# Getting Started with Substrate

**Gautam Dhameja**
Solutions Architect @ Parity
@gautamdhameja

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

## What is Substrate?

Substrate is an **open source**, **modular**, and **extensible** framework for building blockchains.

parity

## What is Substrate?

**Substrate provides all the core components of a Blockchain:**

- Database Layer
- Networking Layer
- Consensus Engine
- Transaction Queue
- Library of Runtime Modules

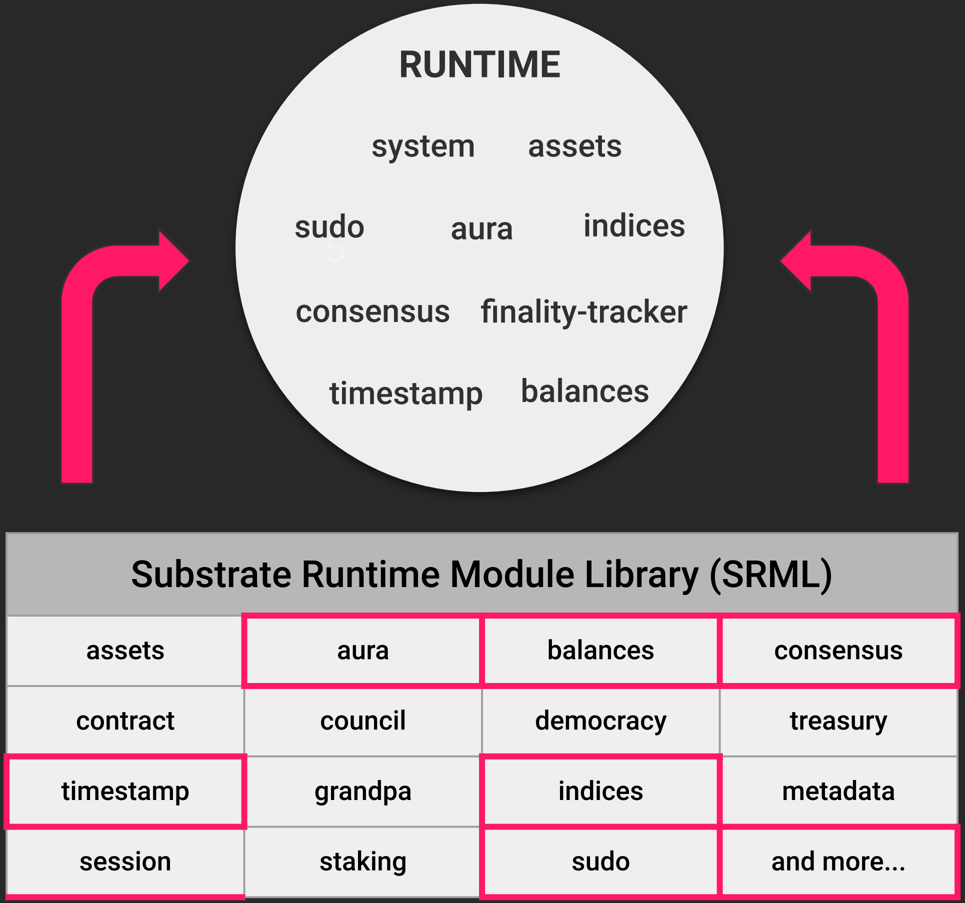
**Each of which can be customized and extended.**

parity

## What is a Runtime?

The runtime is the **block execution logic** of the blockchain, i.e. the State Transition Function.

