Substrate Recipes 🍴😋🍴

Substrate Recipes is a collection of simple code patterns that demonstrate best practices when building blockchains with **Substrate**. The repo used to build this book is open source and open for contributions.

What is Substrate?

Substrate is a framework for building blockchains. To learn more about Substrate, see the official documentation. For a high level overview, see the following blog posts:

- What is Substrate?
- Substrate in a nutshell
- A brief summary of everything Substrate and Polkadot

How to Use This Book

Start by cloning the repo on github:

```
git clone https://github.com/substrate-developer-hub/recipes
```

As you read through the book, practice compiling and testing recipes in `recipes/kitchen`. You can't learn to code by reading about it -- play with the code in the kitchen, extract patterns, and apply them to a problem that you want to solve!

It is useful to recognize that coding is all about abstraction. To accelerate your progress, I recommend skimming the patterns in this book, composing them into interesting projects, and building your own recipes (*and then*, pay it forward and PR the repo!).

Reach out for guidance on Stack Overflow or in the Substrate Technical Riot channel.

Prerequisites

If you do not have `substrate` installed on your machine, run:

```
curl https://getsubstrate.io -sSf | bash
```

While the code compiles, read about how the Substrate runtime architecture composes modules to configure a runtime.

Module

At the moment, this resource focuses primarily on module development patterns, though there are plans to add examples of interesting runtime configurations using the existing modules. To develop in the context of the module, it is sufficient to clone the module-template

```
$ git clone https://github.com/shawntabrizi/substrate-module-template
```

build with

```
$ cargo build
```

test with

```
$ cargo test
```

Runtime

To develop in the context of the runtime, clone the substrate-node-template and add module logic to `runtime/src/template.rs`.

Updating the Runtime

Compile runtime binaries

```
cd runtime
cargo build --release
```

Delete the old chain before you start the new one (*this is a very useful command sequence when building and testing runtimes*)

```
./target/release/substrate-example purge-chain --dev
./target/release/substrate-example --dev
```

Event

In Substrate, transaction finality does not guarantee the execution of functions dependent on the given transaction. To verify that functions have executed successfully, emit an event at the bottom of the function body.

> **Events** notify the off-chain world of successful state transitions

To declare an event, use the `decl_event` macro.

- Adding Machine Example

More Resources

- `decl_event` wiki docs
- Substrate Collectables Tutorial: Creating Events

Adding Machine

A simple adding machine checks for overflow and emits an event with the result, without using storage. In the module file,

```
pub trait Trait: system::Trait {
    type Event: From<Event> + Into<<Self as system::Trait>::Event>;
}

decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        fn deposit_event() = default;

        fn add(_origin, val1: u32, val2: u32) -> Result {
            // checks for overflow
            let result = match val1.checked_add(val2) {
                Some(r) => r,
                None => return Err("Addition overflowed"),
            };
            Self::deposit_event(Event::Added(val1, val2, result));
            Ok(())
        }
    }
}

decl_event!(
    pub enum Event {
        Added(u32, u32, u32),
    }
);
```

If the addition overflows, the method will return the `"Addition overflowed"` without emitting the event. Likewise, events are generally emitted at the bottom of method bodies as an indication of correct execution of all logic therein.

NOTE: The event described above only wraps `u32` values. If we want/need the `Event` type to contain multiple types, then we declare the following in `decl_module`

```
decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        fn deposit_event<T>() = default;
        ...
    }
}
```

and also the `decl_event` would use this generic syntax

```
decl_event!(
    pub enum Event<T> {
        ...
    }
)
```

In some cases, the `where` clause can be used to specify type aliasing for more readable code

```
decl_event!(
    pub enum Event<T>
    where
        Balance = BalanceOf<T>,
        <T as system::Trait>::AccountId,
        <T as system::Trait>::BlockNumber,
        <T as system::Trait>::Hash,
    {
        FakeEvent1(AccountId, Hash, BlockNumber),
        FakeEvent2(AccountId, Balance, BlockNumber),
    }
)
```

Storage

Use the `decl_storage` macro to define type-safe, persistent data that needs to be stored on-chain.

