RChain – A basic Blockchain implementation written in pure Rust

What is inside the story?

In this article, I will take you a bit along my journey into Blockchain. The next project at my work will be fueled by blockchain technology. Since I had/have no real idea about that topic apart from some general high level ideas, I started to grind for information on the topic and finally decided that the best way to overcome the level of just repeating buzzwords and to check my actual level of **understanding** is to get the hands dirty: **Implementing the thing from scratch**. Not everything which would be needed to deploy your own public blockchain, but a working part of the base technology which already resembles a fully functional (local) blockchain. We will use the rather new programming language **Rust** to do so. Since the language does not have a huge user base like Java, C++ or Python, that module will be very **detailed** in the implementation part by not only showcasing the code but **explaining its ideas and syntax**.

- General insights
    - Understand what a blockchain is doing.
        - Ingredients of blockchain frameworks
        - Use cases
        - General types of blockchains (public / private)
    - Blockchain Features
        - Different Types of Transactions
        - Hashing
        - Proof of Work (Part II)
        - Authentication (Signing / Public & Private Keys; Part II)
- Coding takeaways

- Some basic Rust syntax & data structures
- Modules and visibility
- Application of Hash functions
- Generating key pairs for and sign messages (Part II)

There is a lot to cover. If you follow up, you will have a natural **understanding of blockchains,** and you will have a basic, extensible **blockchain prototype** in your hands. Who knows what that might be good for in future 😊? But don't worry, I will try to guide you step by step through the whole thing.

Prerequisites

To understand the article the only thing you will have to bring with you is a basic level of programming experience and the wish to dive into the topic of blockchains. Working knowledge specifically in Rust is not strictly necessary. I will explain the relevant concepts on the go. However, an installation of Rust[1] and a proper code editor would be a plus (e.g.., CLion or Visual Studio Code) if you want to try the code and follow up with the details is recommended.

Fast Track

- Repository: https://github.com/Mereep/rchain
- Code Description: Chapter "Implementation PART I"
- Using the module / showcase: Chapter "Usage example"
- Code documentation: Github HTML preview

Fundamentals

The first part of the article will guide you through some basics and my personal view of the technology in order to have everyone at the same page.

What is a Blockchain?

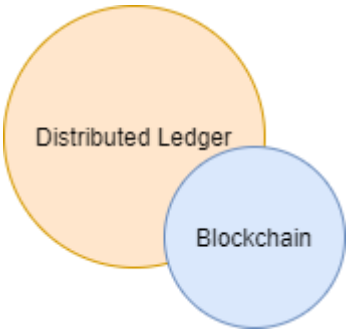
First things first: let us talk a little bit about the blockchain itself. If we want to do so, we first should talk about **Distributed Ledgers**. You have probably heard this term in the same breath as blockchain. Maybe you have come around the peer-to-peer filesharing protocol BitTorrent[2]. This protocol enables filesharing in a **decentralized** manner. Normally, when downloading a file from somewhere over the internet, you will send a request to a **specific server** which stores the whole file and transmits the data to your device. This is classical **client server architecture.** BitTorrent however will share a single file across **multiple actors / peers** at the same time. If you download a file from the Torrent network, you effectively will download it from multiple sources and eventually becoming a source yourself by participating in the network. Distributed ledgers share some common characteristics. The data (i.e., the ledger) - which generally is not just a simple file but rather a kind of database or really **whatever you want it to be** - is **distributed across multiple actors / peers.** There is no single authority in charge for keeping, modifying or distributing that ledger. Rather, all the connected **peers** keep a **copy of the very same ledger.**

This idea comes with a lot of problems and questions like: Who can add information to the network? Who can edit it? Who can read it? How much data can be added? Who pays for the storage? Who pays for the processing? Which data is added first? How to recognize failures? How to prevent malicious participants? How authentication is managed? What about network latency? That's a lot of

[1] https://www.rust-lang.org/tools/install
[2] https://en.wikipedia.org/wiki/BitTorrent

complicated tech to cover. All those questions need some form of **consensus** which all participating peers agree on. Consensus is a rather long topic on its own. There are different types of consensus algorithms suiting different use cases[3] all having their own advantages and disadvantages. In this article we will implement a sort of a **Proof of Work** variant, which is used in known public blockchains like **Ethereum** (might switch to **Proof of Stake** in future versions) or **Bitcoin.** We will come back on that.

Having that said, what is the **connection** between **distributed ledgers and blockchains**? To be honest: Taxonomy wise - it's not that simple[4]. If we want to visualize the relationship in a Venn diagram it might look like that:



So, is a blockchain a distributed ledger? Well it might be; sort of. Does it need to be? Not necessarily. A **blockchain** itself is just a kind of **ledger**. It is a special kind of ledger which **remembers all states-transitions** it ever went through. Imagine having a regular database, where **every change** / **transaction** you commit will be **recorded** such that the final state of the database can always be obtained by executing all the recorded transactions in order. The final state in the context of blockchains is referred as **World State**. Since this state does not contain the whole history of transactions, it is typically much smaller than the actual blockchain. However, it is not strictly necessary to store the world state since it always could be retrieved by replaying all transactions. This **chain of ordered transactions** is **secured** in a way that it is impossible to change any intermediate transaction (previous chain parts) without breaking the whole thing apart. This is often referred as **immutable** state. If you share that blockchain around multiple parties (including the whole world if public) by taking in account the previously mentioned features on consensus, well then you have a distributed ledger where the ledger is that blockchain.

But why is it that people talk about blockchains when they are talking about concrete realizations like Bitcoin or Ethereum? Well, the **blockchain** itself **is a part** of all of them. I would rather refer to them as **Blockchain Frameworks.** Marketing decided that this is the same, but keep in mind: it really isn't. Blockchain Frameworks typically offer a lot of technology and software, mostly together with an **ecosystem** of additional **components** offering different sets of **features,** possibly also with an existing **user base / network**. There are a lot of blockchain frameworks in the wild, which may or may not be compatible to another. Next to the already mentioned Ethereum and Bitcoin, there are prominent representatives like *Hyperledger Fabric, Hyperledger Besu, Substrate, Quora* or *Corda* just to name a view; all having different features and use cases / audiences[5].

[3] https://www.sciencedirect.com/science/article/pii/S240595951930164X#fig1
[4] https://www.theverge.com/2018/3/7/17091766/blockchain-bitcoin-ethereum-cryptocurrency-meaning
[5] https://www.leewayhertz.com/blockchain-platforms-for-top-blockchain-companies/

What we need blockchains for? (Use Cases)

This question is not so easy to answer and might also be somewhat opinionated. Blockchain Frameworks can be categorized into at least two different flavors: **public blockchains** and **private blockchains**. Those are not fundamentally different, but their components and use cases differ.

