**Build State Machine using Rust | Part 1 | Balances Module**

[Abhijit Roy](https://medium.com/@abhi3700?source=post_page-----0ba1d690d012--------------------------------)

·

Follow

Published in

[Rustaceans](https://medium.com/rustaceans?source=post_page-----0ba1d690d012--------------------------------)

·

5 min read

·

4 days ago

54

Alice sends money to Bob via State Machine built in Rust

Hello folks!

Welcome to the new series where we would learn Rust by building State Machine from scratch.

*This is going to be fun for devs learning Substrate in Blockchain sphere.*

**What is State Machine?**

It is simply a black box which takes an input and returns an output. It is responsible for state changes. Suppose, Alice sends 1 ETH to Bob, leading to change in the balance of both of them. This means Alice’s balance is reduced by 1 ETH and the same amount is added to that of Bob. This makes the blockchain systems **deterministic**.

A State machine that returns a deterministic output from a parsed input

More on this [here](https://paritytech.github.io/polkadot-sdk/master/polkadot_sdk_docs/reference_docs/blockchain_state_machines/index.html). The detailed explanation covers how the state transition function utilizes the current block and the previous state to determine and return the next state.

**What is Balances Module?**

It is responsible for handling all the users’ balances.

**How to code?**

**A. Setup**

Let’s do the setup first.

1. Download & install Rust from [here](https://doc.rust-lang.org/book/ch01-01-installation.html).
2. Install [Rust Analyzer](https://marketplace.visualstudio.com/items?itemName=rust-lang.rust-analyzer) extension in VSCode or your favourite editor. Get familiar with all the needed practices from [here](https://code.visualstudio.com/docs/languages/rust).

**B. Create Project**

Run the command:

$ cargo new rust-state-machine

This would create a folder structure like this:

rust-state-machine  
├── Cargo.toml  
└── src  
 └── main.rs

Then, configure a rustfmt.toml file to ensure proper formatting in the entire project with the following content:

# Basic  
edition = "2021"  
hard\_tabs = true  
max\_width = 100  
use\_small\_heuristics = "Max"  
# Imports  
imports\_granularity = "Crate"  
reorder\_imports = true  
# Consistency  
newline\_style = "Unix"  
# Misc  
chain\_width = 80  
spaces\_around\_ranges = false  
binop\_separator = "Back"  
reorder\_impl\_items = false  
match\_arm\_leading\_pipes = "Preserve"  
match\_arm\_blocks = false  
match\_block\_trailing\_comma = true  
trailing\_comma = "Vertical"  
trailing\_semicolon = false  
use\_field\_init\_shorthand = true  
# Format comments  
comment\_width = 100  
wrap\_comments = true

Also, consider adding nightly channel of Rust as this would allow $ cargo fmt command to work without any warning/error as some of the features does require nightly channel of Rust.

Then create a file named: rust-toolchain.toml and add the following content to the file.

[toolchain]  
channel = "nightly"

**C. Add module**

To add the Balances module/pallet, add this line to the main.rs file at the top like shown below.

mod balances;  
  
fn main() {}

And then create a file named balances.rs in the src directory. You should have a folder structure similar to the one shown below.

./  
├── Cargo.lock  
├── Cargo.toml  
├── rust-toolchain.toml  
├── rustfmt.toml  
└── src  
 ├── balances.rs  
 └── main.rs

**D. Add a struct**

The following struct contains the data/info for users’ balances.

pub struct Pallet {  
 balance: BTreeMap<String, u128>  
}

***Q. Why BTreeMap instead of HashMap?  
BTreeMap****is mainly used here for range-based operations. Suppose, need to fetch some keys in an sorted manner. Then, it is better than****HashMap****as is the case here. Although it doesn’t really matter much as both are efficient with O(logN) and O(1) time complexity respectively.*

**E. Implement Functions**

Now, we need to implement the methods:

* new: creates a new balances module to initiate the state for a single run. Here, w.r.t the runtime or state machine, this is initiated in the genesis block itself. Thereby, the balances field would store the users’ balances.

Here is the code:

/// previous code  
  
impl Pallet {  
 pub fn new() -> Self {  
 Self { balances: BTreeMap::new() }  
 }  
}

* set\_balance: set a balance amount for a user.

Here is the code:

/// previous code  
  
impl Pallet {  
 // previous code  
  
 fn set\_balance(&mut self, who: &String, balance: u128) {  
 self.balances.insert(who.clone(), balance);  
 }  
}

We reference the who argument in the function because the user is borrowed for use as defined with derived type. Otherwise, this function would consume this argument and hence can’t be used any further down the program in runtime code. No need to borrow for balance with primitive type.