For crypto*currencies*, storage might consist of a mapping between account keys and corresponding balances.

More generally, blockchains provide an interface to store and interact with data in a verifiable and globally irreversible way. In this context, data is stored in a series of snapshots, each of which may be accessed at a later point in time, but, once created, snapshots are considered irreversible.

Arbitrary data may be stored, as long as its data type is serializable in Substrate i.e. implements `Encode` and `Decode` traits.

Recipes

- Single Value Storage

- Lists as Maps
- Configurable Module Constants

More Resources

- `decl_storage` wiki docs

Single Value

Substrate supports all primitive Rust types (`bool`, `u8`, `u32`, etc) as well as some custom types specific to Substrate (`Hash`, `Balance`, `BlockNumber`, etc).

- Basic Storage
- Storage Interaction
- Getter Syntax
- Setter Syntax
- Substrate Specific Types

Basic Storage

Within a specific module, a single value (`u32` type) is stored in the runtime with this syntax:

```
decl_storage! {
    trait Store for Module<T: Trait> as Example {
        MyValue: u32;
    }
}
```

Storage Interaction

To interact with single storage values, it is necessary to import the `support::StorageValue` type. Functions used to access a `StorageValue` are defined in `srml/support`:

```
/// Get the storage key.
fn key() -> &'static [u8];

/// true if the value is defined in storage.
fn exists<S: Storage>(storage: &S) -> bool {
    storage.exists(Self::key())
}

/// Load the value from the provided storage instance.
fn get<S: Storage>(storage: &S) -> Self::Query;

/// Take a value from storage, removing it afterwards.
fn take<S: Storage>(storage: &S) -> Self::Query;

/// Store a value under this key into the provided storage instance.
fn put<S: Storage>(val: &T, storage: &S) {
    storage.put(Self::key(), val)
}

/// Mutate this value
fn mutate<R, F: FnOnce(&mut Self::Query) -> R, S: Storage>(f: F,
storage: &S) -> R;

/// Clear the storage value.
fn kill<S: Storage>(storage: &S) {
    storage.kill(Self::key())
}
```

Therefore, the syntax to "put" `Value`:

```
<MyValue>::put(1738);
```

and to "get" `Value`:

```
let my_val = <MyValue>::get();
```

Note that we do not need the type `T` because the value is only of one type `u32`. If the `T` was polymorphic over more than one type, the syntax would include `T` in call like

```
<MyValue<T>>::put(178);
```

Getter Syntax

Storage values can also be declared with a `get` function to provide cleaner syntax for getting values.

```
decl_storage! {
    trait Store for Module<T: Trait> as Example {
        MyValue get(value_getter): u32;
    }
}
```

The `get` parameter is optional, but, by including it, the module exposes a getter function (`fn value_getter() -> u32`).

To "get" the `Value` with the getter function

```
let my_val = Self::value_getter();
```

Setter Syntax

Here is an example of a module that stores a `u32` value in runtime storage and provides a function `set_value` to set the given `u32`. This code follows convention for naming setters in Rust.

```rust
use srml_support::{StorageValue, dispatch::Result};

pub trait Trait: system::Trait {}

decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        fn set_value(origin, value: u32) -> Result {
            // check sender signature to verify permissions
            let sender = ensure_signed(origin)?;
            <MyValue>::put(value);
            Ok(())
        }
    }
}

decl_storage! {
    trait Store for Module<T: Trait> as Example {
        MyValue: u32;
    }
}
```

Maps

To use maps in the runtime storage, first import `StorageMap` from `srml/support`

```rust
use support::{StorageMap};
```

With this type, a key-value mapping (between `u32` types) can be stored in runtime storage using the following syntax

```rust
decl_storage! {
    trait Store for Module<T: Trait> as Example {
        MyMap: map u32 => u32;
    }
}
```

