

Skeleton and Rust Architecture

In this article, you'll learn about the basic architecture behind the FT contract that you'll develop while following this "Zero to Hero" series. You'll discover the contract's layout and you'll see how the Rust files are structured in order to build a feature-complete smart contract. New to Rust? If you are new to Rust and want to dive into smart contract development, our [Quick-start guide](#) is a great place to start.

Introduction

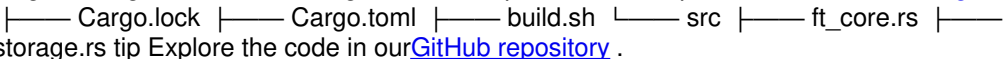
This tutorial presents the code skeleton for the FT smart contract and its file structure. You'll find how all the functions are laid out as well as the missing Rust code that needs to be filled in. Once every file and function has been covered, you'll go through the process of building the mock-up contract to confirm that your Rust toolchain works as expected.

File structure

The repository comes with many different folders. Each folder represents a different milestone of this tutorial starting with the skeleton folder and ending with the finished contract folder. If you step into any of these folders, you'll find that they each follow a regular [Rust](#) project. The file structure for these smart contracts have:

- Cargo.toml
- file to define the code dependencies (similar to package.json in JavaScript and node projects)
- src
- folder where all the Rust source files are stored
- target
- folder where the compiled wasm
- will output to
- build.sh
- script that has been added to provide a convenient way to compile the source code

Source files

File Description [ft_core.rs](#) Contains the logic for transferring and controlling FTs. This file represents the implementation of the [core](#) standard. [lib.rs](#) Holds the smart contract initialization functions and dictates what information is kept on-chain. [metadata.rs](#) Defines the metadata structure. This file represents the implementation of the [metadata](#) extension of the standard. [storage.rs](#) Contains the logic for registration and storage. This file represents the implementation of the [storage management](#) standard.  tip Explore the code in our [GitHub repository](#).

ft_core.rs

Core logic that allows you to transfer FTs between users and query for important information. Method Description [ft_transfer](#) Transfers a specified amount of FTs to a receiver ID. [ft_transfer_call](#) Transfers a specified amount of FTs to a receiver and attempts to perform a cross-contract call on the receiver's contract to execute [theft_on_transfer](#) method. The implementation of this [theft_on_transfer](#) method is up to the contract writer. You'll see an example implementation in the marketplace section of this tutorial. Once [theft_on_transfer](#) finishes executing, [ft_resolve_transfer](#) is called to check if things were successful or not. [ft_total_supply](#) Returns the total amount of fungible tokens in circulation on the contract. [ft_balance_of](#) Returns how many fungible tokens a specific user owns. [ft_on_transfer](#) Method that lives on a receiver's contract. It is called when FTs are transferred to the receiver's contract account via [theft_transfer_call](#) method. It returns how many FTs should be refunded back to the sender. [ft_resolve_transfer](#) Invoked after [theft_on_transfer](#) is finished executing. This function will refund any FTs not used by the receiver contract and will return the net number of FTs sent to the receiver after the refund (if any). 1.skeleton/src/ft_core.rs loading ... [See full example on GitHub](#) You'll learn more about these functions in the [circulating supply](#) and [transfers](#) sections of the tutorial series.

lib.rs

This file outlines what information the contract stores and keeps track of. Method Description [new_default_meta](#) Initializes the contract with default metadata so the user doesn't have to provide any input. In addition, a total supply is passed in which is sent to the owner [new](#) Initializes the contract with the user-provided metadata and total supply. Keep in mind The initialization functions ([new](#), [new_default_meta](#)) can only be called once. 1.skeleton/src/lib.rs loading ... [See full example on GitHub](#) You'll learn more about these functions in the [define a token](#) section of the tutorial series.

metadata.rs

This file is used to outline the metadata for the Fungible Token itself. In addition, you can define a function to view the contract's metadata which is part of the standard's [metadata](#) extension. Name Description FungibleTokenMetadata This structure defines the metadata for the fungible token. ft_metadata This function allows users to query for the token's metadata 1.skeleton/src/metadata.rs loading ... [See full example on GitHub](#) You'll learn more about these functions in the [define a token](#) section of the tutorial series.

storage.rs

Contains the registration logic as per the [storage management](#) standard. Method Description `storage_deposit` Payable method that receives an attached deposit of N for a given account. This will register the user on the contract.

`storage_balance_bounds` Returns the minimum and maximum allowed storage deposit required to interact with the contract. In the FT contract's case, `min = max`.

`storage_balance_of` Returns the total and available storage paid by a given user. In the FT contract's case, available is always 0 since it's used by the contract for registration and you can't overpay for storage.

1.skeleton/src/storage.rs loading ... [See full example on GitHub](#) tip You'll learn more about these functions in the [storage](#) section of the tutorial series.

Building the skeleton

- If you haven't cloned the main repository yet, open a terminal and run:

git clone <https://github.com/near-examples/ft-tutorial/> * Next, build the skeleton contract with the build script found in the `1.skeleton/build.sh` * file.

cd ft-tutorial/1.skeleton ./build.sh cd .. Since this source is just a skeleton you'll get many warnings about unused code, such as:

= note: #[warn(dead_code)] on by default

```
warning: constant is never used:GAS_FOR_RESOLVE_TRANSFER --> src/ft_core.rs:5:1 | 5 | const  
GAS_FOR_RESOLVE_TRANSFER: Gas = Gas(5_000_000_000_000); |  
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

[illegible]

warning: fungible-token (lib) generated 25 warnings Finished release [optimized] target(s) in 1.93s ✨ Done in 2.03s. Don't worry about these warnings, you're not going to deploy this contract yet. Building the skeleton is useful to validate that your Rust toolchain works properly and that you'll be able to compile improved versions of this FT contract in the upcoming tutorials.

Conclusion

You've seen the layout of this FT smart contract, and how all the functions are laid out across the different source files. Using yarn, you've been able to compile the contract, and you'll start fleshing out this skeleton in the next [section](#) of the tutorial.

Versioning for this article At the time of this writing, this example works with the following versions:

- rustc:1.6.0
- near-sdk-rs:4.0.0 [Edit this page](#) Last updated on Jan 19, 2024 by [Damián Parrino](#) Was this page helpful? [Yes](#) [No](#)

[Previous Pre-deployed Contract](#) [Next Defining Your Token](#)