Substrate Runtime

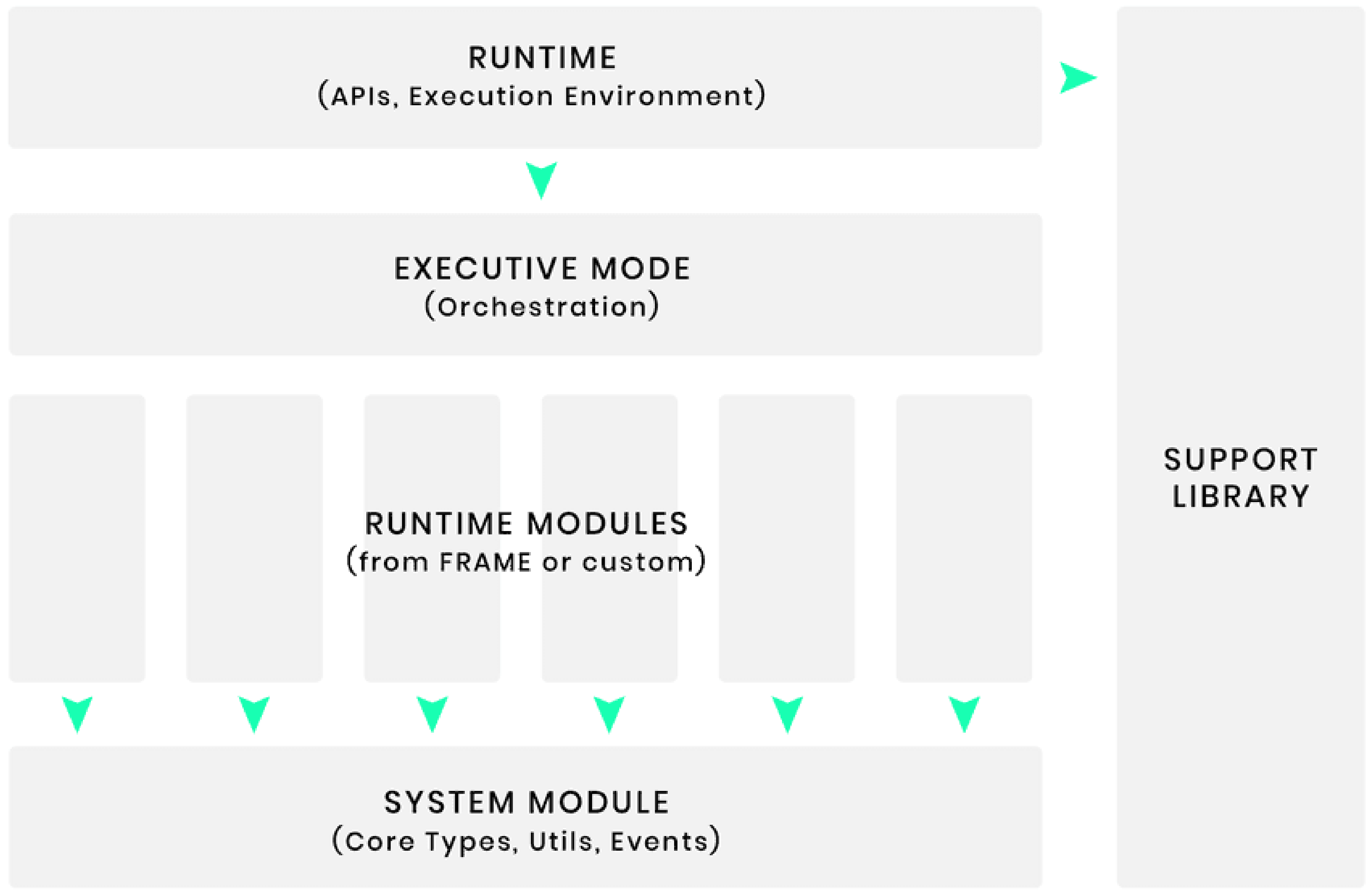
Outline

- Introduction to FRAME and Pallet
- Basic Macro Introduction
- On-chain Storage
- Event
- Error Handling
- Account Origin
- Debug

FRAME

Modules and libraries tofaciliate development

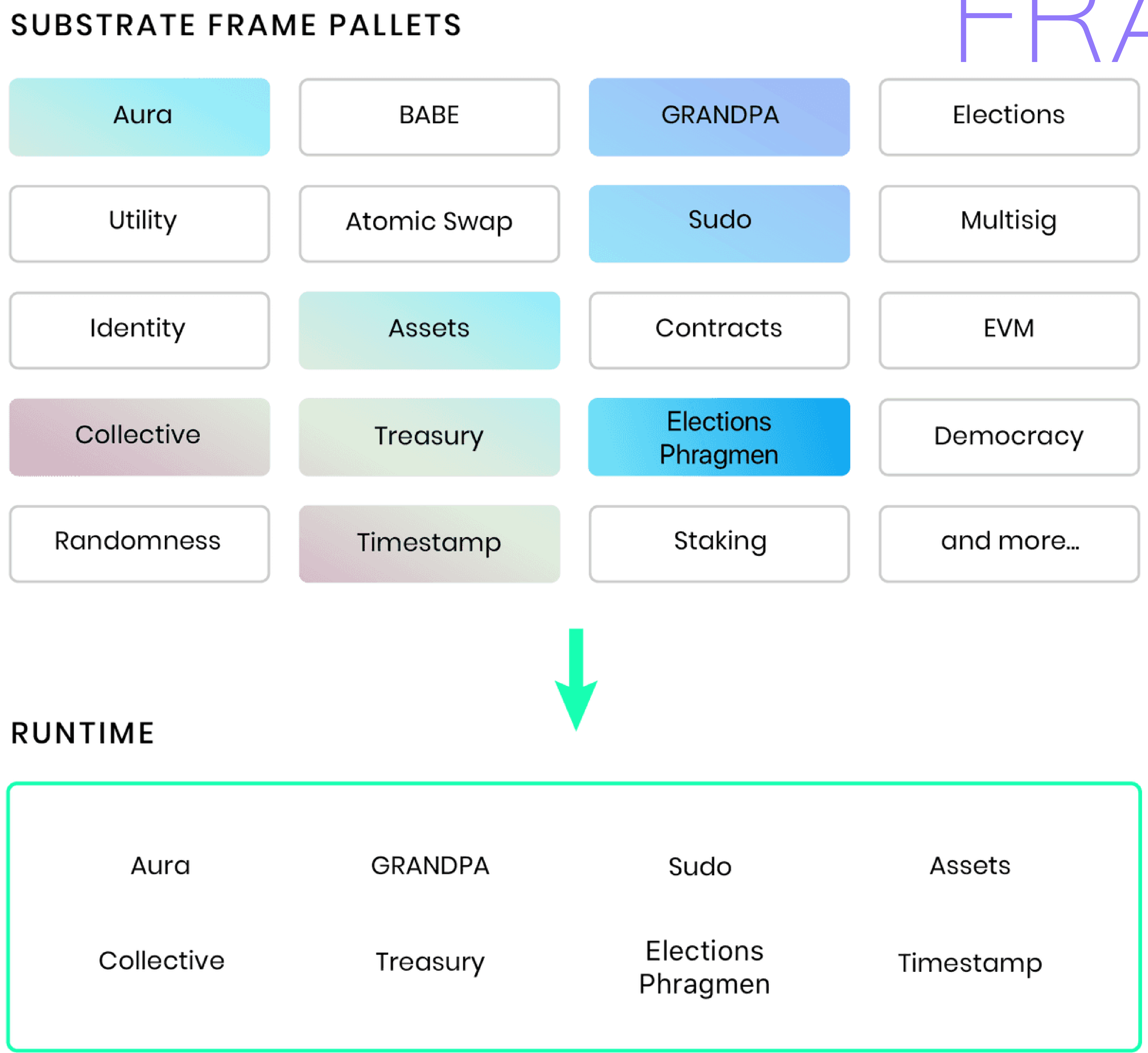
- frame_support::pallet
- pallet::pallet
- pallet::config
- pallet::storage
- pallet::event
- pallet::error
- pallet::call
- pallet::hooks
- construct_runtime



