Runtime storage

Runtime storage allows you to store data in your blockchain that is persisted between blocks and can be accessed from within your runtime logic. Storage should be one of the most critical concerns of a blockchain runtime developer. Well-designed storage systems reduce the load on nodes in the network, which ultimately lowers the overhead costs for participants in your blockchain In other words, the fundamental principle of blockchain runtime storage is to minimize its use.

Substrate exposes a set of layered, modular storage APIs that allow runtime developers to make the storage decisions that suit them best. This document is intended to provide information and best practices about Substrate's runtime storage interfaces.

Storage items

In Substrate, any pallet can introduce new storage items that will become part of your blockchain's state. These storage items can be simple single value items, or more complex storage maps. The type of storage items you choose to implement depends entirely on their intended role within your runtime logic.

FRAME's `Storage` module gives runtime developers access to Substrate's flexible storage APIs, which can support any value that is encodable by SCALE codec. These include:

- Storage Value - used to store any single value, such as a `u64`.
- Storage Map - used to store a key-value mapping, such as account-to-balance.
- Storage Double Map - used as an implementation of a storage map with two keys to provide the ability to efficiently remove all entries that have a common first key.
- Storage N Map - used to store a mapping with any arbitrary number of keys, it can be used as a basis to build a Triple Storage Map, a Quadruple Storage Map and so on.

Storage value

This type of storage item should be used for values that are viewed as a single unit by the runtime. This could be a single primitive value, a single `struct`, or a single collection of related items. If a storage item is used for storing lists of items, runtime developers should be conscious about the size of the lists they use. Large lists incur storage costs just like large `structs`. Furthermore, iterating over a large list in your runtime may result in exceeding the block production time. If this occurs for sovereign chains, the blockchain will slow down. If this occurs for parachains, the blockchain will stop producing blocks and stop functioning.

Although wrapping related items in a shared `struct` is an excellent way to reduce the number of storage reads, at some point the size of the object will begin to incur costs that may outweigh the optimization in storage reads. Read about benchmarking to learn how to optimize execution time.

Refer to the Storage Value documentation for a comprehensive list of the methods that Storage Value exposes.

Storage map

Map data structures are ideal for managing sets of items whose elements will be accessed randomly, as opposed to iterating over them sequentially in their entirety. Storage Maps in Substrate are implemented as key-value mappings that provide a similar interface as traditional hash-maps for enabling random lookups. In order to give runtime engineers increased control, Substrate allows developers to select which hashing algorithms suits their use case the best for generating a map's keys. This is covered in the section on hashing algorithms.

Refer to the Storage Map documentation for a comprehensive list of the methods that Storage Map exposes.

Double storage map

Double Storage Maps are very similar to single Storage Maps except they contain two keys, which is useful for querying values with common keys.

N storage map

N Storage Maps are also very similar to its siblings, namely Storage Maps and Double Storage Maps, but with the ability to hold any arbitrary number of keys.

To specify the keys in an N Storage Map in FRAMEv2, a tuple containing the special `NMapKey` struct must be provided as a type to the Key (i.e. second) type parameter while declaring the `StorageNMap`.

Refer to the N Storage Map documentation for more details about the syntaxes in using a N Storage Map.

Iterating over Storage Maps

Substrate Storage Maps are iterable with respect to their keys and values. Because maps are often used to track unbounded sets of data (such as account balances), iterating over them without caution in the runtime may cause blocks not being able to produced in time. Furthermore, because accessing the elements of a map requires more database reads than accessing the elements of a native list, map iterations are significantly *more* costly than list iterations in terms of execution time.

In general, Substrate focuses on programming according to principles and best practices as opposed to hard and fast rules of right and wrong. The information here aims to help you understand *all* of Substrate's storage capabilities and how to use them in a way that respects the principles around which they were designed. For instance, iterating over storage maps in your runtime is neither right nor wrong—yet, avoiding it would be considered a better approach with respect to best practices.

Substrate's Iterable Storage Map interfaces define the following methods:

- `iter()` - enumerate all elements in the map in no particular order. If you alter the map while doing this, you'll get undefined results. See:
  - `IterableStorageMap`
  - `IterableStorageDoubleMap`
  - `IterableStorageNMap`.
- `drain()` - remove all elements from the map and iterate through them in no particular order. If you add elements to the map while doing this, you'll get undefined results. See:
  - `IterableStorageMap`
  - `IterableStorageDoubleMap`
  - `IterableStorageNMap`

- `translate()` - use the provided function to translate all elements of the map, in no particular order. To remove an element from the map, return `None` from the translation function. See:
  - `IterableStorageMap`
  - `IterableStorageDoubleMap`
  - `IterableStorageNMap`

