Contract Deployment

To add contracts to your application, we'll start by creating a newaztec-nargo project. We'll then compile the contracts, and write a simple script to deploy them to our Sandbox.

info Follow the instructionshere to installaztec-nargo if you haven't done so already.

Initialize Aztec project

Create a newcontracts folder, and from there, initialize a new project calledtoken :

mkdir contracts && cd contracts aztec-nargo new --contract token Then, open thecontracts/token/Nargo.toml configuration file, and add theaztec.nr andvalue_note libraries as dependencies:

[dependencies] aztec

=

{

git

"https://github.com/AztecProtocol/aztec-packages/" ,

tag

"aztec-packages-v0.28.1" ,

directory

"noir-projects/aztec-nr/aztec"

} authwit

=

{

git

"https://github.com/AztecProtocol/aztec-packages/" ,

tag

"aztec-packages-v0.28.1" ,

directory

"noir-projects/aztec-nr/authwit" } compressed_string

=

{ git = "https://github.com/AztecProtocol/aztec-packages/" ,

tag

"aztec-packages-v0.28.1" ,

directory

"noir-projects/aztec-nr/compressed-string" } Last, copy-paste the code from theToken contract intocontracts/token/main.nr :

token_all mod

types ;

// Minimal token implementation that supports AuthWit accounts. // The auth message follows a similar pattern to the cross-chain message and includes a designated caller. // The designated caller is ALWAYS used here, and not based on a flag as cross-chain. // message hash = H([caller, contract, selector, ...args]) // To be read as caller calls function at contract defined by selector with args // Including a nonce in the message hash ensures that the message can only be used once.

contract Token

{ // Libs

use

dep :: compressed_string :: FieldCompressedString ;

use

dep :: aztec :: prelude :: { NoteGetterOptions ,

NoteHeader ,

Map ,

PublicMutable ,

SharedImmutable ,

PrivateSet , FunctionSelector ,

AztecAddress } ; use

dep :: aztec :: hash :: compute_secret_hash ;

use

dep :: authwit :: { auth :: { assert_current_call_valid_authwit , assert_current_call_valid_authwit_public } } ;

use

crate :: types :: { transparent_note :: TransparentNote ,

token_note :: { TokenNote ,

TOKEN_NOTE_LEN } ,

balances_map :: BalancesMap } ;

struct

Storage

{ admin :

PublicMutable < AztecAddress

        , minters :

Map < AztecAddress ,

PublicMutable < bool

                , balances :

BalancesMap < TokenNote

        , total_supply :

PublicMutable < U128

        , pending_shields :

PrivateSet < TransparentNote

, public_balances :

Map < AztecAddress ,

PublicMutable < U128

, symbol :

SharedImmutable < FieldCompressedString

, name :

SharedImmutable < FieldCompressedString

, decimals :

SharedImmutable < u8

, }

[aztec(public)]

[aztec(initializer)]

fn

constructor (admin :

AztecAddress , name :

str < 31

, symbol :

str < 31

, decimals :

u8)

{ assert (! admin . is_zero ( ) ,

"invalid admin") ; storage . admin . write (admin) ; storage . minters . at (admin) . write (true) ; storage . name . initialize (FieldCompressedString :: from_string (name)) ; storage . symbol . initialize (FieldCompressedString :: from_string (symbol)) ; storage . decimals . initialize (decimals) ; }

[aztec(public)]

fn

set_admin (new_admin :

AztecAddress)

{ assert (storage . admin . read ( ) . eq (context . msg_sender ( )) ,

"caller is not admin") ; storage . admin . write (new_admin) ; }

[aztec(public)]

fn

public_get_name ( )

->

pub

FieldCompressedString

{ storage . name . read_public ( ) }

[aztec(private)]

fn

private_get_name ( )

->

pub

FieldCompressedString

{ storage . name . read_private ( ) }

unconstrained fn

un_get_name ( )

->

pub

[u8 ;

31]

{ storage . name . read_public ( ) . to_bytes ( ) }

[aztec(public)]

fn

public_get_symbol ( )

