Staking in depth: Anatomy and Extensions

Hyungsuk Kang

- Former Blockchain research engineer in Samsung Research America
- Ethereum Devcon IV Scholar
- Former Software Engineer at Terra(cosmwasm)
- Tendermint Fellow
- Lead developer in Plasm
- Interests - IoT, distributed system, consensus algorithms
- Publication:
  - Secure DNS name autoconfiguration for IPv6 internet-of-things devices

Brief Introduction

1. Anatomy of Staking Module
2. Possible mutations
3. Staking module set
4. Node configurations

Table of Contents

Staking Module Anatomy

how to eat elephant one bite at at time

Module parts

Configuration Trait

Storage

Module

- extrinsics

- root extrinsics

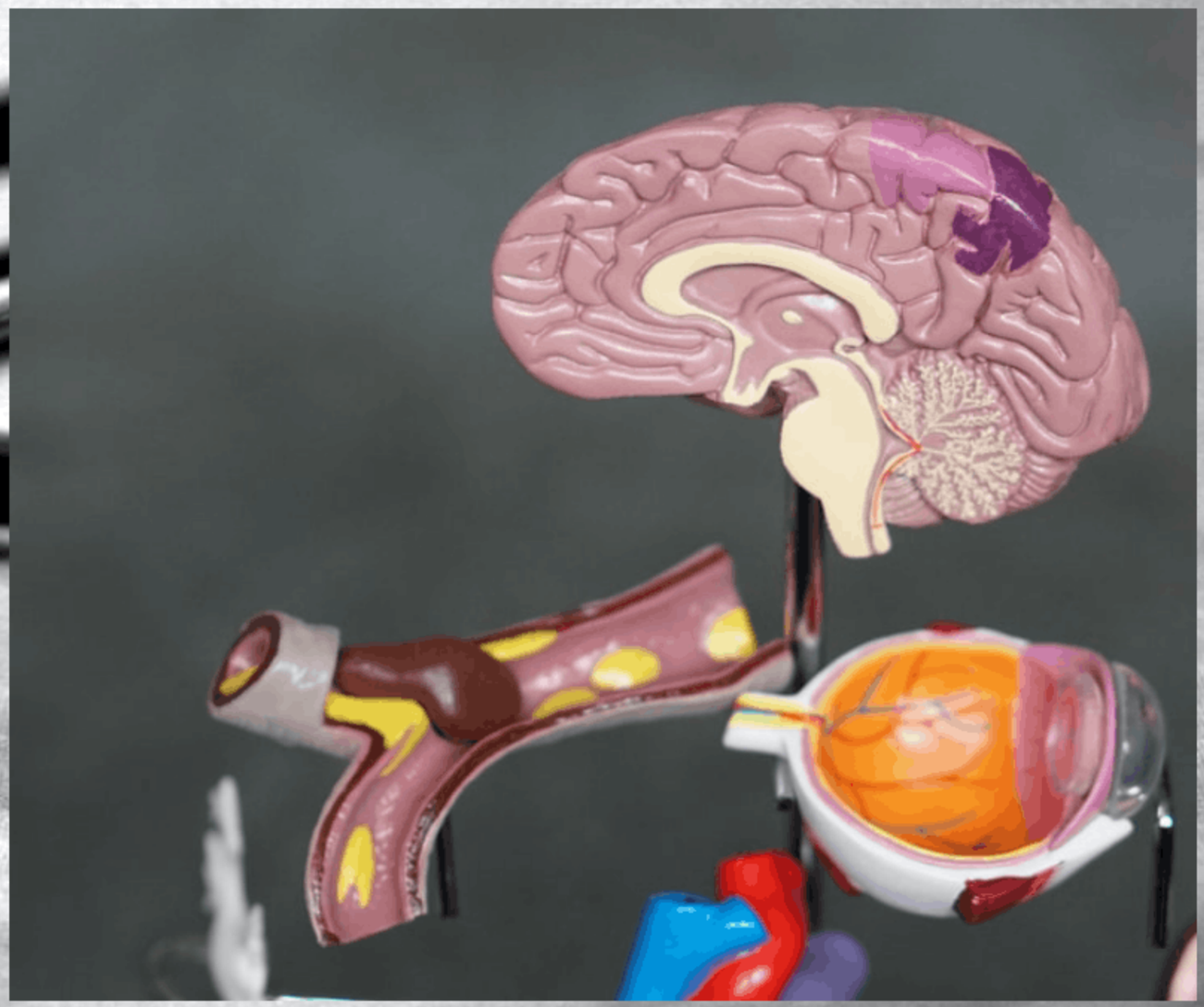
- hooks

- module methods

Event

Error

Dependancy Injection trait



Configuration Trait

- A Trait where primitives from the runtime is imported to each runtime module
- Includes module functions

```rust
pub trait Trait: frame_system::Trait + SendTransactionTypes<Call<Self>> {
    /// The staking balance.
    type Currency: LockableCurrency<Self::AccountId, Moment=Self::BlockNumber>;

    /// Time used for computing era duration.
    ///
    /// It is guaranteed to start being called from the first `on_finalize`. Thus value at genesis
    /// is not used.
    type UnixTime: UnixTime;

    /// Convert a balance into a number used for election calculation. This must fit into a `u64`
    /// but is allowed to be sensibly lossy. The `u64` is used to communicate with the
    /// [`sp_npos_elections`] crate which accepts u64 numbers and does operations in 128.
    /// Consequently, the backward convert is used convert the u128s from sp-elections back to a
    /// [`BalanceOf`].
    type CurrencyToVote: Convert<BalanceOf<Self>, VoteWeight> + Convert<u128, BalanceOf<Self>>;

