Deploying and Interacting with a Voting contract

The **`Vote`** contract in Starknet begins by registering voters through the contract's constructor. Three voters are initialized at this stage, and their addresses are passed to an internal function **`_register_voters`**. This function adds the voters to the contract's state, marking them as registered and eligible to vote.

Within the contract, the constants **`YES`** and **`NO`** are defined to represent the voting options (1 and 0, respectively). These constants facilitate the voting process by standardizing the input values.

Once registered, a voter is able to cast a vote using the **`vote`** function, selecting either the 1 (YES) or 0 (NO) as their vote. When voting, the state of the contract is updated, recording the vote and marking the voter as having voted. This ensures that the voter is not able to cast a vote again within the same proposal. The casting of a vote triggers the **`VoteCast`** event, logging the action.

The contract also monitors unauthorized voting attempts. If an unauthorized action is detected, such as a non-registered user attempting to vote or a user trying to vote again, the **`UnauthorizedAttempt`** event is emitted.

Together, these functions, states, constants, and events create a structured voting system, managing the lifecycle of a vote from registration to casting, event logging, and result retrieval within the Starknet environment. Constants like **`YES`** and **`NO`** help streamline the voting process, while events play a vital role in ensuring transparency and traceability.

Listing {{#ref voting_contract}} shows the `Vote` contract in detail:

```cairo,noplayground
{{#include ../listings/ch16-building-advanced-starknet-smart-contracts/
listing_05_vote_contract/src/lib.cairo}}
```

{{#label voting_contract}}

<span class="caption">Listing {{#ref voting_contract}}: A voting smart contract</span>

Deploying, Calling and Invoking the Voting Contract

Part of the Starknet experience is deploying and interacting with smart contracts.

Once the contract is deployed, we can interact with it by calling and invoking its functions:

- Calling contracts: Interacting with external functions that only read from the state. These functions do not alter the state of the network, so they don't require fees or signing.
- Invoking contracts: Interacting with external functions that can write to the state. These functions do alter the state of the network and require fees and signing.

We will setup a local development node using `katana` to deploy the voting contract. Then, we'll interact with the contract by calling and invoking its functions. You can also use the Goerli Testnet instead of `katana`. However, we recommend using `katana` for local development and testing. You can find the complete tutorial for `katana` in the ["Katana: A Local Node"][katana chapter] chapter of the Starknet Book.

[katana chapter]: https://book.starknet.io/ch02-04-katana.html

The `katana` Local Starknet Node

`katana` is designed to support local development by the [Dojo team][dojo katana]. It will allow you to do everything you need to do with Starknet, but locally. It is a great tool

for development and testing.
To install `katana` from the source code, please refer to the ["Basic Installation"][katana installation] chapter of the Starknet Book.
> Note: Please verify that the version of `katana` match the specified version provided below.
>
> ```bash
> $ katana --version
> katana 0.6.0
> ```
>
> To upgrade `katana` version, refer to the ["Basic Installation"][katana installation] chapter of the Starknet Book.

Once you have `katana` installed, you can start the local Starknet node with:
```bash
katana --accounts 3 --seed 0 --gas-price 250
```
This command will start a local Starknet node with 3 deployed accounts. We will use these accounts to deploy and interact with the voting contract:
```bash
...
PREFUNDED ACCOUNTS
==================
| Account address |
0x03ee9e18edc71a6df30ac3aca2e0b02a198fbce19b7480a63a0d71cbd76652e0
| Private key     |
0x0300001800000000300000180000000000030000000000003006001800006600
| Public key      |
0x01b7b37a580d91bc3ad4f9933ed61f3a395e0e51c9dd5553323b8ca3942bb44e
| Account address |
0x033c627a3e5213790e246a917770ce23d7e562baa5b4d2917c23b1be6d91961c
| Private key     |
0x0333803103001800039980190300d206608b0070db0012135bd1fb5f6282170b
| Public key      |
0x04486e2308ef3513531042acb8ead377b887af16bd4cdd8149812dfef1ba924d
| Account address |
0x01d98d835e43b032254ffbef0f150c5606fa9c5c9310b1fae370ab956a7919f5
| Private key     |
0x07ca856005bee0329def368d34a6711b2d95b09ef9740ebf2c7c7e3b16c1ca9c
| Public key      |
0x07006c42b1cfc8bd45710646a0bb3534b182e83c313c7bc88ecf33b53ba4bcbc
...
```
Before we can interact with the voting contract, we need to prepare the voter and admin accounts on Starknet. Each voter account must be registered and sufficiently funded for voting. For a more detailed understanding of how accounts operate with Account

