Syng: An unopinionated system to sync hierarchical data between devices

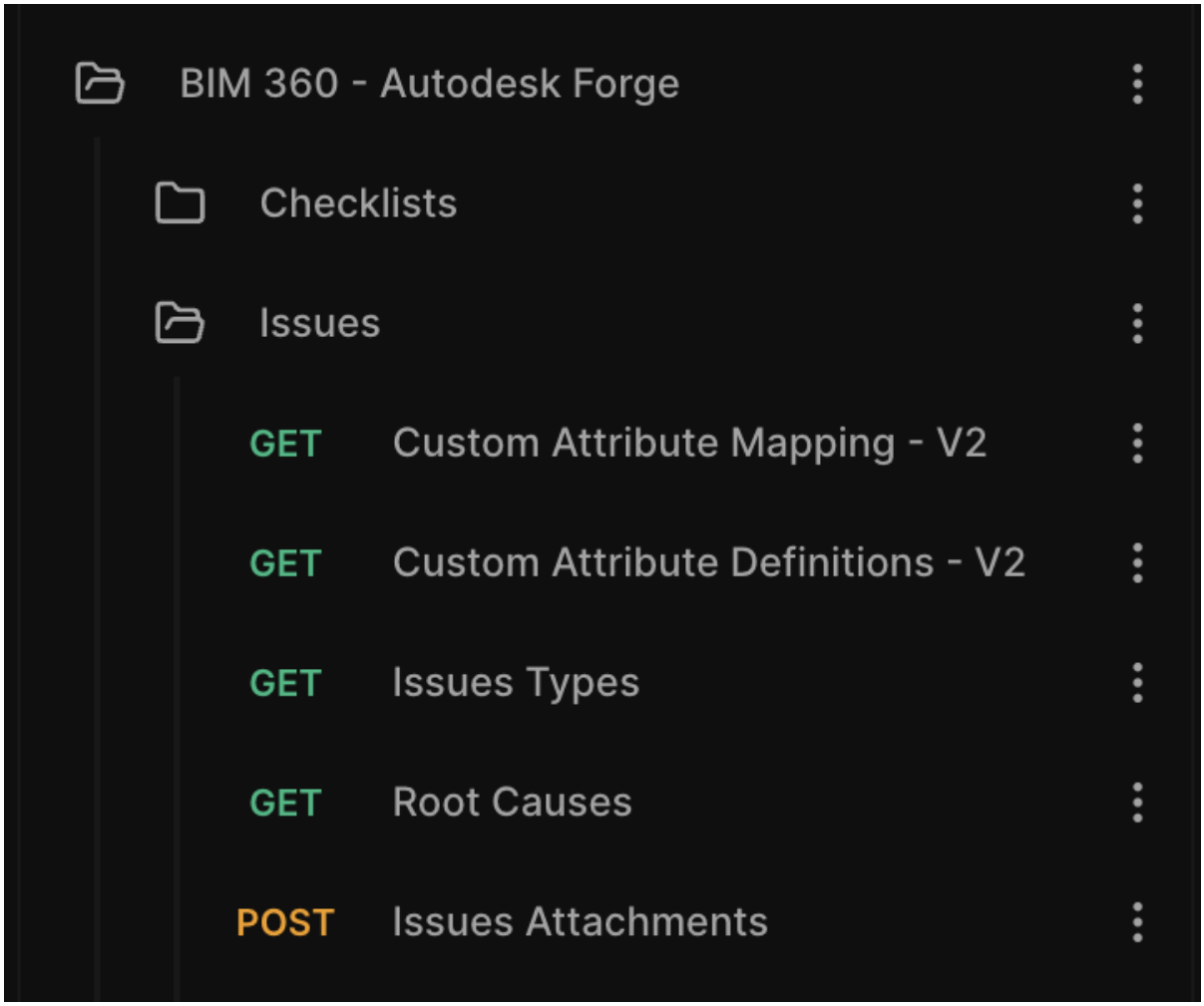
▼ Preface

This doc is written from my need to explain my ideas for this sync system to my team at Hoppscotch and will be worded and explained in that style because I think I have lost my ability to write general scientific paper style stuff right as I graduated from my alma mater, which is something I find deeply funny.

Hopefully I will revisit this later down the line and rewrite this to be more general and open as I am planning to make Syng available as a general library that can be implemented across multiple platforms. Anyways, since this is an implementation definition, I will try to write this down to the best of my knowledge into something that you can understand and be able to replicate a compliant implementation of the system. I am treating this as the source of truth of what Syng is from my end.

▼ The Problem(s)

I am going to explain this from the perspective of the problem I am trying to solve at Hoppscotch. Let me introduce collections, a hierarchical file system folder style structure where you can store your collections into.



