Fragalysis Stack (Kubernetes)
Preparation

1 Installation
  1.1 Cluster Requirements (AWS) ......................................................... 4
  1.2 Container Images (AWS) ............................................................. 5
  1.3 Graph and Stack data (AWS S3) ................................................... 7
  1.4 Installation - Basic (Built-in Infrastructure) ................................. 8
  1.5 Installation - Basic (On-Prem Infrastructure) ............................... 16
  1.6 mArh Screenshots .................................................................. 24
  1.7 AWX Fundamentals ................................................................. 27

2 Development Architecture ......................................................... 31
  2.1 Continuous Development ......................................................... 31
  2.2 Continuous Deployment ......................................................... 41

3 Development Processes ............................................................... 45
  3.1 The AWX Development Server .................................................. 45
  3.2 The AWX Production Server ..................................................... 46
  3.3 Preparing for Kubernetes Development ...................................... 47
  3.4 Cheat Sheets .................................................................. 50
  3.5 Deploying your Stack (“Development for Dummies”) .................. 59
  3.6 Sensitive Material (Ansible Vault) ............................................ 65

4 Administration ...................................................................... 67
  4.1 Cloud Provider (STFC) ............................................................. 67
  4.2 The Rancher Server ............................................................... 67
  4.3 The NFS Servers ................................................................ 70
  4.4 AWX (Adding Developers) ..................................................... 73
  4.5 Related Repositories ............................................................. 75
These instructions cover deployment of a basic production-grade Fragalysis Stack (without custom authentication).

The Fragalysis Stack application consists of a **Stack** and a **Fragment Graph**. These application rely on additional *infrastructure* software in order to operate, specifically:

- You can rely on our *built-in* infrastructure components (which come with a number of cluster pre-requisites) that are handled by our *ansible-infrastructure* repository, which installs the following:
  - An NGINX ingress controller and associated Network Load balancer
  - A RWX storage class (AWS EFS in our case)
  - Pod Security Policies
  - Certificate management
  - **AWX** (the open-source upstream distribution of RedHat’s Ansible Tower)
  - A PostgreSQL database
  - An optional **Keycloak** server.

Once the infrastructure is installed, deployment of the Graph-based fragment database and Fragalysis Stack is achieved through the use of **Jobs** that are configured on the infrastructure AWX server.

If you cannot install our infrastructure (i.e. you’re on a secure or limited *on-prem* cluster) you must provide, at the very least:

- An NGINX ingress controller and associated Network Load balancer
- A RWX storage class (AWS EFS in our case)
- Pod Security Policies
- Certificate management

In addition to the application cluster you will need to provide an AWS S3 bucket (or buckets) from which you can serve the graph and stack data - the applications will not work without any background data.
Fragalysis Stack (Kubernetes)

What follows is a simplified guide to setting up a basic\(^1\) deployment of the Fragalysis Stack through a number of stages: -

## 1.1 Cluster Requirements (AWS)

The following *minimum* (preliminary) cluster requirements will need to be satisfied before the Fragalysis Stack can be deployed and used.

### 1.1.1 Cluster

1. **Kubernetes Admin User.** The Kubernetes cluster must provide a non-tokenised (non-expiring) user with cluster admin privileges. This is the user that the deployment playbooks will use to maintain the cluster.

2. **An AWS IAM user** capable of reading from an AWS S3 bucket, used to provision fragment graph and Fragalysis media data (i.e. a user with at least *AmazonS3ReadOnlyAccess* permissions).

3. **AWS S3.** The cluster must allow READ access to AWS S3 where fragment data for the neo4j graph database and loader (media) data for the Stacks is expected to reside. The bucket name can be configured during deployment.
   - In EKS the *Graph Node* (see below) is likely to need the *arn:aws:iam::aws:policy/AmazonS3ReadOnlyAccess* policy, typically assigned to the node using the `iam -> attachPolicyARNs` block in the cluster definition file. You can see this in the [cluster example in our ansible-infrastructure repository](https://example.com/cluster_example).  
   - We can provide open-access to the existing Fragalysis Stack graph data but if you want to use your own fragment data you will need to ensure you publish it to a suitable bucket that can be accessed by the cluster.

4. **One Application Node.** A compute instance with the following minimum specification: -
   - 8 cores
   - 32Gi RAM
   - 40Gi root volume
   - Kubernetes node labels
     - purpose=application
   - Kubernetes node taints
     - (none)

5. **One Graph Node.** A compute instance with the following minimum specification: -
   - 8 cores
   - >128Gi RAM
   - 40Gi root volume
   - Kubernetes node labels
     - purpose=bigmem
   - Kubernetes node taints
     - purpose=bigmem:NoSchedule

---

\(^1\) A cluster with an AWX server and a Fragalysis Stack without custom credentials or backup and recovery strategies. A cluster with the ability to use the stack with publicly available graph fragment data.
6. **GitHub Access.** The cluster must allow access to Ansible playbooks and roles that are located on publicly accessible repositories on GitHub. The cluster must not be prevented from accessing these repositories. The current list of GitHub repositories is listed below:

- InformaticsMatters/dls-fragalysis-stack-kubernetes
- InformaticsMatters/docker-neo4j-ansible

7. **Hostnames.** You will need to provide routing to your cluster for at least two hostnames, one for the fragalysis stack (i.e. fragalysis.example.com) and one for the AWX server (i.e. awx.example.com).

8. **Networking (ingress).** We deploy the nginx ingress controller as a **DaemonSet**, deployed to each compute instance. This acts as an internal load-balancer and routing service. It directs HTTPS traffic to the corresponding container (Pod).

9. **Networking (load balancing).** We need to load-balance traffic to the cluster. On AWS, rather than create a Application Load Balancer, which would normally result in an ALB instance for each ingress, we create a **LoadBalancer Service**, which creates a single layer-4 AWS Network Load Balancer (NLB) for the entire cluster.

   If the use of an NLB is not acceptable and instead you want to use an ALB or your own load-balancing solution you will be responsible for its installation and management.

10. **Networking (certificates).** The fragalysis stack is a web-based application that the user normally interacts with using a resolvable hostname, i.e. fragalysis.example.com.

    To simplify and streamline deployment, and avoid users having to provide their own certificates, our solution deploys the cert-manager, a native Kubernetes certificate management controller. We use it to automatically issue and renew certificates to allow SSL (HTTPS) connection to the stack using Let’s Encrypt. This relies on the certificate manager’s ability to connect to the Let’s Encrypt service.

    If this is not possible in your cluster and you need HTTPS connections to the stack you’re deploying you will need to provide your own certificate solution.

11. **A RWO storage class.** This is typically GP2 and is often built-in, especially if you’re using EKS.

### 1.1.2 Control Machine

You will also need a suitable **Control Machine**, from which you will be running at least some Ansible playbooks. The control machine’s requirements are covered in the **Command-Line Requirements (Control Machine)** document.

### 1.2 Container Images (AWS)

For a production deployment of the XChem Fragalysis Stack the following list of container images are normally deployed to the AWS EKS Kubernetes cluster. The images are used for the deployments of the core services (Infrastructure and Fragalysis Stack) and optional areas that cover **Backup and Recovery** and **Keycloak**.

While we try to maintain the accuracy of this list some versions may change as development continues, this will most likely affect the version of the Fragalysis Stack as this image is likely to incur regular revision updates as the Stack’s development is a continuous and on-going process. Most other images are unlikely to change often.

---

1 This list does not cover container images that would normally be considered part of Kubernetes.
1.2.1 Infrastructure Core

These images provide certificate generation, the NGINX ingress controller and an AWS EFS volume provisioner.

<table>
<thead>
<tr>
<th>Container Image</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>docker.io/jettech/kube-webhook-certgen</td>
<td>v1.2.0</td>
</tr>
<tr>
<td>quay.io/jetstack/cert-manager-cainjector</td>
<td>v0.12.0</td>
</tr>
<tr>
<td>quay.io/jetstack/cert-manager-controller</td>
<td>v0.12.0</td>
</tr>
<tr>
<td>quay.io/jetstack/cert-manager-webhook</td>
<td>v0.12.0</td>
</tr>
<tr>
<td>quay.io/kubernetes-ingress-controller/nginx-ingress-controller</td>
<td>0.33.0</td>
</tr>
<tr>
<td>quay.io/external_storage/efs-provisioner</td>
<td>v2.4.0</td>
</tr>
</tbody>
</table>

1.2.2 Infrastructure Database (Optional)

<table>
<thead>
<tr>
<th>Container Image</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>docker.io/library/postgres</td>
<td>12.2</td>
</tr>
</tbody>
</table>

1.2.3 Keycloak (optional)

If installing Keycloak, you must install the Infrastructure Database.

<table>
<thead>
<tr>
<th>Container Image</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>docker.io/jboss/keycloak</td>
<td>11.0.0</td>
</tr>
</tbody>
</table>

1.2.4 Infrastructure AWX (Optional)

If installing AWX, you must install the Infrastructure Database.

<table>
<thead>
<tr>
<th>Container Image</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>docker.io/ansible/awx</td>
<td>13.0.0 latest 7</td>
</tr>
<tr>
<td>docker.io/library/redis</td>
<td></td>
</tr>
<tr>
<td>docker.io/library/centos</td>
<td></td>
</tr>
</tbody>
</table>

1.2.5 Fragalysis Stack

These images are required for a named (tagged) production Fragalysis Stack, its own database and S3 data loader. The stack also requires a fragmentation database, provided by a specialised neo4j image.

<table>
<thead>
<tr>
<th>Container Image</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>docker.io/library/busybox</td>
<td>1.28.0</td>
</tr>
<tr>
<td>docker.io/library/postgres</td>
<td>12.2</td>
</tr>
<tr>
<td>docker.io/xchem/fragalysis-stack</td>
<td>2.0.4</td>
</tr>
<tr>
<td>docker.io/informaticsmatters/fragalysis-s3-loader</td>
<td>2.0.3-1</td>
</tr>
<tr>
<td>docker.io/informaticsmatters/neo4j</td>
<td>3.5.20-1</td>
</tr>
<tr>
<td>docker.io/informaticsmatters/neo4j-s3-loader</td>
<td>3.5.20-1</td>
</tr>
</tbody>
</table>
1.2.6 Backup and Recovery (optional)

<table>
<thead>
<tr>
<th>Container Image</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>docker.io/informaticsmatters/sql-backup</td>
<td>2020.4</td>
</tr>
<tr>
<td>docker.io/informaticsmatters/sql-recovery</td>
<td>2020.4</td>
</tr>
</tbody>
</table>

1.3 Graph and Stack data (AWS S3)

The Fragment Graph database needs to be loaded with fragment data before it can be used and the Fragalysis Stack needs to be loaded with target/media data before it can be used.

This data typically resides on an NFS server or an S3 object store.

The graph database pulls the data down as it initialises but the stack, once running, needs to be loaded. The stack is often re-loaded with more data as needs dictate.

Note: You need to have access to this data before you can deploy the graph or the stack.

In order to access AWS S3 data you will need to provide the graph and stack loader with AWS credentials that have suitable permissions. For AWS your policy will need s3:Get* and s3:List* permissions for the buckets and paths you intend to use.

As an example, users of our buckets are given the following policy:

```
{
   "Version": "2012-10-17",
   "Statement": [
     {
       "Effect": "Allow",
       "Action": [
         "s3:Get*
       ],
       "Resource": "arn:aws:s3:::im-fragnet/combination/3/*"
     },
     {
       "Effect": "Allow",
       "Action": [
         "s3:Get*
       ],
       "Resource": "arn:aws:s3:::im-fragalysis/*"
     },
     {
       "Effect": "Allow",
       "Action": [
         "s3:List*
       ],
       "Resource": "arn:aws:s3:::im-fragnet"
     },
     {
       "Effect": "Allow",
       "Action": [
         "s3:List*
       ],
       "Resource": "arn:aws:s3:::im-fragnet"
     }
   ]
}
```

(continues on next page)
1.3.1 Graph data

Graph data consists of Node and Relationship definitions, typically stored in a series of compressed CSV files. On S3 this data is stored as objects on a path. As an example you might have fragalysis-graph bucket and the path combination/3. The bucket and path are not important but the Graph’s initialisation (consisting of a load phase) will simply copy all the objects on the path (not recursively) and expect them to represent a viable graph.

