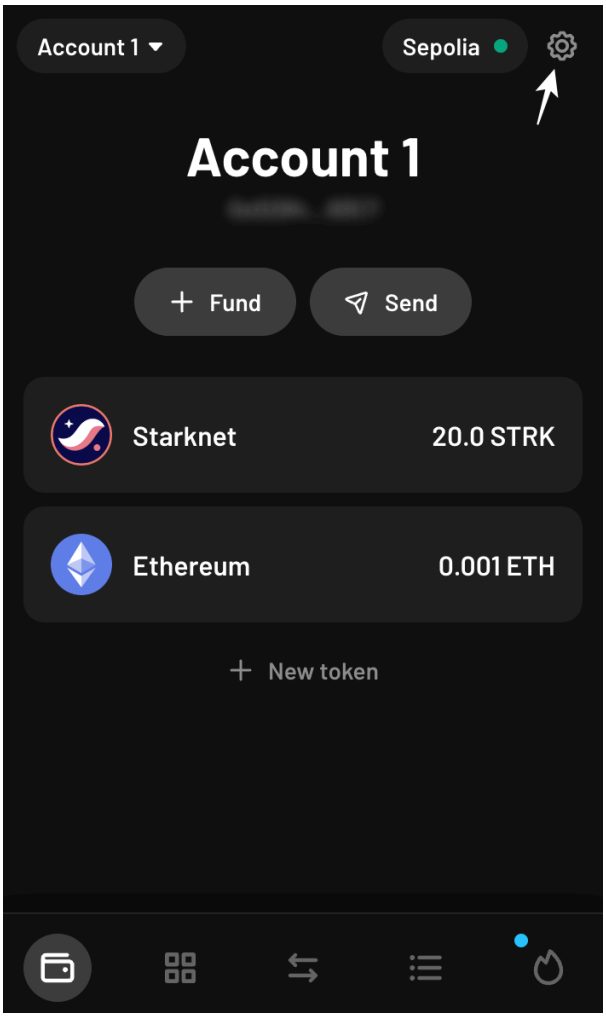
Exporting Your Private Key

This section shows how to export your private key from specific wallets.

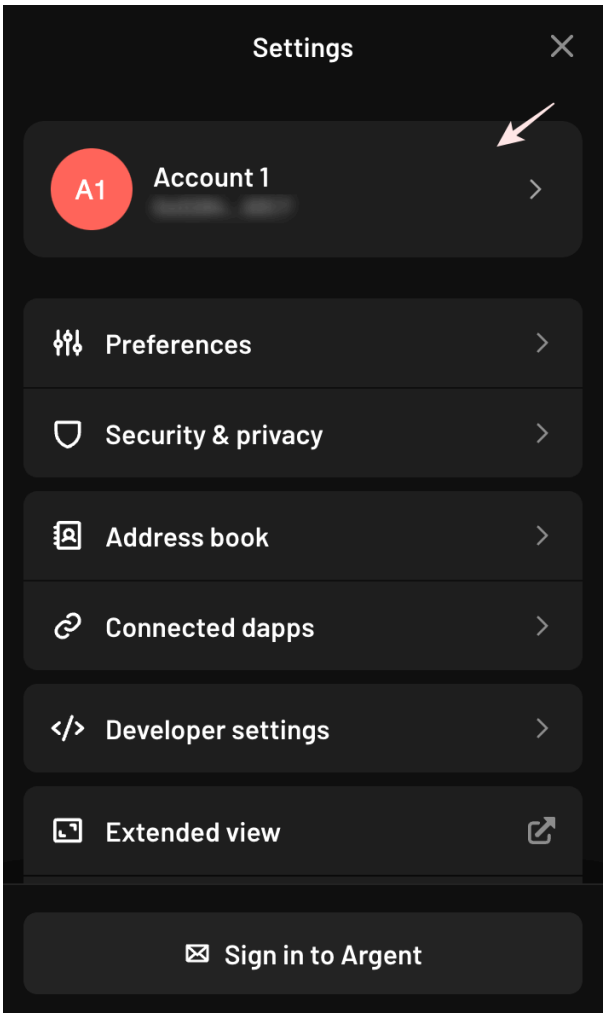
Examples

Argent

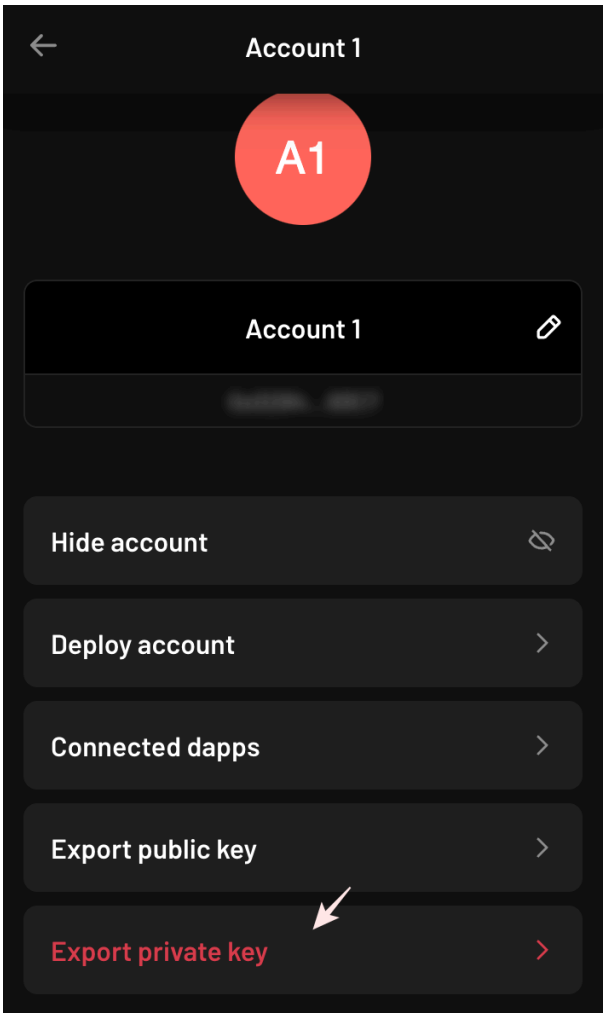
1. Open the Argent app > Settings.



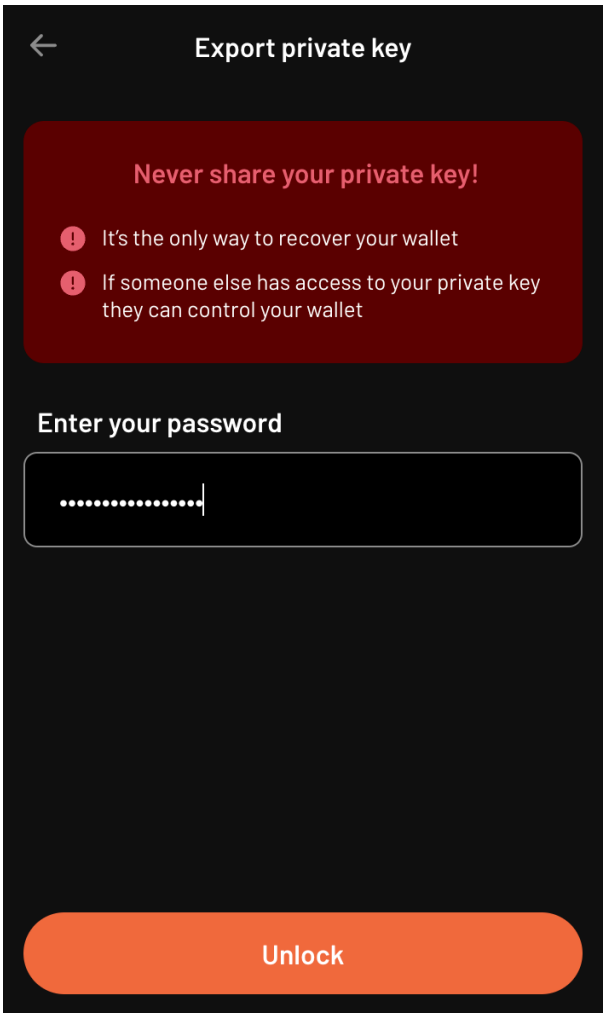
2. Click on the current account.



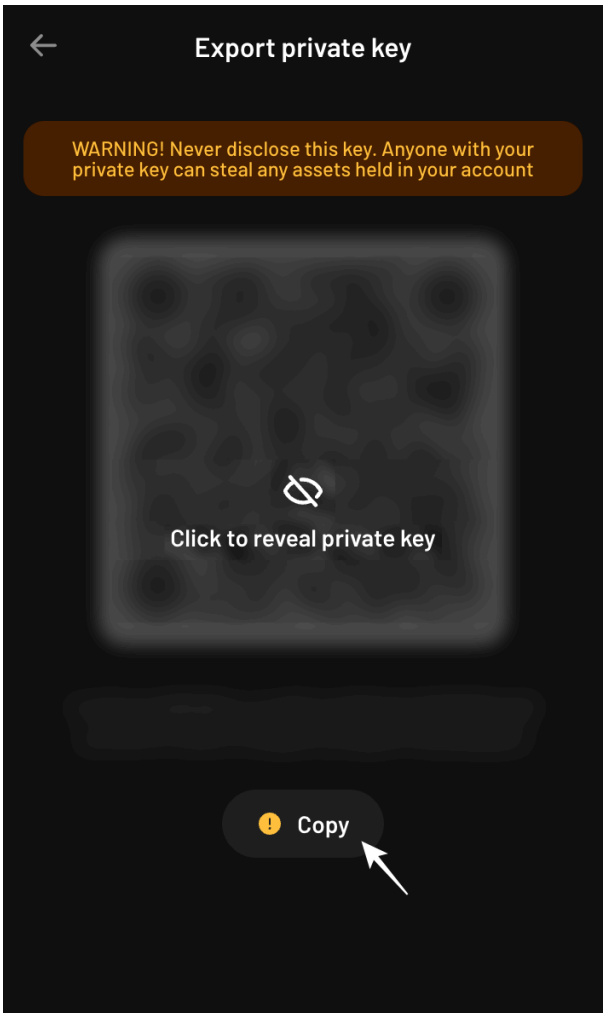
3. Click on "Export private key".



4. Enter your password.

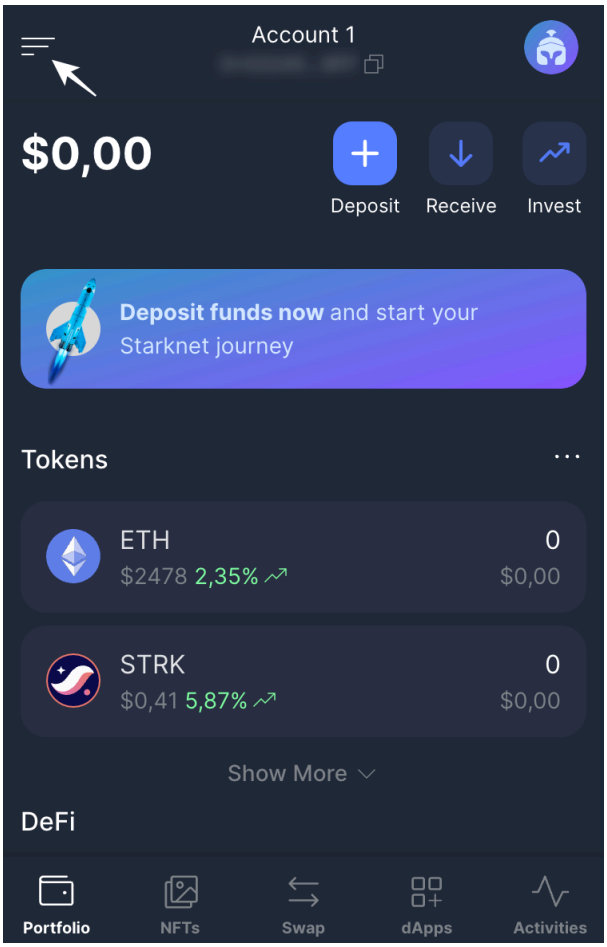


5. Copy your private key.

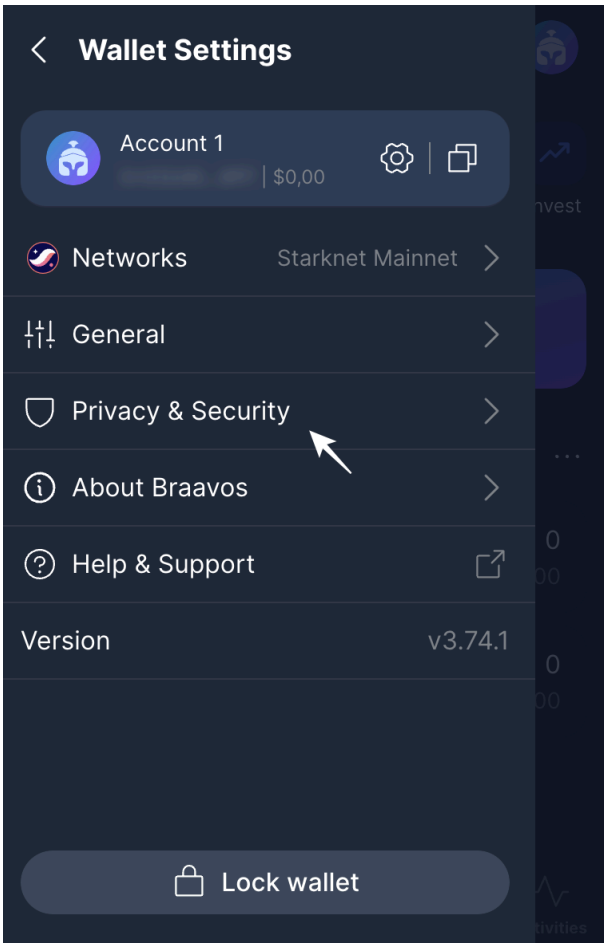


Braavos

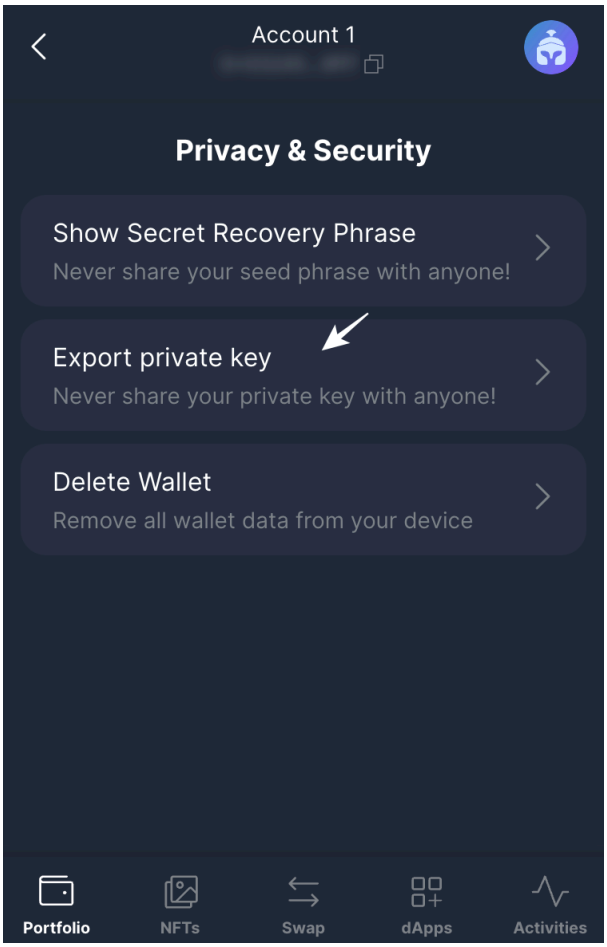
1. Open the Braavos app > Wallet settings.



