

near_bindgen

The `#[near_bindgen]` macro is used on a struct and the function implementations 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 macro will be applied as such:

```
use  
  
near_sdk :: borsh :: { self ,  
BorshDeserialize ,  
BorshSerialize } ; use  
  
near_sdk :: near_bindgen ;
```

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize, Default)]

```
pub  
  
struct  
  
Counter  
  
{ value :  
u64 , }
```

[near_bindgen]

```
impl  
  
Counter  
  
{ pub  
  
fn  
  
increment ( & mut  
self )  
{ self . value +=  
1 ; }  
  
pub  
  
fn  
  
get_count ( & self )  
  
->  
  
u64
```

`{ self . value }` } In this example, the `Counter` struct represents the smart contract state and anything that implements `BorshSerialize` and `BorshDeserialize` can be included, even `collections`, 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.

[near_bindgen]

also annotates the impl for Counter and this will generate any necessary boilerplate to expose the functions. The core interactions that are important to keep in mind:

- Anypub
- functions will be callable externally from any account/contract.* For more information, see [public methods](#)
- self
- can be used in multiple ways to control the [mutability of the contract](#)
- :* Functions that take &self
- - orself
- - will be read-only and do not write the updated state to storage
- - Functions that take &mut self
- - 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 self
- is not included in the function params* This can be useful for some static functionality or returning data embedded in the contract code
- If the function has a return value, it will be serialized and attached as a result through env::value_return

Initialization Methods

By default, the Default::default() 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 #[init] annotation:

[near_bindgen]

```
impl
```

```
Counter
```

```
{
```

[init]

```
pub
```

```
fn
```

```
new ( value :
```

```
u64 )
```

```
->
```

```
Self
```

```
{ log! ( "Custom counter initialization!" ) ; Self
```

```
{ value } } }
```

All contracts are expected to implement Default . If you would like to prohibit the default implementation from being used, the PanicOnDefault derive macro can be used:

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize, PanicOnDefault)]

```
pub
```

```
struct
```

```
Counter
```

```
{ // ... }
```

Payable Methods

Methods can be annotated with `#[payable]` to allow tokens to be transferred with the method invocation. For more information, see [payable methods](#).

To declare a function as payable, use the `#[payable]` annotation as follows:

[payable]

```
pub
fn
my_method ( & mut
self )
{ ... }
```

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 `#[private]` annotation to panic when this method is called externally. See [private methods](#) for more information.

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

[private]

```
pub
fn
my_method ( & mut
self )
```

`{ ... }` Interaction with other macros When `near_bindgen` is built for the `wasm32` target, it generates the external NEAR contract bindings. To achieve this it is actually generating another function with the signature `pub extern "C" fn function_name() that first deserializes the contract struct from NEAR storage and then calls thecontract.function_name(parameter1, parameter2, ...) . If you have annotated your function with any attribute-like macros, these are then executedtwice . Specifically if the attribute like macro makes any modification to the function signature, or inserts any code that depends on the contract struct (in the form of &self , &mut self , or self) this will fail in the second invocation, because the externally exposed function does not have any concept of this struct. It is possible to detect this by checking which build target you are building for and limit the execution of the macro to operate only on the first pass. Edit this page Last updated on Dec 11, 2023 by Damian Parrino Was this page helpful? Yes No`

[Previous](#) [Getting Started](#) [Next](#) [Collections](#)