Story UFC-335: Improve Backups

The following document will go over the design of improving our backup system, as outlined in story ticket UFC-335.

Note: This isn't really a 'story', in that it isn't a 'user' story. It's a 'developer' story; basically, just a super task. The only real reason to classify it as such is so that we can create sub-task tickets.

Acceptance Criteria

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

- Encrypt backups.
- Copy backups to AWS.
- Auto delete old backups.
- Restore latest backup to new feature branches.

Design Brainstorming

The following are the design ideas for each acceptance criteria.

Encrypting Backups

- Use `gcloud kms` to handle encryption?
- Seems like KMS has issue with encrypting large files. Looks like we'll have to use envelope encryption. AKA, what the app does -- have a DEK that encrypts the data and a KEK that encrypts the DEK.
  - The DEK will likely be from `gpg`, so that we'll perform the actual database file encryption with `gpg`, then encrypt the `gpg` key with `gcloud kms`, storing the resulting EDEK in... secrets manager? I guess?
  - Or we could do what we've been doing forever... store the EDEK in the repo. Duh.

So it looks like, if we're doing symmetric encryption with GPG, we just need to specify a 'passphrase', and GPG will handle generating the AES key. So really, the 'secret' we need to deal with it is this passphrase.

Now, traditionally we've specified our secrets in `kubails.json` so that they get injected as env vars. However, I've come to the realisation that that exposing secrets as env vars isn't necessarily the most secure way to do it. Why? Because the secrets are essentially stored in plaintext in Kubernetes. I mean, this isn't really anything new, but...

Ah fuck it, just do it as always. If someone gets into our cluster and can read the secrets, we're already screwed.

Although theoretically, if we wanted to improve this, we could mount the secret as a volume or just store the EDEK in the image and decrypt it at runtime using KMS.

So to recap how this will work...

- Generate a GPG key (DEK) that will be used for encrypting the backup files.
  - Passphrase generator using basic shell commands
- Encrypt this DEK using our KMS key to get an EDEK.
- Store this EDEK as a file in the `backend-database-backup` service.
- Specify the EDEK as a secret in `kubails.json`.
- Modify the backup script of `backend-database-backup` to use `gpg` to use the key to encrypt the backup file before pushing the file to the storage bucket.
- Modify How to Restore a Postgres Dump to include steps on how to decrypt the backups.

Commands

To generate passphrase:

-
  ```
  dd if=/dev/urandom bs=256 count=1 2>/dev/null | base64 | sed
  's/=//g > FILENAME
  ```
  - The `bs=256` is the block size, which controls how big the output is. 256 means our passphrase is... quite large.

To encrypt:

-
  ```
  echo "$ENV_VAR" | gpg --batch --yes --passphrase-fd 0 --symmetric
  --cipher-algo AES256 ENCRYPTED_FILE
  ```

To decrypt:

-
  ```
  echo "$ENV_VAR" | gpg --batch --yes --passphrase-fd 0 --decrypt --
  cipher-algo AES256 ENCRYPTED_FILE
  ```

Copying Backups to AWS

- Instead of doing this as part of the cronjob, how about we attach, e.g., a Cloud Function to the storage bucket to handle it? (or some other event based listener)

- Actually, I've now learned that `gsutil` can directly copy to S3. So it (should) be pretty easy to just add an extra copy command to the `backend-database-backup` job.

The only tricky part is how we handle authenticating to AWS/S3.

- Apparently what we need to do is configure a `~/.boto` file to hold the credentials.
- Apparently Boto is a Python package for the AWS API. So a 'boto' file would be the configuration for this package.
  - Here's GCS' docs on Boto configuration: Boto configuration file | Cloud Storage | Google Cloud
    - Also relevant: Cloud Storage interoperability | Google Cloud
  - And a relevant Stack Overflow thread: Google cloud: Using gsutil to download data from AWS S3 to GCS

Also, we kinda need to deal with setting up an AWS account for uFincs.