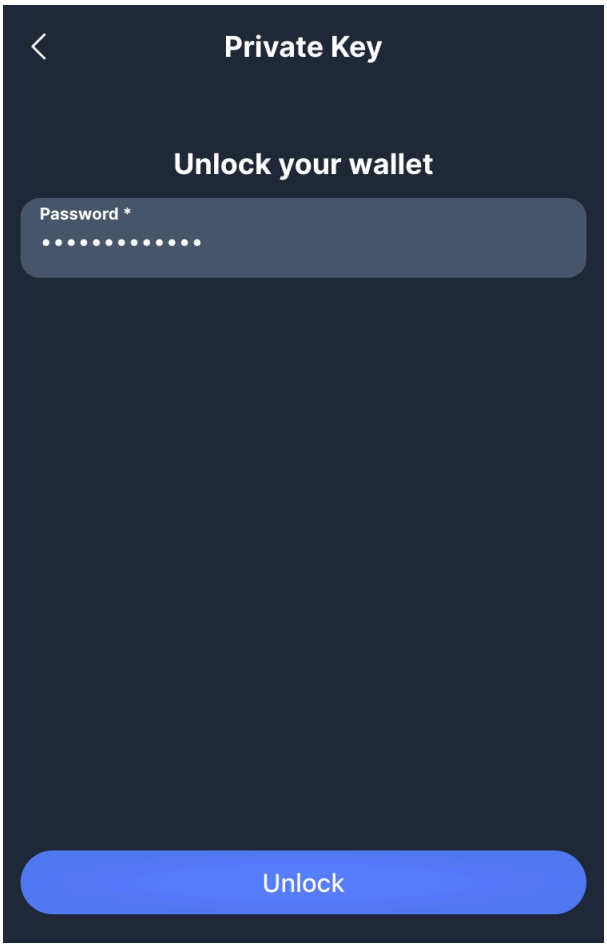
2. Click on "Privacy & Security".



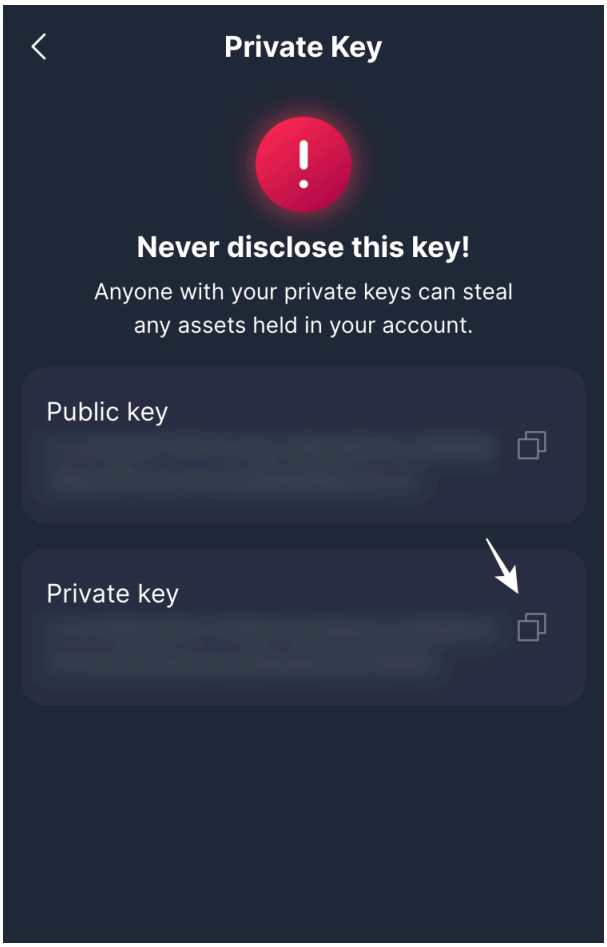
3. Click on "Export private key".



4. Enter your password.



5. Copy your private key.



Importing an Account

Examples

General Example

To import an account into the file holding the accounts info (`~/.starknet_accounts/starknet_open_zeppelin_accounts.json` by default), use the `account import` command.

```
$ sncast \
    account import \
    --network sepolia \
    --name account_123 \
    --address 0x1 \
    --private-key 0x2 \
    --type oz
```

> 📝 **Note** The `--name` can be omitted as this is optional. A default name will be generated for the account.

Passing Private Key in an Interactive

If you don't want to pass the private key in the command (because of safety aspect), you can skip `--private-key` flag. You will be prompted to enter the private key in interactive mode.

```
$ sncast \
    account import \
    --network sepolia \
    --name account_123 \
    --address 0x1 \
    --type oz
```

▶ Output:

Argent

To import Argent account, set the `--type` flag to `argent`.

```
$ sncast \
    account import \
    --network sepolia \
    --name account_argent \
    --address 0x1 \
    --private-key 0x2 \
    --type argent
```

Braavos

To import Braavos account, set the `--type` flag to `braavos`.

```
$ sncast \
    account import \
    --network sepolia \
    --name account_braavos \
    --address 0x1 \
    --private-key 0x2 \
    --type braavos
```

OpenZeppelin

To import OpenZeppelin account, set the `--type` flag to `oz` or `open_zeppelin`.

```
$ sncast \
    account import \
    --network sepolia \
    --name account_oz \
    --address 0x1 \
    --private-key 0x2 \
    --type oz
```

Declaring New Contracts

Starknet provides a distinction between contract class and instance. This is similar to the difference between writing the code of a `class MyClass {}` and creating a new instance of it `let myInstance = MyClass()` in object-oriented programming languages.

Declaring a contract is a necessary step to have your contract available on the network. Once a contract is declared, it then can be deployed and then interacted with.

For a detailed CLI description, see declare command reference.

Examples

General Example

> 📝 **Note** Building a contract before running `declare` is not required. Starknet Foundry `sncast` builds a contract during declaration under the hood using Scarb.

First make sure that you have created a `Scarb.toml` file for your contract (it should be present in project directory or one of its parent directories).

Then run:

```
$ sncast --account my_account \
    declare \
    --network sepolia \
    --contract-name HelloSncast
```

▶ Output:

> 📝 **Note** Contract name is a part after the `mod` keyword in your contract file. It may differ from package name defined in `Scarb.toml` file.

📝 **Note** In the above example we supply `sncast` with `--account` and `--network` flags. If `snfoundry.toml` is present, and has the properties set, values provided using these flags will override values from `snfoundry.toml`. Learn more about `snfoundry.toml` configuration here.

💡 **Info** Max fee will be automatically computed if `--max-fee <MAX_FEE>` is not passed.

Deploying New Contracts

Overview

Starknet Foundry `sncast` supports deploying smart contracts to a given network with the `sncast deploy` command.

It works by invoking a Universal Deployer Contract, which deploys the contract with the given class hash and constructor arguments.

For detailed CLI description, see deploy command reference.

Usage Examples

General Example

After declaring your contract, you can deploy it the following way:

```
$ sncast \
    --account my_account \
    deploy \
    --network sepolia \
    --class-hash
0x0227f52a4d2138816edf8231980d5f9e6e0c8a3deab45b601a1fcee3d4427b02
```

▶ Output:

💡 **Info** Max fee will be automatically computed if `--max-fee <MAX_FEE>` is not passed.

Deploying Contract With Constructor

For such a constructor in the declared contract

```
#[constructor]
fn constructor(ref self: ContractState, first: felt252, second: u256) {
    ...
}
```

you have to pass constructor calldata to deploy it.

```
$ sncast deploy \
    --class-hash
0x02e93ad9922ac92f3eed232be8ca2601fe19f843b7af8233a2e722c9975bc4ea \
    --constructor-calldata 0x1 0x2 0x3
```

▶ Output:

📝 **Note** Although the constructor has only two params you have to pass more because u256 is serialized to two felts. It is important to know how types are serialized because all values passed as constructor calldata are interpreted as a field elements (felt252).

Passing `salt` Argument

Salt is a parameter which modifies contract's address, if not passed it will be automatically generated.

```
$ sncast deploy \
    --class-hash
0x0227f52a4d2138816edf8231980d5f9e6e0c8a3deab45b601a1fcee3d4427b02 \
    --salt 0x123
```

▶ Output:

Passing `unique` Argument

Unique is a parameter which modifies contract's salt with the deployer address. It can be passed even if the `salt` argument was not provided.

```
$ sncast deploy \
    --class-hash
0x0227f52a4d2138816edf8231980d5f9e6e0c8a3deab45b601a1fcee3d4427b02 \
    --unique
```

▶ Output:

Invoking Contracts

Overview

Starknet Foundry `sncast` supports invoking smart contracts on a given network with the `sncast invoke` command.

In most cases, you have to provide:

- Contract address
- Function name
- Function arguments

For detailed CLI description, see invoke command reference.

Examples

General Example

```
$ sncast \
  --account my_account \
  invoke \
  --network sepolia \
  --contract-address
0x522dc7cbe288037382a02569af5a4169531053d284193623948eac8dd051716 \
  --function "add" \
  --arguments 'pokemons::model::PokemonData {'\
'name: "Magmar",'\
'element: pokemons::model::Element::Fire'\
'}'
```

▶ Output:

💡 **Info** Max fee will be automatically computed if `--max-fee <MAX_FEE>` is not passed.

Invoking Function Without Arguments

Not every function accepts parameters. Here is how to call it.

```
$ sncast invoke \
  --contract-address
0x0589a8b8bf819b7820cb699ea1f6c409bc012c9b9160106ddc3dacd6a89653cf \
  --function "get_balance"
```

▶ Output:

Calling Contracts

Overview

Starknet Foundry `sncast` supports calling smart contracts on a given network with the `sncast call` command.

The basic inputs that you need for this command are:

- Contract address
- Function name
- Inputs to the function

For a detailed CLI description, see the call command reference.

Examples

General Example

```
$ sncast \
  call \
  --network sepolia \
  --contract-address
0x522dc7cbe288037382a02569af5a4169531053d284193623948eac8dd051716 \
  --function "balance_of" \
  --arguments '0x0554d15a839f0241ba465bb176d231730c01cf89cdcb95fe896c51d4a6f4bb8f'
```

▶ Output:

📝 **Note** Call does not require passing account-connected parameters (`account` and `accounts-file`) because it doesn't create a transaction.

Passing `block-id` Argument

You can call a contract at the specific block by passing `--block-id` argument.

```
$ sncast call \
  --network sepolia \
  --contract-address
0x522dc7cbe288037382a02569af5a4169531053d284193623948eac8dd051716 \
  --function "balance_of" \
  --arguments '0x0554d15a839f0241ba465bb176d231730c01cf89cdcb95fe896c51d4a6f4bb8f'
\
  --block-id 77864
```

▶ Output:

Performing Multicall

Overview

Starknet Foundry `sncast` supports executing multiple deployments or calls with the `sncast multicall run` command.

> 📝 **Note** `sncast multicall run` executes only one transaction containing all the prepared calls. Which means the fee is paid once.

You need to provide a **path** to a `.toml` file with declarations of desired operations that you want to execute.

You can also compose such config `.toml` file with the `sncast multicall new` command.

For a detailed CLI description, see the multicall command reference.

Example

`multicall run` Example

Example file:

```toml
[[call]]
call_type = "deploy"
class_hash = "0x076e94149fc55e7ad9c5fe3b9af570970ae2cf51205f8452f39753e9497fe849"
inputs = []
id = "map_contract"
unique = false

[[call]]
call_type = "invoke"
contract_address = "map_contract"
function = "put"
inputs = ["0x123", 234]  # Numbers can be used directly without quotes
```

After running `sncast multicall run --path file.toml`, a declared contract will be first deployed, and then its function `put` will be invoked.

📝 **Note** The example above demonstrates the use of the `id` property in a deploy call, which is then referenced as the `contract address` in an invoke call. Additionally, the `id` can be referenced in the inputs of deploy and invoke calls 🔥

💡 **Info** Inputs can be either strings (like `"0x123"`) or numbers (like `234`).

📝 **Note** For numbers larger than 2^63 - 1 (that can't fit into `i64`), use string format (e.g., `"9223372036854775808"`) due to TOML parser limitations.

```
$ sncast multicall run --path multicall_example.toml
```

▶ Output:

💡 **Info** Max fee will be automatically computed if `--max-fee <MAX_FEE>` is not passed.

`multicall new` Example

You can also generate multicall template with `multicall new` command, specifying output path.

```
$ sncast multicall new ./template.toml
```

▶ Output:

⚠️ **Warning** Trying to pass any existing file as an output for `multicall new` will result in error, as the command doesn't overwrite by default.

`multicall new` With `overwrite` Argument

If there is a file with the same name as provided, it can be overwritten.

```
$ sncast multicall new ./template.toml --overwrite
```

▶ Output:

Cairo Deployment Scripts

Overview

⚠⚠⚠ Highly experimental code, a subject to change ⚠⚠⚠

Starknet Foundry cast can be used to run deployment scripts written in Cairo, using `script run` subcommand. It aims to provide similar functionality to Foundry's `forge script`.

To start writing a deployment script in Cairo just add `sncast_std` as a dependency to you scarb package and make sure to have a `main` function in the module you want to run. `sncast_std` docs can be found here.

Please note that `sncast script` **is in development**. While it is already possible to declare, deploy, invoke and call contracts from within Cairo, its interface, internals and feature set can change rapidly each version.

⚠⚠ By default, the nonce for each transaction is being taken from the pending block ⚠⚠

Some RPC nodes can be configured with higher poll intervals, which means they may return "older" nonces in pending blocks, or even not be able to obtain pending blocks at all. This might be the case if you get an error like "Invalid transaction nonce" when running a script, and you may need to manually set both nonce and max_fee for transactions.

Example:

```
    let declare_result = declare(
      "Map",
      FeeSettings {
         max_fee: Option::None,
         max_gas: Option::Some(999999),
         max_gas_unit_price: Option::Some(100000000000)
      },
      Option::Some(nonce)
)
      .expect('declare failed');
```

Some of the planned features that will be included in future versions are:

- dispatchers support
- logging
- account creation/deployment
- multicall support
- dry running the scripts

and more!

State file

By default, when you run a script a state file containing information about previous runs will be created. This file can later be used to skip making changes to the network if they were done previously.

To determine if an operation (a function like declare, deploy or invoke) has to be sent to the network, the script will first check if such operation with given arguments already exists in state file. If it does, and previously ended with a success, its execution will be skipped. Otherwise, sncast will attempt to execute this function, and will write its status to the state file afterwards.

To prevent sncast from using the state file, you can set the --no-state-file flag.

A state file is typically named in a following manner:

```
{script name}_{network name}_state.json
```

Suggested directory structures

As sncast scripts are just regular scarb packages, there are multiple ways to incorporate scripts into your existing scarb workspace. Most common directory structures include:

1. `scripts` directory with all the scripts in the same workspace with cairo contracts (default for `sncast script init`)

```
$ tree
```

▶ Output:

> 📝 **Note** You should add `scripts` to `members` field in your top-level Scarb.toml to be able to run the script from anywhere in the workspace - otherwise you will have to run the script from within its directory. To learn more consult Scarb documentation.

You can also have multiple scripts as separate packages, or multiple modules inside one package, like so:

1a. multiple scripts in one package

```
$ tree
```

▶ Output:

1b. multiple scripts as separate packages

```
$ tree
```

▶ Output:

1c. single script with flat directory structure

```
$ tree
```

▶ Output:

2. scripts disjointed from the workspace with cairo contracts

```
$ tree
```

▶ Output:

In order to use this directory structure you must set any contracts you're using as dependencies in script's Scarb.toml, and override `build-external-contracts` property to build those contracts. To learn more consult Scarb documentation.

This setup can be seen in action in Full Example below.

Examples

Initialize a script

To get started, a deployment script with all required elements can be initialized using the following command:

```
$ sncast script init my_script
```

For more details, see init command.

📝 **Note** To include a newly created script in an existing workspace, it must be manually added to the members list in the `Scarb.toml` file, under the defined workspace. For more detailed information about workspaces, please refer to the Scarb documentation.