FRAME

- frame_support: is a convinience for developers to define bussiness related types, traits and trait functions
- pallet::pallet: declates the pallets of the bussiness
- pallet::config: defines the types related to
- pallet::storage: similar to underlying database and used to define the data to be stored on chain
- pallet::event: about dynamic changes on chain
- pallet::error: defines the errors of the business change
- pallet::call: defines methods and functions that can be called externally
- pallet::hook: are functions ready to block import construct
- construct_runtime: cooperative with the pallet, to combine all bussiness to relize the runtime of the application chain together

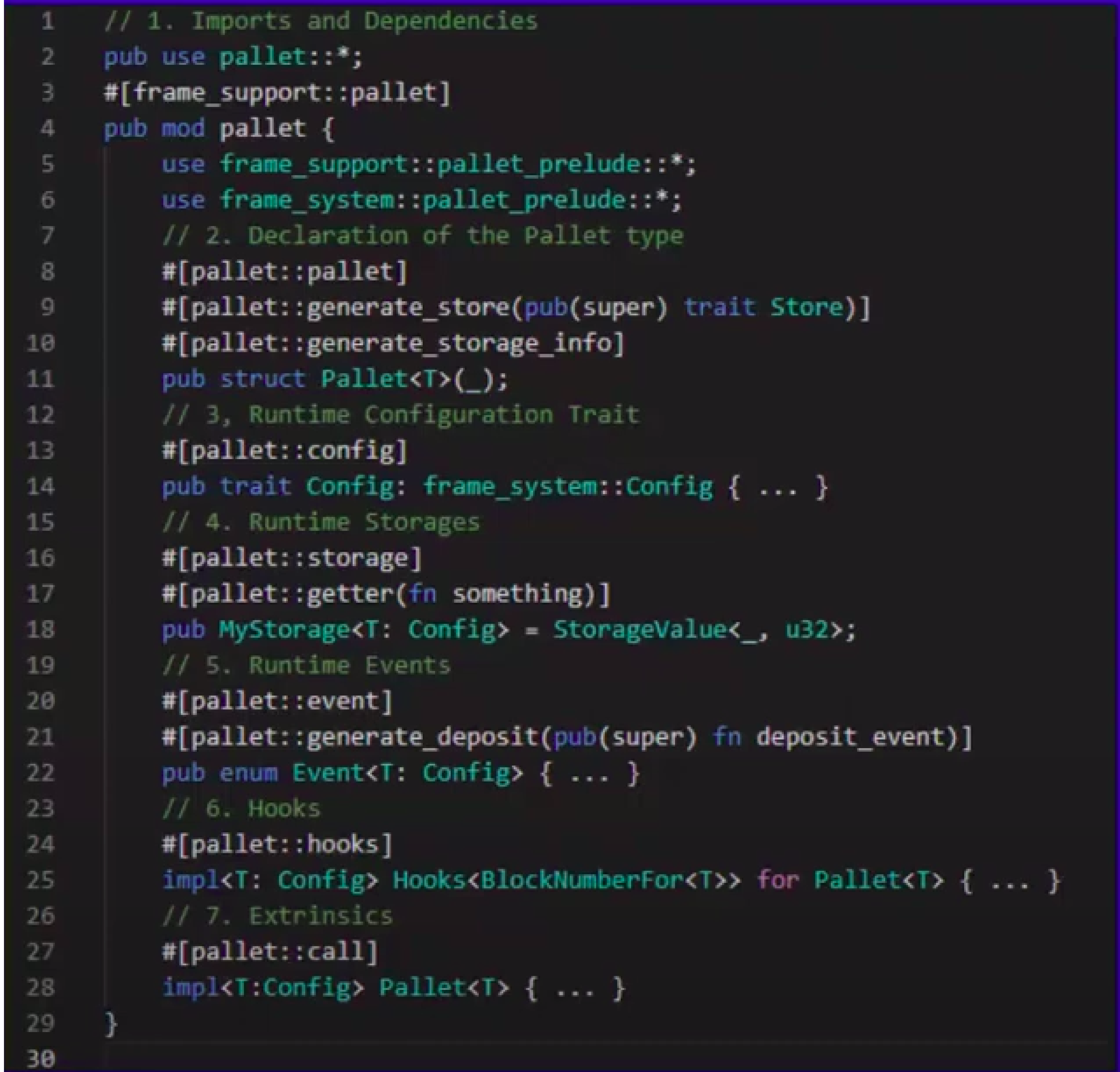
FRAME's Pallets



Skeleton of pallet

1. Imports and Dependencies
2. Declaration of the Pallet type
3. Runtime Configuration Trait
4. Runtime Storage
5. Runtime Events
6. Hooks
7. Extrinsic

