Story UFC-310: Stripe Integration

The following document will go over the design of the Stripe integration, as outlined in Story ticket UFC-310.

Acceptance Criteria

For reference, the following are the acceptance criteria for this ticket:

- After signing up (i.e. creating a user account), I want to be able to select a plan to pay for uFincs.
- Once I select a plan to subscribe to, I am redirected to Stripe Checkout to pay for the subscription.
- Once I have successfully paid for the subscription, I am returned to the uFincs dashboard.
- If I cancel/fail to provide payment for the subscription, I am returned to uFincs to the screen where I pick my plan.
- I should not be able to access my account if my subscription has expired/I haven't paid.
- I should be notified when access to my account will be limited due to non-payment.
- I should be notified when my account will be terminated due to non-payment.
- I should be able to access a Billing Portal to manage my subscription.
- I should be able to cancel my subscription from the Billing Portal.
- I should be able to view my billing history from the Billing Portal.
- I should be able to update my payment method from the Billing Portal.

Design Brainstorming

OK, so there's a good amount of work to be done for this story. Let's break this down into things that need to be done for the Backend and things for the Frontend.

Backend

Here's a todo list:

- Bring in the Stripe package.
- Setup a route for creating a Stripe Checkout session.
- Setup a route for the Stripe webhook.
- Create a `subscriptions` table that stores the following info:
- Associate the `users` table to the `subscriptions` table
- Modify the authentication service so that users without a valid subscription aren't authenticated.
- Modify all authenticated calls to check if the user has a valid subscription.
    - This is to prevent users from logging in before their subscription is over but still making API calls after it finishes.
- Store the Stripe product IDs, price IDs, etc in a JSON config file.
- Add Stripe secrets (primary, webhook, etc) to encrypted secrets file.
    - In the config where they get loaded from the environment variables, make sure to check for the production branch/URL (*not* the production `NODE_ENV`) so that the branch deployments don't use live Stripe mode.
- Create a script for creating the Stripe products/prices so that we can port between accounts easily.
- How on earth are we going to setup webhooks for each branch deployment...
- Need some way to enable/disable subscriptions across the whole app (i.e. a feature flag) so that we can still run the app without subscriptions, until we can get live Stripe working.

Subscriptions Table

There's a lot of stuff we *could* store in this table. By my count, it could be any of the following:

- Stripe customer ID - `customerId`
- Stripe product ID - `productId`
- Stripe price ID - `priceId`
- Stripe subscription ID - `subscriptionId`
- Stripe subscription status - `status`

- Subscription period start - `periodStart`
- Subscription period end - `periodEnd`
- Credit card expiry, brand, last 4 digits?
- Address, ZIP?

However, I think we should go with only the stuff we *critically* need. In particular, let's *not* record any of the card information (even if we're allowed to) and none of the address info (actually, we might need this for tax purposes... but we can always just get it from Stripe directly). We really only need the Stripe IDs and info on when their subscription ends/is valid, for the purposes of validating their account when authenticating.

I wonder if there's merit in storing the Checkout session ID as opposed to/in addition to any of the other information... I suppose, if we have all the other information, we *should* be fine, but... Hmm. Well, let's start with the minimum and see if anything terrible happens.

As such, let's start with:

- `customerId`
- `productId`
- `priceId`
- `subscriptionId`
- `status`
- `periodStart`
- `periodEnd`

For reference, Suggested database architecture for my first SaaS with Stripe?.

- **Update**: Of course, most of the way through development, I finally find a good article detailing a Stripe integration: Implementing API billing with Stripe

Also, need to 'expand' a Checkout session to get the line items: Stripe get product_id / price_id from session object

- This is relevant when wanting to check that the lifetime deal was purchased, since there's no subscription associated with the one-time payment of a lifetime deal.

Feature Flags

So there's 2 ways we can go about handling error flags:

i. Store the flags in a config file and just read from there.

ii. Store the flags in a database table.

1 is simpler, 2 gives us the flexibility to enable/disable flags on a per-deployment basis.

Technically, we could have an enhanced version of 1 where the config specifies whether the flag is enabled for, e.g., local, staging, or production (where 'staging' = every non-master branch deployment). That's probably a good compromise. So... let's go with that.