Minimal Example (Without Contract Deployment)

This example shows how to call an already deployed contract. Please find full example with contract deployment here.

```
use starknet::ContractAddress;
use sncast_std::{call, CallResult};

// A real contract deployed on Sepolia network
const CONTRACT_ADDRESS: felt252 =
    0x07e867f1fa6da2108dd2b3d534f1fbec411c5ec9504eb3baa1e49c7a0bef5ab5;

fn main() {
    let call_result = call(
        CONTRACT_ADDRESS.try_into().unwrap(), selector!("get_greeting"), array![]
    )
        .expect('call failed');

    assert(*call_result.data[1] == 'Hello, Starknet!', *call_result.data[1]);

    println!("{:?}", call_result);
}
```

The script should be included in a Scarb package. The directory structure and config for this example looks like this:

```
$ tree
```

▶ Output:

```
[package]
name = "my_script"
version = "0.1.0"

[dependencies]
starknet = ">=2.8.0"
sncast_std = "0.33.0"
```

To run the script, do:

```
$ sncast \
  script run my_script
  --network sepolia
```

▶ Output:

Full Example (With Contract Deployment)

This example script declares, deploys and interacts with an example `MapContract`:

```
#[starknet::interface]
pub trait IMapContract<State> {
    fn put(ref self: State, key: felt252, value: felt252);
    fn get(self: @State, key: felt252) -> felt252;
}

#[starknet::contract]
pub mod MapContract {
    use starknet::storage::{Map, StorageMapReadAccess, StorageMapWriteAccess};

    #[storage]
    struct Storage {
        storage: Map<felt252, felt252>,
    }

    #[abi(embed_v0)]
    impl MapContractImpl of super::IMapContract<ContractState> {
        fn put(ref self: ContractState, key: felt252, value: felt252) {
            self.storage.write(key, value);
        }

        fn get(self: @ContractState, key: felt252) -> felt252 {
            self.storage.read(key)
        }
    }
}
```

We prepare a script:

```
use sncast_std::{
    declare, deploy, invoke, call, DeclareResult, DeclareResultTrait,
DeployResult, InvokeResult,
    CallResult, get_nonce, FeeSettings
};

fn main() {
    let max_fee = 999999999999999;
    let salt = 0x3;

    let declare_nonce = get_nonce('latest');

    let declare_result = declare(
        "MapContract",
        FeeSettings {
            max_fee: Option::Some(max_fee), max_gas: Option::None,
max_gas_unit_price: Option::None
        },
        Option::Some(declare_nonce)
    )
        .expect('map declare failed');

    let class_hash = declare_result.class_hash();
    let deploy_nonce = get_nonce('pending');

    let deploy_result = deploy(
        *class_hash,
        ArrayTrait::new(),
        Option::Some(salt),
        true,
        FeeSettings {
            max_fee: Option::Some(max_fee), max_gas: Option::None,
max_gas_unit_price: Option::None
        },
        Option::Some(deploy_nonce)
    )
        .expect('map deploy failed');

    assert(deploy_result.transaction_hash != 0, deploy_result.transaction_hash);

    let invoke_nonce = get_nonce('pending');

    let invoke_result = invoke(
        deploy_result.contract_address,
        selector!("put"),
        array![0x1, 0x2],
        FeeSettings {
            max_fee: Option::Some(max_fee), max_gas: Option::None,
max_gas_unit_price: Option::None
        },
        Option::Some(invoke_nonce)
    )
        .expect('map invoke failed');
```

```
    assert(invoke_result.transaction_hash != 0, invoke_result.transaction_hash);

    let call_result = call(deploy_result.contract_address, selector!("get"),
array![0x1])
        .expect('map call failed');

    assert(call_result.data == array![0x2], *call_result.data.at(0));
}
```

The script should be included in a Scarb package. The directory structure and config for this example looks like this:

```
$ tree
```

▶ Output:

```
[package]
name = "map_script"
version = "0.1.0"

[dependencies]
starknet = ">=2.8.0"
sncast_std = "0.33.0"
map = { path = "../contracts" }

[lib]
sierra = true
casm = true

[[target.starknet-contract]]
build-external-contracts = [
    "map::MapContract"
]
```

Please note that `map` contract was specified as the dependency. In our example, it resides in the filesystem. To generate the artifacts for it that will be accessible from the script you need to use the `build-external-contracts` property.

To run the script, do:

```
$ sncast \
  --account example_user \
  script run map_script \
  --network sepolia
```

▶ Output:

As an idempotency feature is turned on by default, executing the same script once again ends with a success and only `call` functions are being executed (as they do not change the network state):

```
$ sncast \
  --account example_user \
  script run map_script \
  --network sepolia
```

▶ Output:

whereas, when we run the same script once again with `--no-state-file` flag set, it fails (as the `Map` contract is already deployed):

```
$ sncast \
  --account example_user \
  script run map_script \
  --network sepolia \
  --no-state-file
```

▶ Output:

Error handling

Each of `declare`, `deploy`, `invoke`, `call` functions return `Result<T, ScriptCommandError>`, where `T` is a corresponding response struct. This allows for various script errors to be handled programmatically. Script errors implement `Debug` trait, allowing the error to be printed to stdout.

Minimal example with `assert!` and `println!`

```
use starknet::ContractAddress;
use sncast_std::{call, CallResult};

// Some nonexistent contract
const CONTRACT_ADDRESS: felt252 = 0x2137;

fn main() {
    // This call fails
    let call_result = call(
        CONTRACT_ADDRESS.try_into().unwrap(), selector!("get_greeting"), array![]
    );

    // Make some assertion
    assert!(call_result.is_err());

    // Print the result error
    println!("Received error: {:?}", call_result.unwrap_err());
}
```

More on deployment scripts errors here.

Inspecting Transactions

Overview

Starknet Foundry `sncast` supports the inspection of transaction statuses on a given network with the `sncast tx-status` command.

For a detailed CLI description, refer to the tx-status command reference.

Usage Examples

Inspecting Transaction Status

You can track the details about the execution and finality status of a transaction in the given network by using the transaction hash as shown below:

```
$ sncast \
 tx-status \
 0x07d2067cd7675f88493a9d773b456c8d941457ecc2f6201d2fe6b0607daadfd1 \
 --network sepolia
```

▶ Output:

Verifying Contracts

Overview

Starknet Foundry `sncast` supports verifying Cairo contract classes with the `sncast verify` command by submitting the source code to a selected verification provider. Verification provides transparency, making the code accessible to users and aiding debugging tools.

The verification provider guarantees that the submitted source code aligns with the deployed contract class on the network by compiling the source code into Sierra bytecode and comparing it with the network-deployed Sierra bytecode.

For detailed CLI description, see verify command reference.

⚠️ **Warning** Please be aware that submitting the source code means it will be publicly exposed through the provider's APIs.

Verification Providers

Walnut

Walnut is a tool for step-by-step debugging of Starknet transactions. You can learn more about Walnut here walnut.dev. Note that Walnut requires you to specify the Starknet version in your `Scarb.toml` config file.

Example

First, ensure that you have created a `Scarb.toml` file for your contract (it should be present in the project directory or one of its parent directories). Make sure the contract has already been deployed on the network.

Then run:

```
$ sncast \
    verify \
    --contract-address
0x0589a8b8bf819b7820cb699ea1f6c409bc012c9b9160106ddc3dacd6a89653cf \
    --contract-name HelloSncast \
    --verifier walnut \
    --network sepolia
```

▶ Output:

📝 **Note** Contract name is a part after the `mod` keyword in your contract file. It may differ from package name defined in `Scarb.toml` file.

Calldata Transformation

For the examples below, we will consider a dedicated contract - `DataTransformerContract`, defined in `data_transformer_contract` project namespace.

It's declared on Sepolia network with class hash `0x02a9b456118a86070a8c116c41b02e490f3dcc9db3cad945b4e9a7fd7cec9168`.

It has a few methods accepting different types and items defined in its namespace:

```
#[derive(Serde, Drop)]
pub struct SimpleStruct {
    a: felt252
}

#[derive(Serde, Drop)]
pub struct NestedStructWithField {
    a: SimpleStruct,
    b: felt252
}

#[derive(Serde, Drop)]
pub enum Enum {
    One: (),
    Two: u128,
    Three: NestedStructWithField
}

#[starknet::interface]
pub trait IDataTransformerContract<TContractState> {
    fn tuple_fn(self: @TContractState, a: (felt252, u8, Enum));
    fn nested_struct_fn(self: @TContractState, a: NestedStructWithField);
    fn complex_fn(
        self: @TContractState,
        arr: Array<Array<felt252>>,
        one: u8,
        two: i8,
        three: ByteArray,
        four: (felt252, u32),
        five: bool,
        six: u256
    );
}


#[starknet::contract]
pub mod DataTransformerContract {
    use super::{NestedStructWithField, Enum};

    #[storage]
    struct Storage {}

    #[abi(embed_v0)]
    impl DataTransformerContractImpl of
super::IDataTransformerContract<ContractState> {
        fn tuple_fn(self: @ContractState, a: (felt252, u8, Enum)) {}

        fn nested_struct_fn(self: @ContractState, a: NestedStructWithField) {}

        fn complex_fn(
            self: @ContractState,
            arr: Array<Array<felt252>>,
            one: u8,
            two: i8,
```

```
            three: ByteArray,
            four: (felt252, u32),
            five: bool,
            six: u256
        ) {}
    }
}
```

A default form of calldata passed to commands requiring it is a series of hex-encoded felts:

```
$ sncast call \
    --network sepolia \
    --contract-address
0x05075f6d418f7c53c6cdc21cbb5aca2b69c83b6fbcc8256300419a9f101c8b77 \
    --function tuple_fn \
    --calldata 0x10 0x3 0x0 \
    --block-id latest
```

💡 **Info** Cast **doesn't verify serialized calldata against the ABI**.
Only expression transformation checks types and arities of functions called on chain.

Using `--arguments`

Instead of serializing calldata yourself, `sncast` allows passing it in a far more handy, human-readable form - as a list of comma-separated Cairo expressions wrapped in single quotes. This can be achieved by using the `--arguments` flag. Cast will perform serialization automatically, based on an ABI of the contract we interact with, following the Starknet specification.

Basic example

We can write the same command as above, but with arguments:

```
$ sncast call \
    --network sepolia \
    --contract-address
0x05075f6d418f7c53c6cdc21cbb5aca2b69c83b6fbcc8256300419a9f101c8b77 \
    --function tuple_fn \
    --arguments '(0x10, 3, hello_sncast::data_transformer_contract::Enum::One)' \
    --block-id latest
```

getting the same result. Note that the arguments must be:

- provided as a single string
- comma (`,`) separated

📝 **Note** User-defined items such as enums and structs should be referred to depending on a way they are defined in ABI.
In general, paths to items have form: `<project-name>::<module-path>::<item-name>`.

Supported Expressions

Cast supports most important Cairo corelib types:

- `bool`
- signed integers (`i8` , `i16` , `i32` , `i64` , `i128`)
- unsigned integers (`u8` , `u16` , `u32` , `u64` , `u96` , `u128` , `u256` , `u384` , `u512`)
- `felt252` (numeric literals and `'shortstrings'`)
- `ByteArray`
- `ContractAddress`
- `ClassHash`
- `StorageAddress`
- `EthAddress`
- `bytes31`
- `Array` - using `array![]` macro

Numeric types (primitives and `felt252`) can be paseed with type suffix specified for example `--arguments 420_u64`.

📝 **Note** Only **constant** expressions are supported. Defining and referencing variables and calling functions (either builtin, user-defined or external) is not allowed.

More Complex Examples

1. `complex_fn` - different data types:

```
$ sncast call \
    --network sepolia \
    --contract-address
0x05075f6d418f7c53c6cdc21cbb5aca2b69c83b6fbcc8256300419a9f101c8b77 \
    --function complex_fn \
    --arguments \
'array![array![1, 2], array![3, 4, 5], array![6]],'\
'12,'\
'-128_i8,'\
'"Some string (a ByteArray)",'\
"('a shortstring', 32_u32),"\
'true,'\
'0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff' \
    --block-id latest
```

📝 **Note** In bash and similar shells indentation and whitespace matters when providing multiline strings with `\`

Remember **not to indent** any line and **not to add whitespace before the `\` character**.

Alternatively, you can continue the single quote for multiple lines.

```
$ sncast call \
    --network sepolia \
    --contract-address
0x05075f6d418f7c53c6cdc21cbb5aca2b69c83b6fbcc8256300419a9f101c8b77 \
    --function complex_fn \
    --arguments 'array![array![1, 2], array![3, 4, 5], array![6]],
12,
-128_i8,
"Some string (a ByteArray)",
('\''a shortstring'\'', 32_u32),
true,
0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff' \
    --block-id latest
```

📝 **Note** In bash and similar shells any `'` must be escaped correctly.

This is also true for `"` when using `"` to wrap the arguments instead of `'`.

---

2. `nested_struct_fn` - struct nesting:

```
$ sncast call \
    --network sepolia \
    --contract-address
0x05075f6d418f7c53c6cdc21cbb5aca2b69c83b6fbcc8256300419a9f101c8b77 \
    --function nested_struct_fn \
    --arguments \
'hello_sncast::data_transformer_contract::NestedStructWithField {'\
'    a: hello_sncast::data_transformer_contract::SimpleStruct { a: 10 },'\
'    b: 12'\
'}'\
     --block-id latest
```

Environment Setup

> 💡 **Info** This tutorial is only relevant if you wish to contribute to Starknet Foundry. If you plan to only use it as a tool for your project, you can skip this part.

Prerequisites

Rust

Install the latest stable Rust version. If you already have Rust installed make sure to upgrade it by running

```
$ rustup update
```

Scarb

You can read more about installing Scarb here.

Please make sure you're using Scarb installed via asdf - otherwise some tests may fail.

> To verify, run:
>
> ```
> $ which scarb
> ```
>
> the result of which should be:
>
> ```
> $HOME/.asdf/shims/scarb
> ```
>
> If you previously installed scarb using an official installer, you may need to remove this installation or modify your PATH to make sure asdf installed one is always used.

cairo-profiler

You can read more about installing `cairo-profiler` here.

> **! Warning**
>
> If you haven't pushed your branch to the remote yet (you've been working only locally), two tests will fail:
>
> - `e2e::running::init_new_project_test`
> - `e2e::running::simple_package_with_git_dependency`
>
> After pushing the branch to the remote, those tests should pass.

Starknet Devnet

To install it run `./scripts/install_devnet.sh`

Universal sierra compiler

Install the latest universal-sierra-compiler version.

Running Tests

Tests can be run with:

```
$ cargo test
```

Formatting and Lints

Starknet Foundry uses rustfmt for formatting. You can run the formatter with

```
$ cargo fmt
```

For linting, it uses clippy. You can run it with this command:

```
$ cargo clippy --all-targets --all-features -- --no-deps -W clippy::pedantic -A
clippy::missing_errors_doc -A clippy::missing_panics_doc -A
clippy::default_trait_access
```

Or using our defined alias

```
$ cargo lint
```

Spelling

Starknet Foundry uses typos for spelling checks.

You can run the checker with

```
$ typos
```

Some typos can be automatically fixed by running

▶ Output:

Contributing

Read the general contribution guideline here

Shell Snippets in Documentation

`snforge` and `sncast` snippets and their outputs present in the docs are automatically run and tested. Some of them need to be configured with specific values to work correctly.

Snippet configuration

To configure a snippet, you need to add a comment block right before it. The comment block should contain the configuration in JSON format. Example:

````
<!-- { "package_name": "hello_starknet", "ignored_output": true } -->
```shell
$ sncast \
    account create \
    --network sepolia \
    --name my_first_account
```

<details>
<summary>Output:</summary>

```shell
command: account create
add_profile: --add-profile flag was not set. No profile added to snfoundry.toml
address: [..]
max_fee: [..]
message: Account successfully created. Prefund generated address with at least
<max_fee> STRK tokens. It is good to send more in the case of higher demand.

To see account creation details, visit:
account: https://sepolia.starkscan.co/contract/[..]
```
</details>
````

Available configuration options

- `ignored` - if set to `true`, the snippet will be ignored and not run.
- `package_name` - the name of the Scarb package in which the snippet should be run.
- `ignored_output` - if set to `true`, the output of executed command will be ignored.

`snforge` CLI Reference

- `snforge test`
- `snforge init`
- `snforge new`
- `snforge clean-cache`
- `snforge check-requirements`

You can check your version of `snforge` via `snforge --version`. To display help run `snforge --help`.

`snforge test`

Run tests for a project in the current directory.

`[TEST_FILTER]`

Passing a test filter will only run tests with an absolute module tree path containing this filter.

