Deploying Contracts

You might want your smart contract to deploy subsequent smart contract code for a few reasons:

- The contract acts as a Factory, a pattern where a parent contract creates many child contracts (Mintbase
- does this to create a new NFT store foranyone who wants one
- ;Rainbow Bridge
- does this to deploy separate Fungible Token contracts foreach bridged token
-)
- The contractupdates its own code
- (callsdeploy
- on itself).
- You could implement a "contract per user" system that creates app-specific subaccounts for users (your-app.user1.near
- ,your-app.user2.near
- , etc) and deploys the same contract to each. This is currently prohibitively expensive due to NEAR'sstorage fees
- , but that may be optimized in the future. If it is, this sort of "sharded app design" may become the more scalable, user-centric approach to contract standards and app mechanics. An early experiment with this paradigm was calledMeta NEAR
- .

If your goal is to deploy to a subaccount of your main contract like Mintbase or the Rainbow Bridge, you will also need to create the account. So, combining concepts from the last few pages, here's what you need:

const

CODE :

& [u8]

=

include_bytes! ("./path/to/compiled.wasm") ;

Promise :: new ("subaccount.example.near" . parse ( ) . unwrap ( )) . create_account ( ) . add_full_access_key (env :: signer_account_pk ( )) . transfer (3_000_000_000_000_000_000_000_000)

// 3e24yN, 3N . deploy_contract (CODE . to_vec ( )) Here's what a full contract might look like, showing a naïve way to passcode as an argument rather than hard-coding it withinclude_bytes! :

use

near_sdk :: { env , near_bindgen ,

AccountId ,

Balance ,

Promise } ;

const

INITIAL_BALANCE :

Balance

=

3_000_000_000_000_000_000_000_000 ;

// 3e24yN, 3N

[near_bindgen]

pub

struct

Contract

{ }

[near_bindgen]

impl

Contract

{

[private]

pub

fn

create_child_contract (prefix :

AccountId , code :

Vec < u8

)

->

Promise

{ let subaccount_id =

AccountId :: new_unchecked (format! ("{}.{}" , prefix ,

env :: current_account_id ( ))) ; Promise :: new (subaccount_id) . create_account ( ) . add_full_access_key (env :: signer_account_pk ( )) . transfer (INITIAL_BALANCE) . deploy_contract (code) } } Why is this a naïve approach? It could run into issues because of the 4MB transaction size limit – the function above would deserialize and heap-allocate a whole contract. For many situations, theinclude_bytes! approach is preferable. If you really need to attach compiled Wasm as an argument, you might be able to copy the approachused by Sputnik DAO v2 . Edit this page Last updatedonOct 19, 2022 bygagdiez Was this page helpful? Yes No

Previous Creating Accounts Next Basic Instructions