Although, putting them in the database means that we can flip flags without having to re-deploy. But it *also* means that the functionality of the app is controlled by something that isn't version controlled (i.e. in code).

Hmm... let's stick with just a config file for now; simpler.

And by config file, I guess that could be the `feathers` config files? But they don't give us the option for a staging environment; only prod or not prod. So... I guess we'll just do it ourselves?

- **Update**: Coming back to this now... I don't know *why* I wanted feature flags that enabled things for staging and dev but not for prod. Maybe because of the webhook situation that was solved below? IDK, but I think we should be fine with just on/off flags.

In terms of how to use the feature flags, I guess they could be wrapped in a Feathers service? That way, there's a consistent interface for fetching them? (as opposed to just reading in the file directly)

Making a Feathers service out of them would also give us a convenient way for passing the flags down to the frontend, as opposed to having a duplicated feature flag file in the frontend code. Having the feature flags dynamically loaded would also enable us to more easily toggle feature flags on the frontend, considering the frontend app is a service worker with offline caching -- having the flags always pulled down on means that we can 'break through' the cache more easily.

In terms of how to actually use the feature flag for subscriptions, here's what we need to guard to turn off subscriptions:

- The authentication service, so that it doesn't check for subscription status when authenticating.
- All authenticated calls, so that they don't check for valid subscription before completing the call.

Pricing the Plans

OK, so it seems like we've hit a *small* snag...

My original plan was to have a monthly plan, an annual plan, and a 5 year plan.

I wanted to price them as $20/month, $120/year (= $10/month), and $400/5 years (= $6.67/month).

However, it seems like I've hit a limitation of Stripe: recurring plans (i.e. `prices`) can only have a recurrence period up to 1 year. So... we can't do a 5 year plan using Stripe's default stuff.

OK, that's fine, I guess we'll just... charge them $400 one time and then manually charge them again every 5 years? I mean, that *would* work. It's just that some of the Stripe pages (namely Checkout and the Customer Portal) wouldn't reflect this accurately.

I mean, it's also not like we'd need to get the logic for charging them after 5 years correct right away, considering we'd have... 5 years to do it.

But then I kept thinking, "why not just make it a lifetime deal?" I mean, lifetime deals are a *lot* more common for other SaaSs, so that'd be a more familiar deal for people. Plus it would simplify our backend a logic a bit (although it's still more complicated than if we could just charge them every 5 years using Stripe...) -- just check if they've paid and they're good to go. Maybe add a `lifetime` flag to the `subscriptions` table to simplify this.

I guess as long as we make it clear that it's the lifetime of the *app*, (not the customer's lifetime), then I think we should be good to go.

Plus, with a lifetime deal, I think we could up the price to $500 (what I had originally wanted), so that should be good as well.

Now it's just a matter of deciding *how* to run the lifetime deal. I'm leaning towards X number of people can sign up for it over running the deal for an unlimited number people but a limited period of time. I'm thinking 100 or 150 people would ideal. If we max out 150 lifetime deals at $500 a piece... $75,000 USD = close to $100,000 CAD right now... that's a pretty good capital infusion.

Plus, there's nothing stopping us from running this deal in the future.

So... I think we should just switch to a lifetime deal instead of a 5 year deal, change the price to $500, and hope that people using a privacy friendly personal finance tool don't ask too many support questions.

Not to mention that a lifetime plan makes the accounting easier, since we don't need to defer it at all. Which isn't better from a SaaS accounting perspective, but it is easier from a tax accounting perspective.

Also, lifetime can be thereotically shorter than 5 years, while also being much less of a legal burden.

update: Having thought about it some more, I think $400 USD is more fair than $500, so... let's go with that.

Note: Apparently AppSumo is a marketplace for lifetime SaaS deals. Didn't realize that's what they're for...

Some **research** on lifetime deals:

- https://www.indiehackers.com/forum/ask-ih-pro-or-against-life-time-deals-d99b582b57
- Indie hackers, I don't understand your Lifetime deals
- https://www.indiehackers.com/@guillaume_lemlist/6be00cd8af
- To Run, Or Not To Run A Lifetime Deal: A True Kick SaaS Story
- Kick SaaS: Lifetime Deals
- Another AppSumo like platform: New Home
- Should you launch your SaaS product on AppSumo? Read our story before deciding.