You must have the following files/objects on the path: -

• load-neo4j.sh

1.3.2 Stack data

Stack data consists of Target definitions, typically stored on a bucket data origin. The files are peculiar to the Fragalysis application. Their their format is not covered here.

Stack data is stored in bucket directories, normally named using the format YYYY-MM-DDTHH but other directories may also exist. If you have target data in the data directory 2020-09-15T16 this is your data origin.

Target data must exist in the following bucket and path: -

• s3://<DATA_BUCKET>/jango-data/<DATA_ORIGIN>

1.4 Installation - Basic (Built-in Infrastructure)

This section documents the installation of Fragalysis on AWS (typically EKS). If you cannot deploy our validated infrastructure you should probably follow the approach described in Installation - Basic (On-Prem Infrastructure). You will need to do this if you have an on-premises or restricted cluster. This might be because you are unable to install AWX (which pulls images from docker hub) or your security group rules prevent you from installing our EFS provisioner.

A basic cluster contains only the essential infrastructure components to support the Fragalysis Stack and includes the deployment of a single, production-grade (versioned), Stack.

The following schematic illustrates what we’ll be installing, which includes a number of infrastructure components as well as the Graph and Fragalysis Stack: -
From this point we assume you have a viable cluster, what follows are the steps required to deploy a single Fragalysis Stack and associated Graph database to that cluster:

1.4.1 Installing the Infrastructure (Basic)

The Fragalysis Stack depends on Kubernetes services and objects created by our ansible-infrastructure project. A basic set of parameters that can be used to create a basic infrastructure (with and without Keycloak). 

**Note:** Allow **15 minutes** to complete this task, which includes installation of a PostgreSQL database and AWX.

Before we begin we must be in possession of the cluster’s kubeconfig file.

Start in a suitable working directory on your control machine (desktop or laptop) and and create and enter a Python 3 virtual environment:

```
$ mkdir -p ./im-demo
$ cd im-demo
$ conda create -n im-demo python=3.8
$ conda activate im-demo
```

Clone the infrastructure project and checkout the stable revision used for the demo and build the HTML-based documentation:

```
```
Fragalysis Stack (Kubernetes)

Note: Try to use the latest tag that’s available. At the time of writing it was tags/2020.28.

With documentation built, the root of it will be found at doc/build/index.html.

Follow the infrastructure project’s **Getting Started** guide and then its **Creating the Infrastructure** guide before returning here.

Once the infrastructure is installed you should be able to navigate to the AWX application server using the hostname you gave it.

With this done we can move to *Configuring AWX (Basic)*.

### 1.4.2 Configuring AWX (Basic)

Note: Allow 5 minutes to complete this task, which involves configuring and checking the AWX application server

Configuration of the AWX server is achieved with a parameter file in the Informatics Matters DLS Kubernetes GitHub repository and the Ansible Galaxy **AWX Composer Role**.

Clone the project into the working directory you created while following the **Installing the Infrastructure (Basic)** guide:

```bash
$ cd <working directory>
$ git clone https://github.com/InformaticsMatters/dls-fragalysis-stack-kubernetes.git
$ cd dls-fragalysis-stack-kubernetes
$ git checkout tags/2020.38
$ pip install -r requirements.txt
$ ansible-galaxy install -r role-requirements.yaml
```

Note: Try to use the latest tag that’s available. At the time of writing it was tags/2020.38.

The demo configuration will create the following objects:

- An organisation
- Credentials
- A team
- A demo user
- Projects
- Job Templates

Start by copying the `config-basic-template.yaml` file to `config-basic.yaml` (which is protected from being committed) and then review it and provide values for all the `SetMe` instances in the file.

The file defines a `tower` variable, used by our **AWX Composer** Ansible Galaxy role.
**Warning:** Before configuring the AWX server you will need an AWS user’s access credentials (typically for S3 access) and credentials for the Kubernetes cluster. Providing values for these will result in the Composer creating aws (SELF) and k8s (SELF) credentials in the AWX server that playbooks rely on in order to deploy the Fragalysis Stack. The AWS user account permissions that are required will ultimately depend on the container images that you intend to deploy. For example, if you expect to use AWS S3 as a source for Fragalysis Graph and Media the container we run to do this will require the AWS account to have the AmazonS3ReadOnlyAccess permission.

You will have to provide suitable environment variables for the built-in credentials:

```bash
$ export AWS_ACCESS_KEY_ID=00000000
$ export AWS_SECRET_ACCESS_KEY=00000000

$ export K8S_AUTH_HOST=https://1.2.3.4:6443
$ export K8S_AUTH_API_KEY=kubeconfig-user-abc:00000000
$ export K8S_AUTH_VERIFY_SSL=no
```

You can now configure the AWX application server using the infrastructure playbook and the `config-basic.yaml` file. From the root of your clone of the dls kubernetes repository run:

```bash
$ ansible localhost \
   -m include_role -a name=informaticsmatters.awx_composer \
   -e @awx-configuration/config-basic.yaml
```

Once complete you should be able to login to the AWX server and navigate to the Templates page and see all the available Job Templates, illustrated in the following example screenshot.

The Job Templates you see will depend on the configuration you’ve applied.
1.4.3 Deploying the Fragment Graph

**Note:** Allow 2 hours to install all of the applications.

With the AWX server configured we can now run the **Job Templates** that are responsible for deploying the various applications.

Start by logging into the AWX application server as the demo user demo. From there you should be able to navigate to the **Templates** screen where all the templates are presented to you.

**Note:** Allow 2 hours to complete this task.

Deploy the Fragmentation graph by launching the **Fragmentation Graph** template.
The jobs have been configured to first present a confirmation dialogue box so that you can adjust some key job variables before they run. For example, the Fragmentation Graph job allows you to provide a path to the graph data you want to deploy (using the `graph_bucket_path` variable).

Acknowledge the dialogue (clicking Next) and then the Launch button.

As the graph initialisation can take some time the job does not (at the time of writing) wait for the graph to initialise. We therefore use the `kubectl` command-line to check on the status of the graph before moving on. Check that the graph namespace exists:

```
$ kubectl get namespace/graph
NAME   STATUS    AGE
graph  Active  7s
```

And then watch the Graph Pod status until it’s Running. The graph contains an initialisation container used to download the graph data to the cluster:

```
$ kubectl get pod/graph-0 -n graph -w
NAME     READY STATUS    RESTARTS AGE
graph-0  0/1    Init:0/1 0 14s
graph-0  0/1    Init:0/1 0 95s
graph-0  0/1    Init:0/1 0 100s
graph-0  0/1    PodInitializing 0 108s
graph-0  1/1    Running   0 114s
```

Once you see Running the Pod has started and you can `ctrl-c` from the command.
Fragalysis Stack (Kubernetes)

The graph needs to *import* the downloaded files into a graph database, which can take a significant length of time, depending on the data that’s been downloaded.

You can *follow* the Graph Pod’s logs and wait for the import process to complete. The graph import typically involved 4 stages that are easily followed from the logs.

The output here has been truncated because there is a lot of it.

Importantly, to be confident the deployment has worked, you must see:

- A section starting (1/4) Node import
- A section starting (2/4) Relationship import
- A section starting (3/4) Relationship linking
- A section starting (4/4) Post processing

And, finally, you’re waiting to see the word *Finished.* issued by the `cypher-runner.sh` script:

```bash
$ kubectl logs pod/graph-0 -n graph -f
[...]
(1/4) Node import 2020-09-16 03:18:22.955+0000
Estimated number of nodes: 40.16 M
Estimated disk space usage: 8.64 GB
Estimated required memory usage: 1.49 GB
.......... .......... .......... .......... .......... 5% 4s 813ms
.......... .......... .......... .......... .......... 10% 3s 609ms
.......... .......... .......... .......... .......... 15% 3s 405ms
.......... .......... .......... .......... .......... 20% 3s 406ms
[...]
(4/4) Post processing 2020-09-16 04:13:13.062+0000
Estimated required memory usage: 1020.01 MB
......... .......... .......... .......... .......... 5% 7s 601ms
.......... .......... .......... .......... .......... 10% 11s 413ms
.......... .......... .......... .......... .......... 15% 12s 209ms
.......... .......... .......... .......... .......... 20% 3s 906ms
[...]
2020-03-19 14:25:08.527+0000 INFO ======== Neo4j 3.5.5 ========
2020-03-19 14:25:08.532+0000 INFO Starting...
2020-03-19 14:25:14.865+0000 INFO Bolt enabled on 0.0.0.0:7687.
2020-03-19 14:25:16.444+0000 INFO Started.
(cypher-runner.sh) Thu Mar 19 14:26:05 UTC 2020 Setting neo4j password...
(cypher-runner.sh) Thu Mar 19 14:26:07 UTC 2020 No legacy script.
(cypher-runner.sh) Thu Mar 19 14:26:07 UTC 2020 Trying /data/cypher-script/cypher-script.once...
(cypher-runner.sh) Thu Mar 19 14:26:08 UTC 2020 No .always script.
(cypher-runner.sh) Thu Mar 19 14:26:08 UTC 2020 Touching /data/data-loader/cypher--runner.executed...
```

Once you see that you can `ctrl-c` from the *follow* command and continue with the remaining applications.

### 1.4.4 Deploying the Stack
Note: Allow 45 minutes to complete this task. 5 minutes for the stack and 40 minutes for the initial (ALL_TARGETS) data load.

With the graph installed we can now start the Fragalysis Stack and its Data Loader.

Deploy Fragalysis by launching the Fragalysis Stack template.

As the stack initialisation is a little more deterministic (and short) the job waits for the stack to become ready before finishing. When this job finishes you know the stack is “up and running”.

You can’t use the stack without any target data so you now need to run the Data Loader.

Deploy the loader by launching the Fragalysis Stack Data Loader template (see below).
This job will also wait for the loader to complete. As we’re running a typical **ALL_TARGETS** load this will take around 40 minutes. The job will time-out after an hour.

Once complete you should be able to navigate to the application by navigating to the URL you used for `stack_hostname`.

### 1.5 Installation - Basic (On-Prem Infrastructure)

Ideally you deploy our validated infrastructure following the *Installation - Basic (Built-in Infrastructure)* approach but, if you cannot deploy this way this section describes the installation for on-premises or restricted clusters. i.e. you may not be able to install AWX (which pulls images form docker hub) or you security group rules prevent you from installing our EFS provisioner.

A *basic* cluster contains only the essential infrastructure components to support the Fragalysis Stack and includes the deployment of a single, production-grade (versioned), Stack.

The following schematic illustrates what we’ll be installing. the important aspect of an *on-premises* installation is that you are responsible for the infrastructure, which consists of security policies, storage provisioners, certificate management and an ingress controller: -
From this point we assume you have a viable cluster and suitable infrastructure. That means that you have the following:

- An NGINX ingress controller and associated Network Load balancer
- A RWX storage class (AWS EFS in our case)
- A RWO storage class (AWS GP2 in our case)
- Pod Security Policies
- Certificate management

What follows are the steps required to deploy a single Fragalysis Stack and associated Graph database to that cluster:

### 1.5.1 Command-Line Requirements (Control Machine)

Requirements for the machine you’ll be running ansible playbooks from.

1. **Applications.** the control machine will need: -
   - Python 3.8
   - Git

2. **GitHub.** In order to deploy the infrastructure the control machine will need access to GitHub where Ansible playbooks and roles are located, specifically: -
Fragalysis Stack (Kubernetes)

- InformaticsMatters/ansible-infrastructure
- InformaticsMatters/dls-fragalysis-stack-kubernetes

3. Ansible Galaxy. In order to configure the infrastructure AWX server the control machine will need access to Ansible Galaxy

4. LENS. Lens is respectable Kubernetes IDE. We use it to monitor our own clusters and, if you do not currently use a Kubernetes dashboard or IDE, you might want to use it too.

5. Credentials. You will need credentials that allow admin privilege access to the Kubernetes cluster and, if using AWS S3 as the origin for Graph fragment and Fragalysis Stack media data, a user with Get access for your chosen AWS S3 bucket.

6. A basic understanding of Ansible (v2.9) would be an advantage - e.g. playbooks, roles, role structure and variable definitions.

