Task UFC-332: Decoupling Migrations

The following document will go over the design of decoupling database migrations from Backend startup, as outlined in task ticket UFC-332.

Acceptance Criteria

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

- Migrations should be run separately from Backend bootup.
- Backend instances should be able to be horizontally scaled without worrying about migration collisions.
- [not directly related] Deployments should be done using the 'rolling update' strategy.

Design Brainstorming

- Add a `job` Helm template that basically performs the init-container steps from the Backend deployment?
- Modify the build pipeline to deploy this job ahead of the rest of the services?
  - Will have to remove the manifest for the job before deploying the rest of the services? So that it isn't deployed again?
- No wait, that doesn't work, because that assumes that the database has already been deployed.
- So... can we couple the migrate step to the database service?
  - Technically... and this'll get real messy with the current Kubails config, but technically... we could give the database a pre-startup command that performs the migrations using the Backend image.
  - After all, the `initContainers` don't *have* to be the same image as the container being deployed.
  - It's just that... specifying this could get messy, from Kubails' config's perspective.
  - I suppose we could just add a `pre_startup_image` option to the config and optionally specify it in the `deployment` template as the pre-startup image (i.e. take it iff it exists; otherwise, use base image).
  - This way, we wouldn't have to modify the build pipeline.

- Now, are there any consequences to doing this?
  - Oh wait... yeah there are. Primarily, WE CAN'T DO THIS.
  - HELLO! IT'S AN INIT CONTAINER. WE NEED THE DATABASE RUNNING TO DO THE MIGRATIONS. WE CAN'T RUN THE MIGRATIONS BEFORE STARTING THE DATABASE.
  - GOD IM STUPID.
  - OK, so... sidecar?
- I mean... if we loop back to the build pipeline changes, technically we could just deploy the database first, then the job, then the rest of the services. We wouldn't even have to remove the database manifests if we did, since re-applying them won't change anything.
  - I think this is the most straightforward approach.
- Another approach would be to follow Running database migrations using jobs and init containers:...
  - Basically, instead of having the Backend containers wait-for the database, they instead wait for the success of a Job that runs the migrations (the Job can wait-for the database).
  - The only niggly thing is that that particular article relies on auto-incrementing the name of the job that is waited for. That is, there is no job cleanup.
  - This would work if we had a Job cleanup mechanism, but there currently exists no such (stable) mechanism in Kubernetes.
    - There are 3rd party solutions, however. For example: Roadsweeper — Kubernetes Job Cleaner
  - This does make me wonder if we can even re-deploy a Job to get it to run again... or if an already created job needs to have a separate "re-run" command. Hmm.
    - The consequence of this is whether the build pipeline would have to check if the job is already created and then must re-run vs just deploying it every time.
    - Man, if only we could have self-cleaning jobs...
    - According to Is it possible to rerun kubernetes job?, the job has to be deleted before it can be 're-run'. That is, no, re-deploying it doesn't re-run it, and no, there is no re-run mechanism.
    - So... just have a delete command that doesn't care if it fails.

    - If we're going the "always delete the job first" approach, then theoretically we could go the approach with the "wait-for-job" so that everything (job, database, and services) are deployed in one step, but we just have one step *before it* that handles deleting the job first.
        - In effect, our build pipeline becomes the job cleanup mechanism. Icky, but should work.
    - Note: We'd need to give more privileges to the default service account for the underlying "wait-for-job" tool to work.
        - See groundnuty/k8s-wait-for#6.
        - Obviously, this is non-optimal.
        - Especially since we can't specify different service accounts for init-containers than the actual container.
        - OK, so... back to running the job and database in a separate step?
- If we're running the job and database in a separate step, then we'll need to wait for the job to complete before we can move on in the build pipeline.
    - Thankfully, there's a `kubectl` command to wait for job completion: `kubectl wait --for=condition=complete job/myjob`
    - However, that only allows us to wait for the 'complete' status. If we wanted to wait for completion or failure, we'd have to do something like Wait for kubernetes job to complete on either failure/success using command line
    - Otherwise, we could wait for just completion and then have a timeout on it.
    - In either case (timeout or job failure), we'd want to abort from the build pipeline.
- Side note: We should improve the rolling update mechanism for deployments.
    - Basically:
        - "By default Kubernetes deploys pods using a “rolling update” strategy, removing 1 old pod at a time (`maxUnavailable: 1`) and adding 1 new pod instead (`maxSurge: 1`), which means that with 3 replicas, you would temporarily lose 33% of your ability to serve end-users’ requests as you roll a new version out.

          Let’s fix this by changing maxUnavailable to be 0. This way, Kubernetes will first deploy one new pod, and will only remove an older one if the deployment was successful. Note that one downside is you need spare capacity in your cluster to temporarily run this extra replica, so if you are already close to capacity you may need to add an extra node."

- How to Correctly Handle DB Schemas During Kubernetes Rollouts

Tasks

- Create the migration job template.
- Make it work.