A normal collection tree you might see in day to day usage of Hoppscotch

For teams, the pulling of the collections in a team is done in a basic round about way:

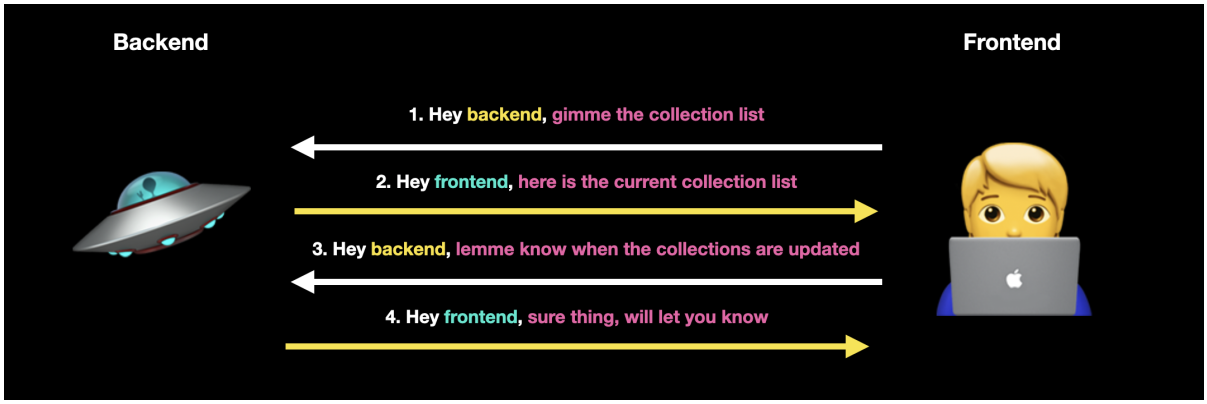
During initial load

In Hoppscotch, you cannot place requests in the root, we only allow collections in the root, so this is guaranteed to be all collections. Hence the collections in the root are called *the root collections of a team*.
In the frontend, we have a class called `TeamCollectionAdapter` [TODO: Link source code] which manages all of the interactions the frontend does with the backend in relation to the collection tree. It maintains an internal copy of the collections tree it knows with this schema.

```
type TreeCollection = {
  id: string; // The ID of the collection in the backend
  name: string; // The name of the collection (same as what is rendered on UI)
  requests: Array<Request> | null; // The requests in the collection
  folders: Array<TreeCollection> | null; // The subfolders of the collection
}
```

Upon fetching the root collections, we do not actually load in the contents of the collection (the requests inside and the folders), instead we set that as `null` to signal that we do not have these info. The adapter loads these stuff when someone requests it to ‘expand’ the collection tree, upon which it sends calls to the backend to get the collections and requests and populate that part of the tree.

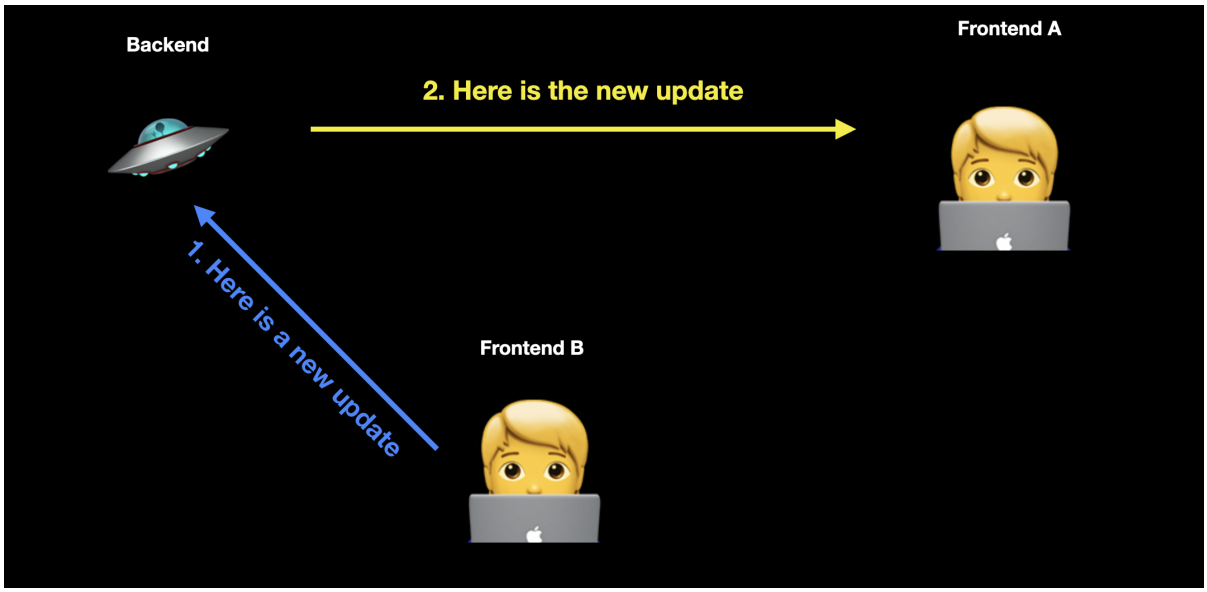
With this, we get the tree loaded in on demand, but we also need to know when the tree changes so we can apply those changes and show it to the user. For this, the backend also provides subscriptions which allows you to get events which happen to the collection tree. The `TeamCollectionAdapter` upon initialization will subscribe to these events and if it sees changes on parts of the tree which is loaded by it, applies those changes in the tree (the rest can be ignored as we can get the latest data upon the *collection expansion* anyways).



