Migrate Magnolia from Cloud 3 to PaaS

latest, 2024-07-17: External release

Table of Contents

Copyright

Revision History

Revision	Date	Comments
`latest`	2024-07-17	External release

Chapter 1. Introduction

This article guides you on from Cloud 3 to PaaS.

There are two main steps when moving from Cloud 3 to PaaS:

1. Migrating the customer's content, datastore, and configuration. See more in Migrating content.
2. Preparing the application to use the PaaS project files. See more in Preparing for PaaS.

Chapter 2. Migrating content

2.1. Why you're here and important tips

In all likelihood, a PaaS onboarding ticket was created to migrate a Cloud 3 customer to PaaS. In this case, the PaaS team should create the necessary (typically `dev` and `prod`) clusters on the PaaS side of things for you. If they haven't, make sure they have done so. You can reach out to them at PaaS Slack external.

Often, Cloud 3 had multiple environments such as `dev`, `uat`, and `prod`, each which includes a public and author instance. You'll need to make sure you map the correct environment to the correct Kubernetes cluster for PaaS.

Example 1. How long will this take?

This should take between `10-30` minutes if you don't encounter any issues and use the faster options described below. This is, of course, an estimate of time and every customer migration varies.

2.2. Prerequisites

- Access to Rancher
- Able to run the `kubectl` command locally
- Access to the Cloud 3 customer package (*request from Helpdesk*)
- Access to the PaaS customer cockpit with appropriate permissions (*request from Helpdesk*)
- Contact with a Magnolia SRE

2.2.1. Required files

You'll need the following files:

- Database shell script
- Datastore shell script

2.3. Prepare the dump file

2.3.1. For SRE

You'll need to request that SRE runs this command for you and shares the link. You can give them the following instructions.

1. First, they'll need to trigger the following command for **all** Cloud 3 environments.

```
pg_dump -Fc -h localhost -p 15432 -U postgres -d magnolia > magnolia-<CUSTOMER-
```

```
NAME>.dump ①
```

① Where the `*.dump` file name matches your own requirements. We recommend naming it based on customer.

2. SRE should share the prepared `.dump` file via Google Drive or SE with the AWS CLI.

2.3.2. For you (the migration guru)

1. Use the groovy script in `groovy-script/print-magnolia-workspaces.groovy` to print the required workspaces.

   You'll need to update the workspaces to `INCLUDEWORKSPACES` in `migrate-<CUSTOMER-NAME>.sh`.

```
...
INCLUDEWORKSPACES=(
#       magnolia_conf_sec-mgnlSystem
                'translationQueues'
                'blog_de_blogs2'
                'advancedCache'
                'workflow'
                'rss'
                'partners'
                'blog_en_blogs2'
                'blog_de_blogs'
                'category'
...
```

2. The PaaS team should have created clusters for you. If not, request that. Assuming they have, download the Kubernetes `kube_config` file.

   This is typically downloadable via a button from Rancher from within the cluster.

3. Export the `kube_config` file to your path.

```
export KUBECONFIG=$PAAS/kube_config/CUSTOMER-NAME-dev.yaml ①
```

① The example here shows a customer's dev cluster.

4. Create a new `migration` container in the cluster.

```
kubectl apply -f magnolia-migration-<CUSTOMER-NAME>-postgresql.yml ①
```

① Where you match per customer the `CUSTOMER-NAME`.

</> `magnolia-migration-postgresql.yml` example file

```
apiVersion: v1
kind: Secret
metadata:
  name: magnolia-postgresql
  namespace: migration
  labels:
    app: magnolia-postgresql
type: Opaque
data:
  postgresql-password: ENTER_YOUR_PASSWORD_HERE
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: magnolia-postgresql
  namespace: migration
spec:
  replicas: 1
  selector:
    matchLabels:
      app: magnolia-postgresql
  template:
    metadata:
      labels:
        app: magnolia-postgresql
    spec:
      volumes:
        - name: data
          emptyDir: {}
#          persistentVolumeClaim:
#            claimName: migration-task-pvc
      containers:
        - name: postgres
          image: postgres:14.4
          env:
            - name: POSTGRES_USER
              value: magnolia
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: magnolia-postgresql
                  key: postgresql-password
          ports:
            - containerPort: 5432
          volumeMounts:
            - name: data
              mountPath: /var/lib/postgresql/data
              subPath: postgres
          resources:
```

```
            requests:
              memory: "512Mi"
              cpu: "500m"
            limits:
              memory: "1Gi"
              cpu: "1"
---
apiVersion: v1
kind: Service
metadata:
  name: magnolia-postgresql
  namespace: migration
spec:
  selector:
    app: magnolia-postgresql
  ports:
    - port: 5432
```

2.4. Get the dump file

Now that we have prepared the dump file, we'll copy it to the container as needed. There are two options below.