Public blockchains are basically accessible by anyone. People can join and participate if they speak the same language, i.e., following the same **protocols** (i.e., having the same consensus). Therefore, there exists a multitude of clients for specific blockchains. Ethereum has more than 10 active projects listed[6] which are implemented in all sorts of programming languages including in Rust (*Parity Ethereum*). The most obvious use case for public blockchains is providing means to supply and store digital value. Bitcoin, the most prominent player in the field, is doing exactly that by providing a digital currency: The Bitcoin (BTC). Due to the limited amount of Coins, the immutability of the ledger and the decentralized nature, those values can neither be randomly appearing, disappearing nor can anybody embezzle anything. They are unique and steady. Changes are not opaque anymore. And here is the important takeaway: there is no need for any participant to **trust** in anybody. Everyone is watching. It is almost impossible to manipulate the system, since the whole system doesn't depend only on a single actor. That also means, **you cannot DDoS[7] the network**. Those attack vectors just disappear. **Security** comes **by design**. Surely enough, currency and coins are not the only thing that can be stored. Contracts and Wills saved forever immutably? Land Registry[8]? Proofing possession of digital goods? Your poem being stored forever? Publication of information that no one ever can destroy? Blockchain got you covered. With features like the Ethereum Virtual Machine (EVM[9]) it is even possible to deploy logic (i.e., code that does state transitions) onto the chain. There possibilities and challenges seem promising for the future. What might smart and creative people come up with in the future?

Private Blockchains are basically not different from public blockchains. They use the same technology but come with additional features, which are not common in public chains. Private blockchains are not meant to be joined by anyone but a **regulated audience**. Those could be different organizations which are sharing some joint workflows, possessions or values. To enable a **collaboration** like that, frameworks also needs features like authentication, authorization, **permissioning** (user roles), rights management, deployment, networking, monitoring and so forth. But why should anyone want to deal with such complicated technology just to share a chain of transactions and data between some limited companies? Well, the audience seems way smaller. Remember the shared state which all the participants store and the consensus which the participants agree on? There is **no central authority** which you just have to **trust**. No one can just manipulate data. There are no two valid realities existing at a time. Also, since every party stores at least one copy of the ledger, there is **no risk** on **loosing access** to the data. If you share some interest with someone else but you cannot absolutely trust on the integrity of the company, private blockchains may have you covered.

I hope that introduction has helped you a bit on gaining a feeling of the technology itself and got us on the same page. If you feel like something is missing or are disagreeing on some statements, please do not hesitate to drop me a message. I really appreciate discussions.

---

[6] https://github.com/ethereum/wiki/wiki/Clients,-tools,-dapp-browsers,-wallets-and-other-projects
[7] https://de.wikipedia.org/wiki/Denial_of_Service
[8] https://medium.com/coreledger/land-registry-on-blockchain-a0da4dd25ea6
[9] https://ethdocs.org/en/latest/contracts-and-transactions/developer-tools.html#the-evm

Implementing the blockchain – PART I

This section will start on implementing the blockchain itself. We will define the project **layout** including the basic **data structures** which constitute the blockchain. Those will serve as a **skeleton** for all the implementations that are necessary later. Please note that I am by no means a Rust (or blockchain) expert. In fact, this is my first Rust project. Given that I am not a C(++) developer, the code may not be idiomatic at all. Please do not use the code I provide as a blueprint for anything you do which might be security relevant in any form. You have been warned. However, I hope it will make the underlying concepts clear and building blocks clear. Again, any feedback you drop me in will be gladly used to improve the code.

The structure of the project

We will go with a very simple folder structure, which basically consists of only **three files**. A main file (*main.rs*) which will contain the code that will invoke the blockchain and does some sample operations to demonstrate the functionality. Another file which will be inside a subdirectory will contain our actual blockchain data structures and implementations. The last file will be the **cargo.toml** file which will hold the external dependencies of the project together with some project meta data. The structure will be as follows:

```
/rchain
|-- src
|---- rchain
|------ mod.rs
|-- main.rs
|-- cargo.toml
```

The *cargo.toml* comes with only one dependency at the moment which will be a hashing-library (*Blake2*[10]) and looks like that:

```
[package]
name = "rchain_v1"
version = "0.1.0"
authors = ["Richard Vogel <webdes87@gmail.com>"]
edition = "2018"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
blake2 = "*"
```

The first time you run cargo build within the directory to build the project and create executables, cargo will download and compile the dependencies for your architecture. That is a great step from complicated dependency management in C(++): it is similar to what **pip** for Python is doing. At the time of writing there are over 44,000 so called **crates** in the index[11]. This is not the same as 255,683

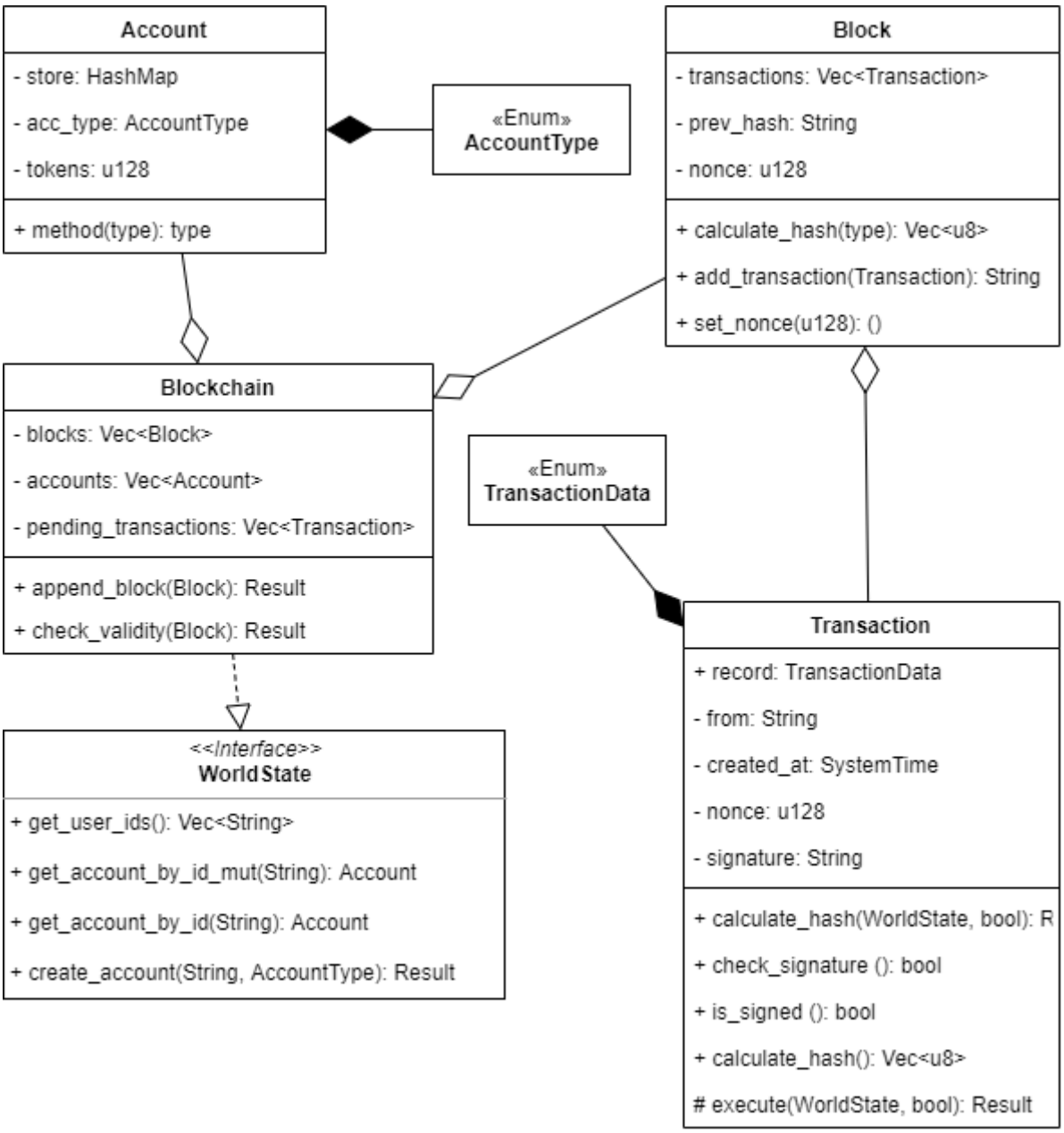
[10] https://docs.rs/blake2/0.9.0/blake2/
[11] https://crates.io

releases in the Python package index but should contain something for most usage purposes. You can also call C-libraries from Rust.

