Modifying the contract

This section will modify the smart contract skeleton from the previous section. This tutorial will start by writing a contract in a somewhat useless way in order to learn the basics. Once we've got a solid understanding, we'll iterate until we have a crossword puzzle.

Add a const, a field, and functions

Let's modify the contract to be:

src/lib.rs loading ... See full example on GitHub We've done a few things here:

1. Set a constant for the puzzle number.
2. Added the fieldcrossword_solution
3. to our main struct.
4. Implemented three functions: one that's view-only and two that are mutable, meaning they have the ability to change state.
5. Used logging, which required the import ofenv
6. from ournear_sdk
7. crate.

Before moving on, let's talk about these changes and how to think about them, beginning with the constant:

const PUZZLE_NUMBER: u8 = 1;

This is an in-memory value, meaning that when the smart contract is spun up and executed in the virtual machine, the value1 is contained in the contract code. This differs from the next change, where a field is added to the struct containing the#[near_bindgen] macro. The fieldcrossword_solution has the type ofString and, like any other fields added to this struct, the value will live inpersistent storage . With NEAR, storage is "paid for" via the native NEAR token (Ⓝ). It is not "state rent" but storage staking, paid once, and returned when storage is deleted. This helps incentivize users to keep their state clean, allowing for a more healthy chain. Read more aboutstorage staking here .

Let's now look at the three new functions:

pub

fn

get_puzzle_number (& self)

->

u8

{ PUZZLE_NUMBER } As is covered in themutability section of these docs , a "view-only" function will have open parenthesis around&self while "change methods" or mutable functions will have&mut self . In the function above, thePUZZLE_NUMBER is returned. A user may call this method using the proper RPC endpoint without signing any transaction, since it's read-only. Think of it like a GET request, but using RPC endpoints that aredocumented here .

Mutable functions, on the other hand, require a signed transaction. The first example is a typical approach where the user supplies a parameter that's assigned to a field:

pub

fn

set_solution (& mut

self , solution :

String)

{ self . crossword_solution = solution ; } The next time the smart contract is called, the contract's fieldcrossword_solution will have changed.

The second example is provided for demonstration purposes:

pub

fn

guess_solution (& mut

self , solution :

String)

{ if solution ==

self . crossword_solution { env :: log_str ("You guessed right!") }

else

{ env :: log_str ("Try again.") } } Notice how we're not saving anything to state and only logging? Why does this need to be mutable?

Well, logging is ultimately captured inside blocks added to the blockchain. (More accurately, transactions are contained in chunks and chunks are contained in blocks. More info in theNomicon spec .) So while it is not changing the data in the fields of the struct, it does cost some amount of gas to log, requiring a signed transaction by an account that pays for this gas.

Building and deploying

Here's what we'll want to do:

Art byjeheycell.near

Build the contract

The skeleton of the Rust contract we copied from the previous section has abuild.sh andbuild.bat file for OS X / Linux and Windows, respectively. For more details on building contracts, please seethis section .

Run the build script and expect to see the compiled Wasm file copied to theres folder, instead of buried in the default folder structure Rust sets up.

./build.sh

Create a subaccount

If you've followed from the previous section, you have NEAR CLI installed and a full-access key on your machine. While developing, it's a best practice to create a subaccount and deploy the contract to it. This makes it easy to quickly delete and recreate the subaccount, which wipes the state swiftly and starts from scratch. Let's use NEAR CLI to create a subaccount and fund with 1 NEAR:

near create-account crossword.friend.testnet --masterAccount friend.testnet --initialBalance 1

If you look again in your home directory's.near-credentials , you'll see a new key for the subaccount with its own key pair. This new account is, for all intents and purposes, completely distinct from the account that created it. It might as well bealice.testnet , as it has, by default, no special relationship with the parent account. To be clear,friend.testnet cannot delete or deploy tocrossword.friend.testnet unless it's done in a single transaction using Batch Actions, which we'll cover later.

Subaccount nesting It's possible to have the accountanother.crossword.friend.testnet , but this account must be created bycrossword.friend.testnet .

friend.testnet cannot createanother.crossword.friend.testnet because accounts may only create a subaccount that's "one level deeper."

See this visualization where two keys belonging tomike.near are able to createnew.mike.near . We'll get into concepts around access keys later.

Art byseanpineda.near We won't get into top-level accounts or implicit accounts, but you may read moreabout that here .

Now that we have a key pair for our subaccount, we can deploy the contract to testnet and interact with it!

What's a codehash?