`-e`, `--exact`

Will only run a test with a name exactly matching the test filter. Test filter must be a whole qualified test name e.g. `package_name::my_test` instead of just `my_test`.

`-x`, `--exit-first`

Stop executing tests after the first failed test.

`-p`, `--package <SPEC>`

Packages to run this command on, can be a concrete package name (`foobar`) or a prefix glob (`foo*`).

`-w`, `--workspace`

Run tests for all packages in the workspace.

`-r`, `--fuzzer-runs` `<FUZZER_RUNS>`

Number of fuzzer runs.

`-s, --fuzzer-seed <FUZZER_SEED>`

Seed for the fuzzer.

`--ignored`

Run only tests marked with `#[ignore]` attribute.

`--include-ignored`

Run all tests regardless of `#[ignore]` attribute.

`--rerun-failed`

Run tests that failed during the last run

`--color <WHEN>`

Control when colored output is used. Valid values:

- `auto` (default): automatically detect if color support is available on the terminal.
- `always`: always display colors.
- `never`: never display colors.

`--detailed-resources`

Display additional info about used resources for passed tests.

`--save-trace-data`

Saves execution traces of test cases which pass and are not fuzz tests. You can use traces for profiling purposes.

`--build-profile`

Saves trace data and then builds profiles of test cases which pass and are not fuzz tests. You need cairo-profiler installed on your system. You can set a custom path to cairo-profiler with `CAIRO_PROFILER` env variable. Profile can be read with pprof, more information: cairo-profiler, pprof

`--coverage`

Saves trace data and then generates coverage report of test cases which pass and are not fuzz tests. You need cairo-coverage installed on your system. You can set a custom path to cairo-coverage with `CAIRO_COVERAGE` env variable.

`--max-n-steps <MAX_N_STEPS>`

Number of maximum steps during a single test. For fuzz tests this value is applied to each subtest separately.

`-F, --features <FEATURES>`

Comma separated list of features to activate.

`--all-features`

Activate all available features.

`--no-default-features`

Do not activate the `default` feature.

`--no-optimization`

Build contract artifacts in a separate starknet contract target. Enabling this flag will slow down the compilation process, but the built contracts will more closely resemble the ones used on real networks. This is set to `true` when using Scarb version less than `2.8.3`.

`-h`, `--help`

Print help.

`snforge init`

Create a new directory with a `snforge` project.

`<NAME>`

Name of a new project.

`-h`, `--help`

Print help.

> ⚠️ **Warning**
>
> The `snforge init` command is deprecated. Please use the `snforge new` command instead.

`snforge new`

Create a new StarkNet Foundry project at the provided path that should either be empty or not exist.

`<PATH>`

Path to a location where the new project will be created.

`-n`, `--name`

Name of a package to create, defaults to the directory name.

`--no-vcs`

Do not initialize a new Git repository.

`--overwrite`

Try to create the project even if the specified directory is not empty, which can result in overwriting existing files

`-h`, `--help`

Print help.

`snforge clean-cache`

Clean `snforge` cache directory.

`-h`, `--help`

Print help.

`snforge check-requirements`

Validate if all `snforge` requirements are installed.

`-h`, `--help`

Print help.

Cheatcodes Reference

- `mock_call` - mocks a number of contract calls to an entry point
- `start_mock_call` - mocks contract call to an entry point
- `stop_mock_call` - cancels the `mock_call` / `start_mock_call` for an entry point
- `get_class_hash` - retrieves a class hash of a contract
- `replace_bytecode` - replace the class hash of a contract
- `l1_handler` - executes a `#[l1_handler]` function to mock a message arriving from Ethereum
- `spy_events` - creates `EventSpy` instance which spies on events emitted by contracts
- `spy_messages_to_l1` - creates `L1MessageSpy` instance which spies on messages to L1 sent by contracts
- `store` - stores values in targeted contact's storage
- `load` - loads values directly from targeted contact's storage
- `CheatSpan` - enum for specifying the number of target calls for a cheat

Execution Info

Caller Address

- `cheat_caller_address` - changes the caller address for contracts, for a number of calls
- `start_cheat_caller_address_global` - changes the caller address for all contracts
- `start_cheat_caller_address` - changes the caller address for contracts
- `stop_cheat_caller_address` - cancels the `cheat_caller_address` / `start_cheat_caller_address` for contracts
- `stop_cheat_caller_address_global` - cancels the `start_cheat_caller_address_global`

Block Info

Block Number

- `cheat_block_number` - changes the block number for contracts, for a number of calls
- `start_cheat_block_number_global` - changes the block number for all contracts
- `start_cheat_block_number` - changes the block number for contracts
- `stop_cheat_block_number` - cancels the `cheat_block_number` / `start_cheat_block_number` for contracts
- `stop_cheat_block_number_global` - cancels the `start_cheat_block_number_global`

Block Timestamp

- `cheat_block_timestamp` - changes the block timestamp for contracts, for a number of calls
- `start_cheat_block_timestamp_global` - changes the block timestamp for all contracts
- `start_cheat_block_timestamp` - changes the block timestamp for contracts
- `stop_cheat_block_timestamp` - cancels the `cheat_block_timestamp` / `start_cheat_block_timestamp` for contracts
- `stop_cheat_block_timestamp_global` - cancels the `start_cheat_block_timestamp_global`

Sequencer Address

- `cheat_sequencer_address` - changes the sequencer address for contracts, for a number of calls
- `start_cheat_sequencer_address_global` - changes the sequencer address for all contracts
- `start_cheat_sequencer_address` - changes the sequencer address for contracts
- `stop_cheat_sequencer_address` - cancels the `cheat_sequencer_address` / `start_cheat_sequencer_address` for contracts
- `stop_cheat_sequencer_address_global` - cancels the `start_cheat_sequencer_address_global`

Transaction Info

Transaction Version

- `cheat_transaction_version` - changes the transaction version for contracts, for a number of calls
- `start_cheat_transaction_version_global` - changes the transaction version for all contracts
- `start_cheat_transaction_version` - changes the transaction version for contracts
- `stop_cheat_transaction_version` - cancels the `cheat_transaction_version` / `start_cheat_transaction_version` for contracts
- `stop_cheat_transaction_version_global` - cancels the `start_cheat_transaction_version_global`

Transaction Max Fee

- `cheat_max_fee` - changes the transaction max fee for contracts, for a number of calls
- `start_cheat_max_fee_global` - changes the transaction max fee for all contracts
- `start_cheat_max_fee` - changes the transaction max fee for contracts
- `stop_cheat_max_fee` - cancels the `cheat_max_fee` / `start_cheat_max_fee` for contracts
- `stop_cheat_max_fee_global` - cancels the `start_cheat_max_fee_global`

Transaction Signature

- `cheat_signature` - changes the transaction signature for contracts, for a number of calls
- `start_cheat_signature_global` - changes the transaction signature for all contracts
- `start_cheat_signature` - changes the transaction signature for contracts
- `stop_cheat_signature` - cancels the `cheat_signature` / `start_cheat_signature` for contracts
- `stop_cheat_signature_global` - cancels the `start_cheat_signature_global`

Transaction Hash

- `cheat_transaction_hash` - changes the transaction hash for contracts, for a number of calls
- `start_cheat_transaction_hash_global` - changes the transaction hash for all contracts
- `start_cheat_transaction_hash` - changes the transaction hash for contracts

- `stop_cheat_transaction_hash` - cancels the `cheat_transaction_hash` / `start_cheat_transaction_hash` for contracts
- `stop_cheat_transaction_hash_global` - cancels the `start_cheat_transaction_hash_global`

Transaction Chain ID

- `cheat_chain_id` - changes the transaction chain_id for contracts, for a number of calls
- `start_cheat_chain_id_global` - changes the transaction chain_id for all contracts
- `start_cheat_chain_id` - changes the transaction chain_id for contracts
- `stop_cheat_chain_id` - cancels the `cheat_chain_id` / `start_cheat_chain_id` for contracts
- `stop_cheat_chain_id_global` - cancels the `start_cheat_chain_id_global`

Transaction Nonce

- `cheat_nonce` - changes the transaction nonce for contracts, for a number of calls
- `start_cheat_nonce_global` - changes the transaction nonce for all contracts
- `start_cheat_nonce` - changes the transaction nonce for contracts
- `stop_cheat_nonce` - cancels the `cheat_nonce` / `start_cheat_nonce` for contracts
- `stop_cheat_nonce_global` - cancels the `start_cheat_nonce_global`

Transaction Resource Bounds

- `cheat_resource_bounds` - changes the transaction resource bounds for contracts, for a number of calls
- `start_cheat_resource_bounds_global` - changes the transaction resource bounds for all contracts
- `start_cheat_resource_bounds` - changes the transaction resource bounds for contracts
- `stop_cheat_resource_bounds` - cancels the `cheat_resource_bounds` / `start_cheat_resource_bounds` for contracts
- `stop_cheat_resource_bounds_global` - cancels the `start_cheat_resource_bounds_global`

Transaction Tip

- `cheat_tip` - changes the transaction tip for contracts, for a number of calls

- `start_cheat_tip_global` - changes the transaction tip for all contracts
- `start_cheat_tip` - changes the transaction tip for contracts
- `stop_cheat_tip` - cancels the `cheat_tip` / `start_cheat_tip` for contracts
- `stop_cheat_tip_global` - cancels the `start_cheat_tip_global`

Transaction Paymaster Data

- `cheat_paymaster_data` - changes the transaction paymaster data for contracts, for a number of calls
- `start_cheat_paymaster_data_global` - changes the transaction paymaster data for all contracts
- `start_cheat_paymaster_data` - changes the transaction paymaster data for contracts
- `stop_cheat_paymaster_data` - cancels the `cheat_paymaster_data` / `start_cheat_paymaster_data` for contracts
- `stop_cheat_paymaster_data_global` - cancels the `start_cheat_paymaster_data_global`

Transaction Nonce Data Availability Mode

- `cheat_nonce_data_availability_mode` - changes the transaction nonce data availability mode for contracts, for a number of calls
- `start_cheat_nonce_data_availability_mode_global` - changes the transaction nonce data availability mode for all contracts
- `start_cheat_nonce_data_availability_mode` - changes the transaction nonce data availability mode for contracts
- `stop_cheat_nonce_data_availability_mode` - cancels the `cheat_nonce_data_availability_mode` / `start_cheat_nonce_data_availability_mode` for contracts
- `stop_cheat_nonce_data_availability_mode_global` - cancels the `start_cheat_nonce_data_availability_mode_global`

Transaction Fee Data Availability Mode

- `cheat_fee_data_availability_mode` - changes the transaction fee data availability mode for contracts, for a number of calls
- `start_cheat_fee_data_availability_mode_global` - changes the transaction fee data availability mode for all contracts
- `start_cheat_fee_data_availability_mode` - changes the transaction fee data availability mode for contracts

- `stop_cheat_fee_data_availability_mode` - cancels the `cheat_fee_data_availability_mode` / `start_cheat_fee_data_availability_mode` for contracts
- `stop_cheat_fee_data_availability_mode_global` - cancels the `start_cheat_fee_data_availability_mode_global`

Transaction Account Deployment

- `cheat_account_deployment_data` - changes the transaction account deployment data for contracts, for a number of calls
- `start_cheat_account_deployment_data_global` - changes the transaction account deployment data for all contracts
- `start_cheat_account_deployment_data` - changes the transaction account deployment data for contracts
- `stop_cheat_account_deployment_data` - cancels the `cheat_account_deployment_data` / `start_cheat_account_deployment_data` for contracts
- `stop_cheat_account_deployment_data_global` - cancels the `start_cheat_account_deployment_data_global`

Account Contract Address

- `cheat_account_contract_address` - changes the address of an account which the transaction originates from, for the given target and span
- `start_cheat_account_contract_address_global` - changes the address of an account which the transaction originates from, for all targets
- `start_cheat_account_contract_address` - changes the address of an account which the transaction originates from, for the given target
- `stop_cheat_account_contract_address` - cancels the `cheat_account_deployment_data` / `start_cheat_account_deployment_data` for the given target
- `stop_cheat_account_contract_address_global` - cancels the `start_cheat_account_contract_address_global`

ℹ️ **Info** To use cheatcodes you need to add `snforge_std` package as a development dependency in your `Scarb.toml` using the appropriate version.

```
[dev-dependencies]
snforge_std = "0.33.0"
```

Cheating Globally

Cheatcodes which have `_global` suffix allow to change specific properties in blockchain state for all targets and for indefinite time span. Therefore, you don't pass the target address, nor the span.

See the Cheating Addresses Globally example.

CheatSpan

```
enum CheatSpan {
    Indefinite: (),
    TargetCalls: usize
}
```

`CheatSpan` is an enum used to specify for how long the target should be cheated for.

- `Indefinite` applies the cheatcode indefinitely, until the cheat is canceled manually (e.g. using `stop_cheat_block_timestamp`).
- `TargetCalls` applies the cheatcode for a specified number of calls to the target, after which the cheat is canceled (or until the cheat is canceled manually).

caller_address

Cheatcodes modifying `caller_address`:

cheat_caller_address

```
fn cheat_caller_address(target: ContractAddress, caller_address: ContractAddress, span: CheatSpan)
```

Changes the caller address for the given target and span.

start_cheat_caller_address_global

```
fn start_cheat_caller_address_global(caller_address: ContractAddress)
```

Changes the caller address for all targets.

start_cheat_caller_address

```
fn start_cheat_caller_address(target: ContractAddress, caller_address: ContractAddress)
```

Changes the caller address for the given target.

stop_cheat_caller_address

```
fn stop_cheat_caller_address(target: ContractAddress)
```

Cancels the `cheat_caller_address` / `start_cheat_caller_address` for the given target.

`stop_cheat_caller_address_global`

```
fn stop_cheat_caller_address_global()
```

Cancels the `start_cheat_caller_address_global`.

block_number

Cheatcodes modifying `block_number`:

cheat_block_number

```
fn cheat_block_number(target: ContractAddress, block_number: u64, span: CheatSpan)
```

Changes the block number for the given target and span.

start_cheat_block_number_global

```
fn start_cheat_block_number_global(block_number: u64)
```

Changes the block number for all targets.

start_cheat_block_number

```
fn start_cheat_block_number(target: ContractAddress, block_number: u64)
```

Changes the block number for the given target.

stop_cheat_block_number

```
fn stop_cheat_block_number(target: ContractAddress)
```

Cancels the `cheat_block_number` / `start_cheat_block_number` for the given target.

`stop_cheat_block_number_global`

```
fn stop_cheat_block_number_global()
```

Cancels the `start_cheat_block_number_global`.

block_timestamp

Cheatcodes modifying `block_timestamp`:

cheat_block_timestamp

```
fn cheat_block_timestamp(target: ContractAddress, block_timestamp: u64, span: CheatSpan)
```

Changes the block timestamp for the given target and span.

start_cheat_block_timestamp_global

```
fn start_cheat_block_timestamp_global(block_timestamp: u64)
```

Changes the block timestamp for all targets.

start_cheat_block_timestamp

```
fn start_cheat_block_timestamp(target: ContractAddress, block_timestamp: u64)
```

Changes the block timestamp for the given target.

stop_cheat_block_timestamp

```
fn stop_cheat_block_timestamp(target: ContractAddress)
```

Cancels the `cheat_block_timestamp` / `start_cheat_block_timestamp` for the given target.

`stop_cheat_block_timestamp_global`

```
fn stop_cheat_block_timestamp_global()
```

Cancels the `start_cheat_block_timestamp_global`.

sequencer_address

Cheatcodes modifying `sequencer_address`:

cheat_sequencer_address

```
fn cheat_sequencer_address(target: ContractAddress, sequencer_address: ContractAddress, span: CheatSpan)
```

Changes the sequencer address for the given target and span.

start_cheat_sequencer_address_global

```
fn start_cheat_sequencer_address_global(sequencer_address: ContractAddress)
```

Changes the sequencer address for all targets.

start_cheat_sequencer_address

```
fn start_cheat_sequencer_address(target: ContractAddress, sequencer_address: ContractAddress)
```

Changes the sequencer address for the given target.

stop_cheat_sequencer_address

```
fn stop_cheat_sequencer_address(target: ContractAddress)
```

Cancels the `cheat_sequencer_address` / `start_cheat_sequencer_address` for the given target.

