Notes on https://github.com/atdixon/me.untethr.nostr-desk - Feb 2023

Okay - going to try to brain dump on this project so that others can fork it, grow it, rebrand it, do whatever with it, ideally all towards the end of fulfilling fiatjaf's vision for nostr—here is fiatfjaf on Telegram:

```
let me just say why I have opened this bounty:

1. I want desktop clients, because they are necessarily and strictly better
than web clients
2. I love clojure even though I don't understand clojure, and I liked nostr-
desk's thread view and speed very much when I tried it
3. I want to see Aaron's ideas about a relay-centric design put in practice
because they may hold the future of Nostr
```

First, we probably want a good OSS license for it so you can rest assured I won't sue anybody and ruin everything in the future.

To that end, someone should make a good case for which OSS license to use and send a PR to the current project with a LICENSE file - and I can merge that and set the license in github etc.

After you fork the project, I will put a note saying something to the effect that this project is stalled please see XYZ project for ongoing development.

I reserve the right to come back to desktop project and take it in some other crazy direction at any time, but my primary hope is to support the new project as much as I can, with advise etc, and maybe even contribute code, etc, because most of my very little attention is on the relay side right now and I am scraping hours here and there already.

Okay - notes. First technical stuff, and then I'll go into all the product management and future wishlist stuff.

JavaFX / cljfx

JavaFX: https://openjfx.io/
cljfx: https://github.com/cljfx/cljfx

The project is based on JavaFX and cljfx (wrapper) at the moment and you're going to want to become intimate with both toolkits.

(HumbleUI, though, looks amazing though: https://github.com/HumbleUI, see early blog post: https://tonsky.me/blog/clojure-ui/, and if I had copious time I might even try to move to using that.)

This section is going to be pretty dense; you'll want to refer back to it as you make more and more sense of the code.

First of all, cljfx is pretty great I think. I love the declarative React model especially for rapidly prototyping UIs.

So general usage looks like this: there's a global ***state** atom. For example in our case one of the fields in this atom's value is :show-new-identity? — when this is updated in the atom to be true, it causes a re-render which ends up setting the showing property on the new identity modal dialog and wha-la the modal shows. That's such an elegant declarative way to manage UI transitions, thank you React.

However, you have to be willing to side-step this model when it makes sense. And there significant cases here this is going to make sense. Specific example for us is how we instantiate the primary view - the timeline, ie, what we call "Home" view at least right now in the current code.

The trick with long list views where you might have 100s or 1000s of records to show as a user scrolls is you don't want to make a UI component that you keep populating with all those records. You need to do something more efficient where you only create a few UI components to represent the list items/rows on the screen - so if the screen is only showing like 50 rows, you want to create 50 or so UI rows and populate them only with the data that the user needs to see in the viewport.

JavaFX's ListView is optimized to do all of this for us. So we just instantiate a singleton ListView and *manage all the updates/mutations to it ourselves*. You will see in the code that we actually use cljfx to instantiate this singleton (via fx/instance + fx/create-component) in view-home-new namespace (create-list-view).

Then we actually keep a reference to this singleton in our ***state** atom value BUT we make sure we NEVER change that reference. So that we never re-render the list view after it's set on the right component (see fx/ext-instance-factory / create) when the view get rendered for the first time — see the main-pain function in the view namespace. Any updates to the ListView after the first time it's "mounted" in the UI tree are explicit mutations — e.g., see timeline-new/update-active-timeline! where we .setItems on the ObservableList that is the data backing the ListView…the ListView reacts to changes to that ObservableList but this reactive behavior is first-class JavaFX, completely outside of the purview of cljfx.)

[NOTE: you will see namespaces like timeline and timeline-new, view-home and view-home-new. Initially this app showed the full tree-like thread of a conversation in the home view. But this was not the best UX practically speaking b/c as data arrives you'd see threads kind of jitter around in re-rendering and it would just take a ton of work to get this right. So the model that current clients are using today where you just show a flat view of events and replies and then click-through to a single threaded view of an event is ideal for now. This is where the "-new" namespaces are headed.]