    /// Tokens have been minted and are unused for validator-reward.
    /// See [Era payout](./index.html#era-payout).
    type RewardRemainder: OnUnbalanced<NegativeImbalanceOf<Self>>;

    /// The overarching event type.
    type Event: From<Event<Self>> + Into<<Self as frame_system::Trait>::Event>;

    /// Handler for the unbalanced reduction when slashing a staker.
    type Slash: OnUnbalanced<NegativeImbalanceOf<Self>>;

    /// Handler for the unbalanced increment when rewarding a staker.
    type Reward: OnUnbalanced<PositiveImbalanceOf<Self>>;

    /// Number of sessions per era.
    type SessionsPerEra: Get<SessionIndex>;

    /// Number of eras that staked funds must remain bonded for.
    type BondingDuration: Get<EraIndex>;

    /// Number of eras that slashes are deferred by, after computation.
    ///
    /// This should be less than the bonding duration. Set to 0 if slashes
    /// should be applied immediately, without opportunity for intervention.
    type SlashDeferDuration: Get<EraIndex>;

    /// The origin which can cancel a deferred slash. Root can always do this.
    type SlashCancelOrigin: EnsureOrigin<Self::Origin>;

    /// Interface for interacting with a session module.
    type SessionInterface: self::SessionInterface<Self::AccountId>;

    /// The NPoS reward curve used to define yearly inflation.
    /// See [Era payout](./index.html#era-payout).
    type RewardCurve: Get<&'static PiecewiseLinear<'static>>;

    /// Something that can estimate the next session change, accurately or as a best effort guess.
    type NextNewSession: EstimateNextNewSession<Self::BlockNumber>;

    /// The number of blocks before the end of the era from which election submissions are allowed.
    ///
    /// Setting this to zero will disable the offchain compute and only on-chain seq-phragmen will
    /// be used.
```

