ARASHTAD

Title	HOW TO SET UP THE DEPENDENCIES FOR RUNNING LOCAL TERRA: ALL YOU NEED TO KNOW
Description	Tutorial
Date	June 03, 2022
Author	Arashtad
Author URI	https://Arashtad.com

In this tutorial-based article, we are going to set up the dependencies for running Local Terra. So, you will be guided through all the installations and commands necessary to start your interaction with local Terra smart contracts. Installing Go, Docker, Terrad, and git cloning local Terra, as well as terra-core repositories, are some of the main steps we are going to take in this tutorial.

INSTALLING THE DEPENDENCIES

For running local terra the main step is to install all of the dependencies required. The first step is to install go. You need to be careful that the version should be higher than 1.17.5.

THE DEPENDENCIES FOR RUNNING LOCAL TERRA #1: GO

To install Go, you should find the latest version for your operating system using this link. Then, the next step is to check the version to make sure it meets the requirement of our version (higher than 1.17.5). To do so, enter the following command in the terminal:

```
go version
```

Result:

```
go version go1.18.1 linux/amd64
```

If you see a result like the above, your installation has been successful. You should also have the Rust programming language installed using the below command (Notice that this command is used for Linux OS) in the terminal:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

For other operating systems, follow the instructions on this link. Now, it is time to git clone the terra-core repository. In the terminal, enter the following command:

```
git clone https://github.com/terra-money/core terra-core
```

After that, change the directory to the terra-core folder by entering the below command in the terminal:

```
cd terra-core
```

To install the downloaded repository, enter the following in the terminal:

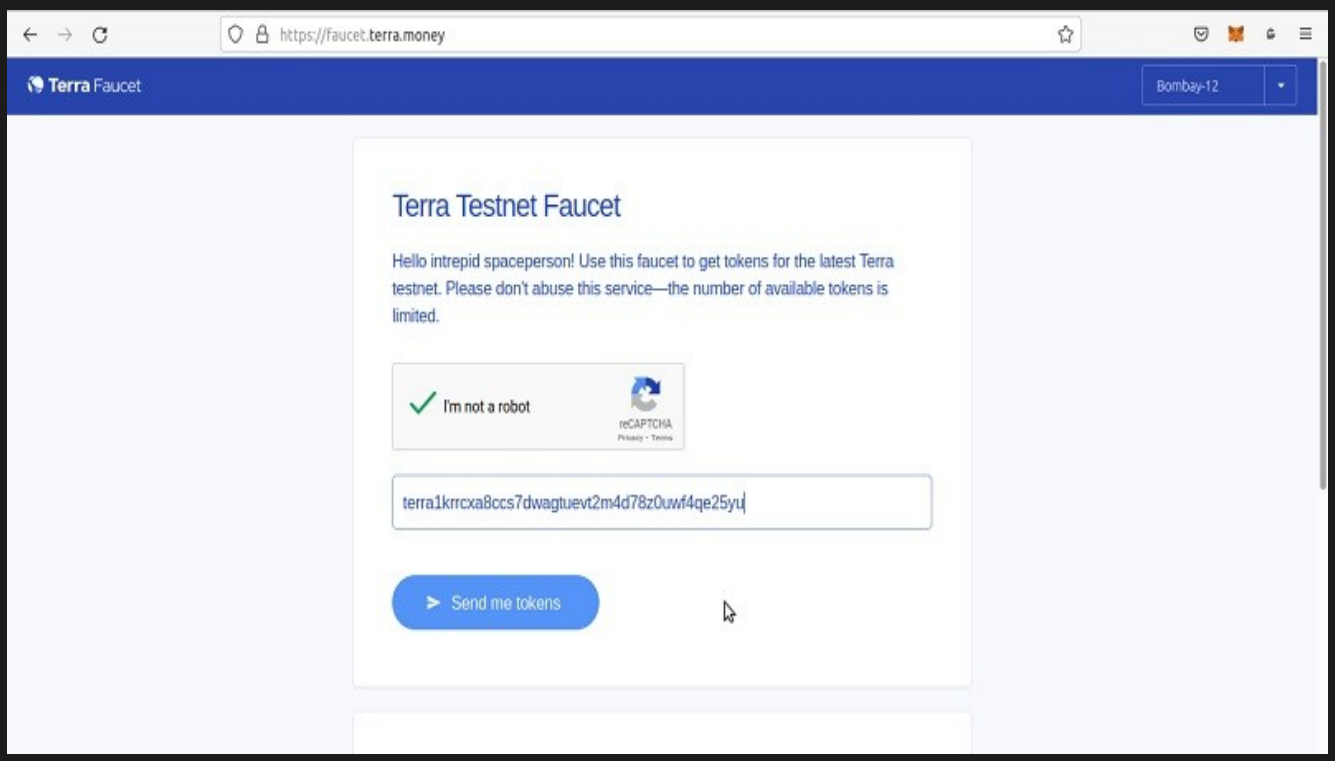
```
git checkout main make install
```

And, the most important step is to git clone the local Terra repository. We follow the same steps for this one as well:

```
git clone --depth 1
https://www.github.com/terra-money/LocalTerra cd LocalTerra
```

THE DEPENDENCIES FOR RUNNING LOCAL TERRA #2: DOCKER

One of the required dependencies for running local Terra is Docker. You should also have docker installed on your operating system. Here we explain the process for Linux:
For the installation guide in other operating systems than Linux, follow the instructions on the Docker.



INSTALLATION GUIDE ON LINUX

This process consists of 5 steps that are described in the followings.

1. Dependencies:

At first, update the system to get the list of available packages and their version numbers. To do so, enter the following command in the terminal:

```
sudo apt update
```

To prepare for the installation run these commands in the terminal one after the other:

```
sudo apt -y install apt-transport-https ca-certificates curl
software-properties-common sudo apt -y remove docker docker-engine
docker.io containerd runc
```

2. Add Docker's Official GPG Key:

Also run the following commands in the terminal:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
--dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

3. Add the Docker Repository to Linux:

To add the Docker repository for Linux, enter the following commands in the terminal:

```
echo "deb [arch=$(dpkg --print-architecture)
signed-by=/usr/share/keyrings/dockerarchive-keyring.gpg]
https://download.docker.com/linux/ubuntu bionic stable" | sudo
tee /etc/apt/sources.list.d/docker.list > /dev/null deb [arch=amd64
signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu bionic stable
```

4. Install the Docker Engine and the Docker Compose:

And finally, It is time to install docker by running these 2 commands in the terminal one after the other.

```
sudo apt update sudo apt install docker-ce docker-ce-cli
containerd.io
```

5. Checking the Installation:

To start with checking the installation, use the command below:

```
sudo usermod -aG docker USER newgrp docker
```

To make sure the installation has been successful, run the following command in the terminal:

```
docker version
```

Result:

```
Client: Docker Engine - Community Version: 20.10.14 API version:
1.41 Go version: go1.16.15 Git commit: a224086 Built: Thu Mar 24
01:47:57 2022 OS/Arch: linux/amd64 Context: default Experimental:
true Server: Engine: Version: 20.10.14 API version: 1.41 (minimum
version 1.12) Go version: go1.16.15 Git commit: 87a90dc Built: Thu
Mar 24 17:15:03 2022 OS/Arch: linux/amd64 Experimental: false
containerd: Version: v1.5.11 GitCommit:
3df54a852345ae127d1fa3092b95168e4a88e2f8 runc: Version: 1.0.3
GitCommit: docker-init: Version: 0.19.0 GitCommit: de40ad
```

After installation for every use, you need to first sign in and run the following commands in the terminal:

```
sudo usermod -aG docker docker login docker-compose up
```

Run the above commands in the local Terra directory terminal.

