Actions (including sending NEAR)

We're going to introduce a new Action:Transfer . In this chapter, we'd like the first person to solve the crossword puzzle to earn some prize money, sent in NEAR.

Art byqiqi04.near

We've already used Actions in the previous chapter , when we deployed and initialized the contract, which used theDeployContract andFunctionCall Action, respectively.

The full list of Actions are available at theNEAR specification site .

By the end of this entire tutorial we'll have used all the Actions highlighted below:

Actions from within a contract

When we deployed and initialized the contract, we used NEAR CLI in our Terminal or Command Prompt app. At a high level, this might feel like we're lobbing a transaction into the blockchain, instructing it to do a couple actions.

It's important to note that you can also execute Actions inside a smart contract, which is what we'll be doing. In the sidebar on the left, you'll see a section calledPromises , which provides examples of this. Perhaps it's worth mentioning that for the Rust SDK, Promises and Actions are somewhat synonymous.

Actions only effect the current contract A contract cannot use theAddKey Action on another account, including the account that just called it. It can only add a key toitself , if that makes sense.

The same idea applies for the other actions as well. You cannot deploy a contract to someone else's account, or delete a different account. (Thankfully 😅)

Similarly, when we use theTransfer Action to send the crossword puzzle winner their prize, the amount is being subtracted from the account balance of the account where the crossword contract is deployed.

The only interesting wrinkle (and what mayseem like an exception) is when a subaccount is created using theCreateAccount Action. During that transaction, you may use Batch Actions to do several things like deploy a contract, transfer NEAR, add a key, call a function, etc. This is common in smart contracts that use a factory pattern, and we'll get to this in future chapters of this tutorial.

Define the prize amount

Let's make it simple and hardcode the prize amount. This is how much NEAR will be given to the first person who solves the crossword puzzle, and will apply to all the crossword puzzles we add. We'll make this amount adjustable in future chapters.

At the top of thelib.rs file we'll add this constant:

contract/src/lib.rs loading ... See full example on GitHub As the code comment mentions, this is 5 NEAR, but look at all those zeroes in the code!

That's the value in yoctoNEAR. This concept is similar to other blockchains. Bitcoin's smallest unit is a satoshi and Ethereum's is a wei.

Art byjrbemint.near

AddingTransfer

In the last chapter we had a simple function calledguess_solution that returnedtrue if the solution was correct, andfalse otherwise. We'll be replacing that function withsubmit_solution as shown below:

contract/src/lib.rs loading ... See full example on GitHub Note the last line in this function, which sends NEAR to the predecessor.

Returning a Promise The last line of the function above ends with a semicolon. If the semicolon were removed, that would tell Rust that we'd like to return this Promise object.

It would be perfectly fine to write the function like this:

pub

fn

submit_solution (& mut

self , solution :

String , memo :

String)

->

Promise

{ // … // Transfer the prize money to the winner Promise :: new (env :: predecessor_account_id ( )) . transfer (PRIZE_AMOUNT) }

Predecessor, signer, and current account

When writing a smart contract you'll commonly want to useenv and the details it provides. We used this in the last chapter for:

- logging (ex:env::log_str("hello friend")
-)
- hashing using sha256 (ex:env::sha256(solution.as_bytes())
-)

There are more functions detailed in theSDK reference docs .

Let's cover three commonly-used functions regarding accounts: predecessor, signer, and current account.

Alice sends a transaction to the contract on banana.near, which does a cross-contract call to cucumber.near. From the perspective of a contract on cucumber.near, we see a list of the predecessor, signer, and current account. Art byyasuoarts.near

1. predecessor account
2. —env::predecessor_account_id()
3. This is the account that was the immediate caller to the smart contract. If this is a simple transaction (no cross-contract calls) fromalice.near
4. tobanana.near
5. , the smart contract atbanana.near
6. considers Alice the predecessor. In this case, Alice wouldalso
7. be the signer.
8. When in doubt, use predecessor
9. As we explore the differences between predecessor and signer, know that it's a more commonbest practice to choose the predecessor
10. .
11. Using the predecessor guards against a potentially malicious contract trying to "fool" another contract that only checks the signer.
12. signer account
13. —env::signer_account_id()
14. The signer is the account that originallysigned
15. the transaction that began the blockchain activity, which may or may not include cross-contract calls. If a function calls results in several cross-contract calls, think of the signer as the account that pushed over the first domino in that chain reaction.
16. Beware of middlemen
17. If your smart contract is checking the ownership over some assets (fungible token, NFTs, etc.) it's probably a bad idea to use the signer account.
18. A confused or malicious contract might act as a middleman and cause unexpected behavior. Ifalice.near
19. accidentally callsevil.near
20. , the contract at that account might do a cross-contract call tovulnerable-nft.near
21. , instructing it to transfer an NFT.
22. Ifvulnerable-nft.near
23. only checks the signer account to determine ownership of the NFT, it might unwittingly give away Alice's property. Checking the predecessor account eliminates this problem.
24. current account
25. —env::current_account_id()
26. The current account is "me" from the perspective of a smart contract.
27. Why would I use that?
28. There might be various reasons to use the current account, but a common use case is checking ownership or handling

callbacks to cross-contract calls.

29. Many smart contracts will want to implement some sort of permission system. A common, rudimentary permission allows certain functions to only be called by the contract owner, AKA the person who owns a private key to the account for this contract.
30. The contract can check that the predecessor and current account are the same, and trust offer more permissions like changing contract settings, upgrading the contract, or other privileged modifications. Edit this page Last updatedonJan 31, 2024 bygagdiez Was this page helpful? Yes No

Previous Using structs and enums Next Add a puzzle