The *blockchain* directory within the project is a **module**. This is Rust's way to organize your source code into multiple files and directories. Again, this is conceptionally similar to Python's modules (without you having to include a *__init__.py* file). You can happily put modules into modules, creating some means of hierarchy. Contrary to Python, Rust supports access modifiers, which can be used to allow and prevent other modules or submodules from accessing specific members.

The Blockchain module

Let's go into the details of the implementation itself. We start with a **UML class diagram** to illustrate how everything falls together within the module (although Rust doesn't have explicit classes). Refer to this graphic to this place when you feel you got lost in all the details.



Some details are a bit simplified or omitted in order to keep the diagram reasonable. We will walk through each of the components step by step. The **code** itself will be heavily **commented and documented**. Please **read the code snippets** which I provide here (also read the code in the repository to get the full picture), even if you do not understand every small detail of the syntax. Rust can be a bit hard on newcomers, especially if you come from a high-level language like JavaScript or Python. I will explain some of the more peculiar Rusty syntax within the article. If you have little to no coding experience and just want to know how blockchains work: still **read the code comments**! The **first part** will focus on the **data structures** itself (i.e., what purpose they serve and what data they contain). The **second part** will focus on some of the more important **implementations** (i.e., functions and methods). The **third part** will finally demonstrate a simple **usage** of the Blockchain-Module.

Data structures

This section of the implementation will explain all data structures and all their attributes / members in detail. Rust lets us split the declaration of the data and its implementation into two different parts. If you have ever written some C oder C++ code, that might be similar to header (.h/.hpp-files) and (.c / .cpp) files which separate declaration and implementation. However, in rust only the data members are declared in this declarative part (except for traits, which are somewhat special to Rust; we will see an example of them here). This section will also provide many concepts of blockchain-relevant concepts. Again, even if you do not understand all the code details, read the text which will make you understand the meaning and usage of each building block.

Blockchain

The structure of the blockchain struct is as follows:

```rust
/// The actual Blockchain container
#[derive(Debug, Clone)]
pub struct Blockchain {
    /// Stores all the blocks which are accepted already within the blockchain
    pub blocks: Vec<Block>,

    /// Lookup from AccountID (will be a public key later) to Account.
    /// Effectively, this represents the WorldState
    pub accounts: HashMap<String, Account>,

    /// Will store transactions which should be added to the chain
    /// but aren't yet
    pending_transactions: Vec<Transaction>
}
```

Rust does **not** have a **class keyword.** This doesn't mean rust is not supporting class-like project structures. We will use **structs** instead, which feature a lot of what classes in other languages can do.

Obviously, the struct stores a list of **Blocks** which we will define in detail later. Also, there is a Map / Dictionary of **Accounts** stored, which keeps track of the current final state of the blockchain. The mapping of that map is *User Id to Account*. For now, the *user id* is arbitrary, in the next article those keys will be used to represent the **public key** of the account. Our blockchains **world state** is completely represented by this Map. Finally. **pending_transactions** will be used for bookkeeping of ready but not yet executed transactions. We will not make use of this for now. This is a preparation for the next article.

The first thing you may notice is that I used "///" for comments here instead of the typical "//". Those are **doc comments**. Rust comes shipped with its own documentation builder, which generates a nice **web interface** for **documenting whole project** and all of its dependencies. Everything that is

marked with "///" will be included as a description within the web interface. You can invoke the builder by typing **cargo doc** within the main folder.

You may also wonder for the line #[derive(Debug, Clone)]. This is the Rust syntax for injecting functionality / behaviors into a struct. Technically, this provides default-implementations of **traits**, which the compiler provides for free (the next part will go into traits). Here we use **Debug** and **Clone.** Debug is a helper for pretty-printing a struct using the **print!**-command, which otherwise would not work as easily; Clone is used to be able to create **duplicates of objects**.

Note the **pub** keyword in front of some of the struct's members. If you do not provide that keyword, those would not be visible / accessible from outside of the struct itself (i.e., default private).

WorldState

As mentioned earlier, the blockchain is just a special type of ledger that stores information in a special way. As seen in the Blockchain structure above, the blocks and the resulting accounts are stored in two different members. If we would not care for the history of the transactions but just for the result, we **could remove the blocks** altogether. Actually, we could just instead write transactions to a file or a database or really wherever you want them to be stored in whatever way you want them to be structured. Maybe take a break here and think a bit why this is true. Although it will introduce some additional difficulty to the article, I wanted to attribute that fact by modelling it into the code. Also, this demonstrates an important part of rusts syntax: the **traits**, which looks like that:

```
/// Represents the current state of the blockchain after all Blocks are executed
/// A world state is technically not necessary since we always could build the information
/// by iterating through all the blocks. Generally, this doesn't seem like a good option
/// However, we do not force the actual Blockchain to implement a WorldState but rather
/// behave like having one. This trait therefore just defines an expected interface into
our Blockchain
/// (Actually it doesn't even care if we the information is stored within a blockchain)
trait WorldState {
    /// Will bring us all registered user ids
    fn get_user_ids(&self) -> Vec<String>;

    /// Will return a account given it's id if is available (mutable)
    fn get_account_by_id_mut(&mut self, id: &String) -> Option<&mut Account>;

    /// Will return a account given it's id if is available
    fn get_account_by_id(&self, id: &String) -> Option<&Account>;

    /// Will add a new account
    fn create_account(&mut self, id: String, account_type: AccountType) -> Result<(),
&'static str>;
}
```

Traits **basically define a behavior** or **interface** which can be shared across multiple objects. This particular interface really just defines some functionality which an object should provide in order to be a *WorldState*. Our Blockchain will implement that trait later. A database backend could do also, or a JSON backend or something else. This functionality is not at all specific to block chains.

You may notice some rather unusual notation when not having seen the rust language. The first thing might be the **"-> ~" notation**. This is just the Rust way to define a return type. If you're a Python developer those look just the same as type hints, except that they're not optional and your program will not compile if you screw them. The next thing to know is that **Rust does not have a null type** (or **None** if you come from Python) and does **not support Exceptions.** Therefore, you see return values like `Option` and `Result`**.** The *Option* indicates that the function might return valid data (e.g.,

`Option<&Account>` indicates that the function **might** return a reference (**&**) to some Account-instance or not).

