Phase 1: Transparent Offline First

This phase pertains to the following acceptance criteria: 1, 2, 3

The first step towards implementing offline first functionality will be to have it work entirely transparent to the user; they won't really know that the app can work offline, but all of their requests will be queued and dispatched in the background.

They won't know how many changes still need to be synced, or have the ability to manually sync things, no service workers, etc. The only thing they'll get is being notified if a request persistently fails and a change has to be rolled back.

Stories

- Story UFC-146: Implementing Phase 1: As a web user, I want to be able to use the app while I'm offline.

Prior Art

Here's some things that I've researched that are relevant for the design at hand.

- `redux-offline`: GitHub - redux-offline/redux-offline: Build Offline-First...
  - A middleware for enabling redux to work offline.
  - In theory, this is trying to solve the problems that we are trying to solve.
  - However, because it acts as a separate middleware layer (that would sit in front of the sagas), it's not ideal when we want to keep all of the business logic as a sagas.
  - There are a couple other reasons that I'm adverse to using it (mainly because it's interface of applying meta properties to actions doesn't really play nice with what we've, again, been doing with sagas), but suffice to say there are some good ideas we can take away from it.
  - In particular, the ideas of effect/commit/rollback, and the idea of having a long poll to the backend server as a check for network connectivity.
    - This second point is particularly noteworthy because the 'native' tools for detecting network 'connectivity' are debatably bad. The Service Worker/Web API method for determining connectivity has implementations that defer between browsers and, for the most part, only look for if the computer is connected to a network (but not necessarily if that network connection is functional). And the React

Native method for connectivity is equally unreliable. As such, long polling the backend server is probably the most reliable method.

- The ideas of effect/commit/rollback will probably form the basis of our offline-first design.

Design Brainstorming (v1)

The following is a brainstorming of ideas for what would be needed to be implemented to achieve each of the acceptance criteria.

OfflineRequestSlice

- So we currently have the concept of a `RequestSlice`... what if we extended it so that we also had an `OfflineRequestSlice`?
  - Basically, you know how `RequestSlice` has the `watchRequestSaga` right now? That is, a function to register and wrap a 'request saga'? What if we instead had *three* such saga wrappers, one each for effect, commit, and rollback.
    - **effect**: This saga would handle making the actual request to the backend to persist the data or make whatever change is needed. If we want to take inspiration from `redux-offline`, then theoretically this could be even simpler and just be a function that maps `payload` to a json object that represents how to call the Feathers service/method. It'll have to depend on how much we think we can abstract without losing customizability.
      - The wrapper would be the thing handling retries, timeouts, and errors. And since it would be responsible for errors, ultimately it would be responsible for raising the condition that forces **rollback** to be used.
        - This package could be helpful for those network related handling tasks: GitHub - connor4312/cockatiel: A resilience and...
      - Hmm... but if this wrapper is responsible for the network related handling tasks, then how does the central request queuing authority fit in?
        - If the wrapper does error handling, then does the queuing authority need to even deal with re-queuing a request? Under what circumstances would a request be re-queued?
        - If the wrapper does something like "try request, and then retry X times with exponential backoff, then error when X retries have failed", what do we do with the error?

        - Instead of having the **effect** wrapper responsible for invoking the **rollback** saga, maybe this should be delegated to the queuing authority?
            - As in, the **effect** saga could throw an error, which would then emit the `effect_failure` action, which would be picked up by the queuing authority, which would then either re-queue the **effect** to try again later, or would decide to **rollback** if it's failed "enough times" (i.e. given some predetermined logic).
            - The queuing authority would thus have to have some metadata to store alongside the **effect** to keep track of things like number of times tried, time of insertion, time of last attempt, etc.
- **commit**: This saga would handle committing the data/change to the local store.
    - I don't necessarily know what the wrapper of this saga would do...
- **rollback**: This saga would handle removing the data/change from the store.
    - The wrapper would handle doing things like dispatching actions to notify the user that the request has failed and been rolled back.
- What about validation? i.e. data validation? Currently, in something like the `accounts` or `transactions` saga, we run validation on the payload before we **commit** to the store or save to the backend. If we split up the task into separate **effect** and **commit** sagas, then won't we have to duplicate this validation logic?
    - My first thought was to have another saga to register, something like a **preprocessing** saga, that would sit in front of the **effect** and **commit** sagas and perform any validation or data processing before them.
    - But then I thought, what if instead of having a separate **preprocessing** saga, we just combine it with the **commit** saga? And it would be responsible for **processing/validating** the data, **committing** it to the store, and then emitting the action indicating to queue the **effect** for the request. So something like this:
        1. A `request` action is fired.