Functions used to access a `StorageValue` are defined in `srml/support`:

```
/// Get the prefix key in storage.
fn prefix() -> &'static [u8];

/// Get the storage key used to fetch a value corresponding to a
specific key.
fn key_for(x: &K) -> Vec<u8>;

/// true if the value is defined in storage.
fn exists<S: Storage>(key: &K, storage: &S) -> bool {
    storage.exists(&Self::key_for(key)[..])
}

/// Load the value associated with the given key from the map.
fn get<S: Storage>(key: &K, storage: &S) -> Self::Query;

/// Take the value under a key.
fn take<S: Storage>(key: &K, storage: &S) -> Self::Query;

/// Store a value to be associated with the given key from the map.
fn insert<S: Storage>(key: &K, val: &V, storage: &S) {
    storage.put(&Self::key_for(key)[..], val);
}

/// Remove the value under a key.
fn remove<S: Storage>(key: &K, storage: &S) {
    storage.kill(&Self::key_for(key)[..]);
}

/// Mutate the value under a key.
fn mutate<R, F: FnOnce(&mut Self::Query) -> R, S: Storage>(key: &K, f:
F, storage: &S) -> R;
```

To insert a `(key, value)` pair into a `StorageMap` named `MyMap`:

```
<MyMap<T>>::insert(key, value);
```

To query `MyMap` for the `value` corresponding to a `key`:

```
let value = <MyMap<T>>::get(key);
```

Implementing Lists with Maps

Substrate does not natively support a list type since it may encourage dangerous habits. Unless explicitly guarded against, a list will add unbounded `O(n)` complexity to an operation that will only charge `O(1)` fees (Big O notation refresher). This opens an economic attack vector on your chain.

Emulate a list with a mapping and a counter like so:

```
use support::{StorageValue, StorageMap};

decl_storage! {
    trait Store for Module<T: Trait> as Example {
        TheList get(the_list): map u32 => T::AccountId;
        TheCounter get(the_counter): u32;
    }
}
```

This code allows us to store a list of participants in the runtime represented by `AccountId`s. Of course, this implementation leaves many unanswered questions such as

- How to add and remove elements?
- How to maintain order under mutating operations?
- How to verify that an element exists before removing/mutating it?

This recipe answers those questions with snippets from relevant code samples:

- Adding/Removing Elements in an Unordered List
- Swap and Pop for Ordered Lists
- Linked Map for Simplified Runtime Logic

Adding/Removing Elements in an Unbounded List

If the size of the list is not relevant, the implementation is straightforward. *Note how it is still necessary to verify the existence of elements in the map before attempting access.*

To add an `AccountId`, increment the `the_count` and insert an `AccountId` at that index:

```
// decl_module block
fn add_member(origin) -> Result {
    let who = ensure_signed(origin)?;

    // increment the counter
    <TheCounter<T>>::mutate(|count| *count + 1);

    // add member at the largest_index
    let largest_index = <TheCounter<T>>::get();
    <TheList<T>>::insert(largest_index, who.clone());

    Self::deposit_event(RawEvent::MemberAdded(who));

    Ok(())
}
```

To remove an `AccountId`, call the `remove` method for the `StorageMap` type at the relevant index. In this case, it isn't necessary to update the indices of other `proposal`s; order is not relevant.

```
// decl_module block
fn remove_member_unbounded(origin, index: u32) -> Result {
    let who = ensure_signed(origin)?;

    // verify existence
    ensure!(<TheList<T>>::exists(index), "an element doesn't exist at
this index");
    let removed_member = <TheList<T>>::get(index);
    <TheList<T>>::remove(index);

    Self::deposit_event(RawEvent::MemberRemoved(removed_member));

    Ok(())
}
```

Because the code doesn't update the indices of other `AccountId`s in the map, it is necessary to verify an `AccountId`'s existence before removing it, mutating it, or performing any other operation.

