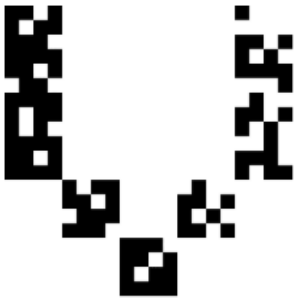


Athena User Manual

Contents

Introduction

What is Athena?

Athena is a web application which integrates into an existing Kubernetes cluster. It allows users to create Docker container-based lab environments, which can be used as a learning aid by educators, or as a testing area for new software deployments.

What is Kubernetes?

Kubernetes is an open-source platform which orchestrates the processes involved in the management of a cluster of container-based systems. Kubernetes manages the deployment and destruction of containers based on predefined rules set by the cluster administrator and ensures that container-based workloads can communicate with each other and the wider network.

While Kubernetes can work with several container technologies, the one most well-documented by the Kubernetes authors is the Docker container; and this is the container technology which has been used to create the Athena system.

What is Docker?

Docker is a container-based virtualisation solution. Like the virtual machine —another virtualisation technology— containers allow a system's user to host entire 'virtual' systems within another system, with each system isolated from each other.

However, containers differ from hardware virtualisation solutions like the virtual machine in the extent of their virtualisation. Unlike the virtual machine, which emulates the entire hardware environment of a computer system, containers simply use their host's system's kernel to access the physical hardware of the host (Microsoft, 2019).

Kubernetes Concepts

Kubernetes contains a multitude of objects and configuration options. While we have included brief explanations of the Kubernetes objects we expect tutors to be interacting with, we recommend that one consults the Kubernetes documentation for additional information or clarification. This documentation is available at kubernetes.io/docs.

Namespaces

The Kubernetes Namespace object is a logical concept used to describe an individual Kubernetes operating environment within the Kubernetes cluster. The Kubernetes documentation suggests that each team or project should be assigned its own namespace — this allows groups with aligned interests to share resources, while still allowing other entities to access their own (potentially isolated) services.

Service objects, Ingress objects, Deployment objects and NetworkPolicy objects are all 'namespaced' - i.e. they are always created within an individual namespace (The Kubernetes Authors, 2020). In the context of the Athena application, this is relevant because the name of a namespaced object must be unique within the namespace, which could affect how one chooses to name resources when creating new templates.

Selectors like the Pod Selector for NetworkPolicies and the MatchLabels selector for a Service are Namespace-scoped - they will only select objects within the selecting object's Namespace by default, and one must identify another Namespace using an explicit selector such as the NetworkPolicy object's NamespaceSelector in order to be able to select Pods outside a Namespace (The Kubernetes Authors, 2020).

The Athena application assigns an individual Namespace to each user of the application.

Deployments and Pods

Kubernetes deploys Docker containers in units called Pods. Each Pod is a Kubernetes object which contains one or more Docker containers. All containers within a Pod appear to the outside world to be a single entity, are assigned a shared Pod IP address, and can communicate with each other using the localhost loopback address.

Deployment files describe how a Kubernetes Pod should be deployed and how it should be maintained, including (but not limited to) the container image(s) used, the access the Pod (or a container within the Pod) is granted to the host, and the storage volumes it should be assigned.

The Athena application requires a Lab Template to contain at least one Deployment file.

Services

While Pods are assigned IPs, this allocation is not predictable, and as Pods are rescheduled to another Node, they can have new IP addresses allocated. This can make establishing connectivity with Pods challenging, which is why Services are used. Services are Kubernetes which act as a proxy for traffic to Pods. Services are assigned to Pods based on the label attached to a Pod - for example, a Service with a label selector of 'lab: example' will select any Pod with the 'lab: example' label, and direct traffic it receives to it.

Services are assigned their own cluster IP address (in a separate IP address range to Pod IPs), and can direct traffic they receive on one port to a different port on the Pod.