2. Instead of the `request` being picked up by the queuing authority, the **processing + commit** saga picks it up.
3. Any data **processing** is performed.
4. The **commit** to the store happens (i.e. the action to save to the store is emitted).
5. If the **processing** and **commit** are all good, a new `effect/queue` action is emitted (this could even happen implicitly by having the **commit** saga just return the value that is to be submitted in the **effect**, and have the **commit** wrapper take that value and automatically fire the `effect/queue` action; if nothing is returned, then it could just take the payload implicitly).
6. The queuing authority listens for all of these `effect/queue` actions and queues the effect.
7. The `effect/queue` action has the `meta` properties that the original `request` action would have had if it was the one being queued (i.e. the actions to dispatch for `effect/start` and the actions to listen for `effect/success` and `effect/failure`, along with all of the `rollback` actions).

  - Adding this extra step to only queue the **effect** once the data has been **processed** and **committed** seems to solve the problem.

- (having brainstormed in the Queuing Authority section...) So now we're looking at each `OfflineRequestSlice` having the following actions:
  - `request/start`, `request/success`, `request/failure`
  - `commit/start`, `commit/success`, `commit/failure`
    - But is there any use for `commit/start`? I would think that the start of the **commit** saga would happen as a result of the usual `request` action, since that's the optimistic trigger.
    - On second thought, I think, in order to maintain backwards compatibility, we shouldn't have any namespaced **commit** actions, but should instead just use the `request/success/failure` actions that we were using before.
  - `rollback/start`, `rollback/success`, `rollback/failure`
    - ?? Maybe, maybe not

        - Yes, I think we'll need these, since the queuing authority will need them for error handling.
    - `request`
    - `clear`
- Second go at revamping the actions we'll need:
    - `request`
        - Starts the process of committing the data; emits `effect/queue`.
    - `success`
    - `failure`
    - `clear`
    - `effect/queue`, `effect/start`, `effect/success`, `effect/failure`
        - In fact, for `effect/queue`, instead of it being an action that's namespaced for each `OfflineRequestSlice`, it could instead be just a plain old action that is exported separately. Since it's only dispatched to and received by the queuing authority, then really that action just belongs to the queuing authority.
        - However, since the **commit** saga wrapper will need to dispatch it, there are two places we could define it:
            1. As part of the `OfflineRequestSlice`.
            2. As part of some definition of the queuing authority that isn't specific to sagas (but could be imported by them). But it can't be in the queuing authority 'slice', since we can't import things from slices into `OfflineRequestSlice`...
            3. So, some other third spot that would get imported by both of the above?
        - Hmm... probably best just to stick it in `OfflineRequestSlice` as something like a static property for now.
    - `rollback/start`, `rollback/success`, `rollback/failure`
- But then where does the `loading` state play into this?
    - I guess if we're going with offline-first, optimistic data changes, then the `loading` state really only plays off the `commit` actions.

        - And then it's up to the queuing authority to deal with 'loading' states for requests (in Phase 3, this might manifest in something like a "Syncing" notifier in the header bar).

    - As for `error` state, I guess there would technically be two things that could go in it? Errors for the **commits**, and errors for the **effects**.

        - Should the errors be separated out in two different state pieces? i.e. `error_commit` and `error_effect`? Or just don't distinguish between the two and use just the one `error` state?

            - Ugh, I'm going to need to build a bloody state machine diagram to model this whole process...

Queuing Authority

- We would also need to handle queuing the **effects** for later execution if the app is currently offline.

    - My first thought was to localize the queues to each `OfflineRequestSlice` (i.e. once the `request` action comes in, queue it into the request slice's store state, then have a process that processes the queues and dispatches the requests), but localizing the queues removes the ability for us to keep chronological order of all requests. That is, we want the order of all requests between all `OfflineRequestSlices` to be consistent so that they are applied in the order they happened.

