Lesson 8 - Cairo contracts

Cairo Contracts

See [Documentation]()

Cairo programs are by default stateless, if we want to write contracts to run on Starknet we need additional context provided by the StarknetOS

Example contract

```rust
#[contract]

mod StarknetContract {

struct Storage {
        balance: felt,
}

// update the balance

#[external]
fn increase_balance(amount: felt) {
        balance::write(balance::read() + amount);
}

// Returns the current balance.

#[view]
fn get_balance() -> felt {
        balance::read()
}

}
```

Compiling the contract

We can use the starknet CLI
`cargo run --bin starknet-compile -- src/mycontract.cairo src/mycontract.sierra --replace-ids`

and then to casm with

`cargo run --bin starknet-sierra-compile -- src/mycontract.sierra src/mycontract.casm`

Decorators

We need to add special annotations to our code when writing contracts
The decorators are added before the function that they refer to

`#[contract]` : Specifies that this is a contract as opposed to a program

`#[event]` : Used to define an event in the contract, it is good practice to emit events for all state changes

Function decorators

`#[external]` : This shows that the function can be called from outside the contract, the external functions in your contract give the 'API'

`#[view]` : You use this to show that the function will now change the state of the contract.

`#[constructor]` : This marks a function as a constructor , we use this to initialise the state of the contract.

`#[l1_handler]` : This is less common, but shows functions that can receive a message from the L1

Storing data

Cairo contracts have state, this is maintained by the Starknet OS.
To define state within our contract we use a struct called `Storage` and within that specify the variables we wish to store.

For example

```
    struct Storage{
        contract_index: u8,
        last_update: u64,
        balances: LegacyMap::<ContractAddress, u256>
    }
```

The equivalent of the mapping data structure in Solidity is
`LegacyMap<T,T>`

Reading and writing to storage

Once we have defined our storage variables, we can read and write to them
with `read` and `write` methods.
For example
`contract_index::read();`
`contract_index::write(index);`

A further example

```
language-rust
      #[view]
      fn get_balance(_address: ContractAddress) -> u256 {
          let bal = balances::read(_address);
          return bal;
      }
```

Importing other contracts

The `use` keyword allows to include other source code in our contract, typically showing functions within libraries that we want to use.

For example

```
   use starknet::get_caller_address;
```

Contract classes

Starknet has a distinction between a contract class and a contract instance.
You are able to deploy a contract class to the network, but this does not give you a contract that users can interact with.
From the contract class you can then create instances of the contract for users to interact with.
In short you

1. Declare a contract class
2. Deploy a contract instance based on that class.

Account Abstraction

See docs from Nethermind
Also this blog from Ethereum and this blog from Gnosis
See this article from Binance Academy

Why Account Abstraction matters

The shift from EOAs to smart contract wallets with arbitrary verification logic paves the way for a series of improvements to wallet designs, as well as reducing complexity for end users. Some of the improvements Account Abstraction brings include:

- Paying for transactions in currencies other than ETH
- The ability for third parties to cover transaction fees
- Support for more efficient signature schemes (Schnorr, BLS) as well as quantum-safe ones (Lamport, Winternitz)
- Support for multisig transactions
- Support for social recovery

Previous solutions relied on centralised relay services or a steep gas overhead, which inevitably fell on the users' EOA.

EIP-4337 is a collaborative effort between the Ethereum Foundation, OpenGSN, and Nethermind to achieve Account Abstraction in a user-friendly, decentralised way.