A summarisation of how the frontend and backend communicate to resolve the tree

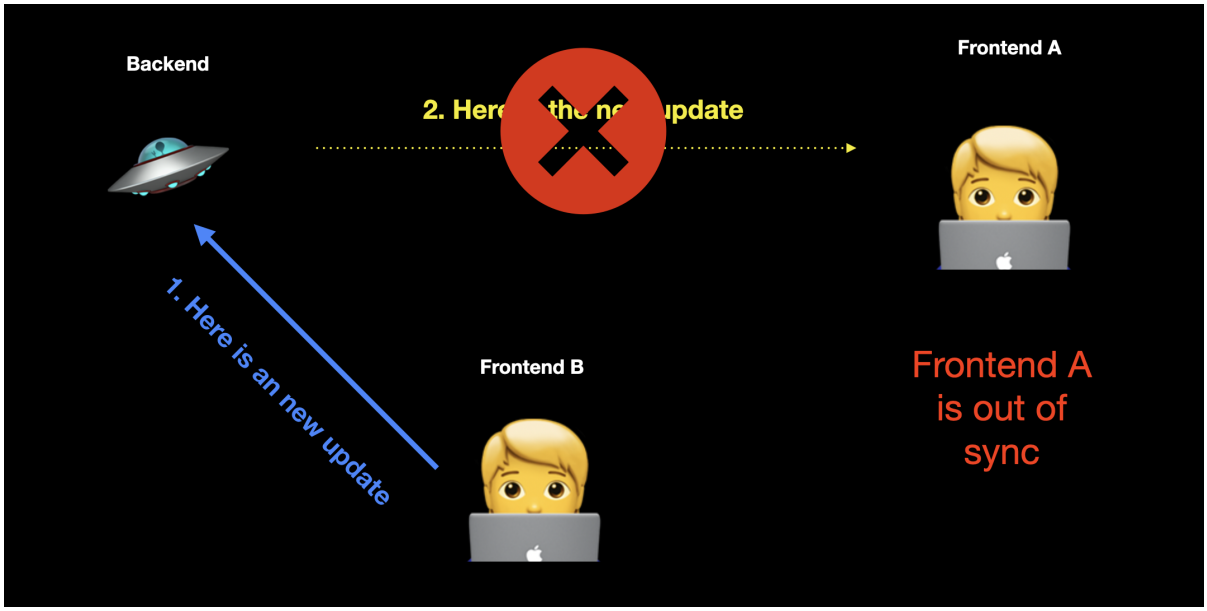
Multiple Users working on the tree / Consistency

With the subscription system in place, the frontend should be ideally well positioned to keep the tree consistent as with what the backend state is and align that in the UI.



Changes being applied and broadcasted throughout the active users

The keyword here is “ideally”. The changes are just broadcasted to subscribers by the backend, but what if the user accidentally drops an event, maybe due to network fluctuations or other weird stuff ? What will happen then ? Well, its pretty simple, the tree that frontend holds will be out of sync and the *adapter* will miss that update.



There are indeed multiple ways to fix this, one way is to take a page from the Actor model style and instead of it being events broadcasted, we can think of it like messages on the inbox where the backend keeps track of active listeners (in a way

similar to actors) and sends messages to their ‘mailbox’ which they can consume later on (even if connection dropped, see Actor model). But even here, we can have situations where long running situations may not be handled (if a TTL is applied) etc.