`stop_cheat_sequencer_address_global`

```
fn stop_cheat_sequencer_address_global()
```

Cancels the `start_cheat_sequencer_address_global`.

Transaction `version`

Cheatcodes modifying transaction `version`:

`cheat_transaction_version`

```
fn cheat_transaction_version(target: ContractAddress, version: felt252, span: CheatSpan)
```

Changes the transaction version for the given target and span.

`start_cheat_transaction_version_global`

```
fn start_cheat_transaction_version_global(version: felt252)
```

Changes the transaction version for all targets.

`start_cheat_transaction_version`

```
fn start_cheat_transaction_version(target: ContractAddress, version: felt252)
```

Changes the transaction version for the given target.

`stop_cheat_transaction_version`

```
fn stop_cheat_transaction_version(target: ContractAddress)
```

Cancels the `cheat_transaction_version` / `start_cheat_transaction_version` for the given target.

`stop_cheat_transaction_version_global`

```
fn stop_cheat_transaction_version_global()
```

Cancels the `start_cheat_transaction_version_global`.

account_contract_address

Cheatcodes modifying `account_contract_address`:

cheat_account_contract_address

```
fn cheat_account_contract_address(target: ContractAddress,
account_contract_address: ContractAddress, span: CheatSpan)
```

Changes the address of an account which the transaction originates from, for the given target and span.

start_cheat_account_contract_address_global

```
fn start_cheat_account_contract_address_global(account_contract_address:
ContractAddress)
```

Changes the address of an account which the transaction originates from, for all targets.

start_cheat_account_contract_address

```
fn start_cheat_account_contract_address(target: ContractAddress,
account_contract_address: ContractAddress)
```

Changes the address of an account which the transaction originates from, for the given target.

`stop_cheat_account_contract_address`

```
fn stop_cheat_account_contract_address(target: ContractAddress)
```

Cancels the `cheat_account_contract_address` / `start_cheat_account_contract_address` for the given target.

`stop_cheat_account_contract_address_global`

```
fn stop_cheat_account_contract_address_global()
```

Cancels the `start_cheat_account_contract_address_global`.

max_fee

Cheatcodes modifying `max_fee`:

cheat_max_fee

```
fn cheat_max_fee(target: ContractAddress, max_fee: u128, span: CheatSpan)
```

Changes the transaction max fee for the given target and span.

start_cheat_max_fee_global

```
fn start_cheat_max_fee_global(max_fee: u128)
```

Changes the transaction max fee for all targets.

start_cheat_max_fee

```
fn start_cheat_max_fee(target: ContractAddress, max_fee: u128)
```

Changes the transaction max fee for the given target.

stop_cheat_max_fee

```
fn stop_cheat_max_fee(target: ContractAddress)
```

Cancels the `cheat_max_fee` / `start_cheat_max_fee` for the given target.

`stop_cheat_max_fee_global`

```
fn stop_cheat_max_fee_global()
```

Cancels the `start_cheat_max_fee_global`.

signature

Cheatcodes modifying `signature`:

cheat_signature

```
fn cheat_signature(target: ContractAddress, signature: Span<felt252>, span: CheatSpan)
```

Changes the transaction signature for the given target and span.

start_cheat_signature_global

```
fn start_cheat_signature_global(signature: Span<felt252>)
```

Changes the transaction signature for all targets.

start_cheat_signature

```
fn start_cheat_signature(target: ContractAddress, signature: Span<felt252>)
```

Changes the transaction signature for the given target.

stop_cheat_signature

```
fn stop_cheat_signature(target: ContractAddress)
```

Cancels the `cheat_signature` / `start_cheat_signature` for the given target.

`stop_cheat_signature_global`

```
fn stop_cheat_signature_global()
```

Cancels the `start_cheat_signature_global`.

transaction_hash

Cheatcodes modifying `transaction_hash`:

cheat_transaction_hash

```
fn cheat_transaction_hash(target: ContractAddress, transaction_hash: felt252, span: CheatSpan)
```

Changes the transaction hash for the given target and span.

start_cheat_transaction_hash_global

```
fn start_cheat_transaction_hash_global(transaction_hash: felt252)
```

Changes the transaction hash for all targets.

start_cheat_transaction_hash

```
fn start_cheat_transaction_hash(target: ContractAddress, transaction_hash: felt252)
```

Changes the transaction hash for the given target.

stop_cheat_transaction_hash

```
fn stop_cheat_transaction_hash(target: ContractAddress)
```

Cancels the `cheat_transaction_hash` / `start_cheat_transaction_hash` for the given target.

`stop_cheat_transaction_hash_global`

```
fn stop_cheat_transaction_hash_global()
```

Cancels the `start_cheat_transaction_hash_global`.

chain_id

Cheatcodes modifying `chain_id`:

cheat_chain_id

```
fn cheat_chain_id(target: ContractAddress, chain_id: felt252, span: CheatSpan)
```

Changes the transaction chain_id for the given target and span.

start_cheat_chain_id_global

```
fn start_cheat_chain_id_global(chain_id: felt252)
```

Changes the transaction chain_id for all targets.

start_cheat_chain_id

```
fn start_cheat_chain_id(target: ContractAddress, chain_id: felt252)
```

Changes the transaction chain_id for the given target.

stop_cheat_chain_id

```
fn stop_cheat_chain_id(target: ContractAddress)
```

Cancels the `cheat_chain_id` / `start_cheat_chain_id` for the given target.

`stop_cheat_chain_id_global`

```
fn stop_cheat_chain_id_global()
```

Cancels the `start_cheat_chain_id_global`.

nonce

Cheatcodes modifying `nonce`:

cheat_nonce

```
fn cheat_nonce(target: ContractAddress, nonce: felt252, span: CheatSpan)
```

Changes the transaction nonce for the given target and span.

start_cheat_nonce_global

```
fn start_cheat_nonce_global(nonce: felt252)
```

Changes the transaction nonce for all targets.

start_cheat_nonce

```
fn start_cheat_nonce(target: ContractAddress, nonce: felt252)
```

Changes the transaction nonce for the given target.

stop_cheat_nonce

```
fn stop_cheat_nonce(target: ContractAddress)
```

Cancels the `cheat_nonce` / `start_cheat_nonce` for the given target.

`stop_cheat_nonce_global`

```
fn stop_cheat_nonce_global()
```

Cancels the `start_cheat_nonce_global`.

resource_bounds

Cheatcodes modifying `resource_bounds`:

cheat_resource_bounds

```
fn cheat_resource_bounds(target: ContractAddress, resource_bounds: Span<ResourceBounds>, span: CheatSpan)
```

Changes the transaction resource bounds for the given target and span.

start_cheat_resource_bounds_global

```
fn start_cheat_resource_bounds_global(resource_bounds: Span<ResourceBounds>)
```

Changes the transaction resource bounds for all targets.

start_cheat_resource_bounds

```
fn start_cheat_resource_bounds(target: ContractAddress, resource_bounds: Span<ResourceBounds>)
```

Changes the transaction resource bounds for the given target.

stop_cheat_resource_bounds

```
fn stop_cheat_resource_bounds(target: ContractAddress)
```

Cancels the `cheat_resource_bounds` / `start_cheat_resource_bounds` for the given target.

`stop_cheat_resource_bounds_global`

```
fn stop_cheat_resource_bounds_global()
```

Cancels the `start_cheat_resource_bounds_global`.

`tip`

Cheatcodes modifying `tip`:

`cheat_tip`

```
fn cheat_tip(target: ContractAddress, tip: u128, span: CheatSpan)
```

Changes the transaction tip for the given target and span.

`start_cheat_tip_global`

```
fn start_cheat_tip_global(tip: u128)
```

Changes the transaction tip for all targets.

`start_cheat_tip`

```
fn start_cheat_tip(target: ContractAddress, tip: u128)
```

Changes the transaction tip for the given target.

`stop_cheat_tip`

```
fn stop_cheat_tip(target: ContractAddress)
```

Cancels the `cheat_tip` / `start_cheat_tip` for the given target.

`stop_cheat_tip_global`

```
fn stop_cheat_tip_global()
```

Cancels the `start_cheat_tip_global`.

paymaster_data

Cheatcodes modifying `paymaster_data`:

cheat_paymaster_data

```
fn cheat_paymaster_data(target: ContractAddress, paymaster_data: Span<felt252>, span: CheatSpan)
```

Changes the transaction paymaster data for the given target and span.

start_cheat_paymaster_data_global

```
fn start_cheat_paymaster_data_global(paymaster_data: Span<felt252>)
```

Changes the transaction paymaster data for all targets.

start_cheat_paymaster_data

```
fn start_cheat_paymaster_data(target: ContractAddress, paymaster_data: Span<felt252>)
```

Changes the transaction paymaster data for the given target.

stop_cheat_paymaster_data

```
fn stop_cheat_paymaster_data(target: ContractAddress)
```

Cancels the `cheat_paymaster_data` / `start_cheat_paymaster_data` for the given target.

`stop_cheat_paymaster_data_global`

```
fn stop_cheat_paymaster_data_global()
```

Cancels the `start_cheat_paymaster_data_global`.

nonce_data_availability_mode

Cheatcodes modifying `nonce_data_availability_mode`:

cheat_nonce_data_availability_mode

```
fn cheat_nonce_data_availability_mode(target: ContractAddress,
nonce_data_availability_mode: u32, span: CheatSpan)
```

Changes the transaction nonce data availability mode for the given target and span.

start_cheat_nonce_data_availability_mode_global

```
fn
start_cheat_nonce_data_availability_mode_global(nonce_data_availability_mode:
u32)
```

Changes the transaction nonce data availability mode for all targets.

start_cheat_nonce_data_availability_mode

```
fn start_cheat_nonce_data_availability_mode(target: ContractAddress,
nonce_data_availability_mode: u32)
```

Changes the transaction nonce data availability mode for the given target.

`stop_cheat_nonce_data_availability_mode`

```
fn stop_cheat_nonce_data_availability_mode(target: ContractAddress)
```

Cancels the `cheat_nonce_data_availability_mode` / `start_cheat_nonce_data_availability_mode` for the given target.

`stop_cheat_nonce_data_availability_mode_global`

```
fn stop_cheat_nonce_data_availability_mode_global()
```

Cancels the `start_cheat_nonce_data_availability_mode_global`.

fee_data_availability_mode

Cheatcodes modifying `fee_data_availability_mode`:

cheat_fee_data_availability_mode

```
fn cheat_fee_data_availability_mode(target: ContractAddress,
fee_data_availability_mode: u32, span: CheatSpan)
```

Changes the transaction fee data availability mode for the given target and span.

start_cheat_fee_data_availability_mode_global

```
fn start_cheat_fee_data_availability_mode_global(fee_data_availability_mode:
u32)
```

Changes the transaction fee data availability mode for all targets.

start_cheat_fee_data_availability_mode

```
fn start_cheat_fee_data_availability_mode(target: ContractAddress,
fee_data_availability_mode: u32)
```

Changes the transaction fee data availability mode for the given target.

`stop_cheat_fee_data_availability_mode`

```
fn stop_cheat_fee_data_availability_mode(target: ContractAddress)
```

Cancels the `cheat_fee_data_availability_mode` / `start_cheat_fee_data_availability_mode` for the given target.

`stop_cheat_fee_data_availability_mode_global`

```
fn stop_cheat_fee_data_availability_mode_global()
```

Cancels the `start_cheat_fee_data_availability_mode_global`.

account_deployment_data

Cheatcodes modifying `account_deployment_data`:

cheat_account_deployment_data

```
fn cheat_account_deployment_data(target: ContractAddress,
account_deployment_data: Span<felt252>, span: CheatSpan)
```

Changes the transaction account deployment data for the given target and span.

start_cheat_account_deployment_data_global

```
fn start_cheat_account_deployment_data_global(account_deployment_data:
Span<felt252>)
```

Changes the transaction account deployment data for all targets.

start_cheat_account_deployment_data

```
fn start_cheat_account_deployment_data(target: ContractAddress,
account_deployment_data: Span<felt252>)
```

Changes the transaction account deployment data for the given target.

`stop_cheat_account_deployment_data`

```
fn stop_cheat_account_deployment_data(target: ContractAddress)
```

Cancels the `cheat_account_deployment_data` / `start_cheat_account_deployment_data` for the given target.

`stop_cheat_account_deployment_data_global`

```
fn stop_cheat_account_deployment_data_global()
```

Cancels the `start_cheat_account_deployment_data_global`.

mock_call

Cheatcodes mocking contract entry point calls:

mock_call

```
fn mock_call<T, impl TSerde: serde::Serde<T>, impl TDestruct: Destruct<T>>(
contract_address: ContractAddress, function_selector: felt252, ret_data: T,
n_times: u32 )
```

Mocks contract call to a `function_selector` of a contract at the given address, for `n_times` first calls that are made to the contract. A call to function `function_selector` will return data provided in `ret_data` argument. An address with no contract can be mocked as well. An entrypoint that is not present on the deployed contract is also possible to mock. Note that the function is not meant for mocking internal calls - it works only for contract entry points.

start_mock_call

```
fn start_mock_call<T, impl TSerde: serde::Serde<T>, impl TDestruct:
Destruct<T>>( contract_address: ContractAddress, function_selector: felt252,
ret_data: T )
```

Mocks contract call to a `function_selector` of a contract at the given address, indefinitely. See `mock_call` for comprehensive definition of how it can be used.

stop_mock_call

```
fn stop_mock_call(contract_address: ContractAddress, function_selector:
felt252)
```

Cancels the `mock_call` / `start_mock_call` for the function `function_selector` of a contract at the given address.

get_class_hash

```
fn get_class_hash(contract_address: ContractAddress) -> ClassHash
```

Returns a class hash of a contract at the specified address.

> 💡 **Tip**
>
> This cheatcode can be used to test if your contract upgrade procedure is correct

`replace_bytecode`

`fn replace_bytecode(contract: ContractAddress, new_class: ClassHash) -> Result<(), ReplaceBytecodeError>`

Replaces class for given contract address. The `new_class` hash has to be declared in order for the replacement class to execute the code when interacting with the contract. Returns `Result::Ok` if the replacement succeeded, and a `ReplaceBytecodeError` with appropriate error type otherwise

ReplaceBytecodeError

An enum with appropriate type of replacement failure

```
pub enum ReplaceBytecodeError {
    /// Means that the contract does not exist, and thus bytecode cannot be replaced
    ContractNotDeployed,
    /// Means that the given class for replacement is not declared
    UndeclaredClassHash,
}
```

l1_handler

```
fn new(target: ContractAddress, selector: felt252) -> L1Handler
```

Returns a structure referring to an L1 handler function.

```
fn execute(self: L1Handler) -> SyscallResult<()>
```

Mocks an L1 -> L2 message from Ethereum handled by the given L1 handler function.

spy_events

```
fn spy_events() -> EventSpy
```

Creates `EventSpy` instance which spies on events emitted after its creation.

```
struct EventSpy {
    ...
}
```

An event spy structure.

```
struct Events {
    events: Array<(ContractAddress, Event)>
}
```

A wrapper structure on an array of events to handle event filtering.

```
struct Event {
    keys: Array<felt252>,
    data: Array<felt252>
}
```

Raw event format (as seen via the RPC-API), can be used for asserting the emitted events.

Implemented traits

EventSpyTrait

```
trait EventSpyTrait {
    fn get_events(ref self: EventSpy) -> Events;
}
```

Gets all events since the creation of the given `EventSpy`.

EventSpyAssertionsTrait

```
trait EventSpyAssertionsTrait<T, impl TEvent: starknet::Event<T>, impl TDrop:
Drop<T>> {
    fn assert_emitted(ref self: EventSpy, events: @Array<(ContractAddress, T)>);
    fn assert_not_emitted(ref self: EventSpy, events: @Array<(ContractAddress,
T)>);
}
```

Allows to assert the expected events emission (or lack thereof), in the scope of the `EventSpy` structure.

EventsFilterTrait

```
trait EventsFilterTrait {
    fn emitted_by(self: @Events, contract_address: ContractAddress) -> Events;
}
```

Filters events emitted by a given `ContractAddress`.