        - And we want this chronological order because things could get really messy if changes start happening out-of-order (e.g. multiple changes that affect a transaction but get overridden if applied in the wrong order).

    - My second thought was that, if we couldn't localize the requests queue in the `OfflineRequestSlice`, then we'd need some central authority that listens for any and all requests and queues them itself. The flow would maybe work something like this:

        - The central authority listens for any and all `request` actions from any and all `OfflineRequestSlices`.

        - A `request` action is dispatched from an `OfflineRequestSlice`.

        - The central authority picks up the request and queues it.

        - Immediately after queuing it, a process runs to try and start processing the queue.

            - If the app is considered 'online', then the process starts firing the requests.

        - If the app is 'offline', then the process just lets the requests sit in the queue until another request is made, or until the app connectivity state switches to 'online'.
            - We'll get into how we could go about handling this 'app connectivity state' in a bit.

- But this brings into question the mechanics of the queuing would be handled.
    - My first thought was to have all of the `OfflineRequestSlice` actions have a certain prefix in the type, so that the queuing authority could `take` them all using a regex.
    - But then how does the queue know how to fire the request? We'd need to do something like register each and every `OfflineRequestSlice` with the queuing authority so that it could invoke their **effect** wrappers.
    - But wait, *they're wrappers*. Registering the `OfflineRequestSlice` doesn't really mean anything when the sagas being wrapped aren't known to the request slice.
    - So... maybe what actually gets queued is an *action* to fire that triggers the **effect** wrapped saga? But how would that work without registering all of the `OfflineRequestSlices`? Listening for all the `request` actions doesn't do us any good when all they have is a payload. But what if we added *more* than just a payload to the standard `request` action creator?
        - We could add something like an `effect` property along with `payload` using createAction (in a `meta` property like is standard?), and then the queue would know what action type to dispatch to fire the **effect** saga.
        - But if we're only dispatching actions, then we lose the ability to wait for completion/failure of the effects. So... we come back to basically what `redux-offline` does and just have meta properties for all of the states: `effect/start`, `effect/success`, `effect/failure`.
            - This means we're going to have a lot more actions to properly model everything → we need start/success/failure actions for both **effect** and **commit** (and **rollback**??).
    - So the flow might look like this:
        - Dispatch `request` action (with a `meta: {effect}` property)→queuing authority picks and queues `request` action → queuing authority dispatches the `effect/start` action → waits for either `effect/success` or `effect/failure`. Since

`rollback` is handled by the `OfflineRequestSlice`, I don't know what the purpose of `effect/failure` is here, since the only use I can think of (from the perspective of the queuing authority) is to re-queue the `request` action.

    - Although, the process for dispatching the `effect/start` actions would have to also do a network connectivity check. I suppose if, during the processing, network connectivity goes down, then the `request` actions will have to be re-queued.

- Seems like we're making the decision to move all of the actual error handling and retry logic into the queuing authority, which means that the actions that get queued will need not only a `meta: {effect}` property, but also `meta: {rollback}` property so that the queuing authority knows which actions to emit for rollback.

    - We're going to have to be really careful with how we define the logic for dealing with failures; we don't want to get into a situation where something like one request fails, but the one following it succeeds, but since both requests acted on the same property, things will now be applied in the wrong order.

    - As such, I think we're going to need to make sure that requests must execute in the order they were queued.

Network Connectivity Discovery

- The network discovery functionality probably makes most sense to make in with whatever the "Queuing Authority" component becomes.

- It basically just needs to handle the following:

    - Every so often (I'm thinking every 30 seconds) ping a 'healthcheck' route on the backend server.

    - Update a boolean piece of state depending on whether or not the ping was successful.

    - That's it.

- The queuing authority can then react to updates that the network discovery service publishes, by processing the queue of actions if the connectivity state changes from offline to online.

- I'm thinking that this can be implemented using an eventChannel or even something as simple as a saga that just runs every 30 seconds (e.g. redux-saga/redux-saga#1610).

    - Honestly, as cool as using channels would be, it's probably easier to just do it with a regular old infinite loop and delay...

Design Brainstorming (v2)

The following takes all of the decisions made during the v1 brainstorm and brainstorms some more.