Caching

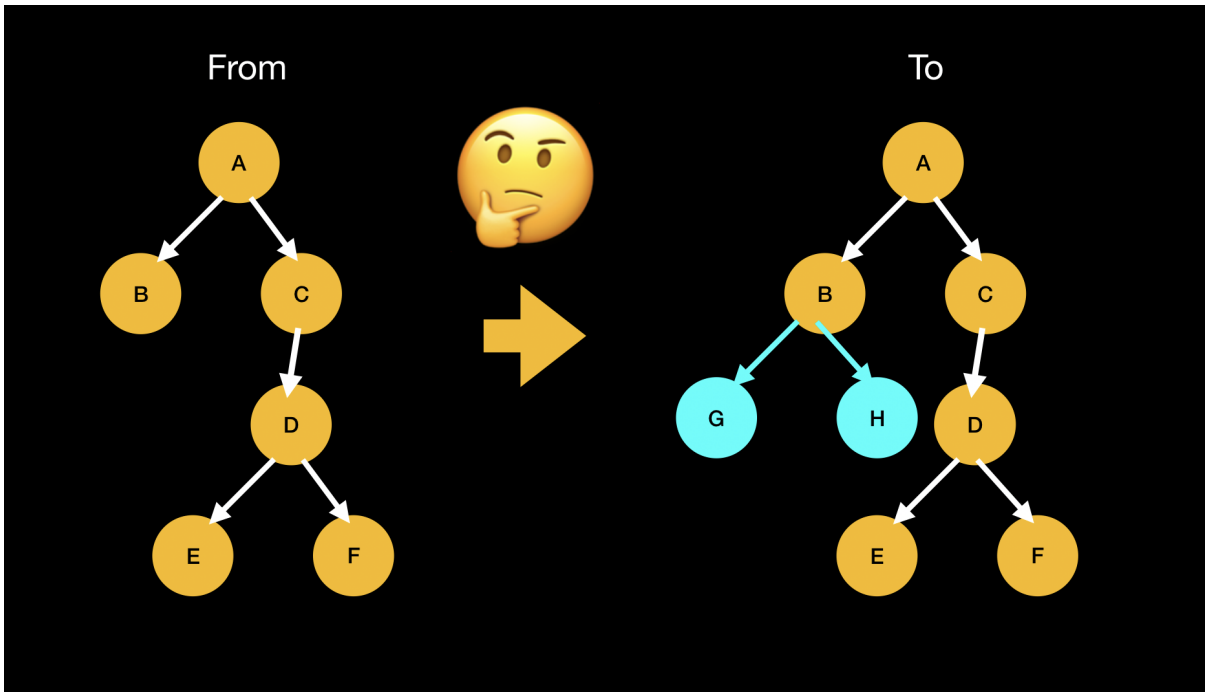
In the current implementation of `TeamCollectionAdapter` in Hoppscotch doesn’t exactly implement any caching at the moment, so every time the user hits the backend for some data, the backend has to get the entire tree (* mostly the root collections and opened folders on the course of operation) to provide to the frontend.

A good implementation for caching can be sourced from the collection and request ids (these are guaranteed to be unique by the backend implementation universally across all teams in Hoppscotch) and the frontend can make an internal map of the collection tree and persist it to the disk. The problem with this solution comes back again to reconciliation, how will we know the cache has invalidated with the current implementation ? Well, a sure to do that is to implement the cache to work with a *stale while revalidate (SWR)* model, where the user is served the cache and then revalidated. But the problem being, we can’t query the backend for what exactly is stale from the last sync point, instead, for revalidation, we pull that entire section down again and overwrite the cache with the new data (revalidation).

Reconciliation is the problem to be solved

From the above two examples, both of them are problems and scenarios where the ability to efficiently reconcile (the ability to figure out the diff) helps to (and/or):

1. Make sure the states are consistent across clients
2. Make it cheaper to transfer and communicate changes.



Solving this problem lies as an important thing in both code and cost wise (factors like networking)

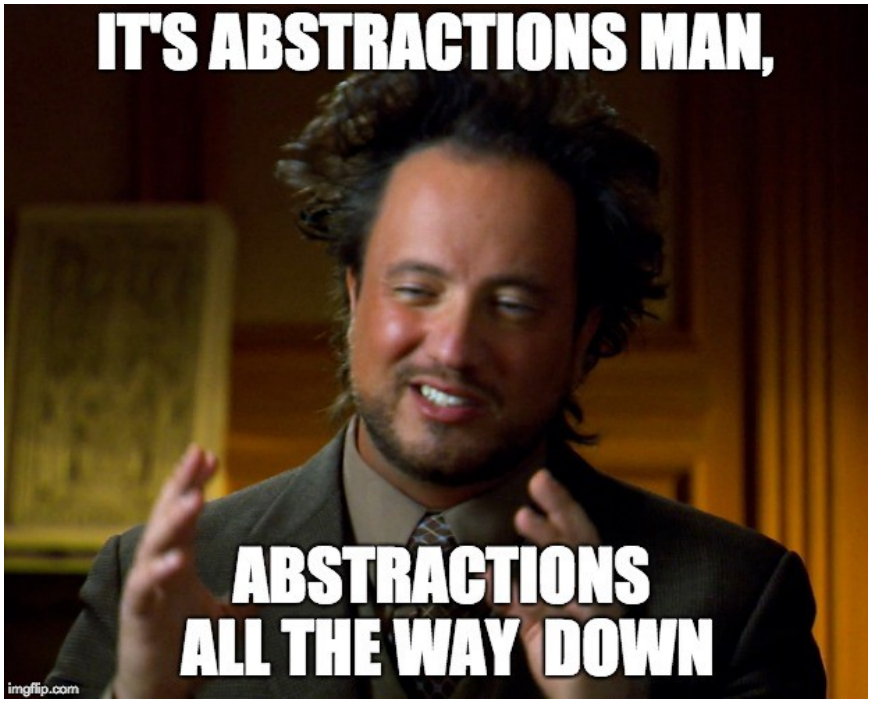
When I initially started thinking about this problem, I approached this whole consistency assurance and caching issues stuff as a frontend problem to be solved and to be removed from the backend and not introduced simplicity, but then as I planned this whole thing out, the frontend code to compensate this whole thing started looking really complex to a degree that it was looking more complex than a database with transaction queues, object graphs, pools and whole other complicated shebang.

