Updating the contract

To reiterate, we'd like anyone to be able to participate in the crossword puzzle, even folks who don't have a NEAR account.

The first person to win will "reserve their spot" and choose where to send the prize money: either an account they own or an account they'd like to create.

Reserving their spot

The plan

When a user first visits the crossword, they only see the crossword. No login button and no fields (like amemo field) to fill out.

On their first visit, our frontend will create a brand new, random seed phrase in their browser. We'll use this seed phrase to create the user's unique key pair. If a random seed phrase is already there, it skips this part. (We covered the code for this ina previous section .)

If the user is the first to solve the puzzle, it discovers the function-call access key and callssubmit_solution with that key. It's basically using someone else's key, as this key is on the crossword account.

We'll be adding a new parameter to thesubmit_solution so the user can include the random, public key we just stored in their browser.

During the execution ofsubmit_solution , because contracts can use Promises to perform Actions, we'll remove the solution public key and add the user's public key.

This will lock out other attempts to solve the crossword puzzle and ensure there is only one winner.

This means that a puzzle can have three states it can be in:

1. Unsolved
2. Solved and not yet claimed (not paid out)
3. Claimed and finalized

The previous chapterwe discussed enums , so this is simply modifying the enumeration variants.

The implementation

First, let's see how thesubmit_solution will verify the correct solution.

contract/src/lib.rs loading ... See full example on GitHub Instead of hashing the plaintext, we simply check that the public key matches what we know the answer is. (The answer being the series of words representing the solution to the crossword puzzle, used as a seed phrase to create a key pair, including a public key.)

Further down in thesubmit_solution method we'll follow our plan byadding a function-call access key (that only the winner has) and removing the access key that was discovered by the winner, so no one else can use it.

Our smart contract is like this carpenter adding a key to itself. Art bycarlcarlkarl.near

contract/src/lib.rs loading ... See full example on GitHub The first promise above adds an access key, and the second deletes the access key on the account that was derived from the solution as a seed phrase.

We delete the function-call access key so there is only one winner. Art byeerie_ram.near

Note that the new function-call access key is able to call two methods we'll be adding:

1. claim_reward
2. — when the user has an existing account and wishes to send the prize to it
3. claim_reward_new_account
4. — when the user doesn't have an account, wants to create one and send the prize to it

Both functions will do cross-contract calls and use callbacks to see the result. We finally get to the meat of this chapter, let's go!

Cross-contract calls

The traits

We're going to be making a cross-contract call to the linkdrop account deployed to thetestnet account. We're also going to have callbacks for that, and for a simple transfer to a (potentially existing) account. We'll create the traits that define both those methods.

contract/src/lib.rs loading ... See full example on GitHub tip It's not necessary to create the trait for the callback as we could have just implemented the functionscallback_after_transfer andcallback_after_create_account in ourCrossword struct implementation. We chose to define the trait and implement it to make the code a bit more readable.

claim_reward

Again, this function is called when the user solves the crossword puzzle and wishes to send the prize money to an existing account.

Seems straightforward, so why would we need a callback? We didn't use a callback in the previous chapter when the user logged in, so what gives?

It's possible that while claiming the prize, the user accidentally fat-fingers their username, or their cat jumps on their keyboard. Instead of typingmike.testnet they typemike.testnzzz and hit send. In short, if we try to send the prize to a non-existent account, we want to catch that.

For brevity, we'll skip some code in this function to focus on the Promise and callback:

pub

fn

claim_reward (& mut

self , crossword_pk :

PublicKey , receiver_acc_id :

String , memo :

String ,)

->

Promise

{ let signer_pk =

env :: signer_account_pk ( ) ; ... Promise :: new (receiver_acc_id . parse ( ) . unwrap ( )) . transfer (reward_amount) . then (Self :: ext (env :: current_account_id ( )) . with_static_gas (GAS_FOR_ACCOUNT_CALLBACK) . callback_after_transfer (crossword_pk , receiver_acc_id , memo , env :: signer_account_pk ( ) ,) ,) } Your IDE is your friend Oftentimes, the IDE can help you.

For instance, in the above snippet we havereceiver_acc_id.parse().unwrap() which might look confusing. You can lean on code examples or documentation to see how this is done, or you can utilize the suggestions from your IDE. Thisclaim_reward method will attempt to use theTransfer Action to send NEAR to the account specified. It might fail on a protocol level (as opposed to a smart contract failure), which would indicate the account doesn't exist.

Let's see how we check this in the callback:

contract/src/lib.rs loading ... See full example on GitHub The#[private] macro Notice that above the function, we have declared it to be private.

This is an ergonomic helper that checks to make sure the predecessor is the current account ID.

We actually saw this done "the long way" in the callback for the linkdrop contract in the previous section .

Every callback will want to have this#[private] macro above it. The snippet above essentially says it expects there to be a Promise result for exactly one Promise, and then sees if that was successful or not. Note that we're not actually getting avalue in this callback, just if it succeeded or failed.

If it succeeded, we proceed to finalize the puzzle, like setting its status to be claimed and finished, removing it from theunsolved_puzzles collection, etc.

claim_reward_new_account

Now we want to handle a more interesting case. We're going to do a cross-contract call to the smart contract located ontestnet and ask it to create an account for us. This name might be unavailable, and this time we get to write a callback thatgets a value .