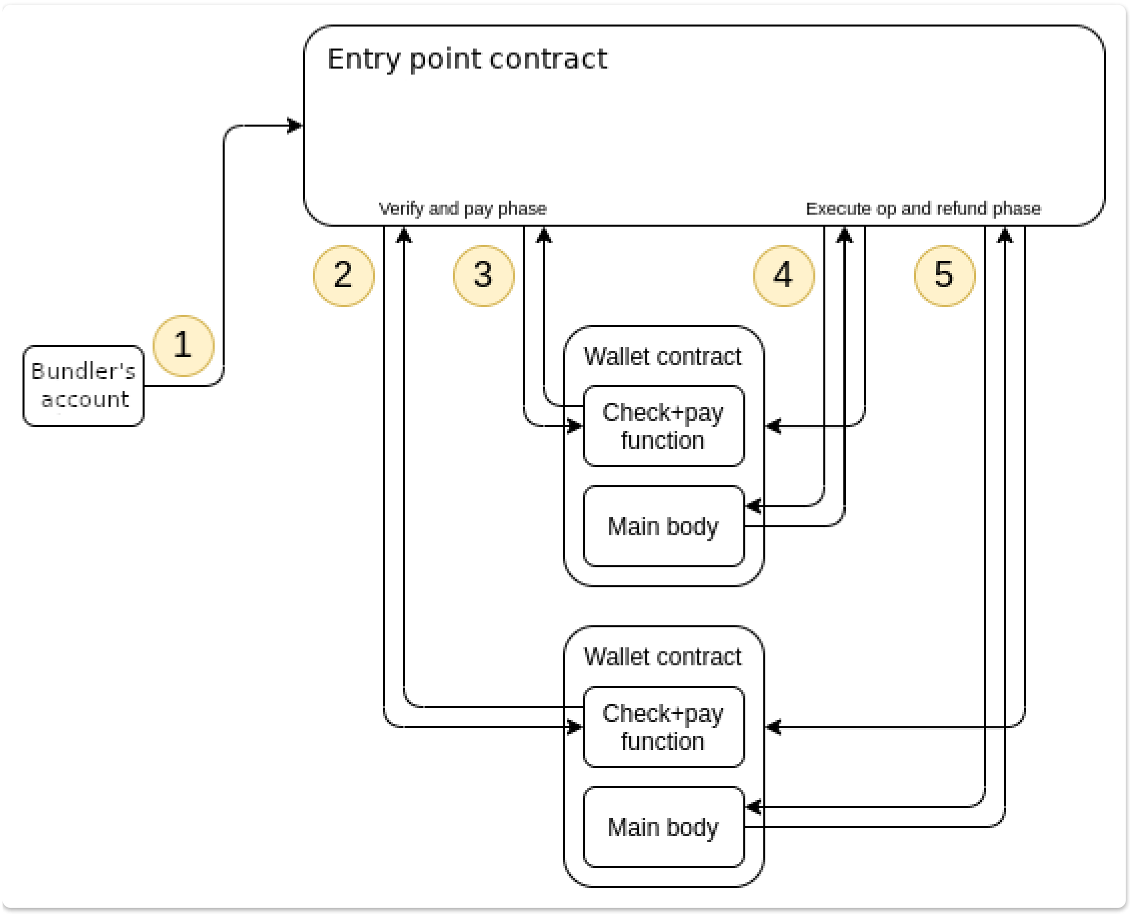
Features

- **Achieve the key goal of account abstraction**: allow users to use smart contract wallets containing arbitrary verification logic instead of EOAs as their primary account. Completely remove any need at all for users to also have EOAs (as status quo SC wallets and EIP-3074 both require)
- **Decentralization**
  - Allow any bundler (think: block builder) to participate in the process of including account-abstracted user operations
  - Work with all activity happening over a public mempool; users do not need to know the direct communication addresses (eg. IP, onion) of any specific actors
  - Avoid trust assumptions on bundlers
- **Do not require any Ethereum consensus changes**: Ethereum consensus layer development is focusing on the merge and later on scalability-oriented features, and there may not be any opportunity for further protocol changes for a long time. Hence, to increase the chance of faster adoption, this proposal avoids Ethereum consensus changes.
- **Try to support other use cases**
  - Privacy-preserving applications
  - Atomic multi-operations (similar goal to EIP-3074)
  - Pay tx fees with ERC-20 tokens, allow developers to pay fees for their users, and EIP-3074-like **sponsored transaction** use cases more generally

- Support aggregated signature (e.g. BLS)

Instead of transactions, users send 'UserOperation' objects into a separate mempool, then actors called 'bundlers' package these up as transactions to a special contract.
The entry point’s `handleOps` function must perform the following steps
It must make two loops, the verification loop and the execution loop. In the verification loop, the `handleOps` call must perform the following steps for each `UserOperation`:

- Create the account if it does not yet exist
- Call **`validateUserOp`** on the account, passing in the `UserOperation`, the required fee and aggregator (if there is one).