Hence I invoked Occam's Razor and tried to look for a simpler answer which lead me to the final conclusion: We need a way to prove and calculate whether two given collection trees are exactly the same in the most cheapest possible way. This is where the idea for Syng is born.

▼ Concept

> **A good idea is something that does not solve just one single problem, but rather can solve multiple problems at once.**
> ***- Shigeru Miyamoto***

The idea behind the design of Syng is from this quote. How can we solve the above problems in a way that is cohesive and organic rather than putting together a quilt of patches to fix the different quirks. Well, you have to think about an abstraction that holds all the things together and the core foundational problems.



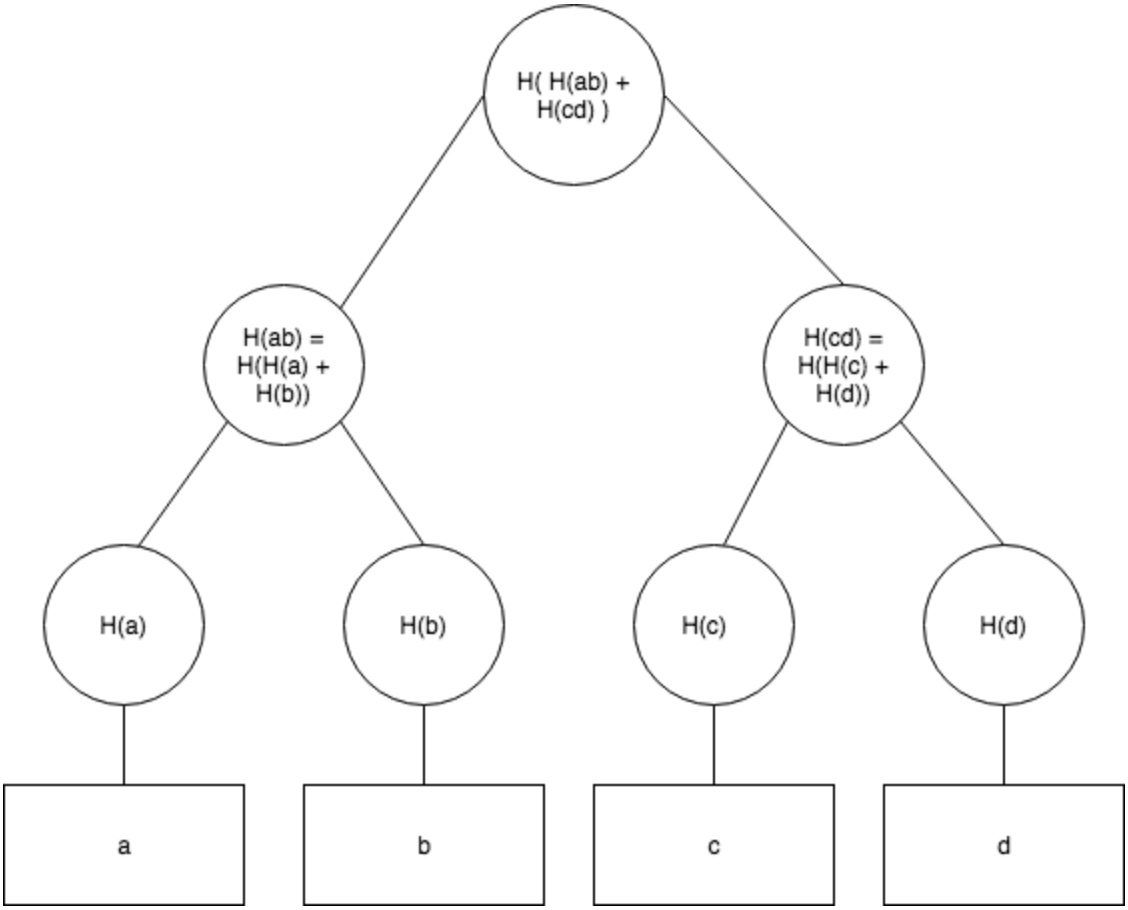
From the above mentioned problems, I came into a conclusion that reconciliation is the root problem to be solved, how can we communicate changes to the collection tree ? how to do that in a way that is computationally cheap ? That is the problem to be solved basically and everything else can be fixed/improved just by doing reconciliations back and forth.

So, here is the big question, how to solve reconciliation and make it cheap ?

That is where my journey took me next. I looked into the most fastest thing which reconciles trees that I have seen, which was Git.

How does Git know the diff between two commits and the changes introduced between two branches ? It can because of a data structure called the Merkle Tree.

A Merkle Tree (also known as Hash Trees) in simple terms is a tree data structure where each node in the tree has a Cryptographic Hash value which depends on the hash value of the children nodes.



A binary merkle tree. (Note: Merkle Trees don't need to necessarily be Binary Trees, Git doesn't do it for example) [Credits: https://ieee.nitk.ac.in/]

Git uses a Merkle Tree to track changes in a codebase. Each commit in a Git repository represents a snapshot of the entire codebase at a particular point in time. Git uses a Merkle Tree to efficiently track the changes between commits.

