Contract Events
Events are custom data structures that are emitted by smart contracts during execution. They provide a way for smart contracts to communicate with the external world by logging information about specific occurrences in a contract.
Events play a crucial role in the integration of smart contracts in real-world applications. Take, for instance, the Non-Fungible Tokens (NFTs) minted on Starknet. An event is emitted each time a token is minted. This event is indexed and stored in some database, allowing applications to display almost instantaneously useful information to users. If the contract doesn't emit an event when minting a new token, it would be less practical, with the need of querying the state of the blockchain to get the data needed.
Defining Events
All the different events in a contract are defined under the `Event` enum, which must implement the `starknet::Event` trait. This trait is defined in the core library as follows:
```cairo,noplayground
{{#include ../listings/ch14-building-starknet-smart-contracts/no_listing_02_event_trait/src/lib.cairo}}
```
The `#[derive(starknet::Event)]` attribute causes the compiler to generate an implementation for the above trait,
instantiated with the `Event` type, which in our example is the following enum:
```cairo,noplayground
{{#include ../listings/ch14-building-starknet-smart-contracts/listing_01_reference_contract/src/lib.cairo:event}}
```
Each variant of the `Event` enum has to be a struct or an enum, and each variant needs to implement the `starknet::Event` trait itself. Moreover, the members of these variants must implement the `Serde` trait (_c.f._ [Appendix C: Serializing with Serde][serde appendix]), as keys/data are added to the event using a serialization process.
The auto-implementation of the `starknet::Event` trait will implement the `append_keys_and_data` function for each variant of our `Event` enum. The generated implementation will append a single key based on the variant name (`StoredName`), and then recursively call `append_keys_and_data` in the impl of the `Event` trait for the variant's type.
In our example, the `Event` enum contains only one variant, which is a struct named `StoredName`. We chose to name our variant with the same name as the struct name, but this is not enforced.
```cairo,noplayground
{{#include ../listings/ch14-building-starknet-smart-contracts/listing_01_reference_contract/src/lib.cairo:storedname}}
```
Whenever an enum that derives the `starknet::Event` trait has an enum variant, this enum is nested by default. Therefore, the list of keys corresponding to the variant's name will include the `sn_keccak` hash of the variant's name itself. This can be superfluous, typically when using embedded components in contracts. Indeed, in such cases, we might want the events defined in the components to be emitted without any additional data, and it could be useful to annotate the enum variant with the `#[flat]`

attribute. By doing so, we allow to opt out of the nested behavior and ignore the variant name in the serialization process. On the other hand, nested events have the benefit of distinguishing between the main contract event and different components events, which might be helpful.
In our contract, we defined an event named `StoredName` that emits the contract address of the caller and the name stored within the contract, where the `user` field is serialized as a key and the `name` field is serialized as data.
Indexing events fields allows for more efficient queries and filtering of events. To index a field as a key of an event, simply annotate it with the `#[key]` attribute as demonstrated in the example for the `user` key. By doing so, any indexed field will allow queries of events that contain a given value for that field with \\(O(log(n)) \\) time complexity, while non indexed fields require any query to iterate over all events, providing \\(O(n) \\) time complexity.
When emitting the event with `self.emit(StoredName { user: user, name: name })`, a key corresponding to the name ` StoredName`, specifically `sn_keccak(StoredName)`, is appended to the keys list. `user`is serialized as key, thanks to the `#[key]` attribute, while address is serialized as data. After everything is processed, we end up with the following keys and data: `keys = [sn_keccak("StoredName"),user]` and `data = [name]`.
[serde appendix]: ./appendix-03-derivable-traits.md#serializing-with_serde
Emitting Events
After defining events, we can emit them using `self.emit`, with the following syntax:
```cairo,noplayground
{{#include ../listings/ch14-building-starknet-smart-contracts/
listing_01_reference_contract/src/lib.cairo:emit_event}}
```
The `emit` function is called on `self` and takes a reference to `self`, i.e., state modification capabilities are required. Therefore, it is not possible to emit events in view functions.