Result:

```
terrad_1 | 1:47PM INF indexed block height=143 module=txindex
terrad_1 | 1:47PM INF Timed out dur=4987.108435 height=144
module=consensus round=0 step=1 terrad_1 | 1:47PM INF received
proposal module=consensus proposal={"Type":32,"block_id":
{"hash":"F4DD36C8FB8D539F3F28ABB8BB7969A099AEE78B707FAE7F3C7DC8D36BF
38B9D","parts":
{"hash":"F92FDEA4153F172B082676706A3B35F54A0EF3D6639FA23A3FD88FFBB40
ACA6E","total":1}},"height":144,"pol_round":-
1,"round":0,"signature":"nDA4YtVFM/a8TXCLMwTwydkfJSwOoI57zNkkKieyv0x
y/ zbnCVIM10cpzh78+0wym1jT3fj16ZqH5OiFE3kaAg==","timestamp":"2022-
05- 07T13:47:21.846543194Z"} terrad_1 | 1:47PM INF received complete
proposal block
hash=F4DD36C8FB8D539F3F28ABB8BB7969A099AEE78B707FAE7F3C7DC8D36BF38B9
D height=144 module=consensus terrad_1 | 1:47PM INF finalizing
commit of block
hash=F4DD36C8FB8D539F3F28ABB8BB7969A099AEE78B707FAE7F3C7DC8D36BF38B9
D height=144 module=consensus num_txs=0
root=EFAA4A74DD1E73032E0FF553A37CCDFAD443AF0EE395516EB247948ADAF45A2
1 terrad_1 | 1:47PM INF minted coins from module account
amount=226570495uluna from=mint module=x/bank terrad_1 | 1:47PM INF
executed block height=144 module=state num_invalid_txs=0
num_valid_txs=0 terrad_1 | 1:47PM INF commit synced
commit=436F6D6D697449447B5B31313820383520353820323436203632203239
2031332031363020333720313939203439203935203730203230302032313120323
1203235332032303920323138203139372031353820313439203231372032203920
3132382032323520313620313634203133392031313920323 25D3A39307D terrad_1
| 1:47PM INF committed state app_hash=76553AF63E1D0-
DA025C7315F46C8D3DDFDD1DAC59E95D9020980E110A48B7716 height=144
module=state num_txs=0 terrad_1 | 1:47PM INF indexed block
height=144 module=txindex Now, you should install cargo: To install
cargo, run these 3 commands in the terminal separately: cargo
install cargo-generate --features vendored-openssl cargo install
cargo-run-scrip
```

For the following command in the terminal, instead of YourProjectName, you can enter any name you want for your project to be created (make sure you run this command in the directory of local Terra you have just git cloned)::

```
cargo generate --git https://github.com/CosmWasm/cw-template.git --
name YourProjectName
```