Similarly, for indicating a function which may fail or succeed (where you might use Exceptions in other languages) you can use the **Error**-type. That's a bit more involved in this example as this is written as "`-> Result<(), &'static str>`". Consider the two cases: a function might succeed or fail (possibly for multiple reasons, just like there are different types of Exceptions). For this reason, you will have to provide a data type for each case: the success case and the fail case. For that function we provided `()` for the success- and `&'static str` for the fail case. Both need explanation.

The () just indicates that if we succeed, we really don't want to return any additional information. It is just a type carrying no information at all (somewhat like *void* in C(++) or more generally a unit-type[12]). The `&'static str` type is a bit more special. The 'static notation really is something unique to Rust. This is a so-called lifetime specifier and is one of the reasons Rust feels like being a managed language (like Java, JavaScript or Python) while not needing any Garbage Collector[13] that tidies up the mess we leave in the memory. And as you can guess, things get complicated there. We will not dig into this. Whether can this article provide enough insides (it would fill whole other articles) nor do I feel Rust-mature enough to explain that. For now: this specific lifetime specifier just means that we want to *return a pointer to a string which is baked into the binary* (thus the value is *static* for the lifetime of the whole program). The last thing to note here is that `Result` and `Option` are technically nothing but common Rust Enumerations. We will see soon that Rust Enumerations can carry variable data within their variants, which also enables this exact use cas. Due to the frequent usage, those Enums are globally available in every module per default.

To finish that chapter, notice one uncommon keyword we did not cover: `mut`. If you want a variable in rust to really behave like a variable (i.e., being mutable), you will have to make this intent clear using the `mut` -keyword as given. Otherwise they will behave just like constants do.

Block

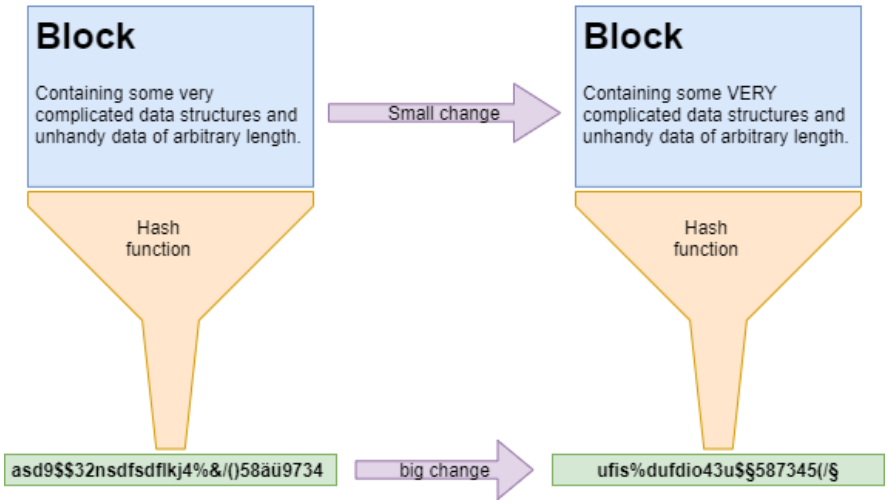
The Block structure holds the actual information (i.e., transactions) we want to store within the blockchain. Also it provides the magic that ties those pices together to actually form a blockchain. The definition looks as follows:

```
/// One single part of the blockchain.
/// Basically contains a list of transactions
#[derive(Clone, Debug)]
pub struct Block {
    /// Actions that this block includes
    /// There has to be at least one
    pub(crate) transactions: Vec<Transaction>,

    /// This actually connects the blocks together
    prev_hash: Option<String>,

    /// We store the hash of the block here also in order to
    /// save the last block from being tampered with later on
    hash: Option<String>,

    /// Some arbitrary number which will be later used for Proof of Work
    nonce: u128,
}
```

[12] en.wikipedia.org/wiki/Unit_type
[13] https://de.wikipedia.org/wiki/Garbage_Collection

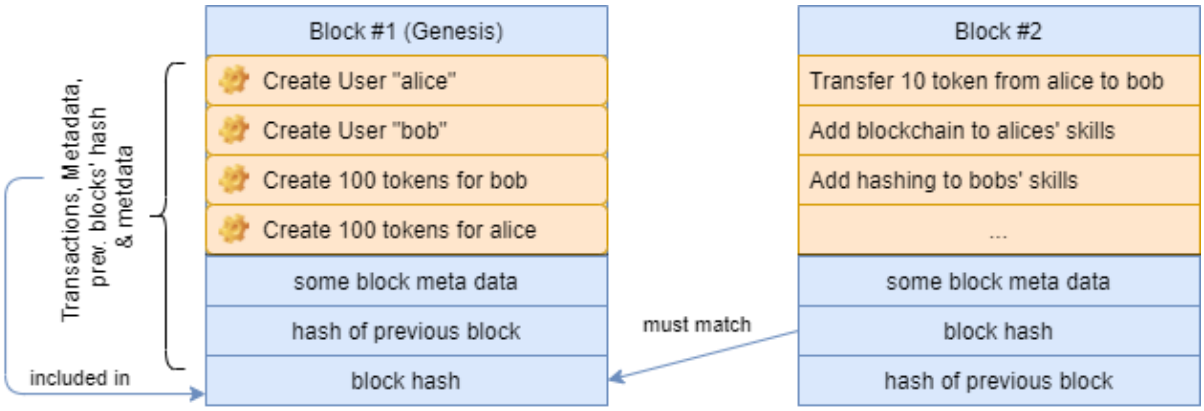
If you've read the previous chapters, you will recognize most of the syntax here, except the unique pub(crate). This is a special access modifier which allows all methods withing that project access the variable, while third parties could not. We will make use of that at the usage example. Let's focus on the members.

Typically, a block does not include only one action to be performed, therefore **transactions** member stores a *vector* of all the actions it includes. The `hash` will store a normalized representation of the whole block (including all transactions) inside. For generating such a hash, there is a hash function needed. You might have heard of MD5 or SHA-hashes. Those get some **arbitrary long input** data and produce an (almost) **unique and handy output** while preserving some cryptographic properties[14]. Since those hashes are the ingredient which makes a blockchain being able to glue together, I want to graphically illustrate what a hash-function actually is doing. Note that even the smallest change in input will produce a significantly different output. This is one of the properties a good hash function should provide. We will make use of that property when implementing adding the **Proof of Work** in the next article.



We will generate and store that very hash of the block within each transaction. If someone would change just anything in the block, those hashes would not match anymore, which easily can be checked by just hashing again. And here comes **the glue of the blockchain**: Blocks (except the first one) will also store the **hash of their previous block** (here: `prev_hash`). So, if **someone tampers** with the blockchain, that person would not only invalidate that very same block but also would **invalidate all pointers to previous hashes in the upcoming blocks**. **The whole blockchain will fall apart.** A concrete blockchain with two connected blocks could look like that:

[14] https://www.tutorialspoint.com/cryptography/cryptography_hash_functions.htm



Normally, adding blocks to a distributed blockchain is **computationally intensive** and / or needs authorization. Again, we will see why when adding the **Proof of Work** in part II to our blockchain. For now, let's just say, that this is practically infeasible in real scenarios unless quantum computers might change that. The last variable, the `nonce` (number used once) will be (miss-)used later too for that the proof of work. We will talk about that nonce also when talking about Transactions.

