Testing Components
Testing components is a bit different than testing contracts.
Contracts need to be tested against a specific state, which can be achieved by either deploying the contract in a test, or by simply getting the `ContractState` object and modifying it in the context of your tests.
Components are a generic construct, meant to be integrated in contracts, that can't be deployed on their own and don't have a `ContractState` object that we could use. So how do we test them?
Let's consider that we want to test a very simple component called "Counter", that will allow each contract to have a counter that can be incremented. The component is defined in Listing {{#ref test_component}}:
```cairo, noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/
listing_04_test_component/src/counter.cairo:component}}
```
{{#label test_component}}
Listing {{#ref test_component}}: A simple Counter component</
span>
Testing the Component by Deploying a Mock Contract
The easiest way to test a component is to integrate it within a mock contract. This mock contract is only used for testing purposes, and only integrates the component you want to test. This allows you to test the component in the context of a contract, and to use a Dispatcher to call the component's entry points.
We can define such a mock contract as follows:
```cairo, noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/
listing_04_test_component/src/lib.cairo:mock_contract}}
```
This contract is entirely dedicated to testing the `Counter` component. It embeds the component with the `component!` macro, exposes the component's entry points by annotating the impl aliases with `#[abi(embed_v0)]`.
We also need to define an interface that will be required to interact externally with this mock contract.
```cairo, noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/
listing_04_test_component/src/counter.cairo:interface}}
```
We can now write tests for the component by deploying this mock contract and calling its entry points, as we would with a typical contract.
```cairo, noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/
listing_04_test_component/src/tests_deployed.cairo}}
```
Testing Components Without Deploying a Contract
In [Components under the hood][components inner working], we saw that components leveraged genericity to define storage and logic that could be embedded in multiple

contracts. If a contract embeds a component, a `HasComponent` trait is created in this contract, and the component methods are made available.
This informs us that if we can provide a concrete `TContractState` that implements the `HasComponent` trait to the `ComponentState` struct, should be able to directly invoke the methods of the component using this concrete `ComponentState` object, without having to deploy a mock.
Let's see how we can do that by using type aliases. We still need to define a mock contract - let's use the same as above - but this time, we won't need to deploy it.
First, we need to define a concrete implementation of the generic `ComponentState` type using a type alias. We will use the `MockContract::ContractState` type to do so.
```cairo, noplayground
{{#rustdoc_include ../listings/ch16-building-advanced-starknet-smart-contracts/
listing_04_test_component/src/tests_direct.cairo:type_alias}}
```
We defined the `TestingState` type as an alias of the `CounterComponent::ComponentState<MockContract::ContractState>` type. By passing the `MockContract::ContractState` type as a concrete type for `ComponentState`, we aliased a concrete implementation of the `ComponentState` struct to `TestingState`.
Because `MockContract` embeds `CounterComponent`, the methods of `CounterComponent` defined in the `CounterImpl` block can now be used on a `TestingState` object.
Now that we have made these methods available, we need to instantiate an object of type `TestingState`, that we will use to test the component. We can do so by calling the `component_state_for_testing` function, which automatically infers that it should return an object of type `TestingState`.
We can even implement this as part of the `Default` trait, which allows us to return an empty `TestingState` with the `Default::default()` syntax.
Let's summarize what we've done so far:
- We defined a mock contract that embeds the component we want to test.
- We defined a concrete implementation of `ComponentState<TContractState>` using a type alias with `MockContract::ContractState`, that we named `TestingState`.
- We defined a function that uses `component_state_for_testing` to return a `TestingState` object.
We can now write tests for the component by calling its functions directly, without having to deploy a mock contract. This approach is more lightweight than the previous one, and it allows testing internal functions of the component that are not exposed to the outside world trivially.
```cairo, noplayground
{{#rustdoc_include ../listings/ch16-building-advanced-starknet-smart-contracts/
listing_04_test_component/src/tests_direct.cairo:test}}
```
[components inner working]: ./ch16-02-01-under-the-hood.md