Services are assigned a fully-qualified name following the format:

<service-name>.<namespace>.<svc>.<cluster-domain>.<tld>

Within a Namespace, Services can be reached by using the domain-name <service-name>.<namespace>. On creation, Pods have information about currently running Services embedded in environment variables within the container, and these variables can be used by Pods to discover other services in the Namespace. To allow authors of Lab Templates the greatest flexibility, the Athena system deploys Services first, to ensure that all Pods have up to date information about Service availability.
There are three types of Service:

- ClusterIP Services, which are as described above
- NodePort Services, which are as described above, but are assigned a port on the cluster's Nodes, and can be reached at <Node-IP>:<assigned-port>
- LoadBalancer Services, which are assigned a dedicated IP address on the network that the cluster is connected to. This may be a public or private IP address - the suggested configuration for the Athena application assumes that it is being deployed to a private Local Area Network.

Typically, LoadBalancer Services are provisioned by a cloud provider at a customer's request. Kubernetes does not have a built-in implementation for LoadBalancers which can be used using 'on premises' or 'bare-metal' hardware. To resolve this, the provided configuration uses the MetalLB plugin, which adds LoadBalancer support for 'bare-metal' clusters.

Ingresses

Kubernetes Ingresses are load-balancers which are typically used to direct HTTP traffic to the appropriate back-end Service for the traffic. Ingresses in the implementation used for the Athena application terminate HTTPS connections at the Ingress. Behind the scenes, Ingresses rely on an Ingress Controller to route traffic (The Kubernetes Authors, 2020).

The implementation used in the Athena application relies on the 'ingress-nginx' implementation of the Ingress Controller, which is maintained by the Kubernetes Project. ingress-nginx, as the name might suggest, uses a NGINX backend to direct traffic. It should not be confused with NGINX's own Ingress Controller implementation (The Kubernetes Authors, 2020).

As previously noted, there are other Ingress Controller options available, but one would need to reconfigure the Athena application to integrate properly with them should one wish to completely replace ingress-nginx. Should one need the features of another Ingress Controller for other applications, one can deploy multiple ingress controllers, and the Ingress Controller used by the Athena application will not require additional configuration to achieve this.

NetworkPolicy

As the name might suggest, NetworkPolicy objects control the flow of network traffic. The rules contained within each NetworkPolicy define which network devices can communicate with a Pod, and vice-versa.

In order to use NetworkPolicy, Kubernetes requires the operator of a cluster to choose a Container Network Interface (CNI) plugin. The Kubernetes implementation that the Athena application was developed around used the Project Calico CNI plugin, which enables full support of Kubernetes NetworkPolicy while not introducing potentially unwanted complexity from features like overlay networking (Tigera, 2020).

NetworkPolicy objects apply their rules to Pods based on the labels that the Pod currently has — for example, a NetworkPolicy which is applied to Pods with the label ‘lab: example’ might allow traffic from and to Pods with the ‘lab: example’ label.

We suggest that when designing a lab template, one defines a NetworkPolicy to only allow traffic within a lab environment, and to and from the Athena Ingress Controller. This prevents traffic from other lab environments interfering with the user experience. Further configuration is at one's discretion.

Docker and the Athena application

Docker Repositories

There are many publicly available Docker images which can be found on the Docker Hub. If one's organisation intends to only use images from public Docker repositories, one does not necessarily need to do any additional configuration.

However, if one wishes to use a private repository to host Docker images which are used by the Athena application, one should configure the Athena application's ServiceAccount to use this repository when creating Pods. This involves creating a Kubernetes Secret which contains the Docker login credentials for the repository (or repositories) the Athena application should use when attempting to pull images.

The process below is an example of the process one might follow to create the Secret and update the Athena ServiceAccount:

1. On a master node, log in to one or more private repositories with Docker using the **docker login** command.
2. Run the following command, substituting <> fields with the appropriate values for the system being used

.

kubectl create secret generic <secret-name> \
--from-file=.dockerconfigjson=<path/to/.docker/config.json> \
--type=kubernetes.io/dockerconfigjson

The '.docker' folder is typically found in the system user's home directory (~/).

3. Edit the athena-service-account.yaml file, uncommenting the 'imagePullSecrets' section and entering the created Secret's name in the name field.

Custom Docker Images

If there are no existing public Docker images which fit one's needs, it is possible to create custom Docker images by writing a Dockerfile which contains the instructions that Docker will follow to build an image.

While there are no specific limitations on what Docker images can be used in the Athena system, one should consider the following when writing Dockerfiles:

1. Docker containers are expected to contain a single process, and do not have a built-in init system that is responsible for resource management when a process ends. If running multiple processes, one may wish to use a program such as supervisord, which can help manage the starting and stopping of processes within a container.