When a user makes changes to a codebase and creates a new commit, Git creates a new tree object that represents the updated codebase. The new tree object is constructed by recursively hashing pairs of files and directories until a single hash remains, known as the "tree hash". The tree hash is then used to construct the commit object, which contains metadata about the commit, such as the commit message and the author's name and email.

To track changes between commits, Git stores the hash of the parent commit in each new commit object. This creates a chain of commits that can be traversed by following the hashes of the parent commits. To efficiently track changes, Git uses the Merkle Tree to compare the tree hash of the current commit to the tree hash of its parent commit. This allows Git to quickly determine which files and directories have been added, modified, or deleted between the two commits..

This isn't a post about Git, so I have gone through only the whirlwind stuff, for more info on the Git internals, I recommend watching this talk: Advanced Git: Graphs, Hashes, and Compression, Oh My! - YouTube

I had initially thought about actually just running everything through Git as it is a proven protocol and funnily enough in JavaScript fashion of course following Atwood's Law, there is a JavaScript implementation of Git called isomorphic-git. Although I did consider it a bit more seriously initially, we are not exactly looking for version control here and Git adds a lot more overhead because of it being a VCS. For example, let's say I want to get the latest tree, Git will give you all the intermediate trees in between (as objects) through commits but as a sync system we don't need the intermediate ones exactly.

This is where the idea for a separate system was born. Syng's name is so because its a recursive acronym for "**SYng's Not Git**". It embodies what we need to know about it, being that it is a system that is very much inspired by Git in the systems it places but it being its own thing with it being a sync system rather than a VCS.

Along with that, the system is designed to do only one thing and this spec will follow only that one specific ability, the ability to communicate whether two given trees are the same and if not, what parts of the tree exactly have changed and it tries to do that in the cheapest possible way, both in:

- Computationally cheap to calculate the diff.
- Less amount of data as possible required to communicate and verify trees

The system may sacrifice other attributes of performance like read,write or update performance in order to gain these abilities. But in those places the system tries to

do as less as possible so that engineering decisions can be made around on how those issues are dealt with (for example, Syng doesn't even say how the tree should be stored, as long as the operations can be served the correct info through the interface, see the *Backend* interface below for more info).

> **Do One Thing And Do It Well.**
> ***- Unix Philosophy [Derived]***

▼ The System

Syng is a Merkle tree implementation intended for syncing, but before getting into the actual algorithm and stuff regarding the syncing operations and how they go about, we have to start building up the concepts from the ground up to understand better how they all work.

These are the core primitives that Syng implementors define/use:

▼ *SyngObject*

A *SyngObject* is a small data structure that actually stores the data in the syng tree. It is basically a *struct* containing 2 properties:

- `fields`: a map from strings into strings
- `children`: an array of *Syng Object IDs* (see below) that represent the children of this node.

```
// This does need not be the verbatim definition, things can be different
// and this is just an example
type SyngObject = {
  fields: Map<string, string>,
  children: Array<SyngObjectID> // SyngObjectID is a string following its spec below
}
```

The `fields` section is usually used to store content that needs to be represented in the system and is usually something implementors define and manage. The

iteration order of `fields` section is not guaranteed to be stable and can be different based on how the implementor implements the Map, so do be careful not to rely on the order and rather rely just on keys for lookups.
The `children` section represent the children objects of this object within the tree. There might be an assumption since its an *SyngObjectID,* the IDs inside it will be unique and hence the `children` array is actually a Set (as there are no duplicates). That is a wrong assumption as there can be objects with the same *SyngObjectID* but as different siblings (see *SyngObjectID* below). The iteration ordering in the `children` array will be preserved and the implementors shouldn't be messing with that ordering.

▼ *SyngObjectID*

The *SyngObjectID* is not a data structure per se, but it is an ID that is used to tag every *SyngObject*. It has these goals:

- Deterministic: Its not random!
- Reproducible: any device looking into the same *SyngObject* should generate the same output.
- Verifiable: (kinda an extension of the Reproducible part), the contents of a *SyngObject* should be verifiable from the *SyngObjectID* assigned to it.
- Collision-free (as possible): Although same *SyngObject*'s can generate the same *ObjectID*, unique *SyngObject*'s shouldn't have the same *SyngObjectIDs.* The system may fundamentally break if there is a conflict.

In the reference implementation for Syng, I have defined the *SyngObjectID* as the SHA-256 hash of the CBOR representation of the *SyngObject.*

▼ *SyngBackend*

A *SyngBackend* is an interface an implementor has to basically implement in order for the syng tree operations to work. This allows the implementor to store and represent the data however they want and as long as they can conform to this interface, they will be able to use the tree operations syng provides to mutate the tree.

Here is a (non-verbatim, but representing all the operations, for example, below doesn't represent error handling or asynchronity) TypeScript definition of the implementation:

```
interface SyngBackend {
  // Dealing with the current root SyngObject of the syng tree
  getRootObjectID(): SyngObjectID | undefined
  getRootObject(): SyngObject | undefined
  setRootObject(id: SyngObjectID | undefined): void

  // Dealing with reading and writing arbitrary objects
  readObject(id: SyngObjectID): SyngObject | undefined
  writeObject(obj: SyngObject): ObjectID | undefined
  hasObject(id: SyngObjectID): Promise<boolean>
}
```

Here is a clarification of each of the backend operations.