We're almost ready to deploy the smart contract to the account, but first let's take a look at the account we're going to deploy to. Remember, this is the subaccount we created earlier. To view the state easily with NEAR CLI, you may run this command:

near state crossword.friend.testnet

What you'll see is something like this:

{ amount :

'6273260568737488799170194446' , block_hash :

'CMFVLYy6UP6c6vrWiSf1atWviayfZF2fgPoqKeUVtLhi' , block_height :

61764892 , code_hash :

'11111111111111111111111111111111' , locked :

'0' , storage_paid_at :

0 , storage_usage :

4236 , formattedAmount :

'6,273.260568737488799170194446' } Note thecode_hash here is all ones. This indicates that there is no contract deployed to this account.

Let's deploy the contract (to the subaccount we created) and then check this again.

Deploy the contract

Ensure that in your command line application, you're in the directory that contains theres directory, then run:

near deploy crossword.friend.testnet --wasmFile res/my_crossword.wasm Congratulations, you've deployed the smart contract! Note that NEAR CLI will output a link toNEAR Explorer where you can inspect details of the transaction.

Lastly, let's run this command again and notice that thecode_hash is no longer all ones. This is the hash of the smart contract deployed to the account.

near state crossword.friend.testnet Note : deploying a contract is often done on the command line. While it may betechnically possible to deploy via a frontend, the CLI is likely the best approach. If you're aiming to use a factory model, (where a smart contract deploys contract code to a subaccount) this isn't covered in the tutorial, but you may reference thecontracts in SputnikDAO .

Call the contract methods (interact!)

Let's first call the method that's view-only:

near view crossword.friend.testnet get_puzzle_number Your command prompt will show the result is1 . Since this method doesn't take any arguments, we don't pass any. We could have added'{}' to the end of the command as well.

Next, we'll add a crossword solution as a string (later we'll do this in a better way) argument:

near call crossword.friend.testnet set_solution '{"solution": "near nomicon ref finance"}' --accountId friend.testnet Windows users Windows users will have to modify these commands a bit as the Command Prompt doesn't like single quotes as we have above. The command must use escaped quotes like so:

near call crossword.friend.testnet set_solution "{\"solution\": \"near nomicon ref finance\"}" --accountId friend.testnet Note that we used NEAR CLI'sview command , and didn't include an--accountId flag. As mentioned earlier, this is because we are not signing a transaction. This second method uses the NEAR CLIcall command which does sign a transaction and requires the user to specify a NEAR account that will sign it, using the credentials files we looked at.

The last method we have will check the argument against what is stored in state and write a log about whether the crossword solution is correct or incorrect.

Correct:

near call crossword.friend.testnet guess_solution '{"solution": "near nomicon ref finance"}' --accountId friend.testnet You'll see something like this:

Notice the log we wrote is output as well as a link to NEAR Explorer.

Incorrect:

near call crossword.friend.testnet guess_solution '{"solution": "wrong answers here"}' --accountId friend.testnet As you can imagine, the above command will show something similar, except the logs will indicate that you've given the wrong solution.

Reset the account's contract and state

We'll be iterating on this smart contract during this tutorial, and in some cases it's best to start fresh with the NEAR subaccount we created. The pattern to follow is todelete the account (sending all remaining testnet Ⓝ to a recipient) and thencreate the account again.

Sorry, your browser doesn't support embedded videos. Deleting a recreating a subaccount will clear the state and give us a fresh start. Animation byiambon.near Using NEAR CLI, the commands will look like this:

near delete crossword.friend.testnet friend.testnet near create-account crossword.friend.testnet --masterAccount friend.testnet The first command deletescrossword.friend.testnet and sends the rest of its NEAR tofriend.testnet .

Wrapping up

So far, we're writing a simplified version of smart contract and approaching the crossword puzzle in a novice way. Remember that blockchain is an open ledger, meaning everyone can see the state of smart contracts and transactions taking place.

How would you do that? You may hit an RPC endpoint corresponding toview_state and see for yourself. Note: this quick example serves as demonstration purposes, but note that the string being returned is Borsh-serialized and contains more info than just the letters.

curl -d '{"jsonrpc": "2.0", "method": "query", "id": "see-state", "params": {"request_type": "view_state", "finality": "final", "account_id": "crossword.friend.testnet", "prefix_base64": ""}}' -H 'Content-Type: application/json' https://rpc.testnet.near.org

More on this RPC endpoint in theNEAR docs . In this section, we saved the crossword solution as plain text, which is likely not a great idea if we want to hide the solution to players of this crossword puzzle. Even though we don't have a function calledshow_solution that returns the struct'scrossword_solution field, the value is stored transparently in state. We won't get into viewing contract state at this moment, but know it's rather easyand documented here .

The next section will explore hiding the answer from end users playing the crossword puzzle.Edit this page Last updatedonJan 19, 2024 byDamián Parrino Was this page helpful? Yes No

Previous Set up Rust and a contract skeleton Next Hash the solution, unit tests, and an init method