

Substrate Kitties - Create Kitties

Learn how to add uniqueness to your Substrate Kitties



Polkadot

Intermediate

Rust

1 hour

Written by Kresna Sucandra

☰ Show Table of Contents

🔗 Part II: Uniqueness, custom types and storage maps

In this part of the tutorial, we'll build out the components of our pallet needed to manage the creation and ownership of our Kitties.

Overview

This part of the tutorial dives into some pillar concepts for developing pallets with FRAME. Ontop of learning how to use existing types and traits, you'll learn how create your own types like providing your pallet with a Gender type. At the end of this part, you will have implemented all remaining storage items according to the logic outlined for the Substrate Kitty application [in the overview of this tutorial](#).


Learning outcomes

- ➡ Writing a custom struct and using it in storage.
- ➡ Creating a custom type and implementing it for your pallet's `Config` trait.
- ➡ Using the Randomness trait in a helper function.
- ➡ Adding `StorageMap` items to a pallet.

Steps

Kitty struct scaffolding

We added additional comments to the code from Part I (in the `/pallets/mykitties/src/lib.rs` file) to better assist you with the action items in this part of the tutorial. To follow each step more easily, you can just replace your code with the **helper code** provided below:

 Note If you're feeling confident, you can use the code from the previous part and use the comments marked as "TODO" to follow each step instead of pasting in the helper file for this part.

```
#![cfg_attr(not(feature = "std"), no_std)]

pub use pallet::*;

#[frame_support::pallet]
pub mod pallet {
    use frame_support::{sp_runtime::traits::{Hash, Zero},
                        dispatch::{DispatchResultWithPostInfo, DispatchResult},
                        traits::{Currency, ExistenceRequirement, Randomness},
                        pallet_prelude::*};
    use frame_system::pallet_prelude::*;
    use sp_core::H256;

    // ACTION number 1: Write a Struct to hold Kitty information.

    // ACTION number 2: Enum declaration for Gender.

    // ACTION number 3: Implementation to handle Gender type in Kitty struct.

    #[pallet::pallet]
    #[pallet::generate_store(trait Store)]
    pub struct Pallet<T>(_);

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

        // ACTION number 5: Specify the type for Randomness we want to specify
    }

    // Errors.
```

```

#[pallet::error]
pub enum Error<T> {
    // TODO Part III
}

// Events.
#[pallet::event]
#[pallet::metadata(T::AccountId = "AccountId")]
#[pallet::generate_deposit(pub(super) fn deposit_event)]
pub enum Event<T: Config> {
    // TODO Part III
}

#[pallet::storage]
#[pallet::getter(fn all_kitties_count)]
pub(super) type AllKittiesCount<T: Config> = StorageValue<_, u64, ValueQue

// ACTION number 6: Add Nonce storage item.

// ACTION number 9: Remaining storage items.

// TODO Part IV: Our pallet's genesis configuration.

#[pallet::call]
impl<T: Config> Pallet<T> {

    // TODO Part III: create_kitty

    // TODO Part III: set_price

    // TODO Part III: transfer

    // TODO Part III: buy_kitty

    // TODO Part III: breed_kitty
}

// ACTION number 4: helper function for Kitty struct

impl<T: Config> Pallet<T> {
    // TODO Part III: helper functions for dispatchable functions

    // ACTION number 7: increment_nonce helper

    // ACTION number 8: random_hash helper

    // TODO: mint, transfer_from

```

```
}  
}
```

What information to include

Structs are a useful tool to help store data that have things in common. For our purposes, our Kitty will carry multiple traits which we can store in a single struct instead of using separate storage items. This comes in handy when trying to optimize for storage reads and writes because our runtime will need to perform less read/writes to update multiple values. Read more about storage best practices [here](#).

Let's first go over what information a single Kitty will carry:

- `id` : a unique hash to identify each Kitty.
- `dna` : the hash used to identify the DNA of a Kitty, which corresponds to its unique features. DNA is also used to breed new Kitties and to keep track of different Kitty generations.
- `price` : this is a balance that corresponds to the amount needed to buy a Kitty and determined by its owner.
- `gender` : an enum that can be either `Male` or `Female` .

Sketching out the types held by our struct

Looking at the items of our struct, we can deduce the following types:

- `Hash` for `id` and `dna` (this comes from the `use frame_support::sp_runtime::traits::Hash` import at the top of our pallet)
- `Balance` for `price` (this comes from `pallet_balances::Config` we've exposed to our pallet's `Config` trait)
- `Gender` for `gender` (we're going to need to create this!)

We also use the derive macro to include [various helper traits](#) for using our struct. Our struct will look like this (replace ACTION number 1 in your working codebase with it):

```
// Struct for holding Kitty information.  
#[derive(Clone, Encode, Decode, Default, PartialEq)]  
pub struct Kitty<Hash, Balance> {  
    id: Hash,  
    dna: Hash,  
    price: Balance,  
    gender: Gender,  
}
```


We've already given our pallet access to Substrate's `Hash` type as part of our pallet's scaffolding in the previous part. This will be the type for a Kitty ID and DNA, also used in the `Randomness` algorithm. As for `Balance`, this type is being accessed by our pallet's configuration trait. We'll be using `Balance` in the dispatchable functions we write in Part III.