Recommend SRE to share the `.dump` file in a Google Drive location. This is the most efficient way to download the dump file.

`gdown` from container

The first option is to use the `gdown` command and dowload the dump directly from Google Drive. *This is the faster of the two options.*

1. Go to into the `migration` container.

```
exec kubectl exec -i -t -n migration magnolia-postgresql-88b6d8fbf-2jmh4 -c
prod -- sh -c "clear; (bash || ash || sh)"
```

2. Download the file from within the container.

```
apt update
apt install python3-pip
pip install gdown ①
gdown ${google-drive-link} ②
```

① `gdown` is a python specific command for downloading Google Drive files.

② You'll get this from the link shared with you for the migration.

3. Share the file in the `anyone with the link` mode if needed.

After the file is downloaded, return the file to restricted mode. This is critical for data protection.

Fastest ϟ S3 via AWS CLI

1. Install the AWS CLI.
2. Run the following commands:

```
export AWS_ACCESS_KEY_ID="ACCESS_KEY" ①
export AWS_SECRET_ACCESS_KEY="SECRET_ACCESS_KEY" ②
aws s3 cp s3://dump-db/live/live-public1-magnolia_conf_sec.dump ./live-
public1-magnolia_conf_sec.dump
aws s3 cp s3://dump-db/live/live-public1-magnolia.dump ./live-public1-
magnolia.dump
```

① The AWS access key.

② The AWS secret access key.

Copy from local env

The other option is to copy the file from the local environment and use the `kubectl cp` command to send to the container.

This is the slower of the options and is only recommend if you cannot get the Google Drive link.

figure 1. Author

```
kubectl cp /path/magnolia-author1-<CUSTOMER-NAME>.dump migration/magnolia-
postgresql-88b6d8fbf-bwnx9:/tmp/magnolia-author1-live-<CUSTOMER-NAME>.dump
```

figure 2. Public

```
kubectl cp /path/magnolia-public1-live-<CUSTOMER-NAME>.dump migration/magnolia-
postgresql-88b6d8fbf-bwnx9:/tmp/magnolia-public1-live-<CUSTOMER-NAME>.dump
```

Also, you'll need to copy the script locally to the container path:

```
kubectl cp migrate-<CUSTOMER-NAME>-db.sh migration/magnolia-postgresql-
88b6d8fbf-2jmh4:/tmp/migrate-<CUSTOMER-NAME>-db.sh
kubectl cp migrate-<CUSTOMER-NAME>-db-ds_datastore.sh migration/magnolia-
postgresql-88b6d8fbf-2jmh4:/tmp/migrate-<CUSTOMER-NAME>-db-ds_datastore.sh
```

2.5. Migration examples

2.5.1. Content

This copies the `.dump` file to your target database.

See the table below for more details on the commands.

```
/bin/zsh ./migrate-<CUSTOMER-NAME>-db.sh -h dev-magnolia-helm-public-db.dev -p 5432 -f
/tmp/magnolia-public1-live-<CUSTOMER-NAME>.dump -d public -u postgres
# Author
./migrate-<CUSTOMER-NAME>-db.sh -h prod-magnolia-helm-author-db.prod -p 5432 -f
/tmp/magnolia-author1-live-<CUSTOMER-NAME>.dump -d author -u postgres
# Public
./migrate-<CUSTOMER-NAME>-db.sh -h prod-magnolia-helm-public-db.prod -p 5432 -f
/tmp/magnolia-public1-live-<CUSTOMER-NAME>.dump -d public -u postgres
```

2.5.2. Datastore

```
/bin/zsh migrate-<CUSTOMER-NAME>-db-ds_datastore.sh -h localhost -p 59638 -f
$HOME/Magnolia/backup/author/magnolia-author1-live-<CUSTOMER-NAME>.dump -d magnolia -u
magnolia
# Author
./migrate-<CUSTOMER-NAME>-db-ds_datastore.sh -h prod-magnolia-helm-author-db.prod -p
5432 -f /tmp/magnolia-author1-live-<CUSTOMER-NAME>.dump -d author -u postgres
# Public:
./migrate-<CUSTOMER-NAME>-db-ds_datastore.sh -h prod-magnolia-helm-public-db.prod -p
5432 -f /tmp/magnolia-public1-live-<CUSTOMER-NAME>.dump -d public -u postgres
```

2.6. Migrate configuration and permissions

Migrating configuration as well as permissions-based information such as users, roles, and groups, you'll need to use the JCR Tools App. For more on that, see here.

