Using collections

As mentioned in the previous chapter, theonline Rust Book is a great reference for folks getting started with Rust, but there are concepts that differ when we're dealing with the blockchain. One of these differences is the use of collections.

The reference-level documentation of the Rust SDK explains this concept well:

Motivation for specialized collections Collections that offer an alternative to standard containers from [Rust's] std::collections::* byutilizing the underlying blockchain trie storage more efficiently . For example, the following smart contract does not work with state efficiently, becauseit will load the entire HashMap at the beginning of the contract call , and will save it entirely at the end, in cases when there is state modification.This is fine for small number of elements, but very inefficient for large numbers . —NEAR SDK reference documentation In chapter 1, we set the crossword puzzle solution hash when we first deployed the contract and called the initialization methodnew , passing it. This would only allow us to have only one puzzle, but let's allow for many.

At a high level, let's discuss what we'll want to add if our contract is to store multiple crossword puzzles. First, we'll have the concept of many puzzles where some of them will have different states (unfinished and finished) and we'll want to know which ones are unsolved in quick way. Another thing, which is a general rule of thumb when writing smart contracts, is to anticipate what might happen if it gets a lot of usage. What if we end up with 10,000 crossword puzzles? How might that affect how many data structures we use and which ones?

LookupMap and UnorderedSet

Let's try having two specialized NEAR collections:

1. ALookupMap
2. which will store key-value pairs. (Solution hash » Puzzle object)
3. AnUnorderedSet
4. containing a set (list with no duplicates) of the solution hashes for puzzles which have not been solved yet.

As you look at the list of specialized collections in the Rust SDK, you might notice some begin withLookup while others haveUnordered . As is written in the reference documentation, theLookup is non-iterable while theUnordered collections are iterable. This means if you will need to loop through the list of contents of this data structure, you'll likely use an iterable data structure. If you'll only ever be adding and retrieving data by the key, and the key will always be known, it's more efficient to use a non-iterable collection.

So why would we have two data structures here? Again, if we end up with a large number of puzzles, we might not be able to loop through all the puzzles, looking for ones that are unsolved. Because of the limit of gas execution per transaction, we must be conscious that there can be operations which will eventually exceed this limit. I suppose we could assume that ourUnorderedSet of unsolved puzzles wouldn't contain tens of thousands of puzzles. That's one way to avoid running into limits, but we could also learn how to utilizepagination through an iterable collection like anUnorderedSet which we'll get to later.

Think of our collection as having multiple pages of puzzle hashes. Art bypierced_staggg.near

As we remember from the previous chapter, every smart contract has a primary struct containing the#[near_bindgen] macro.

Naming the primary struct Note in theprevious chapter we named our primary structContract , but in this chapter we'll call itCrossword.

The name of the struct doesn't matter and there's nothing special about naming itContract , though you might see that convention used in several smart contracts on NEAR. We've named it something different simply to illustrate that there's no magic behind the scenes. Thisdoes mean, however, that ourimpl block will also beCrossword . Here's how our struct will look with the iterable and non-iterable NEAR collections:

contract/src/lib.rs loading ... See full example on GitHub Above, we have thepuzzles andunsolved_puzzles fields which are collections.

We also have anowner_id so we can exercise a common pattern in smart contract development: implementing a rudimentary permission system which can restrict access to certain functions. We'll expand on this thought in a moment.

The snippet below shows the first method in the implementation of theCrossword struct, where thenew function sets up these two specialized collections.

contract/src/lib.rs loading ... See full example on GitHub So during the initialization function (new) we're setting theowner_id . For our purposes the owner will likely be the contract itself, but there can be several reasons to have it be a DAO or another user. Next, let's look at theb"c" andb"u" bits for the collection fields.

Collections have prefixes

Above, thenew function is initializing the struct's fields by giving them a unique prefix. You can learn more about the prefixes here , but know that these prefixes (c andu) should be short and different.

Let's take a peek at how we'll add a new crossword puzzle. Note that there will be a new struct here,Answer , which we haven't defined yet. We'll also be introducing the concept of enums, likePuzzleStatus::Solved andPuzzleStatus::Unsolved . We'll be covering these in the next section.

Unlike the previous chapter where there was only one crossword puzzle, we'll be inserting into our new collections, so let's create anew_puzzle method.

contract/src/lib.rs loading ... See full example on GitHub Now we're set up to store multiple puzzles!

Permissions or permissionless?

Guarding which accounts can enter the smart contract logic. Art byconnoisseur_dane.near

Is NEAR permissionless?

Yes.

What did you mean by a permission system earlier, and what are the ways you can control permissions?

There are two ways that permissions can be controlled:

1. In the smart contract code itself
2. When using function-call access keys

We'll get to the second topic in later in this chapter, but will focus on the first item.

As you can see in the previous snippet, the first thing that happens in thenew_puzzle method is a check. It looks to see if the predecessor (the person who most recently called this method, sometimes the same as the signer) is the same as theowner_id that we set during the contract's initialization.

If someone else is trying to callnew_puzzle , this check will fail and the smart contract will panic, going no further. This example is the simplest form of a permission. Much more complex system can exist for users. The SputnikDAO smart contracts, for instance, implement custom policies. It's up the smart contract developer to write their roles/policies and apply them to users. Sometimes an allow-list (or whitelist) is used.

In short,any account with a full-access key can call any method on a smart contract , but that doesn't mean the smart contract will let them continue execution. It's up to the developer to protect their functions with guards like the one innew_puzzle .

Let's dive into structs and enums next. Edit this page Last updatedonJan 19, 2024 byDamián Parrino Was this page helpful? Yes No

Previous Overview Next Using structs and enums