2. The more applications one installs, the larger the Docker image gets. One of the key benefits of Docker images is that they are typically a lot smaller than virtual machine images, and this benefit is reduced the more applications one installs. To minimise the size of each image, consider first creating an image which includes the software which you expect to be needed in every lab. Once this base image has been defined, one can create variants of this image for each lab, with each lab only having the software needed for the lab installed.

3. If one chooses to follow the above approach to building images, a secondary benefit is achieved at the same time: Docker images which share the same base image will only take up space it takes to describe the differences between them. This is likely to take up less space than creating custom images 'from scratch' every time.

4. The more applications one installs, the more system resources the Docker container needs, and while containers can be allocated additional resources by changing the relevant Deployment file, system resources are not infinite. More minimal Docker containers will be more responsive to more users.

The Athena Application

Preparation and Installation

To deploy the Athena application, you will need to have an operational Kubernetes cluster which supports the following features:

- Kubernetes NetworkPolicy
- LoadBalancer Services

This manual is supplied with a brief installation guide which will take you through the process of provisioning a bare-metal Kubernetes cluster that meets these requirements.

Configuring the Cluster

Installing Prerequisites

The provided installation script will install Docker and Kubernetes (and their dependencies) on an Ubuntu server, according to the processes outlined at the links below:

- Install Docker Engine on Ubuntu
- Container runtimes
- Installing kubeadm

Initialising the cluster

Once Kubernetes and Docker are installed, one can initialise the Kubernetes cluster using the kubeadm init command. This command will either accept arguments from the command line or will read the equivalent values from a config file supplied by the user using the '--config <file>' option.

We strongly recommend that the 'control-plane-node' option is configured with a valid DNS name which resolves to at least one master Node IP address. This value must be set at cluster initialisation and is used by worker Nodes to locate a master Node (should it be set). This allows for greater flexibility in changing the makeup of the master Node pool - should one add another master Node, traffic from worker Nodes can be split between the master Nodes using DNS-based load-balancing, and the process of moving master Nodes to different IP addresses is greatly simplified.

The provided command contains the default values for the 'pod-network-cidr' and 'service-cidr' options - please review these, as they will be the IP address ranges used for Pod and ClusterIP Services respectively.

**kubeadm init --service-cidr 10.96.0.0/12 **
**--pod-network-cidr 192.168.0.0/16 **

--control-plane-endpoint [your-DNS-name]:6443

Once the cluster has been initialised, the kubeadm init command will output a 'join string' which can be entered on each worker Node to join it to the cluster.

Installing a networking plugin

At this stage, one should install a Container Network Interface (CNI) plugin using the kubectl apply -f <CNI.yaml> command in order to enable full networking functionality.

We have supplied a slightly-modified Calico YAML file which uses IPIP encapsulation for cross-subnet traffic -- we feel that this provides a good balance between performance and flexibility, as Nodes in the same subnet can communicate without the added overhead of IPIP encapsulation, while retaining the original cluster-internal routing information of the packet in situations where it is more likely to be modified. Should one wish to use the standard Calico installation, or make one's own modifications, one can find an unchanged version on the Project Calico website.

Installing MetalLB

Once a CNI has been installed, the next step is to install MetalLB, a plugin which enables LoadBalancer Service functionality for bare-metal Kubernetes clusters. More information about MetalLB (including additional configuration information) can be found at https://metallb.universe.tf.

The following commands are taken directly from the MetalLB installation guide and are reproduced here for the sake of convenience.

kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.9.3/manifests/namespace.yaml

kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.9.3/manifests/metallb.yaml

kubectl create secret generic -n metallb-system memberlist --from-literal=secretkey="$(openssl rand -base64 128)"

After MetalLB has been installed, it must be configured before it becomes operational. We have supplied a basic configuration file which configures MetalLB to advertise LoadBalancer IP addresses at Layer 2 -- this configuration will almost certainly need to be modified to reflect the IP address ranges which MetalLB can assign IP addresses from on your network.

Once these changes have been made, apply the configuration file using the **kubectl apply -f** command.