OfflineRequestSlice

- Having thought about **rollback** some more, how on Earth are we going to handle rollback of destructive changes like a *delete* or *update* operation? Obviously, rolling back an *create* operation is trivial enough, but destructive operations lose data once they've been committed to the store.
    - As a result, I think we're going to need to send along more data when queuing up a request. In particular, instead of the `payload` containing just the data that was committed, it should also have the old state that can be rolled back to.
    - For example, the `payload` for a *delete* request could look like the following:
        - `{oldState: {resource being deleted}, newState: {id of resource being deleted}`
    - Here, `oldState` would just be whatever data is needed by the **rollback** saga to revert a change made by `newState` (so it would defer depending on what type of operation is being performed -- for example, a *create* operation wouldn't necessarily need an `oldState` to be able to perform a rollback; it would just need the `newState`, which would have the ID that can be used to lookup and remove the object).
    - However, including the `oldState` as part of the queued request info could introduce problems relating to state data and out-of-order modifications, if things are rolled back in weird orders (particularly with *update* operations).
        - Updates will just need to be really particular about their `oldState`; it'll have to be really specific with just the ID of the object being updated, the property being updated, and the old value. Anything more could introduce problems.
        - Although even that gets sketchy if an *update* is rolled back after a *delete* operation goes through...
            - Wait, but that shouldn't be impossible if we strictly enforce that requests are executed (or in this case, rolled back) in the order that they happen? So... maybe it'll be fine?
            - Should still handle something like this anyways (although I guess it generically gets handled by the error handler as part of the **rollback** wrapper and the `rollback/failure` action).

    - Still don't know what to do if/when we ever hit a `rollback/failure`. Like, just chuck away the request?

- Also, have we been making the assumption that **rollback** only rolls back the local store, as a result of a failed request? Is there any case where we would need to roll back the backend server? There probably is. If that's the case, does that mean that **rollback** *also* needs retry logic???
    - I think, for the sake of simplicity, we'll just ignore this for now.

OfflineRequestManager

Formerly known as the "Queuing Authority", this combines the functionality needed for the queuing authority and the network connectivity discovery service.

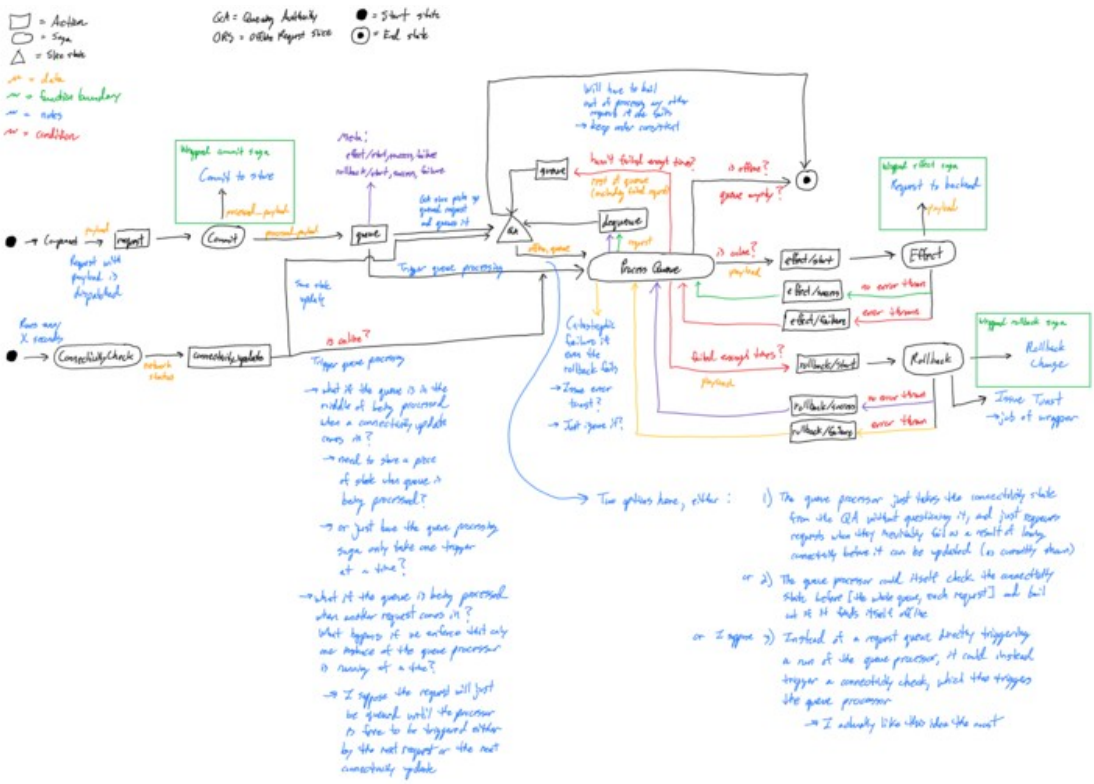
- Whenever the queue processing is happening, when we need to wait for `effect_success/failure` or `rollback_success/failure` to happen, we also need to have a timeout running as a tertiary effect, as another form of error handling. I think a 5 to 10 second timeout is reasonable for most requests (although the timeout should probably be increased as the number of retries increases, ala exponential backoff).
- To address the problem of a request being queued while the queue processor is running, maybe instead of the processor taking the whole queue at one point in time, it just pulls the head of the queue and processes one at a time until either something fails or the queue is empty. This way, if a new request is queued at the end of the queue, the processor will just pick it up like nothing had happened.

