Substrate Kitties - Dispatchables & Events

Learn how to use storage with your Substrate Pallet

Polkadot Intermediate Rust 1 hour

Written by Kresna Sucandra

Show Table of Contents

Part III: Dispatchables, Events and Errors

Write a dispatchable function that creates a Kitty capable of emitting its associated Event.

Overview

In the previous section of this tutorial, we laid down the foundations geared to manage the ownership of our Kitties — even though they don't really exist yet! In this part of the tutorial, we'll be putting these foundations to use by giving our pallet the ability to create a Kitty using the storage items we declared in the previous part. Breaking things down a little, we're going to:

- **Write** `create_kitty` : a dispatchable or publicly callable function allowing an account to mint a Kitty.
- **Write** `mint()` : a helper function that updates our pallet's storage items and performs error checks, called by `create_kitty` .
- **Include** `Events` : using FRAME's `#[pallet::events]` macro.

At the end of this part, we'll check that everything compiles without error and call our `create_kitty` extrinsic using the PolkadotJS Apps UI.

> Note If you're feeling confident, you can continue building on your codebase from the previous part. If you prefer using the "ACTION" items as a way to assist you through each step, replace the code from the Part II with this part's helper code.

Learning outcomes

➡️ Write a dispatchable function that updates storage items using a helper function.

➡️ Write a private helper function with error handling

➡️ Write and use pallet Events and Errors.

➡️ Use PolkadotJS Apps UI to test pallet functionality.

Steps

Public and private functions

Before we dive right in, it's important to understand the pallet design decisions we'll be making around coding up our Kitty pallet's minting and ownership management capabilities.

As developers, we want to make sure the code we write is efficient and elegant. Oftentimes, optimizing for one optimizes for the other. The way we're going to set up our pallet up to optimize for both will be to break-up the "heavy lifting" dispatchable functions into private helper functions. This improves code readability and reusability too. As we'll see, we can create private functions which can be called by multiple dispatchable functions without compromizing on security. In fact, building this way can be considered an additive security feauture.

> ℹ️ Check out this how-to guide about writing and using helper functions to learn more.

Before jumping into implementing this approach, let's first paint the big picture of what combining dispatchables and helper functions looks like:

`create_kitty` (dispatchable function)

- check the origin is signed
- generate a random hash with the signing account
- create a new Kitty object using the random hash
- call a private `mint()` function
- increment the nonce using `increment_nonce()` from Part II

`mint` (private helper function)

- check that the Kitty doesn't already exist
- update storage with the new Kitty ID (for all Kitties and for the owner's account)
- update the new total Kitty count for storage and the new owner's account
- deposit an Event to signal that a Kitty has succesfully been created

Write the

A dispatchable in FRAME always follows the same structure. All pallet dispatchables live under the `#[pallet::call]` macro which requires declaring the dispatchables section with `impl<T: Config> Pallet<T> {}`. Read the documentation on these FRAME macros to learn how they work. All we need to know here is that they're a useful feature of FRAME that minimizes the code required to write for pallets to be properly integrated in a Substrate chain's runtime.

Weights

As per the requirement for `#[pallet::call]` described in the its documentation, every dispatchable function must have an associated weight to it. Weights are an important part of developing with Substrate as they provide safe-guards around the amount of computation to fit in a block at execution time. Substrate's weighting system forces developers to think about the computational complexity each extrinsic carries before it is called so that a node will account for it's worst case, avoiding lagging the network with extrinsics that may take longer than the specified block time. Weights are also intimately linked to the fee system for a signed extrinsic.

For this simple application, we're going to default all weights to 100.