Installing ingress-nginx

The final step is to install ingress-nginx. Once again, we have supplied a slightly modified version of the default 'bare-metal' configuration. Changes made include:

- Changing the ingress-nginx configuration so that ingress-nginx uses a LoadBalancer service rather than a NodePort service for external connectivity
- Changing the 'Ingress Class' value to be 'nginx-athena'. This matches the 'kubernetes.io/ingress.class: nginx-athena' annotation which must be on all Ingresses used in Lab Templates

Like the other plugins, ingress-nginx can be installed using the **kubectl apply -f <filename>** command
At this point, the cluster is ready for deployment of the Athena application.

Installing the Application

Docker Images

It is intended that the Athena application will be deployed to the same Kubernetes cluster that will be used to host Athena's lab environments. To this end, both Dockerfiles for building application and database images have been supplied, along with the files which are expected to be needed for application deployment.

Dockerfiles used to build the Athena application and database images have been included. These Dockerfiles will build the

If this is the first time installing the Athena application and the application and database Docker images have not yet been built, the following instructions may assist with this:

1. Using a command-line interface, navigate to the directory of the image you wish to build.
2. Run the following command:

docker build -t <tag-name> .

The specifics of the tag name are not important for this process — if you have a Docker repository you wish to upload the image to, add the name of the repository as a prefix to the tag.

Once the images have been built, the image names should be inserted into the relevant Kubernetes Deployment file.

Kubernetes Configuration

To aid in configuring your Kubernetes cluster, the following files have been provided:

1. A ClusterRole definition which defines the scope of the Athena application's access to the Kubernetes API.
2. A ServiceAccount definition for the Athena application's ServiceAccount.
3. A ClusterRoleBinding definition which assigns this role to the Athena application's ServiceAccount.
4. Secret definitions for the required OAuth and Service Account credentials, as well as the application database's credentials.
5. Namespace definitions for the Athena application's namespace and the 'kube-system' namespace.
6. A PersistentVolume and PersistentVolumeClaim, which are used to give the application database Pod a place to store application data. While not required, it is highly recommended that this is used to avoid having to manually repopulate the application database should the Pod be deleted (for whatever reason).
7. Service definitions which target the Athena application and database Pods.
8. Deployment definitions for the Athena application and its SQL Server database.
9. An Ingress definition which will direct traffic intended for the Athena application to the correct Service.
10. NetworkPolicy definitions which will explicitly allow traffic between the Athena application and database, and between the Athena application and the Athena Ingress. These are not required if no other NetworkPolicies are currently applied to the Athena application's Pods, but ensure that the application will have the network access it needs to function should one apply additional NetworkPolicies.

We suggest you edit these files to your liking, then apply them using the **kubectl apply -f <filename>** command in the order listed above. By creating the Microsoft SQL Server Express container, you will be agreeing to the End License User Agreement, available here.

At this point, the system will be running, and there is one more configuration step to perform.

The application will be running in Developer mode so you can update the application database to fit the application's needs. Visit the domain you have allocated to the application, and login to the application with a it.weltec.ac.nz email address. You will be prompted to apply migrations to the database; click the Apply Migrations button and refresh the page once the migrations have been applied. You should now be logged in.

Now that the database has been configured, you should use the **kubectl delete deployment -n athena-app athena-app** to delete the Deployment that is currently running. Remove the following lines from the Deployment file for the application:

```
- name: ASPNETCORE_ENVIRONMENT
  value: "Development"
```

Redeploy the application's deployment file using the **kubectl apply -f** command. The system is now ready to use.

Lab Templates

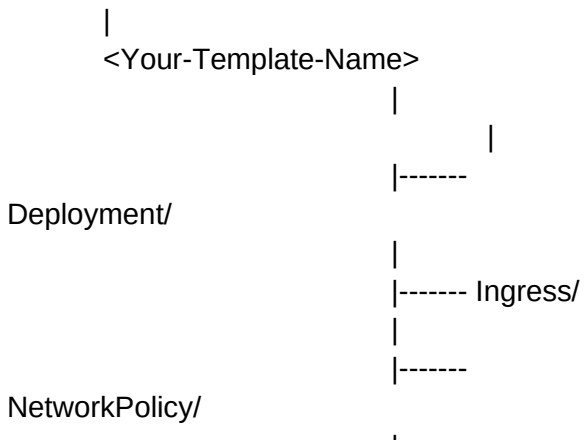
Creating a new Template

We suggest that when drafting the files associated with a Lab Template that one follows the following steps:

1. Prior to creating any Lab Template YAML files, test Docker images thoroughly using a local instance of Docker.
2. Write the first draft of the Template's YAML files, ensuring that the Template contains at least one Deployment
3. Test the YAML files in a development environment like Minikube (if one is available). Otherwise, if API access has been made available, execute the following command for each of the YAML files used for the Template:

   kubectl apply -f <file-name> --dry-run=client

4. The node will read the files and attempt a dry-run deployment of the object specified in the file. If this command is successful, there are no syntax issues with the file.
5. At this point, the files can be tested in the Athena application. Begin by creating a directory structure like the one below:



6. Copy the YAML files to the appropriate directories within this structure. Make sure that there is at least one Deployment file within the Deployment directory.
7. Copy the Template root to the Templates folder used by the Athena application. This could be a directory on one of the cluster hosts, or a network share which the Athena application connects to.
8. Add the Lab Template in the Athena application, and link it to a Group you are a member of.
9. Create a Lab Environment using the Template.

10. If everything functions as expected, this is all that needs to be done. If errors are encountered, we suggest checking the ‘Common Issues’ section to see if the issue is addressed there.

Lab Labels

All objects which are created by the Athena application must have a label indicating which lab they are associated with, following the format 'lab: <value>'. The value here can be any value that the Kubernetes label specification will accept, but one must make sure that this label is present on all objects associated with a given Lab Template. The Athena application will endeavour to identify if this is missing or if there is a mismatch on when one creates a Template in the Athena application, but will not perform this check again.

If updating a Lab Template, make sure that all labels are correct and match the value supplied to the Athena application. Athena relies on this value to terminate the Kubernetes objects associated with a lab environment; if there is a mismatch, not all resources will be deleted.

Following these guidelines should reduce the chances of having issues with label mismatches:

- Ensure that all Deployment labels are consistent with the Pod template they contain, and the selector used to select said pod template before writing any other part of the Deployment file.
- With the Deployments defined, one can move on to Services. It should be clear what labels one can use to select Pods with at this point; at a minimum, ensure that the Service selects Pods based on its 'lab' label value.
- Ingress objects must also be labeled with the 'lab' label, but no further action is required here.
- As with the Ingress object, NetworkPolicy objects should be labeled with the 'lab' label, but no further action is required (but may be desired).

Ingress Specifics

Annotations Required by Default

By default, Ingress objects in Lab Templates **must** have the following annotation:

kubernetes.io/ingress.class: nginx-athena

This annotation will cause the Ingress Controller with a matching ingress-class value to serve traffic for the paths specified in the Ingress.

This configuration gives other cluster users greater flexibility in their choice of Ingress Controller for a particular task or project — rather than being forced to use the ingress-nginx controller, they can select another option, and use its ingress-class value instead.

Athena-specific Annotations

The Ingresses used in Lab Templates support all the features of the ingress-nginx controller, which can create problems when using regular expressions in Ingress path fields

The nginx.ingress.kubernetes.io/rewrite-target annotation directs the Ingress controller to rewrite the requested path of an HTTP request to the value specified. If we take the example given by the ingress-nginx documentation for this feature, this value is '/$2', which will rewrite the requested path passed to the backend to be /[pattern-matching-second-regex-capture-group].

This functions as expected but can cause problems when presenting the final lab interface URL to a user of the Athena application, as the application also uses the path value to construct the URL. athena.local/user-id/juice-shop(/)(.*) does not match the intended path.

For this reason, we have implemented the following annotations:

athena-regex
athena-regex-rewrite

If an Ingress path contains a regular expression which should be removed before presenting the URL to the user, specify the regular expression to remove using the **athena-regex** annotation. The following annotation would direct the Athena application to remove the '(/)(.*)' regular expression from all paths:

athena-regex: (/)(.*)

Should one want this regex value to be replaced with another string of characters when it is presented to users, specify the replacement string using the **athena-regex-rewrite** annotation. The following annotations would direct the Athena application to remove the '(/)(.*)' regular expression from all paths and replace it with the '/' character prior to presenting it:

athena-regex: (/)(.*)
athena-regex-rewrite: "/"

The 'no-publish' Service

There may be circumstances where one does not wish to publish all paths in an Ingress object as separate interfaces in the Athena application, yet they must still be made available to users. Any Ingress Path with a Backend Service Name containing 'no-publish' somewhere in its name will not be displayed as a separate interface to users within the application, but will still be available should a user request the path via another interface. An example of this can be found in the 'template2' Template provided with this application. The VNC client has its path published as an interface, and when a user loads the interface, the VNC client opens a WebSocket connection to the unpublished path.