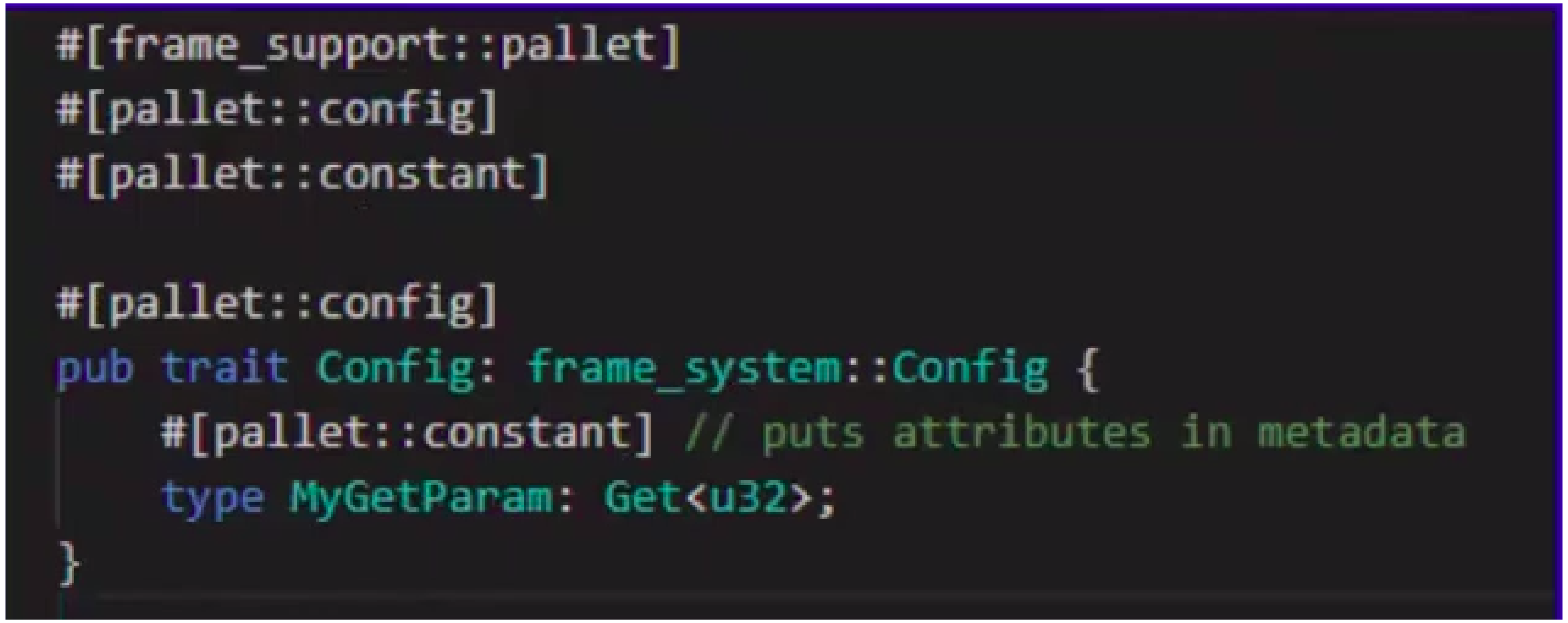
Skeleton of Pallet



```rust
// 1. Imports and Dependencies
pub use pallet::*;
#[frame_support::pallet]
pub mod pallet {
    use frame_support::pallet_prelude::*;
    use frame_system::pallet_prelude::*;
    // 2. Declaration of the Pallet type
    #[pallet::pallet]
    #[pallet::generate_store(pub(super) trait Store)]
    #[pallet::generate_storage_info]
    pub struct Pallet<T>(_);
    // 3, Runtime Configuration Trait
    #[pallet::config]
    pub trait Config: frame_system::Config { ... }
    // 4. Runtime Storages
    #[pallet::storage]
    #[pallet::getter(fn something)]
    pub MyStorage<T: Config> = StorageValue<_, u32>;
    // 5. Runtime Events
    #[pallet::event]
    #[pallet::generate_deposit(pub(super) fn deposit_event)]
    pub enum Event<T: Config> { ... }
    // 6. Hooks
    #[pallet::hooks]
    impl<T: Config> Hooks<BlockNumberFor<T>> for Pallet<T> { ... }
    // 7. Extrinsics
    #[pallet::call]
    impl<T:Config> Pallet<T> { ... }
}
```

FRAME Macros and Attributes

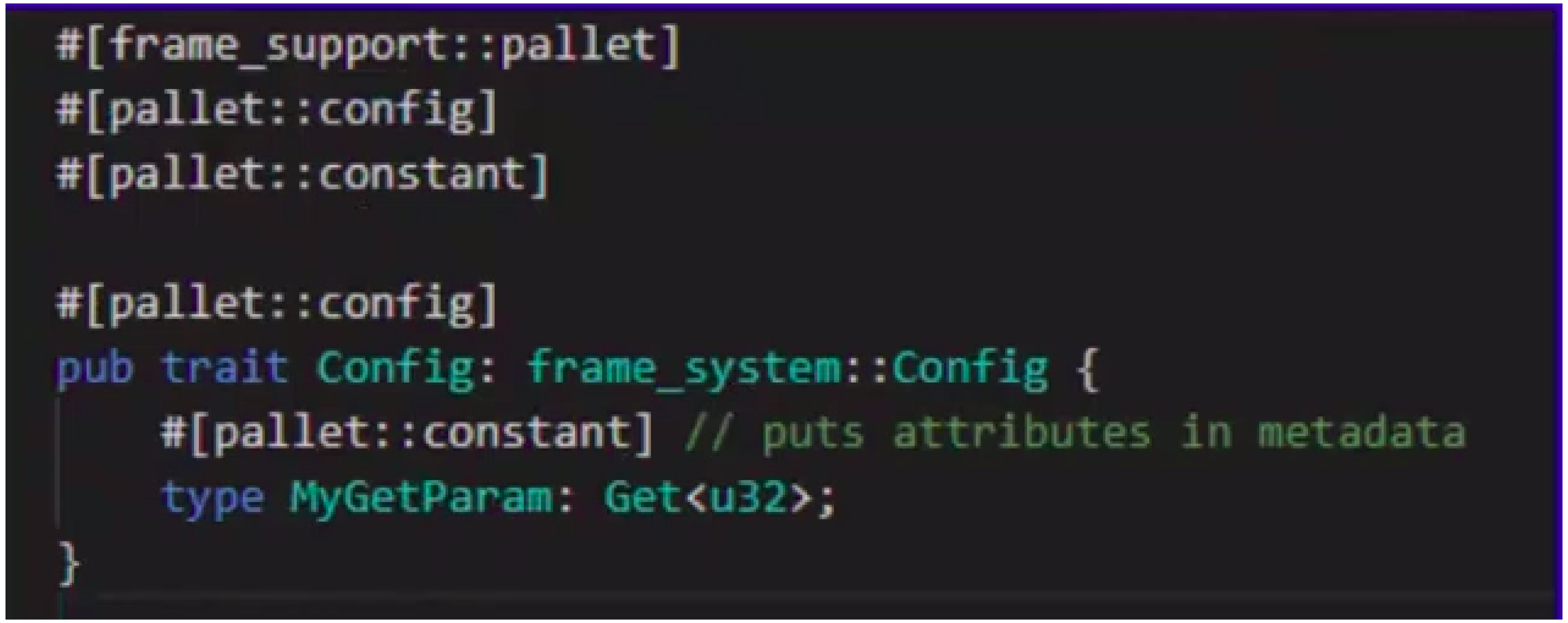
- Substrate uses Rust macro to aggregate logic derives from pallet, implements for use time
- macros allow developers to focus on runtime logic instead of encoding and decoding on-chain variabbles or writting extensive blogs of code to implement basic blockchain foundations
  - releases a lot of heavy work in blockchain development
  - elimates the needs of duplication of code



```
#[frame_support::pallet]
#[pallet::config]
#[pallet::constant]

#[pallet::config]
pub trait Config: frame_system::Config {
    #[pallet::constant] // puts attributes in metadata
    type MyGetParam: Get<u32>;
}
```