7. An understanding of YAML files.

Virtual environment

A lot of our work, and Ansible, will require the execution of Python scripts. Typically this is done from within a virtual environment. With the above requirements clone this repository and follow its README.md Preparation Notes.

1.5.2 Deploying the Fragment Graph

Note: Allow 2 hours to install all of the applications.

The playbooks that we'd normally run from AWX can be executed form the command line. That’s what we’ll be doing here.

The steps we’ll follow here are:

- Clone the Graph playbook repository
- Create a parameter file to satisfy your cluster
- Run the playbook

Clone the graph repo

The repository contains playbooks and roles for the deployment of a neo4j graph database and associated fragmentation data. From your virtual environment clone it and (ideally) switch to the most recent tag:

```bash
$ git clone https://github.com/InformaticsMatters/docker-neo4j-ansible
$ cd docker-neo4j-ansible
$ git checkout tags/2.4.4
```

Create parameters

The Graph deployment is flexible and is controlled by a number of Ansible variables. To control your deployment you are likely to have to define your own set of variable values in a parameter file - not all of those that are set in the repo may be of use to you.

A parameters-template.yaml file contains a small set of significant ones.
Copy the example in the repository, inspect it and set/change any values you need to:

```bash
$ cp parameter-template.yaml parameter.yaml
[edit parameter.yaml]
```

`parameter.yaml` is protected from being committed by the project’s `.gitignore` file.

The `parameter-template.yaml` contains only an example set of role variables - you should familiarise yourself with the role and inspect the other parameters (in the role’s `defaults/main.yaml` and `vars/main.yaml`), just in case you also need to adjust them.

**Run the playbook**

With a set of parameters created, deploy the Graph using the `site.yaml` playbook:

```bash
$ ansible-playbook -e @parameter.yaml site.yaml
[...]
```

As the graph initialisation can take some time the job does not (at the time of writing) wait for the graph to initialise. We therefore use the `kubectl` command-line (in a separate terminal/shell) to check on the status of the graph before moving on.

First, check that the graph namespace exists:

```bash
$ kubectl get namespace/graph
NAME   STATUS AGE
graph   Active 7s
```

And then watch the Graph Pod status until it’s Running. The graph contains an initialisation container used to download the graph data to the cluster:

```bash
$ kubectl get pod/graph-0 -n graph -w
NAME   READY STATUS    RESTARTS AGE
graph-0 0/1 Init:0/1 0     14s
graph-0 0/1 Init:0/1 0     95s
graph-0 0/1 Init:0/1 0    100s
graph-0 0/1 PodInitializing 0   108s
graph-0 1/1 Running 0     114s
```

Once you see Running the Pod has started and you can `ctrl-c` from the command.

The graph needs to import the downloaded files into a graph database, which can take a significant length of time, depending on the data that’s been downloaded.

You can follow the Graph Pod’s logs and wait for the import process to complete. The graph import typically involved 4 stages that are easily followed from the logs.

The output here has been truncated because there is a lot of it.

Importantly, to be confident the deployment has worked, you must see: -

- A section starting (1/4) Node import
- A section starting (2/4) Relationship import
- A section starting (3/4) Relationship linking
- A section starting (4/4) Post processing

And, finally, you’re waiting to see the word Finished. issued by the `cypher-runner.sh` script:
$ kubectl logs pod/graph-0 -n graph -f

(1/4) Node import 2020-09-16 03:18:22.955+0000
Estimated number of nodes: 40.16 M
Estimated disk space usage: 8.64 GB
Estimated required memory usage: 1.49 GB

.......... .......... .......... .......... .......... 5% 4s 813ms
.......... .......... .......... .......... .......... 10% 3s 609ms
.......... .......... .......... .......... .......... 15% 3s 405ms
.......... .......... .......... .......... .......... 20% 3s 406ms

(2/4) Node import 2020-09-16 03:18:22.955+0000
Estimated number of nodes: 40.16 M
Estimated disk space usage: 8.64 GB
Estimated required memory usage: 1.49 GB

.......... .......... .......... .......... .......... 5% 4s 813ms
.......... .......... .......... .......... .......... 10% 3s 609ms
.......... .......... .......... .......... .......... 15% 3s 405ms
.......... .......... .......... .......... .......... 20% 3s 406ms

(3/4) Node import 2020-09-16 03:18:22.955+0000
Estimated number of nodes: 40.16 M
Estimated disk space usage: 8.64 GB
Estimated required memory usage: 1.49 GB

.......... .......... .......... .......... .......... 5% 4s 813ms
.......... .......... .......... .......... .......... 10% 3s 609ms
.......... .......... .......... .......... .......... 15% 3s 405ms
.......... .......... .......... .......... .......... 20% 3s 406ms

(4/4) Post processing 2020-09-16 04:13:13.062+0000
Estimated required memory usage: 1020.01 MB

.......... .......... .......... .......... .......... 5% 7s 601ms
.......... .......... .......... .......... .......... 10% 11s 413ms
.......... .......... .......... .......... .......... 15% 12s 209ms
.......... .......... .......... .......... .......... 20% 3s 906ms

2020-03-19 14:25:08.527+0000 INFO ======== Neo4j 3.5.5 ========
2020-03-19 14:25:08.532+0000 INFO Starting...
2020-03-19 14:25:14.865+0000 INFO Bolt enabled on 0.0.0.0:7687.
2020-03-19 14:25:16.444+0000 INFO Started.

Once you see that you can `ctrl-c` from the `follow` command and continue with the remaining applications.

1.5.3 Deploying the Stack

**Note:** Allow **45 minutes** to complete this task. 5 minutes for the stack and 40 minutes for the initial (ALL_TARGETS) data load.

With the graph installed we can now start the Fragalysis Stack and its Data Loader.

The playbooks that we’d normally run from AWX can be executed form the command line. That’s what we’ll be doing here.

The steps we’ll follow here are:

- Clone the Fragalysis Kubernetes playbook repository (this repo)
- Create a parameter file to satisfy your cluster
- Run the stack installation playbook
- Run the stack loader
Clone the stack repo

The repository contains playbooks and roles for the deployment of the stack, its loader and more. From your virtual environment clone it and (ideally) switch to the most recent tag:

```
$ git clone https://github.com/InformaticsMatters/dls-fragalysis-stack-kubernetes
$ cd dls-fragalysis-stack-kubernetes
$ git checkout tags/2020.51
```

Create parameter files

The Stack and its loader are controlled by a number of Ansible variables. To control your deployment you are likely to have to define your own set of variable values in parameter files - not all of those that are set in the repo may be of use to you.

You will probably want to create separate parameter files for the stack playbook and its loader:

- `parameter-stack.yaml`
- `parameter-loader.yaml`

There are a lot of playbooks and roles so we have not (as yet) provided any examples but a typical set that you need to set for the stack are as follows:

```yaml
---
all_image_preset_pullsecret_name: ''
stack_hostname: example.com
stack_sa_psp: im-core-unrestricted
stack_image_registry: docker.io
stack_image_tag: latest
stack_replicas: 2
stack_include_sensitive: no
stack_namespace: fragalysis-production
stack_is_for_developer: no
stack_media_vol_storageclass: efs
busybox_image_registry: docker.io
database_vol_storageclass: gp2
graph_hostname: graph.graph.svc
graph_password: (the password you used for the graph deployment)
```

Also, create a set of variables to control the stack loader:

```yaml
---
all_image_preset_pullsecret_name: ''
loader_type: s3
loader_data_origin: 2020-09-15T16:00Z
loader_s3_bucket_name: im-fragalysis
loader_s3_image_registry: docker.io
loader_s3_image: informaticsmatters/fragalysis-s3-loader
loader_s3_image_tag: 1.0.7-2
loader_include_sensitive: no
```

(continues on next page)
Run the stack playbook

With a set of parameters created, deploy the Stack using the \texttt{site-fragalysis-stack.yaml} playbook:

\begin{verbatim}
$ ansible-playbook -e @parameter-stack.yaml site-fragalysis-stack.yaml
\end{verbatim}

The stack consists (mainly) of a namespace, a Fragalysis Stack \texttt{StatefulSet}, and a postgres database that it uses. The stack deployment waits for the Graph before it completes initialisation so you must have previously installed the graph before you deploy any stacks.

Use \texttt{kubectl} or \texttt{Lens} to make sure the stack the database are running before trying to run the loader.

Run the stack loader playbook

You can’t use the stack without any target data so you now need to run the Data Loader.

Run the loader (assuming you’ve setup or have access to a suitable AWS S3 bucket and data) playbook:

\begin{verbatim}
$ ansible-playbook -e @parameter-loader.yaml site-data-loader.yaml
\end{verbatim}

This playbook will wait for the loader to complete. If you’re loading a typical \texttt{ALL\_TARGETS} load this will take around 40 minutes to complete. The playbook will time-out after an hour.

Once complete you should be able to navigate to the application by navigating to the URL you used for \texttt{stack\_hostname}.

1.5.4 Example sets of variables (AWS)

If you’re deploying the graph and stack \texttt{on\_prem} the following sets of example variables provide a basic set of minimal values that you might want to use.\footnote{These would be suitable for a site where you’ve provided your own \texttt{infrastructure} (i.e. nginx and EFS) but are using our \texttt{core}, i.e. our Pod Security Policy and have access to graph and stack data on an S3 bucket.}

The following also assumes that your images\footnote{The image names used here are for illustrative purposes only. If in doubt, ask us for a set of up-to-date images and tags.} are hosted on a private registry (artifactrepo.xzy.com/\texttt{docker.io} in these snippets) and you have created a \texttt{pull secret} called \texttt{secret-artifactory} in namespaces that you’ve pre-allocated.

Infrastructure

You could run the infrastructure playbook with the following:

\begin{verbatim}
\texttt{cm\_state: absent}
\texttt{cinder\_state: absent}
\texttt{efs\_state: absent}
\texttt{nfs\_state: absent}
\texttt{infra\_state: absent}
\end{verbatim}
Graph

You could run the graph playbook with the following:

```yaml
graph_password: -SetMe-
graph_loader_image: informaticsmatters/neo4j-s3-loader
graph_image: informaticsmatters/neo4j
graph_tag: '3.5.20'
graph_volume_size_g: 800
graph_pvc_storage_class: gp2
graph_bucket: im-fragnet
graph_bucket_path: combination/3
graph_wipe: no
graph_pagecache_size_g: 80
graph_mem_request_g: 96
wait_for_bind: no
graph_image_registry: artifactrepo.xzy.com/docker.io
all_image_preset_pullsecret_name: secret-artifactory
```

Stack

You could run the stack playbook with the following:

```yaml
stack_image: xchem/fragalysis-stack
stack_image_tag: '2.0.5'
stack_is_for_developer: false
stack_media_vol_size_g: 12
stack_media_vol_storageclass: <YOUR EFS STORAGE CLASS NAME>
stack_namespace: fragalysis-production
stack_replicas: 2
stack_include_sensitive: no
database_vol_storageclass: gp2

graph_hostname: graph.graph.svc
graph_password: <THE SAME ONE YOU USED TO DEPLOY THE GRAPH>
stack_cert_issuer: ''
stack_hostname: fragalysis.xyz.com
# Registries
stack_image_registry: artifactrepo.xzy.com/docker.io
database_image_registry: artifactrepo.xzy.com/docker.io
busybox_image_registry: artifactrepo.xzy.com/docker.io
# Pull secret name
all_image_preset_pullsecret_name: secret-artifactory
```

Loader

You could run the loader playbook with the following:

```yaml
""
Fragalysis Stack (Kubernetes)