Declaring storage items

Runtime storage items are created with `#[pallet::storage]` in any FRAME-based pallet. Here is an example of declaring the four different types of storage items:

RUST

```
#[pallet::storage]
type SomePrivateValue<T> = StorageValue<_, u32, ValueQuery>;

#[pallet::storage]
#[pallet::getter(fn some_primitive_value)]
pub(super) type SomePrimitiveValue<T> = StorageValue<_, u32, ValueQuer

#[pallet::storage]
pub(super) type SomeComplexValue<T: Config> = StorageValue<_, T::Accou

#[pallet::storage]
#[pallet::getter(fn some_map)]
pub(super) type SomeMap<T: Config> = StorageMap<_, Blake2_128Concat, T

#[pallet::storage]
pub(super) type SomeDoubleMap<T: Config> = StorageDoubleMap<_, Blake2_

#[pallet::storage]
#[pallet::getter(fn some_nmap)]
pub(super) type SomeNMap<T: Config> = StorageNMap<
    _,
    (
        NMapKey<Blake2_128Concat, u32>,
        NMapKey<Blake2_128Concat, T::AccountId>,
        NMapKey<Twox64Concat, u32>,
    ),
    u32,
    ValueQuery,
>;
```

Notice that the map's storage items specify the hashing algorithm that will be used.

QueryKindTrait

The implementation of the `QueryKindTrait` passed to the storage item determines how the storage should be handled when there is no value in storage. With `OptionQuery`, when no value is in storage the `get` method will return `None`. With `ValueQuery`, when no value is in storage the `get` method will return the value configured with the `OnEmpty` generic. For cases with a specific default value to configure, it is recommended to use `ValueQuery`.

Visibility

In the examples above, all the storage items except `SomePrivateValue` are made public by way of the `pub` keyword. Blockchain storage is always publicly visible from *outside* of the runtime; the visibility of Substrate storage items only impacts whether or not other pallets *within* the runtime will be able to access a storage item.

Getter methods

The `#[pallet::getter(..)]` macro provides an optional `get` extension that can be used to implement a getter method for a storage item on the module that contains that storage item; the extension takes the desired name of the getter function as an argument. If you omit this optional extension, you will still be able to access the storage item's value, but you will not be able to do so by way of a getter method implemented on the module; instead, you will need to use the storage item's `get` method.

The optional `getter` extension only impact the way that a storage item can be accessed from *within* Substrate code—you will always be able to query the storage of your runtime to get the value of a storage item.

Here is an example that implements a getter method named `some_value` for a Storage Value named `SomeValue`. This pallet would now have access to a `Self::some_value()` method in addition to the `SomeValue::get()` method:

RUST

```rust
#[pallet::storage]
#[pallet::getter(fn some_value)]
pub(super) type SomeValue = StorageValue<_, u64, ValueQuery>;
```

Default values

Substrate allows you to specify a default value that is returned when a storage item's value is not set. Although the default value does **not** actually occupy runtime storage, the runtime logic will see this value during execution.

Here is an example of specifying a default value in storage:

RUST

```rust
#[pallet::type_value]
pub(super) fn MyDefault<T: Config>() -> T::Balance { 3.into() }
#[pallet::storage]
pub(super) type MyStorageValue<T: Config> =
    StorageValue<Value = T::Balance, QueryKind = ValueQuery, OnEmpty =
```

Notice that for the sake of adding clarity to each storage field, the syntax above is the non-abbreviated version of declaring storage items.

Accessing storage items

Blockchains that are built with Substrate expose a remote procedure call (RPC) server that can be used to query runtime storage. You can use software libraries like Polkadot JS to easily interact with the RPC server from your code and access storage items. The Polkadot JS team also maintains the Polkadot Apps UI, which is a fully-featured web app for interacting with Substrate-based blockchains, including querying storage.

Hashing algorithms

A novel feature of Storage Maps in Substrate is that they allow developers to specify the hashing algorithm that will be used to generate a map's keys. A Rust object that is used to encapsulate hashing logic is referred to as a "hasher". Broadly speaking, the hashers that are available to Substrate developers can be described in two ways: (1) whether or not they are cryptographic; and (2) whether or not they produce a transparent output.

For the sake of completeness, the characteristics of non-transparent hashing algorithms are described below, but keep in mind that any hasher that does not produce a transparent output has been deprecated for FRAME-based blockchains.