This finishes the description of the block itself. Please make sure understanding the topic of hashes, and how they are used to tie the blocks together, since it is the **integral component of the blockchains' tamper-resistance**.

Transaction

The *Transaction* structure represents some request to the blockchain which on execution will lead to a change of the *WorldState* (a state transition) and looks as follows:

```
/// Stores a request to the blockchain
#[derive(Clone, Debug)]
pub struct Transaction {

    /// Unique number (will be used for randomization later; prevents replay
attacks)
    nonce: u128,

    /// Account ID
    from: String,

    /// Stores the time the transaction was created
    created_at: SystemTime,

    /// the type of the transaction and its additional information
    pub(crate) record: TransactionData,

    /// Signature of the hash of the whole message
    signature: Option<String>,
}
```

Every transaction must be linked to an existing **user** at a specific **time**, whom will be stored in the `from` and the `created_at` field, respectively. Those are probably the easiest to understand fields.

The `record` holds the actual information on the change that we want to commit. It represents a command together with its accompanied data (`TransactionData`), which we as the developer allow to be transmitted to the blockchain. We will watch them later.

The `signature` will hold your digital sign of the message. This field will make sure that no one except the person which is stored in the `from` field can be the one who sent the transaction, i.e., no one will be able to submit transactions on behalf of someone else. Therefore, it represents the base layer for **authentication**. This will be achieved using public and private keys which we will create directly from code**.** The second part of the implementation will dive into the details.

We already have seen the `nonce` in the block structure. A *nonce* represents a number, which will be used exactly once for a specific purpose. This is what its acronym **number used once** is representing. The system should make sure, that no two transactions having the same *nonce* should be accepted in the system. Although it is not completely implemented, I marked the code position where you would have to add some code in order to enforce that rule. But why should anybody care for this at all? Using a *nonce* will help **preventing** at least **three problems**: **networks problems**, **user errors** and **malicious intends**. All due to one single number. The **basic thought** is as follows: when not accepting a nonce twice, a **request** which will be **transmitted more than once** to the system will be **rejected**. This fixes (i) faulty network stacks, which for some reason transmit a network package twice, (ii) users or faulty user interfaces which for whatever reason accidently transmit the same message multiple times (you don't want to send the very same payment accidently twice), and (iii) some attacker might capture the request as is and – without any change – may submit it again. This attack can also make you spend your hard-earned coins multiple times, too. The attack (which is not unique to distributed blockchains) is known as **replay attack** for obvious reasons. If we would not have a nonce**,** even a **signed message cannot prevent replay attacks.**

TransactionData

The TransactionData structure is part of the transaction itself. Here we define, which actions our blockchain should be able to process. If you want to extend the prototype and add features to your personal blockchain, this data structure is the place to start. The current definition looks like that:

```
/// A single operation to be stored on the chain
/// Noticeable, enums in rust actually can carry data in a
/// tuple-like structure (CreateUserAccount) or a dictionary-like (the
ChangeStoreValue)
#[derive(Clone, Debug, PartialEq)]
pub enum TransactionData {

    /// Will be used to store a new user account
    CreateUserAccount (String),

    /// Will be used to change or create a arbitrary value into an account
    ChangeStoreValue {key: String, value: String},

    /// Will be used to move tokens from one owner to another
    TransferTokens {to: String, amount: u128},

    /// Just create tokens out of nowhere
    CreateTokens {receiver: String, amount: u128},

    // ... Extend it as you wish, you get the idea
}
```

You may have noticed that this time, we are not defining a `struct` but rather an enumeration (`enum`). Basically, an enumeration represents a data type that **only holds a defined set of values** (variants) and only exactly one at a time. Most programming languages support enumerations. This specific example defines the four variants *`CreateUserAccount`*, *`ChangeStoreValue`*, *`TransferTokens`*, *`CreateTokens`*. Somewhat **specific to Rust** is that every variant may hold **additional variable** data.

The *CreateUserAccount* holds exactly one **unnamed** value of type `String` which represents the account id of the user account which should be created. Those arguments can also be named for better clarity (as in *TransferTokens* {to: String, amount: u128}). The upcoming implementations will provide an example on how to use and destructure those values.

Account and AccountType

Finally, we will focus on the definition of the Accounts which represents the result of the execution of all blocks within the blockchain (i.e., the world state):

```
/// Represents an account on the blockchain
/// This is basically the primary part of the "world state" of the blockchain
/// It is the final status after performing all blocks in order
#[derive(Clone, Debug)]
pub struct Account {

    /// We want the account to be able to store any information we want
(Dictionary)
    store: HashMap<String, String>,

    /// store if this is a user account or sth else
    acc_type: AccountType,

    /// Amount of tokens that account owns (like BTC or ETH)
    tokens: u128,
}
```

An account of our blockchain hold `tokens` which represent the accounts credit (i.e., this carries the value like ETH or BTC). I did not decide for name of the coins, so if you feel like, just refactor that member to your liking. The meaning of the other two members may not be obvious. Let's start with the `store`. Again, that member holds a dictionary of values. As you may have noticed, there was a variant *ChangeStoreValue* defined for the `TransactionData`. This transaction is meant to write **arbitrary values** into the user account and store it into the `store`. We could use it for example to store different kind of digital values (e.g., purchases of digital music) or physical values (e.g., owned properties) or just any other information we like into the account.

The remaining member `acc_type` is intended to represent the **type of account** and is modelled as enumeration just like `TransactionData`:

```
/// We can support different types of accounts
/// which could be used to represent different roles within the system
/// This is just for later extension, for now we will only use User accounts
#[derive(Clone, Debug)]
pub enum AccountType {

    /// A common user account
    User,

    /// An account that technically does not represent an individual
    /// Think of this like a SmartContract in Ethereum. We will not use it
    /// in our implementation. It's just here if you want to go on implementing
    /// to provide a starting point for more :)
    Contract,

    /// Add whatever roles you need.
    /// Again, we will NOT make use of this for the example here
    Validator { // Again, enum members in rust may store additional data
        correctly_validated_blocks: u128,
```

```
        incorrectly_validated_blocks: u128,
        you_get_the_idea: bool
    },
}
```

Different accounts can be used as the base building block for realizing your own realization of **permissioning**. Maybe you want only specific accounts to be able to submit specific requests? Or only some accounts have permissions to add or verify blocks, create tokens or hold business logic (like smart contracts or chain code)? Here is the starting point for those implementations.

This finishes the declarations of the data structures. If you made it until here: you have **covered** almost anything which is needed to understand the nuts and bolts of this blockchain, **well done** 😊. The rest will be needy greedy details. But that is what you came for, right?

Implementations

While the last section was very detailed and covered almost every line of code, this section will only **focus** on **the important parts** of the implementations. However, if you don't come here to learn some Rust: most of the details here are actually treated already informatively in the sections before. You may just skim read that section.