```
loader_s3_bucket_name: im-fragalysis
loader_data_origin: 2020-09-15T16
loader_type: s3
loader_s3_image: xchem/fragalysis-s3-loader
loader_s3_image_tag: '2.0.5'
loader_include_sensitive: no

stack_is_for_developer: false
stack_namespace: fragalysis-production

# Registries
loader_s3_image_registry: artifactrepo.xzy.com/docker.io

# Pull secret name
all_image_preset_pullsecret_name: secret-artifactory
```

1.6 mArh Screenshots

1.6.1 Target List

If you’ve loaded data for the mArh target using the example data origin 2020-09-15T16 then, when you navigate to the stack instance you should be presented with mArh in the target list (there may be others depending on what else you may have loaded) as shown below:

![mArh Target List](image)

1.6.2 Hit Navigator

If you click on the mArh target you should see the Hit screen, with a cluster selector and navigator, as shown below:

![Hit Navigator](image)
To test the graph database connection de-select the EU0034_0_A molecule by clicking its dark-blue “L” (ligand) and “P” (sidechains) boxes and instead click “L” and “P” for the 2nd molecule (EU0046_0_A). Now also click the “V” (vectors) button.

With this done, if your database connection is working, you should should see arrows and cylinders attached to the molecule, as shown below:—

---

1 Initial queries for the graph database often take a few seconds so you might need to wait a while for the first query to complete.
1.6.3 Alternative Molecules

Click the cylinder:

And you should then see a number of alternative molecules presented to you, extracted from the fragment database, on the right-hand side of the application window:

If that works then your stack deployment has been successful!
1.7 AWX Fundamentals

A brief overview of AWX and our use of it.

AWX is a task engine built on top of Ansible that is able to simplify, manage monitor application deployments through the use of Ansible playbooks present in repositories (GitLab, GitHub etc.). We’re not going to go into detail about how the AWX server we use is fully configured, instead we’re just going to explore some key aspects that are essential to deploying the Fragalysis Stack.

AWX provides Job definitions that allow the execution of Ansible Playbooks located in Git repositories. AWX executes these plays on Hosts defined in Inventories within AWX.

AWX is useful layer on top of raw Ansible because it provides:

1. A centrally administered authoritative catalogue of Jobs (plays)
2. Controlled access to Jobs through the use of Organisations, Teams and Users. Jobs can be setup so they’re only available to permitted Users, Teams or Organisations, with varying levels of Access. Full access to a Job can be granted with Admin permissions or limited to Execution-only rights for named Users (or Teams or Organisations).
3. Job execution History
4. Job failure Notifications (e.g. via email)

AWX is a task engine we use to execute Ansible playbooks that manage Kubernetes objects.
1.7.1 Objects

To run a playbook in AWX you typically need to create the following minimum set of objects: -

- An **Organisation**
- A **Team**
- A **User**
- An **Inventory** and **Hosts**, often something simple like `localhost`
- **Credentials**, like Kubernetes cluster credentials
- A **Project**, which is a reference to a GitHib project containing the play
- A **Job Template**, that joins the inventory, credentials and project together with the opportunity to over-ride default variables in the corresponding play

**AWX in Operation**

The AWX server is deployed with our Kubernetes Infrastructure, along with a Keycloak instance and PostgrSQL database. It provides a web interface that allows Jos to be executed that deploy Fragalysis application components (like the Graph database and Fragalysis itself). Deployment of applications often depends on: -

- Access to **playbooks** resident in external Git repositories
- Access to publicly available **container images**
- Configuration **data** (typically graph and Fragalysis data) held externally (e.g. on AWS S3)
Because the Production and Development clusters serve two very different purposes, and one is much more sensitive than the other, separate AWX servers exist - one in the Production cluster and one in the Development cluster. Users may not have access to both and Developers are unlikely to have access to the Production AWX server.
2.1 Continuous Development

Fragalysis Stack development procedures for Kubernetes deployment and the role of Travis in the Fragalysis Stack CI process.

The OpenShift deployment of the Stack used Jenkins as the CI/CD build framework, relying on the container registry and build capabilities provided by OpenShift. Kubernetes has no registry as such and although you can install Jenkins its role is diminished and it’s of much less value than its OpenShift counterpart.

Instead, in the Kubernetes world, without a clear CI/CD framework emerging, we have switched to relying on an external build process (Travis) to build, test and deploy the Fragalysis Stack container images.

Travis is an external cloud-based service, free for open-source projects that is mature used by many to test and build software. It is programmed through the use of .travis.yml files placed in the root directory of GitHub projects.

We have added facilities to chain builds (for one Travis GitHub project to trigger another) using our Trigger Travis repository and to also deploy container images to the cluster using pre-deployed AWX Jobs using our Trigger AWX repository.

Why not use Jenkins?

- Jenkins is indeed powerful but it relies on Agents in order to run Jobs. Setting these agents up requires effort. Travis runs in the cloud where it has agents, of all kinds, that are simple and easy to configure.

- Jenkins is a complex service with a need for persistence (volumes) that would eat away at the Kubernetes cluster in both CPU, disk and cost. Travis does not require resources in the Kubernetes cluster

- Job control in Jenkins is defined through the Groovy language. It’s powerful but it’s yet another language and its syntax is not the easiest to master. Travis is programmed in YAML, a syntax many are already familiar with. It is simple, and easy to read.

- Management and configuration of the Jenkins server is not trivial and incurs management and maintenance costs. Travis needs no management.
2.1.1 Fragalysis Stack Repositories

The stack is distributed as two container images, a Loader and a Stack. There are five GitHub repositories involved in the build of these two images:

- `fragalysis`
- `fragalysis-backend`
- `fragalysis-frontend`
- `fragalysis-loader`
- `fragalysis-stack`

The by-product of each repository is:

- **fragalysis** The output of the `fragalysis` repository is a small package of Python code, written to PyPI when the repository is tagged. The package is part of the `fragalysis-backend` image’s Python requirements.

- **fragalysis-backend** The output of the `fragalysis-backend` is a container image, written to Docker Hub. This image is used as a FROM image in both the Loader and Stack. The backend FROM image is based on `informaticsmatters/rdkit-python-debian:latest`.

- **fragalysis-frontend** The output of the `fragalysis-frontend` is nothing. The code is instead cloned into the container image of the `fragalysis-stack` when it is built.

- **fragalysis-loader** The output of the `fragalysis-loader` is a container image, written to Docker Hub. It uses the image produced by the `fragalysis-backend` as its FROM image.

- **fragalysis-stack** The output of the `fragalysis-stack` is a container image, written to Docker Hub. Like the Loader it uses the image produced by the `fragalysis-backend` as its FROM image.

2.1.2 Build example (master)

Let’s see how Travis works for the Fragalysis Stack by exploring a simple example, where a user-change to a repository’s master branch results in the stack being re-built, illustrated by the following diagram.

---

1 Publishing to PyPi does not currently result in a trigger of the backend. It is something we can contemplate in the new development.
The diagram illustrates a user making a change (A) to the master branch of fragalysis-backend repository. The following steps occur, in approximate order:

1. Travis detects the change and creates a VM on which the build (and testing) takes place. The result of the build is a docker push to Docker Hub. The image pushed is xchem/fragalysis-backend:latest where the docker user is xchem, the project is fragalysis-backend and the tag is latest (the significance of these values will become important later).

2. At the end of the build of fragalysis-backend Travis is configured to trigger a build in the remote repository fragalysis-stack. There’s a new backend image so the stack, which depends on it, is instructed to build. It uses the Trigger Travis code to do this.

3. As the Loader also depends on the output of this build Travis also triggers* the fragalysis-loader to build.

4. The fragalysis-loader Travis session (triggered by the backend) builds and, as its output is a container image, the image is pushed to Docker Hub. The image pushed is xchem/fragalysis-loader:latest.

5. The fragalysis-stack Travis session (also triggered by the backend changes above) builds and its image is pushed to Docker Hub. The image pushed is xchem/fragalysis-stack:latest

---

2 This is achieved through a POST operation to the Travis REST API naming the downstream repository and passing in some extra material.
2.1.3 More scenarios (here be Dragons)

That’s a simplistic illustration of a build chain from one master branch rippling through the dependent builds on the master branch.

But software development’s more complicated than just changes to the master branch and, in these cases, Travis will need some help.

How does Travis know which repos to trigger?

This is the responsibility of the repository owner. Our Trigger Travis utility is used to simplify the calls the Travis API but the owner of each repository needs to know which repositories to trigger and simply adds calls to the Trigger Travis at a suitable point in their own .travis.yml file.

The mechanism is essentially a push-driven trigger from upstream repository to downstream. A downstream repository cannot monitor upstream repositories, the author has to know which repositories depend on their code.

Because Jenkins runs continuously it does allow Jobs to watch other builds (Jobs) that are upstream and trigger downstream builds (Jobs). But this advantage is considered insignificant compared to the disadvantages (discussed earlier).

How does a repo know what container tag to use?

By convention, in a CI/CD sense, automated builds on master produce container images tagged latest. The Travis build can be easily organised to produce a tag that is the branch name if the build is on a branch. Branch 1-defect might therefore produce images that are pushed to docker using the tag 1-defect

How do I instruct the downstream to use may image?

In our example we’ve assumed the branch being manipulated is master and in this very simple workflow we want all the dependent master branches to build resulting in their own latest images.

But what if you’re working on a defect on the backend, on a branch called 1-defect? Do you want to trigger a rebuild of the Stack’s latest image from fragalysis-backend:latest? No, you want the stack to use fragalysis-backend:1-defect as its FROM.

So this is where the Trigger Travis utility, the Travis REST API and your .travis.yml file in both your upstream and downstream repositories become a little more complex...

The downstream (Stack) repository’s .travis.yml file is configured to expect a number of environment variables, which have default values, namely:

- BE_NAMESPACE AND BE_IMAGE_TAG (defaulting to xchem and latest)
- FE_NAMESPACE AND FE_BRANCH (defaulting to xchem and master)

All the upstream repository’s .travis.yml has to do is ensure that it injects its own value for these variables using it’s STACK_VARS variable. For this example we’d set this to BE_IMAGE_TAG=1-defect, BE_NAMESPACE=xchem and the triggered build will produce for us a Stack image based on our 1-defect backend image.

Brilliant!

But hold on - the stack will be based on 1-defect while producing a latest.

We can add more logic to our downstream repository so that the tag it uses is actually based on the tag found in the BE_IMAGE_TAG value.
Simple . . . ish

**But what if you forget to set the variable?** After all, when you create your `backend` branch you need to adjust your own Travis settings to provide a value for the variable. If you forget (and you will) you’ll end up causing a new build of `latest` in the downstream projects that contains your (probably untested) patch. Not what others might expect from `latest`.

**What if I want to trigger a non-master downstream branch?**

That’s a very good question.

If I have a `1-defect` branch in the `upstream` build and I want to trigger the `1-defect` branch in the `downstream` project?

It’s solved by the `Trigger Travis` utility, which allows you to pass in a branch definition so that `Travis` build the branch you name rather than the default `master`.

Brilliant!

If you’re clever enough you could even pass this value on to `downstreams` of the `downstream`, but that doesn’t apply in our case and starts to get complex very quickly.

**But what if you forget to set the variable?** Mmmm . . . OK . . . I see a pattern emerging here.

Basically this is where it all gets rather messy, complex and complicated and unless you are very, very disciplined in your project organisation and development you should be treading extremely carefully.

**I have a fork of the frontend, how do I...**

Here we’d like changes in a branch of a fork of one repository to trigger the build of a branch in the fork of another repository...

STOP! It’s just getting mind-bendingly complex.

Mmmmm We’re starting to sink deeper into a very complicated world.

Hold on - `Jenkins` seemed fine. Have we lost something useful?

Yes . . . but that usefulness came with significant cost: -

`Jenkins` could do this easily because it was cloning the repositories and building them, while pushing to Docker registries while armed with keys to the xchem Docker Hub account. We had the secrets safely stored in `Jenkins`. That is something we cannot achieve in the `Travis` world - we can't give everyone a key, that’s not secure.

Also, creating OpenShift deployments per developer and configuring Jenkins takes several hours, probably half a day.

So here we have a situation that was easily solved in `Jenkins` and OpenShift that becomes enormously complicated (and probably impossible or at the very least extremely undesirable) in the `Travis` World.

It’s here we have to think about how developers develop code for the Fragalysis Stack and Kubernetes.

We need an altogether simpler approach.

### 2.1.4 Development Recommendation

For the main production images for STAGING (`latest`) and PRODUCTION (`tagged`) we...

1. . . . utilise `Travis` build triggers in the main `xchem` repositories. The build triggers are used exclusively for the automatic production of `latest` images on the `master` branch.

2.1. Continuous Development 35
2. Similarly, Travis builds tagged images on the main `xchem` repositories based on the presence of a release (or tag) in the repository. `fragalysis-backend:1.0.0` is automatically produced when the owner applies the tag `1.0.0` to the `fragalysis-backend` repository.

The main stack deployment is therefore automatic, continuous, fast but, above all, simple.

Individual developers...

3. ... work on branches of the main repositories or on branches of forks of the main repos.

4. No images are automatically produced from changes to branches or forks.

5. Developers are responsible for building their own container images and for pushing them to Docker Hub. Tina working on branch `1-defect` in a fork of the `fragalysis-frontend` repository is responsible for producing the corresponding stack image by (ideally) also forking and manipulating the `fragalysis-stack` repository so that it clones her frontend code rather than the code from `xchem/fragalysis-frontend`.

6. In order to deploy their project to Kubernetes (the subject of another Guide), users may push their container image to any Docker Hub namespace, project or tag. Tina can push her image as `xwz/stack-tina:1-defect` if she chooses. This works because she will have deployed her project to Kubernetes (now a developer responsibility) so that her cloud deployment’s stack should run using the image `xwz/stack-tina:1-defect` (rather than the default `xchem/fragalysis-stack:latest`). Tina can also select the version of the database she wants to use and the URL of the graph database. When she’s done she destroys the Kubernetes project.

