Substrate Kitties - Interacting with your Kitties

Learn how to add interactivity to your Substrate Kitties

Polkadot Intermediate Rust 1 hour

Written by Kresna Sucandra

Show Table of Contents

Part IV: Interacting with your Kitties

Add pallet capabilities that unleash the potential of your Substrate Kitty application.

Overview

Up until this point in the tutorial, we've built a chain capable of only creating and tracking the ownership of Kitties. In this part of the tutorial, we want to make our runtime more like a game by introducing other functions like buying and selling Kitties. In order to achieve this, we'll first need to enable users to update the price of their Kitty. Then we can add functionality to enable users to transfer, buy and breed Kitties.

Learning outcomes

- Learn how to create a dispatchable that updates an object in storage.
- Getting a value from a struct in storage.
- How to use the `transfer` from FRAME's Currency trait.
- How to write sanity check using `ensure!()`.

Steps

Set a price for each Kitty

In the helper file for this part of the tutorial, you'll notice that the structure of `set_price` is already laid out.

Your job is to replace ACTION lines number 1, number 2 and number 3 lines with what you'll learn in sections A-D below.

Checking Kitty owner

As we create functions which modify objects in storage, we should always check that only the appropriate users are successful when calling those dispatchable functions.

For modifying a Kitty object, we'll first need to get the `Hash` value of the owner of the Kitty to ensure that it's the same as the sender.

> `KittyOwner` stores a mapping to an `Option<T::AccountId>` but that given `Hash` may not point to a generated and owned Kitty yet. This means, whenever we fetch the owner of a Kitty, we need to resolve the possibility that it returns `None`. This could be caused by bad user input or even some sort of problem with our runtime. Checking will help prevent these kinds of problems.

The general pattern for an ownership check will look something like this:

```
let owner = Self::owner_of(object_id).ok_or("No owner for this object")?;

ensure!(owner == sender, "You are not the owner");
```

Your turn! Paste in this code snippet to replace ACTION number 1:

```
// Check that the Kitty has an owner (i.e. if it exists).
let owner = Self::owner_of(kitty_id).ok_or("No owner for this kitt
// Make sure the owner matches the corresponding owner.
ensure!(owner == sender, "You do not own this cat");
```

Updating a the price of our Kitty object

Every Kitty object has a price attribute that we've set to `0u8` as a default value inside the `create_kitty` function in Part II:

```
let new_kitty = Kitty {
    id: random_hash,
    dna: random_hash,
```

```
    price: 0u8.into(), // <-- here
    gender: Kitty::<T, T>::gender(random_hash),
};
```

Note We'll use this default value to check if the Kitty is for sale when we write the `buy_kitty` dispatchable in the next section.

To update the price of a Kitty, we'll need to:

- Get the Kitty object.
- Update the price.
- Push it back into storage.

Changing a value in an existing object in storage would be written in the following way:

```
let mut object = Self::get_object(object_id);
object.value = new_value;

<Object<T>>::insert(object_id, object);
```

Note Rust expects you to declare a variable as mutable (using the `mut` keyword) whenever its value is going to be updated.

Your turn! Paste in the following snippet to replace the ACTION number 2 line:

```
// Set the Kitty price.
let mut kitty = Self::kitty(kitty_id);
kitty.price = new_price;

// Update new Kitty infomation to storage.
<Kitties<T>>::insert(kitty_id, kitty);
```

Sanity checks

In a similar vain of checking permissions, we also need to ensure that our runtime performs regular sanity checks to mitigate any risk that things go wrong — such as users flooding the chain with

heavy transactions or anything that could break the chain.

If we're creating a function geared to update the value of an object in storage, the first thing we better do is make sure the object exists at all. By using `ensure!`, we can create a safe guard against poor user input, whether malicious or unintentional. For example:

```
ensure!(<MyObject<T>>::exists(object_id));
```

This is already declared in the template helper code, but is an important check to remind ourselves about.