▼ ***getRootObjectID***

Each backend is supposed to hold a single syng tree and each tree may or may not have a root *SyngObject* (will be undefined if the tree is empty). *getRootObjectID* is supposed to return the ID of that root node if it exists.

▼ ***getRootObject***

Returns the root object that is stored in this backend's tree. This might seem to be trivial to have two functions one to get the ID and other to get the object but you can get the ID from the object if you want to. This is implemented so the backend implementor can provide more efficient implementations in cases where that is possible.

▼ **setRootObject**

Sets the given object ID as the current root object of the syng tree for this backend. This function should error out if the given input is not a valid object ID that is known by the backend.
If undefined is given as the object ID then it sets the syng tree as empty.

▼ ***readObject***

Pretty self explanatory. It returns you the corresponding object for the given object ID.

▼ ***writeObject***

Writes an object into the object store (or similar) into the backend. Writing a store doesn't replace anything, it just adds it to the object store or similar abstractions that the implementors could use.

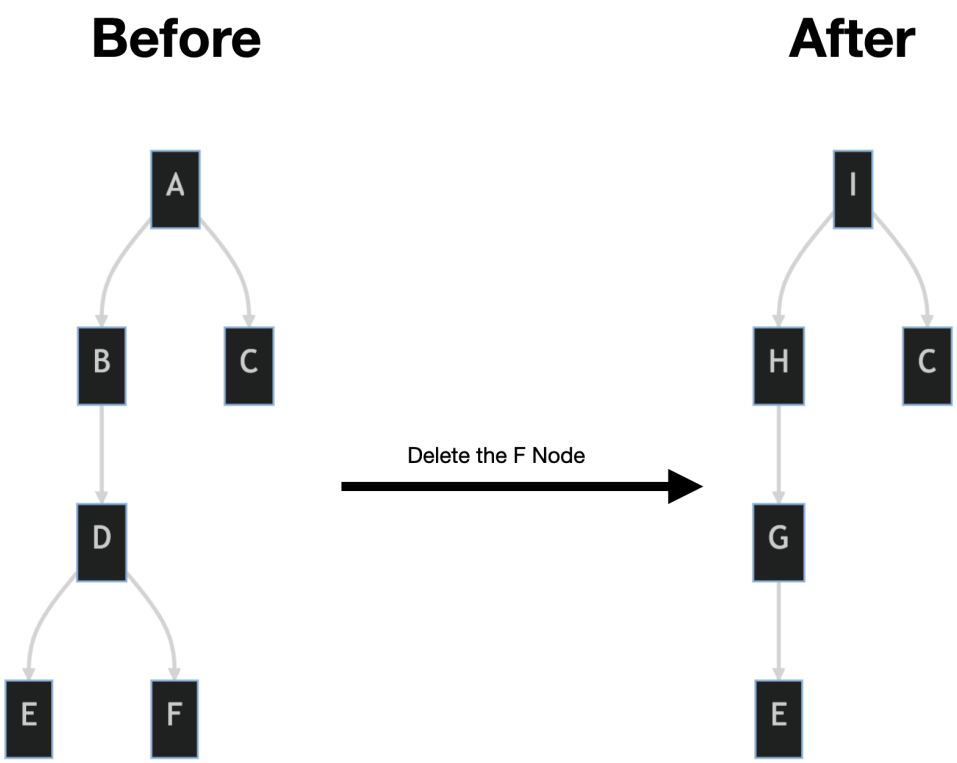
▼ ***hasObject***

Checks whether the object with the given ID exists in the store. In the standard implementation written in Rust, this is implemented with a blanket implementation (a default implementation) to use *readObject* and see whether it returned a valid object.

With the above primitives, Syng gives you operations to work with the tree and make changes to it, which are:

▼ Tree Operations (treeops)

When working with Syng Objects and updating trees, we cannot use the *readObject* and *writeObject* calls in the backend directly to manipulate the tree. This is because when we do an update to an object nested inside other objects, its IDs get updated, resulting to its entry in the parent's children to be updated, this will in turn change the ID of the grandparent and so on until it hits the root node.



The boxes represent the objects, arrows represent children and the letters represent the object IDs, the IDs change as the content of the objects change. Notice how the change propagated up.

To make these sort of updates easier, Syng implementations also provide higher level tree operations where you can ask them to run the mutations and side effects of updating the tree.

Since object IDs can't be used to uniquely identify a node in the tree, these tree ops use what are called index paths, which are arrays of 0 indexed unsigned numbers which are the child indexes through the corresponding nodes. For example, in the above operation example, the delete operation is done on index path, [0, 0, 0, 1].