Other

- Multiple offline devices using the app and modifying the same resource. What happens? What *should* happen?

Flow Diagram

Here's a really rough diagram that more or less covers how this is all going to work (along with some more brainstorming):



Design (Finalized)

The offline first functionality will fundamentally be implemented as two pieces: `OfflineRequestSlice` and `OfflineRequestManager`.

OfflineRequestSlice

- Will be a more-or-less drop-in replacement for `RequestSlice`.
- In order to accommodate running the external requests (for now on known as **effects**), the entire lifecycle has been extended to three phases (each handled by one *saga* each):
  - **commit**: Data/changes are immediately applied to the store in order to give the user the fastest optimistic UI updates; fires off the action that causes the `OfflineRequestManager` to queue the **effect**.
  - **effect**: The external request to the backend is queued so that it can potentially be stored for later execution if the user isn't currently online.
  - **rollback**: Error handling for rolling back changes to the store done as part of **commit** if the **effect** fails to be properly executed.
- These new phases mean that new *actions* are required to handle all the possible states that a request could be in:

  - `effect/start`, `effect/success`, `effect/failure`
  - `rollback/start`, `rollback/success`, `rollback/failure`

    Along with the existing actions:

  - `request`, `success`, `failure`
    - These actions now only pertain to the **commit** phase of a request.
    - These actions are kept un-namespaced in order to maintain some level of backwards-compatibility with the `RequestSlice`.
  - `clear`

- Although there are new phases, the *state* for the whole `OfflineRequestSlice` (i.e. `{loading, error}`) will remain the same.
  - This can be done because the **effect** and **rollback**'s loading/error 'state' is handled outside of the normal UI flow (by the `OfflineRequestManager`).
    - The visual indication for **effect** and **rollback**'s `loading` state will come as part of Epic Phase 3.
    - The visual indication for their `error` state will come in the form of toasts (for now, as part of Epic Phase 1) and later as an improved indicator as part of Epic Phase 3.
  - As such, the `{loading, error}` state only pertains to the **commit** phase.
  - Obviously, since **commit** is supposed to be very fast, there won't *really* be a point to having the `loading` state; however, it seems fine to leave it around for backwards-compatibility reasons.

OfflineRequestManager

- Is the authority responsible for queuing and executing **effects** from the `OfflineRequestSlices`.
- Since all it queues and processes are actions, this service is completely decoupled from the logic of how the **effects** or **rollbacks** are executed.
- However, it does need to be smart about how the queue is processed, in particular regards to handling **effect** errors, setting up retries, and setting appropriate timeouts.
- This service is also responsible for managing the network connectivity state of the app (i.e. checking whether or not the app has a connection to the backend server).
- There will be one *action* that this service exposes for others (i.e. `OfflineRequestSlice`) to use:

    - `enqueue`
        - Adds an **effect** to the end of the queue for processing once the app is online.
        - Triggers a connectivity check so that the queue processor can be triggered.
        - `payload` is the data returned from the **commit** saga (i.e. the data to persist to the backend).
        - `meta` contains the actions that are dispatched to control the effect/rollback lifecycle:
            - `effectStart`, `effectSuccess`, `effectFailure`
            - `rollbackStart`, `rollbackSuccess`, `rollbackFailure`
- There will be three additional *actions* that will be used internally by the service and its sagas:
    - `dequeue`
        - Removes the head element from the queue.
        - No `payload`.
    - `increment_attempts`
        - Adds 1 to the `attempts` of the given effect.
    - `update_connectivity`
        - Updates the store state with the latest connectivity state.
        - `payload` is the latest connectivity state.
    - `process_queue`
        - Causes the queue processor to start working.
        - No `payload`.
- This service is, obviously, also going to need some of its own store *state*, as follows:
    - `effectsById`
        - An index of **effect** objects indexed by an ID that is auto-generated for each **effect**.
        - Alongside the **effect** object will need to be some extra metadata:

        - `attempts`
            - The number of times the **effect** has been executed and failed.
        - `insertionTime`
            - The date/time when the **effect** was inserted into the queue
                - In the current design, this isn't actually used for anything, but it just feels like something that'll be needed in the future.
        - `lastAttemptTime`
            - The date/time when the **effect** was last executed.
                - In the current design, this isn't actually used for anything, but it just feels like something that'll be needed in the future.
    - `queue`
        - The queue of **effect** IDs, with the head at the end of the array and the tail at the start of the array (as a result of how `unshift` and `pop` work).
    - `isOnline`
        - Whether or not the app currently has a connection to the backend server.
- We're also going to need a handful of *sagas* to handle everything:
    - `processQueue`
        - Triggered by the `process_queue` action.
        - Loops through the queue, taking the head **effect** one at a time, processing it until it either succeeds (and can be dequeued), it fails and has to be enqueued again (with a longer timeout next time), or it fails and has failed enough times to trigger a **rollback**.
        - Only one instance of the saga running at any time (i.e. it uses `take` in an infinite loop instead of `takeEvery`) since we need to ensure that only the latest **effect** is being processed at any one time.
    - `connectivityCheck`
        - Triggered by the `enqueue` action.

        - Triggered by running at a set interval.
        - Constantly checks for a connection to the backend server at a set interval (e.g. 30 seconds); fires off the `update_connectivity` action to update the state.
        - An optimization could be made so that it only fires an `update_connectivity` action when the state is 'online', or when moving from 'online' to 'offline'.
            - We need the 'online' updates to always be fired so that the queue processor can be triggered.
            - But we can limit when 'offline' updates are made since 'offline' updates made while the system is already 'offline' don't do anything (for now?).
        - Idea: What if we listened for the native `online` and `offline` events (i.e. Navigator.onLine - Web APIs | MDN) and used them as additional triggers? This way, we could immediately trigger a check when the user, for example, actually does reconnect to a network.
    - `triggerQueueWhenOnline`
        - Triggered by the `update_connectivity` action.
        - Checks if the connectivity state is 'online' (from the action `payload`); fires off the `process_queue` action if it is.

During Implementation Thoughts (March 2020)

- So after thinking about some of the retry logic out loud, I have come to conclusion that what we have -- where we retry X times and roll back if they all fail -- is completely inadequate for what we're trying to achieve. Here's why.
    - Somehow, I've failed to account for the very base case for offline-first: using the application entirely offline! Or at the very least, offline for an indeterminate (but potentially very long) amount of time.
    - As such, the system of retry X times and roll back makes no sense because the system could be offline for as long as X retries, roll back, and the user will be like "WTF" because they expect that it'll sync once they're back online.
    - As such, the retry mechanism will have to scrapped.

^ This my friends is what happens when you come back to try and finish implementing something after a couple months.

Of course I *did* account for the base case of being offline. The retry logic currently only applies to when the **effect** fails. Effects won't even be tried if the system is offline.

So... **FALSE ALARM! ALL IS WELL**.