But outside of contexts like the Home/Timeline view that need to be managed explicitly like this, cljfx can manage other parts of the UI that are fairly standard-fare. The primary namespace that defines the entire UI tree is the `view` namespace. The function `stage` defines the stage/window and references `root` function which declares the BorderPane that has left / center / bottom elements, and so on.

Relay UI. Later I'm going to complain about the relay settings UX approach and how we should move this product to be more like a relay browser, but currently the relays are managed by clicking in the status bar (bottom-right) and you can add relays and set whether you want them to be read/write.

When the main process is started, a periodic job is started that - presently every 10s - will look at the connected status of all the relays and update a :connected-info in the global ***state** -

which causes the status-bar at the bottom to re-render … any connected healthy relay will show up as a green dot. Any relay that is not connected (either due to error or because the relay is turned off for both read and write) will show up as gray. You can hover over any dot to see the relay URL that corresponds to the dot.)

Whenever relays are edited, the :relays field in the ***state** atom value should get replaced and a re-render of the status bar should occur via clfjx.

(Note: For some reason there is a :refresh-relays-ts field in ***state** that I update at certain moments, but I can't recall why I did this and if it's truly needed. I will say that I've used this pattern before, where if I wanted to force a re-render of some component even if cljfx otherwise wouldn't notice a relevant state change, I'd use a property like this (:refresh-relays-ts) which is just a timestamp that is fed down to the component—here, the status bar—so that just by changing this value to a new latest timestamp I can force cljfx to re-render a component.) This pattern can come in useful at select time - eg, if you are updating a mutable value in the state (not idiomatic in immutable clojure) but, if you need to do this for some reason, cljfx can't detect that then this timestamp pattern can help… anyway that explains that pattern, and you may wish to play around with the need for this pattern in this particular case with relays, go for it, but it's not hurting anything and you've probably got bigger fish to fry.)

Styles:

Ok, so JavaFX uses a bastardized CSS format. And it sucks because it can be surprising when components play well with it or not. Sometimes you need to direct styling with first-class component features and sometimes you can get generalized css to work. Getting all this right requires understanding JavaFX deeper. Hopefully the code has enough examples to make reasonably productive with copy pasta today.

You can style stuff either directly in clfjx components with :style or you can refer to :style-class and use a css stylesheet. We do both here. See style.css.

IMPORTANT: if you want to get style.css to reload for you in the repl so you can iterate on styles without having to restart java process every time, you will want to set the "cljfx.style.mode" system property (see the top of the app namespace—this is commented out) but it is very important that you don't package and deploy the client with this enabled—both for performance but also I've done mac packages in other projects and enabling this will just break stuff in production usage, just keep this in mind. Ie, as a deploy checklist make sure you don't have that system property set to true.)

Here's something that is super helpful when trying to get UI layouts right. There is a utility in style namespace called `BORDER|` just set it as the :style on any component and you'll get a red border around the component so you can see how the layout is going for it. Example:

```
{ …
  :style (BORDER|)
  …
}
```

You can also make it other colors and thicker:

```
{…
 :style (BORDER| :green 10)
…}
```

And merge in other styles:

```
{…
 :style (BORDER| {:-fx-background-color })
…}
```

This is really helpful to quickly see how layouts are going. Keep in mind the “cljfx.style.mode” system property — so that you can change components with this BORDER| thing and then get a re-render to happen and iterate like this in a REPL instead of slow iterations involving restarting the process.

Events and Effects.

Okay - how do you do events. By example. Currently you can click on a “keycard” on the left-hand side to switch identities. So you’ll see something like this in the code:

```
:on-mouse-clicked {:event/type :click-keycard :public-key public-key}
```

That’s cljfx declarative coolness. What this does is cause a dispatch to the handle function in our event namespace. The event type will be :click-keycard and you can have arbitrary other values like :public-key here come with the event. We’ve got a case statement in event/handle that does the right thing here.