First-time account creation

It is an important design goal of this proposal to replicate the key property of EOAs that users do not need to perform some custom action or rely on an existing user to create their wallet; they can simply generate an address locally and immediately start accepting funds.

The wallet creation itself is done by a “factory” contract, with wallet-specific data. The factory is expected to use CREATE2 (not CREATE) to create the wallet, so that the order of creation of wallets doesn’t interfere with the generated addresses.

There is a certain synergy here with some of the MEV solutions.

Starknet JS

Introduction

Documentation
https://www.starknetjs.com/docs/API/provider

This is modelled on libraries such as Web3.js

The main areas are

- Provider API - connecting to starknet
- Account API - connection with an account
- Signer API - allows signatures
- Contract API - an object representing a contract
- Utils API - Utility methods

Installation

```
npm install starknet@next
```

Provider API

You can create a provider with

```
const provider = new starknet.Provider()
```

or if you want specify the network

```
const provider = new starknet.Provider({
sequencer: {
       network: 'mainnet-alpha' // or 'goerli-alpha'
       }
})
```

To interact with a contract we use the provider we set up

Provider methods

callContract

```
provider.callContract(call [ , blockIdentifier ]) => _Promise
```

The call object has the following structure

- call.contractAddress - Address of the contract
- call.entrypoint - Entrypoint of the call (method name)
- call.calldata - Payload for the invoking method

Response

```
  {
result: string[];
}
```

getTransactionReceipt

```
  provider.getTransactionReceipt(txHash) => _Promise
```

Response

```
  {
transaction_hash: string;
status: 'NOT_RECEIVED' | 'RECEIVED' | 'PENDING' | 'ACCEPTED_ON_L2' |
'ACCEPTED_ON_L1' | 'REJECTED';
actual_fee?: string;
status_data?: string;
messages_sent?: Array<MessageToL1>;
events?: Array<Event>;
l1_origin_message?: MessageToL2;
}
```

Deploy Contract

```
  provider.deployContract(payload [ , abi ]) => _Promise
```

Response

```
  {
transaction_hash: string;
contract_address?: string;
};
```

Wait For Transaction

`provider.waitForTransaction(txHash [ , retryInterval]) => Promise < void >`

Wait for the transaction to be accepted on L2 or L1.

Other methods

- getBlock
- getClassAt
- getStorageAt
- getTransaction
- declareContract
- waitForTransaction

A useful library is get-starknet which provides connection methods.
If you are connecting with a wallet use the connect method from the get-starknet module

```
  const starknet = await connect()
// connect to the wallet
await starknet?.enable({ starknetVersion: "v4" })
const provider = starknet.account
```

Signer API

The Signer API allows you to sign transactions and messages

You can generate a key pair by using the utility functions

`ec.genKeyPair()`
or
`getKeyPair(private_key)`

The signer object is then created with

`new starknet.Signer(keyPair)`

You can then sign messages

```
  signer.signMessage(data, accountAddress) => _Promise
```

Code Example

```
  const privateKey = stark.randomAddress();
const starkKeyPair = ec.genKeyPair(privateKey);
const starkKeyPub = ec.getStarkKey(starkKeyPair);
```

Account API

The Account object extends the Provider object
To create the account object, an account contract needs to have been deployed, see below for guide to deploy an account contract.

```
  const account  = new starknet.Account(Provider, address, starkKeyPair)
```

Account Properties

```
  account.address =>string
```

Account Methods

```
  account.getNonce() => Promise
account.estimateFee(calls [ , options ]) => _Promise
account.execute(calls [ , abi , transactionsDetail ]) => _Promise
account.signMessage(typedData) => _Promise
account.hashMessage(typedData) => _Promise
account.verifyMessageHash(hash, signature) => _Promise
account.verifyMessage(typedData, signature) => _Promise
```

See guideto creating and deploying an account

Contract

Creating the contract object

```
  new starknet.Contract(abi, address, providerOrAccount)

contract.attach(address)` _for changing the address of the connected contract_

contract.connect(providerOrAccount)` _for changing the provider or account_
```

Contract Properties

```
  contract.address => string
contract.providerOrAccount => ProviderInterface | AccountInterface
contract.deployTransactionHash => string | null
contract.abi => Abi
```

Contract Interaction

1. View Functions

contract.METHOD_NAME(...args [, overrides]) => Promise < Result >

The type of the result depends on the ABI.
The result object will be returned with each parameter available positionally and if the parameter is named, it will also be available by its name.