`spy_messages_to_l1`

```
fn spy_messages_to_l1() -> MessageToL1Spy
```

Creates `MessageToL1Spy` instance that spies on all messages sent to L1 after its creation.

```
struct MessageToL1Spy {
    // ..
}
```

Message spy structure allowing to get messages emitted only after its creation.

```
struct MessagesToL1 {
    messages: Array<(ContractAddress, MessageToL1)>
}
```

A wrapper structure on an array of messages to handle filtering smoothly. `messages` is an array of `(l2_sender_address, message)` tuples.

```
struct MessageToL1 {
    /// An ethereum address where the message is destined to go
    to_address: EthAddress,
    /// Actual payload which will be delivered to L1 contract
    payload: Array<felt252>
}
```

Raw message to L1 format (as seen via the RPC-API), can be used for asserting the sent messages.

Implemented traits

MessageToL1SpyTrait

```
trait MessageToL1SpyTrait {
    /// Gets all messages given [`MessageToL1Spy`] spies for.
    fn get_messages(ref self: MessageToL1Spy) -> MessagesToL1;
}
```

Gets all messages since the creation of the given `MessageToL1Spy`.

MessageToL1SpyAssertionsTrait

```
trait MessageToL1SpyAssertionsTrait {
    fn assert_sent(ref self: MessageToL1Spy, messages: @Array<(ContractAddress,
MessageToL1)>);
    fn assert_not_sent(ref self: MessageToL1Spy, messages:
@Array<(ContractAddress, MessageToL1)>);
}
```

Allows to assert the expected sent messages (or lack thereof), in the scope of `MessageToL1Spy` structure.

MessageToL1FilterTrait

```
trait MessageToL1FilterTrait {
    /// Filter messages emitted by a sender of a given [`ContractAddress`]
    fn sent_by(self: @MessagesToL1, contract_address: ContractAddress) ->
MessagesToL1;
    /// Filter messages emitted by a receiver of a given ethereum address
    fn sent_to(self: @MessagesToL1, to_address: EthAddress) -> MessagesToL1;
}
```

Filters messages emitted by a given `ContractAddress`, or sent to given `EthAddress`.

store

```
fn store(target: ContractAddress, storage_address: felt252, serialized_value: Span<felt252>)
```

Stores felts from `serialized_value` in `target` contract's storage, starting at `storage_address`.

load

```
fn load(target: ContractAddress, storage_address: felt252, size: felt252) -> Array<felt252>
```

Loads `size` felts from `target` contract's storage into an `Array`, starting at `storage_address`.

`generate_random_felt`

```
fn generate_random_felt() -> Felt252
```

Generates a (pseudo) random `Felt252` value.

Library Reference

> ℹ️ **Info** Full documentation for the `snforge` library can be found here.

- `declare` - declares a contract and returns a `ContractClass` which can be interacted with later
- `get_call_trace` - gets current test call trace (with contracts interactions included)
- `fs` - module containing functions for interacting with the filesystem
- `env` - module containing functions for interacting with the system environment
- `signature` - module containing struct and trait for creating `ecdsa` signatures

> ℹ️ **Info** To use cheatcodes you need to add `snforge_std` package as a development dependency in your `Scarb.toml` using the appropriate version.

```
[dev-dependencies]
snforge_std = "0.33.0"
```

`byte_array` Module

Module containing utilities for manipulating `ByteArray`s.

Functions

```
fn try_deserialize_bytearray_error(x: Span<felt252>) -> Result<ByteArray, ByteArray>
```

This function is meant to transform a serialized output from a contract call into a `ByteArray`. Returns the parsed `ByteArray`, or an `Err` with reason, if the parsing failed.

declare

```
#[derive(Drop, Serde, Clone)]
enum DeclareResult {
    Success: ContractClass,
    AlreadyDeclared: ContractClass,
}

trait DeclareResultTrait {
    /// Gets inner `ContractClass`
    /// `self` - an instance of the struct `DeclareResult` which is obtained by
calling `declare`
    // Returns the `@ContractClass`
    fn contract_class(self: @DeclareResult) -> @ContractClass;
}

fn declare(contract: ByteArray) -> Result<DeclareResult, Array<felt252>>
```

Declares a contract for later deployment.

Returns the `DeclareResult` that encapsulated possible outcomes in the enum:

- `Success`: Contains the successfully declared `ContractClass`.
- `AlreadyDeclared`: Contains `ContractClass` and signals that the contract has already been declared.

See docs of `ContractClass` for more info about the resulting struct.

ContractClass

A struct which enables interaction with given class hash. It can be obtained by using declare, or created with an arbitrary `ClassHash`.

```
struct ContractClass {
    class_hash: ClassHash,
}
```

Implemented traits

ContractClassTrait

```
trait ContractClassTrait {
    fn precalculate_address(
        self: @ContractClass, constructor_calldata: @Array::<felt252>
    ) -> ContractAddress;

    fn deploy(
        self: @ContractClass, constructor_calldata: @Array::<felt252>
    ) -> SyscallResult<(ContractAddress, Span<felt252>)>;

    fn deploy_at(
        self: @ContractClass,
        constructor_calldata: @Array::<felt252>,
        contract_address: ContractAddress
    ) -> SyscallResult<(ContractAddress, Span<felt252>)>;

    fn new<T, +Into<T, ClassHash>>(class_hash: T) -> ContractClass;
}
```

`get_call_trace`

```
fn get_call_trace() -> CallTrace;
```

(For whole structure definition, please refer to `snforge-std` source)

Gets current call trace of the test, up to the last call made to a contract.

The whole structure is represented as a tree of calls, in which each contract interaction is a new execution scope - thus resulting in a new nested trace.

> **Note**
>
> The topmost-call is representing the test call, which will always be present if you're running a test.

Displaying the trace

The `CallTrace` structure implements a `Display` trait, for a pretty-print with indentations

```
println!("{}", get_call_trace());
```

`fs` Module

Module containing functions for interacting with the filesystem.

`File`

```
trait FileTrait {
    fn new(path: ByteArray) -> File;
}
```

`FileParser<T>`

```
trait FileParser<T, +Serde<T>> {
    fn parse_txt(file: @File) -> Option<T>;
    fn parse_json(file: @File) -> Option<T>;
}
```

`read_txt` & `read_json`

```
fn read_txt(file: @File) -> Array<felt252>;
fn read_json(file: @File) -> Array<felt252>;
```

File format

Some rules have to be checked when providing a file for snforge, in order for correct parsing behavior. Different ones apply for JSON and plain text files.

Plain text files

- Elements have to be separated with newlines
- Elements have to be either:

- integers in range of `[0, P)` where P is `Cairo Prime` either in decimal or `0x` prefixed hex format
- single line short strings (`felt252`) of length `<=31` surrounded by `''` i.e., `'short string'` , new lines can be used with `\n` and `'` with `\'`
- single line strings (`ByteArray`) surrounded by `""` i.e., `"very very very very looooooong string"` , new lines can be used with `\n` and `"` with `\"`

JSON files

- Elements have to be either:
  - integers in range of `[0, P)` where P is `Cairo Prime`
  - single line strings (`ByteArray`) i.e. `"very very very very looooooong string"` , new lines can be used with `\n` and `"` with `\"`
  - array of integers or strings fulfilling the above conditions

---

> ⚠️ **Warning**
>
> A JSON object is an unordered data structure. To make reading JSONs deterministic, the values are read from the JSON in an order that is alphabetical in respect to JSON keys. Nested JSON values are sorted by the flattened format keys `(a.b.c)` .

Example

For example, this plain text file content:

```
1
2
'hello'
10
"world"
```

or this JSON file content:

```
{
  "a": 1,
  "nested": {
    "b": 2,
    "c": 448378203247
  },
  "d": 10,
  "e": "world"
}
```

(note that short strings cannot be used in JSON file)

could be parsed to the following struct in cairo, via `parse_txt` / `parse_json`:

```
A {
    a: 1,
    nested: B {
        b: 2,
        c: 'hello',
    },
    d: 10,
    e: "world"
}
```

or to an array, via `read_txt` / `read_json`:

```
array![1, 2, 'hello', 10, 0, 512970878052, 5]
```

`env` Module

Module containing functions for interacting with the system environment.

`var`

```
fn var(name: ByteArray) -> Array<felt252>
```

Reads an environment variable, without parsing it.

The serialized output is correlated with the inferred input type, same as during reading from a file.

> 📝 **Note**
>
> If you want snfoundry to treat your variable like a short string, surround it with 'single quotes'.
>
> If you would like it to be serialized as a `ByteArray`, use "double quoting". It will be then de-serializable with `Serde`.

`signature` Module

Module containing `KeyPair` struct and interface for creating `ecdsa` signatures.

- `signature::stark_curve` - implementation of `KeyPairTrait` for the STARK curve
- `signature::secp256k1_curve` - implementation of `KeyPairTrait` for Secp256k1 Curve
- `signature::secp256r1_curve` - implementation of `KeyPairTrait` for Secp256r1 Curve

> ⚠️ **Security Warning**
>
> Please note that cryptography in Starknet Foundry is still experimental and **has not been audited**.
>
> Use at your own risk!

`KeyPair`

```
struct KeyPair<SK, PK> {
    secret_key: SK,
    public_key: PK,
}
```

`KeyPairTrait`

```
trait KeyPairTrait<SK, PK> {
    fn generate() -> KeyPair<SK, PK>;
    fn from_secret_key(secret_key: SK) -> KeyPair<SK, PK>;
}
```

`SignerTrait`

```
trait SignerTrait<T, H, U> {
    fn sign(self: T, message_hash: H) -> Result<U, SignError> ;
}
```

`VerifierTrait`

```
trait VerifierTrait<T, H, U> {
    fn verify(self: T, message_hash: H, signature: U) -> bool;
}
```

Example

```
use snforge_std::signature::KeyPairTrait;

use snforge_std::signature::secp256r1_curve::{Secp256r1CurveKeyPairImpl,
Secp256r1CurveSignerImpl, Secp256r1CurveVerifierImpl};
use snforge_std::signature::secp256k1_curve::{Secp256k1CurveKeyPairImpl,
Secp256k1CurveSignerImpl, Secp256k1CurveVerifierImpl};
use snforge_std::signature::stark_curve::{StarkCurveKeyPairImpl,
StarkCurveSignerImpl, StarkCurveVerifierImpl};

use starknet::secp256r1::{Secp256r1Point, Secp256r1PointImpl};
use starknet::secp256k1::{Secp256k1Point, Secp256k1PointImpl};
use core::starknet::SyscallResultTrait;

#[test]
fn test_using_curves() {
    // Secp256r1
    let key_pair = KeyPairTrait::<u256, Secp256r1Point>::generate();
    let (r, s): (u256, u256) = key_pair.sign(msg_hash).unwrap();
    let is_valid = key_pair.verify(msg_hash, (r, s));

    // Secp256k1
    let key_pair = KeyPairTrait::<u256, Secp256k1Point>::generate();
    let (r, s): (u256, u256) = key_pair.sign(msg_hash).unwrap();
    let is_valid = key_pair.verify(msg_hash, (r, s));

    // StarkCurve
    let key_pair = KeyPairTrait::<felt252, felt252>::generate();
    let (r, s): (felt252, felt252) = key_pair.sign(msg_hash).unwrap();
    let is_valid = key_pair.verify(msg_hash, (r, s));
}
```

`sncast` CLI Reference

`sncast` common flags

`--profile, -p <PROFILE_NAME>`

Optional.

Used for both `snfoundry.toml` and `Scarb.toml` if specified. Defaults to `default` (`snfoundry.toml`) and `dev` (`Scarb.toml`).

`--account, -a <ACCOUNT_NAME>`

Optional.

Account name used to interact with the network, aliased in open zeppelin accounts file.

Overrides account from `snfoundry.toml`.

If used with `--keystore`, should be a path to starkli account JSON file.

`--accounts-file, -f <PATH_TO_ACCOUNTS_FILE>`

Optional.

Path to the open zeppelin accounts file holding accounts info. Defaults to `~/.starknet_accounts/starknet_open_zeppelin_accounts.json`.

`--keystore, -k <PATH_TO_KEYSTORE_FILE>`

Optional.

Path to keystore file. When specified, the --account argument must be a path to starkli account JSON file.

`--int-format`

Optional.

If passed, values will be displayed in decimal format. Default is addresses as hex and fees as int.

`--hex-format`

Optional.

If passed, values will be displayed in hex format. Default is addresses as hex and fees as int.

`--json, -j`

Optional.

If passed, output will be displayed in json format.

`--wait, -w`

Optional.

If passed, command will wait until transaction is accepted or rejected.

`--wait-timeout <TIME_IN_SECONDS>`

Optional.

If `--wait` is passed, this will set the time after which `sncast` times out. Defaults to 60s.

`--wait-retry-timeout <TIME_IN_SECONDS>`

Optional.

If `--wait` is passed, this will set the retry interval - how often `sncast` should fetch tx info from the node. Defaults to 5s.

`--version, -v`

Prints out `sncast` version.

`--help, -h`

Prints out help.

`account`

Provides a set of account management commands.

It has the following subcommands:

- `import`
- `create`
- `deploy`
- `delete`
- `list`

`import`

Import an account to accounts file.

Account information will be saved to the file specified by `--accounts-file` argument, which is `~/.starknet_accounts/starknet_open_zeppelin_accounts.json` by default.

`--name, -n <NAME>`

Optional.

Name of the account to be imported.

`--address, -a <ADDRESS>`

Required.

Address of the account.

`--type, -t <ACCOUNT_TYPE>`

Required.

Type of the account. Possible values: oz, argent, braavos.

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with public provider

Possible values: `mainnet`, `sepolia`.

`--class-hash, -c <CLASS_HASH>`

Optional.

Class hash of the account.

`--private-key <PRIVATE_KEY>`

Optional.

Account private key.

`--private-key-file <PRIVATE_KEY_FILE_PATH>`

Optional. If neither `--private-key` nor `--private-key-file` is passed, the user will be prompted to enter the account private key.

Path to the file holding account private key.

`--salt, -s <SALT>`

Optional.

Salt for the account address.

`--add-profile <NAME>`

Optional.

If passed, a profile with corresponding name will be added to the local snfoundry.toml.

`--silent`

Optional.

If passed, the command will not trigger an interactive prompt to add an account as a default

`create`

Prepare all prerequisites for account deployment.

Account information will be saved to the file specified by `--accounts-file` argument, which is `~/.starknet_accounts/starknet_open_zeppelin_accounts.json` by default.

`--name, -n <ACCOUNT_NAME>`

Required.

Account name under which account information is going to be saved.

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with a public provider

Possible values: `mainnet`, `sepolia`.

`--type, -t <ACCOUNT_TYPE>`

Optional. Required if `--class-hash` is passed.

Type of the account. Possible values: oz, argent, braavos. Defaults to oz.

Versions of the account contracts:

Account Contract	Version	Class Hash
`oz`	v0.14.0	0x00e2eb8f5672af4e6a4e8a8f1b44989685e668489b0a25437733
`argent`	v0.3.1	0x029927c8af6bccf3f6fda035981e765a7bdbf18a2dc0d630494f8
`braavos`	v1.0.0	0x00816dd0297efc55dc1e7559020a3a825e81ef734b558f03c833

`--salt, -s <SALT>`

Optional.

Salt for the account address. If omitted random one will be generated.

`--add-profile <NAME>`

Optional.

If passed, a profile with corresponding name will be added to the local snfoundry.toml.

`--class-hash, -c`

Optional.

Class hash of a custom openzeppelin account contract declared to the network.

`--silent`

Optional.

If passed, the command will not trigger an interactive prompt to add an account as a default

`deploy`

Deploy previously created account to Starknet.

`--name, -n <ACCOUNT_NAME>`

Required.

Name of the (previously created) account to be deployed.

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with a public provider

Possible values: `mainnet`, `sepolia`.

`--max-fee, -m <MAX_FEE>`

Optional.

Maximum fee for the `deploy_account` transaction in Fri or Wei depending on fee token or transaction version. When not used, defaults to auto-estimation. Must be greater than zero.

`--max-gas <MAX_GAS>`

Optional.