So we can do whatever we want when an event occurs. But cljfx has an “effects” model .. and what that means is that we can return from event/handle what we effect we want to occur.

For better or worse what we are doing here is whatever immediate stuff needs to happen and then returning effects that are either “foreground” (:fg) or “background” (:bg) … we return nil in some cases which means do nothing else. But otherwise we return things like this:

```
[ [:fg <some function to run on UI thread>] ]

[ [:bg <some function to run on a background thread] ]
```

(Note the vector of vectors — you might get confusing / obtuse errors if you mistakenly only return one vector)

There are probably “better” patterns here and you can get really declarative with effects but I preferred just to think in imperative functions instead of all the indirection so this is what you see here.

In the specific case of clicking the key card, this is what happens right now:

* event/handle gets invoked
* the event type is :click-keycard so we invoke the private click-contact-card function which returns a :bg effect
* the bg effect is a function that runs on the background thread which invokes timeline-new/update-active-timeline! which updates the *state :active-key so that the active identity is identified in the ui (today an ugly fat border is put around the key card) …
* AND we mutate the data underneath the ListView (mentioned above)
* (currently the code also calls the old timeline/update-… function which is deprecated, see note above about “-new” namespaces)

UI Thread.

By the way, just in case you're not coming from UI backgrounds. Almost every UI system out there is going to have this concept of a UI thread. The key thing is that any updates to UI components MUST OCCUR on the UI thread.

Aka the FX thread.

Using cljfx, you can fx/on-fx-thread or fx/run-later to run relevant stuff on the UI thread.

If you don't manage doing updates correctly on the UI thread you can get to all kinds of weird stuff.

Rendering Notes / RichTextFX.

Okay here's where there may be some real challenges.

We use this FX component to render notes - https://github.com/FXMisc/RichTextFX

And we do some of our customizations of it in the me.untethr.nostr.rich-text

The entry point / component factor is our `create*` function in that namespace.

At least today we can render hyperlinks.

I have not yet tried to render images and such.

Doing a good job here may involve getting really down into the weeds of that RichText components and javafx to take over rendering and do it ourselves. May god have mercy on us in this effort.

Final Notes on UI.

Desktop UI coding is never a pretty ordeal, and I can't say it's ideal here because we have extra complexities in JavaFX itself.

I also can't say I'm managing everything perfectly even as I've had to workaround shit with these platforms.

The payoff is great though - b/c fiatjaf is right - the speedy desktop experience is so much better than shitty web experiences.

Storage / Sqlite

The SQL tables model is hopefully self-explanatory.

Although the DB indexes are likely in need of some attention - they are probably all wrong right now. But wait until you see any real perf issues before you go deep on that. Clients aren't really operating at the scales of relays so this may never be an issue. I can advise on this if needed.

There is no DB migration pattern right now. So may need to design that - there are probably some good clojure answers to this. For certain there are Java answers to this and these can be used right in clojure. Liquibase was something a long time ago when I had to deal with SQL migrations but there may be more modern tools.

Timeline Model.

See the `defrecord TimelineNew` in the timeline namespace:

```
(defrecord TimelineNew
  ;; these field values are only ever mutated on fx thread
  [^ObservableList adapted-list
   ^ObservableList observable-list ;; contains UITextNoteWrapper
   ^HashMap author-pubkey->item-id-set
   ^HashMap item-id->index
   ^HashSet item-ids
   timeline-epoch-vol])
```

When you have multiple identities / multiple keycards on the left hand side, we keep track of a TimelineNew instance for each identity. TimelineNew here is the data model.

The per identity timeline are In the global ***state** value, in the :identity-timeline-new field. This is a Clojure map with pubkey keys and the value is TimelineNew instance.

These instances are created with the new-instance function in that namespace and we create the underlying observable-list which will populate with all of the notes that we want in this persons's timeline - and we update it as they come at any time.

The adapted list is just a "view" list over the observable-list that filters it to a hard-coded last 20 days (not necessary and should consider removing this at some point) AND it ensures the list is sorted by created-at time so that the timeline is in order!