Modifying Authentication

OK, so originally I wanted to change the `authentication` service so that users without an active subscription wouldn't be authenticated, buttt.... I also wanted to always redirect users without an active subscription to the `Checkout` scene. So... What do we do?

There's a couple scenarios we need to handle here:

- Users who have had a subscription and then stopped paying it.
    - These are theoretically the users who wouldn't be allowed to authenticate, but now that I think about it...
- Users who just signed up and don't have a subscription yet.
    - These users should be allowed to authenticate and just get redirected to the Checkout scene.

Now that I've actually thought about this more, users in the first case should really be in the same case as the second case users: if their subscription is expired, they should just be told to subscribe again.

So now that we've established that inactive subscription users can at least authenticate past the login screen, what do we do?

We essentially just need an `hasActiveSubscription` flag, so that, when the flag is set, they just get redirected to the Checkout scene.

Of course, we need to guard against malicious users manually changing the flag, so we still need to prevent all other authenticated API calls from allowing these users to get/change data.

So I think that's our solution.

Per-Branch Webhooks

So, I came across a new problem... How the heck are we gonna register the per-branch environments as webhooks on Stripe so that we can properly test things in the per-branch environments?

Well there's 2 solutions that I've come up with so far, and they both concern creating more than one Stripe account:

i. We create a new Stripe account for each branch. This way, it gets its own set of products, prices, subscriptions, and customers. We just run each non-master Stripe account in test mode. This is the most 'branch separated' option.

ii. We have two Stripe accounts: one for prod and one for testing (i.e everything else). In the test account, for each new branch, we would just register a webhook (and unregister it once the branch has been removed).

I think we just go with 2. because it's simpler. Can just hard-code the product/price/whatever IDs for the two accounts and we should be fine. Sure, that second test account will end up getting quite cluttered, but whatever...

In either case, the real problem is that we need to be able to hook into `kubails`' namespace cleanup command so that we can run a custom script. There's 2 ways I see going about this:

i. Return the removed namespaces (i.e. branches) from the command, then use that output as the input for the script that removes the Stripe webhooks.

ii. Add an option to pass in a custom script and have `kubails` execute the script with the removed namespaces.

It's probably easier (and more unix-y) to go with 1., so let's do that.

OK, so we now need to setup a Stripe account with a second test account. That means we'll need to script the creation of the products and whatnot. I've been thinking whether or not it'd be worth having Terraform control this (since there *is* a provider for Stripe, albeit a third-party one). However, I don't really want to have to tie passing the Stripe API token into Terraform, so I think it'd be best to keep them separate and just have the Stripe stuff as a standalone set of scripts.

- Although, I wonder if there would be merit in having a separate Terraform setup for handling just the Stripe stuff. That way, it wouldn't interfere with `kubails`.

How to Deploy Per-Branch Webhooks

OK, so here's where we run into trouble... when we create the per-branch webhooks in the test Stripe account, how do we handle the Webhook Secrets that need to be passed to the Backend service? Because each instance of a webhook (i.e. each url/branch) will have its own secret.

Let's start with the fact that there are two problems here:

i. How do we persist the webhook secrets?

   - This is a problem because, when we create the webhook using the `stripe` CLI, we receive the secret when we create the webhook, but we *can't* query for it on subsequent builds (for obvious reasons). So how do we get the secret to inject it on subsequent builds?
   - I'm thinking we could use GCP Secret Manager for this. Store the webhook secret using the branch as the key and then we can query it up. If it doesn't exist, then it just means we haven't created the webhook yet.

ii. How do we inject the secret into the Backend service?

   - This is something I'm less sure about... Currently, we don't have a mechanism for injecting environment variables from the build process. Or at least, we don't have a *good* one.
   - Technically speaking, we *could* inject the webhook secret by adding an environment variable to the `kubails.json` config for the Backend service and then rewriting the value in the build pipeline. This way, it would be injected when generate the manifests.
   - Hacky? Absolutely. But it *would* work.
   - And you know what? I think that's good enough for me.
   - So I think we can hardcode the default webhook secret to the one used when running `stripe listen` and then just overwrite it with the environment variable if it isn't whatever default placeholder value we choose.

Now all we have to worry about is making sure the secrets get deleted when we delete the webhooks.