Result:

```
Generating template ... [ 1/36] Done: .cargo/config [ 2/36]
Done: .cargo [ 3/36] Skipped: .circleci/config.yml [ 4/36]
Done: .circleci [ 1/36] Done: .cargo/config [ 2/36] Done: .cargo
[ 3/36] Skipped: .circleci/config.yml [ 4/36] Done: .circleci
[ 5/36] Done: .editorconfig [ 6/36] Done:
.github/workflows/Basic.yml [ 7/36] Done: .github/workflows [ 8/36]
Done: .github [ 9/36] Done: .gitignore [10/36]
Done: .gitpod.Dockerfile [11/36] Done: .gitpod.yml [ 1/36]
Done: .cargo/config [ 2/36] Done: .cargo [ 3/36]
Skipped: .circleci/config.yml [ 4/36] Done: .circleci [ 5/36]
Done: .editorconfig [ 6/36] Done: .github/workflows/Basic.yml
[ 7/36] Done: .github/workflows [ 8/36] Done: .github [ 9/36]
Done: .gitignore [10/36] Done: .gitpod.Dockerfile [11/36]
Done: .gitpod.yml [12/36] Done: Cargo.lock [13/36] Done: Cargo.toml
[14/36] Done: Developing.md [15/36] Done: Importing.md [16/36] Done:
LICENSE [17/36] Done: NOTICE [ 1/36] Done: .cargo/config [ 2/36]
Done: .cargo [ 3/36] Skipped: .circleci/config.yml [ 4/36]
Done: .circleci [ 5/36] Done: .editorconfig [ 6/36] Done:
.github/workflows/Basic.yml [ 7/36] Done: .github/workflows [ 8/36]
Done: .github [ 9/36] Done: .gitignore [10/36]
Done: .gitpod.Dockerfile [11/36] Done: .gitpod.yml [12/36] Done:
Cargo.lock [13/36] Done: Cargo.toml [14/36] Done: Developing.md
[15/36] Done: Importing.md [16/36] Done: LICENSE [17/36] Done:
NOTICE [18/36] Done: Publishing.md [19/36] Done: README.md [20/36]
Done: examples/schema.rs [21/36] Done: examples [22/36] Done:
rustfmt.toml [23/36] Done: schema/count_response.json [24/36] Done:
schema/execute_msg.json [ 1/36] Done: .cargo/config [ 2/36]
Done: .cargo [ 3/36] Skipped: .circleci/config.yml [ 4/36]
Done: .circleci [ 5/36] Done: .editorconfig [ 6/36] Done:
.github/workflows/Basic.yml [ 7/36] Done: .github/workflows [ 8/36]
Done: .github [ 9/36] Done: .gitignore [10/36]
Done: .gitpod.Dockerfile [11/36] Done: .gitpod.yml [12/36] Done:
Cargo.lock [13/36] Done: Cargo.toml [14/36] Done: Developing.md
[15/36] Done: Importing.md [16/36] Done: LICENSE [17/36] Done:
NOTICE [18/36] Done: Publishing.md [19/36] Done: README.md [20/36]
Done: examples/schema.rs [21/36] Done: examples [22/36] Done:
rustfmt.toml [23/36] Done: schema/count_response.json [24/36] Done:
schema/execute_msg.json [25/36] Done: schema/instantiate_msg.json
[26/36] Done: schema/query_msg.json [27/36] Done: schema/state.json
[28/36] Done: schema [29/36] Done: src/contract.rs [30/36] Done:
```

```
src/error.rs [31/36] Done: src/helpers.rs [32/36] Done:
src/integration_tests.rs [33/36] Done: src/lib.rs [34/36] Done:
src/msg.rs [35/36] Done: src/state.rs [36/36] Done: src Moving
generated files into: `/home/mohamad/LocalTerra/my-project/np`...
Initializing a fresh Git repository Done! New project created
/home/mohamad/LocalTerra/my-project/np
```

COSMWASM SMART CONTRACTS ON LOCAL TERRA

In this section, we are going to get familiar with CosmWasm smart contracts written inRust programming language for interacting with local Terra through Python or JavaScript. These smart contracts have different functions that we need to send specific JSON messages to interact with them. In the end, we are going to create the artifacts folder containing the necessary binary files to interact with local Terra.

If you open the YourProjectName folder and in its src folder, the Msg.rs and contract.rs alongside with other Rust script files are found. The Msg.rs contract is related to the 3 kinds of messages we can send to our contract and it is composed of 3 main parts:

1. InstantiateMsg struct:

This message sets the state in the smart contract meaning an initial state must be given to the smart contract when it is launched.

2. ExecuteMsg enum:

This is a message that executes an action to the change of state, such as posting a message to the blockchain.

3. QueryMsg Query Msg:

This message is for querying data from the chain.

```
use schemars::JsonSchema;
use serde::{Deserialize, Serialize};
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq,
JsonSchema)]
pub struct InstantiateMsg {
 pub count: i32,
}
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq,
JsonSchema)]
#[serde(rename_all = "snake_case")]
pub enum ExecuteMsg {
 Increment {},
 Reset { count: i32 },
```

```
}
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq,
JsonSchema)]
#[serde(rename_all = "snake_case")]
pub enum QueryMsg {
 // GetCount returns the current count as a json-encoded number
 GetCount {},
}
// We define a custom struct for each query response
#[derive(Serialize, Deserialize, Clone, Debug, PartialEq,
JsonSchema)]
pub struct CountResponse {
 pub count: i32,
}
```

ON DEPENDENCIES FOR RUNNING LOCAL TERRA: WHAT IS INSIDE THE CONTRACT.RS FILE?