Find ACTION number 1 and replace it with the following code:

```
#[pallet::weight(100)]
pub fn create_kitty(origin: OriginFor<T>) -> DispatchResultWithPostInfo {
    let sender = ensure_signed(origin)?;
    let random_hash = Self::random_hash(&sender);

    let new_kitty = Kitty {
        id: random_hash,
        dna: random_hash,
        price: 0u8.into(),
        gender: Kitty::<T, T>::gender(random_hash),
    };

    Self::mint(sender, random_hash, new_kitty)?;
    Self::increment_nonce()?;

    Ok(().into())
}
```

Write the

As seen when we wrote `create_kitty` in the previous section, we'll need to create `mint()` for writing our new unique Kitty object to the various storage items declared in Part II of this tutorial.

Let's get right to it. Our `mint()` function will take the following arguments:

- **`to`** : of type `T::AccountId`
- **`kitty_id`** : of type `T::Hash`
- **`new_kitty`** : of type `Kitty<T::Hash, T::Balance>`

And it will return `DispatchResult`.

> Note Why "DispatchResult" and not "DispatchResultWithPostInfo" ? In `create_kitty` our return was of type `DispatchResultWithPostInfo`. Since `mint()` is a helper for `create_kitty`, we don't need to overwrite `PostDispatchInfo`, so we can use a return type of `DispatchResult` — its unaugmented version.

Paste in the following code snippet to write the beginning of the `mint` function, replacing ACTION number 2 in the working codebase:

```
// Helper to mint a Kitty.
fn mint(
    to: T::AccountId,
    kitty_id: T::Hash,
    new_kitty: Kitty<T::Hash, T::Balance>,
) -> DispatchResult {
    ensure!(
        !<KittyOwner<T>>::contains_key(kitty_id),
        "Kitty already contains_key"
    );
    // Update total Kitty counts.
    let owned_kitty_count = Self::owned_kitty_count(&to);
    let new_owned_kitty_count = owned_kitty_count
        .checked_add(1)
        .ok_or("Overflow adding a new kitty to account balance")?;

    let all_kitties_count = Self::all_kitties_count();
    let new_all_kitties_count = all_kitties_count
        .checked_add(1)
        .ok_or("Overflow adding a new kitty to total supply")?;

    // Update storage with new Kitty.
    <Kitties<T>>::insert(kitty_id, new_kitty);
    <KittyOwner<T>>::insert(kitty_id, Some(&to));

    // Write Kitty counting information to storage.
    <AllKittiesArray<T>>::insert(new_all_kitties_count, kitty_id);
    <AllKittiesCount<T>>::put(new_all_kitties_count);
```

```
<AllKittiesIndex<T>>::insert(kitty_id, new_all_kitties_count);

// Write Kitty counting information to storage.
<OwnedKittiesArray<T>>::insert((to.clone(), new_owned_kitty_count), kitty_
<OwnedKittiesCount<T>>::insert(&to, new_owned_kitty_count);
<OwnedKittiesIndex<T>>::insert(kitty_id, new_owned_kitty_count);
```

Let's go over what the above code is doing.

The first thing we're doing is to check whether the Kitty being passed in doesn't already exist. To accomplish this, we use the built-in `ensure!` macro that Rust provides us, along with a method provided by FRAME's `StorageMap` called `contains_key`.

`contains_key` will check if a key matches the Hash value in an existing Kitty object. And `ensure!` will return an error if the storage map already contains the given Kitty ID.

Once we've done the check, we proceed with updating our storage items with the Kitty object passed into our function call. To do this, we make use of the `insert` method from our StorageMap API, using the following pattern:

```
<SomeStorageMapStruct<T>>::insert(some_key, new_key);
```

Finally, we compute a few variables to update our storage items that keep track of:

- The indices and count **for all** Kitties.
- The indices and count **of owned** Kitties.

All this requires us to do is add 1 to the current values held by `<AllKittiesCount<T>>` and `<OwnedKittiesCount<T>>`. We can use the same pattern as we did in the previous part when we created `increment_nonce`, using Rust's `checked_add` and `ok_or`. Generically, this looks like:

```
let new_value = previous_value.checked_add(1).ok_or("Overflow error!");
```

A quick recap of our storage items

- **`<Kitties<T>>`**: Stores a Kitty's unique traits and price, by storing the Kitty object.
- **`<KittyOwner<T>>`**: Keeps track of what accounts own what Kitty.
- **`<AllKittiesArray<T>>`**: An index to track of all Kitties.
- **`<AllKittiesCount<T>>`**: Stores the total amount of Kitties in existence.
- **`<AllKittiesIndex<T>>`**: Keeps track of all the Kitties.