BTW, in order to use the Stripe CLI in the build pipeline, we'll need to specify the API key as a substitution variable. As for actually *running* the CLI, I guess we'll need a Docker image that has the CLI installed? But also the GCP CLI? Hmm... Should we just bake it into the Kubails image? Or create a custom one? It's probably easiest to just bake it into the Kubails image... We're gonna need to re-deploy Kubails anyways because of the changes we need to make to get the `cleanup-namespaces` command to output the cleaned up namespaces.

- An alternative would be to write the webhook creation script as a node script in the Backend service, run the script using the built Backend image (that has the `stripe` package installed), have the script output the secret (as a console log) so that it can then be read into a script that is run from the Kubails image to store the secret into Secret Manager using the GCP CLI that the Kubails image has access to.
- That... could also work.

Also, we'll need to add the API keys for both the test and prod Stripe accounts as substitution variables and switch between them based on the branch.

Migrating Existing Users to Subscriptions

OK, so migrating users is kinda of a pickle. Because there are 2 kinds of users we could be migrating:

- Users who have an account but don't have a subscription.
- Users who have an account *with unencrypted data* but don't have a subscription.

The first kind are basically taken care of with the current implementation. The second, however... are not. Why? Because the current encryption migration process happens as part of the login process, right after logging in. However, if a user doesn't have a subscription yet, then we can't push their migrated encrypted data to the backend, because they'll be denied by the "must have subscription to do any requests" hook.

And this is quite problematic since all existing prod users are the second type.

Now, this isn't actually that big of a problem, since all the existing prod users (besides myself) are either (who has a single transaction), and everybody else (unknown people who we also don't care about).

So... it would be a pretty plan to just wipe out the data for all the other users in prod.

However, I think we should still try and make this work. Currently I think there's two ways we can go about this:

- Move the encryption migration process to after checkout (once the user has a subscription)
  - Not trivial
- Or modify the `authenticate` hook to ignore the subscription requirement if a user doesn't have an edek/kek salt.

Let's see if we can the second to work...

And nope, too much work.

So yeah, let's just wipe prod. This will also enable us to remove all of the encryption migration code, which is a plus.

Frontend

- Update the CSP to allow Stripe stuff.
  - See https://stripe.com/docs/security/guide for more details.

- Create the `Checkout` scene where the user chooses their plan and gets whisked away to Stripe Checkout.
    - Don't forget that it should accept a URL query param to pre-select the plan from the marketing site.
- Add the Billing section to the Settings that redirects the user to the Stripe Billing Portal.
- Create the `BillingService` to encapsulate the Stripe calls.
- Create the `billing` slice and sagas.
- Need some way to enable/disable subscriptions across the whole app (i.e. a feature flag).
- Figure out how the hell we're gonna update the e2e tests to support this...
    - I'm pretty sure Cypress just breaks when navigating to another domain, so this might be the first time we have to actually check if we're running in Cypress to get around certain functionality...
    - Also, we could update the seed data so that the test account(s) have a lifetime subscription. That could also help.

Accessing the Checkout Scene

Sigh... OK, so I basically figured out what I wanted to do before writing this section, but I'll write down my thoughts here anyways.

Basically, I had created a `hasActiveSubscription` flag that gets passed alongside the `user` object during authentication. However, I found that a simple true/false flag wouldn't be enough to convey how I wanted the app to work.

So how did I want the app to work? Well, it essentially comes down to *when* the user can access the `Checkout` scene. Initially, I had planned so that the would get redirected there whenever they didn't have an `active` subscription. But thinking about it more, I decided that it would be a poor user experience that, if their subscription goes past the end of their paid-for period (i.e. they miss a payment), they would just be kicked to the `Checkout` scene. This just didn't make sense to me.

As such, I decided that, alongside the `active` and `inactive` states for a `subscription`, there should also be an `expired` state. This state just means that the current date is beyond the end of the subscription's end period. When a subscription hits this state, the user should still be able to access the app, but in a *read-only* state. This way, they still have a chance to access the settings and access the Billing Portal to update their payment details (if, for example, they *want* to continue their subscription and maybe their card expired or something) or to cancel their account (if they no longer want to pay for a subscription).

