Components: Under the Hood

Components provide powerful modularity to Starknet contracts. But how does this magic actually happen behind the scenes?

This chapter will dive deep into the compiler internals to explain the mechanisms that enable component composability.

A Primer on Embeddable Impls

Before digging into components, we need to understand _embeddable impls_.

An impl of a Starknet interface trait (marked with `#[starknet::interface]`) can be made embeddable. Embeddable impls can be injected into any contract, adding new entry points and modifying the ABI of the contract.

Let's look at an example to see this in action:

```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/no_listing_01_embeddable/src/lib.cairo}}
```

By embedding `SimpleImpl`, we externally expose `ret4` in the contract's ABI.

Now that we're more familiar with the embedding mechanism, we can now see how components build on this.

Inside Components: Generic Impls

Recall the impl block syntax used in components:

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

The key points:

- `OwnableImpl` requires the implementation of the `HasComponent<TContractState>` trait by the underlying contract, which is automatically generated with the `component!()` macro when using a component inside a contract.
  The compiler will generate an impl that wraps any function in `OwnableImpl`, replacing the `self: ComponentState<TContractState>` argument with `self: TContractState`, where access to the component state is made via the `get_component` function in the `HasComponent<TContractState>` trait.
  For each component, the compiler generates a `HasComponent` trait. This trait defines the interface to bridge between the actual `TContractState` of a generic contract, and `ComponentState<TContractState>`.
  ```cairo,noplayground
  // generated per component
  trait HasComponent<TContractState> {
      fn get_component(self: @TContractState) -> @ComponentState<TContractState>;
      fn get_component_mut(ref self: TContractState) -> ComponentState<TContractState>;
      fn get_contract(self: @ComponentState<TContractState>) -> @TContractState;
      fn get_contract_mut(ref self: ComponentState<TContractState>) -> TContractState;
      fn emit<S, impl IntoImp: traits::Into<S, Event>>(ref self: ComponentState<TContractState>, event: S);
```

}
```

In our context `ComponentState<TContractState>` is a type specific to the ownable component, i.e. it has members based on the storage variables defined in `ownable_component::Storage`. Moving from the generic `TContractState` to `ComponentState<TContractState>` will allow us to embed `Ownable` in any contract that wants to use it. The opposite direction (`ComponentState<TContractState>` to `ContractState`) is useful for dependencies (see the `Upgradeable` component depending on an `IOwnable` implementation example in the [Components dependencies ](./ch16-02-02-component-dependencies.md) section).

To put it briefly, one should think of an implementation of the above `HasComponent<T>` as saying: **“Contract whose state T has the upgradeable component”.**

- `Ownable` is annotated with the `embeddable_as(<name>)` attribute: `embeddable_as` is similar to `embeddable`; it only applies to impls of `starknet::interface` traits and allows embedding this impl in a contract module. That said, `embeddable_as(<name>)` has another role in the context of components. Eventually, when embedding `OwnableImpl` in some contract, we expect to get an impl with the following functions:

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

Note that while starting with a function receiving the generic type `ComponentState<TContractState>`, we want to end up with a function receiving `ContractState`. This is where `embeddable_as(<name>)` comes in. To see the full picture, we need to see what is the impl generated by the compiler due to the `embeddable_as(Ownable)` annotation:

```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/no_listing_02_embeddable_as_output/src/lib.cairo}}
```

Note that thanks to having an impl of `HasComponent<TContractState>`, the compiler was able to wrap our functions in a new impl that doesn’t directly know about the `ComponentState` type. `Ownable`, whose name we chose when writing `embeddable_as(Ownable)`, is the impl that we will embed in a contract that wants ownership.

Contract Integration

We've seen how generic impls enable component reusability. Next let's see how a contract integrates a component.

The contract uses an **impl alias** to instantiate the component's generic impl with the concrete `ContractState` of the contract.

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

```
The above lines use the Cairo impl embedding mechanism alongside the impl alias syntax. We're instantiating the generic `OwnableImpl<TContractState>` with the concrete type `ContractState`. Recall that `OwnableImpl<TContractState>` has the `HasComponent<TContractState>` generic impl parameter. An implementation of this trait is generated by the `component!` macro.
Note that only the using contract
could have implemented this trait since only it knows about both the contract state and the component state.
This glues everything together to inject the component logic into the contract.
## Key Takeaways
- Embeddable impls allow injecting components logic into contracts by adding entry points and modifying the contract ABI.
- The compiler automatically generates a `HasComponent` trait implementation when a component is used in a contract. This creates a bridge between the contract's state and the component's state, enabling interaction between the two.
- Components encapsulate reusable logic in a generic, contract-agnostic way. Contracts integrate components through impl aliases and access them via the generated `HasComponent` trait.
- Components build on embeddable impls by defining generic component logic that can be integrated into any contract wanting to use that component. Impl aliases instantiate these generic impls with the contract's concrete storage types.
```