Task UFC-342: Docker/Kubernetes Improvements

The following document is less of a pre-design document and more so a post-design document. Why? Cause I'm writing it to record the reasons I made certain decisions as part of the task, UFC-342: various improvements for the services from a Docker/Kubernetes POV.

Design Decisions

Separate Dev vs Prod Dockerfiles

So this is probably a hot take, relative to what I'm used to... but I've decided to have separate Dockerfiles for dev and prod.

Why? Primarily to reduce the attack surface area of the prod containers.

See, I've switched to using a multi-stage prod Dockerfile where the final stage uses the Node `distroless` image. Basically, it doesn't have any shell, no shell tools, basically nothing but the Node runtime itself. As such, there's significantly fewer things to attack vs the regular alpine Node image.

Also, the containers now run as non-root users. A consequence of this was that the frontend/marketing services could no longer bind to port 80. Which isn't really a big deal considering it doesn't matter what port they expose when it just gets rebound by the service. So now those services always expose over port 3000 (just like in dev).

That is to say, our prod images are now theoretically significantly more secure than our old images (which are now our dev images). An added benefit to this is that the prod images are also smaller.

So how does this work then? Well, our regular `docker-compose.yaml` config now explicitly uses the `Dockerfile.development` files (again, these are our old images). This way, we can have access to all our dev tools (particularly `npm`) in dev.

Yeah that's right, our prod images don't even have NPM. A consequence of this is that we had to manually write some Node scripts to invoke what were previously just NPM commands (particularly, `db-migrate`and `db-reset`).

However, we can still use `docker-compose` to bring up the prod images locally. There's a separate config for it and a new `make start-prod` command. I've verified that the e2e tests still (mostly) work against it (the only things broken are the no-account sign up flow, since using the prod images means that subscriptions are enabled, and the tests don't account for that).

So yeah, if someone were to try and hack into the pods running in the cluster... well, they wouldn't be able to get far. No shell, no shell tools to navigate around... I haven't the slightly clue what they'd *actually* be able to do. But it shouldn't be much.

Another consequence of the new prod Dockerfiles is that it's actually a multi-stage build. The first stage is regular old node-alpine, where it handles running the build, but also now the CI.

Yes, the CI commands are now run as part of the image building process. The primary consequence of this is that we no longer have a need for the separate CI step during the build pipeline; it's now 'contained' in the image build step. Doesn't really change anything, but I guess shortening the already-lengthy build pipeline (at least, from a lines-of-code perspective) is good.

Additionally, since the prod Dockerfiles are now multi-stage, the difference between `dependencies` and `devDependencies` actually kinda matters (for backend and marketing, but not frontend). This is because we can now 'cache' the non-dev dependencies separately so that only they make it into the final distroless image.

- Of course, since we have to split the dev/non-dev deps for frontend because of Cypress, this doesn't apply to it.

Also, because the prod images don't have `npm`, we had to adjust the `cypress_reset_db.sh` script to account for this and call a node script to reset the DB instead. Not really a big deal, since it's only relevant when running the e2e tests against the prod config.

We also had to use a 'custom' server for the marketing service, since that was easiest than trying to figure out how to invoke `next` without `npm`.

- Actually, once we figured out how to invoke `node_module` bin files without npm, we could *technically* have done that for the above, but since we needed to add the healthcheck endpoint to the marketing service *anyways*, made sense to do it as part of the custom server.

Although a similar change also had to be made for the `inject_stripe_webhook_secret.sh` script.

Descheduler

The descheduler was brought in to solve one particular problem: balancing an unbalanced cluster. See, when a node gets preempted, the replicas (pods) that were assigned to it (by our lovely pod anti-affinity rules) would get put onto the remaining node (cause we only have 2 nodes right now).

However, since the anti-affinity rules only apply at scheduling time, Kubernetes would never move any of the pods over to whatever new second node would be brought up (as long as the pods were running normally).

As such, the `descheduler` was brought in with its `RemoveDuplicates` rule that kills the duplicate pods from the one node so that they get re-scheduled onto the new node.

That's it. It kills duplicates every hour (if there any), and Kubernetes makes things right after that. Nothing crazy, but it greatly helps with HA, since we can rely on the replicas being evenly balanced between the nodes.

ClusterIP Services instead of NodePort

I was literally just reading through some random article on Kubernetes services when I realized that "wait, couldn't our services use `ClusterIP` instead of `NodePort` if everything's being routed by the ingress?"

And turns out, yes it can. So now all of the services are run using `ClusterIP`. No more exposed node ports (well, at least fewer exposed node ports -- `ingress-nginx` still needs some).

However, an annoying consequence of this change is that we need to delete the old services before deploying the new ones. Cause `ClusterIP` services aren't compatible with a certain `nodePort`property that gets added to `NodePort` services after they're created. So we basically need to delete the services from `master` before merging this task.

StatefulSet vs Deployment for Database

So... I don't know if it's because we upgraded Kubernetes recently -- or because we enabled `maxUnavailable: 0` for rolling updates -- but I *actually* ran into a case where the database tried to deploy a new pod, but it wouldn't ever come up because it was in a deadlock waiting for the old pod to release the persistent volume claim.

Turns out, this *exact* scenario is why `StatefulSets` exist. Which makes total sense, in hindsight.

So now we run the database using a `StatefulSet`. In practice, not much has changed, but it should be more stable/reliable.

Although, we did have to change the service to being a 'headless' service. That is, it's a `ClusterIP` service without an actual address... I don't know, but that's what `StatefulSets` required.

However, just like the `ClusterIP` service change above, there's a manual deletion component to this change: we'll have to manually delete the existing database deployment from `master`, as well as manually restore the database to the new `StatefulSet` instance. Not really a big deal.

In fact, because we have to destroy so much stuff, we might as well just delete the `master` namespace altogether and let the build pipeline figure it out. In theory, this shouldn't cause us any issues (minus a bit of downtime) since our infra is *so good*, but ya never know. I've made backups just in case.

Readiness vs Liveness Probes

As explained in this article, if you have to choose between readiness probes and liveness probes, prefer readiness probes. Why? Because they don't cause containers to be murdered if they fail (or something).

Anywho, this is just another way to say that I've finally got around to adding a readiness probe to all of the main service deployments (aka everything but the database).

This means that each service (backend, frontend, and marketing) now has a `/healthz` healthcheck endpoint. Yes, that also means that we changed the name of the Backend's endpoint from `/healthcheck` to `/healthz`.

Why is it `/healthz`? Where does the convention of using /healthz for application...

- tl;dr cause Google

BTW, a consequence of using readiness probes is that they are constantly pinged. Ya know, cause they're a healthcheck. Not really a problem, since our healthchecks just send back "1".

Changing the 'migration-job' template to just 'job'

Once I realized that the `command` attribute used in a pod template is *actually* more like Docker's `ENTRYPOINT` and *not* `CMD`, I realized that what we actually needed to use was the `arg` attribute.

This is particularly relevant with our new prod images, since the `ENTRYPOINT` (as far as I can tell) is always `node`. As such, we can only pass args to it.

So now the `migration-job` template doesn't need specific `migration_command` Kubails attribute, and just turns into a regular old `job` template. Good!