The override can identify the block : `overrides.blockIdentifier`

Code Example

`const bal = await contract.get_balance()`

2. Write Functions

contract.METHOD_NAME(...args [, overrides]) => Promise < AddTransactionResponse >

Overrides can be

- overrides.signature - Signature that will be used for the transaction
- overrides.maxFee - Max Fee for the transaction
- overrides.nonce - Nonce for the transaction

Code Example

`await contract.increase_balance(13)`

Utils

Useful Methods

- toBN

  `toBN(number: BigNumberish, base?: number | 'hex'): BN`

  Converts BigNumberish to BN.
  Returns a BN.
- uint256ToBN

  `uint256ToBN(uint256: Uint256): BN`

  Function to convert `Uint256` to `BN` (big number), which uses the `bn.js` library.
- getStarkKey

  `getStarkKey(keyPair: KeyPair): string`

  Public key defined over a Stark-friendly elliptic curve that is different from the standard Ethereum elliptic curve
- getKeyPairFromPublicKey

  `getKeyPairFromPublicKey(publicKey: BigNumberish): KeyPair`

  Takes a public key and casts it into `elliptic` KeyPair format.
  Returns keyPair with public key only, which can be used to verify signatures, but can't sign anything.
- sign

  `sign(keyPair: KeyPair, msgHash: string): Signature`

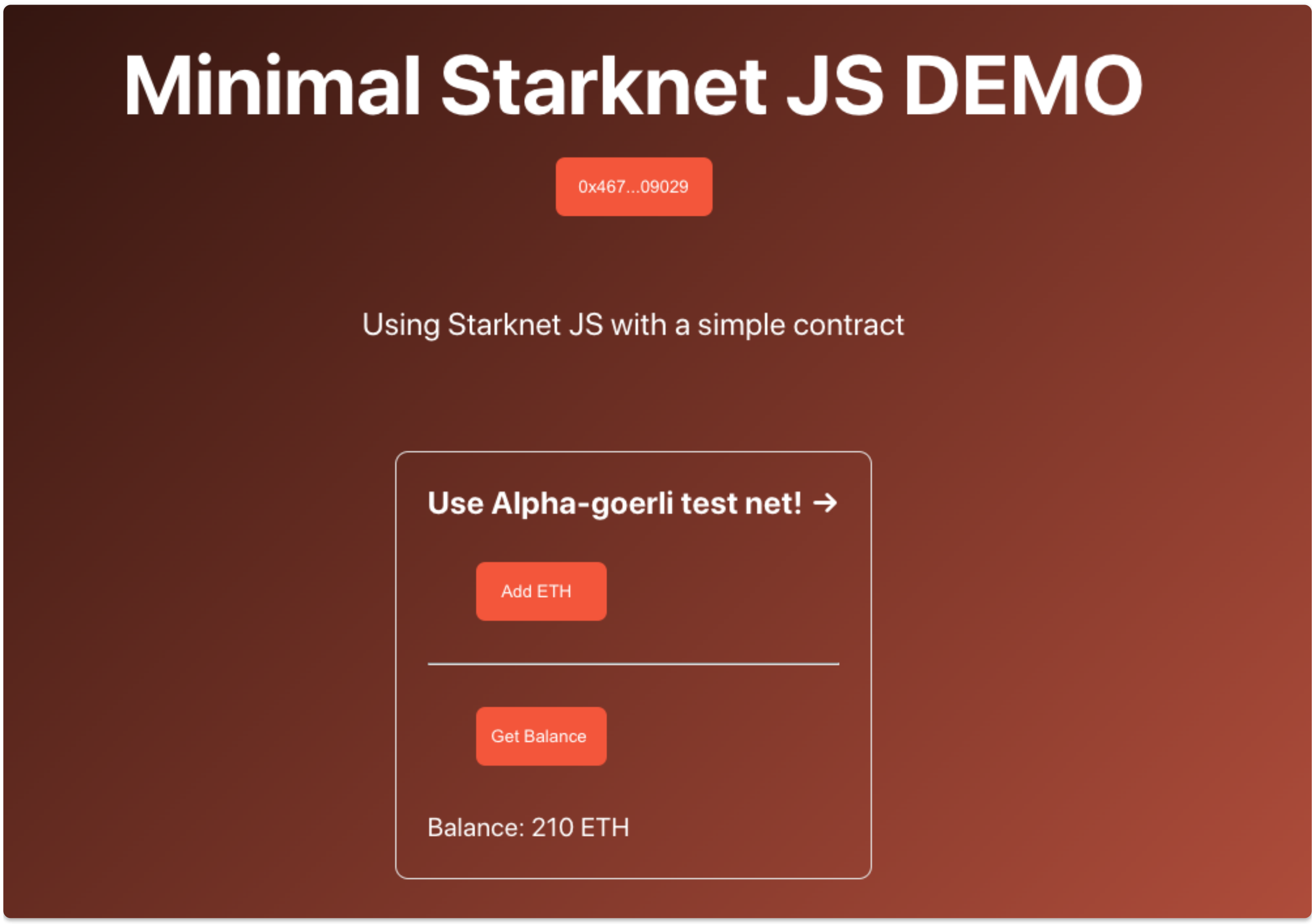
  Signs a message using the provided key.
  keyPair should be an KeyPair with a valid private key.
  Returns an Signature.
- verify