- `<OwnedKittiesArray<T>>` : Keep track of who a Kitty is owned by.
- `<OwnedKittiesCount<T>>` : Keeps track of the total amount of Kitties owned.
- `<OwnedKittiesIndex<T>>` : Keeps track of all owned Kitties by index.

> Note There's 8 storage items in total and each type of storage exposes a number of different methods. Have a glance at the methods `StorageValue` and `StorageMap` expose to learn more.

Implement pallet Events

In Substrate, even though a transaction may be finalized, it does not necessarily imply that the function executed by that transaction fully succeeded. To verify this, we make our pallet emit an Event at the end of the function. This not only reports the success of a function's execution, but also tells the "off-chain world" that some particular state transition has happened.

FRAME helps us easily manage and declare our pallet's events using the `#[pallet::event]` macro. With FRAME macros, events are just an enum declared like this:

```
#[pallet::event]
#[pallet::generate_deposit(pub(super) fn deposit_event)]
pub enum Event<T: Config>{
    /// A function succeeded. [time, day]
    Success(T::Time, T::Day),
}
```

As you can see in the above snippet, we use:

`#[pallet::generate_deposit(pub(super) fn deposit_event)]`

This allows us to deposit a specifc event using the pattern below:

```
Self::deposit_event(Event::Success(var_time, var_day));
```

In order to use events inside our pallet, we need to have the `Event` type declared inside our pallet's configuration trait, `Config`. Additionally — just as when adding any type to our pallet's `Config` trait — we need to let our runtime know about it.

This pattern is the same as when we added the `KittyRandomness` type in Part II of this tutorial and has already been included from the initial scaffolding of our codebase:

```
/// Configure the pallet by specifying the parameters and types it depends o
#[pallet::config]
pub trait Config: pallet_balances::Config + frame_system::Config {
    /// Because this pallet emits events, it depends on the runtime's defini
    type Event: From<Event<Self>> + IsType<<Self as frame_system::Config>::E
    //--snip--//
}
```

> Notice that each event deposit is meant to be informative which is why it carries the various types associated with it.
>
> It's good practice to get in the habit of documenting your event declarations so that your code is easy to read. It is convention to document events as such:
>
> `/// Description. [types]`
>
> Learn more about events here.

Declare your pallet events by replacing the ACTION #3 line with:

```
Created(T::AccountId, T::Hash),
PriceSet(T::AccountId, T::Hash, T::Balance),
Transferred(T::AccountId, T::AccountId, T::Hash),
Bought(T::AccountId, T::AccountId, T::Hash, T::Balance),
```

We'll be using most of these events in Part IV of this tutorial. For now let's use the relevant event for our `mint` function.

In order to complete our `mint` function, replace the ACTION number 4 line with:

```
Self::deposit_event(Event::Created(to, kitty_id));
```

> Note If you're building your codebase from the previous part (and haven't been using the helper file for this part) you'll need to add `Ok(())` and properly close the `mint` function.

Error handling

In Part II when we created the `increment_nonce` function, we specified the error message *"Overflow"* using Rust's `ok_or` function. FRAME provides us with an error handling system using `[#pallet::errors]` which allows us to specify errors for our pallet and use them across our pallet's functions.

In this case, let's declare a single error for when checking for overflow in the `increment_nonce` function.