Maximum gas for the `deploy_account` transaction. When not used, defaults to auto-estimation. Must be greater than zero. (Only for STRK fee payment)

`--max-gas-unit-price <MAX_GAS_UNIT_PRICE>`

Optional.

Maximum gas unit price for the `deploy_account` transaction paid in Fri. When not used, defaults to auto-estimation. Must be greater than zero. (Only for STRK fee payment)

`delete`

Delete an account from `accounts-file` and its associated snfoundry profile.

`--name, -n <ACCOUNT_NAME>`

Required.

Account name which is going to be deleted.

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with a public provider

Possible values: `mainnet`, `sepolia`.

`--network-name`

Optional.

Network in `accounts-file` associated with the account. By default, the network passed as `--network` of RPC node.

`--yes`

Optional.

If passed, assume "yes" as answer to confirmation prompt and run non-interactively

`list`

List all available accounts.

Account information will be retrieved from the file specified in user's environment. The output format is dependent on user's configuration, either provided via CLI or specified in `snfoundry.toml`. Hides user's private keys by default.

⚠️ **Warning** This command outputs cryptographic information about accounts, e.g. user's private key. Use it responsibly to not cause any vulnerabilities to your environment and confidential data.

Required Common Arguments — Passed By CLI or Specified in `snfoundry.toml`

- `accounts-file`

Optional Common Arguments — Passed By CLI or Specified in `snfoundry.toml`

- `int-format`
- `hex-format`
- `json`

`--display-private-keys`, `-p`

Optional.

If passed, show private keys along with the rest of the account information.

`declare`

Send a declare transaction of Cairo contract to Starknet.

Required Common Arguments — Passed By CLI or Specified in `snfoundry.toml`

- `account`

`--contract-name, -c <CONTRACT_NAME>`

Required.

Name of the contract. Contract name is a part after the mod keyword in your contract file.

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with public provider

Possible values: `mainnet`, `sepolia`.

`--max-fee, -m <MAX_FEE>`

Optional.

Maximum fee for the `declare` transaction in Fri or Wei depending on fee token or transaction version. When not used, defaults to auto-estimation. Must be greater than zero.

`--max-gas <MAX_GAS>`

Optional.

Maximum gas for the `declare` transaction. When not used, defaults to auto-estimation. Must be greater than zero. (Only for STRK fee payment)

`--max-gas-unit-price <MAX_GAS_UNIT_PRICE>`

Optional.

Maximum gas unit price for the `declare` transaction paid in Fri. When not used, defaults to auto-estimation. Must be greater than zero. (Only for STRK fee payment)

`--nonce, -n <NONCE>`

Optional.

Nonce for transaction. If not provided, nonce will be set automatically.

`--package <NAME>`

Optional.

Name of the package that should be used.

If supplied, a contract from this package will be used. Required if more than one package exists in a workspace.

`deploy`

Deploy a contract to Starknet.

Required Common Arguments — Passed By CLI or Specified in `snfoundry.toml`

- `account`

`--class-hash, -g <CLASS_HASH>`

Required.

Class hash of contract to deploy.

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with public provider

Possible values: `mainnet`, `sepolia`.

`--constructor-calldata, -c <CONSTRUCTOR_CALLDATA>`

Optional.

Calldata for the contract constructor.

`--salt, -s <SALT>`

Optional.

Salt for the contract address.

`--unique`

Optional.

If passed, the salt will be additionally modified with an account address.

`--max-fee, -m <MAX_FEE>`

Optional.

Maximum fee for the `deploy` transaction in Fri or Wei depending on fee token or transaction version. When not used, defaults to auto-estimation. Must be greater than zero.

`--max-gas <MAX_GAS>`

Optional.

Maximum gas for the `deploy` transaction. When not used, defaults to auto-estimation. Must be greater than zero. (Only for STRK fee payment)

`--max-gas-unit-price <MAX_GAS_UNIT_PRICE>`

Optional.

Maximum gas unit price for the `deploy` transaction paid in Fri. When not used, defaults to auto-estimation. Must be greater than zero. (Only for STRK fee payment)

`--nonce, -n <NONCE>`

Optional.

Nonce for transaction. If not provided, nonce will be set automatically.

`invoke`

Send an invoke transaction to Starknet.

Required Common Arguments — Passed By CLI or Specified in `snfoundry.toml`

- `account`

`--contract-address, -a <CONTRACT_ADDRESS>`

Required.

The address of the contract being called in hex (prefixed with '0x') or decimal representation.

`--function, -f <FUNCTION_NAME>`

Required.

The name of the function to call.

`--calldata, -c <CALLDATA>`

Optional.

Inputs to the function, represented by a list of space-delimited values `0x1 2 0x3`. Calldata arguments may be either 0x hex or decimal felts.

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with public provider

Possible values: `mainnet`, `sepolia`.

`--max-fee, -m <MAX_FEE>`

Optional.

Maximum fee for the `invoke` transaction in Fri or Wei depending on fee token or transaction version. When not used, defaults to auto-estimation. Must be greater than zero.

`--max-gas <MAX_GAS>`

Optional.

Maximum gas for the `invoke` transaction. When not used, defaults to auto-estimation. Must be greater than zero. (Only for STRK fee payment)

`--max-gas-unit-price <MAX_GAS_UNIT_PRICE>`

Optional.

Maximum gas unit price for the `invoke` transaction paid in Fri. When not used, defaults to auto-estimation. Must be greater than zero. (Only for STRK fee payment)

`--nonce, -n <NONCE>`

Optional.

Nonce for transaction. If not provided, nonce will be set automatically.

`call`

Call a smart contract on Starknet with the given parameters.

`--contract-address, -a <CONTRACT_ADDRESS>`

Required.

The address of the contract being called in hex (prefixed with '0x') or decimal representation.

`--function, -f <FUNCTION_NAME>`

Required.

The name of the function being called.

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with public provider

Possible values: `mainnet`, `sepolia`.

`--calldata, -c <CALLDATA>`

Optional.

Inputs to the function, represented by a list of space-delimited values, e.g. `0x1 2 0x3`. Calldata arguments may be either 0x hex or decimal felts.

`--block-id, -b <BLOCK_ID>`

Optional.

Block identifier on which call should be performed. Possible values: `pending`, `latest`, block hash (0x prefixed string), and block number (u64). `pending` is used as a default value.

`multicall`

Provides utilities for performing multicalls on Starknet.

Multicall has the following subcommands:

- `new`
- `run`

`new`

Generates an empty template for the multicall `.toml` file that may be later used with the `run` subcommand. It writes it to a file provided as a required argument.

Usage

`multicall new <OUTPUT-PATH> [OPTIONS]`

Arguments

`OUTPUT-PATH` - a path to a file to write the generated `.toml` to.

`--output-path, -p <PATH>`

Optional.

Specifies a file path where the template should be saved. If omitted, the template contents will be printed out to the stdout.

`--overwrite, -o <OVERWRITE>`

Optional.

If the file specified by `--output-path` already exists, this parameter overwrites it.

`run`

Execute a single multicall transaction containing every call from passed file.

Required Common Arguments — Passed By CLI or Specified in `snfoundry.toml`

- `account`

`--path, -p <PATH>`

Required.

Path to a TOML file with call declarations.

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with public provider

Possible values: `mainnet`, `sepolia`.

`--max-fee, -m <MAX_FEE>`

Optional.

Maximum fee for the `invoke` transaction in Fri or Wei depending on fee token or transaction version. When not used, defaults to auto-estimation. Must be greater than zero.

`--max-gas <MAX_GAS>`

Optional.

Maximum gas for the `invoke` transaction. When not used, defaults to auto-estimation. Must be greater than zero. (Only for STRK fee payment)

`--max-gas-unit-price <MAX_GAS_UNIT_PRICE>`

Optional.

Maximum gas unit price for the `invoke` transaction paid in Fri. When not used, defaults to auto-estimation. Must be greater than zero. (Only for STRK fee payment)

File example:

```
[[call]]
call_type = "deploy"
class_hash = "0x076e94149fc55e7ad9c5fe3b9af570970ae2cf51205f8452f39753e9497fe849"
inputs = []
id = "map_contract"
unique = false

[[call]]
call_type = "invoke"
contract_address =
"0x38b7b9507ccf73d79cb42c2cc4e58cf3af1248f342112879bfdf5aa4f606cc9"
function = "put"
inputs = ["0x123", "234"]

[[call]]
call_type = "invoke"
contract_address = "map_contract"
function = "put"
inputs = ["0x123", "234"]

[[call]]
call_type = "deploy"
class_hash = "0x2bb3d35dba2984b3d0cd0901b4e7de5411daff6bff5e072060bcfadbbd257b1"
inputs = ["0x123", "map_contract"]
unique = false
```

`show_config`

Prints the config currently being used

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with public provider

Possible values: `mainnet`, `sepolia`.

`script`

Provides a set of commands to manage deployment scripts.

Script has the following subcommands:

- `init`
- `run`

`init`

Create a deployment script template.

The command creates the following file and directory structure:

```
.
└── scripts
    └── my_script
        ├── Scarb.toml
        └── src
            ├── lib.cairo
            └── my_script.cairo
```

`<SCRIPT_NAME>`

Required.

Name of a script to create.

`run`

Compile and run a cairo deployment script.

Required Common Arguments — Passed By CLI or Specified in `snfoundry.toml`

- `account`

`<MODULE_NAME>`

Required.

Script module name that contains the 'main' function that will be executed.

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with public provider

Possible values: `mainnet`, `sepolia`.

`--package <NAME>`

Optional.

Name of the package that should be used.

If supplied, a script from this package will be used. Required if more than one package exists in a workspace.

`--no-state-file`

Optional.

Do not read/write state from/to the state file.

If set, a script will not read the state from the state file, and will not write a state to it.

`tx-status`

Get the status of a transaction

`<TRANSACTION_HASH>`

Required.

Hash of the transaction

`--url, -u <RPC_URL>`

Optional.

Starknet RPC node url address.

Overrides url from `snfoundry.toml`.

`--network <NETWORK>`

Optional.

Use predefined network with public provider

Possible values: `mainnet`, `sepolia`.

`verify`

Verify Cairo contract on a chosen verification provider.

`--contract-address, -a <CONTRACT_ADDRESS>`

Required.

The address of the contract that is to be verified.

`--contract-name <CONTRACT_NAME>`

Required.

The name of the contract. The contract name is the part after the `mod` keyword in your contract file.

`--verifier, -v <VERIFIER>`

Optional.

The verification provider to use for the verification. Possible values are:

- `walnut`

`--network, -n <NETWORK>`

Required.

The network on which block explorer will perform the verification. Possible values are:

- `mainnet`
- `sepolia`

`--package <NAME>`

Optional.

Name of the package that should be used.

If supplied, a contract from this package will be used. Required if more than one package exists in a workspace.

`--confirm-verification`

Optional.

If passed, assume "yes" as answer to confirmation prompt and run non-interactively.

Library Reference

> **Info** Full documentation for the `sncast` library can be found here.

- `declare` - declares a contract
- `deploy` - deploys a contract
- `invoke` - invokes a contract's function
- `call` - calls a contract's function
- `get_nonce` - gets account's nonce for a given block tag
- `tx_status` - gets the status of a transaction using its hash
- `errors` - sncast_std error types reference

> **Info** To use the library functions you need to add `sncast_std` package as a dependency in your `Scarb.toml` using the appropriate version.

```
[dependencies]
sncast_std = "0.33.0"
```

`declare`

`pub fn declare(contract_name: ByteArray, fee_settings: FeeSettings, nonce: Option<felt252>) -> Result<DeclareResult, ScriptCommandError>`

Declares a contract and returns `DeclareResult`.

- `contract_name` - name of a contract as Cairo string. It is a name of the contract (part after `mod` keyword) e.g. `"HelloStarknet"`.
- `fee_settings` - fee settings for the transaction.
- `nonce` - nonce for declare transaction. If not provided, nonce will be set automatically.

```
use sncast_std::{declare, DeclareResult, FeeSettings};

fn main() {
    let max_fee = 9999999;

    let result = declare(
        "HelloStarknet",
        FeeSettings {
            max_fee: Option::Some(max_fee), max_gas: Option::None,
max_gas_unit_price: Option::None
        },
        Option::None
    )
        .expect('declare failed');

    println!("declare result: {}", result);
    println!("debug declare result: {:?}", result);
}
```

Returned Type

- If the contract has not been declared, `DeclareResult::Success` is returned containing respective transaction hash.
- If the contract has already been declared, `DeclareResult::AlreadyDeclared` is returned.

Getting the Class Hash

Both variants contain `class_hash` of the declared contract. Import `DeclareResultTrait` to access it.

```
pub trait DeclareResultTrait {
    fn class_hash(self: @DeclareResult) -> @ClassHash;
}
```

Structures Used by the Command

```
#[derive(Drop, Copy, Debug, Serde)]
pub enum DeclareResult {
    Success: DeclareTransactionResult,
    AlreadyDeclared: AlreadyDeclaredResult,
}

#[derive(Drop, Copy, Debug, Serde)]
pub struct DeclareTransactionResult {
    pub class_hash: ClassHash,
    pub transaction_hash: felt252,
}

#[derive(Drop, Copy, Debug, Serde)]
pub struct AlreadyDeclaredResult {
    pub class_hash: ClassHash,
}
#[derive(Drop, Copy, Debug, Serde, PartialEq)]
pub struct FeeSettings {
    pub max_fee: Option<felt252>,
    pub max_gas: Option<u64>,
    pub max_gas_unit_price: Option<u128>,
}
```

deploy

```
pub fn deploy( class_hash: ClassHash, constructor_calldata: Array::<felt252>, salt: Option<felt252>, unique: bool, fee_settings: FeeSettings, nonce: Option<felt252> ) -> Result<DeployResult, ScriptCommandError>
```

Deploys a contract and returns `DeployResult`.

```
#[derive(Drop, Clone, Debug)]
pub struct DeployResult {
    pub contract_address: ContractAddress,
    pub transaction_hash: felt252,
}

#[derive(Drop, Copy, Debug, Serde, PartialEq)]
pub struct FeeSettings {
    pub max_fee: Option<felt252>,
    pub max_gas: Option<u64>,
    pub max_gas_unit_price: Option<u128>,
}
```

- `class_hash` - class hash of a contract to deploy.
- `constructor_calldata` - calldata for the contract constructor.
- `salt` - salt for the contract address.
- `unique` - determines if salt should be further modified with the account address.
- `fee_settings` - fee settings for the transaction.
- `nonce` - nonce for declare transaction. If not provided, nonce will be set automatically.

```
use starknet::ClassHash;
use sncast_std::{deploy, DeployResult, FeeSettings};

fn main() {
    let max_fee = 9999999;
    let salt = 0x1;
    let nonce = 0x1;

    let class_hash: ClassHash =
0x03a8b191831033ba48ee176d5dde7088e71c853002b02a1cfa5a760aa98be046
        .try_into()
        .expect('Invalid class hash value');

    let result = deploy(
        class_hash,
        ArrayTrait::new(),
        Option::Some(salt),
        true,
        FeeSettings {
            max_fee: Option::Some(max_fee), max_gas: Option::None,
max_gas_unit_price: Option::None
        },
        Option::Some(nonce)
    )
        .expect('deploy failed');

    println!("deploy result: {}", result);
    println!("debug deploy result: {:?}", result);
}
```

invoke

```
pub fn invoke( contract_address: ContractAddress, entry_point_selector: felt252, calldata: Array::<felt252>, fee_settings: FeeSettings, nonce: Option<felt252> ) -> Result<InvokeResult, ScriptCommandError>
```

Invokes a contract and returns `InvokeResult`.

