

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 for [anyone who wants one](#)
- [;Rainbow Bridge](#)
- does this to deploy separate Fungible Token contracts for [each bridged token](#)
-)
- The contract [updates its own code](#)
- (calls `deploy`
- 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's [storage 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 called [Meta](#)
[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:

```
import
```

```
{ includeBytes ,
```

```
NearPromise , near }
```

```
from
```

```
"near-sdk-js" ;
```

```
const
```

```
CODE
```

```
=
```

```
includeBytes ( "./res/contract.wasm" ) ;
```

```
NearPromise . new ( "subaccount.example.near" ) . createAccount ( ) . addFullAccessKey ( near . signerAccountPk ( ) ) .  
transfer ( BigInt ( 3_000_000_000_000_000_000_000 ) )
```

```
// 3e24yN, 3N . deployContract ( CODE ) ; Here's what a full contract might look like, showing a naïve way to passcode as  
an argument rather than hard-coding it with includeBytes :
```

```
import
```

```
{
```

```
NearPromise , near , validateAccountId }
```

```
from
```

```
"near-sdk-js" ;
```

```
const
```

```
INITIAL_BALANCE
```

```
=
```

```
BigInt ( 3_000_000_000_000_000_000_000 ) ;
```

```
// 3e24yN, 3N
```

```
@ NearBindgen ( { } ) export
```

```
class
```

Contract

```
{ @ call ( {
```

```
privateFunction :
```

```
true
```

```
} ) createAccount ( { prefix , code } )
```

```
{ const subAccountId =
```

```
{ prefix } . { near . currentAccountId ( ) } ; validateAccountId ( subAccountId ) ; NearPromise . new ( subAccountId ) . createAccount  
( ) . addFullAccessKey ( near . signerAccountPk ( ) ) . transfer ( INITIAL_BALANCE ) . deployContract ( 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, theincludeBytes 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 updated on Aug 4, 2023 by Luís Freitas Was this page helpful? Yes No
```

[Previous Creating Accounts](#) [Next Basic Instructions](#)