->

pub

FieldCompressedString

{ storage . symbol . read_public ( ) }

[aztec(private)]

fn

private_get_symbol ( )

->

pub

FieldCompressedString

{ storage . symbol . read_private ( ) }

unconstrained fn

un_get_symbol ( )

->

pub

[u8 ;

31]

{ storage . symbol . read_public ( ) . to_bytes ( ) }

[aztec(public)]

fn

public_get_decimals ( )

->

pub

u8

{ storage . decimals . read_public ( ) }

[aztec(private)]

fn

private_get_decimals ( )

->

pub

u8

{ storage . decimals . read_private ( ) }

unconstrained fn

un_get_decimals ( )

->

pub

u8

{ storage . decimals . read_public ( ) }

[aztec(public)]

fn

set_minter (minter :

AztecAddress , approve :

bool)

{ assert (storage . admin . read ( ) . eq (context . msg_sender ( )) ,

"caller is not admin") ; storage . minters . at (minter) . write (approve) ; }

[aztec(public)]

fn

mint_public (to :

AztecAddress , amount :

Field)

{ assert (storage . minters . at (context . msg_sender ( )) . read ( ) ,

"caller is not minter") ; let amount =

U128 :: from_integer (amount) ; let new_balance = storage . public_balances . at (to) . read ( ) . add (amount) ; let supply = storage . total_supply . read ( ) . add (amount) ;

storage . public_balances . at (to) . write (new_balance) ; storage . total_supply . write (supply) ; }

[aztec(public)]

fn

mint_private (amount :

Field , secret_hash :

Field)

{ assert (storage . minters . at (context . msg_sender ( )) . read ( ) ,

"caller is not minter") ; let pending_shields = storage . pending_shields ; let

mut note =

TransparentNote :: new (amount , secret_hash) ; let supply = storage . total_supply . read ( ) . add (U128 :: from_integer (amount)) ;

storage . total_supply . write (supply) ; pending_shields . insert_from_public (& mut note) ; }

[aztec(private)]

fn

privately_mint_private_note (amount :

Field)

{ storage . balances . add (context . msg_sender ( ) ,

U128 :: from_integer (amount)) ; let selector =

FunctionSelector :: from_signature ("assert_minter_and_mint((Field),Field)") ; let _void = context . call_public_function (context . this_address ( ) , selector , [context . msg_sender ( ) . to_field ( ) , amount]) ; }

[aztec(public)]

[aztec(internal)]

fn

assert_minter_and_mint (minter :

AztecAddress , amount :

Field)

{ assert (storage . minters . at (minter) . read ( ) ,

"caller is not minter") ; let supply = storage . total_supply . read ( )

+

U128 :: from_integer (amount) ; storage . total_supply . write (supply) ; }

[aztec(public)]

fn

shield (from :

AztecAddress , amount :

Field , secret_hash :

Field , nonce :

Field)

{ if

(! from . eq (context . msg_sender ( )))

{ // The redeem is only spendable once, so we need to ensure that you cannot insert multiple shields from the same message. assert_current_call_valid_authwit_public (& mut context , from) ; }

else

{ assert (nonce ==

0 ,

"invalid nonce") ; }

let amount =

U128 :: from_integer (amount) ; let from_balance = storage . public_balances . at (from) . read ( ) . sub (amount) ;

let pending_shields = storage . pending_shields ; let

mut note =

TransparentNote :: new (amount . to_integer ( ) , secret_hash) ;

storage . public_balances . at (from) . write (from_balance) ; pending_shields . insert_from_public (& mut note) ; }

[aztec(public)]

fn

transfer_public (from :

AztecAddress , to :

AztecAddress , amount :

Field , nonce :

Field)

{ if

(! from . eq (context . msg_sender ( )))