The operations we have in treeops are as follows:

▼ ***addChildObject***

This function takes the backend, index path, the *SyngObject* to add and an enumeration about where the object will be written in on the parent (whether the object is written at the end of the children list or at a defined position). It returns the object ID of the object which is represents the input object. If the index path is invalid, it should return an error.

▼ *removeChildObject*

Given a backend and an index path, it deletes the object at the given path. If the index path did not resolve correctly, it should throw/return an error.

▼ *updateObject*

Given a backend, index path and the new object definition, it changes the object at that position at the tree. If the index path is invalid, the function should throw/return an error.

▼ Tree Diffing (Delta)

The whole point Syng is to be able to be able to diff, communicate and share tree changes easily, hence, implementors should also include utilities to do just that. The standard defines certain things regarding the delta and how it is communicated.

▼ The Delta Definition

There is a standard delta definition that Syng can generate and be applied (via *applyDelta*) and it looks sorta like this.

```
type SyngDelta = {
  startPoint: SyngObjectID | undefined,
  newRootNode: SyngObjectID | undefined,
  newObjects: Map<SyngObjectID, SyngObject>
}
```

- `startPoint` is the point of the tree to which this delta can be applied to. During the apply process, Syng will check if this aligns with the current root node of the target tree so that the delta can be applied gracefully. A value of `undefined` means it is to be applied to an empty tree.

- `newRootObject` is the SyngObjectID of the root node which will be the new root node of the target after the apply process is complete. If the value is `undefined` then applying the delta leads to the target tree being set to empty. By the nature of how Syng works, `newObject` will most probably be an object defined in the `newObjects` map, but it doesn't necessarily be so (implementations can choose to optimise diffs if they know the target tree might already have the object).
- `newObjects` is a map from the *SyngObjectID to the SyngObject*. The key ID is the hash of the corresponding *SyngObject*. The ordering of this map doesn't need to be enforced in the implementations and hence can be unstable (in ordering).

▼ Delta Application Possible Errors

Apply a delta is a process with a lot of moving parts so there are multiple classes of errors that can happen upon applying a delta (not exhaustive as implementors can choose to add more):

- **Current Tree Drifted:** The `startPoint` defined in the delta is not the same as the target tree's root object.
- **Delta Missing Objects:** Upon resolving the delta, it was found that the object contains object(s) that are missing both on the target object store and the `newObjects` field on the delta.
- **New Delta Root Object Invalid:** The `newRootObject` defined in the delta is not valid and not found in the object store of both the ticket or the `newObjects` field.

▼ Provided Operations

▼ generateDeltaFromPoint

This function takes the backend, and a *SyngObjectID* on the object store and generates the delta to take the input object ID (as the root object) into the current state of the tree. It returns the delta it generated on the processes for that update.

▼ applyDelta

This function takes a delta and the backend and applies the delta to the backend after a validation to make sure the delta is proper. If the validation fails, it can return the errors defined in the “Delta Application Possible Errors” section above.

That is it as for Syng as a system, it is pretty small, really out of the way and just tries to provide utilities to perform these operations. How things are stored, shared and everything is completely up to the implementor to implement, this is an intentional design choice to allow Syng to adopt any sort of problem domain more easily.

I have also cared to not mention the words ‘server’ and ‘client’ deliberately because Syng doesn’t care if the architecture is P2P or Server-Client, you can apply it on both the cases and just build on top of it to enforce platform rules (like consistency or source of truths).

I will try to give most of the commonly asked questions in the section below and I suggest you read it as well in case you have any doubts, if I haven’t answered your question, please let me know so I can introduce that into the document.

▼ Probable Questions

▼ How will I reorder objects in the Syng Tree ?

Well, currently you can use an *updateObject* call on the parent of the object you want to reorder and update the `children` array of the object to reorder at the moment. But a treeop for this is trivial to be implemented and maybe implemented later down the line as the system proves itself.

▼ How will I move objects in the Syng Tree ?

Similar to reordering, moving can be implemented through 2 *updateObject* where you remove the object from the `children` array from the source and write it in on the corresponding position on the destination.

▼ Syng doesn’t seem to implement a push operation to push the data, how will I sync then ?

Syng doesn’t provide a push operation so the implementors can adopt Syng for their usage basis. For example, how will the push take place and over what

protocol. In Hoppscotch's case we use GraphQL and WebSockets (via GraphQL Subscriptions) to communicate data between server and clients, how will that look like ? If Syng was adopted with those in mind, it won't be flexible. Hence, the exercise is left to the implementor. Generally, people just find ways to communicate the delta definition back and forth over a protocol like REST.

▼ **In order to calculate the deltas, won't I need to store the old objects ? Isn't that inefficient ?**

Yes and to perform delta generations you will most probably need to hold on to them. This is true. But, the implementors can choose to cleanup and remove unreachable nodes if they want to. For example, in Hoppscotch's use case, we would have to keep the backend objects even the unreachable ones available so when the frontend asks for changes down the line, we can base the changes off of that root object state with the old objects. But the frontend, doesn't necessarily need the old data, so it can be more frugal on it allowing to store older data as the frontend typically doesn't need the old data after the sync is confirmed and applied.