FRAME Macros and Attributes

- pallet_config macro provides constants that are part of the config trait and gives infomation so about external tools to use forward runtime



```
#[frame_support::pallet]
#[pallet::config]
#[pallet::constant]

#[pallet::config]
pub trait Config: frame_system::Config {
    #[pallet::constant] // puts attributes in metadata
    type MyGetParam: Get<u32>;
}
```

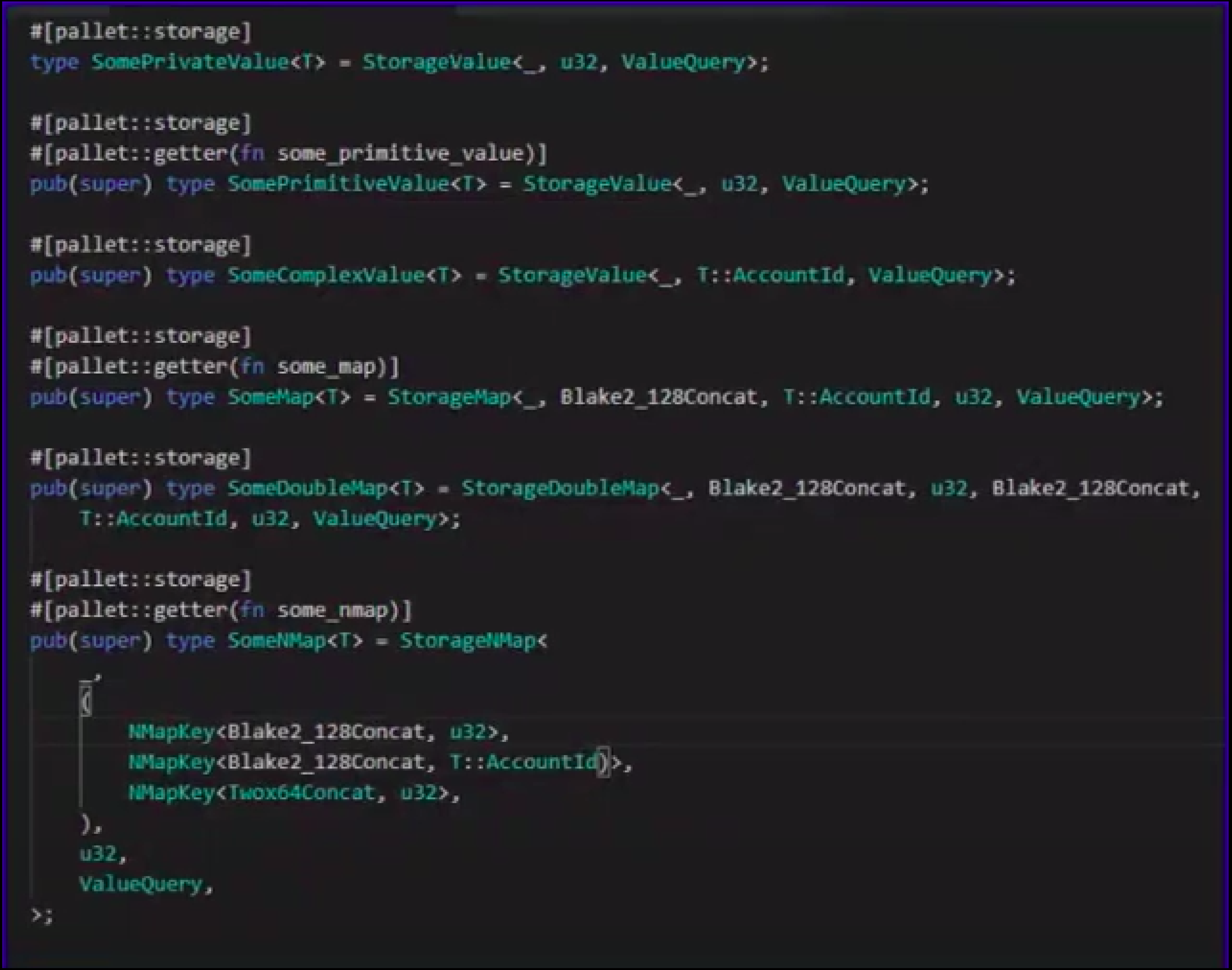
FRAME Macros and Attributes

- #[pallet::hooks]
- #[pallet::error]
- #[pallet::event]
- #[pallet::storage]
-
- construct_runtimes!
- parameter_types!: used for declaring parameter types to be assigned to a pallet configurable trait. associated type during runtime construction
- impl_runtime_api!: used for generating the api implementations for the client side
- app_crypto!: to specify a cryptographic key pairs and is signature algorithm that's out to be managed by a pallet

