Testing Smart Contracts

Testing smart contracts is a critical part of the development process. It is important to ensure that smart contracts behave as expected and that they are secure.

In a previous section of the Cairo Book, we learned how to write and structure our tests for Cairo programs. We demonstrated how these tests could be run using the `scarb` command-line tool.

While this approach is useful for testing standalone Cairo programs and functions, it lacks functionality for testing smart contracts that require control over the contract state and execution context. Therefore, in this section, we will introduce how to use Starknet Foundry, a smart contract development toolchain for Starknet, to test your Cairo contracts.

Throughout this chapter, we will be using as an example the `PizzaFactory` contract in Listing {{#ref pizza-factory}} to demonstrate how to write tests with Starknet Foundry.

```cairo,noplayground
{{#rustdoc_include ../listings/ch17-starknet-smart-contracts-security/
listing_02_pizza_factory_snfoundry/src/pizza.cairo}}
```

{{#label pizza-factory}}
Listing {{#ref pizza-factory}}: A pizza factory that needs to be tested

Configuring your Scarb project with Starknet Foundry

The settings of your Scarb project can be configured in the `Scarb.toml` file. To use Starknet Foundry as your testing tool, you will need to add it as a dev dependency in your `Scarb.toml` file. At the time of writing, the latest version of Starknet Foundry is `v0.22.0` - but you should use the latest version.

```toml,noplayground
[dev-dependencies]
snforge_std = { git = "https://github.com/foundry-rs/starknet-foundry.git", tag = "v0.22.0" }
[scripts]
test = "snforge test"
```

The `scarb test` command is configured to execute `scarb cairo-test` by default. In our settings, we have configured it to execute `snforge test` instead. This will allow us to run our tests using Starknet Foundry when we run the `scarb test` command.

Once your project is configured, you will need to install Starknet Foundry by following the installation guide from the [Starknet Foundry Documentation](https://foundry-rs.github.io/starknet-foundry/getting-started/installation.html). As usual, we recommend to use `asdf` to manage versions of your development tools.

Testing Smart Contracts with Starknet Foundry

The usual command to run your tests using Starknet Foundry is `snforge test`. However, when we configured our projects, we defined that the `scarb test` command will run the `snforge test` command. Therefore, during the rest of this chapter, consider that the `scarb test` command will be using `snforge test` under the hood.

The usual testing flow of a contract is as follows:

1. Declare the class of the contract to test, identified by its name
2. Serialize the constructor calldata into an array

3. Deploy the contract and retrieve its address
4. Interact with the contract's entrypoint to test various scenarios
Deploying the Contract to Test
In Listing {{#ref contract-deployment}}, we wrote a function that deploys the `PizzaFactory` contract and sets up the dispatcher for interactions.
```cairo,noplayground
{{#rustdoc_include ../listings/ch17-starknet-smart-contracts-security/listing_02_pizza_factory_snfoundry/src/tests/foundry_test.cairo:deployment}}
```
{{#label contract-deployment}}
Listing {{#ref contract-deployment}} Deploying the contract to test
Testing our Contract
Determining the behavior that your contract should respect is the first step in writing tests. In the `PizzaFactory` contract, we determined that the contract should have the following behavior:
- Upon deployment, the contract owner should be set to the address provided in the constructor, and the factory should have 10 units of pepperoni and pineapple, and no pizzas created.
- If someone tries to make a pizza and they are not the owner, the operation should fail. Otherwise, the pizza count should be incremented, and an event should be emitted.
- If someone tries to take ownership of the contract and they are not the owner, the operation should fail. Otherwise, the owner should be updated.
Accessing Storage Variables with `load`
```cairo,noplayground
{{#rustdoc_include ../listings/ch17-starknet-smart-contracts-security/listing_02_pizza_factory_snfoundry/src/tests/foundry_test.cairo:test_constructor}}
```
{{#label test-constructor}}
Listing {{#ref test-constructor}}: Testing the initial state by loading storage variables
Once our contract is deployed, we want to assert that the initial values are set as expected. If our contract has an entrypoint that returns the value of a storage variable, we can call this entrypoint. Otherwise, we can use the `load` function from `snforge` to load the value of a storage variable inside our contract, even if not exposed by an entrypoint.
Mocking the Caller Address with `start_cheat_caller_address`
The security of our factory relies on the owner being the only one able to make pizzas and transfer ownership. To test this, we can use the `start_cheat_caller_address` function to mock the caller address and assert that the contract behaves as expected.
```cairo,noplayground
{{#rustdoc_include ../listings/ch17-starknet-smart-contracts-security/listing_02_pizza_factory_snfoundry/src/tests/foundry_test.cairo:test_owner}}
```
{{#label test-owner}}
Listing {{#ref test-owner}}: Testing ownership of the contract by