Abstraction, refer to the ["Account Abstraction"][aa chapter] chapter of the Starknet Book.
[dojo katana]: https://github.com/dojoengine/dojo/blob/main/crates/katana
[katana installation]: https://book.starknet.io/ch02-01-basic-installation.html#katana-node-installation
[aa chapter]: https://book.starknet.io/ch04-00-account-abstraction.html

Smart Wallets for Voting

Aside from Scarb you will need to have Starkli installed. Starkli is a command line tool that allows you to interact with Starknet. You can find the installation instructions in the ["Basic Installation"][starkli installation] chapter of the Starknet Book.

> Note: Please verify that the version of `starkli` match the specified version provided below.
>
> ```bash
> $ starkli --version
> 0.2.9 (0535f44)
> ```
>
> To upgrade `starkli` to `0.2.9`, use the `starkliup -v 0.2.9` command, or simply `starkliup` which installed the latest stable version.

For each smart wallet we'll use, we must create a Signer within the encrypted keystore and an Account Descriptor. This process is also detailed in the ["Testnet Deployment"][signer creation] chapter of the Starknet Book.
We can create Signers and Account Descriptors for the accounts we want to use for voting. Let's create a smart wallet for voting in our smart contract.
Firstly, we create a signer from a private key:

```bash
starkli signer keystore from-key ~/.starkli-wallets/deployer/account0_keystore.json
```

Then, we create the Account Descriptor by fetching the katana account we want to use:

```bash
starkli account fetch <KATANA ACCOUNT ADDRESS> --rpc http://0.0.0.0:5050 --output ~/.starkli-wallets/deployer/account0_account.json
```

This command will create a new `account0_account.json` file containing the following details:

```bash
{
  "version": 1,
  "variant": {
        "type": "open_zeppelin",
        "version": 1,
        "public_key": "<SMART_WALLET_PUBLIC_KEY>"
  },
    "deployment": {
        "status": "deployed",
```

```
      "class_hash": "<SMART_WALLET_CLASS_HASH>",
      "address": "<SMART_WALLET_ADDRESS>"
  }
}
```
You can retrieve the smart wallet class hash (it will be the same for all your smart wallets) with the following command. Notice the use of the `--rpc` flag and the RPC endpoint provided by `katana`:
```
starkli class-hash-at <SMART_WALLET_ADDRESS> --rpc http://0.0.0.0:5050
```
For the public key, you can use the `starkli signer keystore inspect` command with the directory of the keystore json file:
```bash
starkli signer keystore inspect ~/.starkli-wallets/deployer/account0_keystore.json
```
This process is identical for `account_1` and `account_2` in case you want to have a second and a third voter.
[starkli installation]: https://book.starknet.io/ch02-01-basic-installation.html#starkli-installation)
[signer creation]: https://book.starknet.io/ch02-05-testnet-deployment.html?highlight=signer#creating-a-signer
Contract Deployment
Before deploying, we need to declare the contract. We can do this with the `starkli declare` command:
```bash
starkli declare target/dev/starknetbook_chapter_2_Vote.sierra.json --rpc http://0.0.0.0:5050 --account ~/.starkli-wallets/deployer/account0_account.json --keystore ~/.starkli-wallets/deployer/account0_keystore.json
```
If the compiler version you're using is older than the one used by Starkli and you encounter a `compiler-version` error while using the command above, you can specify a compiler version to use in the command by adding the `--compiler-version x.y.z` flag.
If you're still encountering issues with the compiler version, try upgrading Starkli using the command: `starkliup` to make sure you're using the latest version of starkli.
The class hash of the contract is: `0x06974677a079b7edfadcd70aa4d12aac0263a4cda379009fca125e0ab1a9ba52`. You can declare this contract on Sepolia testnet and see that the class hash will correspond.
The `--rpc` flag specifies the RPC endpoint to use (the one provided by `katana`). The `--account` flag specifies the account to use for signing the transaction. The account we use here is the one we created in the previous step. The `--keystore` flag specifies the keystore file to use for signing the transaction.
Since we are using a local node, the transaction will achieve finality immediately. If you are using the Goerli Testnet, you will need to wait for the transaction to be final, which usually takes a few seconds.
The following command deploys the voting contract and registers voter_0, voter_1, and

