Hash the solution, add basic unit tests

In the previous section, we stored the crossword solution as plain text as aString type on the smart contract. If we're trying to hide the solution from the users, this isn't a great approach as it'll be public to anyone looking at the state. Let's instead hash our crossword solution and store that instead. There are different ways to hash data, but let's usesha256 which is one of the hashing algorithms available inthe Rust SDK .

Remind me about hashing Without getting into much detail, hashing is a "one-way" function that will output a result from a given input. If you have input (in our case, the crossword puzzle solution) you can get a hash, but if you have a hash you cannot get the input. This basic idea is foundational to information theory and security.

Later on in this tutorial, we'll switch from usingsha256 to using cryptographic key pairs to illustrate additional NEAR concepts.

Learn more about hashing fromEvgeny Kapun 's presentation on the subject. You may find other NEAR-related videos from the channel linked in the screenshot below.

Helper unit test during rapid iteration

As mentioned in the first section of thisBasics chapter, our smart contract is technically a library as defined in the manifest file. For our purposes, a consequence of writing a library in Rust is not having a "main" function that runs. You may find many online tutorials where the commandcargo run is used during development. We don't have this luxury, but we can use unit tests to interact with our smart contract. This is likely more convenient than building the contract, deploying to a blockchain network, and calling a method.

We'll add a dependency to thehex crate to make things easier. As you may remember, dependencies live in the manifest file.

contract/Cargo.toml loading ... See full example on GitHub Let's write a unit test that acts as a helper during development. This unit test will sha256 hash the input"near nomicon ref finance" and print it in a human-readable, hex format. (We'll typically put unit tests at the bottom of thelib.rs file.)

[cfg(test)]

mod

tests

{ use

super :: * ; use

near_sdk :: test_utils :: { get_logs ,

VMContextBuilder } ; use

near_sdk :: { testing_env ,

AccountId } ;

[test]

fn

debug_get_hash ( )

{ // Basic set up for a unit test testing_env! (VMContextBuilder :: new ( ) . build ( )) ;

// Using a unit test to rapidly debug and iterate let debug_solution =

"near nomicon ref finance" ; let debug_hash_bytes =

env :: sha256 (debug_solution . as_bytes ( )) ; let debug_hash_string =

hex :: encode (debug_hash_bytes) ; println! ("Let's debug: {:?}" , debug_hash_string) ; } } What is that{:?} thing? Take a look at different formatting traits that are covered in thestd Rust docs regarding this. This is aDebug formatting trait and can prove to be useful during development. Run the unit tests with the command:

cargo test -- --nocapture You'll see this output:

… running 1 test Let's debug: "69c2feb084439956193f4c21936025f14a5a5a78979d67ae34762e18a7206a0f" test tests::debug_get_hash ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s This means when you sha256 the input"near nomicon ref finance" it produces the hash:69c2feb084439956193f4c21936025f14a5a5a78979d67ae34762e18a7206a0f

Note on the test flags You may also run tests using:

cargo test Note that the test command we ran had additional flags. Those flags told Rustnot to hide the output from the tests. You can read more about this inthe cargo docs . Go ahead and try running the tests using the command above, without the additional flags, and note that we won't see the debug message. The unit test above is meant for debugging and quickly running snippets of code. Some may find this a useful technique when getting familiar with Rust and writing smart contracts. Next we'll write a real unit test that applies to this early version of our crossword puzzle contract.

Write a regular unit test

Let's add this unit test (inside themod tests {} block, under our previous unit test) and analyze it:

contract/src/lib.rs loading ... See full example on GitHub The first few lines of code will be used commonly when writing unit tests. It uses theVMContextBuilder to create some basic context for a transaction, then sets up the testing environment.

Next, an object is created representing the contract and theset_solution function is called. After that, theguess_solution function is called twice: first with the incorrect solution and then the correct one. We can check the logs to determine that the function is acting as expected.