Other than this we have some HashMaps and HashSet that indexes items in the timeline so that as stuff comes in we can quickly decide how to update the timeline. I could write more here but hopefully the code speaks some for itself but let me know if I should document more on this.

ONE NOTE: the code for the old Timeline is still there - and it's all commented out from the UI but you could re-enable it if you want to play with it. HOWEVER it is is crazy. There was so much indexing and juggling involved and it may still have same bugs. And is using a closure graph library. Why? the reason is this: We were supporting a threaded view in the ListView but notes can come from our relays and our local database and populate the timeline but they can come in any order! So there was all this graph theory juggling to put in shell notes and readjust the thread when intermediate notes come in … what a nightmare. It was ultimately working but the experience was not great and the code complexity just not great so maybe some future thing could be done here but not doing this now would get in the way of realizing an easy version of this product and it's probably not even a good approach in the future term.

THAT SAID, you now have the task of implementing an open thread view. Which is a bit easier because you can just query for the entire thread in one go and not show it to the user until you feel you've got everything you need to render. (Also note that at the time of this implementation there was no such thing as an EOSE message so there was no way to know when you got all the results for query which put more pressure on rendering everything right away whenever a note would show up.)

The dispatch-text-note! function in this namespace is responsible for adding a newly arrived note to the timeline, using the TimelineNew data model and indices to do the job here. This update is necessarily done on the UI thread and mutates the observable-list, which in turn immediately updates the UI ListView if the associated TimelineNew is the one backing the timeline that that is currently in view for the current identity/keycard.

Nostr Event Handling.

First we use Aleph and Manifold libraries … https://aleph.io/ so that we speak streams everywhere.

When the app first starts up, it mounts the UI renderer and the next thing it does is start a stream consumer via consume/start! in the consume namespace.

More on relay connection management later but basically this consumer stream is arranged so that it gets ALL events from ALL connected relays.

The consumer logic's job then is to handle all events — which for notes involves going through some crazy pinko chip logic (see verify-maybe-persist-event!) to determine if we already have this event, have we already verified its signature, does the incoming event match etc, and finally verifying the event if it's the first time we're seeing it AND then caching both in memory and in the database that we have this event. So the next one doesn't need to verify and all that.

This may be premature optimization but hey it's there now.

So in the consume namespace you'll see the handling of events — if it's a note type you'll see the code that dispatches it to the TimelineNew (see above). And if it's other types we have the scaffolding there to handle other types even if the functions are no-ops today.

For kind 0 metadata updates, we update the metadata cache…

Metadata Cache.

The metadata cache contains profile information for all the pub keys we might see — our own and our follows and pulleys that show up in our views, timelines, global feed (if we ever implement one), etc.

Okay - this is cool - when we render we can use the metadata cache to find out if user has an image and we can render that instead.

One caveat though - we are not reactive on this case. What this means is that if we're subscribed to the events of one of our followers. And that follower updates his or her profile image, our cache will get updated, but our UI components will not. Sometimes you'll see the update of a profile as you scroll through the timeline ListView — because as we unload/reload cells we'll pick up the new metadata cache stuff.

But in general you won't see the updates. More thinking about how to make this update components is needed, and it the mechanics here may need to be reconciled altogether but this is probably lower priority until other bigger fish are fried.

There is also an image cache - see avatar namespace.

Work needs to be done to (figure out how to) get JavaFX to render images better.

App Startup Sequence.

Currently:

```
(defn -main
  [& _]
  (fx/mount-renderer *state renderer)
  (consume/start! db *state metadata-cache daemon-scheduled-executor)
  (util/schedule! daemon-scheduled-executor load-identities! 1000)
  (util/schedule! daemon-scheduled-executor load-relays! 3000)
  (util/schedule-with-fixed-delay!
    daemon-scheduled-executor update-connected-info! 4000 10000)
  ;; CONSIDER shutdown hooks, graceful executor shutdown etc
  )
```

First thing - mount the cljfx reactive UI renderer stuff.
Second thing - start consuming events from the stream (discussed above).