Storage

- A key-value interface storage for blockchain state
- Trie is managed automatically
- getter is provided

```
decl_storage! {
    trait Store for Module<T: Trait> as Staking {
        /// Number of eras to keep in history.
        ///
        /// Information is kept for eras in `[current_era - history_depth; current_era]`.
        ///
        /// Must be more than the number of eras delayed by session otherwise. I.e. active era must
        /// always be in history. I.e. `active_era > current_era - history_depth` must be
        /// guaranteed.
        HistoryDepth get(fn history_depth) config(): u32 = 84;

        /// The ideal number of staking participants.
        pub ValidatorCount get(fn validator_count) config(): u32;

        /// Minimum number of staking participants before emergency conditions are imposed.
        pub MinimumValidatorCount get(fn minimum_validator_count) config(): u32;

        /// Any validators that may never be slashed or forcibly kicked. It's a Vec since they're
        /// easy to initialize and the performance hit is minimal (we expect no more than four
        /// invulnerables) and restricted to testnets.
        pub Invulnerables get(fn invulnerables) config(): Vec<T::AccountId>;

        /// Map from all locked "stash" accounts to the controller account.
        pub Bonded get(fn bonded): map hasher(twox_64_concat) T::AccountId => Option<T::AccountId>;

        /// Map from all (unlocked) "controller" accounts to the info regarding the staking.
        pub Ledger get(fn ledger):
            map hasher(blake2_128_concat) T::AccountId
            => Option<StakingLedger<T::AccountId, BalanceOf<T>>>;

        /// Where the reward payment should be made. Keyed by stash.
        pub Payee get(fn payee): map hasher(twox_64_concat) T::AccountId => RewardDestination<T::AccountId>;

        /// The map from (wannabe) validator stash key to the preferences of that validator.
        pub Validators get(fn validators):
            map hasher(twox_64_concat) T::AccountId => ValidatorPrefs;

        /// The map from nominator stash key to the set of stash keys of all validators to nominate.
        pub Nominators get(fn nominators):
            map hasher(twox_64_concat) T::AccountId => Option<Nominations<T::AccountId>>;

        /// The current era index.
        ///
        /// This is the latest planned era, depending on how the Session pallet queues the validator
        /// set, it might be active or not.
        pub CurrentEra get(fn current_era): Option<EraIndex>;

        /// The active era information, it holds index and start.
        ///
        /// The active era is the era currently rewarded.
        /// Validator set of this era must be equal to `SessionInterface::validators`.
        pub ActiveEra get(fn active_era): Option<ActiveEraInfo>;

        /// The session index at which the era start for the last `HISTORY_DEPTH` eras.
```

Module

- The main body of algorithms in each module
- Functions are divided into 3 kinds
  - Extrinsics in `decl_module`
  - Root extrinsics in `decl_module`
  - hooks(on_initialize, on_finalize)
  - Module methods

```
decl_module! {
    pub struct Module<T: Trait> for enum Call where origin: T::Origin {
        /// Number of sessions per era.
        const SessionsPerEra: SessionIndex = T::SessionsPerEra::get();
```

```
impl<T: Trait> Module<T> {
    /// The total balance that can be slashed from a stash account as of right now.
    pub fn slashable_balance_of(stash: &T::AccountId) -> BalanceOf<T> {
        // Weight note: consider making the stake accessible through stash.
        Self::bonded(stash).and_then(Self::ledger).map(|l| l.active).unwrap_or_default()
    }

    /// internal impl of [`slashable_balance_of`] that returns [`VoteWeight`].
    fn slashable_balance_of_vote_weight(stash: &T::AccountId) -> VoteWeight {
        <T::CurrencyToVote as Convert<BalanceOf<T>, VoteWeight>>::convert(
            Self::slashable_balance_of(stash)
        )
    }
```

Extrinsics(Module)

- General functions that can be executed from polkadot.js
- Executed via api.tx.{module}.method(arguments, {callback}) after websocket connection

```rust
/// (Re-)set the controller of a stash.
///
/// Effects will be felt at the beginning of the next era.
///
/// The dispatch origin for this call must be _Signed_ by the stash, not the controller.
///
/// # <weight>
/// - Independent of the arguments. Insignificant complexity.
/// - Contains a limited number of reads.
/// - Writes are limited to the `origin` account key.
/// ----------
/// Weight: O(1)
/// DB Weight:
/// - Read: Bonded, Ledger New Controller, Ledger Old Controller
/// - Write: Bonded, Ledger New Controller, Ledger Old Controller
/// # </weight>
#[weight = T::WeightInfo::set_controller()]
fn set_controller(origin, controller: <T::Lookup as StaticLookup>::Source) {
    let stash = ensure_signed(origin)?;
    let old_controller = Self::bonded(&stash).ok_or(Error::<T>::NotStash)?;
    let controller = T::Lookup::lookup(controller)?;
    if <Ledger<T>>::contains_key(&controller) {
        Err(Error::<T>::AlreadyPaired)?
    }
    if controller != old_controller {
        <Bonded<T>>::insert(&stash, &controller);
        if let Some(l) = <Ledger<T>>::take(&old_controller) {
            <Ledger<T>>::insert(&controller, l);
        }
    }
}
```