Swap and Pop for Ordered Lists

To preserve storage so that the list doesn't continue growing even after removing elements, invoke the **swap and pop** algorithm:

1. swap the element to be removed with the element at the head of the *list* (the element with the highest index in the map)
2. remove the element recently placed at the highest index
3. decrement the `TheCount` value.

Use the *swap and pop* algorithm to remove elements from the list.

```
// decl_module block
fn remove_member_ordered(origin, index: u32) -> Result {
    let who = ensure_signed(origin)?;

    ensure!(<TheList<T>>::exists(index), "an element doesn't exist at
this index");

    let largest_index = <TheCounter<T>>::get();
    let member_to_remove = <TheList<T>>::take(index);
    // swap
    if index != largest_index {
    let temp = <TheList<T>>::take(largest_index);
    <TheList<T>>::insert(index, temp);
    <TheList<T>>::insert(largest_index, member_to_remove.clone());
    }
    // pop
    <TheList<T>>::remove(largest_index);
    <TheCounter<T>>::mutate(|count| *count - 1);


Self::deposit_event(RawEvent::MemberRemoved(member_to_remove.clone()));

    Ok(())
}
```

Keep the same logic for inserting proposals (increment `TheCount` and insert the entry at the head of the list)

Linked Map

To trade performance for *relatively* simple code, utilize the `linked_map` data structure. By implementing `EnumerableStorageMap` in addition to `StorageMap`, `linked_map` provides a method `head` which yields the head of the *list*, thereby making it unnecessary to also store the `LargestIndex`. The `enumerate` method also returns an `Iterator` ordered according to when `(key, value)` pairs were inserted into the map.

To use `linked_map`, import `EnumerableStorageMap`. Here is the new declaration in the `decl_storage` block:

```
use support::{StorageMap, EnumerableStorageMap}; // no StorageValue
necessary

decl_storage! {
    trait Store for Module<T: Trait> as Example {
        LinkedList get(linked_list): linked_map u32 => T::AccountId;
        LinkedCounter get(linked_counter): u32;
    }
}
```

The `add_member_linked` method is logically equivalent to the previous `add` method. Here is the new `remove_member_linked` method:

```
// decl_module block
fn remove_member_linked(origin, index: u32) -> Result {
    let who = ensure_signed(origin)?;

    ensure!(<LinkedList<T>>::exists(index), "A member does not exist at
this index");

    let head_index = <LinkedList<T>>::head().unwrap();
    let member_to_remove = <LinkedList<T>>::take(index);
    let head_member = <LinkedList<T>>::get(head_index);
    <LinkedList<T>>::insert(index, head_member);
    <LinkedList<T>>::remove(head_index);

    Ok(())
}
```

The only caveat is that this implementation incurs some performance costs (vs solely using `StorageMap` and `StorageValue`) because `linked_map` heap allocates the entire map as an iterator in order to implement the `enumerate` method.

Configurable Module Constants

To declare constant values within a runtime, it is necessary to import the `Get` trait from the `support` module

```
use support::traits::Get;
```

Constants can be declared in the `pub trait` block of the module using the `Get<T>` syntax for any type `T`.

```
pub trait Trait: system::Trait {
    type Event: From<Event> + Into<<Self as system::Trait>::Event>;

    type Currency: Currency<Self::AccountId> +
ReservableCurrency<Self::AccountId>;

    type MaxAddend: Get<u32>;

    // frequency with which the this value is deleted
    type ClearFrequency: Get<Self::BlockNumber>;
}
```

In order to make these constants accessible within the module, it is necessary to declare them with the `const` syntax in the `decl_module` block. Usually constants are declared at the top of this block, under `fn deposit_event`.

```
decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        fn deposit_event() = default;

        const MaxAddend: u32 = T::MaxAddend::get();

        const ClearFrequency: T::BlockNumber =
T::ClearFrequency::get();
    }
}
```

This example manipulates a single value in storage declared as `SingleValue`.

```
decl_storage! {
    trait Store for Module<T: Trait> as Example {
        SingleValue get(single_value): u32;
    }
}
```

