# Sketches of IPLD example apps & their data structures

Compiled 2018–06–27

## 0 Introduction

This document is compiled from a selection of five examples that were brainstormed during a session in Lisbon on 2018-05-13, linked from the bottom of this Google doc. Each sketch fits on its own page. I (`@davidad`) have edited them somewhat, mostly to harmonize variations in the choice of syntax[1].

Potential uses of this document include:

- as a ballpark sketch of the motivation and scope of IPLD's medium-term ambitions
- as an inspiration for generating new open questions ("wait, how would *that* work?")
- to provide a grounding, concrete reference point for consideration of future IPLD language features

Note that the final example, IPFS Directories, involves `traits` (also known as interfaces or typeclasses), which may be a post-L0 feature.

[1] This is to facilitate comparisons across the different examples; the decisions of what kinds of syntax to use here have not been thought through carefully and are not intended as proposals about the default surface-syntax of L0.

# 1 Filecoin (orig. `@dignifiedquire`)

```
type Block = struct {
    // reference to the previous block
    // only `None`, for the genesis block
    parent: Option<Cid<Block>>,
    // hash of the statetree after execution of all included messages
    state: Cid<StateTree>,
    // a list of all state transitions this block entails
    timestamp: DateTime,
    messages: Cid<Vec<Message>>,
    // a list of all receipts, for the messages included in this block
    message_receipts: Cid<Vec<MessageReceipt>>,
};

impl Block {
    fn Hash() -> Hash {}
};

type Message = struct {
    // None if this message is not signed
    signature: Option<Hash>,
    // what is the Method type, would be great to tie in the exports with IPLD, not clear how
    method: Method,
    args: Vec<u8>,
    to: Address,
    from: Address,
};

type MessageReceipt = struct {
    exit_code: u8,
    result_ptr: u32,
    result_size: u32,
};

type Address = Vec<u8>;
type Hash = Vec<u8>;

type StateTree = Map<Address, Actor>;

type Actor = struct {
    balance: u64,
    // how is storage modeled?
    // could be just Vec<u8>
    storage: Cid<Vec<u8>>,
    public_key: Option<Vec<u8>>,
};

// How to model concrete actors vs the type actor that represents an actor on the host level?
impl Actor {
    // is this right? how to do this?
    fn Exports(&self) -> Vec<Method>;

    // public exports go here, but they are not part of the actor
    // need to mention actors' types, i.e. IPLD about IPLD types. not sure how to express this
};
```

## 2 Decentralized Twitter

```
type User = struct {
  id: PubKey,
  handle: Utf8String,
  displayName: Utf8String,
  bio: BoundedArray<Utf8Char, 140>,
  profilePic: Hash<Image>,
  coverPic: Hash<Image>,
  tweet_HEAD: LatestSigned<TweetEntry, id>,
  like_HEAD: LatestSigned<LikeEntry, id>,
};

type TweetEntry = signed struct {
  parents: List<Hash<TweetEntry>>, // previous commits from this user
  timestamp: DateTime,
  tweet: enum Tweet {
    PlainTweet: struct {
      text: BoundedArray<Utf8Char, 280>,
    },
    ReplyTweet: struct {
      inReplyTo: List<Hash<TweetEntry>>,
      text: BoundedArray<Utf8Char, 280>,
    },
    PlainRetweet: struct {
      retweetOf: Hash<TweetEntry>,
    },
    RetweetWithComment: struct {
      retweetOf: Hash<TweetEntry>,
      text: BoundedArray<Utf8Char, 280>,
    },
  },
};

type LikeEntry = signed struct {
  parents: List<Hash<LikeEntry>>, // previous commits from this user
  likeOf: Hash<TweetEntry>,
};
```

# 3 IPRS (orig. @diasdavid)

```
type SignedRecord = struct {
  record: Record,              // record or link to record
  Signature: Vec<u8>,          // signature of the record (Value + Scheme + ValidityData)
  PublicKey: PublicKey,
};

type Record = struct {
  Value:    []byte,
  Scheme:   Cid,               // scheme to validate the record (expiry date, geographic location, etc)
  ValidityData: ValidityData,
};

type ValidityData = struct {
  ExpiryDate: DateTime,
  GeoLocation: XYCoordinates,  // XYCoordinates and range
  Ancestry: Cid,               // only valid in the presence of another record
  SeqNumber: u32,
};

type Identity = struct {
  SecretKey: SecretKey,
  PublicKey: PublicKey,
};

type SecretKey = Vec<u8>;     // possibly more than an array in the future (i.e Linked-Data Key / MultiKey)
type PublicKey = Vec<u8>;
```

# 4 Collaborative Real-Time Applications (orig. `@diasdavid`)

```
type User = struct {
  KeyPair: KeyPair,
  Details: Person, // http://schema.org/Person
};

type Capability = struct {    // granting access to a private resource through symmetric crypto
  CypherData: CID,
  AccessKey: SharedKey,
};

type Authorization = struct   // granting other users permission or self proving permission
  IssuerSignature: Vec<u8>,
  Receiver: Publickey,
};

type KeyPair = struct {
  SecretKey: SecretKey,
  PublicKey: PublicKey,
};

type SecretKey = Vec<u8>;     // possibly more than an array in the future (i.e Linked-Data Key / MultiKey)
type PublicKey = Vec<u8>;
type SharedKey = Vec<u8>;     // symmetricKey
```

# 5 IPFS Directories (orig. `@stebalien`)

```
trait Iterable {
    type Item;
    fn map<T>(self, cb: fn(item: &Self::Item) -> T) -> Iterable<Item = T>;
    fn skip(self, n: NonNegative) -> Iterable<Item = Self::Item>;
    fn take(self, n: NonNegative) -> Iterable<Item = Self::Item>;
};

trait Collection: Iterable {
    fn length(&self) -> NonNegative;
};

struct Range {
    start: NonNegative,
    length: NonNegative,
};

trait IndexedCollection: Collection + Map<Key=NonNegative, Value=Self::Item> {
    fn slice(self, range: Range) -> Option<IndexedCollection>;
    fn splice(self, range: Range, Collection<Item=Self::Item>) -> Option<Collection<Item=Self::Item>>
};

trait Map {
  type Key;
  type Value;

  fn get(&self, name: Self::Key) -> Option<Self::Value>;
  fn update(self, name: Self::Key, value: Self::Value) -> Option<Map<Key=Self::Key, Value=Self::Value>>;
};

trait FsObject {
    fn created_at(&self) -> Time;
    fn modified_at(&self) -> Time;
};

// Directories are collections of file objects.
//
// To *use* the file objects, you'll have to *cast* to a Directory, File, etc.
trait Directory:
    Collection<Item = Cid<FsObject>> + Map<Key = String, Value = Cid<FsObject>> + FsObject
{
};
trait RegularFile: FsObject + Collection<Item = u8> {};
```