  `verify(keyPair: KeyPair | KeyPair[], msgHash: string, sig: Signature): boolean`

  Verifies a message using the provided key.
  keyPair should be an KeyPair with a valid public key.
  sig should be an Signature.
  Returns true if the verification succeeds.

Example in repo

Code
Based on tutorial from @darlingtonnnam



Links

Starknet.js workshop : https://github.com/0xs34n/starknet.js-workshop
Tutorial on medium :https://medium.com/@darlingtonnnam/an-in-depth-guide-to-getting-started-with-starknet-js-a55c04d0ccb7

Alternatives to starknet.js

Rust library
Python library

Warp

Warp allows you transpile Solidity contracts into Cairo

Installation Instructions

See Warp installation instructions

Note WARP is still using Cairo v 0.11
Cairo v 1.0 development is on this branch

Using Warp

`warp transpile example_contracts/ERC20.sol`

`warp transpile example_contracts/ERC20.sol --compile-cairo`

You can then deploy your cairo code to the network, with the following commands you need to specify the network, in our case alpha-goerli

```
  warp deploy ERC20.json --network alpha-goerli
```

```
  Deploy transaction was sent.
Contract address:
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a
Transaction hash: 0x32ca42d1341703cc957845ea53a71b3eb2e762ff148cb9dc522322eede94b65
```

You can invoke a transaction on your contract

```

warp invoke --program test.json --address
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a  --network
```

```
alpha-goerli --function store --inputs [13]
```

```
  Invoke transaction was sent.
Contract address:
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a
Transaction hash: 0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881
```

And check the status

```
  warp status 0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881 --
network alpha-goerli
```