Reviewing an Existing Lab Template

In this section, we will review the Juice Shop Lab Template supplied with the Athena system. This template is based on the OWASP Juice Shop image developed by Björn Kimminich, and demonstrates every Kubernetes object type used in a Lab Template.

Deployment

This Deployment file describes how the single Pod in this Lab should be deployed. Each individual system in a lab requires its own Deployment file — however, if creating a deployment file for each system seems like a daunting task, consider the following: once one has a single known-good Deployment file, it is relatively simple to alter it to fit one's needs. The deployment file used for this template is a good example of this - it is very basic but provides a skeleton to add additional tweaks should another system be added.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: juice-shop-deployment
  labels:
    lab: juice-shop
spec:
  selector:
    matchLabels:
      lab: juice-shop
  template:
    metadata:
      labels:
        lab: juice-shop
    spec:
      containers:
        - name: juice-shop
          image: bkimminich/juice-shop
          ports:
            - containerPort: 3000
```

Fields of Interest

Kind: Indicates that this is a Deployment resource.

Name: All object names in a Template are arbitrary — however, it is still important to consider how common a Deployment's name is across the entire library of Lab Templates. If the name is too common, a user may be prevented from creating another lab environment due to a name conflict.

Labels: Each label takes the form of a key-value pair - these are (again) arbitrary, but it is critical that each Lab Template has a unique 'lab:' label value.

Label keys and values can contain alphanumeric characters, along with the '-', '.', and '/', and both must start and end with an alphanumeric character.

Labels are used when selecting resources; this can be seen in the selector section of this object.

Selector: In the context of a Deployment, Selectors determine the Pod that the Deployment object should manage.

While multiple selectors types exist, the most widely available across different resources is the 'MatchLabels' selector. The MatchLabels selector attempts to match a resource (in this case, a Pod) with a certain label (or labels). If multiple label values are present in a MatchLabels selector, all labels must be present on the resource being matched.

Template: The contents of the Templates section of a Deployment describes a Pod that should be created by the Deployment. One should ensure that the Pod described by the Template section contains the labels specified in the Selectors section.

Containers: Each item in the containers section describes a container that should run in a Pod. At a minimum, each item should have a name, an image, and a port (or ports) that the container can be reached on.

Service

As explained earlier, Services connect Pods to the wider network - any Pod that must be reachable from an Ingress (as opposed to via another Pod) probably needs a Service. As this is a very basic Template, only one Service is required.

In addition to the standard ClusterIP Service, one could also choose to configure a Service to be a NodePort or LoadBalancer service — however, this is generally not recommended for Lab Templates, as the demand for node ports or LAN IP addresses could outstrip supply.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: juice-shop-interface
  labels:
    lab: juice-shop
spec:
  selector:
    lab: juice-shop
  ports:
    - port: 80
      targetPort: 3000
```

Fields of Interest

Name: The Service is an object which is often identified by its name alone. The Service above would be reachable via DNS lookup for juice-shop-interface.<namespace-name>, should the DNS lookup come from within the Service's namespace.

Selector: Similarly to the Deployment above, this Service is selecting a Pod based on the presence of the *lab: juice-shop* label; however, in this case, once the Service has located a Pod which meets its criteria, it will forward traffic it receives to the selected Pod.

Ports: We see from this section that the Service is receiving traffic on port 80, and forwarding it to port 3000. While it may not seem like much, this can be very useful when trying to repurpose an existing Docker image - instead of spending time trying to determine how to serve traffic on another port, one can simply direct traffic intended for one port to a completely different port.