voter_2 as eligible voters. These are the constructor arguments, so add a voter account that you can later vote with.
```bash
starkli deploy <class_hash_of_the_contract_to_be_deployed> <voter_0_address>
<voter_1_address> <voter_2_address> --rpc http://0.0.0.0:5050 --account ~/.starkli-
wallets/deployer/account0_account.json --keystore ~/.starkli-wallets/deployer/
account0_keystore.json
```
An example command:
```bash
starkli deploy
0x06974677a079b7edfadcd70aa4d12aac0263a4cda379009fca125e0ab1a9ba52
0x03ee9e18edc71a6df30ac3aca2e0b02a198fbce19b7480a63a0d71cbd76652e0
0x033c627a3e5213790e246a917770ce23d7e562baa5b4d2917c23b1be6d91961c
0x01d98d835e43b032254ffbef0f150c5606fa9c5c9310b1fae370ab956a7919f5 --rpc
http://0.0.0.0:5050 --account ~/.starkli-wallets/deployer/account0_account.json --
keystore ~/.starkli-wallets/deployer/account0_keystore.json
```
In this case, the contract has been deployed at an specific address: `0x05ea3a690be71c7fcd83945517f82e8861a97d42fca8ec9a2c46831d11f33349`. This address will be different for you. We will use this address to interact with the contract.

Voter Eligibility Verification

In our voting contract, we have two functions to validate voter eligibility, `voter_can_vote` and `is_voter_registered`. These are external read functions, which mean they don't alter the state of the contract but only read the current state.
The `is_voter_registered` function checks whether a particular address is registered as an eligible voter in the contract. The `voter_can_vote` function, on the other hand, checks whether the voter at a specific address is currently eligible to vote, i.e., they are registered and haven't voted already.
You can call these functions using the `starkli call` command. Note that the `call` command is used for read functions, while the `invoke` command is used for functions that can also write to storage. The `call` command does not require signing, while the `invoke` command does.
```bash+
starkli call
0x05ea3a690be71c7fcd83945517f82e8861a97d42fca8ec9a2c46831d11f33349
voter_can_vote
0x03ee9e18edc71a6df30ac3aca2e0b02a198fbce19b7480a63a0d71cbd76652e0 --rpc
http://0.0.0.0:5050
```
First we added the address of the contract, then the function we want to call, and finally the input for the function. In this case, we are checking whether the voter at the address `0x03ee9e18edc71a6df30ac3aca2e0b02a198fbce19b7480a63a0d71cbd76652e0` can vote.
Since we provided a registered voter address as an input, the result is 1 (boolean true), indicating the voter is eligible to vote.