Cryptographic hashing algorithms

Cryptographic hashing algorithms enable us to build tools that make it extremely difficult to manipulate the input of a hashing algorithm to influence its output. For example, a cryptographic hashing algorithm would produce a wide distribution of outputs even if the inputs were the numbers 1 through 10. It is critical to use cryptographic hashing algorithms when users are able to influence the keys of a Storage Map. Failure to do so creates an attack vector that makes it easy for malicious actors to degrade the performance of your blockchain network. An example of a map that should use a cryptographic hash algorithm to generate its keys is a map used to track account balances. In this case, it is important to use a cryptographic hashing algorithm so that an attacker cannot bombard your system with many small transfers to sequential account numbers. Without the appropriate cryptographic hashing algorithm this would create an imbalanced storage structure that would suffer in performance. Read more about common hashers in Subsrate in Common Substrate hashers.

Cryptographic hashing algorithms are more complex and resource-intensive than their non-cryptographic counterparts, which is why it is important for runtime engineers to understand their appropriate usages in order to make the best use of the flexibility Substrate provides.

Transparent hashing algorithms

A transparent hashing algorithm is one that makes it easy to discover and verify the input that was used to generate a given output. In Substrate, hashing algorithms are made transparent by concatenating the algorithm's input to its output. This makes it trivial for users to retrieve a key's original unhashed value and verify it if they'd like (by re-hashing it). The creators of Substrate have **deprecated the use of non-transparent hashers** within FRAME-based runtimes, so this information is provided primarily for completeness. In fact, it is *necessary* to use a transparent hashing algorithm if you would like to access iterable map capabilities.

Common Substrate hashers

This table lists some common hashers used in Substrate and denotes those that are cryptographic and those that are transparent:

Hasher	Cryptographic	Transparent
Blake2 128 Concat	X	X
TwoX 64 Concat		X
Identity		X

The Identity hasher encapsulates a hashing algorithm that has an output equal to its input (the identity function). This type of hasher should only be used when the starting key is already a cryptographic hash.

Genesis configuration

Substrate's runtime storage APIs include capabilities to initialize storage items in the genesis block of your blockchain. The genesis storage configuration APIs expose a number of mechanisms for initializing storage, all of which have entry points in `#[pallet::genesis_config]`. The `GenesisConfig` data type is defined under the attribute `#[pallet::genesis_config]` and the attribute `#[pallet::genesis_build]` is used to build the genesis configuration.

To consume a pallet's genesis configuration capabilities, you must include the `Config` element when adding the pallet to your runtime. All the `GenesisConfig` types for the pallets that inform a runtime will be aggregated into a single `GenesisConfig` type for that runtime, which implements the `BuildStorage` trait. For example, in the `node_template_runtime::GenesisConfig` struct, each attribute on this type corresponds to a `GenesisConfig` from the runtime's pallets that has a `Config` element. Ultimately, the runtime's `GenesisConfig` is exposed by way of the `ChainSpec` trait.

For a complete and concrete example of using Substrate's genesis storage configuration capabilities, refer to the genesis configuration for the Society pallet's storage in the chain specification that ships with the Substrate code base. Keep reading for a more detailed descriptions of these capabilities.

genesis_config

The `#[pallet::genesis_config]` macro provides an extension that will add an attribute to the pallet's `GenesisConfig` data type. The value of this attribute will be used as the initial value of the storage item in your chain's genesis block. The `config` extension takes a parameter that will determine the name of the attribute on the `GenesisConfig` data type—this parameter is optional if the `get` extension is provided.

Here is an example that demonstrates using the `config` extension with a Storage Value named `MyVal` to create an attribute named `init_val` on the `GenesisConfig` data type for the Storage Value's pallet. This attribute is then used in an example that demonstrates using the `GenesisConfig` types to set the Storage Value's initial value in your chain's genesis block.

In `my_pallet/src/lib.rs` :

RUST

```rust
#[pallet::genesis_config]
pub struct GenesisConfig<T: Config> {
    pub init_val: u64,
  }
```

In `chain_spec.rs` :

RUST

```rust
GenesisConfig {
    my_pallet: MyPalletConfig {
        init_val: 221u64 + SOME_CONSTANT_VALUE,
    },
}
```

genesis_build

The `#[pallet::genesis_build]` attribute allows you to define how `genesis_configuration` is built within the pallet itself (this gives you access to the pallet's private functions).

Here is an example that demonstrates using `#[pallet::genesis_config]` and `#[pallet::genesis_build]` to set the initial value of a storage item. In this case, the example involves two storage items: one that represents a list of member account IDs and another that designates a special member from the list (the prime member).