Storage Items

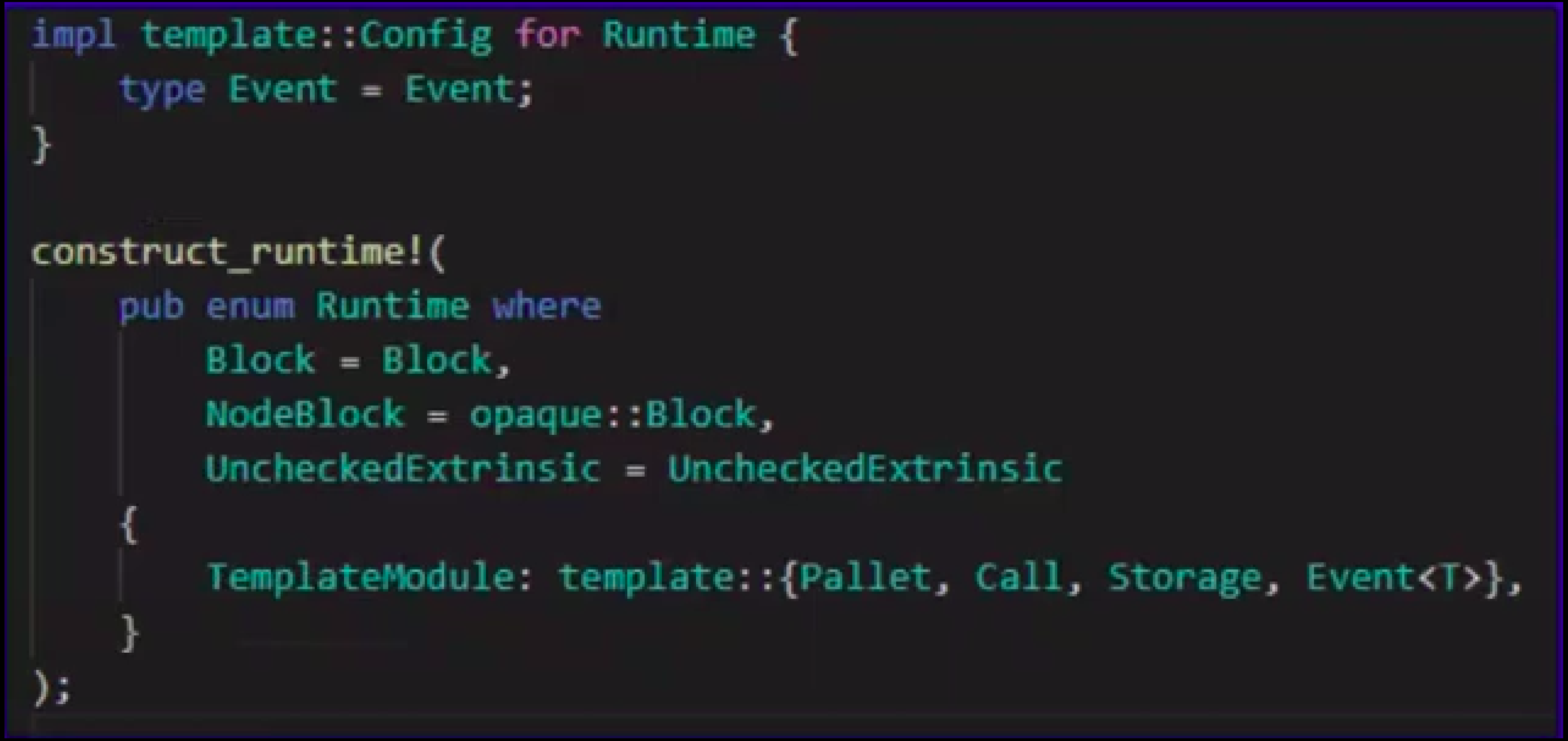
- allows to store data in the blockchain which persists between blocks and can be accessed from a runtime logic
- well-designed storage system
  - reduces the loads of nodes in the network
  - reduces the indirect costs of blockchain participants
  - to minimize its use
- the storage items you choose to implement depends entirely on their expected role in the runtime logic
  - Storage Value
  - Storage Map
  - Storage Double Map: storage map with two keys
  - Storage N Map: store a mapping with multiple keys