Another important Rust file is Contract.rs. In the next part, we are going to refer to the functions of this contract and use them in python. The important functions that we are going to interact with later using the python scripts are instantiated, execute, try increment, try_reset, and query_count. We will later see how we can interact with these functions with the JSON messages sent to them through python codes. As you can see the programming language that these smart contracts are written in is Rust.

```
use cosmwasm_std::entry_point;
use cosmwasm_std::{to_binary, Binary, Deps, DepsMut, Env,
MessageInfo, Response, StdResult};
use cw2::set_contract_version;
use crate::error::ContractError;
use crate::msg::{CountResponse, ExecuteMsg, InstantiateMsg,
QueryMsg};
use crate::state::{State, STATE};
// version info for migration info
const CONTRACT_NAME: &str = "crates.io:my-project";
const CONTRACT_VERSION: &str = env!("CARGO_PKG_VERSION");
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn instantiate(
 deps: DepsMut,
 _env: Env,
 info: MessageInfo,
 msg: InstantiateMsg,
) -> Result {
```

```
 let state = State {
 count: msg.count,
 owner: info.sender.clone(),
 };
 set_contract_version(deps.storage, CONTRACT_NAME,
CONTRACT_VERSION)?;
 STATE.save(deps.storage, &state)?;
 Ok(Response::new()
 .add_attribute("method", "instantiate")
 .add_attribute("owner", info.sender)
 .add_attribute("count", msg.count.to_string()))
}
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn execute(
 deps: DepsMut,
 _env: Env,
 info: MessageInfo,
 msg: ExecuteMsg,
) -> Result {
 match msg {
 ExecuteMsg::Increment {} => try_increment(deps),
 ExecuteMsg::Reset { count } => try_reset(deps, info, count),
 }
}
pub fn try_increment(deps: DepsMut) -> Result {
 STATE.update(deps.storage, |mut state| -> Result<_, ContractError> {
 state.count += 1;
 Ok(state)
 })?;
 Ok(Response::new().add_attribute("method", "try_increment"))
}
pub fn try_reset(deps: DepsMut, info: MessageInfo, count: i32) ->
Result {
 STATE.update(deps.storage, |mut state| -> Result<_, ContractError> {
 if info.sender != state.owner {
 return Err(ContractError::Unauthorized {});
 }
 state.count = count;
 Ok(state)
 })?;
 Ok(Response::new().add_attribute("method", "reset"))
}
```

```
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn query(deps: Deps, _env: Env, msg: QueryMsg) -> StdResult {
 match msg {
 QueryMsg::GetCount {} => to_binary(&query_count(deps)?),
 }
}
fn query_count(deps: Deps) -> StdResult {
 let state = STATE.load(deps.storage)?;
 Ok(CountResponse { count: state.count })
}
#[cfg(test)]
mod tests {
 use super::*;
 use cosmwasm_std::testing::{mock_dependencies, mock_env, mock_info};
 use cosmwasm_std::{coins, from_binary};
 #[test]
 fn proper_initialization() {
 let mut deps = mock_dependencies(&[]);
 let msg = InstantiateMsg { count: 17 };
 let info = mock_info("creator", &coins(1000, "earth"));
 // we can just call .unwrap() to assert this was a success
 let res = instantiate(deps.as_mut(), mock_env(), info,
msg).unwrap();
 assert_eq!(0, res.messages.len());
 // it worked, let's query the state
 let res = query(deps.as_ref(), mock_env(), QueryMsg::GetCount
{}).unwrap();
 let value: CountResponse = from_binary(&res).unwrap();
 assert_eq!(17, value.count);
 }
 #[test]
 fn increment() {
 let mut deps = mock_dependencies(&coins(2, "token"));
 let msg = InstantiateMsg { count: 17 };
 let info = mock_info("creator", &coins(2, "token"));
 let _res = instantiate(deps.as_mut(), mock_env(), info,
msg).unwrap();
 // beneficiary can release it
 let info = mock_info("anyone", &coins(2, "token"));
 let msg = ExecuteMsg::Increment {};
 let _res = execute(deps.as_mut(), mock_env(), info, msg).unwrap();
 // should increase counter by 1
```

```
 let res = query(deps.as_ref(), mock_env(), QueryMsg::GetCount
{}).unwrap();
 let value: CountResponse = from_binary(&res).unwrap();
 assert_eq!(18, value.count);
 }
 #[test]
 fn reset() {
 let mut deps = mock_dependencies(&coins(2, "token"));
 let msg = InstantiateMsg { count: 17 };
 let info = mock_info("creator", &coins(2, "token"));
 let _res = instantiate(deps.as_mut(), mock_env(), info,
msg).unwrap();
 // beneficiary can release it
 let unauth_info = mock_info("anyone", &coins(2, "token"));
 let msg = ExecuteMsg::Reset { count: 5 };
 let res = execute(deps.as_mut(), mock_env(), unauth_info, msg);
 match res {
 Err(ContractError::Unauthorized {}) => {}
 _ => panic!("Must return unauthorized error"),
 }
 // only the original creator can reset the counter
 let auth_info = mock_info("creator", &coins(2, "token"));
 let msg = ExecuteMsg::Reset { count: 5 };
 let _res = execute(deps.as_mut(), mock_env(), auth_info,
msg).unwrap();
 // should now be 5
 let res = query(deps.as_ref(), mock_env(), QueryMsg::GetCount
{}).unwrap();
 let value: CountResponse = from_binary(&res).unwrap();
 assert_eq!(5, value.count);
 }
}
```