It is composed of **Runtime Modules**.



| Substrate Runtime Module Library (SRML) | | | |
|---|---|---|---|
| assets | aura | balances | consensus |
| contract | council | democracy | treasury |
| timestamp | grandpa | indices | metadata |
| session | staking | sudo | and more... |

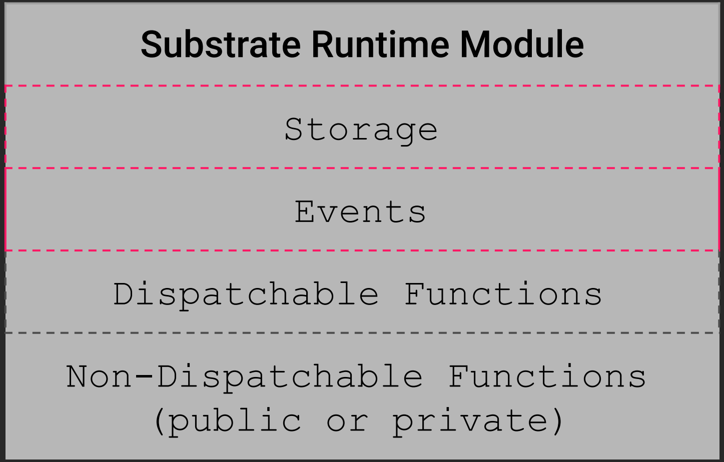
parity

# Runtime Module

A runtime module packages together

- Functions (dispatchable, public or private, mutable or immutable)
- Storage items
- Events

to support a certain set of features for a runtime.



Eg: The **Assets** Module in SRML is for creating and managing fungible assets.

## Setup and Installation

## Installation

Install dependencies + Substrate node:

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

Install only dependencies:

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

## Bootstrapping the runtime

Create a new template runtime:

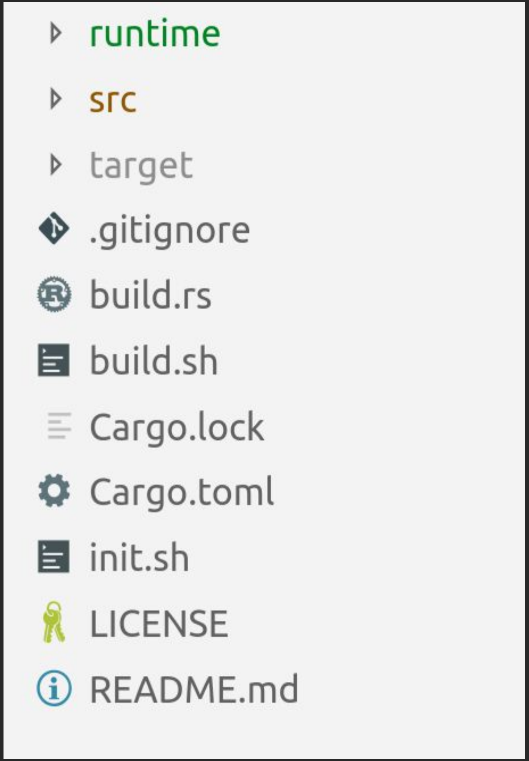
```
substrate-node-new
```

- Downloads the *substrate-node-template* codebase
- Compiles it for Wasm and Native environments
- Provides a hack ready Substrate node

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

$ cd substrate-package

$ ./substrate-package-rename.sh substratekitties <your_name>

$ cd substratekitties

$ ./init.sh
```

## Substrate Node Template

- A working substrate node
- Includes from SRML
  - Accounts, Balances, Fees, Runtime Upgrades, and more...
- Easily add and remove modules from the SRML
- Create your own modules to customize your chain functionality



```
runtime
src
target
.gitignore
build.rs
build.sh
Cargo.lock
Cargo.toml
init.sh
LICENSE
README.md
```

parity

```
$ ./build.sh
```

## Developing a Runtime Module

## Skeleton of a Module

```
use support::{decl_module, decl_storage, decl_event,...};
pub trait Trait: system::Trait {...}


decl_storage! {...} // storage
decl_event! {...}   // events
decl_module! {...}  // dispatchable calls


impl<T: Trait> Module<T> {...} // non-dispatchable functions
```

parity

## Macros

`decl_storage!` `decl_module!` `decl_event!`

- Rust code which can generate more code
- Used to simplify the creation of modules
- Generate types and traits used by the runtime

parity

## Designing the runtime - Storage

- On-chain or not?
- Simple data structures
- Resource efficient state changes
- Complex data structures lead to complex logic

## Declaring Storage