Ingress

The Ingress is the recommended way of connecting users with their Lab Environment. The Ingress acts in a similar fashion to a reverse proxy, retrieving resources from Services on behalf of external clients. These Services then forward the requests from the Ingress to their corresponding Pod to handle the request.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: nginx-athena
    nginx.ingress.kubernetes.io/rewrite-target: /$1
    athena-regex: (.*)
  labels:
    lab: juice-shop
  name: ingress-juice
spec:
  rules:
    - host: athena.err0r.stream
      http:
        paths:
          - backend:
              serviceName: juice-shop-interface
              servicePort: 80
            path: /juice-shop/(.*)
```

Fields of Interest

apiVersion: Note that earlier versions of the Ingress have the apiVersion of extensions/v1beta1. Ensure that any Ingress used with the Athena application has an apiVersion of networking.k8s.io/v1beta.

Annotations: The Ingress for this template is the first (and only) example of annotations in this Lab Template. Annotations provide information about an object but are not a potential selection criterion (unlike labels).

The first annotation indicates that the Ingress Controller with the ingress class value matching 'nginx-athena' should manage traffic for this Ingress.

The second annotation — 'rewrite-target'— tells NGINX to rewrite the requested path to "/" + whatever the 1st regular expression found in the path matches in the user's requested path (The Kubernetes Authors, 2020).

The last annotation ensures that the path presented to a user in the Athena web application does not contain regular expressions. The 'athena-regex' value tells the Athena application what regular expression pattern it should remove from the path prior to presenting it to users.

Host: This field defines the hostname that the Ingress will respond to requests for.

Backend: The Service which will receive a user's request, and the port that it will receive it on

Path: The request path that the Ingress will forward traffic for. In cases like this, where the path contains a regular expression, the Ingress will forward any traffic that matches the regular expression.

This Ingress would forward traffic for both http://athena.err0r.stream/juice-shop/apple-n-orange and http://athena.err0r.stream/juice-shop/apples to the juice-shop-interface Service on port 80.

NetworkPolicy

By default, Kubernetes does not block any traffic between Pods, whether they are within the same Namespace or in a different one. Because of this, the Kubernetes Authors suggest that a 'deny-all' NetworkPolicy should be implemented to prevent unwanted traffic from reaching Pods (The Kubernetes Authors, 2020). This policy could prevent incoming traffic, outgoing traffic, or both. The cluster administrator is then free to apply additional NetworkPolicies to a namespace.

When multiple NetworkPolicies match a particular Pod, the policy that is applied is a union of the policies that match the Pod. The following NetworkPolicy was written operating under the assumption that all traffic that is not explicitly allowed is blocked by a previously installed policy

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: juice-shop-policy
  labels:
    lab: juice-shop
spec:
  podSelector:
    matchLabels:
      lab: juice-shop
  ingress:
    - from:
      - podSelector:
          matchLabels:
            lab: juice-shop
      - namespaceSelector:
          matchLabels:
            app.kubernetes.io/name: ingress-nginx
  egress:
    - to:
      - podSelector:
          matchLabels:
            lab: juice-shop
      - namespaceSelector:
          matchLabels:
            app.kubernetes.io/name: ingress-nginx
```

Fields of Interest

PodSelector: The podSelector in this NetworkPolicy indicates which Pods this policy should apply to.
From/To: Each 'from' or 'to' item is a rule which identifies a type of traffic to allow in or out of the selected Pod(s). In this case, both sections have the same NamespaceSelector.

NamespaceSelector: This selects Pods from a Namespace which has the label app.kubernetes.io/name: ingress-nginx. This matches the namespace used by ingress-nginx, allowing traffic to and from the Ingress Controller.

Troubleshooting Common Configuration Errors

In this section, we examine several scenarios that may arise when testing Lab Templates. While it is often possible to resolve issues simply by reexamining Template files, troubleshooting will be made much simpler if one has access to the Kubernetes API and the **kubectl** application.

None of the lab interfaces for a Pod are responding.

There are two categories of problem that the root cause of this issue could fall into: one, that the Pod is not running properly, or two, is not reachable by the Ingress.

1. Verify that the Pod is in a 'Running' state and that it does not have an abnormal number of restarts (most Pods will not need to restart at all).

   The **kubectl get pod -n [your-Athena-user-ID]** command will display all Pods in your namespace, and the **kubectl describe pod -n [Athena-userID] [pod-name]** command will show the recent history of the Pod, including any crashes or errors.

2. If the Pod is not in a 'Running' state, or has recently crashed, check its logs using the **kubectl logs -n [Athena-userID] [Pod-name]** command**.**
   Issues with a container within a Pod can often be diagnosed and resolved from a local Docker environment.
   Otherwise, verify that you have specified the correct Docker image in your Deployment file for the Pod.