For type `Gender`, we'll need to build out our own custom enum and helper functions. Now's a good time to do that.

Writing a custom type for

We've just created a struct that requires a custom type called `Gender`. This type will handle an enum defining our Kitty's gender. To create it, you'll build out the following parts:

- **An enum declaration**, which specifies `Male` and `Female` values.
- **A function to configure a default value**, based on the enum.

 Info Setting up our `Gender` enum using a function to configure its default value will allow us to derive a Kitty's > gender by passing in the randomness created by each Kitty's DNA.

Enums

Replace ACTION item number 2 with the following enum declaration:

```
#[derive(Encode, Decode, Debug, Clone, PartialEq)]
pub enum Gender {
    Male,
    Female,
}
```

Notice the use of the `derive macro` which must precede the enum declaration. This wraps our enum in the data structures it will need to interface with other types in our runtime.

Then, we need a function to define a default implementation for our enum by using Rust's `Default trait`.

Replace the line containing ACTION number 3 with:

```
impl Default for Gender {
    fn default() -> Self {
        Gender::Male
    }
}
```

```
}  
}
```

💡 **This is like saying:** let's give our enum a special trait that allows us to initialize it to a specific value.

Great, we now know how to create a custom struct and specify its default value. But what about providing a way for a Kitty struct to be *assigned* a gender value? For that we need to learn one more thing.

Configuring functions for our Kitty struct

Configuring a struct is useful in order to pre-define a value in our struct. For example, when setting a value in relation to what another function returns. In our case we have a similar situation where we need to configure our Kitty struct in such a way that sets `Gender` according to a Kitty's DNA.

We'll only be using this function when we get to [creating Kitties](#). Regardless, learn how to write it now and get it out of the way.

When you're implementing the configuration trait for a struct inside a FRAME pallet, you're doing the same type of thing as implementing some trait for an enum except you're implementing the generic configuration trait, `Config`. In our case we'll create a public function called `gender` that returns the `Gender` type and takes `dna` as a parameter to choose between `Gender` enum values.

Replace ACTION number 4 with the following code snippet:

```
impl<T: Config> Kitty<T, T> {  
    pub fn gender(dna: T::Hash) -> Gender {  
        if dna.as_ref()[0] % 2 == 0 {  
            Gender::Male  
        } else {  
            Gender::Female  
        }  
    }  
}
```

Now whenever `gender()` is called inside our pallet, it will return a pseudo random enum value for `Gender`.

Implement on-chain randomness

If we want to be able to tell these Kitties apart, we need to start giving them unique properties! We already have the `Hash` type specified which will hold the values for these properties. Now, all that's left is to *actually* generate a unique ID and some random DNA for each Kitty.

We'll be using the **Randomness trait** from `frame_support` to do this. It will generate a random seed which will create new unique Kitties as well as breed new ones. We'll also need a nonce which we'll create as a separate function and a hashing function which we'll get by using the **Hash trait** that we imported in the step to add the `hash` dependency.

In order to implement the `Randomness` trait for our runtime, we must:

Specify it in our pallet's configuration trait.

The `Randomness` trait from `frame_support` requires specifying it with a parameter to replace the `Output` and `BlockNumber` generics. Take a look at [the documentation](#) and the source code implementation to understand how this works. For our purposes, we want the output of functions using this trait to be `H256` which you'll notice should already be declared at the top of your working codebase.

Replace the ACTION number 5 line with:


```
type KittyRandomness: Randomness<H256, u32>;
```

Specify it for our runtime.

Given that we're adding a new type for the configuration of our pallet, we need to tell our runtime about its implementation. This could come in handy if ever we wanted to change the algorithm that `KittyRandomness` is using, without needing to modify where it's used inside our pallet.

To showcase this point, we're going to implement `KittyRandomness` by assigning it to an instance of **FRAME's** `RandomnessCollectiveFlip`. Conveniently, the Node Template already has an instance of the `RandomnessCollectiveFlip` pallet. All you need to do is **include the `KittyRandomness` type for your runtime inside `runtime/src/lib.rs`**:

```
impl pallet_kitties::Config for Runtime {
    type Event = Event;
    type KittyRandomness = RandomnessCollectiveFlip; // <-- ACTION: add this l
}
```

 Check out this [how-to guide](#) on implementing randomness in case you get stuck.

Nonce

Note on why our random hashing function needs a nonce



Our goal is to create as much entropy as we can for `hash_of` to produce enough randomness when creating Kitty IDs and DNA. Since `random_seed()` does not change for multiple transactions in the same block, and since it may not even generate a random seed for the first 80 blocks, we need to create a nonce for our pallet to manage and use in our private `random_hash` function.

We'll use the nonce provided by `frame_system::AccountInfo` and create a storage item to keep track of it as we modify it.