{ assert_current_call_valid_authwit_public (& mut context , from) ; }

else

{ assert (nonce ==

0 ,

"invalid nonce") ; }

let amount =

U128 :: from_integer (amount) ; let from_balance = storage . public_balances . at (from) . read ( ) . sub (amount) ; storage . public_balances . at (from) . write (from_balance) ;

let to_balance = storage . public_balances . at (to) . read ( ) . add (amount) ; storage . public_balances . at (to) . write (to_balance) ; }

[aztec(public)]

fn

burn_public (from :

AztecAddress , amount :

Field , nonce :

Field)

{ if

(! from . eq (context . msg_sender ( )))

{ assert_current_call_valid_authwit_public (& mut context , from) ; }

else

{ assert (nonce ==

0 ,

"invalid nonce") ; }

let amount =

U128 :: from_integer (amount) ; let from_balance = storage . public_balances . at (from) . read ( ) . sub (amount) ; storage . public_balances . at (from) . write (from_balance) ;

let new_supply = storage . total_supply . read ( ) . sub (amount) ; storage . total_supply . write (new_supply) ; }

[aztec(private)]

fn

redeem_shield (to :

AztecAddress , amount :

Field , secret :

Field)

{ let pending_shields = storage . pending_shields ; let secret_hash =

compute_secret_hash (secret) ; // Get 1 note (set_limit(1)) which has amount stored in field with index 0 (select(0, amount)) and secret_hash // stored in field with index 1 (select(1, secret_hash)). let options =

NoteGetterOptions :: new ( ) . select (0 , amount ,

Option :: none ( )) . select (1 , secret_hash ,

Option :: none ( )) . set_limit (1) ; let notes = pending_shields . get_notes (options) ; let note = notes [0] . unwrap_unchecked ( ) ; // Remove the note from the pending shields set pending_shields . remove (note) ;

// Add the token note to user's balances set storage . balances . add (to ,

U128 :: from_integer (amount)) ; }

[aztec(private)]

```
fn
unshield ( from :
AztecAddress , to :
AztecAddress , amount :
Field , nonce :
Field )
{ if
( ! from . eq ( context . msg_sender ( ) ) )
{ assert_current_call_valid_authwit ( & mut context , from ) ; }
else
{ assert ( nonce ==
0 ,
"invalid nonce" ) ; }
storage . balances . sub ( from ,
U128 :: from_integer ( amount ) ) ;
let selector =
FunctionSelector :: from_signature ( "_increase_public_balance((Field),Field)" ) ; let _void = context . call_public_function (
context . this_address ( ) , selector ,
[ to . to_field ( ) , amount ] ) ; }
```

[aztec(private)]

```
fn
transfer ( from :
AztecAddress , to :
AztecAddress , amount :
Field , nonce :
Field )
{ if
( ! from . eq ( context . msg_sender ( ) ) )
{ assert_current_call_valid_authwit ( & mut context , from ) ; }
else
{ assert ( nonce ==
0 ,
"invalid nonce" ) ; }
let amount =
U128 :: from_integer ( amount ) ; storage . balances . sub ( from , amount ) ; storage . balances . add ( to , amount ) ; }
```

[aztec(private)]

fn

burn (from :

AztecAddress , amount :

Field , nonce :

Field)

{ if

(! from . eq (context . msg_sender ( )))

{ assert_current_call_valid_authwit (& mut context , from) ; }

else

{ assert (nonce ==

0 ,

"invalid nonce") ; }

storage . balances . sub (from ,

U128 :: from_integer (amount)) ;

let selector =

FunctionSelector :: from_signature ("_reduce_total_supply(Field)") ; let _void = context . call_public_function (context . this_address ( ) , selector ,

[amount]) ; }

/// Internal ///

[aztec(public)]

[aztec(internal)]

fn

_increase_public_balance (to :

AztecAddress , amount :

Field)

{ let new_balance = storage . public_balances . at (to) . read ( ) . add (U128 :: from_integer (amount)) ; storage . public_balances . at (to) . write (new_balance) ; }

[aztec(public)]

[aztec(internal)]

fn

_reduce_total_supply (amount :

Field)

{ // Only to be called from burn. let new_supply = storage . total_supply . read ( ) . sub (U128 :: from_integer (amount)) ; storage . total_supply . write (new_supply) ; }