1. First, migrate the `config` workspace.
    a. Use the JCR exporter to export the latest `config`` workspace for `author` and `public`. Alternatively, you can replace existing nodes with the same ID. However, this is for XML only.
    b. Go to the target `author` and `public` environment, and use the JCR importer to import the downloaded `config`` workspace.
2. Now, let's migrate the `users` workspace.
    a. Use JCR exporter to export latest `users` workspace in `author`.
    b. Open `users.xml` in your text editor.

c. Remove the entire `system` node.

d. Put the `users.xml` file in your bootstrap folder in your light-modules like `/light-modules/bootstraps/users.xml`.

e. Configure the helm `values.yml` file for both `magnoliaAuthor` and `magnoliaPublic`:

```
catalinaExtraEnv:
   magnolia.content.bootstrap.dir: '/mgnl-home/modules/bootstraps'
   magnolia.content.bootstrap.createTasks: onchange ①
```

① As soon as you reload your PaaS instance, a task will appear in your admincentral. Run the task, and it will bootstrap the configuration for users, roles, and groups.

3. Remove the index folder and restart both author and publics to reindex again.

2.7. Clean up the `migration` container

To clean up the migration *container/workspace*, run the following.

```
kubectl delete -f magnolia-migration-postgresql.yml ①
```

① This should match the filname for your particular migration.

2.8. Commands table

File/flag	Notes
`migrate-<CUSTOMER-NAME>-db.sh`	Your shell script.
`-h dev-magnolia-helm-public-db.dev`	The target namespace.
`-p 5432`	The port.
`-f /tmp/magnolia-public1-live-<CUSTOMER-NAME>.dump`	The location of the `.dump` file.
`-d public`	The schema such as public or author.
`-u postgres`	The username for your database.

2.9. Troubeshooting

2.9.1. Permission denied

If you're denied access on the files, try to change the permissions on the file with the `chmod` command.

```
chmod +x migrate-<CUSTOMER-NAME>-db-ds_datastore.sh
chmod +x migrate-<CUSTOMER-NAME>-db.sh
```

2.9.2. Delivery endpoints not returning workspace data

For example, if you were denied access to `\<URL>/.rest/delivery/partners`.

1. Remove index folder in both `author` and `public` and restart to index again.

```
cd /mgnl-home/repositories/magnolia/workspaces/
find . -name index -exec rm -rf {} \;
```

Follow the steps here if you need help.

2.9.3. Replica issues

When we scale public replicas to `2`, we might face the issue on the newly-created pod:

figure 3. Example

```
2022-12-30 12:40:53,402 WARN  org.apache.jackrabbit.core.NodeImpl : Fallback to
nt:unstructured due to unknown node type '{http://www.magnolia.info/jcr/mgnl}block' of
node /your-nodes
```

To fix this:

1. Copy `/mgnl-home/repositories/magnolia/repository/nodetypes/custom_nodetypes.xml` from `public-0` (original public) to new `public-1` (new public).
2. Restart `public-1` (new public).

2.9.4. PostgreSQL version

It's possible that you will need to align your PostgreSQL version from **Cloud 3** to **PaaS**. If so, make sure the `tag` in your `values.yml` file for **PaaS** is the same as the PostgreSQL version as the Cloud 3 database you're moving from:

```
...
db:
  tag: 12.8 ①
...
```

① Where `tag` is the version.

Chapter 3. Preparing for PaaS

Now, that you've moved your content, you'll need to set up the PaaS project side.

3.1. Prerequisites

- Access to the Cloud parent POM file

3.2. Instructions

1. You just need to ensure that the application you're moving from Cloud 3 to PaaS uses the Cloud parent POM file.

This file ensures that all **mandatory** libraries needed to deploy PaaS are available.

2. Ensure the project uses the latest magnolia version.

Follow the steps in paas:ROOT:setup.pdf or run `maven archetype` for the latest project structure.

3. *optional* If the project is using any Cloud 3 specific maven plugins, you'll need to remove those.

If we're hosting for the PaaS customer, the CI/CD setup is handled by the PaaS team and there's nothing else to do.

If they're not using GitLab via us, they'll need to implement a CI/CD build in the customer's source control.