Components: Lego-Like Building Blocks for Smart Contracts

Developing contracts sharing a common logic and storage can be painful and bug-prone, as this logic can hardly be reused and needs to be reimplemented in each contract. But what if there was a way to snap in just the extra functionality you need inside your contract, separating the core logic of your contract from the rest?

Components provide exactly that. They are modular add-ons encapsulating reusable logic, storage, and events that can be incorporated into multiple contracts.

They can be used to extend a contract's functionality, without having to reimplement the same logic over and over again.

Think of components as Lego blocks. They allow you to enrich your contracts by plugging in a module that you or someone else wrote. This module can be a simple one, like an ownership component, or more complex like a full-fledged ERC20 token.

A component is a separate module that can contain storage, events, and functions. Unlike a contract, a component cannot be declared or deployed. Its logic will eventually be part of the contract’s bytecode it has been embedded in.

What's in a Component?

A component is very similar to a contract. It can contain:

- Storage variables
- Events
- External and internal functions

Unlike a contract, a component cannot be deployed on its own. The component's code becomes part of the contract it's embedded to.

Creating Components

To create a component, first define it in its own module decorated with a `#[starknet::component]` attribute. Within this module, you can declare a `Storage` struct and `Event` enum, as usually done in [contracts][contract anatomy].

The next step is to define the component interface, containing the signatures of the functions that will allow external access to the component's logic. You can define the interface of the component by declaring a trait with the `#[starknet::interface]` attribute, just as you would with contracts. This interface will be used to enable external access to the component's functions using the [dispatcher][contract dispatcher] pattern.

The actual implementation of the component's external logic is done in an `impl` block marked as `#[embeddable_as(name)]`. Usually, this `impl` block will be an implementation of the trait defining the interface of the component.

> Note: `name` is the name that we’ll be using in the contract to refer to the
> component. It is different than the name of your impl.

You can also define internal functions that will not be accessible externally, by simply omitting the `#[embeddable_as(name)]` attribute above the internal `impl` block. You will be able to use these internal functions inside the contract you embed the component in, but not interact with it from outside, as they're not a part of the abi of the contract.

Functions within these `impl` block expect arguments like `ref self:

ComponentState<TContractState>` (for state-modifying functions) or `self: @ComponentState<TContractState>` (for view functions). This makes the impl generic over `TContractState`, allowing us to use this component in any contract.

[contract anatomy]: ./ch13-02-anatomy-of-a-simple-contract.md
[contract dispatcher]: ./ch15-02-interacting-with-another-contract.md

Example: an Ownable Component

> & þ The example shown below has not been audited and is not intended for
> production use. The authors are not responsible for any damages caused by the
> use of this code.

The interface of the Ownable component, defining the methods available externally to manage ownership of a contract, would look like this:

```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/listing_02_ownable_component/src/component.cairo:interface}}
```

The component itself is defined as:

```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/listing_02_ownable_component/src/component.cairo:component}}
```

This syntax is actually quite similar to the syntax used for contracts. The only differences relate to the `#[embeddable_as]` attribute above the impl and the genericity of the impl block that we will dissect in details.

As you can see, our component has two `impl` blocks: one corresponding to the implementation of the interface trait, and one containing methods that should not be exposed externally and are only meant for internal use. Exposing the `assert_only_owner` as part of the interface wouldn't make sense, as it's only meant to be used internally by a contract embedding the component.

A Closer Look at the `impl` Block

```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/listing_02_ownable_component/src/component.cairo:impl_signature}}
```

The `#[embeddable_as]` attribute is used to mark the impl as embeddable inside a contract. It allows us to specify the name of the impl that will be used in the contract to refer to this component. In this case, the component will be referred to as `Ownable` in contracts embedding it.

The implementation itself is generic over `ComponentState<TContractState>`, with the added restriction that `TContractState` must implement the `HasComponent<T>` trait. This allows us to use the component in any contract, as long as the contract implements the `HasComponent` trait. Understanding this mechanism in details is not required to use components, but if you're curious about the inner workings, you can read more in the ["Components Under the Hood"][components inner working] section.

One of the major differences from a regular smart contract is that access to

storage and events is done via the generic `ComponentState<TContractState>` type and not `ContractState`. Note that while the type is different, accessing storage or emitting events is done similarly via `self.storage_var_name.read()` or `self.emit(...).`

> Note: To avoid the confusion between the embeddable name and the impl name, we > recommend keeping the suffix `Impl` in the impl name.

[components inner working]: ./ch16-02-01-under-the-hood.md

Migrating a Contract to a Component

Since both contracts and components share a lot of similarities, it's actually very easy to migrate from a contract to a component. The only changes required are:

- Adding the `#[starknet::component]` attribute to the module.
- Adding the `#[embeddable_as(name)]` attribute to the `impl` block that will be embedded in another contract.
- Adding generic parameters to the `impl` block:
  - Adding `TContractState` as a generic parameter.
  - Adding `+HasComponent<TContractState>` as an impl restriction.
- Changing the type of the `self` argument in the functions inside the `impl` block to `ComponentState<TContractState>` instead of `ContractState`.

For traits that do not have an explicit definition and are generated using `#[generate_trait]`, the logic is the same - but the trait is generic over `TContractState` instead of `ComponentState<TContractState>`, as demonstrated in the example with the `InternalTrait`.

Using Components Inside a Contract

The major strength of components is how it allows reusing already built primitives inside your contracts with a restricted amount of boilerplate. To integrate a component into your contract, you need to:

1. Declare it with the `component!()` macro, specifying
   1. The path to the component `path::to::component`.
   2. The name of the variable in your contract's storage referring to this component's storage (e.g. `ownable`).
   3. The name of the variant in your contract's event enum referring to this component's events (e.g. `OwnableEvent`).
2. Add the path to the component's storage and events to the contract's `Storage` and `Event`. They must match the names provided in step 1 (e.g. `ownable: ownable_component::Storage` and `OwnableEvent: ownable_component::Event`).
   The storage variable **MUST** be annotated with the `#[substorage(v0)]` attribute.
3. Embed the component's logic defined inside your contract, by instantiating the component's generic impl with a concrete `ContractState` using an impl alias. This alias must be annotated with `#[abi(embed_v0)]` to externally expose the component's functions.
   As you can see, the InternalImpl is not marked with `#[abi(embed_v0)]`. Indeed, we don't want to expose externally the functions defined in this impl. However, we might still want to access them internally.

For example, to embed the `Ownable` component defined above, we would do the following:

```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/listing_02_ownable_component/src/contract.cairo:all}}
```

The component's logic is now seamlessly part of the contract! We can interact with the components functions externally by calling them using the `IOwnableDispatcher` instantiated with the contract's address.

```cairo
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/listing_02_ownable_component/src/component.cairo:interface}}
```

Stacking Components for Maximum Composability

The composability of components really shines when combining multiple of them together. Each adds its features onto the contract. You can rely on [Openzeppelin's][OpenZeppelin Cairo Contracts] implementation of components to quickly plug-in all the common functionalities you need a contract to have.

Developers can focus on their core contract logic while relying on battle-tested and audited components for everything else.

Components can even [depend][component dependencies] on other components by restricting the `TContractstate` they're generic on to implement the trait of another component.

Before we dive into this mechanism, let's first look at [how components work under the hood][components inner working].

[OpenZeppelin Cairo Contracts]: https://github.com/OpenZeppelin/cairo-contracts
[component dependencies]: ./ch16-02-02-component-dependencies.md
[components inner working]: ./ch16-02-01-under-the-hood.md