Root Extrinsics(Module)

- Requires `ensure_root` macro to check whether this come from root or guard to be executed through democracy module

```rust
/// (Re-)set the controller of a stash.
///
/// Effects will be felt at the beginning of the next era.
///
/// The dispatch origin for this call must be _Signed_ by the stash, not the controller.
///
/// # <weight>
/// - Independent of the arguments. Insignificant complexity.
/// - Contains a limited number of reads.
/// - Writes are limited to the `origin` account key.
/// ----------
/// Weight: O(1)
/// DB Weight:
/// - Read: Bonded, Ledger New Controller, Ledger Old Controller
/// - Write: Bonded, Ledger New Controller, Ledger Old Controller
/// # </weight>
#[weight = T::WeightInfo::set_controller()]
fn set_controller(origin, controller: <T::Lookup as StaticLookup>::Source) {
    let stash = ensure_signed(origin)?;
    let old_controller = Self::bonded(&stash).ok_or(Error::<T>::NotStash)?;
    let controller = T::Lookup::lookup(controller)?;
    if <Ledger<T>>::contains_key(&controller) {
        Err(Error::<T>::AlreadyPaired)?
    }
    if controller != old_controller {
        <Bonded<T>>::insert(&stash, &controller);
        if let Some(l) = <Ledger<T>>::take(&old_controller) {
            <Ledger<T>>::insert(&controller, l);
        }
    }
}
```

Hooks

- Repetitive task that can be executed in each module in every block
- on_finalize: executes right after the last extrinsic is executed from a block
- on_initialize: executes right before the first extrinsic is executed from a block

Module Methods

- Methods for repeating operations within the module
- Used for simulating test cases regarding sessions

```rust
impl<T: Trait> Module<T> {
    /// The total balance that can be slashed from a stash account as of right now.
    pub fn slashable_balance_of(stash: &T::AccountId) -> BalanceOf<T> {
        // Weight note: consider making the stake accessible through stash.
        Self::bonded(stash).and_then(Self::ledger).map(|l| l.active).unwrap_or_default()
    }

    /// internal impl of [`slashable_balance_of`] that returns [`VoteWeight`].
    fn slashable_balance_of_vote_weight(stash: &T::AccountId) -> VoteWeight {
        <T::CurrencyToVote as Convert<BalanceOf<T>, VoteWeight>>::convert(
            Self::slashable_balance_of(stash)
        )
    }
```

Event & Error

- Events are Events that are returned in executing extrinsics

```
decl_event!(
    pub enum Event<T> where Balance = BalanceOf<T>, <T as frame_system::Trait>::AccountId {
        /// The era payout has been set; the first balance is the validator-payout; the second is
        /// the remainder from the maximum amount of reward.
        /// \[era_index, validator_payout, remainder\]
        EraPayout(EraIndex, Balance, Balance),
        /// The staker has been rewarded by this amount. \[stash, amount\]
        Reward(AccountId, Balance),
        /// One validator (and its nominators) has been slashed by the given amount.
        /// \[validator, amount\]
        Slash(AccountId, Balance),
        /// An old slashing report from a prior era was discarded because it could
        /// not be processed. \[session_index\]
        OldSlashingReportDiscarded(SessionIndex),
        /// A new set of stakers was elected with the given \[compute\].
        StakingElection(ElectionCompute),
        /// A new solution for the upcoming election has been stored. \[compute\]
        SolutionStored(ElectionCompute),
        /// An account has bonded this amount. \[stash, amount\]
        ///
        /// NOTE: This event is only emitted when funds are bonded via a dispatchable. Notably,
        /// it will not be emitted for staking rewards when they are added to stake.
        Bonded(AccountId, Balance),
        /// An account has unbonded this amount. \[stash, amount\]
        Unbonded(AccountId, Balance),
        /// An account has called `withdraw_unbonded` and removed unbonding chunks worth `Balance`
        /// from the unlocking queue. \[stash, amount\]
        Withdrawn(AccountId, Balance),
    }
);
```