So we'll need to do a couple things. First, create a storage item for the nonce value. Replace the ACTION number 6 line with:

```
#[pallet::storage]
#[pallet::getter(fn get_nonce)]
pub(super) type Nonce<T: Config> = StorageValue<
    -,
    u64,
    ValueQuery
>;
```

Second, create a function that increments the nonce. Replace ACTION number 7 with:

```
fn increment_nonce() -> DispatchResult {
    <Nonce<T>>::try_mutate(|nonce| {
        let next = nonce.checked_add(1).ok_or("Overflow")?; // TODO Part III:
        *nonce = next;

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

Implementing the random hashing function

Now that we have all the bits and pieces required to build our hashing function, we can finally build it. Replace the ACTION number 8 line with:

```
fn random_hash(sender: &T::AccountId) -> T::Hash {
    let nonce = <Nonce<T>>::get();
    let seed = T::KittyRandomness::random_seed();
```



```
T::Hashing::hash_of(&(seed, &sender, nonce))  
}
```

Our function takes in an `AccountId` and returns the hash of a **random seed**, **AccountId** and the **current nonce**.

Write remaining storage items

Understanding storage item logic


To easily track all of our kitties, we're going to standardize our logic to use a unique ID as the global key for our storage items. This means that a single unique key will point to our Kitty object, and all other links to ownership will point to that key.

In order for this to work, we need to make sure that the ID for a new Kitty is always unique. We can do this with a new storage item `Kitties` which will be a mapping from an ID (Hash) to the Kitty object.

With this object, we can easily check for collisions by simply checking whether this storage item already contains a mapping using a particular ID. For example, from inside a dispatchable function we could check using:

```
ensure!(!<Kitties<T>>::exists(new_id), "This new id already exists");
```

We'll be needing a total of 9 storage items for our Kitty pallet. We already included `Nonce` and we've already created the basis for our Kitty object — we just need to implement a way to keep track of them now!

 Tip Our pallet's logic can best be understood by examining the storage items we'll be using. In other words, **the way we define the conditions for reading and writing to our runtime's storage helps us breakdown the items we'll need to enable NFT capabilities.**

We care about state transitions and persistence around two main concepts our runtime needs to be made aware of:

1. Unique assets, like currency or Kitties
2. Helper datastructures, like the nonce, counters or account maps

This already starts to lay the foundations for our Kitty pallet logic. But there's an important layer beneath these two concepts: our runtime needs to have a sense of asset ownership as well as the ability to keep track of changes in ownership and owned quantities.

In our application, ontop of keeping a single storage instance for Kitty objects, we need a number of storage items to keep track of:

- Kitties in existence
- Owned Kitties

Note The overarching pattern for our Kitty application is to keep track of *who* and *what*.



This boils down to the following storage items (in addition to `Kitties` and `Nonce`):

Tracking ownership

- `<KittyOwner<T>>` : Keeps track of what accounts own what Kitty.
- `<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.

Tracking existing Kitties

- `<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.

Using a

Every storage item declaration will follow a similar pattern as when we wrote the storage item for `Nonce`. The only difference is which data structure type each storage item requires.

To create a storage instance for the Kitty struct, we'll be using `StorageMap` — a hash-map provided to us by FRAME. This differs from the storage instance we created for `Nonce` which, because we wanted it to keep track of a single `u64` value, therefore we used `StorageValue`. Here, we need our storage to keep track of a map of hash IDs and Kitty objects.



Reminder Every storage item in a FRAME pallet must use the attribute macro `# [pallet::storage]`.

Here's what the `Kitties` storage item looks like:

```
#[pallet::storage]
#[pallet::getter(fn kitty)]
```

```
pub(super) type Kitties<T: Config> = StorageMap<
    -,
    Twox64Concat,
    T::Hash,
    Kitty<T::Hash, T::Balance>,
    ValueQuery
>;
```

Breaking it down, we declare the storage type and assign a `StorageMap` that takes:

- The `Twox64Concat` hashing algorithm.
- A key of type `T::Hash`.
- A value of type `Kitty<T::Hash, T::Balance>`.

Your turn! Copy the code snippet above to replace line ACTION number 9. Then, use the storage items outlined in section 4A to help you finish writing the remaining storage items. Follow the same pattern we used for `Nonce` and `Kitties` — just remember what type each item is meant to store!

HINT: Remember to include a getter function for each storage item — except those handling indices. This will help you think about what each storage item is intended for.

Assuming you've finished implementing all of your storage items, now's a good time to check that your pallet compiles correctly:

```
cargo build -p pallet-kitties
```

Running into difficulties? Check your solution against the [completed helper code](#) for this part of the tutorial.

Congratulations! If you've made it this far, you now have the foundations for your pallet to handle the creation and changes in ownership of your Kitties! In this part of the tutorial, we've learned:

- How to write a struct and use it in a `StorageMap`.
- How to implement a custom type.
- How to set a default enum value for a custom type.
- How to create a function to set a value for that custom type.
- How to implement the Randomness trait to write a function that generates randomness using a nonce.

- How to write `StorageMap` storage items.

Next steps

- Create a dispatchable function that mints a new Kitty
- Create a helper function to handle storage updates
- Implement Errors and Events

[Click here](#) to go to the next part of this tutorial series.

[Figment](#)

[Terms of Use](#)

[Privacy Policy](#)

[Contributor Terms](#)