In any case, I think we're taking the more user-friendly approach here by letting former-customers still access and export their data if they so please. As long as it's in a read-only mode, it doesn't really impact us (since they can't make calls to the backend, beyond authentication calls) so they should have the right to at least view and export their data.

I think this is a much fairer policy/UX than just dumping them back to the `Checkout` scene and holding their data hostage until they pay again.

Feature Flags on the Frontend

OK, so how do we want to handle feature flags on the frontend?

Well, above in the backend planning section, I decided that I wanted to always pull down the flags from the Backend on every page refresh. That's about as far as I got, though.

So how should we handle these flags on the frontend? Well, it's a piece of state, that is needed globally, so I can only assume that the best place to put them is in a Redux slice. Additionally, since it's a piece of state we want to always fetch (regardless of even if the user is logged in), then we probably also want a corresponding saga to handle the data fetching.

In terms of accessing/using the flags, I suppose a selector makes the most sense, along with an associated hook wrapped around it for any component level shenanigans.

And... that's it, I think.

Now, in terms of what needs to be wrapped around in the subscriptions feature flag:

- Modify all of the subscription selectors from the `user` slice to be cross slice selectors that reference the feature flag and default to usable values when subscriptions are disabled.
    - e.g. `selectSubscriptionEnablesAppAccess` should default to `true` when the flag is off, `selectReadOnlySubscription` should default to `false`, etc
- Modify the `authSignUpFromNoAccount` saga to only push to the `Checkout` scene when the flag is on and to handle the data migration when the flag is off (cause it can't rely on the app boot process to do when the user gets redirected to the app right away).

I... think that might actually be all? Because pretty much everything that relates to the subscription 'feature' is already controlled by the existing selectors, so just making those selectors also rely on the feature flag should be good enough.

E2E Tests

Yeah, so since we can't test the complete subscription flow with Cypress (since Cypress ceases to work once the domain changes -- which is problematic since Stripe Checkout and Customer Portal are on a different domain), I've decided we're just *not* gonna test *anything* related to subscriptions with e2e tests. How? By checking for the `window.Cypress` flag in

the `subscriptions` feature flag selector -- basically, we just turn off subscriptions in Cypress.

Is this bad? Well yeah, obviously. But really, the only things we're missing out on testing are the Checkout scene, the Billing section of the Settings, and the subscription specific redirection logic. Which, from the Frontend's perspective, is all *pretty* simple, so things shouldn't break *too* badly... It's the backend stuff that really requires testing, but since that also requires listening for events and a bunch of database stuff... meh... Screw testing one of the most important parts of the app!

Stripe Configuration

Product + Prices

- Standard
  - $20/month
  - $120/year
- Founder
  - $500 one time

Billing → Subscriptions and Emails

The following are changes I made to the email notification settings.

- Send emails about expiring cards → turn on
- Send emails when card payments fail → turn on
- If all retries for a payment fail → cancel the subscription
- If all retires for a payment fail → leave the invoice as-is
- If a dispute is opened → leave the subscription as-is
- Send a Stripe-hosted link for cardholders to authenticate when required → turn on
  - Send reminders if 3D Secure isn't completed → use default settings

Billing → Customer Portal

- Billing history → turn on
- Billing information → turn on → Email address, Billing address
- Payment methods → turn on
- Cancel subscriptions → turn on, cancel immediately, don't prorate

- Update subscriptions → turn on, prorate subscription + issue invoice immediately
- Products → Standard → Monthly + Annually
  - Can't switch to a lifetime 'subscription', since it's not a recurring plan...
- Business information → Links →
  - Terms of Service → fill with something
  - Privacy Policy → fill with something

Payments → Checkout Settings

- Turn off "Use Apple Pay" and "Use Google Pay"
- Branding Settings → TODO

Business Settings → Emails

https://stripe.com/docs/receipts#receipts-with-billing

- Email customers about... Successful payments → turn on
- Email customers about... Refunds → turn on
- Configure support email and email domain

User Profile

- Turn on 2FA

Update: Further modifications to the Stripe config will be noted in Stripe Config.

Component Breakdown

Atoms

- N/A

Molecules

- SubscriptionPlan

Organisms

- SubscriptionPlanForm (OverlineHeading + SubscriptionPlan + Divider + SubmitButton)

Scenes

- Checkout (SubscriptionPlanForm)

Tasks

- Built the Stripe integration.