- Errors are the Error type to describe errors without generic string

```
decl_error! {
    /// Error for the staking module.
    pub enum Error for Module<T: Trait> {
        /// Not a controller account.
        NotController,
        /// Not a stash account.
        NotStash,
        /// Stash is already bonded.
        AlreadyBonded,
        /// Controller is already paired.
        AlreadyPaired,
        /// Targets cannot be empty.
        EmptyTargets,
        /// Duplicate index.
        DuplicateIndex,
```

Dependency Injection Trait

- Used to reduce ambiguation on module's trait due to Rust's trait inheritance
- Declare public trait from inside the module or aggregated support package
- implement the trait in a module and set dependency as the implemented module in others

```rust
/// A currency whose accounts can have liquidity restrictions.
pub trait LockableCurrency<AccountId>: Currency<AccountId> {
    /// The quantity used to denote time; usually just a `BlockNumber`.
    type Moment;

    /// The maximum number of locks a user should have on their account.
    type MaxLocks: Get<u32>;

    /// Create a new balance lock on account `who`.
    ///
    /// If the new lock is valid (i.e. not already expired), it will push the struct to
    /// the `Locks` vec in storage. Note that you can lock more funds than a user has.
    ///
    /// If the lock `id` already exists, this will update it.
    fn set_lock(
        id: LockIdentifier,
        who: &AccountId,
        amount: Self::Balance,
        reasons: WithdrawReasons,
    );

    /// Changes a balance lock (selected by `id`) so that it becomes less liquid in all
    /// parameters or creates a new one if it does not exist.
    ///
    /// Calling `extend_lock` on an existing lock `id` differs from `set_lock` in that it
    /// applies the most severe constraints of the two, while `set_lock` replaces the lock
    /// with the new parameters. As in, `extend_lock` will set:
    /// - maximum `amount`
    /// - bitwise mask of all `reasons`
```

```rust
impl<T: Config<I>, I: 'static> LockableCurrency<T::AccountId> for Pallet<T, I>
where
    T::Balance: MaybeSerializeDeserialize + Debug
{
    type Moment = T::BlockNumber;

    type MaxLocks = T::MaxLocks;

    // Set a lock on the balance of `who`.
    // Is a no-op if lock amount is zero or `reasons` `is_none()`.
    fn set_lock(
        id: LockIdentifier,
        who: &T::AccountId,
        amount: T::Balance,
        reasons: WithdrawReasons,
    ) {
```

```rust
impl pallet_staking::Config for Runtime {
    type Currency = Balances;
    type UnixTime = Timestamp;
    type CurrencyToVote = U128CurrencyToVote;
    type RewardRemainder = Treasury;
    type Event = Event;
    type Slash = Treasury; // send the slashed funds to the treasury.
    type Reward = (); // rewards are minted from the void
    type SessionsPerEra = SessionsPerEra;
    type BondingDuration = BondingDuration;
    type SlashDeferDuration = SlashDeferDuration;
    /// A super-majority of the council can cancel the slash.
    type SlashCancelOrigin = EnsureOneOf<
        AccountId,
        EnsureRoot<AccountId>,
        pallet_collective::EnsureProportionAtLeast< 3,  4,  AccountId,  CouncilCollective>
```

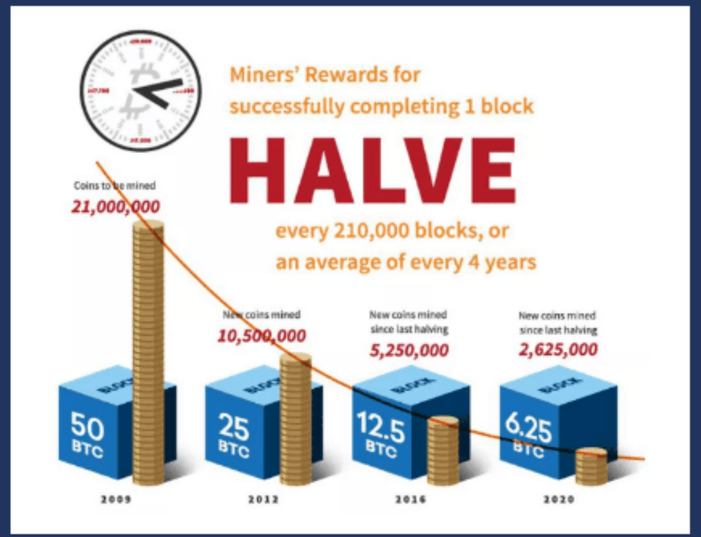
```rust
pub trait Config: frame_system::Config + SendTransactionTypes<Call<Self>> {
    /// The staking balance.
    type Currency: LockableCurrency<Self::AccountId, Moment = Self::BlockNumber>;
```

Possible Mutations

What new chains can come out with this?