- Need to remember how the whole root/IAM user thing is setup...
- Then need to figure out how to setup the equivalent of a service account in AWS for just S3...
  - Apparently we need an `aws_access_key_id` and an `aws_secret_access_key` and these properties go in the `boto` file.
  - Oh I see, these two properties are just the 'access keys' for an IAM user. So we'll just need to create an IAM user with restricted permissions. AKA, there isn't a separate concept for 'service accounts' in AWS.
- Don't forget to add AWS to our Paid Services doc.

Based on the fact that the `boto` file is the one that will contain the AWS credentials/config, I guess we could store the file as a secret then mount it? Although, that would require `deployment` template changes, plus I don't know if we can even mount a secret file directly to the home folder.

So it probably makes more sense to just use our existing secrets mechanism and store the AWS access key in a secret, inject as an environment variable and then... have the backup script write that env var into the `boto` file? A templated `boto` file? Not exactly pretty, but should work.

- Hmm, I don't know if `gsutil` generates its own `boto` file (you know, to store config/credentials for accessing GCP), but if it does, then that throws a small wrench into this plan.
- I'd rather not have to have to script custom "find this spot in the file and insert this text", but we do what we have to.

Alternatively, instead of using `gsutil` to perform the copy, we could always just, ya know, include the `aws` CLI and copy the file directly to S3 using that. Ya know what, let's just do that. It's probably easier than messing around with some bespoke config file.

- Create an AWS account.
- Setup root/IAM user for myself.
- Setup an IAM user as the S3 writer and get the access key/ID.
- Create the S3 bucket, choose an appropriate storage class, and make sure it isn't public.
  - Ah crap, are we gonna have to add this AWS config to Terraform?
  - At worst, we should at least document how to setup the account + bucket.
- Add the AWS CLI to the `backend-database-backup` Dockerfile.
- Store the S3 writer access key/ID as secrets in the `backend-database-backup` service.
- Add the secrets to `kubails.json`.
  - See Environment variables to configure the AWS CLI - AWS Command Line Interface for the names the env vars need to be.
- Modify the `backend-database-backup` script to copy the encrypted backup to AWS S3.

Storage Class

I've made the decision to just use the STANDARD storage class for the backup bucket. This is primarily because we can probably keep our usage under the free tier limits (5GB of STANDARD) for quite some time.

Although I've now just learned that the 5GB of free STANDARD storage is only during the 12 month trial. After that, there's only an amount of GLACIAL storage available at the baseline free tier.

Auto Deleting Old Backups

AKA, lifecycle rule on the storage bucket. Just don't forget to add the rule in Terraform. And to take a backup of everything before cleaning things up.

- Take backup of all backups.
- Add lifecycle rule to Terraform.

Restoring Latest Backups to Feature Branches

Now that we have the separate step for deploying + migrating the database in the build pipeline, it should be much easier to add another step to restore the latest backup to the database.

Just need to write a script that does the following:

- Grab the latest backup file from the S3 bucket (??)
- Decrypt the backup (??)
- Perform the How to Restore a Postgres Dump process.

Three things just occurred to me:

i. We have to make sure the restore script is only run on non-master branches.

ii. This restore script *has* to run before migrations are done.

iii. This restore script could actually be a `job` just like the migration job.

Easy enough to ensure 1.

Easy enough to ensure 2.

3 is also easy enough and actually provides us the way to inject the key (as a secret) to decrypt the backup. So I think making this a job (but in a separate folder) is our best bet.

I do think we should be able to re-purpose much of the existing `migration-job` template; just modify it to be a generic `job` template and change the `migration_command` Kubails config attribute to just `command`.

I was gonna say that one thing we do need to keep in mind is that re-deploying the restore job during the 'deploy all services' step would be bad, until I remembered that re-deploying a job on top of an already completed job does nothing.

Restore on Every Deploy or Only on Branch Creation?