Account

We start with implementation of the `Account` structure. I chose that structure first because it really contains nothing interesting and no functionality at all so we can focus on Rust. So here it is:

```
impl Account {
    /// Constructor
    pub fn new(account_type: AccountType) -> Self {
        return Self {
            tokens: 0,
            acc_type: account_type,
            store: HashMap::new()
        }
    }
}
```

Structure implementations start with the `impl` keyword followed by the structure we want to implement. The only function we want to add is the `new` function. Which acts as a constructor for the Account (a function which is used to **construct** the object in a defined way). **Rust has no strict definition for constructor** functions (like *__init__()* in Python or a function that's named like the class itself in Java and C++). In fact, that the function is called new is completely arbitrary. But since constructors are so common in object-oriented thinking, defining a **function named new is the de-facto standard in** Rust for implementing constructors. The action that function performs is construction the Account with some default values. You may have noticed the return type `Self` (notice the uppercase *S*) which is also used to construct the structure. This is a **special type** that just represents the **currents structure's type** (NOT its instance, which is written with lower case *s*). If you have Python background: that's conceptionally similar to the *cls* parameter in a *@classmethod*) There as not much interesting going on except for the fact, that the `store` is initialized with an empty `HashMap` by also calling the `new` function which is an example for that de-facto standard within the standard library. The rest of the chapter will not list constructors anymore. You can expect them to be available where reasonable, we will focus on the relevant code.

Append a block to the blockchain

The functionality to add a block to the blockchain is a core ingredient. Here you will have to **enforce** all **policies** a block has to comply with in order to be accepted. Due to the function is very lengthy and doing multiple things the code will be snippet and explained in pieces.

```
impl Blockchain {

- snip -

/// Will add a block to the Blockchain
/// @TODO every simple step could be refactored into a separate function for
/// better testability and code-reusability
pub fn append_block(&mut self, block: Block) -> Result<(), String>{

    // The genesis block may create user out of nowhere,
    // and also may do some other things
    let is_genesis = self.len() == 0;
```

This snippet defines the function `append_block`. This function should add the Block and return a `Result` indicating the success or failure of the operation (in joint with an appropriate error text). The parameter `&mut self` indicates a **reference to the instance itself** (instance function) and indicates that we are going to possibly change the structures state (we possibly add a block) using the `mut` modifier. This is similar to the (implicit) this pointer in *C++*, *Java* or *JavaScript* or to the *self* parameter in Python. Now the only thing we do is creating an indicator variable `genesis` which will be *true* if the length of the blockchain is zero (i.e., we are adding the first block which is denoted as genesis block). Note that even if **Rust is a strongly typed language** you don't always have to provide the data type explicitly. In this case the compiler infers that we are defining a Boolean variable (bool). In modern *C++ and C#* you would use the *auto* or the *var* keyword in recent Java versions.

```
// Check if the hash matches the transactions
if !block.verify_own_hash() {
    return Err("The block hash is mismatching! (Code: 93820394)".into());
}

where verify_own_hash is defined as a function on Block as:

/// Checks if the hash is set and matches the blocks interna
pub fn verify_own_hash(&self) -> bool {
    if self.hash.is_some() && // Hash set
        self.hash.as_ref().unwrap().eq(
            &byte_vector_to_string(
                &self.calculate_hash())) { // Hash equals calculated hash

        return true;
    }
    false
}
```

This code snippet checks first if the hash is set and if its set it will verify if its valid by comparing it to the correct hash. This is done by calling `.is_some()` on the block's `hash` member. Remember that it is declared as `hash: Option<String>`, a variable which may or may not hold data. The indicator `.is_some()` will indicate if the variable does actually contain data. Knowing this fact, we can safely use `.unwrap()` to get the value. This call would **panic** (exiting the program with error) if no value was set. Since we do not want to take ownership of the variable inside, we use `.as_ref()` to tell Rust that we just want its reference. The call to `.eq` on that String will compare it to the calculated hash

(`.calculate_hash()`) defined for the Block. We will inspect that function later. If `verify_own_hash()` evaluates to *false* we will create the *Err* holding the appropriate error message. Notice the `.into()` call after the error message? This is the way for implicitly convert between compatible types. When creating a string using `"Some String"` we actually create a `&str` type (pointer to fixed length string somewhere in memory) but the result wants to store a String (owned possibly mutable String, which is not exactly the same), so we need to convert those.

Next, we **check** if the block is meant to be **appended to the current last block**. We do this by checking if `prev_hash` stores the actual the hash of the last block in the blockchain:

```
// Check if the newly added block is meant to be appended onto the last block
if !(block.prev_hash == self.get_last_block_hash()) {
    return Err("The new block has to point to the previous block (Code:
3948230)".into())
}
```

Even though both `prev_hash` and `hash` are of type `Option<String>` we can actually compare them via `==` since Rust provides a standard implementation which evaluates to true if both Options contain no value or both contain a value *and* both are the same. This is exactly what we want, since it deals also with the genesis block correctly which will not have a previous hash (yes, since it is the first).

Now, we **check if the block contains transactions**. This is just a policy we enforce, you might remove it if you want to support empty blocks.

```
// There has to be at least one transaction inside the queue
if block.get_transaction_count() == 0 {
    return Err("There has to be at least one transaction \
    inside the block! (Code: 9482930)".into());
}
```

There is nothing new to that code. Let's move on to something more involved: verifying each transaction in the block. We want the **transactions** to be **authentic** (not tampered with) and we want them to be **valid**, i.e., the execution of the transaction must be possible. We do not want a transaction to spent tokens, which the account does not supply. Because a block contains multiple transactions, we will need some kind of **rollback-functionality**. Otherwise, if a transaction is executed correctly and later transaction is not (forcing us to reject the block) we might end up with **an inconsistence World State**. That's something **we do not** want. Since I don't want to implement some complicated rollback algorithm, we just clone the whole state even though it really is not the best idea, since this could eat very much memory if the state is big:

```
// This is expensive and just used for rollback if some transactions succeed
whilst
// others don't (prevent inconsistent states)
// Arguably, that could be implemented more resource-aware
let old_state = self.accounts.clone();
```

Remember the `#[derive(Clone)]` on top of each struct? **Cloning** an object actually is and rather **complicated** operations since it has to clone all members and all structs which might contain structs which, … you get the point. Rust lets us get away with a standard implementation by deriving from the clone trait for many standard data types.

Now we try to execute each operation on the chain to verify their validity:

```
// Execute each transaction and rollback if something goes wrong
for (i, transaction) in block.transactions.iter().enumerate() {

    // Execute the transaction
    if let Err(err) = transaction.execute(self, &is_genesis) {
        // Recover state on failure
        self.accounts = old_state;

        // ... and reject the block
        return Err(format!("Could not execute transaction {} due to `{}`. Rolling
back (Code: 38203984)", i+1, err));
    }
}
```

Firstly, we iterate over all the blocks' transactions this is done by a `for`-loop that iterates over (index, transaction)-pairs that are emitted by `block.transactions.iter().enumerate()`. That's equivalent to *enumerate()* in Python. Basically, `.iter()` creates an **enumerable** that emits a reference to each transaction of the list of transaction while `.enumerate()` consumes that list and **emits a pair** that not only contains the transaction but also its position in the list.

Inside the loop we try to execute each transaction by calling `transaction.execute()`, which we will inspect later. The execute function itself indicates its success or failure by returning a `-> Result`. The statement `if let Err(err)` is a pattern matching syntax which checks if the transaction returns an Error variant (`Err`) and if that's the case it assigns the Errors data (the error String) to the variable `err` before executing the block within the curly brackets. If the transaction executes correctly (i.e., if it returns the *Results'* an *Ok* variant) it just goes on with the loop. If it fails to execute, we write the old world state (`old_state`) and return an Error ourselves. The `format!` syntax builds our actual String using placeholders. Again, this is comparable to Pythons' string formatting (*.format*).