`SingleValue` is set to `0` every `ClearFrequency` number of blocks. *This logic is in the `on_finalize` block and is covered in deeper detail in the Blockchain Event Loop recipe.*

```
fn on_finalize(n: T::BlockNumber) {
    if (n % T::ClearFrequency::get()).is_zero() {
        let c_val = <SingleValue>::get();
        <SingleValue>::put(0u32); // is this cheaper than killing?
        Self::deposit_event(Event::Cleared(c_val));
    }
}
```

Signed transactions may invoke the `add_value` runtime method to increase `SingleValue` as long as each call adds less than `MaxAddend`. *There is no anti-sybil mechanism so a user could just split a larger request into multiple smaller requests to overcome the* `MaxAddend`, but overflow is still handled appropriately.

```
fn add_value(origin, val_to_add: u32) -> Result {
    let _ = ensure_signed(origin)?;
    ensure!(val_to_add <= T::MaxAddend::get(), "value must be <=
maximum add amount constant");

    // previous single value
    let c_val = <SingleValue>::get();

    // checks for overflow
    let result = match c_val.checked_add(val_to_add) {
        Some(r) => r,
        None => return Err("Addition overflowed"),
    };
    <SingleValue>::put(result);
    Self::deposit_event(Event::Added(c_val, val_to_add, result));
    Ok(())
}
```

In more complex patterns, the constant value may be used as a static, base value that is scaled by a multiplier to incorporate stateful context for calculating some dynamic fee (ie floating transaction fees).

To see another example of how to use tuples to emulate higher order arrays, see the Substrate Collectables Tutorial.

NOTE: DoubleMap is a map with two keys; this storage item may also be useful for implementing higher order arrays

SRML Tour

srml-tour intends to explain the features of SRML modules, demonstrate use cases, and explore the code. It is *in progress*, tracked in issues.

smpl-treasury

recipe, kitchen/treasury

1. instantiate a pot
2. proxy spending through the pot
3. schedule spending with configurable module constants

smpl-treasury

This recipe demonstrates how srml/treasury instantiates a pot of funds and schedules funding. *See kitchen/treasury for the full code*

Instantiate a Pot

To instantiate a pool of funds, import `ModuleId` and `AccountIdConversion` from sr-primitives.

```
use runtime_primitives::{ModuleId, traits::AccountIdConversion};
```

With these imports, a `MODULE_ID` constant can be generated as an identifier for the pool of funds. This identifier can be converted into an `AccountId` with the `into_account()` method provided by the `AccountIdConversion` trait.

```
const MODULE_ID: ModuleId = ModuleId(*b"example ");

impl<T: Trait> Module<T> {
    pub fn account_id() -> T::AccountId {
        MODULE_ID.into_account()
    }

    fn pot() -> BalanceOf<T> {
        T::Currency::free_balance(&Self::account_id())
    }
}
```

Accessing the pot's balance is as simple as using the `Currency` trait to access the balance of the associated `AccountId`.

Proxy Transfers

In srml/treasury, approved spending proposals are queued in runtime storage before they are scheduled for execution. For the example dispatch queue, each entry represents a request to transfer `BalanceOf<T>` to `T::AccountId` from the pot.

```
decl_storage! {
    trait Store for Module<T: Trait> as STreasury {
        /// the amount, the address to which it is sent
        SpendQ get(spend_q): Vec<(T::AccountId, BalanceOf<T>)>;
    }
}
```

In other words, the dispatch queue holds the `AccountId` of the recipient (destination) in the first field of the tuple and the `BalanceOf<T>` in the second field. The runtime method for adding a spend request to the queue looks like this

```
decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        // uses the example treasury as a proxy for transferring funds
        fn proxy_transfer(origin, dest: T::AccountId, amount:
BalanceOf<T>) -> Result {
            let sender = ensure_signed(origin)?;

            let _ = T::Currency::transfer(&sender, &Self::account_id(),
amount)?;
            <SpendQ<T>>::mutate(|requests| requests.push((dest.clone(),
amount)));
            Self::deposit_event(RawEvent::ProxyTransfer(dest, amount));
            Ok(())
        }
    }
}
```

This method transfers some funds to the pot along with the request to transfer the same funds from the pot to a recipient (the input field `dest: T::AccountId`).