Examples



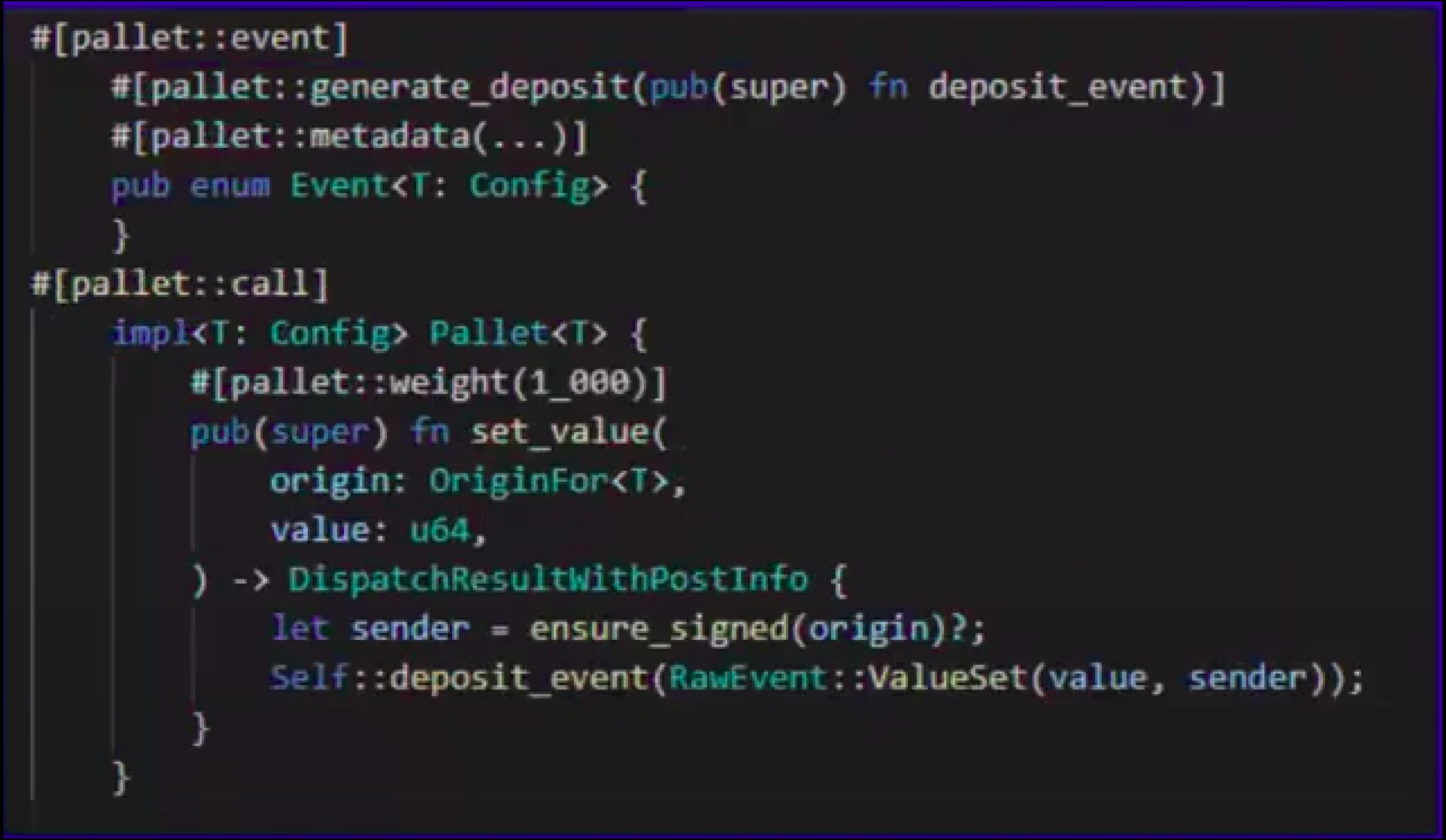
```
#[pallet::storage]
type SomePrivateValue<T> = StorageValue<_, u32, ValueQuery>;

#[pallet::storage]
#[pallet::getter(fn some_primitive_value)]
pub(super) type SomePrimitiveValue<T> = StorageValue<_, u32, ValueQuery>;

#[pallet::storage]
pub(super) type SomeComplexValue<T> = StorageValue<_, T::AccountId, ValueQuery>;

#[pallet::storage]
#[pallet::getter(fn some_map)]
pub(super) type SomeMap<T> = StorageMap<_, Blake2_128Concat, T::AccountId, u32, ValueQuery>;

#[pallet::storage]
pub(super) type SomeDoubleMap<T> = StorageDoubleMap<_, Blake2_128Concat, u32, Blake2_128Concat,
    T::AccountId, u32, ValueQuery>;

#[pallet::storage]
#[pallet::getter(fn some_nmap)]
pub(super) type SomeNMap<T> = StorageNMap<
    _,
    (
        NMapKey<Blake2_128Concat, u32>,
        NMapKey<Blake2_128Concat, T::AccountId>,
        NMapKey<Twox64Concat, u32>,
    ),
    u32,
    ValueQuery,
>;
```

Exposing Events



```rust
impl template::Config for Runtime {
    type Event = Event;
}

construct_runtime!(
    pub enum Runtime where
        Block = Block,
        NodeBlock = opaque::Block,
        UncheckedExtrinsic = UncheckedExtrinsic
    {
        TemplateModule: template::{Pallet, Call, Storage, Event<T>},
    }
);
```

Depositing an Event

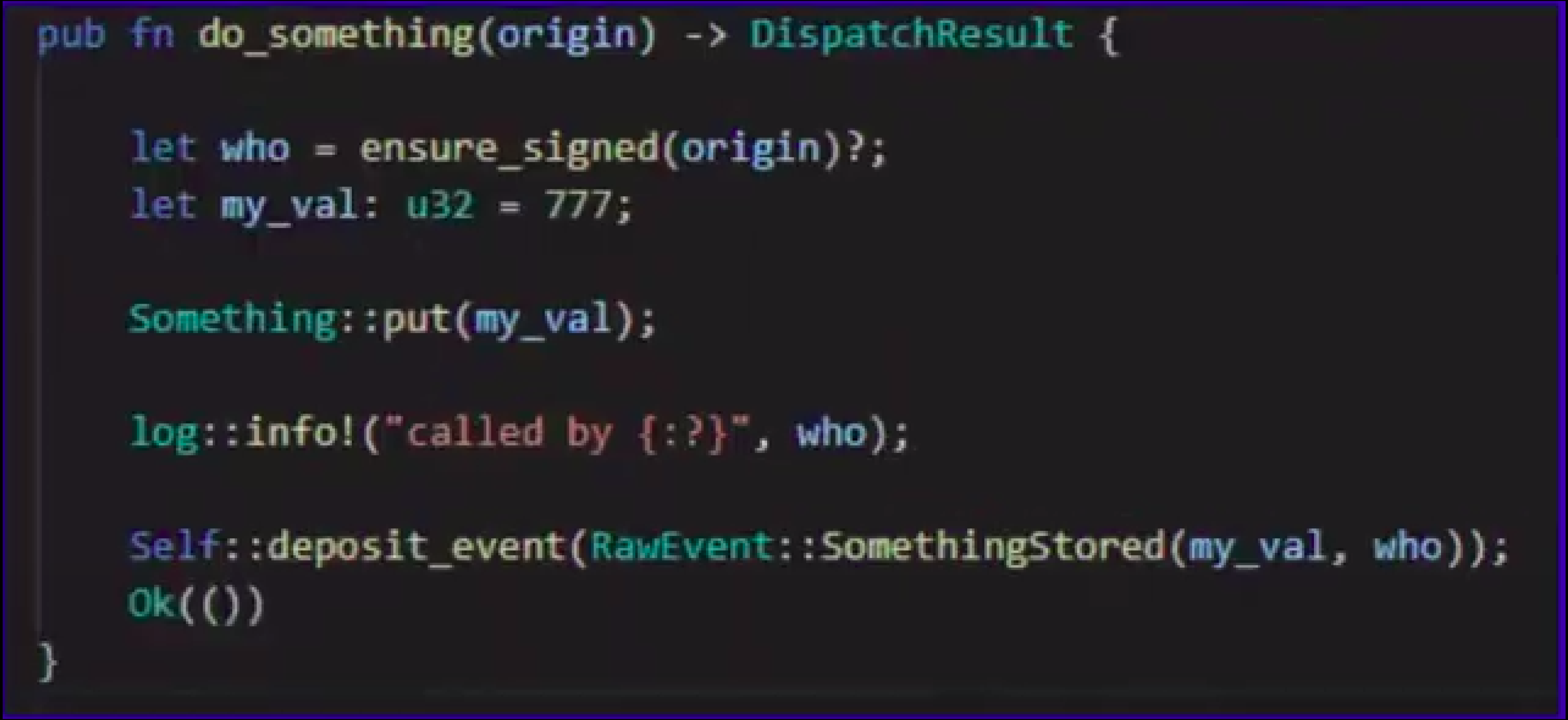


```
#[pallet::event]
    #[pallet::generate_deposit(pub(super) fn deposit_event)]
    #[pallet::metadata(...)]
    pub enum Event<T: Config> {
    }
#[pallet::call]
    impl<T: Config> Pallet<T> {
        #[pallet::weight(1_000)]
        pub(super) fn set_value(
            origin: OriginFor<T>,
            value: u64,
        ) -> DispatchResultWithPostInfo {
            let sender = ensure_signed(origin)?;
            Self::deposit_event(RawEvent::ValueSet(value, sender));
        }
    }
```