Finally we immediately schedule to load-identities! asynchronously which will hydrate our global *state value with stuff including initial timelines (getting events from the database) and then setting up nostr subscriptions to relays.

By the way all this code was written before the EOSE event / nip came into existence. So we always subscribe to events from the past 45 days.

What we really need to do is keep track of per relay “watermarks” — so we don’t go back and query for the same events over and over again.

We should keep a watermark for a given relay that says up to what timestamp we’ve got all the data for that relay. Use EOSE as the notice that we’ve got all the events for a query. Etc.

This is going to take some work - b/c we probably want to “walk back” in pages using until and limit in subscriptions to gather event data. But the end result will be we are mostly caught up with all relays so that when we start the app, we get everything historically from the database and our query to relays is just for stuff fairly recent and going forward.

(Future note: old events can populate on relays after the fact. For example if a relay backfills some data or mirrors from another relay. So having an eose watermark for a relay won’t always guarantee that you’ve got all events that the relay has prior to that watermark. So there may be some play where eventually — or per user explicit/manual action — you go back and query to see if there is any new old stuff for a relay…)

In startup, the other thing we do is schedule an async load of relays. Which invokes the relay-conn/update-relays! function in the relay-conn namespace…

Relay Connection Management.

Here’s the public contract of the relay-conn namespace.

Note a lot of internal work is done to make this a thread-safe, concurrently accessed interface.

Note that if a disconnect occurs the management here will attempt to re-establish connection. Because of the way we chain connection promises, if you send something to a relay that is not yet connected, we will have a chain listener on it such that the send will eventually get through so long as we are able to finally make a connection to the relay (this needs to be verified fully in the face of continuous retries, but this is ultimately the desired behavior — relays are not going to be 100% uptime but users are most likely going to want their events sent to the relays they choose - or be notified if they are not getting there successfully…)

This namespace could probably be factored out, given a Java interface wrapper, and made into a library for other JVM (kotlin, java, etc) users:

`update-relays!`

```
  -- change the set of relays and/or their read or write status
```

`send!*`

```
  -- send event object to a given relay url, either using an existing persistent
connection or creating a connection on demand

    (i.e., we keep persistent connections to relays that are marked as read, but we
don't for write-only relays, so these will get open & closed on-demand)
```

`subscribe-all!`

```
  -- establish subscription to all relays marked for read

    (note if update-relays! is used to add new relays, outstanding subscriptions will
get automatically and transparently added to these new relays)
```

`unsubscribe-all!`

```
  -- kill a subscription by id that was previously established via subscribe-all
```

`sink-stream`

```
  -- return back a stream that will get *all* events for every subscribed read relay
```

`connected-info`

```
  -- return a map where the keys are a relay-url and the value is true if the relay
is successfully connected

    (this is polled frequently by a scheduled recurring job and used to populate the
green dots in the status bar)
```

Note one of the things we do whenever we see one of our own (identity) contact lists change is call subscribe/overwrite-subscriptions! which in turn replaces all subscriptions with a new that includes all the changes to our contact list.

Product Notes.

Finally where should we take this from a product perspective. That's your choice ultimately but I have some ideas an opinions.

- current relay settings management. we should work toward become more of a relay browser. there are some good initial design possibilities out there. i would focus on super simple starts BUT i would love it if an early desktop version just simply gave me a list of all known relays but did NOT connect to any but rather gave me the ability to browse to any one that i chose. And ultimately gave me information about what on that relay we've seen on other relays previously.
- left-hand side identity management. i think it'd be better if the real estate were mostly the core experience and identity management was elsewhere in a modal or something

Basically if we could get a first version out with just basic relay browsing capability that would be amazing.

Other Thoughts.

One strategy might be to start a new project and pull code from this one and build up the UI from there.

Given that so much of this is prototype and the protocol and thinking has changed so much since this one's inception.