mocking the caller address
Using `start_cheat_caller_address`, we call the `change_owner` function first as the owner, and then as a different address. We assert that the operation fails when the caller is not the owner, and that the owner is updated when the caller is the owner.
Capturing Events with `spy_events`
When a pizza is created, the contract emits an event. To test this, we can use the `spy_events` function to capture the emitted events and assert that the event was emitted with the expected parameters. Naturally, we can also assert that the pizza count was incremented, and that only the owner can make a pizza.
```cairo,noplayground
{{#rustdoc_include ../listings/ch17-starknet-smart-contracts-security/
listing_02_pizza_factory_snfoundry/src/tests/foundry_test.cairo:test_make_pizza}}
```
{{#label capture-pizza-emission-event}}
Listing {{#ref capture-pizza-emission-event}}: Testing the events emitted when a pizza is created
Accessing Internal Functions with `contract_state_for_testing`
All the tests we have seen so far have been using a workflow that involves deploying the contract and interacting with the contract's entrypoints. However, sometimes we may want to test the internals of the contract directly, without deploying the contract. How could this be done, if we were reasoning in purely Cairo terms?
Recall the struct `ContractState`, which is used as a parameter to all the entrypoints of a contract. To make it short, this struct contains zero-sized fields, corresponding to the storage variables of the contract. The only purpose of these fields is to allow the Cairo compiler to generate the correct code for accessing the storage variables. If we could create an instance of this struct, we could access these storage variables directly, without deploying the contract...
...and this is exactly what the `contract_state_for_testing` function does! It creates an instance of the `ContractState` struct, allowing us to call any function that takes as parameter a `ContractState` struct, without deploying the contract. To interact with the storage variables properly, we need to manually import the traits that define access to the storage variables.
```cairo,noplayground
{{#rustdoc_include ../listings/ch17-starknet-smart-contracts-security/
listing_02_pizza_factory_snfoundry/src/tests/foundry_test.cairo:import_internal}}
```
{{#label test-internal}}
Listing {{#ref test-internal}}: Unit testing our contract without deployment
These imports give us access to our internal functions (notably, `set_owner`), as well as the
read/write access to the `owner` storage variable. Once we have these, we can interact with the
contract directly, changing the address of the owner by calling the `set_owner` method, accessible
through `InternalTrait`, and reading the `owner` storage variable.

> Note: Both approaches cannot be used at the same time. If you decide to deploy the contract, you interact with it using the dispatcher. If you decide to test the internal functions, you interact with the `ContractState` object directly.

```bash,noplayground
{{#include ../listings/ch17-starknet-smart-contracts-security/
listing_02_pizza_factory_snfoundry/output.txt}}
```

The output of the tests shows that all the tests passed successfully, along with an estimation of the gas consumed by each test.

Summary

In this chapter, we learned how to test smart contracts using Starknet Foundry. We demonstrated how to deploy a contract and interact with it using the dispatcher. We also showed how to test the contract's behavior by mocking the caller address and capturing events. Finally, we demonstrated how to test the internal functions of the contract directly, without deploying the contract.

To learn more about Starknet Foundry, refer to the [Starknet Foundry documentation](https://foundry-rs.github.io/starknet-foundry/index.html).