Errors

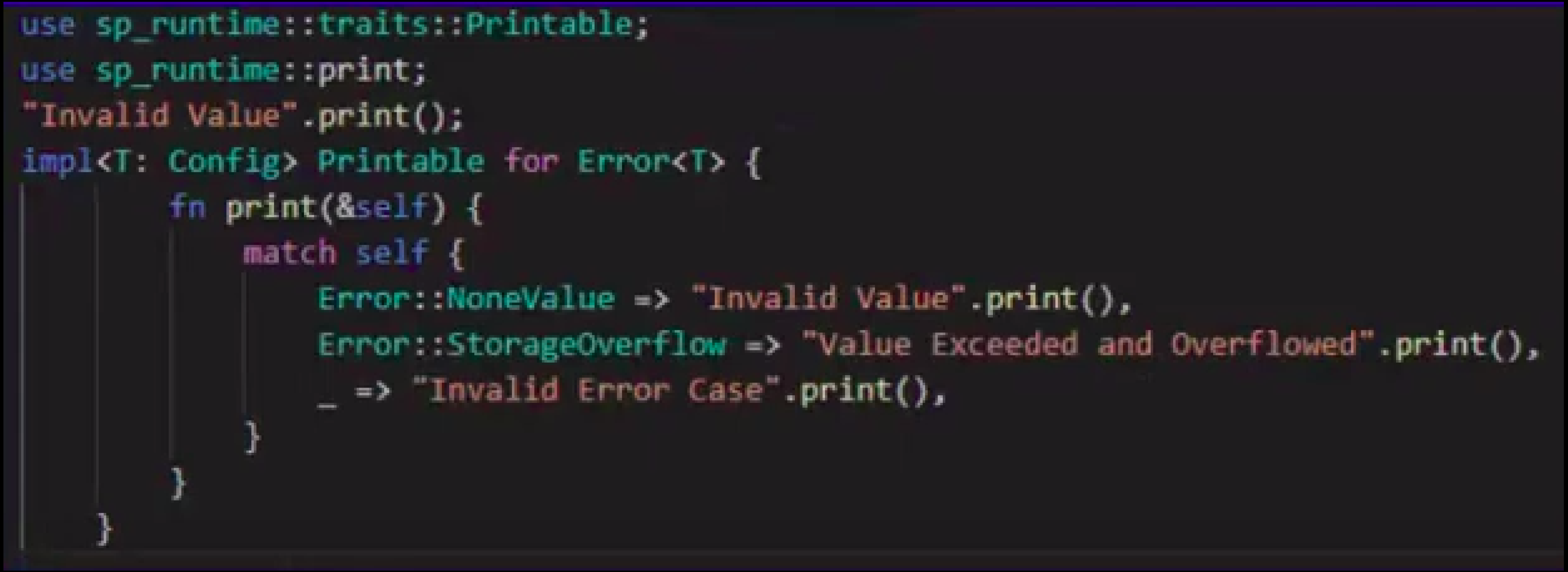
```
#[pallet::error]
pub enum Error<T> {
    InvalidParameter,
    OutOfSpace,
}

frame_support::ensure!(param < T::MaxVal::get(),
Error::<T>::InvalidParameter);
```

Debugging - Logging Utilities



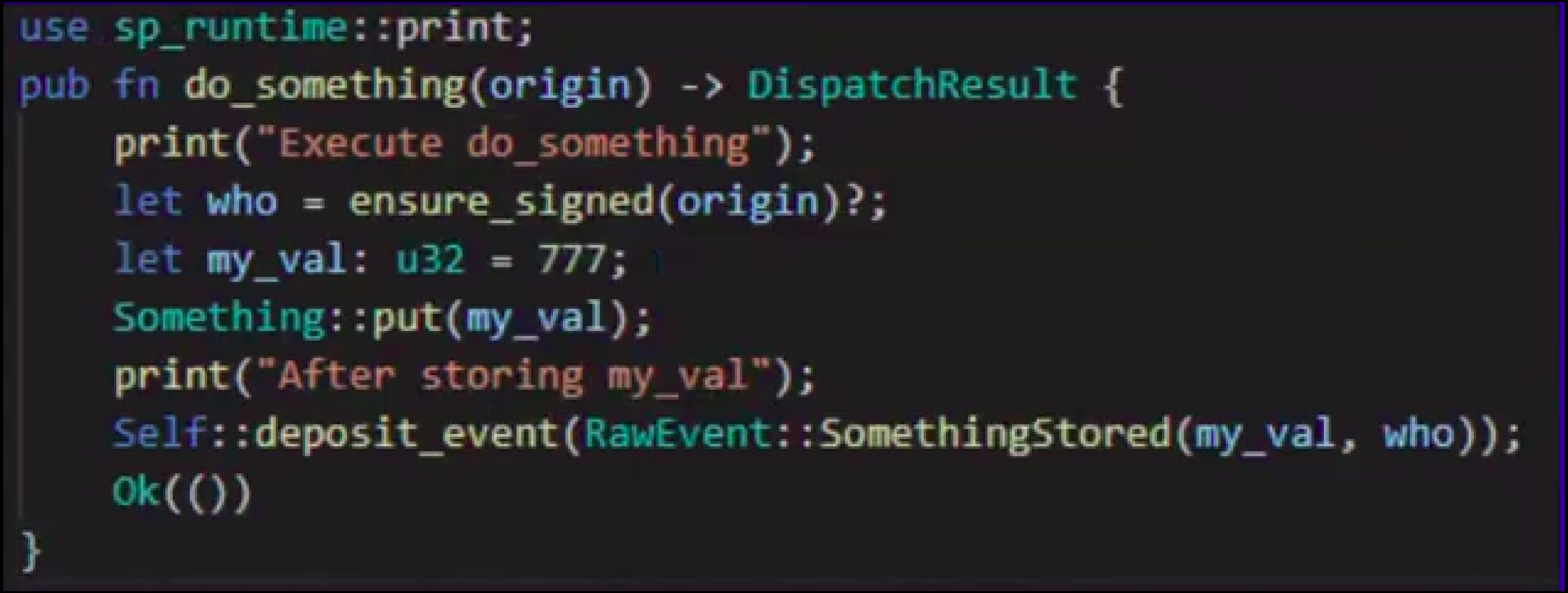
```rust
pub fn do_something(origin) -> DispatchResult {

    let who = ensure_signed(origin)?;
    let my_val: u32 = 777;

    Something::put(my_val);

    log::info!("called by {:?}", who);

    Self::deposit_event(RawEvent::SomethingStored(my_val, who));
    Ok(())
}
```