NOTE: *Instead of relying on direct requests, srml/treasury coordinates spending decisions through a proposal process.*

Scheduling Spending

To schedule spending like `srml/treasury`, first add a configurable module constant in the `Trait`. This constant determines how often the spending queue is executed.

```
pub trait Trait: system::Trait {
    /// Period between successive spends.
    type SpendPeriod: Get<Self::BlockNumber>;
}
```

This constant is invoked in the runtime method `on_finalize` to schedule spending every `T::SpendPeriod::get()` blocks.

```
decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        // other runtime methods
        fn on_finalize(n: T::BlockNumber) {
            if (n % T::SpendPeriod::get()).is_zero() {
                Self::spend_funds();
            }
        }
    }
}
```

To see the logic within `spend_funds`, see the kitchen/treasury. This recipe could be extended to give priority to certain spend requests or set a cap on the spends for a given `spend_funds()` call.

Safety and Optimization

Unlike conventional software development kits that abstract away low-level decisions, Substrate grants developers fine-grain control over the underlying implementation. This approach fosters high-performance, modular applications. At the same time, it also demands increased attention from developers. To quote the late Uncle Ben, **with great power comes great responsibility**.

Indeed, Substrate developers have to exercise incredible caution. The bare-metal control that they maintain over the runtime logic introduces new attack vectors. In the context of blockchains, the cost of bugs scale with the amount of capital secured by the application. Likewise, developers should *generally* abide by a few rules when building with Substrate. These rules may not hold in every situation; Substrate offers optimization in context.

- Module Development Criteria

- Declarative Programming
- Optimizations

Testing

Testing is not (yet) covered in the Substrate Recipes, but there is a great introduction to testing in the context of Substrate in the Crypto Collectables Tutorial. I also have enjoyed the following articles/papers on testing that apply to code organization more generally:

- Conditional Compilation and Rust Unit Testing
- Design for Testability
- How I Test
- Simple Testing Can Prevent Most Critical Failures

Module Development Criteria

1. Modules should be independent pieces of code; if your module is tied to many other modules, it should be a smart contract. See the substrate-contracts-workshop for more details with respect to smart contract programming on Substrate.

2. It should not be possible for your code to panic after storage changes. Poor error handling in Substrate can *brick* the blockchain, rendering it useless thereafter. With this in mind, it is very important to structure code according to declarative, condition-oriented design patterns. *See more in the declarative programming section.*

Declarative Programming

Within each runtime module function, it is important to perform all checks prior to any storage changes. When coding on most smart contract platforms, the stakes are lower because panics on contract calls will revert any storage changes. Conversely, Substrate requires greater attention to detail because mid-function panics will persist any prior changes made to storage.

- Using the Ensure Macro
- Verifying Signed Messages

Using the Ensure Macro

Substrate developers should use `ensure!` checks at the top of each runtime function's logic to verify that all of the requisite checks pass before performing any storage changes. *Note that this is similar to `require()` checks at the top of function bodies in Solidity contracts.*

The Social Network recipe demonstrated how we can create separate runtime methods to verify necessary conditions in the main methods.

```
impl<T: Trait> Module<T> {
  pub fn friend_exists(current: T::AccountId, friend: T::AccountId) ->
bool {
    // search for friend in AllFriends vector
    <AllFriends<T>>::get(current).iter()
          .any(|&ref a| a == &friend)
  }

  pub fn is_blocked(current: T::AccountId, other_user: T::AccountId) ->
bool {
    // search for friend in Blocked vector
    <Blocked<T>>::get(current).iter()
          .any(|&ref a| a == &other_user)
  }
}
```