Note on assertions This unit test uses theassert_eq! macro. Similar macros likeassert! andassert_ne! are commonly used in Rust. These are great to use in unit tests. However, these will add unnecessary overhead when added to contract logic, and it's recommended to use therequire! macro . See more information on this andother efficiency tips here . Again, we can run all the unit tests with:

cargo test -- --nocapture Run only one test To only run this latest test, use the command:

cargo test check_guess_solution -- --nocapture

Modifyingset_solution

Theoverview section of this chapter tells us we want to have a single crossword puzzle and the user solving the puzzle should not be able to know the solution. Using a hash addresses this, and we can keepcrossword_solution 's field type, asString will work just fine. The overview also indicates we only want the author of the crossword puzzle to be able to set the solution. As it stands, our functionset_solution can be called by anyone with a full-access key. It's trivial for someone to create a NEAR account and call this function, changing the solution. Let's fix that.

Let's have the solution be set once, right after deploying the smart contract.

Here we'll use the#[near_bindgen] macro on a function callednew , which is a common pattern.

contract/src/lib.rs loading ... See full example on GitHub Let's call this method on a fresh contract.

Build (for Windows it's build.bat)

./build.sh

Create fresh account if you wish, which is good practice

near delete crossword.friend.testnet friend.testnet near create-account crossword.friend.testnet --masterAccount friend.testnet

Deploy

near deploy crossword.friend.testnet --wasmFile res/my_crossword.wasm

Call the "new" method

near call crossword.friend.testnet new '{"solution": "69c2feb084439956193f4c21936025f14a5a5a78979d67ae34762e18a7206a0f"}' --accountId crossword.friend.testnet Now the crossword solution, as a hash, is stored instead. If you try calling the last command again, you'll get the error message, thanks to the#[init] macro:The contract has already been initialized

First use of Batch Actions

This is close to what we want, but what if a person deploys their smart contract andsomeone else quickly calls thenew function before them? We want to make sure the same person who deployed the contract sets the solution, and we can do this using Batch Actions. Besides, why send two transactions when we can do it in one? (Technical details covered in the spec for abatch transaction here .)

Art bydobulyo.near

Batch Actions in use Batch Actions are common in this instance, where we want to deploy and call an initialization function. They're also common when using a factory pattern, where a subaccount is created, a smart contract is deployed to it, a key is added, and a function is called.

Here's a truncated snippet from a useful (though somewhat advanced) repository with a wealth of useful code:

staking-pool-factory/src/lib.rs loading ... See full example on GitHub We'll get into Actions later in this tutorial, but in the meantime here's a handyreference from the spec . As you can from the info bubble above, we can batchDeploy andFunctionCall Actions. This is exactly what we want to do for our crossword puzzle, and luckily, NEAR CLI has aflag especially for this .

Let's run this again with the handy--initFunction and--initArgs flags:

Create fresh account if you wish, which is good practice

near delete crossword.friend.testnet friend.testnet near create-account crossword.friend.testnet --masterAccount friend.testnet

Deploy

near deploy crossword.friend.testnet --wasmFile res/my_crossword.wasm \ --initFunction 'new' \ --initArgs '{"solution": "69c2feb084439956193f4c21936025f14a5a5a78979d67ae34762e18a7206a0f"}' Now that we're using Batch Actions, no one can call thisnew method before us.

Batch action failures If one Action in a set of Batch Actions fails, the entire transaction is reverted. This is good to note because sharded, proof-of-stake systems do not work like proof-of-work where a complex transaction with multiple cross-contract calls reverts if one call fails. With NEAR, cross-contract calls use callbacks to ensure expected behavior, but we'll get to that later.

Get ready for our frontend

In the previous section we showed that we could use acurl command to view the state of the contract without explicitly having a function that returns a value from state. Now that we've demonstrated that and hashed the solution, let's add a short view-only functionget_solution .

In the next section we'll add a simple frontend for our single, hardcoded crossword puzzle. We'll want to easily call a function to get the final solution hash. We can use this opportunity to remove the functionget_puzzle_number and the constant it returns, as these were use for informative purposes.

We'll also modify ourguess_solution to return a boolean value, which will also make things easier for our frontend.

contract/src/lib.rs loading ... See full example on GitHub Theget_solution method can be called with:

near view crossword.friend.testnet get_solution In the next section we'll add a simple frontend. Following chapters will illustrate more NEAR concepts built on top of this idea. Edit this page Last updatedonJan 19, 2024 byDamián Parrino Was this page helpful? Yes No

Previous Add basic code, create a subaccount, and call methodsNext Add simple frontend