The above places significant responsibility on the developer - they have to create the images, they have to push them, they have to create the Kubernetes deployments (subject of another guide) and they have to understand the build process.

But, this is a significantly simpler and a relatively pain-free route to supporting unlimited multi-developer deployments than could be achieved by any automatic system in the timescale available.

After all, if you’re expect to have 20 or 30 developers all on different forks and branches, all developing different aspects of the code, an automatic build system would be enormously complex, fragile and costly to maintain.

### 2.1.5 Development Examples

To further illustrate the knock-on effect of the above recommendation for individual developers, i.e. that developers are responsible for their own container images using repository forks and branches, a few examples follow.

The following relies on the use of standard Docker build arguments and the ability to use build-time args in the FROM statement, i.e. Docker v17.05 or later.

#### Developing Front-end (F/E) Code Example

Here you’re developing front-end code, relying on a published backend image and the existing stack implementation.
1. The developer forks `xchem/fragslysis-frontend`, into, say `alan/fragslysis-frontend` (A).
2. The developer creates a branch and clones it, e.g. 1-fix, in order to make changes (B).
3. The developer clones `xchem/fragslysis-stack` (C).
4. When a stack image is to be tested the developer builds the stack (locally) using Docker. This could be achieved through the use of a build script\(^3\) where the developer provides a suitable set of build-args, as shown (D).
5. Upon conclusion of development a pull-request on the f/e repository propagates the changes back to the XChem repo. The produced stack, built from a tagged b/e and the code in the developer’s 1-fix branch of their front-end repo fork, can then be pushed to Docker-hub and the Kubernetes cluster triggered to pull and run the updated code.

The diagram also illustrates how the XChem STAGING/latest Fragalysis Stack is built and deployed (automatically using Travis). This official stack uses a tagged b/e image (the same version in this example) but its build args (E) are such that it uses the master branch of the xchem project as the source of the front-end code\(^4\).

**Developing Back-end (B/E) Code Example**

Here you’re developing back-end code, relying on existing front-end and stack implementation.

---

\(^3\) The build script will help by forcing a pull of the dependent backend container image for example.

\(^4\) Ideally this would actually be a tag rather than master
Developing B/E Code

Here, in a less cluttered diagram:

1. The developer forks xchem/fragalysis-backend, into, say alan/fragalysis-backend (A)
2. The developer creates a branch and clones it, e.g. 1-fix, in order to make changes (B)
3. The developer clones xchem/fragalysis-stack (C)
4. When a stack image is to be tested the developer needs to build their own b/e image (D) (which they can optionally push to Docker hub) and then build the stack (locally), providing suitable build-args, as shown (E).
5. Upon conclusion of development a pull-request on the b/e repository propagates the changes back to the XChem repo.

Developing Stack Code Example

Here you’re developing stack code, relying on a published back-end image and front-end implementation.
1. The developer forks the fragalysis stack repository (say to a; an) (A)

2. The developer creates a branch and clones it, e.g. 1-fix, in order to make changes (B)

3. When a stack image needs to be tested the developer needs to build their own stack image, which is pushed to Docker hub (C) providing suitable build-args, as shown (D).

4. Upon conclusion of development a pull-request on the stack repository propagates the changes back to the XChem repo.

**Developing Everything Example**

Here you’re developing front-end, back-end and stack code.
This is essentially a combination of the three prior scenarios.

1. The developer forks each repository (say to alan) (A)

2. The developer creates a feature branch in each fork and then clones that to make changes (B). In the diagram we have branches 1-fix, 2-fix and 4-feature for the f/e, b/e and stack respectively.

3. When a stack is to be tested the developer first builds their own b/e (C) using minimal build arguments. The user then builds their own stack, from a clone of their code branch. Here you can see the stack is configured to use the alan/fragalysis-backend:2-fix image and a clone of the f/e 1-fix branch.

4. The pushed stack can then be deployed to the Kubernetes cluster.

5. Upon conclusion of development pull-requests for b/e, f/e and stack repositories are made in order to propagate the changes back to the XChem repos.

### 2.1.6 Development Prep and Cheat-Sheets

The following documents provide helpful start-up guides for development, the debugging of your deployed applications.

#### The Fragalysis Stack Loader

The *loader* is used to provision new Fragalysis Stack data (both Django media and the related database entries).

---

5 Automation to the image project from the project fork should be possible so the user may not have to specify anything in this case.
How it works

Data to be loaded is first placed on the Media NFS server, where an NFS directory acts as the root of all of the data (/nfs/kubernetes-fs-media). New data directories are typically named with the format YYYY-MM-DDTHH, e.g. 2020-06-03T09.

In order to load the data into a specific stack you simply run an appropriate Media Loader Job on the AWX server for your cluster, providing the data directory for the media as the Job’s loader_data_origin value (e.g. 2020-06-03T09).

The Job instantiates the loader container, which mounts the NFS directory and copies the data into /code/media/NEW_DATA in the Stack container before calling loader.py module in the loader Pod.

As well as an NFS loader a loaders also exists that will copy data from AWS S3, although this will require suitable credentials.

The NFS media volume (the ‘origin’)

You will need to put new data files in a subdirectory of /nfs/kubernetes-fs-media on the NFS server that has been commissioned in the STFC xchem-follow-up project.

- The NFS server address is 130.246.213.186 (at the moment this is not available from outside the project’s network).
- The server account for connection is fragalysis
- You will need the private part of the key-pair in order to access the server

Data and the directory it’s in must be available to all.

Running the loader

You will find a Media Loader Template Job on your cluster’s AWX server. On the production cluster you will find one for the Production stack (Production media Loader) and the Staging stack (Staging media Loader).

- Change the Job Template’s loader_data_origin variable value before running the Job. The value must match a sub-directory that has been populated on the media volume, e.g. 2020-06-10T09 (i.e. the sub-directory not the full path).

When executed, The Job runs and waits for completion of the loader.

2.2 Continuous Deployment

The Fragalysis Stack automated deployment mechanism and the role of Travis and AWX.

2.2.1 Travis

The various Fragalysis repositories are registered with Travis, which monitors each repository, spinning up VMs in response to changes in order to run build commands located in the project’s .travis.yml.

As well as the .travis.yml instructions, which are visible to all, the automated build relies on sensitive information that includes Docker Hub and AWX credentials. This information is provided in the corresponding repository’s Travis management console, where encrypted material is injected into the build using user-defined environment variables.
• Only commits to **master** and **tag** operations result in the above automated sequence. Developers working on their own branches or forks are required to build their own images.

### 2.2.2 AWX

Deployment relies on pre-configured Jobs on the corresponding cluster’s AWX server. For example, on the production cluster the continuous delivery described below relies on the following Jobs, runnable by the designated Travis CI/CD user:

- **Production Fragalysis Stack (Version Change)**
- **Staging Fragalysis Stack (Version Change)**

Prior to automated deployment the user is expected to have deployed the **Staging** and **Production** stacks using the following pre-configured jobs:

- **Production Fragalysis Stack**
- **Staging Fragalysis Stack**

These jobs are only expected to be run once.

### 2.2.3 Fragalysis Stack

Automated deployment of the Fragalysis Stack is achieved through the Travis CI/CD framework ans AWX. As changes are committed to the **master** branch of the stack's GitHub repositories (**xchem/fragalysis-frontend**, **xchem/fragalysis-backend** and the **xchem/fragalysis-stack** itself) Travis launches a VM and runs the build instructions located in each project’s `.travis.yml` file.

As an example, a typical sequence of **actions** that occur in response to a commit to the **master** branch of **xchem/fragalysis-frontend** repository can be seen illustrated in the following diagram and described below:
Deployment actions (from commits)

Although Travis launches a build for every change (regardless of branch) the automated actions shown above and described here only take place when a repository’s master branch changes (or is tagged).

1. A user accepts a Pull Request or makes a direct change on the master branch to the frontend code.
2. A few moments later Travis detects the change, clones the Frontend code and executes the commands in its .travis.yml. At the time of writing Frontend changes do not result anything being built from within its own repository. The frontend code is actually cloned into the Stack when it is built.
3. On success, the Frontend build’s instructions trigger the Stack’s Travis build.
4. The by-product of the Stack build is the fragalysis-stack container image. This is pushed to Docker Hub from the Travis build VM. Here we see the latest tag being used.
5. At the end of a successful Stack build, and a new latest image pushed to Docker Hub the Frontend .travis.yml has an instruction to trigger the launch of a pre-existing Job in the AWX server. This is achieved through the use of the AWX (Tower) CLI. The Job’s name is injected into the build using a Travis environment variable.

The Travis build waits for the AWX Job (the Stack’s version-change playbook in this case) to complete. This can take several minutes but by waiting any failure in deployment is immediately detected.

The build knows that the stack needs to be built because this dependency is hard-coded into the Frontend’s .travis.yml.
6. The AWX playbook (the Stack version change playbook in this case) executes. The playbook is cloned from its GitHub repository, known to AWX, into the AWX Pod on the Kubernetes cluster.

7. Playbook execution, using the Ansible built-in k8s module, results in a forced redeployment of the Stack’s Pods (managed as a multi-Pod StatefulSet). The Pods restart one-by-one in an orderly fashion until all the Pods have been replaced using a fresh pull of the latest container images. This roll-out strategy prevents any break in service.

Steps 2 to 7 above also take place when commits are made to the fragalysis-backend repository.

Every change to the Stack’s master branch results in the execution of the following AWX Job Template (on the PRODUCTION cluster’s AWX server): -

- Staging Fragalysis Stack (Version Change)

Every production-grade Tag on the Stack repo results in the execution of the following AWX Job Template (on the PRODUCTION cluster’s AWX server): -

- Production Fragalysis Stack (Version Change)

Performance

The time between a commit to the Frontend repository and the start of the roll-out of the changes in the cluster is around 8 or 9 minutes. Each Stack Pod takes around 3 minutes before it’s providing a service endpoint. A two-Pod StateFulSet will take around 15 minutes to fully deploy.

2.2.4 Fragalysis Loader

The Loader, like the stack above, is built in the same automated fashion. It is triggered by changes either to the master branch of the xchem/fragalysis-loader or via a Travis API trigger from a fragalysis-backend Travis build process, which takes place whenever the Backend master code changes.

2.2.5 The Travis-Trigger Utility

The Travis build trigger logic used by the repositories is provided by a small Python module, cloned into the build process from our Trigger Travis GitHub project.

2.2.6 The AWX-Trigger Utility

The Travis build trigger logic used by the repositories is provided by a small script that drives the Tower CLI, cloned into the build process from our Trigger AWX GitHub project.

\(^2\) A tag like 1.0.3, one that has three numbers separated by a period.
Development Processes

Fragalysis Stack deployment procedures and the role of AWX (Ansible Tower) in the Fragalysis Stack _kubernetes_ process.

The deployment of the Fragalysis Stack, i.e. its deployment to a cloud-based Kubernetes cluster are managed by Ansible playbooks (and Roles) located in the DLS Kubernetes GitHub project and an installation of an AWX server, a web-based user interface, REST API, and task engine built on top of Ansible, which will be used to initiate deployments.

Separate AWX servers exist on the Production and Development clusters. Developers normally only have access to the server on the Development cluster.

Deployments (plays) require a number of sensitive values (tokens, passwords and the like) which will need to be limited to specific Teams of Users. Management of this aspect of Ansible is complex but it is simplified through the use of AWX.

Essentially the development workflow (explained below) is: -

1. **Build** a Fragalysis Stack (and Loader) image
2. **Run** your “User (<YOU>)…” AWX Job
3. **Replicate** the postgres data into your stack (once)
4. **Replicate** the Django media data into your stack (once)
5. **Visit** your stack

Then repeat steps 1, 2 and 5 each time you build the stack.

### 3.1 The AWX Development Server

You will find the Production AWX server at [https://awx-xchem.informaticsmaters.org](https://awx-xchem.informaticsmaters.org).
3.1.1 Jobs controlling the Graph Database

Graph Blah
Graph [DESTROY] Blah

3.1.2 Jobs Controlling Developer Stacks

TBD Blah

3.2 The AWX Production Server

You will find the Production AWX server at https://awx-xchem-production.informatics-matters.org. Only a limited number of users will have access to this server as it is used to control the Staging and Production deployments of the Fragalysis Stack and the Production Graph database.

There are Jobs to update the Staging and Production stacks but no Jobs to destroy (delete) them. Both the Production and Staging stacks are expected to be persistent deployments that change but they are never removed.

3.2.1 Jobs controlling the Graph Database

Graph Blah
Graph [DESTROY] Blah

3.2.2 Jobs Controlling the Production Stack

Production Fragalysis Stack Blah
Production Fragalysis Stack (Version Change) Blah
Production Fragalysis Stack Database Replicator (CronJob) Blah
Production Fragalysis Stack Media Replicator (CronJob) Blah
Production Media Loader Blah

3.2.3 Jobs Controlling the Staging Stack

Staging Fragalysis Stack Blah
Staging Fragalysis Stack (Version Change) Blah
Staging Fragalysis Stack Database Replicator (CronJob) Blah
Staging Fragalysis Stack Media Replicator (CronJob) Blah
Staging Media Loader Blah
3.3 Preparing for Kubernetes Development

In order to develop applications you are likely to need to debug **Pods** (Containers), poke-around inside them and occasionally restart them. You can do this with the command-line (using *kubectl*) or with the *lens* IDE.

Before starting with either you will need a configuration file, called a *kube config*. Your system administrator should be able to provide you with one for the cluster you’re deploying to.

Armed with this, follow instructions based on whether you’ll be using *kubectl* or *lens*.

### 3.3.1 Preparing for *kubectl*

You will need to install a compatible version of *kubectl* and then create your configuration file.

**Installing kubectl**

Follow the Kubernetes instructions. If in doubt always install the very latest version. It must be at least as good as the one used by the cluster (which is probably v1.17.5).

**Setup your KUBECONFIG file**

Kubernetes configurations (small YAML files) are usually located in your `$HOME/.kube` directory. Take the configuration you’ve been given and place it in a new file. For example the configuration for the XChem Development cluster should be placed in `$HOME/.kube/config-xchem-developer`. Set the KUBECONFIG environment variable to the new file:

```
$ export KUBECONFIG=$HOME/.kube/config-xchem-developer
```

You can test the config with *kubectl* with a simple version command, which should display the client and server versions (1.18.0 and 1.17.5 respectively in the following example):

```
$ kubectl version
Server Version: version.Info{Major:"1", Minor:"17", GitVersion:"v1.17.5", GitCommit:"e0fccaf69541e3750d460ba0f9743b90336e24f", GitTreeState:"clean", BuildDate:"2020-04-16T11:35:47", GoVersion:"go1.13.9", Compiler:"gc", Platform:"linux/amd64"}
```

### 3.3.2 Preparing for *lens*

You will need to install *lens*, a neat community-driven IDE for Kubernetes, and then provide your kubernetes configuration file.

**Installing lens**

Refer to the *lens* website for installation instructions. You should install version 3.4 or better.
Provide your KUBECONFIG

The first time you start lens you will need to provide a cluster configuration.

Click the Add clusters icon in the application side-bar...

Then select Custom ... from the Choose config: drop-down, and then paste the content of the Kubernetes configuration file into the Kubeconfig: panel and then finish by clicking Add Cluster(s) ...
If successful you should be presented with an overview of the cluster workloads (as shown below).
Now that you’re connected to the cluster through Lens you can quickly navigate to the **Workloads** (which includes Pods, Deployments, StatefulSets and Jobs), **Configuration** (for ConfigMaps, Secrets etc.), **Network** (for Services and Ingresses), **Storage** (so see PersistentVolumes and PersistentVolumeClaims) and **Namespaces** (to see all the namespaces).

**Adding a cluster icon**

If you want to add a distinctive icon for the cluster you can. Simply right-click the exiting cluster icon in the side-bar and select Settings. Use the **Browse for new icon...** to select an icon of your choice.

Two icons can be found in this repository that can be used for the Development and Production clusters (which can be found in the project’s `lens/icons` directory).

### 3.4 Cheat Sheets

Some handy commands for kubectl and `lens`.

#### 3.4.1 A Developer Cheat-Sheet for kubectl

Some useful kubectl command that might be useful during development. This is only a taster* of common commands, you can also refer to the official Kubernetes Cheat Sheet for many more.
### Setting the default namespace

To quickly restrict your commands to a given namespace, e.g. your own Fragalysis Stack deployment you can use the `set-context` command. If your stack is deployed to `stack-alan-default` you can set the default namespace to it with the following command: -

```
$ kubectl config set-context --current --namespace=stack-alan-default
```

From this point forward you need to add `-n stack-alan-default` to each command you want to run. This is probably the most important command to you should run - by setting the default namespace to that for your stack it’ll be much more difficult for you to make the mistake of adjusting the content of namespaces being used by others.

The following *cheats* assume that you have set the default namespace.

### List namespace content

You can see all the (typical) objects in a namespace with the following command:

```
$ kubectl get all
```

It’ll display information about **Pods, Services, Jobs, CronJobs, Deployments** and **StatefulSets**. Ironically all doesn’t show you everything.

### List more namespace content (PVCs, Secrets, Ingress)

You can display information about any object in a namespace as long as you know what it is you want to see. Some common objects not present in the default set of **all** objects are volume claims (a **pvc**), secrets and external ingress paths. You can use recognised abbreviations or the full object name:

```
$ kubectl get pvc
$ kubectl get svc
$ kubectl get service
$ kubectl get ing
```

And combine them:

```
$ kubectl get pvc,svc,ing
```

### Display Pod logs

Kubernetes keeps the stdout of each container in a Pod. The term Pod is implied when using the command so you simply need to provide the name of the Pod, you don’t have to tell Kubernetes it’s a Pod:

```
$ kubectl logs stack-0
```

You can **follow** logs:

```
$ kubectl logs stack-0 -f
```

You can see the logs from the last 10 minutes, or the last 10 lines:

```
$ kubectl logs stack-0 --since=10m
$ kubectl logs stack-0 --tail=10
```

You can see the logs from the previous container instance, if it exists (in the case of a restarted container):

```
$ kubectl logs stack-0 -p
```

### Shell into a Pod container

You can get an interactive container shell in a Pod assuming you know what shell the container is sing (typically **bash**). To get into a running Fragalysis Stack you can use this command:

```
$ kubectl exec -it pod/stack-0 bash
```

Once you’re done you can **ctrl-d** to get out of the container.

### Watching object state changes

You can watch rolling update status of the stack until completion using the `--watch` argument

### Restarting a Pod (scale down and up)

You can **bounce** (restart) a container. This won’t normally re-deploy a new image - for that you really need to be using the AWX console. But the following might be useful if you simply want to restart a Pod, which is achieved by scaling down and then back up. Here we scale down the Stack’s **StatefulSet** before scaling it bak up again.

Remember that restarting a Pod will cause it to loose any data that is not actively persisted.
Fragalysis Stack (Kubernetes)

$ kubectl scale --replicas=0 statefulset/stack

$ kubectl scale --replicas=1 statefulset/stack

3.4.2 A Developer Cheat-Sheet for Lens

Some useful lens operations that might be useful during development.

Viewing namespace Pods  Selecting Workloads -> Pods will normally show you all the Pods in all namespaces.

Use the Namespace Filter to select one (or more) namespaces you want to view. For example the Staging Stack…
Display Pod container logs  With a Pod visible in the Workloads screen you can click it to see a description panel for the Pod slide-in from the right...
In the banner of the panel that’s presented you’ll see four icons representing the **Shell, Logs, Edit and Delete** actions...
Click the logs icon to see the Pod’s logs...

Shell into a Pod container  With a Pod visible in the Workloads screen you can click it to see a description panel for the Pod slide-in from the right...
In the banner of the panel that’s presented you’ll see four icons representing the Shell, Logs, Edit and Delete actions...
Click the shell icon to be taken to an interactive shell in your chosen Pod...

To exit the shell hit the shell terminal window’s close icon.

**Restarting a Pod (scale down and up)** In order to scale-up (or scale-down) a set of Pods you will need to edit their Deployment, or StatefulSet, whichever is appropriate. The Fragalysis Stack Pod, for example, is managed by a StatefulSet. With Workloads -> StatefulSets selected (and the appropriate Stack’s namespace chosen in the namespace filter) click Stack to see its description.
Click the pencil (Edit) Icon in the description banner to be presented with the YAML representation of the stack’s StatefulSet.
Scroll down if required to locate the `replicas:` field. To scale-down the Pods set this to 0 and then click **Save & Close**. To scale the Pods up, set this field to 1 or more.

### 3.5 Deploying your Stack (“Development for Dummies”)

This document goes through the deployment of a full developer stack. It will show how to:

- Set up the first instance of the developer stack
- Clone the database and media from the production stack into your developer stack
- Set Travis up to automatically push to your stack when changes are made to the master branch on your forks of the fragalysis repositories

#### 3.5.1 Getting started

**Forking from GitHub**

The fist step is to ensure that you have forked the master branch of each component of the fragalysis stack:

1. xchem/fragalysis-backend
2. xchem/fragalysis-frontend
3. xchem/fragalysis-loader
4. xchem/fragalysis-stack

Old forks should have the current version of master put in - changes to the stack include changes to how CI/CD works, so it is important that all forks are brought up-to-date with the master branch of the XChem repositories.
Setting up with AWX

The stack image for your development environment can be accessed at https://awx-xchem.informatics.matters.org/#/login.

Once you are logged in, you can find the templates for the jobs you can run under Templates on the left-hand side of the page. For standard developers, there are two kinds of jobs:

- **Common […]** - these jobs are for duplicating the database and media components into your own stack
- **User […]** - these jobs are for deploying or destroying the stack

If Rachael hasn’t already run the first stack job for you, here’s how to do it:

1. Run the User (<name>) Developer Fragalysis Stack job
   - NB: **DO NOT** navigate to your stack URL until you have completed steps 2 and 3 below
   - NB: the URL for your stack is spat out in the output of this job
   - NB: you must be logged in as your own user to create your own stack. **DO NOT** run any other user’s job under any circumstances

2. Run the Common Database Replicator (One-Time) job
   - NB: you must be logged in as your own user for this to populate your stack

3. Run the Common Media Replicator (One-Time) job
   - NB: you must be logged in as your own user for this to populate your stack

Each job can be launched by clicking on the rocket icon next to the template, this will open the job template, and allow you to launch it. For these first steps, it is fine to run the jobs without changing any parameters.

When the jobs are finished, navigate to the URL spat out by the job in step 1 above. You should now have a copy of the stack that is created from the xchem/master branches.

3.5.2 Setting up Travis for CI/CD

In this deployment, we use Travis to do CI/CD, rather than Travis for CI, and Jenkins for CD. It’s much easier to customise how your stack is built by using Travis.

**Prerequisites**

- You have a travis account
- You have added your forks to your Travis account
- You have a Travis api-token
- You have a dockerhub account

**Using the right Travis version**

The CI/CD implementation will only work running from the newest version of Travis, which is now found at https://travis-ci.com/. However, to set this up correctly, you will need to start from https://travis-ci.org/.

1. Log in to https://travis-ci.org/ with your github account

---

1 Rachael (rachael.skyner@diamond.ac.uk) will give you your username and password to log in
2. On the left-hand side of the first page that loads, under your user account, there will be a request beta access button - click it.

3. This will take you to https://travis-ci.com/

4. If you are using an organisation account (e.g. xchem) then you will need to request access and grant it from github

Once you have granted access to Travis, your repositories should appear in the dashboard of Travis

**Getting a Travis API token**

The following should give you an API access token:

```
$ gem install travis && travis login --com && travis token --com
```

Keep it safe - we need it to allow us to trigger builds for automated deployment.

**Intro to how the CI/CD works**

This is a cubersome but worthwhile step. First of all, you should decide which codebases you want to work on, and therefore incorporated changes from into your own stack.

Once you have decided, you can set up your Travis jobs to automatically trigger builds when you push to certain branches, and automatically deploy your stack or component images to your own dockerhub account. The AWX job that we set up earlier looks for the dockerhub stack image that you build and push with Travis, by taking a single variable, which is the endpoint for your stack image. e.g. rachael/fragalysis-stack. The stack job template assumes it is always looking for the stack image tagged as :latest at that endpoint.

The quickest way to see what the different build variables are, and what they do, is to look at the .travis.yml file in each repository. The comments at the top of those files describe the variables in detail.

Here, I’ll list all the variables that can be added for each deployment. The backend, frontend and loader configurations are optional, depending on what code-base you want to work on. However, you should configure your stack variables if you want to automatically push to your live deployment when you push changes to a branch (I’d suggest setting this up just for master)

**Adding variables to Travis**

1. Log in to Travis
2. Navigate to the Travis job on the left-hand side (it will appear there after you add them)
3. Click on the burger menu
4. Click on the Settings option
5. Add the relevant options under Environment variables - make sure to not show any sensitive info in the build logs

**Travis environment variable descriptions**

**Backend variables (Optional)**

Variables related to images (Dockerhub):
Fragalysis Stack (Kubernetes)

- **PUBLISH_IMAGES** - set this to yes to push any built image to docker
- **DOCKER_USERNAME** - Dockerhub username to allow you to push
- **DOCKER_PASSWORD** - Dockerhub password to allow you to push
- **BE_NAMESPACE** - the Dockerhub namespace you want to push to (e.g. reskyner if you’re pushing to reskyner/fragalysis-backend)

Variables related to GitHub fragalysis-stack repo:
- **STACK_NAMESPACE** - GitHub user for stack
- **STACK_BRANCH** - GitHub user branch for stack

Variables related to auto-triggerring stack build:
- **TRAVIS_ACCESS_TOKEN** - your Travis access token
- **TRIGGER_DOWNSTREAM** - set to yes to trigger a stack build when back-end build is successful

Optional (have defaults):
- **BE_IMAGE_TAG** (default = latest) (dockerhub if not latest)
- **LOADER_NAMESPACE** - xchem (unless working on loader)
- **LOADER_BRANCH** - master (unless working on loader)

**Frontend variables (Optional)**

Variables related to automated build (Travis):
- **TRIGGER_DOWNSTREAM** - yes to trigger build of stack & loader
- **TRAVIS_ACCESS_TOKEN** - needed for the trigger

Variables related to images (Dockerhub):
- **BE_NAMESPACE** - docker namespace (default xchem)

Variables related to frontend GitHub repo:
- **FE_NAMESPACE** – front-end user/account
- **FE_BRANCH** - branch

Variables related to stack GitHub repo:
- **STACK_NAMESPACE** – stack user/account
- **STACK_BRANCH** - GitHub user/account branch

**Loader variables (Optional)**

Variables related to loader image (Dockerhub):
- **PUBLISH_IMAGES** - yes to push to docker
- **DOCKER_USERNAME** - dockerhub username
- **DOCKER_PASSWORD** - dockerhub password
- **LOADER_NAMESPACE** - the Dockerhub namespace you want to push to (e.g. reskyner if you’re pushing to reskyner/loader)
Variables to decide which backend image to use when building the loader image (optional - will default to xchem/master):

- **BE_NAMESPACE** - the Dockerhub namespace you want to use (e.g. reskyner if you’re using reskyner/loader)
- **BE_IMAGE_TAG** - version of image to use (optional, will default to :latest)

### Stack variables (Mandatory for automated builds)

Variables related to stack image - the one your stack will use (Dockerhub):

- **PUBLISH_IMAGES** - yes to push to docker
- **DOCKER_USERNAME** - dockerhub username to allow push
- **DOCKER_PASSWORD** - dockerhub password to allow push
- **STACK_NAMESPACE** - the Dockerhub namespace you want to push to (e.g. reskyner if you’re pushing to reskyner/fragalysis-stack)

Variables setting which back-end image to use (optional - will default to xchem/master):

- **BE_NAMESPACE** - the Dockerhub namespace you want to use (e.g. reskyner if you’re using reskyner/fragalysis-stack)
- **BE_IMAGE_TAG** - docker image tag (optional, will default to :latest)

Variables to control automatic pushing to your AWX stack:

- **AWX_HOST** - AWX url (for devs: https://awx-xchem.informatics.maters.org/)
- **AWX_USER** - AWX username provided by Rachael
- **AWX_USER_PASSWORD** - AWX password provided by Rachael
- **TRIGGER_AWX** – yes to push to AWX
- **AWX_DEV_JOB_NAME** - name of the developer AWX job to trigger stack auto build:
  - NB: As the Job Name contains spaces it needs to be placed between double quotes, e.g. "User (Rachael) Developer Fragalysis Stack (Version Change)". This should be done with any variable value that contains spaces.
  - NB: Change the name to your name!

### Recommended set-up for front-end developers

1. Fork the xchem/fragalysis-backend repo from GitHub
2. Fork the xchem/fragalysis-stack repo from GitHub
3. Add your forks to Travis
4. Setup the following environment variables for the front-end Travis jobs:

   - Variables related to automated build (Travis):

     ```
     TRIGGER_DOWNSTREAM = yes
     TRAVIS_ACCESS_TOKEN = <your access token here>
     ```
Fragalysis Stack (Kubernetes)

5. Setup the following environment variables for the stack Travis jobs:

- Variables related to stack GitHub repo:

  FE_NAMESPACE = <your GitHub account name here>
  FE_BRANCH = master

- Variables related to stack GitHub repo:

  STACK_NAMESPACE = <your GitHub account name here>
  STACK_BRANCH = master

5. Setup the following environment variables for the stack Travis jobs:

- Variables related to stack image - the one your stack will use (Dockerhub):

  PUBLISH_IMAGES = yes
  DOCKER_USERNAME = <Your dockerhub username here>
  DOCKER_PASSWORD = <Your dockerhub password here>
  STACK_NAMESPACE = <your GitHub account name here>

- Variables setting which back-end image to use (optional as it will default to xchem/master):

  BE_NAMESPACE = <Your dockerhub username here>

- Variables to control automatic pushing to your AWX stack:

  AWX_HOST = https://awx-xchem.informaticsmatters.org/
  AWX_USER = <Your AWX username here>
  AWX_USER_PASSWORD = <Your AWX password here>
  TRIGGER_AWX = yes
  AWX_DEV_JOB_NAME = "User (<Your name here>) Developer Fragalysis Stack → (Version Change)"

6. Alter the User (<Your name here>) Developer Fragalysis Stack (Version Change) job in AWX:

- Click on the templates on the left hand side
- Click on the job name

- Under EXTRA VARIABLES change stack_image: xchem/fragalysis-stack to point to your image (e.g. reskyner/fragalysis-stack)

Now that you’ve done this, every time you push a change from a branch into master in your frontend fork:

- The tests for the front-end will run in Travis
- If the tests run, the back-end and stack jobs will be triggered
- When the stack-job completes, an image of that stack will be pushed to your Dockerhub repo
- After the image is pushed, a job is triggered in AWX
- That job takes the image that has just been pushed and re-builds the stack with it

Alternative deployment strategy - Developing locally

On the xchem/fragalysis-backend and xchem/fragalysis-frontend repositories, there are instructions on how to set up a local development environment using Docker in the README.md files in the root of the respective repository.
Part of the process of using this local environment includes building the backend and/or frontend images, and using them locally, and then using those images to build a stack image. Because the stack image is all that is needed to push a new version into a live stack, the following process can be used to use those locally built images to push to your stack on AWX:

1. Log in to Docker:

   ```bash
   $ docker login --username=<your hub username> --password=<your password>
   ```

2. Build your image by executing the docker build command. `DOCKER_ACC` is the name of your account, `DOCKER_REPO` is your image name and `IMG_TAG` is your tag:

   ```bash
   $ docker build -t $DOCKER_ACC/$DOCKER_REPO:$IMG_TAG .
   ```

   e.g. `docker build -t reskyner/fragalysis-stack:latest .` is the command for Rachael to build her stack image, ready to push to Dockerhub.

3. Now, you can push this image to your hub by executing the docker push command:

   ```bash
   $ sudo docker push $DOCKER_ACC/$DOCKER_REPO:$IMG_TAG
   ```

   This will push the image up to Dockerhub. The only image you need to push is the stack image, as this is the image used by AWX to build your stack.

4. Go to AWX, and navigate to your User (<name>) Developer Fragalysis Stack (Version Change) job template.

5. In the EXTRA VARIABLES section, change `stack_image: xchem/fragalysis-stack` to point to your image (e.g. `reskyner/fragalysis-stack`).

6. Save and launch the job.

7. Navigate to the stack to see the changes from your local dev environment live in the wild!

### 3.6 Sensitive Material (Ansible Vault)

Deployment of the stack relies on some sensitive variables that may include things like hostnames, users and passwords.

To simplify automated deployment, this information is stored in this repository in an encrypted form using Ansible Vault. The AWX server is in possession of the encryption key (itself encrypted) so the Job Template that relies on these variables can safely decrypt them when the corresponding playbook is executed.

Developers don’t need the decryption keys so they cannot see them.

#### 3.6.1 Deploying a user stack (Development Cluster)

The AWX server should have been set up with Job Templates to deploy and un-deploy stack instances. There are also common templates to synchronise the stack’s built-in PostgreSQL database and Django media files (if required) to those used by the Production server.

Developers will have been given access to the server and a set of templates should be visible to them when the login.

The Fragalysis Stack application is deployed to Kubernetes from container images present on a public registry (i.e. Docker Hub). To deploy your stack, you will first need to have built your container image and pushed it to somewhere like Docker Hub.
Fragalysis Stack (Kubernetes)

To deploy your stack you should click on Fragalysis Stack Job Template that will have been created for you, i.e. USER (<YOU>) Developer Fragalysis Stack. This template will give you an opportunity to fine-tune the deployment for your specific image. That essentially means ensuring the following variables are set for your needs:

1. `stack_image` - the container registry project and image name for your stack, i.e. `xchem/fragalsyis-stack` for official images.
2. `stack_image_tag` - the tag you've assigned to your image (i.e. `latest`)

With variables set you just SAVE them (if you’ve changed them) and then click LAUNCH to run the deployment playbook.

**Ingress and namespace**

Your stack is deployed to a Kubernetes namespace that’s unique to you. The playbook will display this value at the end of the deployment along with the URI that should direct traffic to your stack instance.

**Loading target data**

You can load target data into your stack using another AWX Job Template. You should find a User (<YOU>) Developer Data Loader Job Template, which relies on a loader container image that can synchronise data from the internal NFS server that hosts the target data.

When you run the loader job you simply need to ensure that the following Job Template variables are appropriately set:

```plaintext
loader_data_origin
stack_name (normally just left at 'default')
```

**Replicating target data (from Production)**

Instead of loading your own target data you can simply replicate data from the latest Production stack (which is made available in the early hours of each morning) into your stack’s instance.

Two Job Templates exist to allow this, and they can be run once your stack has initialised:

1. **Common Database Replicator (One-Time)** to replicate the Production PostgreSQL `frag` database
2. **Common Media Replicator (One-Time)** to replicate the Production stack’s media files

**Redeploying a Stack (a version or code change)**

If you’ve already deployed your stack and have now produced a new Docker image (even a new `latest` image) you can quickly redeploy the image by running the User (<YOU>) Developer Fragalysis Stack (Version Change) Job Template.

This relies on the stack having been deployed (i.e. with a database and persistent volumes) and simply causes the Pod to restart while also re-pulling the image from Docker Hub.

You can just run the original User (<YOU>) Developer Fragalysis Stack Job template but that takes a little longer to run.

---

2 There are a some other variables but these are the key ones.
Fragalysis Stack and cluster administrative documentation.

4.1 Cloud Provider (STFC)

Notes for the STFC OpenStack cloud provider compute instances.

4.1.1 Instance base image

The cluster nodes are based on a refined image called `ubuntu-bionic-xchem-noupdate4-noquattor`. This special Ubuntu image (prepared by the STFC team) is also pre-configured to disable automated system updates driven by the provider’s use of the Quattor toolkit (see the provider’s NoQuattor notes).

If the automated updates are not disabled there’s a risk that the node’s Docker service gets regularly stopped and updated.

- This image *MUST* be used for all Kubernetes compute instances that you create using Rancher Node Templates.
- This image should also be used for additional compute resources, like an NFS server providing NFS volumes to the clusters.
- This image should also be used for the Rancher RKE nodes.

If the image cannot be used for an instance follow the provider’s NoQuattor notes on how to disable automated updates.

4.2 The Rancher Server

The DEVELOPMENT and PRODUCTION clusters have been created with and are managed by Rancher, deployed to the STFC/OpenStack cluster on a dedicated kubernetes cluster configured using RKE.

---

1. ID: 12cd22a2-637f-490e-b703-681dace0f761
https://rancher-xchem.informaticsproblems.org

... where you will need suitable credentials in order to log-in.

A simplified depiction of the clusters can be seen in this diagram. Each cluster consists of key `etcd` and `control plane` nodes and various worker (`app` and `graph`) nodes. The instances are created and managed by the Rancher server.

A full description of the Rancher installation and its configuration can be found in the following external (GoogleDoc) document:

- OpenStack K8S clusters with Rancher (AWS)

**Warning:** The cluster instances are created automatically by the Rancher server. **DO NOT edit or delete any compute instance that may be a Rancher-managed Kubernetes instance via the STFC/OpenStack console.**

To help you identify them the instances use a naming convention. In our case instance names that belong the the cluster hosting Rancher begin `rke-`. Instance names that belong to the DEVELOPMENT or PRODUCTION cluster begin `xch-`.

### 4.2.1 Cluster etcd backups

All Kubernetes objects are stored on `etcd`. Periodically backing up the etcd cluster data is important to recover Kubernetes clusters under disaster scenarios, such as losing all master nodes. The snapshot file contains all the Kubernetes states and critical information.
The backup of etcd for the RKE and Application clusters is automated using features built-in to RKE and Rancher, with snapshots written to an Informatics Matters AWS S3 bucket (detailed in the following sections).

The credentials used to create AWS S3 backups are those of the user fragalysis-loader on the Informatics Matters AWS account. The secret access key is stored in the Informatics Matters KeePassXC application (under AWS -> AWS S3 (Fragalysis) User).

Application clusters

A snapshot of each cluster’s etcd content is configured to occur regularly with the local copy also copied to an Informatics Matters AWS S3 target in the eu-central-1 region.

Backups of the application clusters is performed by Rancher.

Typical automated AWS S3 backup settings are illustrated in the following Rancher cluster configuration screenshot:

Cluster bucket and path details are as follows:

- The PRODUCTION cluster’s etcd is backed up to im-rancher/xchem-production. This occurs every 6 hours and 28 copies are kept (a 7-day approximate history)
- The DEVELOPMENT cluster’s etcd is backed up to im-rancher/xchem. This occurs every 6 hours and 42 copies are kept (a 10-day approximate history)

Individual backup file size is approximately 7MB for DEVELOPMENT and 21MB for PRODUCTION.

Rancher (RKE) cluster

The etcd material for the RKE-formed Kubernetes cluster that hosts the Rancher server is also backed up. This backup is performed by RKE and configured using the RKE cluster.yml, which can be found in the rancher directory of the Fragalysis deployment repository.

Cluster bucket and path details are as follows:

- The RKE cluster’s etcd is backed up to im-rancher/rancher-xchem. This occurs every 6 hours and 21 copies are kept (a 5-day approximate history)
Individual backup file size is approximately 8MB, with 3 files created per backup.

For details of backup configuration refer to the Rancher RKE backup documentation.

### 4.2.2 Cluster etcd recovery

**Application clusters**

Restoring an application cluster (PRODUCTION or DEVELOPMENT) from a backup is relatively straightforward. It can be done from within the Rancher console for the chosen cluster.

Follow the Rancher Restoring a Cluster from Backup documentation, remembering that we’re using a post v2.4.0 Rancher installation.

**Rancher (RKE) cluster**

Restoring the RKE-based cluster is a little more complicated, compared to restoring an application cluster, and you should follow the Rancher Restoring Backups—Kubernetes installs documentation, following the appropriate S3-based instructions.

Some example RKE scenarios are illustrated.

### 4.3 The NFS Servers

There is an NFS server to provide dynamic provisioning and shared (replication) volumes for both the DEVELOPMENT and PRODUCTION clusters. An separate NFS sever also exists to provide a a volume for Fragalysis (Django) Media files, used by the Media Loader AWX jobs.

#### 4.3.1 The Dynamic Volume Server

This is a `c2.large` (8 core, 8Gi) instance type with Floating IP 130.246.213.182 that has two attached volumes that act as roots for the respective cluster’s NFS provisioner. All application-based NFS volumes (the Kubernetes Persistent Volume Claims) are served from one of the devices (one device dedicated to the development cluster and one for production cluster).

Two more volumes are used to provide static exports for the production media and database replicas.

---

2 We use RKE that’s superior to v0.2.0
An AWX job launches a CronJob that replicates the production media (using rsync) and database (using pg_dump) to the static volumes. These replicas form the basis of the re-import into the Staging stack and any User stack that requires them.

### 4.3.2 The Static Media Volume Server

This is a `m1.large` (2 core, 8Gi) instance type with Floating IP `130.246.213.186` that has one attached volume that acts as static root for new Fragalysis (Django) Media data.
A user deposits new files onto the NFS export (the External Process) for use by Loader Jobs in production, staging and user stacks.

4.3.3 Creating the servers

The servers were instantiated manually using the STFC/OpenStack console and then configured using the site-nfs.yaml and site-nfs-media.yaml plays in the root of this project.

This repository should be cloned to the STFC project bastion machine.

The Dynamic Volume Server

1. Using the STFC/OpenStack Console...
   1. Create an NFS server instance
   2. Create four suitably sized volumes and attach them to the NFS server. There is a volume for dynamic allocation of development and production cluster volumes (say 4TB each), a volume for the media replica (say 100Gi) and one for the database replica (say 100Gi)
   3. Attach the designated floating IP (130.246.213.182)
2. Review the site-nfs.yaml file and ensure the various device: values are correct (i.e. the device mappings are what you expect)
3. From the bastion machine run the playbook (i.e. run `ansible-playbook site-nfs.yaml`) 
With this done your NFS server should be available from either cluster.

**The Static Volume Server**

1. Using the STFC/OpenStack Console…
   1. Create an NFS server instance
   2. Create a suitably sized volume (say 1TB) and attach it to the NFS server
   3. Attach the designated floating IP (130.246.213.186)
2. Review the `site-nfs-media.yaml` file and ensure the `device:` value is correct (i.e. the device mapping is what you expect)
3. From the bastion machine run the playbook (i.e. run `ansible-playbook site-nfs-media.yaml`)
With this done your NFS server should be available from either cluster.

**4.3.4 Volume Snapshots**

A *Scheduled* AWX Job Template runs regularly\(^1\) to create volume snapshots of the key NFS volumes. These snapshots are not guaranteed to be re-usable, especially of the data on them was being modified during the snapshot (which is why snapshots are taken at an obscure time). The snapshots are made in the event of disaster recovery. Data should always be backed up separately.

The AWX Job that’s responsible for creating volume snapshots (which runs on the DEVELOPER cluster) is:

• **Maintenance Volume Snapshot**

It is driven by the `snapshot_volumes` Ansible variable, a list of volume names in the xchem-follow-up Project. The default value of the variable is:

```
snapshot_volumes:
  - kubernetes-nfs-dev-dynamic
  - kubernetes-nfs-prod-dynamic
  - kubernetes-nfs-frag-stack-media
```

Once a snapshot is made the playbook removes others older than 6 days so, in the default setup, only one snapshot is maintained.

**4.4 AWX (Adding Developers)**

Before a developer can deploy their Fragalysis Stack they will need a user account on the Development AWX sever and Jobs duplicated and assigned to them.

**4.4.1 Adding Users**

Using an administrative account on the Development AWX server navigate to Users in the side-panel and click the + (Add) icon in the upper-right-hand corner.

• Provide suitable values for all the fields.

\(^1\) At 04:04 on Sundays at the time of writing.
• Most USER TYPE values a likely to be Normal User
• Click Save

Now you need to assign permissions and duplicate Job templates. The user will only see templates that you assign to them, as described below.

### 4.4.2 Permissions and Duplicating Templates

Now that you’ve created a user navigate to the Templates panel using the side-bar. Here we need to provide access to common Jobs and duplicate Jobs we expect the developer to edit for the User you’ve just added.

#### Common Templates

Here we give users access to each template that begins with the word Common. For each Common template:

- Click it to navigate to it
- Click the PERMISSIONS button at the top of the panel
- Click the + (Add) icon to add a user
- Click the empty checkbox next to the appropriate user from the dialogue box that appears
- In the Please assign roles to the selected users/teams section select the Execute role - the user needs to execute common Jobs but is not expected to edit them.
- Click Save

#### Developer-specific Templates

Here we need to Duplicate each template that begins Developer. For each Developer template:

- Click the Copy Job Template at the end of the template row
- The template is duplicated initially by by adding the text *<HH:MM:SS> to the end of the new template name, Click this new template name to edit it
- Change the duplicated template’s NAME by removing the *<HH:MM:SS from the end and add User (<user>) at the beginning.
- Click the PERMISSIONS button at the top of the panel
- Click the + (Add) icon to add the user to the template
- In the Please assign roles to the selected users/teams section select the Admin role - the user needs to be able to edit and execute the developer Jobs.
- Click Save

### 4.4.3 Provide the Development Kubernetes Config file

Developers will need a kubernetes configuration file in order to interact with the cluster. Ideally each developer will need a config file that has a kubernetes user created specifically for them, or you can share a common configuration.

- Create a user on Rancher (and export the kubeconfig)
- Or circulate the generic config-xchem-development-developer configuration
### 4.4.4 Here Be Dragons!

**DO NOT** ...circulate a cluster *-admin* configuration or *-diamond* configuration outside of the Diamond organisation.

**DO NOT** ...run job templates in the development cluster designed for other users. The developer and common templates are `conscious` of the logged-in user and, as the stack namespaces are automatically generated based on the `username` running another user’s job as will have unexpected and potentially damaging side-effects.

Only run developer and common jobs when you’re logged-in as a developer account, never as an administrator.

This excludes the **Graph** deployment - these jobs can be run by anyone who has permissions - it’s namespaces is fixed.

### 4.5 Related Repositories

A list of repositories related to Fragalysis and its deployment and management on Kubernetes.

#### 4.5.1 Informatics Matters Repositories

Repositories hosted on the Informatics Matters GitHub account.

- ansible-infrastructure
- ansible-role-awx-composer
- ansible-role-cert-manager
- ansible-role-infrastructure-data
- ansible-role-infrastructure-user
- bandr
- dls-fragalysis-stack-kubernetes
- docker-neo4j
- docker-neo4j-ansible
- docker-volume-replicator
- trigger-awx
- trigger-travis

The following repositories are employed to generate neo4j-compliant fragmentation data: -

- fragmentor

Cluster formation (using STFC OpenStack) resources are managed by the following repositories: -

- ansible-bastion
- ansible-galaxy-cloud

---

1 This repository creates a Nextflow/Slurm/Munge/Pulsar cluster that can be used but it was not employed to create the existing Galaxy/Condor cluster, which created independently of our tools.
4.5.2 XChem Repositories

Repositories hosted on the XChem GitHub account.

- fragalysis
- fragalysis-backend
- fragalysis-frontend
- fragalysis-loader
- fragalysis-stack

4.5.3 3rd Party Repositories

Repositories hosted on GitHub that we rely on but have little or no control over:

- kubernetes/ingress-nginx