First, declare the error using the provided FRAME macro under `#[pallet::error]` (replace line ACTION #5a):

```
/// Nonce has overflowed past u64 limits
NonceOverflow,
```

Then, use it on `ok_or` inside `increment_nonce` (replace line ACTION number 5b):

```
let next = nonce.checked_add(1).ok_or(Error::<T>::NonceOverflow)?;
```

Now's a good time to see if your chain can compile. Instead of only checking if your pallet compiles, run the following command to see if everything can build:

`cargo +nightly build --release`

> Tip If you ran into errors, scroll to the first error message in your terminal, identify what line is giving an error and check whether you've followed each step correctly. Sometimes a mismatch of curly brackets will unleash a whole bunch of errors that are difficult to understand — double check your code!

Did that build fine? Congratulations! That's the core functionality of our Kitties pallet. In the next step you'll be able to see everything you've built so far in action.

Testing with PolkadotJS Apps

Assuming that you successfully built your chain, let's run it and use the PolkadotJS Apps UI to interact with it.

In your chain's project directory, run:

```
./target/release/node-kitties --tmp --dev
```

By doing this, we're specifying to run a temporary chain in developer mode, so as not to need to purge storage each time we want to start a fresh chain.

Assuming that blocks are being finalized (which you should be able to see from your terminal in which you ran the above command), head over to Poladot.js Apps.

Follow these steps:

1. Check that you're connected to Local Node, under "Development". Your node will default to `127.0.0.1.:9944`.
2. Tell the UI about your custom types.

This requires you to paste them into the "*Settings*" → "*Developers*" section. 3. Go to "*Developer*" → "*Extrinsics*". Paste this in the JSON code editor:

```
{
  "AccountInfo": {
    "nonce": "Index",
    "consumers": "RefCount",
    "providers": "RefCount",
    "data": "AccountData"
  },
  "Address": "MultiAddress",
  "LookupSource": "AccountId",
  "Gender": {
    "_enum": ["male", "female"]
  },
  "Kitty": {
    "id": "H256",
    "dna": "H256",
    "price": "Balance",
    "gender": "Gender"
  }
}
```

> The reason we need this is because we created types that PolkadotJS Apps isn't designed to read by default. By adding them, it can properly decode each of our storage items that rely on custom types.

3. Now go to: *"Developer"* → *"Extrinsics"* and submit a signed extrinsic using *substrateKitties* by calling the `createKitty()` dispatchable. Make 3 different transactions from Alice, Bob and Charlie's accounts

4. Check for the associated event *"Created"* by going to "*Network*" → "*Explorer*". You should be able to see the event emitted and query its block details.
5. Check your newly created Kitty's details by going to "*Developer*" → "*Chain State*". Select the *substrateKitties* pallet and query `Kitties(Hash): Kitty`. **Note:** You'll notice that this is actually querying all of your pallet's storage items!

Be sure to uncheck the "include option" box and you should be able to see the details of your newly minted Kitty in the following format:

```
substrateKitties.kitties: Kitty
[
    [
      0xf78c3e9c10498c350d639a10fa773cb878b85bf2c5989e38f8b1cd5c0bafb7ee
    ],
    {
      id: 0xf78c3e9c10498c350d639a10fa773cb878b85bf2c5989e38f8b1cd5c0bafb7ee,
      dna: 0xf78c3e9c10498c350d639a10fa773cb878b85bf2c5989e38f8b1cd5c0bafb7ee,
      price: 0,
      gender: female
    }
 ],
```

5. Check that other storage items correctly reflect the creation of additional Kitties.

Congratulations! You're pretty much able to take it from here at this point! We've learned how to implement the key parts of what powers a FRAME pallet and how to put them to use. All part IV of this tutorial covers is adding more capabilities to our pallet by taking what we've learnt in this part.

To recap, in this part of the tutorial you've learnt how to:

- Distinguish between implementing a dispatchable function and a private helper function.
- Use `#[pallet::call]`, `#[pallet::events]` and `#[pallet::error]`.
- Implement basic error checking with FRAME.
- Update values in storage with safety checks.
- Implement FRAME events and use them in a function.
- Query storage items and chain state using the PolkadotJS Apps UI.

Next steps

- Create a dispatchable to buy a Kitty
- Create a dispatchable to transfer a Kitty

- Create a dispatchable to breed two Kitties