Deposit an event

Once all checks are passed and the new price is written to storage, we can deposit an event just like we did in Part III. Replace the line marked as ACTION number 3 with:

```
Self::deposit_event(Event::PriceSet(sender, kitty_id, new_price));
```

Now whenever the `set_price` dispatchable is called successfully, it will emit a `PriceSet` event. 🎉

Transfer a Kitty

Our pallet's storage items already handles ownership of Kitties — this means that all our `transfer_kitty` function needs to do is update following storage items:

- `KittyOwner`: to update the owner of the Kitty.
- `OwnedKittiesArray`: to update the owned Kitty map for each acount.
- `OwnedKittiesIndex`: to update the owned Kitty index for each owner.
- `OwnedKittiesCount`: to update the amount of Kitties a account has.

You already have the tools you'll need to create the transfer functionality from part 1. The main difference in how we will create this functionality for our runtime is that it will have **two parts to it**:

1. **A dispatchable function called `transfer()`**: this is what's exposed by your pallet.
2. **A private function called `transfer_from()`**: this will be a helper function called by `transfer()` to handle all storage updates when transferring a Kitty.

Separating the logic this way makes the private `transfer_from()` function reusable by other dispatchable functions of our pallet, without needing to rewrite the same logic over and over again.

In our case, we're going to reuse it for the `buy_kitty` dispatchable we're creating in the next section.

Paste in the following snippet to replace ACTION number 4 in the template code:

```
#[pallet::weight(100)]
pub fn transfer(
    origin: OriginFor<T>,
    to: T::AccountId,
    kitty_id: T::Hash,
) -> DispatchResultWithPostInfo {
    let sender = ensure_signed(origin)?;

    // Verify Kitty owner: must be the account invoking this transaction.
    let owner = Self::owner_of(kitty_id).ok_or("No owner for this kitty")?;
    ensure!(owner == sender, "You do not own this kitty");

    // Transfer.
    Self::transfer_from(sender, to, kitty_id)?;

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

Buy a Kitty

Check a Kitty is for Sale

We can use the `set_price()` function to check if the Kitty is for sale. Remember that we said a Kitty with the price of `0` means it's not for sale? Easy enough then: write this check by simple using `ensure!()` (replacing line ACTION number 5):

```
// Check if the Kitty is for sale.
ensure!(!kitty_price.is_zero(), "This Kitty is not for sale!");
ensure!(
    kitty_price <= ask_price,
    "This Kitty is out of your budget!"
);
```

Making a Payment

In the Step 2, we added the functions necessary to transfer the *ownership* of our Kitties. But, in the event that the Kitty was bought or sold, we never actually specified a currrency associated to our pallet.

In this step we'll learn how to use FRAME's Currency trait to adjust account balances using its very own `transfer` method. It's useful to understand why it's important to use the `transfer` method in particular and how we'll be accessing it:

- The reason we'll be using it is to ensure our runtime has the same understanding of currency throughout the pallets it interacts with. The way that we ensure this is to use the `Currency` trait from `frame_support`.

- Conveniently, it handles a `Balance` type, making it compatible with `pallet_balances` which we've been using in our pallet's configuration trait. Take a look at how the `transfer` function we'll be using is structured (from the Rust docs):

```
fn transfer(
    source: &AccountId,
    dest: &AccountId,
    value: Self::Balance,
    existence_requirement: ExistenceRequirement
) -> DispatchResult
```

Now we can finally make use of the `frame_support` imports – `Currency` and `ExistenceRequirement` – that we initially started with in Part I.

Feeling confident?

Here's how you would write `buy_Kitty` from scratch.

Perform basic sanity checks

- it will take 3 arguments: `origin`, `Kitty_id` and `max_price`
- check that `Kitty_id` corresponds to a Kitty in storage
- check that the Kitty has an owner

Check if purchasing a Kitty is authorized

- check that the account buying the Kitty doesn't already own it
- check that the price of the Kitty is not zero (if it is, throw an error)
- check that the Kitty price is not greater than `ask_price`

Update storage items

- use the `transfer` method from the `Currency` trait to update account balances
- use our pallet's `transfer_from` function to change the ownership of the Kitty from `owner` to `sender`

- update the price of the Kitty to the price it was sold at

Your turn! Paste in the following code snippet, replacing ACTION number 6:

```
<pallet_balances::Pallet<T> as Currency<_>>::transfer(
      &sender,
      &owner,
      kitty_price,
      ExistenceRequirement::KeepAlive,
  )?;
```

Now that that the `transfer` method from FRAME's Currency trait has been called, we can call a private helper function called `transfer_from` (which we'll write later) to write the new changes in ownership to storage. Replace this with what's on line ACTION number 7:

```
// Transfer ownership of Kitty.
Self::transfer_from(owner.clone(), sender.clone(), kitty_id).expect(
    "`owner` is shown to own the kitty; \
    `owner` must have greater than 0 kitties, so transfer cannot cause underfl
    `all_kitty_count` shares the same type as `owned_kitty_count` \
    and minting ensure there won't ever be more than `max()` kitties, \
    which means transfer cannot cause an overflow; \
    qed",
);

// Set the price of the Kitty to the new price it was sold at.
kitty.price = ask_price.into();
<Kitties<T>>::insert(kitty_id, kitty);
```

Breed Kitties

The logic behind breeding two Kitties is to multiply each corresponding DNA segment from two Kitties, which will produce a new DNA sequence. Then, that DNA is used when minting a new Kitty.

Paste in the following to complete the `breed_kitty` function, replacing line ACTION number 8:

```
    let random_hash = Self::random_hash(&sender);
    let kitty_1 = Self::kitty(kitty_id_1);
    let kitty_2 = Self::kitty(kitty_id_2);

    let mut final_dna = kitty_1.dna;
```

```
        for (i, (dna_2_element, r)) in kitty_2
            .dna
            .as_ref()
            .iter()
            .zip(random_hash.as_ref().iter())
            .enumerate()
        {
            if r % 2 == 0 {
                final_dna.as_mut()[i] = *dna_2_element;
            }
        }
```

Now that we've used the user inputs of Kitty IDs and combined them to create a new unique Kitty ID, we can use the `mint()` function to wirte that new Kitty to storage (replace line ACTION number 9):

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

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

Of course, after calling the `mint` function, we call `increment_nonce()` to maintain maximum entropy as described in Part II.

Write the transfer_from helper

Similar to writing the `mint` function, the `transfer_from` function is a helper called by the public `buy_kitty` dipatchable to perform all storage updates once a Kitty has been bought and sold.

Copy the following to replace ACTION number 10:

```
// Helper to handle transferring a Kitty from one account to another.
fn transfer_from(
    from: T::AccountId,
    to: T::AccountId,
    kitty_id: T::Hash,
) -> DispatchResult {
    // Verify that the owner is the rightful owner of this Kitty.
    let owner = Self::owner_of(kitty_id).ok_or("No owner for this kitty")?;
```

```
    ensure!(owner == from, "'from' account does not own this kitty");

    // Address to send from.
    let owned_kitty_count_from = Self::owned_kitty_count(&from);

    // Address to send to.
    let owned_kitty_count_to = Self::owned_kitty_count(&to);

    // Increment the amount of owned Kitties by 1.
    let new_owned_kitty_count_to = owned_kitty_count_to
        .checked_add(1)
        .ok_or("Transfer causes overflow of 'to' kitty balance")?;

    // Increment the amount of owned Kitties by 1.
    let new_owned_kitty_count_from = owned_kitty_count_from
        .checked_sub(1)
        .ok_or("Transfer causes underflow of 'from' kitty balance")?;

    // Get current Kitty index.
    let kitty_index = <OwnedKittiesIndex<T>>::get(kitty_id);

    // Update storage items that require updated index.
    if kitty_index != new_owned_kitty_count_from {
        let last_kitty_id =
            <OwnedKittiesArray<T>>::get((from.clone(), new_owned_kitty_count_f
        <OwnedKittiesArray<T>>::insert((from.clone(), kitty_index), last_kitty
        <OwnedKittiesIndex<T>>::insert(last_kitty_id, kitty_index);
    }

    // Write new Kitty ownership to storage items.
    <KittyOwner<T>>::insert(&kitty_id, Some(&to));
    <OwnedKittiesIndex<T>>::insert(kitty_id, owned_kitty_count_to);