Retry and Undo Interaction

- Something that I didn't consider is the interaction between the retry algo and the undo algo when deleting e.g. accounts or transactions.
    - Currently, if something like a transaction gets deleted, gets **committed**, errors out during the **effect**, and then the user invokes the undo operation before the **effect** has gotten through all of the retries, then the system is in a really wonky state where it still has a deletion **effect** in the queue but the user no longer wants it.
        - Not to mention that the undo will trigger a creation **effect** that will error out if for some reason it can get through but the deletion can, because it'll have a duplicate ID.
    - As such, the **effect** still has the chance of going through during the retry algo and causing the user's decision to undo to be.. undone.
    - As a result, I guess we need some way to manually remove certain **effects** from the queue.
        - Or, we need to modify the `undoableDestroy` to be offline-first friendly. Currently, it progresses through the undo steps by just waiting on the success of the **commit** steps, but this obviously isn't good enough for our new use case.
            - Actually, having walked through the below flows, I don't think this is the case anymore.
        - Theoretically, I believe the happy path flow would be something like this (for the offline case):
            i. Undo algo starts.
            ii. Destroy commit is successful.
            iii. Queue destroy effect.
            iv. Show user the undo toast.
            v. User chooses to undo.
            vi. Create commit is successful.
            vii. Queue create effect.

viii. User comes online.

ix. Destroy effect is successful.

x. Create effect is successful.

- However, what if *just* the destroy **effect** fails while the user is online? Then the following happens:

  i. Undo algo starts.

  ii. Destroy commit is successful.

  iii. Queue destroy effect and it fails.

  iv. Show user the undo toast.

  v. User chooses to undo.

  vi. Create commit is successful.

  vii. Queue create effect.

  viii. Destroy effect fails all retries and triggers rollback.

  ix. Rollback re-creates the transaction, but then silently fails because the transaction already exists but doesn't trigger an error because there is logic to prevent creating duplicate transactions at the store level.

  x. Create effect fires and fails because the transaction was never deleted on the backend and thus a duplication error occurs.

  xi. Create effect fails all retries and triggers rollback.

  xii. Rollback deletes the transaction.

    - This is obviously pretty freaking bad. The user would be *so* confused. Hell, I was confused just trying to reason through the whole flow.

- So, how do we fix this? Well... Ideally, we'd be able to queue the destroy and create effects only at the end of the whole undo algo. However, since the user expects that the transaction actually gets deleted once the undo toast is present, this can't be done; the destroy effect has to be queued before the toast is shown.

- Well, the way I ended up fixing this was potentially more roundabout or more straightforward, depending on how you look at it.

        - Basically, all I did was enable the error to be passed to the `rollback` saga so that it can do some error handling before performing the actual rollback.
        - For creation, this meant checking the error for a Feathers error message indicating that the transaction was a duplicate.
        - For deletion, this meant checking the store for the transaction before trying to re-create it and issue an error toast.
    - This seems to solve the problem of this error flow and adds some extra error handling for other potential cases, so I think this is good.

Duplicate Fetch Effects after a Failed Effect

- When an effect fails, the user can refresh the page to have the manager fire the queue processor again.
- However, since the page got refreshed, this triggers another round of the app boot fetch effects to be queued.
- If the user keeps refreshing and the first effect keeps failing, then these app boot effects will just keep piling up.
- As such, I think it would be useful to introduce the concept of a 'unique' or 'non-duplicatable' effect that could be applied to these fetch effects.
    - This way, when the multiple of same type of effect get queued, the manager can just throw out the duplicates.

The merit of having 'fetchAll' as an 'offlineRequestSlice'

- Obviously, as it has been coded, `fetchAll` effects for `accounts` and `importProfiles` are offline request slices.
- However, this was more so for convenience of co-locating these slices with the other (actually offline-first) slices.
- A thought I had was "why can't these things be just regular request slices? Without the 'commit/effect/rollback' lifecycle, but still the 'loading/error' lifecycle?".
- Then I thought, "well, I wouldn't have a separate `accountsRequestsSlice` and `accountsOfflineRequestsSlice`, cause that'd be quite messy".
- Then I thought, "well, maybe I should create a helper function that allows the option to specify a set of offline request slices and then a set of regular request slices. Maybe something like they are just separate args."
- And then I thought, totally unrelatedly, "hmm, the naming for these things is kinda inconsistent. Internally, there is the concept of a `RequestSlice` and a `RequestSliceSet`. However, externally, we name them as `RequestsSlice` to refer

to a `RequestSliceSet` (i.e. `accountsRequestsSlice` actually refers to a `RequestSliceSet`, which is many `RequestSlices`, even though the variable is named in the singular). So either we should change the external names to `accountsRequestSliceSet` or change the internal name to `RequestSlices` and then the external to `accountsRequestSlices`. Hmm..."