which will give a answer similar to

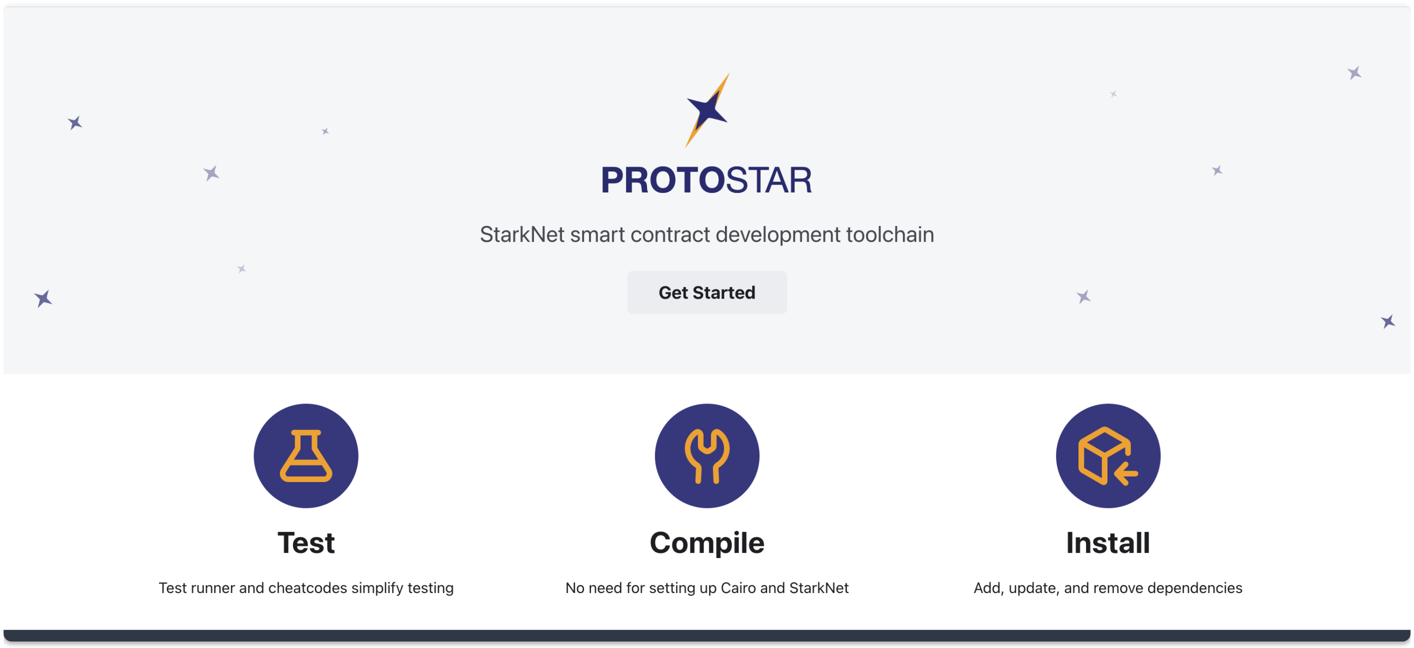
```
        {
                "block_hash":
"0x1c55254f16d087f0bf7776183c4d38549680e68600394167f304f1afe5a035e",
                "tx_status": "ACCEPTED_ON_L1"
        }
```

You should be able to see the details on the block explorer
[Voyager Block Explorer]

There is also now a [vyper transpiler]

Cairo Development Tools

Protostar


PROTOSTAR
StarkNet smart contract development toolchain
Get Started
Test
Test runner and cheatcodes simplify testing
Compile
No need for setting up Cairo and StarkNet
Install
Add, update, and remove dependencies

See Protostar

Features

CLI toolchain
Unit test programs and contracts
Deploy contracts

Plugins

Plugins are available for popular IDEs offering some degree of language support

VSCode plugin for Cairo :
Hardhat plugin

Foundry experimental

Developing contracts with Protostar

Creating a project

Run
`protostar init-cairo1 my-project`

this gives a default template. Note the layout for multiple contracts is for the moment more complex.

Configuration

This is specified in `protostar.toml`
The sections are self explanatory, for example

```
  [contracts]
my_contract = ["my_contract"]
other_contract = ["other_contract"]
```

Any contract with an entry point, that is a function marked `#[external]` or `#[view]` needs to be specified in the `protostar.toml` file

Compiling contracts

Once you have written your contracts and specified them in the `protostar.toml` file as above
Run
`protostar build-cairo1`
If the compilation is successful it should return a class hash.
It will add compilation artefacts to the out directory.

Declaring a contract class

Use the command, see docs
`declare-cairo1`
For example
`protostar declare my_contract --network testnet`

Deploying a contract instance

Use the command
`deploy-cairo`
For example
`protostar deploy-cairo 0xdeadbeef --network testnet`

Writing tests

The format follows the normal rust format for tests, that is you use the decorator `test`
All test files should be placed in the test directory

```rust
#[test]
fn test_bool_operators() {
        assert(1 == 1);
}
```

To write failing tests, you use the `panic` keyword, for example

```rust
#[test]
fn test_with_panic() {
        let mut val = 42;
        panic(val)
}
```

To run the tests use the command

`test-cairo1`

Cheatcodes

These are available to support your testing.
See Cheatcodes for the full list.