Well, if everything went fine, we just append the block and indicate everything went fine by returning the Ok-variant:

```
// Everything went fine... append the block
self.blocks.push(block);

Ok(())
```

In Part II of this series, we will **come back** to this function when adding **Proof of Work** and **authentication** (signatures).

Hashing

In the last function we already used hashes and called the Block's `calculate_hash()` function. If you do not know what those hashes actually are, please refer back to the Block data structure description. So that's how we squash a complete block into a single hash value:

```
/// Will calculate the hash of the whole block including transactions Blake2
hasher
pub fn calculate_hash(&self) -> Vec<u8> {
    let mut hasher = Blake2b::new();

    for transaction in self.transactions.iter() {
        hasher.update(transaction.calculate_hash())
    }

    let block_as_string = format!("{:?}", (&self.prev_hash, &self.nonce));
```

```
    hasher.update(&block_as_bytes);

    return Vec::from(hasher.finalize().as_ref());
}
```

This is the place where we make use of the external crate Blake2. After creating an instance of Blake2 (hasher) we loop over all transactions inside the current block instance (`&self`). For each transaction we call the hasher's `.update()` function. Each update will feed some new data into the hasher which are some arbitrary bytes (datatype `u8`), which in this code is actually a hash itself: the hash of the transaction. Yes: we **actually hash the hashes** of the transactions. Since the block itself also contains some **data** which is **unique to the block**, we will have to **take those into account** also when creating the whole block's hash. We do this by just formatting the relevant information (previous hash and nonce) to a string and feeding it to the update function. This is somewhat lazy. We could feed both values separately (which would make the representation more stable) but since those values are Option types we would have to account for them being values or none. This blows up the code unnecessarily.

To tell the hasher we are **done adding data** and start the hashing, we use its `finalize()` function, which will output the hash as an array of bytes ([`u8`]). We transform that array to a Vector of bytes and return it.

The transactions `calculate_hash()` function looks very similar to this function, that's why skip its details here.

Verify the blockchain

To check if the blockchain that is presented to you is valid (i.e., no one tampered with), we need to basically do the same checks (making sure the blocks are not tampered and correctly connected) as when adding the block to the blockchain, while just omitting some different messages. Refer to the Block's function `check_validity()` for details.

Executing a transaction

Normally, the execution of a transaction comes with some change in the world state. Specifically, for this implementation this actually is a change in the accounts dictionary of the Blockchain struct. Remember, when we dealt with the world state we modeled those specifics away and said our blockchain should implement a specific interface (the trait WorldState) instead, which just describes the functionality the backend (in that case the blockchain) should support? Our Blockchain struct implements that interface like that:

```
impl WorldState for Blockchain {

    fn get_user_ids(&self) -> Vec<String> {
        self.accounts.keys().map(|s| s.clone()).collect()
    }

… snip …
```

Here we define the implementation of trait `WorldState` for our `Blockchain`. Specifically, that snippet shows the implementation of the function `get_user_ids()`, which just returns all keys of our accounts directory.

Because we did not come around that kind of syntax for now, let me explain this expression. The `.keys()` function returns all indices of the accounts array (which represent the unique ids of the

accounts). We could use the keys as iterable in a "`for id in` …" loop. But instead we make use of functional programming paradigms here. If you coded in R or Python you have probably come around the `map` function. This is basically just a function which takes every element in a **iterable** (here: the account ids) and feeds them into another function, which can **transform** them. This idiom is technically referred to as **function composition**[15]. Here, the inline function "`|s| s.clone()`" really just takes each value into s (which is actually of type `&String`) and clones that value into a new position in memory. The final `.collect()` just turns the resulting iterable into the return type `Vec<String>`.

Now having that settled, we can now come to the actual implementation of the `execute()` function in the transaction which starts like that:

```
impl Transaction {
    … snip …

    /// Will change the world state according to the transactions commands
    pub fn execute<T: WorldState>(&self, world_state: &mut T, is_initial: &bool) -
> Result<(), &'static str> {
```

This looks a bit weird if you're not used to template parameters. The interesting part might be the `<T: WorldState>` syntax right after the functions' name `execute`. You can read the whole line like that: declare a function public function named execute which takes a parameter `world_state` whose type conforms to the trait `WorldState`. You basically **declare a new type** and set the constraint that this very type should implement the `WorldState`. You do not fix it to any specific type (we could easily just replace remove the `T` and replace it with the `world_state: &mut Blockchain`), which is arguably easier to read. However, this implementation would allow us to use any transaction we define on any backend that implements the `WorldState` trait. This backend could easily be a PostreSQL or mySQL handler. The actions we perform are for the most parts not specific to blockchains.

Having this covered, lets take a look in the implementation which goes on like that:

```
if let Some(account) = world_state.get_account_by_id(&self.from) {
    // Do some more checkups later on...
} else {
    if !is_initial {
        return Err("Account does not exist (Code: 93482390)");
    }
}
```

Before each execution we check if the user who wants to execute it exists. Here we use the traits function `get_account_by_id()` which is defined as `fn get_account_by_id(&self, id: &String) -> Option<&Account>;` (i.e., as a function which might return an account or not). Again, we destructure that `Option` by using the `if let` pattern matching syntax which will execute the block if the function returns "some account". If it doesn't we drop an error message. After the prechecks we implement the logic for each transaction:

```
// match is like a switch (pattern matching) in C++ or Java
// We will check for the type of transaction here and execute its logic
return match &self.record {

    TransactionData::CreateUserAccount(account) => {
```

[15] https://en.wikipedia.org/wiki/Function_composition

```
        world_state.create_account(account.into(), AccountType::User)
    }

    TransactionData::CreateTokens { receiver, amount } => {…},

     … snip (more implementations)…,

    _ => { // Not implemented transaction type
    Err("Unknown Transaction type (not implemented) (Code: 487289724389)")
    }
```

The statement `match &self.record` is just another way of **pattern matching**. Recall that the `record` is of type `TransactionData` which is an enumeration holding the action to execute and possibly additional data needed for execution. Here we test for each possible variant of the enumeration `TransactionData` and execute the appropriate actions on the `WorldState` if possible. If we fail to do so for any reason, we will return an *`Err`* (which would force the blockchain implementation to rollback the state). If you are interested in the concrete realizations of the omitted commands, I encourage you to watch the source 😊.

Those are all the implementations for the first part. If you made it until here and where no Rust developer, I am aware this was a lot to cover. Do not worry if some syntax still feels alien. Some of it does for me still. However, the **great error messages** spilled out by the **compiler** are often quite helpful in spotting the problems. If you write code, its sometimes even fun to fix them and make the compiler happy (I fixed like a hundred of them while writing this little piece of code 😉).

If you compile the code and see some warning indicating unused variables of code and unconstructed variants: you can safely ignore them. Most of them are just some preparations for Part II of this writeup or examples.