Debugging - Printable Trait



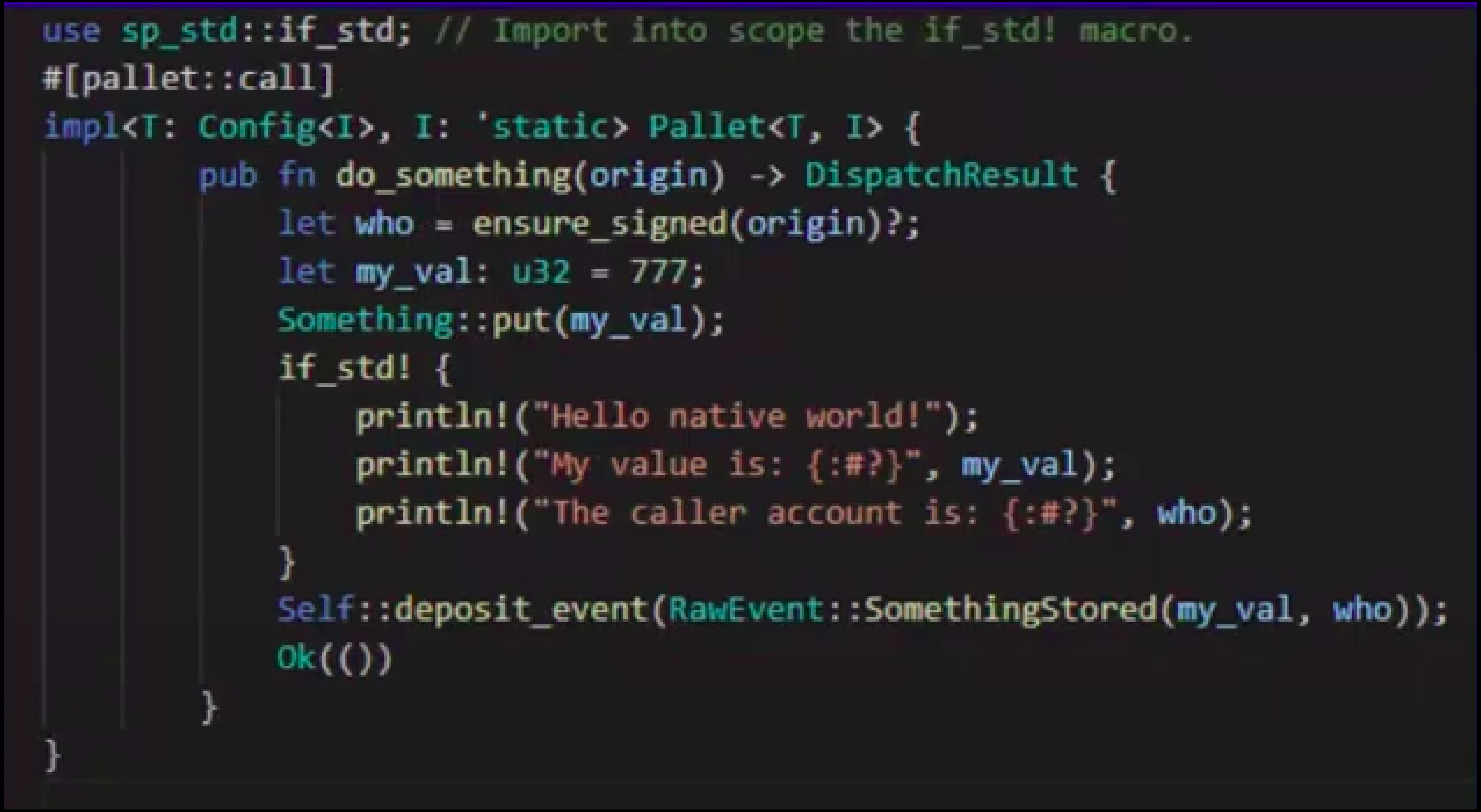
```rust
use sp_runtime::traits::Printable;
use sp_runtime::print;
"Invalid Value".print();
impl<T: Config> Printable for Error<T> {
        fn print(&self) {
            match self {
                Error::NoneValue => "Invalid Value".print(),
                Error::StorageOverflow => "Value Exceeded and Overflowed".print(),
                _ => "Invalid Error Case".print(),
            }
        }
    }
```

Debugging - Substrate's Own Print Function



```
use sp_runtime::print;
pub fn do_something(origin) -> DispatchResult {
    print("Execute do_something");
    let who = ensure_signed(origin)?;
    let my_val: u32 = 777;
    Something::put(my_val);
    print("After storing my_val");
    Self::deposit_event(RawEvent::SomethingStored(my_val, who));
    Ok(())
}
```

Debugging - If std



```rust
use sp_std::if_std; // Import into scope the if_std! macro.
#[pallet::call]
impl<T: Config<I>, I: 'static> Pallet<T, I> {
        pub fn do_something(origin) -> DispatchResult {
            let who = ensure_signed(origin)?;
            let my_val: u32 = 777;
            Something::put(my_val);
            if_std! {
                println!("Hello native world!");
                println!("My value is: {:#?}", my_val);
                println!("The caller account is: {:#?}", who);
            }
            Self::deposit_event(RawEvent::SomethingStored(my_val, who));
            Ok(())
        }
}
```