    <OwnedKittiesArray<T>>::remove((from.clone(), new_owned_kitty_count_from))
    <OwnedKittiesArray<T>>::insert((to.clone(), owned_kitty_count_to), kitty_i

    <OwnedKittiesCount<T>>::insert(&from, new_owned_kitty_count_from);
    <OwnedKittiesCount<T>>::insert(&to, new_owned_kitty_count_to);

    Self::deposit_event(Event::Transferred(from, to, kitty_id));

    Ok(())
}
```

Write the transfer dispatchable

Now that we've included the `transfer_from` helper function, let's write out the `transfer` dispatchable. This will allow any user to attempt to transfer a Kitty to another account (it's important to *verify first and write last*!).

Paste the following snippet to replace ACTION number 11:

```
#[pallet::weight(100)]
pub fn transfer(
    origin: OriginFor<T>,
    to: T::AccountId,
    kitty_id: T::Hash,
) -> DispatchResultWithPostInfo {
    let sender = ensure_signed(origin)?;

    // Verify Kitty owner: must be the account invoking this transaction.
    let owner = Self::owner_of(kitty_id).ok_or("No owner for this kitty")?;
    ensure!(owner == sender, "You do not own this kitty");

    // Transfer.
    Self::transfer_from(sender, to, kitty_id)?;

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

By now the above pattern should be familiar. We always check that the transaction is signed; then we verify that the Kitty is owned by the sender of this transaction; and last we call the `transfer_from` helper which will update all storage items appropriately.

Genesis configuration

The final step before our pallet is ready to be used is to set the genesis state of our storage items. We'll make use of FRAME's `[pallet::genesis_config]` to do this. Essentially, we're declaring what the Kitties object in storage contains in the genesis block. Copy the following code to replace ACTION number 12:

```
// Our pallet's genesis configuration.
#[pallet::genesis_config]
pub struct GenesisConfig<T: Config> {
    pub kitties: Vec<(T::AccountId, T::Hash, T::Balance)>,
}

// Required to implement default for GenesisConfig.
#[cfg(feature = "std")]
impl<T: Config> Default for GenesisConfig<T> {
```

```
    fn default() -> GenesisConfig<T> {
        GenesisConfig { kitties: vec![] }
    }
}

#[pallet::genesis_build]
impl<T: Config> GenesisBuild<T> for GenesisConfig<T> {
    fn build(&self) {
        for &(ref acct, hash, balance) in &self.kitties {
            let k = Kitty {
                id: hash,
                dna: hash,
                price: balance,
                gender: Gender::Male,
            };

            let _ = <Pallet<T>>::mint(acct.clone(), hash, k);
        }
    }
}
```

Update runtime/lib.rs and interact with your Kitties

If you've completed all of the preceding parts and steps of this tutorial, you're all geared up to run your chain and start interacting with all the new capabilities of your Kitties pallet.

Build and run your chain using the following commands:

```
cargo build --release
./target/release/node-kitties --dev
```

Now check your work using the Polkadot-JS Apps UI just like we did in the previous part. Once your chain is running and connected to the PolkadotJS Apps UI, perform these manual checks:

- Fund multiple users with tokens so they can all participate
- Have each user create multiple Kitties
- Try to transfer a Kitty from one user to another using the right and wrong owner
- Try to set the price of a Kitty using the right and wrong owner
- Buy a Kitty using an owner and another user
- Use too little funds to purchase a Kitty
- Overspend on the cost of the Kitty and ensure that the balance is reduced appropriately

- Breed a Kitty and check that the new DNA is a mix of the old and new

After all of these actions, confirm that all users have the right number of Kitties, the total Kitty count is correct, and any other storage variables are correctly represented

Conclusion

Congratulations! You've successfully created the backend of a fully functional Substrate chain capable of creating and managing Substrate Kitties. It could also be abstracted to other NFT-like use cases. Most importantly, at this point in the tutorial you should have all the knowledge you need to start creating your own pallet logic and dispatchable functions.

About the Author

Contributed by Kresna Sucandra, a medical doctor turned programmer and a Substrate ecosystem contributor.

References

This tutorial is part of Substrate Developer Hub's How-to Guides tutorial by Sacha Lansky

Figment Terms of Use Privacy Policy Contributor Terms