3. Verify that Ingresses have endpoints to direct traffic to and that the paths to your Pod interfaces are configured as you expect. You can examine existing Ingress objects using the following command:

   kubectl describe ingress -n [Athena-userID] [Ingress-name]

4. Verify that you have supplied Service files with the lab environment, and that these are configured to direct traffic to the appropriate port on your Pod. Also verify that the Services are selecting the correct Pod, either by querying the cluster directly using the

   kubectl describe service -n [Athena-userID] [Service-name]

   command, or by comparing your Deployment file and the Service file(s) which target the Pod specified within.

5. Check that NetworkPolicy is not preventing traffic from the Ingress Controller reaching your Pod. Issuing the **kubectl get networkpolicy -n [your-Athena-user-ID]** command will display all the NetworkPolicy objects in your namespace.

   Examine these policies using the **kubectl describe networkpolicy -n [your-Athena-user-ID] [network-policy-name]** command. The command's output will show how Kubernetes is interpreting the policy.

6. At this point, it is unlikely that there is an issue with the configuration of the Lab Template files. If you have not done so already, test the Docker image on your own system, mapping the ports of the container to your host system. The following command will start a Docker container using your image, and will publish the ports of the container to random ports on your host machine:

   docker run -rm -d -P <image-name>

   Verify that you can reach the container's services without further interaction with the container and that its behaviour matches your expectations.

I can't access a lab interface, but other interfaces for the same Pod are working as expected.

1. Check the URL that is being presented to the user within the Athena application. If this matches your expectations, move on to the next steps.

   If it does not, alter your Ingress file so it does match what you expect. If you are using regular expression matching, you **must** use the 'athena-regex' annotation (and potentially the 'athena-regex-replace' annotation as well).

2. Verify that Ingresses have endpoints to direct traffic to and that the paths to your Pod interfaces are configured as you expect.

   If you do not have access to the cluster, all you can do at this stage is verify that your Ingress file is directing traffic where it should. Pay close attention to the backend Service and port for each path.

   If you have been given access to the Kubernetes cluster, you can view the current endpoints for an active Ingress using the command:

   kubectl describe ingress -n [Athena-userID] [Ingress-name]

   The output of this command will show you how the Kubernetes cluster is interpreting your Ingress YAML file, and the Pod endpoints it has been able to connect to via a Service.

If all Ingress paths have endpoints, this means that all Services are active, and were able to select an endpoint to direct traffic to.

If any Ingress paths do not have endpoints, there is either a mistake in your Ingress file — you are trying to direct traffic to a Service that doesn't exist, or have specified the wrong port— or there is an issue with the endpoint itself.

3. Check that Services are selecting the right Pods, and that Ingresses are configured to direct traffic to the right Service on the right port.

   If there are paths missing an endpoint, verify that the Service has been created and is functioning as expected by using the following command:

   kubectl describe service -n [Athena-userID] [Service-name]

   This will show which Pod the Service has selected, the ports it is receiving traffic on, and where it is forwarding that traffic to.

4. If the Service has not selected a Pod (or has selected the wrong Pod), one should update the Service's Pod selector so that it will exclusively target the correct Pod. This may require you to add additional labels to both the Service's selector and the Pod's specification within your Deployment file.

I've verified that the Ingress is directing traffic to the appropriate container endpoint, but there is still no output from the interface.

Check that all required Ingress paths are defined so that the path requested matches the path value for your Ingress. This may involve implementing regular expression matching in situations where the container interface is expected to display multiple different files — for example, a web page which loads images hosted by the container in a subdirectory.

The Lab Environment is taking a long time to load.

Complex environments will take more time to deploy than simple environments. If all containers are working as expected, the only option is to reduce the size of the Docker images used in the environment, which can be done by reducing the number of running applications and services within a container.

If the Lab Environment takes a long time to load, and a container is not responding as expected, this could be because the container is failing to start or is crashing repeatedly. If you suspect this is the case, you can check the container's logs while the Lab Environment is running by using the following **kubectl** command on a master Node:

kubectl logs -n [your-Athena-userID] [pod-name] [container-name]

If you do not know the Pod's name, it can be found with the following command:

kubectl get pods -n [your-Athena-userID]

The container logs will contain the output of the container's standard output.