In `my_pallet/src/lib.rs` :

RUST

```rust
#[pallet::genesis_config]
struct GenesisConfig {
    members: Vec<T::AccountId>,
```

```
    prime: T::AccountId,
}

#[pallet::genesis_build]
impl<T: Config> GenesisBuild<T> for GenesisConfig {
    fn build(&self) {
        Pallet::<T>::initialize_members(&self.members);
        SomeStorageItem::<T>::put(self.prime);
    }
}
```

In `chain_spec.rs`:

```
RUST

GenesisConfig {
    my_pallet: MyPalletConfig {
        members: LIST_OF_IDS,
        prime: ID,
    },
}
```

You can also use `genesis_build` to define a `GenesisConfig` attribute that is not bound to a particular storage item. This may be desireable if you wish to invoke a private helper function within your pallet that sets several storage items, or invoke a function defined in some other pallets included within your pallet. For example, using an imaginary private function called `intitialize_members`, this would look like:

In `my_pallet/src/lib.rs`:

```
RUST

#[pallet::genesis_config]
struct GenesisConfig {
    members: Vec<T::AccountId>,
    prime: T::AccountId,
}

#[pallet::genesis_build]
impl<T: Config> GenesisBuild<T> for GenesisConfig {
    fn build(&self) {
        Pallet::<T>::initialize_members(&config.members);
        SomeStorageItem::<T>::put(self.prime);
    }
}
```

In `chain_spec.rs`:

```
RUST

GenesisConfig {
    my_pallet: MyPalletConfig {
        members: LIST_OF_IDS,
        prime: ID,
    },
}
```

Best practices

Substrate is designed to provide a flexible framework that allows you to build the blockchain that suits your needs. However, the Substrate codebase adheres to a number of best practices in order to promote the creation of blockchain networks that are secure, performant, and maintainable in

the long-term. The following sections outline best practices for using Substrate storage and also describe the important first principles that motivated them.

What to store

Remember, the fundamental principle of blockchain runtime storage is to minimize its use. Only *consensus-critical* data should be stored in your runtime. When possible, use techniques like hashing to reduce the amount of data you must store. For example, many of Substrate's governance capabilities—such as the Democracy pallet's `propose` function allow network participants to vote on the *hash* of a dispatchable call, which is always bounded in size, as opposed to the call itself, which may be unbounded in length. This is especially true in the case of runtime upgrades where the dispatchable call takes an entire runtime Wasm blob as its parameter. Because these governance mechanisms are implemented *on-chain*, all the information that is needed to come to consensus on the state of a given proposal must also be stored on-chain - this includes *what* is being voted on. However, by binding an on-chain proposal to its hash, Substrate's governance mechanisms allow this to be done in a way that defers bringing all the data associated with a proposal on-chain until *after* it has been approved. This means that storage is not wasted on proposals that fail.

Once a proposal has passed, someone can initiate the actual dispatchable call (including all its parameters), which will be hashed and compared to the hash in the proposal. Another common pattern for using hashes to minimize data that is stored on-chain is to store the pre-image associated with an object in IPFS; this means that only the IPFS location (a hash that is bounded in size) needs to be stored on-chain.

Hashes are only one mechanism that can be used to control the size of runtime storage. An example of another mechanism is bounds.

Verify first, write last

Substrate does not cache state prior to extrinsic dispatch. Instead, it applies changes directly as they are invoked. If an extrinsic fails, any state changes will persist. Because of this, it is important not to make any storage mutations until it is certain that all preconditions have been met. In general, code blocks that may result in mutating storage should be structured as follows:

```rust
{
  // all checks and throwing code go here

  // ** no throwing code below this line **

  // all event emissions & storage writes go here
}
```

Do not use runtime storage to store intermediate or transient data within the context of an operation that is logically atomic or data that will not be needed if the operation is to fail. This does not mean that runtime storage should not be used to track the state of ongoing actions that require multiple atomic operations, as in the case of the multi-signature capabilities from the Utility pallet. In this case, runtime storage is used to track the signatories on a dispatchable call even though a given call may never receive enough signatures to actually be invoked. In this case, each signature is considered an atomic event in the ongoing multi-signature operation; the data needed to record a single signature is not stored until after all the preconditions associated with that signature have been met.

Create bounds

Creating bounds on the size of storage items is an extremely effective way to control the use of runtime storage and one that is used repeatedly throughout the Substrate codebase. In general, any storage item whose size is determined by user action should have a bound on it. The multi-

signature capabilities from the Multisig pallet that were described above are one such example. In this case, the list of signatories associated with a multi-signature operation is provided by the multi-signature participants. Because this signatory list is necessary to come to consensus on the state of the multi-signature operation, it must be stored in the runtime. However, in order to give runtime developers control over how much space in storage these lists may occupy, the Utility pallet requires users to configure a bound on this number that will be included as a precondition before anything is written to storage.

Where to go next

Check out some guides covering various topics on storage:

- Mint basic tokens using a storage map.
- How-to: Create a storage structure
- StorageValue
- StorageMap
- StorageDoubleMap
- StorageNMap

Previous Next

Rococo Network Sq

Privacy Policy

Terms of Services