- `contract_address` - address of the contract to invoke.
- `entry_point_selector` - the selector of the function to invoke.
- `calldata` - inputs to the function to be invoked.
- `fee_settings` - fee settings for the transaction.
- `nonce` - nonce for declare transaction. If not provided, nonce will be set automatically.

```
use starknet::ContractAddress;
use sncast_std::{invoke, InvokeResult, FeeSettings};

fn main() {
    let contract_address: ContractAddress =
        0x1e52f6ebc3e594d2a6dc2a0d7d193cb50144cfdfb7fdd9519135c29b67e427
        .try_into()
        .expect('Invalid contract address value');

    let result = invoke(
        contract_address,
        selector!("put"),
        array![0x1, 0x2],
        FeeSettings {
            max_fee: Option::None, max_gas: Option::None, max_gas_unit_price: Option::None
        },
        Option::None
    )
        .expect('invoke failed');

    println!("invoke result: {}", result);
    println!("debug invoke result: {:?}", result);
}
```

Structures used by the command:

```rust
#[derive(Drop, Clone, Debug)]
pub struct InvokeResult {
    pub transaction_hash: felt252,
}

#[derive(Drop, Copy, Debug, Serde, PartialEq)]
pub struct FeeSettings {
    pub max_fee: Option<felt252>,
    pub max_gas: Option<u64>,
    pub max_gas_unit_price: Option<u128>,
}
```

call

```
pub fn call( contract_address: ContractAddress, function_selector: felt252,
calldata: Array::<felt252> ) -> Result<CallResult, ScriptCommandError>
```

Calls a contract and returns `CallResult`.

- `contract_address` - address of the contract to call.
- `function_selector` - the selector of the function to call.
- `calldata` - inputs to the function to be called.

```
use sncast_std::{call, CallResult};
use starknet::ContractAddress;

fn main() {
    let contract_address: ContractAddress =
        0x1e52f6ebc3e594d2a6dc2a0d7d193cb50144cfdfb7fdd9519135c29b67e427
        .try_into()
        .expect('Invalid contract address value');

    let result = call(contract_address, selector!("get"), array!
[0x1]).expect('call failed');

    println!("call result: {}", result);
    println!("debug call result: {:?}", result);
}
```

Structure used by the command:

```
#[derive(Drop, Clone, Debug)]
pub struct CallResult {
    pub data: Array::<felt252>,
}
```

get_nonce

```
pub fn get_nonce(block_tag: felt252) -> felt252
```

Gets nonce of an account for a given block tag (`pending` or `latest`) and returns nonce as `felt252` .

- `block_tag` - block tag name, one of `pending` or `latest` .

```
use sncast_std::get_nonce;

fn main() {
    let nonce = get_nonce('latest');
    println!("nonce: {}", nonce);
    println!("debug nonce: {:?}", nonce);
}
```

tx_status

```
pub fn tx_status(transaction_hash: felt252) -> Result<TxStatusResult, ScriptCommandError>
```

Gets the status of a transaction using its hash and returns `TxStatusResult`.

- `transaction_hash` - hash of the transaction

```
use sncast_std::{tx_status};

fn main() {
    let transaction_hash = 0x00ae35dacba17cde62b8ceb12e3b18f4ab6e103fa2d5e3d9821cb9dc59d59a3c;
    let status = tx_status(transaction_hash).expect('Failed to get status');

    println!("transaction status: {:?}", status);
}
```

Structures used by the command:

```
#[derive(Drop, Clone, Debug, Serde, PartialEq)]
pub enum FinalityStatus {
    Received,
    Rejected,
    AcceptedOnL2,
    AcceptedOnL1
}


#[derive(Drop, Copy, Debug, Serde, PartialEq)]
pub enum ExecutionStatus {
    Succeeded,
    Reverted,
}


#[derive(Drop, Clone, Debug, Serde, PartialEq)]
pub struct TxStatusResult {
    pub finality_status: FinalityStatus,
    pub execution_status: Option<ExecutionStatus>
}
```

errors

```
#[derive(Drop, PartialEq, Serde, Debug)]
pub struct ErrorData {
    msg: ByteArray
}

#[derive(Drop, PartialEq, Serde, Debug)]
pub struct TransactionExecutionErrorData {
    transaction_index: felt252,
    execution_error: ByteArray,
}

#[derive(Drop, Serde, PartialEq, Debug)]
pub enum StarknetError {
    /// Failed to receive transaction
    FailedToReceiveTransaction,
    /// Contract not found
    ContractNotFound,
    /// Block not found
    BlockNotFound,
    /// Invalid transaction index in a block
    InvalidTransactionIndex,
    /// Class hash not found
    ClassHashNotFound,
    /// Transaction hash not found
    TransactionHashNotFound,
    /// Contract error
    ContractError: ErrorData,
    /// Transaction execution error
    TransactionExecutionError: TransactionExecutionErrorData,
    /// Class already declared
    ClassAlreadyDeclared,
    /// Invalid transaction nonce
    InvalidTransactionNonce,
    /// Max fee is smaller than the minimal transaction cost (validation plus fee
transfer)
    InsufficientMaxFee,
    /// Account balance is smaller than the transaction's max_fee
    InsufficientAccountBalance,
    /// Account validation failed
    ValidationFailure: ErrorData,
    /// Compilation failed
    CompilationFailed,
    /// Contract class size it too large
    ContractClassSizeIsTooLarge,
    /// Sender address in not an account contract
    NonAccount,
    /// A transaction with the same hash already exists in the mempool
    DuplicateTx,
    /// the compiled class hash did not match the one supplied in the transaction
    CompiledClassHashMismatch,
```

```
    /// the transaction version is not supported
    UnsupportedTxVersion,
    /// the contract class version is not supported
    UnsupportedContractClassVersion,
    /// An unexpected error occurred
    UnexpectedError: ErrorData,
}

#[derive(Drop, Serde, PartialEq, Debug)]
pub enum ProviderError {
    StarknetError: StarknetError,
    RateLimited,
    UnknownError: ErrorData,
}

#[derive(Drop, Serde, PartialEq, Debug)]
pub enum TransactionError {
    Rejected,
    Reverted: ErrorData,
}

#[derive(Drop, Serde, PartialEq, Debug)]
pub enum WaitForTransactionError {
    TransactionError: TransactionError,
    TimedOut,
    ProviderError: ProviderError,
}

#[derive(Drop, Serde, PartialEq, Debug)]
pub enum ScriptCommandError {
    UnknownError: ErrorData,
    ContractArtifactsNotFound: ErrorData,
    WaitForTransactionError: WaitForTransactionError,
    ProviderError: ProviderError,
}
```

The Manifest Format

The `snfoundry.toml` contains the project's manifest and allows specifying sncast settings. You can configure sncast settings and arguments instead of providing them in the CLI along with the commands. If `snfoundry.toml` is not found in the root directory, sncast will look for it in all parent directories. If it is not found, default values will be used.

`snfoundry.toml` Contents

`[sncast.<profile-name>]`

```toml
[sncast.myprofile]
# ...
```

All fields are optional and do not have to be provided. In case a field is not defined in a manifest file, it must be provided in CLI when executing a relevant `sncast` command. Profiles allow you to define different sets of configurations for various environments or use cases. For more details, see the profiles explanation.

The `url` field specifies the address of RPC provider.

```toml
[sncast.myprofile]
url = "http://example.com"
```

`accounts-file`

The `accounts-file` field specifies the path to a file containing account information. If not provided, the default path is `~/.starknet_accounts/starknet_open_zeppelin_accounts.json`.

```toml
[sncast.myprofile]
accounts-file = "path/to/accounts.json"
```

`account`

The `account` field specifies which account from the `accounts-file` to use for transactions.

```
[sncast.myprofile]
account = "user-dev"
```

keystore

The `keystore` field specifies the path to the keystore file.

```
[sncast.myprofile]
keystore = "path/to/keystore"
```

wait-params

The `wait-params` field defines the waiting parameters for transactions. By default, timeout (in seconds) is set to `300` and retry-interval (in seconds) to `5`. This means transactions will be checked every `5 seconds`, with a total of `60 attempts` before timing out.

```
[sncast.myprofile]
wait-params = { timeout = 300, retry-interval = 5 }
```

show-explorer-links

Enable printing links pointing to pages with transaction details in the chosen block explorer

```
[sncast.myprofile]
show-explorer-links = true
```

block-explorer

The `block-explorer` field specifies the block explorer service used to display links to transaction details.

Value	URL
StarkScan	`https://starkscan.co`
Voyager	`https://voyager.online`
ViewBlock	`https://viewblock.io/starknet`
OkLink	`https://www.oklink.com/starknet`
NftScan	`https://starknet.nftscan.com`

```
[sncast.myprofile]
block-explorer = "StarkScan"
```

Complete Example of `snfoundry.toml` File

```
[sncast.myprofile1]
url = "http://127.0.0.1:5050/"
accounts-file = "../account-file"
account = "mainuser"
keystore = "~/keystore"
wait-params = { timeout = 500, retry-interval = 10 }
block-explorer = "StarkScan"
show-explorer-links = true

[sncast.dev]
url = "http://127.0.0.1:5056/rpc"
account = "devuser"
```

The Manifest Format

The `Scarb.toml` contains the package manifest that is needed in package compilation process. It can be used to provide configuration for Starknet Foundry Forge. For more, see official Scarb documentation.

`Scarb.toml` Contents

`[tool.snforge]`

```
[tool.snforge]
# ...
```

Allows to configure `snforge` settings. All fields are optional.

`exit_first`

The `exit_first` fields specifies whether to stop tests execution immediately upon the first failure. See more about stopping test execution after first failed test.

```
[tool.snforge]
exit_first = true
```

`fuzzer_runs`

The `fuzzer_runs` field specifies the number of runs of the random fuzzer.

`fuzzer_seed`

The `fuzzer_seed` field specifies the seed for the random fuzzer.

See more about fuzzer.

Example of fuzzer configuration

```
[tool.snforge]
fuzzer_runs = 1234
fuzzer_seed = 1111
```

[[tool.snforge.fork]]

```
[[tool.snforge.fork]]
# ...
```

Allows to configure forked tests. If defined, all fields outlined below must also be defined. See more about fork testing.

name

The `name` field specifies the name of the fork.

```
[[tool.snforge.fork]]
name = "SOME_NAME"
```

url

The `url` field specifies the address of RPC provider.

```
[[tool.snforge.fork]]
url = "http://your.rpc.url"
```

block_id.<tag|number|hash>

The `block_id` field specifies the block to fork from. It can be specified by `tag`, `number` or `hash`.

```
[[tool.snforge.fork]]
block_id.hash = "0x123"
```

Example configuration with two forks

```
[[tool.snforge.fork]]
name = "SOME_NAME"
url = "http://your.rpc.url"
block_id.tag = "latest"

[[tool.snforge.fork]]
name = "SOME_SECOND_NAME"
url = "http://your.second.rpc.url"
block_id.number = "123"
```

`[tool.scarb]`

```
[tool.scarb]
allow-prebuilt-plugins = ["snforge_std"]
```

Note: This configuration requires Scarb version >= 2.10.0 .

It allows `scarb` to download precompiled dependencies used by `snforge_std` from the registry. The `snforge_std` library depends on a Cairo plugin that is written in Rust, and otherwise is compiled locally on the user's side.

`[profile.<dev|release>.cairo]`

By default, these arguments do not need to be defined. Only set them to use profiler or coverage.

Adjust Cairo compiler configuration parameters when compiling this package. These options are not taken into consideration when this package is used as a dependency for another package. All fields are optional.

```
[profile.dev.cairo]
# ...
```

`unstable-add-statements-code-locations-debug-info`

See `unstable-add-statements-code-locations-debug-info` in Scarb documentation.

```
[profile.dev.cairo]
unstable-add-statements-code-locations-debug-info = true
```

`unstable-add-statements-functions-debug-info`

See `unstable-add-statements-functions-debug-info` in Scarb documentation.

```
[profile.dev.cairo]
unstable-add-statements-functions-debug-info = true
```

`inlining-strategy`

See `inlining-strategy` in Scarb documentation.

```
[profile.dev.cairo]
inlining-strategy = "avoid"
```

Example of configuration which allows coverage report generation

```
[profile.dev.cairo]
unstable-add-statements-code-locations-debug-info = true
unstable-add-statements-functions-debug-info = true
inlining-strategy = "avoid"
```

`[features]`

A package defines a set of named features in the `[features]` section of `Scarb.toml` file. Each defined feature can list other features that should be enabled with it. All fields are optional.

```
[features]
# ...
```

`<feature-name>`

The `<feature-name>` field specifies the name of the feature and list of other features that should be enabled with it. See features in Scarb documentation.

```
[features]
enable_for_tests = []
```

Example of `Scarb.toml` allowing conditional contracts compilation

Firstly, define a contract in the src directory with a `#[cfg(feature: '<FEATURE_NAME>')]` attribute:

```
#[starknet::contract]
#[cfg(feature: 'enable_for_tests')]
mod MockContract {
    // ...
}
```

Then update Scarb.toml so it includes the following lines:

```
[features]
enable_for_tests = []
```

`[[target.starknet-contract]]`

The `starknet-contract` target allows to build the package as a Starknet Contract. See more about Starknet Contract Target in Scarb documentation.

```
[[target.starknet-contract]]
# ...
```

`sierra`

See more about Sierra contract class generation in Scarb documentation.

```
[[target.starknet-contract]]
sierra = true
```

`casm`

Enabling `casm = true` in Scarb.toml causes unnecessary overhead and should be disabled unless required by other tools. Tools like `snforge` and `sncast` recompile Sierra to CASM separately, resulting in redundant processing. This duplicates CASM generation, significantly impacting performance, especially for large Sierra programs. See more about CASM contract class generation in Scarb documentation.

```
[[target.starknet-contract]]
casm = true
```

`build-external-contracts`

The `build-external-contracts` allows to use contracts from your dependencies inside your tests. It accepts a list of strings, each of which is a reference to a contract defined in a dependency. You need to add dependency which implements this contracts to your Scarb.toml. See more about compiling external contracts in Scarb documentation.

```
[[target.starknet-contract]]
build-external-contracts = ["openzeppelin::account::account::Account"]
```

Example of configuration which allows to use external contracts in tests

```
# ...
[dependencies]
starknet = ">=2.8.2"
openzeppelin = { git = "https://github.com/OpenZeppelin/cairo-contracts.git",
branch = "cairo-2" }

[[target.starknet-contract]]
build-external-contracts = ["openzeppelin::account::account::Account"]
# ...
```

Complete example of `Scarb.toml`

```toml
[package]
name = "example_package"
version = "0.1.0"
edition = "2023_11"

# See more keys and their definitions at
https://docs.swmansion.com/scarb/docs/reference/manifest.html

[dependencies]
starknet = "2.8.2"

[dev-dependencies]
snforge_std = "0.33.0"
starknet = ">=2.8.2"
openzeppelin = { git = "https://github.com/OpenZeppelin/cairo-contracts.git",
branch = "cairo-2" }

[[target.starknet-contract]]
sierra = true
build-external-contracts = ["openzeppelin::account::account::Account"]

[scripts]
test = "snforge test"
# foo = { path = "vendor/foo" }

[tool.snforge]
exit_first = true
fuzzer_runs = 1234
fuzzer_seed = 1111

[[tool.snforge.fork]]
name = "SOME_NAME"
url = "http://your.rpc.url"
block_id.tag = "latest"

[[tool.snforge.fork]]
name = "SOME_SECOND_NAME"
url = "http://your.second.rpc.url"
block_id.number = "123"

[[tool.snforge.fork]]
name = "SOME_THIRD_NAME"
url = "http://your.third.rpc.url"
block_id.hash = "0x123"

[profile.dev.cairo]
unstable-add-statements-code-locations-debug-info = true
unstable-add-statements-functions-debug-info = true
inlining-strategy = "avoid"
```

```
[features]
enable_for_tests = []
```

Starknet Foundry Github Action

If you wish to use Starknet Foundry in your Github Actions workflow, you can use the setup-snfoundry action. This action installs the necessary `snforge` and `sncast` binaries.

📝 **Note** At this moment, only Linux and MacOS are supported.

Example workflow

Make sure you pass the valid path to `Scarb.lock` to setup-scarb action. This way, all dependencies including snforge_scarb_plugin will be cached between runs.

```yaml
name: My workflow
on:
  push:
  pull_request:
jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup Starknet Foundry
        uses: foundry-rs/setup-snfoundry@v3

      - name: Setup Scarb
        uses: software-mansion/setup-scarb@v1
        with:
          scarb-lock: ./hello_starknet/Scarb.lock

      - name: Run tests
        run: cd hello_starknet && snforge test
```