* balance: get/view a user’s balance.

Here is the code:

impl Pallet {  
 /// previous code  
  
 fn balance(&self, who: &String) -> u128 {  
 \*self.balances.get(who).unwrap\_or(&0)  
 }  
}

*In Rust, we always try to avoid unwrap for better error handling. So, the balance of any user is defined as 0 by default. \* (dereferencing) is used to return the owned type i.e. u128 instead of &u128.*

* transfer: moves the amount from the balance of one user to another. This involves checking if it exceeds the MIN/MAX permissible value as balance.

Here is the code:

impl Pallet {  
 /// previous code  
  
 fn transfer(&mut self, from: &String, to: &String, amount: u128) -> Result<(), &'static str> {  
 // get the old balances  
 let from\_old\_balance = self.balance(from);  
 let to\_old\_balance = self.balance(to);  
  
 // calculate the new balances  
 let from\_new\_balance =  
 from\_old\_balance.checked\_sub(amount).ok\_or("Insufficient balance")?;  
 let to\_new\_balance = to\_old\_balance.checked\_add(amount).ok\_or("Exceeds MAX balance")?;  
  
 // set the new balances  
 self.set\_balance(from, from\_new\_balance);  
 self.set\_balance(to, to\_new\_balance);  
  
 Ok(())  
 }  
}

The function return type is chosen to be Result<(), &'static str for error handling. The error would be thrown as Err("some message"), if any, else returns as Ok(()).

When calculating the new balances, the conditions of MIN/MAX is checked using ok\_or as it returns Result<T, E> type to match the function’s return type.

**F. Write tests (integration)**

Now, we need to write some tests to see if it’s running fine.

*The tests are mainly based on integration test.*

For simplicity, 2 tests have been written here:

* init\_balances: Test if the balances are instantiated properly with 0as balance for a new user.

Here is the code:

// previous code  
  
#[cfg(test)]  
mod tests {  
 #[test]  
 fn init\_balances() {  
 let mut balances = super::Pallet::new();  
  
 assert\_eq!(balances.balance(&"alice".to\_string()), 0);  
 balances.set\_balance(&"alice".to\_string(), 100);  
 assert\_eq!(balances.balance(&"alice".to\_string()), 100);  
 assert\_eq!(balances.balance(&"bob".to\_string()), 0);  
 }  
}

We check if Alice has zero balance in the beginning when the Balances module is instantiated/created. And then a new balance is set for Alice which is supposed to change the balance to 100. But, for Bob the balance remains to be zero as preset value.

* test\_transfer: Test if amount is being properly transferred from one user to another.

Here is the code:

#[cfg(test)]  
mod tests {  
 // previous code  
  
 #[test]  
 fn test\_transfer() {  
 let mut balances = super::Pallet::new();  
  
 assert\_eq!(  
 balances.transfer(&"alice".to\_string(), &"bob".to\_string(), 100),  
 Err("Insufficient balance")  
 );  
  
 balances.set\_balance(&"alice".to\_string(), 100);  
 assert!(balances.transfer(&"alice".to\_string(), &"bob".to\_string(), 51).is\_ok());  
  
 // check the balances  
 assert\_eq!(balances.balance(&"alice".to\_string()), 49);  
 assert\_eq!(balances.balance(&"bob".to\_string()), 51);  
 }  
}

After instantiating the Balances module/pallet, we test if Alice is able to transfer any amount. But it fails ❌ as expected.

Then, we view the balances of Alice and Bob after transferring some amount from 100 as previous balance of Alice.

[**Full Code**](https://gist.github.com/abhi3700/14ef90859bc23801d1b6862139593d5d)**👨‍💻**

**Summary**

This Balances module/pallet is a simple way to understand how a token’s operation works under the hood. This is one way of looking at how the state machine behaves like a deterministic one.

*I give a huge shoutout 📢 to some of the works already done by engineers at Parity. My goal has been to simplify this as much as possible. And also to add my perspective wherever I thought as required.*

**Rustaceans 🚀**

Thank you for being a part of the Rustaceans community! Before you go:

* Show your appreciation with a clap and follow the publication
* Discover how you can contribute your own insights to ***[Rustaceans](https://medium.com/rustaceans/rustaceans-submissions-are-now-open-924e67cc595d)***.
* Connect with us: [***X***](https://twitter.com/rustaceans_rs) | [***Weekly Rust Newsletter***](https://substack.com/@rustbytes)