NearBindgen

The@NearBindgen({}) decorator is used on the contract class to generate the necessary code to be a valid NEAR contract and expose the intended functions to be able to be called externally.

For example, on a simple counter contract, the decorator will be applied as such:

import

{

NearBindgen , near , call , view }

from

'near-sdk-js'

@ NearBindgen ({ }) class

Counter

{ val : number =

0 ;

@ view ({ })

// Public read-only method: Returns the counter value. get_num ( ) : number { return

this . val }

@ call ({ })

// Public method: Increment the counter. increment ( )

{ this . val

+=

1 ; near . log (Increased number to { this . val }) }

@ call ({ })

// Public method: Decrement the counter. decrement ( )

{ this . val

-=

1 ; near . log (Decreased number to { this . val }) }

@ call ({ })

// Public method - Reset to zero. reset ( )

{ this . val

=

0 ; near . log (Reset counter to zero) } } In this example, theCounter class represents the smart contract state and anything that implements serialization and deserialization methods can be included, such ascollections , which will be covered in the next section. Whenever a function is called, the state will be loaded and deserialized, so it's important to keep this amount of data loaded as minimal as possible.

The core interactions that are important to keep in mind:

- Anycall
- orview
- orinitialize
- functions will be callable externally from any account/contract.* For more information, seepublic methods
- view
- orcall

- decorators can be used in multiple ways to control the mutability of the contract:* Functions that are decorated withview
  - will be read-only and do not write the updated state to storage
  - Functions that are decorated withcall
  - allow for mutating state, and state will always be written back at the end of the function call
- Exposed functions can omit reading and writing to state if class variables are not accessed in the function* This can be useful for some static functionality or returning data embedded in the contract code

Initialization Methods

By default, thedefault() implementation of a contract will be used to initialize a contract. There can be a custom initialization function which takes parameters or performs custom logic with the following@initialize({}) decorator:

@ NearBindgen ({ }) class

Counter

{ @ initialize ({ }) init (val) :

void

{ this . val

= val ; } }

Payable Methods

call method decorators can be annotated with{ payableFunction: true } to allow tokens to be transferred with the method invocation. For more information, seepayable methods .

To declare a function as payable, use the annotation as follows:

@ NearBindgen ({ }) class

Counter

{ @ call ({

payableFunction :

true

}) increment ( ) :

void

{ this . val

+=

1 ; } }

Private Methods

Some methods need to be exposed to allow the contract to call a method on itself through a promise, but want to disallow any other contract to call it. For this, use the{ privateFunction: true } annotation to throw an error when this method is called externally. Seeprivate methods for more information.

This annotation can be applied to any method through the following:

@ NearBindgen ({ }) class

Counter

{ @ call ({

privateFunction :

true

}) private_increment ( ) :

void

{ this . val

+=

1 ; } }Edit this page Last updatedonNov 16, 2023 byLyudmil Ivanov Was this page helpful? Yes No

Previous Getting Started Next Collections