```
decl_storage! {
    trait Store for Module<T: Trait> as TemplateModule {
        // Here we are declaring a StorageValue, `SomeValue` as a u32
        // `get(some_value)` defines a getter function
        // Getter called with `Self::some_value()`
        SomeValue get(some_value): u32;
        // Here we are declaring a StorageMap from an AccountId to a Hash
        // Getter called with `Self::some_map(account_id)`
        SomeMap get(some_map): map T::AccountId => u32;
    }
}
```

parity

## Designing the runtime - Events

- No success return values
- Communicate state changes
- Business events vs. System events

## Declaring Events

```
decl_event!(
    pub enum Event<T>
    where
        <T as system::Trait>::AccountId
    {
        // Event `ValueStored` deposits values of type `AccountId` and `u32`
        ValueStored(AccountId, u32),
    }
);
```

parity

## Implementing the runtime logic

- **Validate** - check all conditions on input
- **Update** - write to storage
- **Communicate** - emit events
- Ok(())

parity

## Declaring Dispatchable Functions

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

    pub fn store_value(origin, input: u32) -> Result {
      let sender = ensure_signed(origin)?; // Check for transaction
      <SomeMap<T>>::insert(sender, input); // Insert key/value in StorageMap
      Self::deposit_event(RawEvent::ValueStored(sender, input)); // Emit
Event
      Ok(()) // Return Ok at the end of a function
}}}
```

## Declaring Public and Private Functions

```
impl<T: Trait> Module<T> {
  fn mint(to: T::AccountId, id: T::Hash) -> Result { }
  pub fn transfer(from: T::AccountId, to: T::AccountId, id: T::Hash)
  -> Result { }
}
```

These can also be called from other modules if marked public.

parity

```
$ cargo build --release
```

## Best Practices

## Best Practices

- Never panic!
  - Handle errors gracefully.
- Verify first; commit last
  - There is no revert like in smart contracts.
- Resources used = Price paid
  - Optimize storage and logic.

# Handling Errors in Your Runtime

- Your Runtime should never panic:
  - An unrecoverable error in Rust, which immediately terminates the thread
- Instead, you must perform “safe” operations which explicitly handles errors
- For example, safe math:

```
// BAD
let a = u8::max_value() + 1; // What should Rust do?
// GOOD
let a = u8::max_value().checked_add(1).ok_or("Overflow!");
```

parity

## Option Instead of Null

Options let you be explicit about variables having some or no value

```
// Definition of Option type
enum Option<T> {
    Some(T),
    None,
}
```

```
let a = u8::max_value().checked_add(1)
a == None // True
let b = u8::max_value().checked_sub(1)
b == Some(254) // True
```

parity

## Result Instead of Panic

Result is a richer version of Option that describes possible error instead of possible absence.

```
// Definition of Result type
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```
// Result in Substrate found in support::dispatch::Result
pub type Result = result::Result<(), &'static str>;
```

parity

## Verify First, Write Last

- A “bad transaction” does not work the same as Ethereum
- Ethereum: State is reverted, storage is untouched, and a fee is paid
- Substrate: State changes will persist if an `Err` is returned
- Needed for situations like:
  - Increasing Account transaction nonce, even with failed transactions
  - Charging transaction fees even when “out of gas”
- Need to be conscious of this pattern when making “sub-functions”

## Resources

parity

Kitty by David Revoy

## Substrate Collectables Workshop

- Run a local Substrate node
- Learn about runtime development and best practices
- Build a working chain with UI
- Minimal Rust Experience

**tiny.cc/substrate-workshop**

parity

## Next Steps For You!

- ✓ Clone and follow instruction from the Substrate Package
  - ➢ tiny.cc/substrate-package
- ✓ Join and ask questions in the Substrate Technical channel on Riot
  - ➢ tiny.cc/substrate-technical
- ✓ Explore and read the Substrate Runtime Module Library
  - ➢ tiny.cc/substrate-srml
- ✓ BUILD ON SUBSTRATE!

## Questions?

tiny.cc/substrate-technical

parity