Next, let's call the `is_voter_registered` function using an unregistered account address to observe the output:
```bash
starkli call
0x05ea3a690be71c7fcd83945517f82e8861a97d42fca8ec9a2c46831d11f33349
is_voter_registered 0x44444444444444444 --rpc http://0.0.0.0:5050
```
With an unregistered account address, the terminal output is 0 (i.e., false), confirming that the account is not eligible to vote.
Casting a Vote
Now that we have established how to verify voter eligibility, we can vote! To vote, we interact with the `vote` function, which is flagged as external, necessitating the use of the `starknet invoke` command.
The `invoke` command syntax resembles the `call` command, but for voting, we submit either `1` (for Yes) or `0` (for No) as our input. When we invoke the `vote` function, we are charged a fee, and the transaction must be signed by the voter; we are writing to the contract's storage.
```bash
//Voting Yes
starkli invoke
0x05ea3a690be71c7fcd83945517f82e8861a97d42fca8ec9a2c46831d11f33349 vote 1
--rpc http://0.0.0.0:5050 --account ~/.starkli-wallets/deployer/account0_account.json --
keystore ~/.starkli-wallets/deployer/account0_keystore.json
//Voting No
starkli invoke
0x05ea3a690be71c7fcd83945517f82e8861a97d42fca8ec9a2c46831d11f33349 vote 0
--rpc http://0.0.0.0:5050 --account ~/.starkli-wallets/deployer/account0_account.json --
keystore ~/.starkli-wallets/deployer/account0_keystore.json
```
You will be prompted to enter the password for the signer. Once you enter the password, the transaction will be signed and submitted to the Starknet network. You will receive the transaction hash as output. With the starkli transaction command, you can get more details about the transaction:
```bash
starkli transaction <TRANSACTION_HASH> --rpc http://0.0.0.0:5050
```
This returns:
```bash
{
  "transaction_hash":
"0x5604a97922b6811060e70ed0b40959ea9e20c726220b526ec690de8923907fd",
  "max_fee": "0x430e81",
  "version": "0x1",
  "signature": [
    "0x75e5e4880d7a8301b35ff4a1ed1e3d72fffefa64bb6c306c314496e6e402d57",
    "0xbb6c459b395a535dcd00d8ab13d7ed71273da4a8e9c1f4afe9b9f4254a6f51"
  ],
  "nonce": "0x3",
  "type": "INVOKE",
  "sender_address":
"0x3ee9e18edc71a6df30ac3aca2e0b02a198fbce19b7480a63a0d71cbd76652e0",
  "calldata": [
    "0x1",
    "0x5ea3a690be71c7fcd83945517f82e8861a97d42fca8ec9a2c46831d11f33349",
    "0x132bdf85fc8aa10ac3c22f02317f8f53d4b4f52235ed1eabb3a4cbbe08b5c41",
    "0x0",
    "0x1",
    "0x1",
    "0x1"
  ]
}
```
If you try to vote twice with the same signer you will get an error:
```bash
Error: code=ContractError, message="Contract error"
```
The error is not very informative, but you can get more details when looking at the output in the terminal where you started `katana` (our local Starknet node):
```bash
...
Transaction execution error: "Error in the called contract
(0x03ee9e18edc71a6df30ac3aca2e0b02a198fbce19b7480a63a0d71cbd76652e0):
    Error at pc=0:81:
    Got an exception while executing a hint: Custom Hint Error: Execution failed. Failure
reason: \"USER_ALREADY_VOTED\".
    ...
```
The key for the error is `USER_ALREADY_VOTED`.
```bash
assert!(can_vote, "USER_ALREADY_VOTED");
```
We can repeat the process to create Signers and Account Descriptors for the accounts we want to use for voting. Remember that each Signer must be created from a private key, and each Account Descriptor must be created from a public key, a smart wallet address, and the smart wallet class hash (which is the same for each voter).
```bash
starkli invoke
0x05ea3a690be71c7fcd83945517f82e8861a97d42fca8ec9a2c46831d11f33349 vote 0
--rpc http://0.0.0.0:5050 --account ~/.starkli-wallets/deployer/account1_account.json --
keystore ~/.starkli-wallets/deployer/account1_keystore.json
starkli invoke
0x05ea3a690be71c7fcd83945517f82e8861a97d42fca8ec9a2c46831d11f33349 vote 1
```

--rpc http://0.0.0.0:5050 --account ~/.starkli-wallets/deployer/account2_account.json --keystore ~/.starkli-wallets/deployer/account2_keystore.json
~~~
### Visualizing Vote Outcomes
To examine the voting results, we invoke the `get_vote_status` function, another view function, through the `starknet call` command.
```bash
starkli call
0x05ea3a690be71c7fcd83945517f82e8861a97d42fca8ec9a2c46831d11f33349
get_vote_status --rpc http://0.0.0.0:5050
~~~
The output reveals the tally of "Yes" and "No" votes along with their relative percentages.