/// Unconstrained ///

unconstrained fn

```
admin ( )
->
pub
Field
{ storage . admin . read ( ) . to_field ( ) }
unconstrained fn
is_minter ( minter :
AztecAddress )
->
pub
bool
{ storage . minters . at ( minter ) . read ( ) }
unconstrained fn
total_supply ( )
->
pub
Field
{ storage . total_supply . read ( ) . to_integer ( ) }
unconstrained fn
balance_of_private ( owner :
AztecAddress )
->
pub
Field
{ storage . balances . balance_of ( owner ) . to_integer ( ) }
unconstrained fn
balance_of_public ( owner :
AztecAddress )
->
pub
Field
{ storage . public_balances . at ( owner ) . read ( ) . to_integer ( ) }
```
Source code: noir-projects/noir-contracts/contracts/token_contract/src/main.nr#L1-L367

Helper files

TheToken contract also requires some helper files. You can view the fileshere . Copy thetypes.nr and thetypes folder intocontracts/token/src .

Compile your contract

We'll now useaztec-nargo tocompile our project. If you haven't installed aztec-nargo and aztec-cli already, it comes with the sandbox, so you can install it via theSandbox install command .

Now run the following from your contract folder (containing Nargo.toml):

```
aztec-nargo compile
```

Deploy your contracts

Let's now write a script for deploying your contracts to the Sandbox. We'll create a Private eXecution Environment (PXE) client, and then use theContractDeployer class to deploy our contracts, and store the deployment address to a local JSON file.

Create a new filesrc/deploy.mjs :

```
// src/deploy.mjs import

{ writeFileSync }

from

'fs' ; import

{ Contract , loadContractArtifact , createPXEClient }

from

'@aztec/aztec.js' ; import

{ getInitialTestAccountsWallets }

from

'@aztec/accounts/testing' ; import TokenContractJson from

"../contracts/token/target/token_contract-Token.json"

assert

{

type :

"json"

} ;

const

{

PXE_URL

=

'http://localhost:8080'

}

= process . env ;

async

function

main ( )

{ const pxe =

createPXEClient ( PXE_URL ) ; const

[ ownerWallet ]

=
```

await

getInitialTestAccountsWallets (pxe) ; const ownerAddress = ownerWallet . getCompleteAddress ( ) ;

const TokenContractArtifact =

loadContractArtifact (TokenContractJson) ; const token =

await Contract . deploy (ownerWallet , TokenContractArtifact ,

[ownerAddress ,

'TokenName' ,

'TKN' ,

18]) . send ( ) . deployed ( ) ;

console . log (Token deployed at { token . address . toString ( ) }) ;

const addresses =

{

token : token . address . toString ( )

} ; writeFileSync ('addresses.json' ,

JSON . stringify (addresses ,

null ,

2)) ; }

main ( ) . catch ((err)

=>

{ console . error (Error in deployment script: { err }) ; process . exit (1) ; }) ; We import the contract artifacts we have generated plus the dependencies we'll need, and then we can deploy the contracts by adding the following code to thesrc/deploy.mjs file. Here, we are using theContractDeployer class with the compiled artifact to send a new deployment transaction. Thewait method will block execution until the transaction is successfully mined, and return a receipt with the deployed contract address.

Note that the token's_initialize() method expects anowner address to mint an initial set of tokens to. We are using the first account from the Sandbox for this.

info If you are using the generated typescript classes, you can drop the genericContractDeployer in favor of using thedeploy method of the generated class, which will automatically load the artifact for you and type-check the constructor arguments:

await Token . deploy (client) . send ( ) . wait ( ) ; Run the snippet above asnode src/deploy.mjs , and you should see the following output, along with a newaddresses.json file in your project root:

Token deployed to 0x2950b0f290422ff86b8ee8b91af4417e1464ddfd9dda26de8af52dac9ea4f869

Next steps

Now that we have our contracts set up, it's time to actuallystart writing our application that will be interacting with them. Edit this page