CREATING A PROJECT FOLDER USING CARGO

Now, let's run the following command in the terminal to create binaries and other dependency files so that we can eventually interact with local Terra using python Terra SDK.

```
cargo run-script optimize
```

Result:

```
Running script 'optimize': 'docker run --rm -v "$(pwd)":/code --
mount type=volume,source="$(basename
"$(pwd)")_cache",target=/code/target --mount
type=volume,source=registry_cache,target=/usr/local/cargo/registry
cosmwasm/rustoptimizer:0.12.5 ' Info: RUSTC_WRAPPER=sccache Info:
sccache stats before build Compile requests 0 Compile requests
executed 0 Cache hits 0 Cache misses 0 Cache timeouts 0 Cache read
errors 0 Forced recaches 0 Cache write errors 0 Compilation failures
0 Cache errors 0 Non-cacheable compilations 0 Non-cacheable calls 0
Non-compilation calls 0 Unsupported compiler calls 0 Average cache
write 0.000 s Average cache read miss 0.000 s Average cache read hit
0.000 s Failed distributed compilations 0 Cache location Local disk:
"/root/.cache/sccache" Cache size 0 bytes Max cache size 10 GiB
Building contract in /code ... Finished release [optimized]
target(s) in 0.16s Creating intermediate hash for
my_project.wasm ...
ec1944cdda3c5f8f6968d59b990d6f92800f65d9655a53a45ee29984f6c5d882
./target/wasm32-unknown-unknown/release/my_project.wasm Optimizing
my_project.wasm ... Creating hashes ...
fa445512e5b3274ddda6474335d8515c660c5fcdf03b8b2d7f468c12f3e55564
my_project.wasm Info: sccache stats after build Compile requests 0
Compile requests executed 0 Cache hits 0 Cache misses 0 Cache
timeouts 0 Cache read errors 0 Forced recaches 0 Cache write errors
0 Compilation failures 0 Cache errors 0 Non-cacheable compilations 0
Non-cacheable calls 0 Non-compilation calls 0 Unsupported compiler
calls 0 Average cache write 0.000 s Average cache read miss 0.000 s
Average cache read hit 0.000 s Failed distributed compilations 0
Cache location Local disk: "/root/.cache/sccache" Cache size 0 bytes
Max cache size 10 GiB done Finished, status of exit status: 0
```

And in the project name folder, we are going to see that a new folder has been created called artifacts. Now, we are ready to write our python codes to interact with local Terra and use CosmWasm based contracts written in Rust.

CONCLUSION:

In this tutorial, we have managed to install the dependencies required to run local Terra, such as GO and Rust programming languages, Docker, Cargo, and some Github repositories like terra-core and local-terra. We have also connected to Local Terra network using the docker-compose up command. In the end, we have created our project folder and files using the cargo generate command.
Moreover, we have taken a quick look at some of the important smart contracts written in Rust programming language and got familiar with their functions so that we can interact with them later using our python scripts. We also created the artifacts folder using the cargo command to have everything set up for interacting with the local Terra.