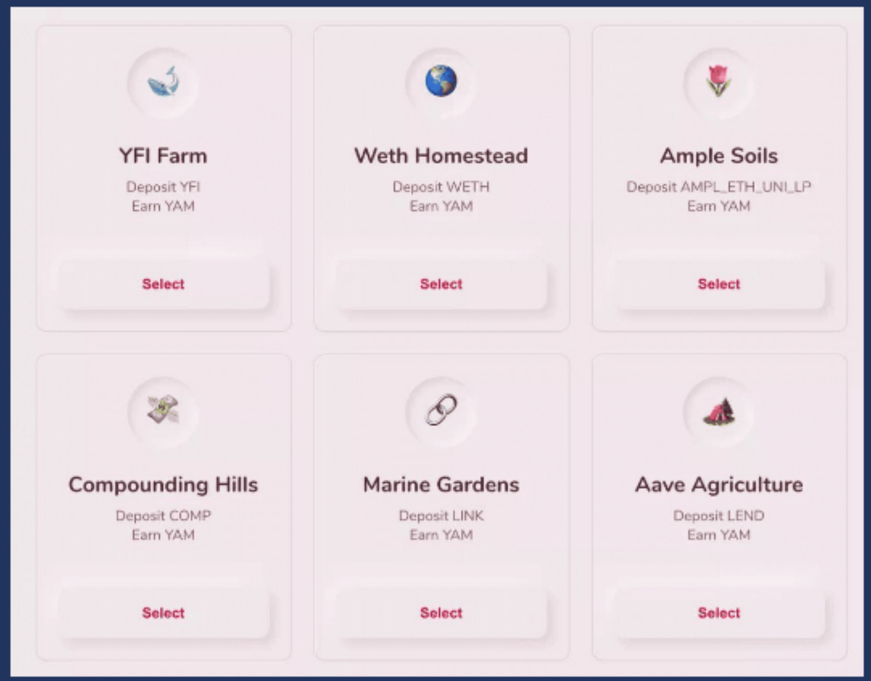
New inflation logic

- Inside inflation.rs file, there is a logic which manages the total reward minted in and each era.
- Make it like Bitcoin where blockchain does not make total reward when the total supply exceeds a certain number and record only imbalances for the total reward
- Let inflation governed by democracy(Something that FED cannnot do)



Add new way to use block rewards

- Turns out the block reward can be split
- Reward is given to dapps in plasm's case
- Try to come up with your own slashing/reward logic in each era



Staking module sets

What other modules are needed to report?

Reporting modules

- Babe: Equivocation report for BABE
- Grandpa: Equivocation report for Grandpa
- ImOnline: Validator reporting that it is online in each era

Tracking modules

- Authorship: Records author in each block
- Offences: Module for recording generic offence cases in a blockchain for evidence
- AuthorityDiscovery: Retrieve current authority set for generating a block
- Session: manages session key for validator node and checks if a session has passed

Treasury Module

- Treasury: Collects slashed balances here and use it for good
- Collective: Council management module
- Elections: Gateway for becoming a council to manage treasury and cancel slashing

Node Configuration

Here comes the hardest part

You need to change

- Your runtime dependancies
- Your node api dependancies and logics in service.rs and api.rs
- Probably your workspaces


Codes to look at:
paritytech/substrate's api and runtime lib.rs file
or here with substrate node template

Let's draw!

- Get 30 mins to imagine your token economics
- Share each other on how you are going to implement it
- Show your code in discord when you make it, no pressure

Hyungsuk Kang
Stake Technologies
Software Engineer
lead dev @Plasm

Twitter
@hskang0525

Email
hskang9@gmail.com

or

hyungsuk@stake.co.jp