"*By returning `bool`, we can easily use these methods in `ensure!` statements to verify relevant state conditions before making requests in the main runtime methods.*"

```
// in the remove_friend method
ensure!(Self::friend_exists(user.clone(), old_friend.clone()), "old
friend is not a friend");

...
// in the block method
ensure!(!Self::is_blocked(user.clone(), blocked_user.clone()), "user is
already blocked");
```

Indeed, this pattern of extracting runtime checks into separate functions and invoking the `ensure` macro in their place is useful. It produces readable code and encourages targeted testing to more easily identify the source of logic errors.

For a deeper dive into the "Verify First, Write Last" pattern, see the relevant section in the Substrate Collectables tutorial as well as Substrate Best Practices. This github comment is also very useful for visualizing the declarative pattern in practice.

Bonus Reading

- Design for Testability
- Condition-Oriented Programming
- Declarative Smart Contracts

Verifying Signed Messages

It is often useful to designate some functions as permissioned and, therefore, accessible only to a defined group. In this case, we check that the transaction that invokes the runtime function is signed before verifying that the signature corresponds to a member of the permissioned set.

```
let who = ensure_signed(origin)?;
ensure!(Self::is_member(&who), "user is not a member of the group");
```

We can define `is_member` similar to the helper methods in the Social Network recipe by defining a vector of `AccountId`s (`current_member`) that contains all members. We then search this vector for the `AccountId` in question within the body of the `is_member` method.

```
impl<T: Trait> Module<T> {
    pub fn is_member(who: &T::AccountId) -> bool {
        Self::current_member().iter()
            .any(|&ref a| a == who)
    }
}
```

To read more about checking for signed messages, see the relevant section in the Substrate collectables tutorial.

Checking for Collisions

Often times we may intend for keys to be unique identifiers that map to a specific storage item. In this case, it is necessary to check for collisions before adding new entries.

For example, it is common to use the hash of an object as the unique identifier in a map defined in the `decl_storage` block. Before adding a new value to the map, check that the key (hash) doesn't already have an associated value in the map. If it does, it is necessary to decide between the new item and the existing item to prevent an inadvertent key collision. In most cases, the new value is rejected.

```
fn insert_value(origin, hash: Hash, value: u32) {
    // check that key doesn't have an associated value
    ensure!( !(Self::map::exists(&hash)), "key already has an
associated value" );

    // add key-value pair
    <Map<T>>::insert(hash, value);
}
```

See how the Substrate Collectables Tutorial covers this pattern.

Optimization Tricks

Runtime overhead in Substrate corresponds to the efficiency of the underlying Rust code. Therefore, it is essential to use clean, efficient Rust patterns for performance releases. This section introduces common approaches for optimizing Rust code in general and links to resources that may guide further investigation.

- Premature Optimization
- Efficiency => Security
- Zero-Cost Abstractions
- Entering `unsafe` Waters 🏴‍☠️
- Fearless Concurrency && Asynchrony

This section was inspired by and pulls heavily from

- Achieving Warp Speed with Rust by Jack Fransham, `troubles.md`
- High Performance Rust by Iban Eguia Moraza

Premature Optimization

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. - Page 268 of Structured Programming with `goto` Statements by Donald Knuth

Before worrying about performance optimizations, focus on *optimizing* for readability, simplicity, and maintainability. The first step when building anything is achieving basic functionality. Only after establishing a minimal viable sample is it appropriate to consider performance-based enhancements. With that said, severe inefficiency does open attack vectors for Substrate runtimes (*see the next section*). Moreover, the tradeoff between optimization and simplicity is not always so clear...

A common misconception is that optimized code is necessarily more complicated, and that therefore optimization always represents a trade-off. However, in practice, better factored code often runs faster and uses less memory as well. In this regard, optimization is closely related to refactoring, since in both cases we are paying into the code so that we may draw back out again later if we need to. - src

Rust API Guidelines

- Official Rust API Guidelines
- Rust Unofficial Design Patterns
- Elegant Library API Guidelines by Pascal Hertleif

Also, use clippy!