Again, for brevity, we'll show the meat of theclaim_reward_new_account method:

pub

fn

claim_reward_new_account (& mut

self , crossword_pk :

PublicKey , new_acc_id :

String , new_pk :

PublicKey , memo :

String ,)

->

Promise

{ ... ext_linkdrop :: ext (AccountId :: from (self . creator_account . clone ( ))) . with_attached_deposit (reward_amount) . with_static_gas (GAS_FOR_ACCOUNT_CREATION)

// This amount of gas will be split . create_account (new_acc_id . parse ( ) . unwrap ( ) , new_pk) . then (// Chain a promise callback to ourselves Self :: ext (env :: current_account_id ( )) . with_static_gas (GAS_FOR_ACCOUNT_CALLBACK) . callback_after_create_account (crossword_pk , new_acc_id , memo , env :: signer_account_pk ( ) ,) ,) } Then the callback:

contract/src/lib.rs loading ... See full example on GitHub In the above snippet, there's one difference from the callback we saw inclaim_reward : we capture the value returned from the smart contract we just called. Since the linkdrop contract returns a bool, we can expect that type. (See the comments with "NOTE:" above, highlighting this.)

Callbacks

The way that the callback works is that you start with theSelf::ext() and pass in the current acount ID usingenv::current_account_id() . This is essentially saying that you want to call a function that lives on the current account ID.

You then have a couple of config options that each start with.with_* :

1. You can attach a deposit of Ⓝ, in yoctoⓃ to the call by specifying the.with_attached_deposit()
2. method but it is defaulted to 0 (1 Ⓝ = 10000000000000000000000000 yoctoⓃ, or 1^24 yoctoⓃ).
3. You can attach a static amount of GAS by specifying the.with_static_gas()
4. method but it is defaulted to 0.
5. You can attach an unused GAS weight by specifying the.with_unused_gas_weight()
6. method but it is defaulted to 1. The unused GAS will be split amongst all the functions in the current execution depending on their weights. If there is only 1 function, any weight above 1 will result in all the unused GAS being attached to that function. If you specify a weight of 0, however, the unused GAS will not be attached to that function. If you have two functions, one with a weight of 3, and one with a weight of 1, the first function will get 3/4 of the unused GAS and the other function will get 1/4 of the unused GAS.

After you've added the desired configurations to the call, you execute the function and pass in the parameters. In this case, we call the functioncallback_after_create_account and pass in the crossword public key, the new account ID, the memo, and the signer's public key.

This function will be called with static GAS equal toGAS_FOR_ACCOUNT_CALLBACK and will have no deposit attached. In addition, since thewith_unused_gas_weight() method wasn't called, it will default to a weight of 1 meaning that it will split all the unused GAS with thecreate_account function to be added on top of theGAS_FOR_ACCOUNT_CALLBACK .

. then (// Chain a promise callback to ourselves Self :: ext (env :: current_account_id ( )) . with_static_gas (GAS_FOR_ACCOUNT_CALLBACK) . callback_after_create_account (crossword_pk , new_acc_id , memo , env :: signer_account_pk ( ) ,) ,) Consider changing contract state in callback It's not always the case, but often you'll want to change the contract state in the callback.

The callback is a safe place where we have knowledge of what's happened after cross-contract calls or Actions. If your smart contract is changing statebefore doing a cross-contract call, make sure there's a good reason for it. It might be best to move this logic into the callback. So what parameters should I pass into a callback?

There's no one-size-fits-all solution, but perhaps there's some advice that can be helpful.

Try to pass parameters that would be unwise to trust coming from another source. For instance, if an account calls a method to transfer some digital asset, and you need to do a cross-contract call, don't rely on the results of contract call to determine ownership. If the original function call determines the owner of a digital asset, pass this to the callback.

Passing parameters to callbacks is also a handy way to save fetching data from persistent collections twice: once in the initial method and again in the callback. Instead, just pass them along and save some CPU cycles.

Checking the public key

The last simple change in this section is to modify the way we verify if a user has found the crossword solution.

In previous chapters we hashed the plaintext solution and compared it to the known solution's hash.

Here we're able to simply check the signer's public key, which is available in theenv object[undersigner_account_pk](). .

We'll do this check in both when the solution is submitted, and when the prize is claimed.

When the crossword is solved

// The solver_pk parameter is the public key generated and stored in their browser pub

fn

submit_solution (& mut

self , solver_pk :

PublicKey)

{ let answer_pk =

env :: signer_account_pk ( ) ; // check to see if the answer_pk from signer is in the puzzles let

mut puzzle =

self . puzzles . get (& answer_pk) . expect ("ERR_NOT_CORRECT_ANSWER") ;

When prize is claimed

pub

fn

claim_reward (& mut

self , crossword_pk :

PublicKey , receiver_acc_id :

String , memo :

String ,)

->

Promise

{ let signer_pk =

env :: signer_account_pk ( ) ; ... // Check that puzzle is solved and the signer has the right public key match puzzle . status { PuzzleStatus :: Solved

{ solver_pk : puzzle_pk , }

=>

{ // Check to see if signer_pk matches assert_eq! (signer_pk , puzzle_pk ,

"You're not the person who can claim this, or else you need to use your function-call access key, friend.") ; } _ =>

{ env :: panic_str ("puzzle should have Solved status to be claimed") ; } } ; ... } Edit this page Last updated on Jan 31, 2024 by gagdiez Was this page helpful? Yes No

Previous Linkdrop contract Next Base64 params, wrap up