Now here's a question: do we execute the restore job on every deploy of feature branches, or only on the creation of the branch? Two sides:

i. Deploying every time keeps the feature branch fresh with the latest data from prod, enabling users to always be in-sync for testing purposes.

ii. Not deploying every time means that changes made to that feature branch's database actually persist for the life of the branch. Importantly, this means that users who test out a new feature on the branch will keep whatever data they created, rather than having it blown away on every deploy.

I think I'm erring on the side of "not deploying every time". As such, how do we ensure we only deploy once?

- Well, we *could* check if the namespace exists or not before deploying the job (actually, we'd need to check before deploying the database, since it's that deploy command that creates the namespace, but details), but I think we have an even easier solution.
- Namely, what I stated above: re-deploying a job on top of a completed job doesn't re-run it. As such, as long as it runs and completes once, we don't actually have to do anything past that to ensure it only runs once.

Preventing Job Deployment in Regular Deploy Step

Ah shit, we also need to make sure the job doesn't run as part of the regular deploy step.

- Add a `kubails.json` prop for "only running on non-production" branches? Then... modify the CLI to read that value and exclude it? That's too much work...
- Or we could just read that value into the template and then... not generate anything? So that deploying it does nothing?
- Or we could go really jank and just delete the generated manifests as part of the database migration step?

Those are 3 all totally valid solutions (in that they solve the problem), but in decreasing order of janky/tech debtyness.

- Modifying the Kubails CLI to support this option is probably the most robust option (especially since then we could propagate it out so that the service isn't even built/CI'd on master), but it's certainly the most work of the 3 options. However, doing this would also be useful for future uses of Kubails.
- Just blanking out the generated manifest is pretty damn simple, but I don't actually know if that works. `kubectl` might throw an error about trying to deploy an empty file. But another benefit of this approach is that the CI pipeline doesn't need to be modified to have any `if namespace === master` conditionals.
  - Yep, `kubectl apply` complains that there are "no objects to apply" with an empty file. Damn.
- Just deleting the generated manifests as part of the build pipeline is just... so scuffed. First, we *do* need the `if namespace === master` conditional to check if we should run the restore job in non-master branches. Then we need to make sure to cleanup up the manifests at the end of that build step *regardless* so that they don't get deployed as part of the next step.
  - Also, I don't think *this* even works, because the underlying implementation of `kubails cluster manifests deploy` checks each folder individually and deploys it, instead of just applying the entire `manifests/generated` folder. As

    such, it'll probably throw an error when it can't find the manifests for the restore service.

- I suppose an even *more* scuffed option would be to specify the explicit list of services to deploy as part of the full deploy step.
- I suppose we could just check the `$NAMESPACE` value in the restore script itself and just bail out early if we're running in master. That... is simple, and should work. Won't require any Kubails tweaks nor any extra build pipeline changes.
    - Let's go with this!

Problems Running Kubectl Commands in a Job

Hmm, might have hit a snag while writing the script. I don't think we'll be able to execute `kubectl` commands from inside a job. Or at least, we shouldn't be able to, by default. I assume we'd need to give it extra permissions to do this.

So if we don't want to do it as a job, then I guess it's back to doing it as part of the build pipeline? That would remove my concerns about accidentally deploying the 'service' to master, since it would no longer be a service.

The only nibbly thing would be accessing the decryption key. We'd have to decrypt the secrets for the `backend-database-backup` service then read in the key from that.

Other than that, it *should* be fine.

Oh wait, nope. *Now* we'd have to have a check for whether or not the branch/namespace was newly created, since we can't rely on just pushing the job over and over and not running. So, before the database deploy command, we'd need to check for the existence of the namespace (don't forget to lowercase the branch name).

In summary

- Write the script that will grab the latest backup file, decrypt it, and restore it.
    - Make sure the restore job only runs on non-master branches.
- Modify the build pipeline to execute the restore job after deploying the database but before the migration job.