Efficiency => Security in Substrate

We call an algorithm *efficient* if its running time is polynomial in the size of the input, and *highly efficient* if its running time is linear in the size of the input. It is important for all on-chain algorithms to be highly efficient, because they must scale linearly as the size of the Polkadot network grows. In contrast, off-chain algorithms are only required to be efficient. - Web3 Research

See Substrate Best Practices for more details on how efficiency influences the runtime's economic security.

Related Reading

- Onwards; Underpriced EVM Operations, September 2016

- Under-Priced DOS Attacks on Ethereum

Rust Zero-Cost Abstractions

Substrate developers should take advantage of Rust's zero cost abstractions.

Articles

- Abstraction without overhead: traits in Rust
- Effectively Using Iterators in Rust
- Type States

Tweets

- iterate over a slice rather than a `vec!`

Video

- An introduction to structs, traits, and zero-cost abstractions

Entering `unsafe` Waters 🏴☠️

Please read The Rustonomicon before experimenting with the dark magic that is `unsafe`

To access an element in a specific position, use the `get()` method. This method performs a double bound check.

```
for arr in array_of_arrays {
    if let Some(elem) = arr.iter().get(1738) {
        println!("{}", elem);
    }
}
```

The `.get()` call performs two checks:

1. checks that the index will return `Some(elem)` or `None`
2. checks that the returned element is of type `Some` or `None`

If bound checking has already been performed independently of the call, we can invoke `.getunchecked()` to access the element. Although this is `unsafe` to use, it is equivalent to C/C++ indexing, thereby improving performance when we already know the element's location.

```
for arr in array_of_arrays {
    println!("{}", unsafe { arr.get_unchecked(1738) })
}
```

NOTE: if we don't verify the input to `.getunchecked()`, the caller may access whatever is stored in the location even if it is a memory address outside the slice

Fearless Concurrency && Asynchrony

As a systems programming language, Rust provides significant flexibility with respect to low-level optimizations. Specifically, Rust provides fine-grain control over how you perform computation, delegate said computation to the OS's threads, and schedule state transitions within a given thread. There isn't space in this book to go into significant detail, but I'll try to provide resources/reading that have helped me get up to speed. For a high-level overview, Stjepan Glavina provides the following descriptions in Lock-free Rust: Crossbeam in 2019:

- **Rayon** splits your data into distinct pieces, gives each piece to a thread to do some kind of computation on it, and finally aggregates results. Its goal is to distribute CPU-intensive tasks onto a thread pool.
- **Tokio** runs tasks which sometimes need to be paused in order to wait for asynchronous events. Handling tons of such tasks is no problem. Its goal is to distribute IO-intensive tasks onto a thread pool.
- **Crossbeam** is all about low-level concurrency: atomics, concurrent data structures, synchronization primitives. Same idea as the `std::sync` module, but bigger. Its goal is to provide tools on top of which libraries like Rayon and Tokio can be built.

To dive deeper down these 🐰 holes

- Asynchrony
- Concurrency

Asynchrony

Are we `async` yet?

Conceptual

- RustLatam 2019 - Without Boats: Zero-Cost Async IO
- Introduction to Async/Await Programming (withoutboats/wakers-i):
- Futures (by Aaron Turon)

Projects

- Rust Asynchronous Ecosystem Working Group
- romio
- Tokio Docs

Concurrency

Conceptual

- Rust Concurrency Explained
- Lock-free Rust: Crossbeam in 2019
- Crossbeam Research Meta-link

Projects

- sled
- servo
- TiKV

Dessert

Check out **awesome-substrate** for projects, events, and all the latest Substrate news!

Featured Tutorials

- Substrate Collectables Workshop
- Substrate Verifiable Credentials Workshop
- Substrate TCR Tutorial
- Substrate Contracts Workshop