Usage example

This example will guide you through

1. Initializing a valid blockchain which creates and adding some transactions
2. Tampering with a transaction and detecting it
3. Tampering with a block and detecting it

The contents of the following file are found in the *main.rs* file. If you read the implementations above, this code will be fairly simple to understand. We start by declaring our module (`rchain`), import the dependencies, define the main function and create our blockchain object:

```
mod rchain;
use rchain::{Blockchain, Block, Transaction, TransactionData};
use std::borrow::BorrowMut;

fn main() {
    println!("Demo RChain Version 1\n---------");

    // Create a new Blockchain
    let mut bc = Blockchain::new();
```

Like in C(++) or Java, a Rust program have a standard entry point declared by a function named `main()`, which we defined in that snippet.

Now we create an object for the genesis block and define the users we want to create:

```
// Create an empty block (first block has no prev_block)
let mut genesis = Block::new(None);

let initial_users = vec!("alice", "bob");
```

After doing so we loop over the `initial_users` vector in order to create transactions that **create those users** in the blockchain and assign them 100,000,000 tokens each:

```
for user in initial_users {

    let create_transaction =
         Transaction::new(user.into(),
          TransactionData::CreateUserAccount(user.into()),
          0);

    let token_action =
            Transaction::new(user.into(),
             TransactionData::CreateTokens {
              receiver: user.into(),
              amount: 100_000_000},
            0);

    genesis.add_transaction(create_transaction);
    genesis.add_transaction(token_action);
}
```

The last lines add all the transactions to the genesis block. The value `0` in the transaction's constructor should be the *nonce*. However, since we do not enforce this nonce at the blockchain right now we just fill it with a dummy zero. It will be a good starting point for your **own coding** adventures **to implement the policy** to enforce of a correct **nonce**.

The created and filled block is now added to the blockchain. The result of the append function will be printed after which the whole blockchain also will be printed as a string representation:

```
let mut res = bc.append_block(genesis);
println!("Genesis block successfully added: {:?}", res);
println!("Full blockchain printout");
println!("{:#?}", bc);
```

The `{:#?}` formatter is used for **pretty-printing**. The output of the object will look almost like a valid JSON-object with spacings nicely laid out, whereas `{:?}` will not print out those spacings. You should see the line `Genesis block successfully added: Ok(())` in your terminal, after which the blockchain output starts. Notice the printout of the "`accounts`" which should contain **alice and bob both having 100,000,000 tokens**.

Now we add another block containing an action to **transfer one token from alice to bob.** The blockchain will again be printed. You should notice the changes in the balance of the accounts.

```
// Transfer 1 token from alice to bob
let mut block2 = Block::new(bc.get_last_block_hash());
block2.add_transaction(Transaction::new(
    "alice".into(),
    TransactionData::TransferTokens {to: "bob".into(), amount: 1}, 0));
```

```
res = bc.append_block(block2);
println!("Block added: {:?}", res);
println!("Full blockchain printout");
println!("{:#?}", bc);
println!("Blockchain valid: {:?}", bc.check_validity());
```

The last line should report a valid blockchain as `Blockchain valid: Ok(())`.

Now that everything seems fine, the **first attack** will start. Bob was not satisfied with this one coin Alice transferred. So, he decides to **directly change the transaction in the blockchain**. Since we do not support signed messages for now, this can be easily done:

```
// let's clone the current blockchain before tempering
let mut bc_attack_1 = bc.clone();
// get the transaction as mutable (second block, first transaction; the token
transfer)
let transaction_data = bc_attack_1.blocks[1].transactions[0].borrow_mut();

// change the amount value of the transaction INSIDE the chain
match transaction_data.record.borrow_mut() {
    &mut TransactionData::TransferTokens {to:_, ref mut amount} => {
        *amount = 100; // Actually change the value in place
    },

    _ => {} // We know that that recors is a TransferToken Action so we ignore the
rest
}
```

We first clone our working blockchain to have a new one to operate on which has the same state. Then we get ourselves a mutable reference to the second transaction in the first block using `bc_attack_1.blocks[1].transactions[0].borrow_mut()`. Now we destructure the `record` which is stored in the transaction (remember: this is a `TransactionData` struct) using the `match`-syntax. This is the *`TransferTokens`* action we wanted to manipulate. Since the values should be **changed in-place** (directly in the memory where they are stored), we have to account for that fact by using the appropriate patterns for the match construct (`&mut, ref mut amount`). The syntax `{to:_` just indicates that we do not care for that variable and silence a compiler warning.

Now we set the value at the memory position where `amount` refers to, to 100, which is Bob's desired amount of coins, using `*amount = 100`. After doing so we check the blockchain again:

```
println!("Is the Blockchain still valid? {:#?}", bc_attack_1.check_validity());
```

Which should print: Is the Blockchain still valid? `Err("Stored hash for Block #2 does not match calculated hash (Code: 665234234)",)`

The blockchain is invalid.

While Bob failed, Alice also wants to try her luck. Since she didn't want to steal coins from Bob, she changed the *`CreateTokens`* action in the genesis block to create her account having 100,000,000,000 tokens to start with. Also, she noticed Bob's mistake and didn't want to run into the same trap, so she decided to also update the manipulated blocks hash. The basic attack looks conceptually the same as Bob's attack. However, she added the following line:

```
bc_attack_2.blocks[0].update_hash();
```

But even though the block now is valid, it is not connected anymore to the next correct block:

```
println!("Is the Blockchain still valid? {:#?}", bc_attack_2.check_validity());
```

will print something like (your output will slightly differ):

```
Is the Blockchain still valid? Err("Block #1 is not connected to
previous block (Hashes do not match. Should be
`ÜQ\u{14}²Jüø9É7x;\u{86}æÇÐ\u{8d}À¤þMy<\u{19}óí\u{5}Qoó;ÞNc\u{12}u×1
qÕ@Öáø\u{87}áHÿÞ²Àpª£\u{4}·\u{1e}íµ«Þû±¬` but is
``óç\u{93},\nbÿÁ=\u{9b}°\u{98}&¼-
8$Ö~ü}òB)F°yXÙ#ãL\u{84}dÜcû/\u{9a}½I\u{16}É\u{98}¥çC;ðJè\u{9c}ý\u{9f
}\u{9a}/¼\u{1}þ<òÕû`)",)
```

The blockchain broke again

Conclusion & Outlook

This writeup covered a lot of blockchain basics and went down the rabbit hole by implementing the blockchain ledger as a simple prototype using the rather new programming language Rust. The usage of the implementation was demonstrated showing two possible attacks to the blockchain ledger. The current state of the implementation covers a blockchain prototype, which supports multiple transactions which can easily be extended. It supports a simple rollback functionality in case transactions cannot be executed.

The upcoming Part II will add the following functionality:

- Sign transactions using public/private keys to make them tamper proof
- Add a transaction queue to the blockchain which holds requested transactions which are not yet within a executed (candidates for blocks)
- Add **Proof of Work** to secure the blockchain against tampering

I hope you had a nice time reading through all the texts. Feel free to go on implementing some of the missing functionality and bring you own ideas. There are some @TODO markers which are a good starting point.

I am happy to hear any suggestions for improvements. If there is something that can be done better or easier, if there is something you'd like to add, if you see mistakes or if you need some more explanation somewhere: feel free to write me any suggestions. Just drop me a message. I hope to see you in Part 2.