Docker for Enterprise Operations Exercises

Some of these exercises ask you to run things on your local machine. In order to do so, you will need Docker as well as some other bash utilities installed; in case this is a problem, the ‘elk‘ node (username ‘ubuntu‘) has everything you'll need all pre-configured. **When the instructions refer to 'your' machine, this means either your laptop or the ‘elk‘ node - whichever you have chosen to use.**

Be aware that copy-pasting of commands or code snippets from this PDF may apply changes to some characters e.g. quotes, tabs, which may leads to errors. Please consider typing suggested commands and code snippets in case you encounter any issues.

Windows Users: Please note that in all exercises we will use Unix style paths using forward slashes (‘/‘) instead of backslashes (‘\‘). On Windows you can work directly with such paths by either using a **Bash** terminal or a **Powershell** terminal. Powershell can work with both Windows and Unix style paths.

Contents

1 Install UCP

In this exercise, you'll set up a single Universal Control Plane manager, with a pair of worker nodes.

Pre-requisites:

- Three nodes running centos:7.3 or later, and Docker EE 17.06.2-ee or later, named as per:
- `ucp-manager-0`
- `ucp-node-0`
- `ucp-node-1`
- A UCP license. If you don't have one, a trial license is available at https://store.docker.com/bundles/docker-datacenter/purchase?plan=free-trial.

1.1 Checking ntp

1. While not strictly required, the `ntp` utility ensures clock synchronization across your cluster, helping you avoid a number of related problems. Check that `ntp` is running on `ucp-manager-0`, `ucp-node-0` and `ucp-node-1`:

```
$ sudo /bin/systemctl status ntpd.service
  ntpd.service - Network Time Service
    Loaded: loaded (/usr/lib/systemd/system/ntpd.service; enabled; vendor preset: disabled)
    Active: active (running) since Sat 2017-10-14 02:14:40 UTC; 39min ago
    ...
```

Note it's important that `ntp` is up and running *before* installing UCP.

1.2 Installing UCP

1. On `ucp-manager-0`, use the UCP bootstrapper to install UCP:

```
$ UCP_IP=<ucp-manager-0 IP>          #Public IP address of ucp-manager-0
$ UCP_FQDN=<ucp-manager-0 FQDN>      #FQDN of the ucp-manager-0
$ docker container run --rm -it --name ucp \
    -v /var/run/docker.sock:/var/run/docker.sock \
    docker/ucp:3.0.2 install \
    --admin-username admin \
    --admin-password adminadmin \
    --san ${UCP_IP} \
    --san ${UCP_FQDN}
```

1.3 Licensing the installation

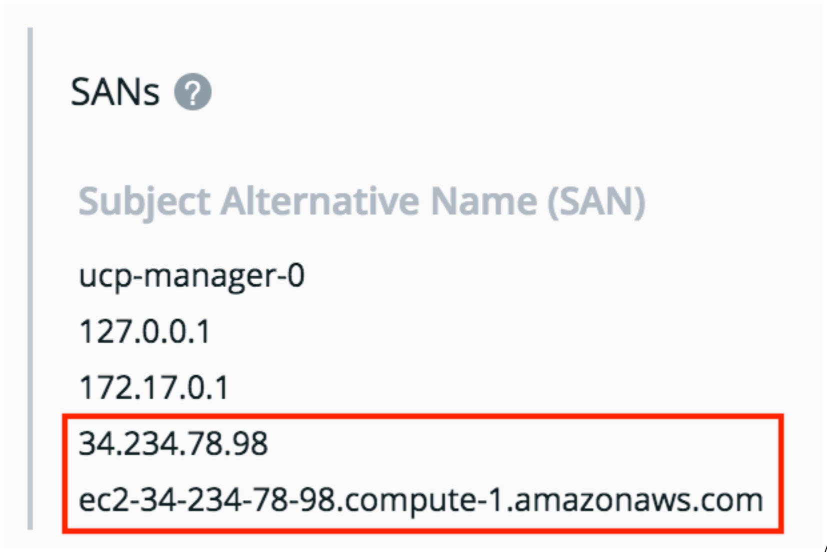
1. In your browser, visit the UCP interface at `https://<public IP of ucp-manager-0>`.

   Note: Most browsers will display some sort of warning message regarding the connection. This is because we are using the default self signed certificates as opposed to a certificate from a trusted source. You can tell the browser to ignore the warning and proceed.

2. Login to your Admin account; as per the flags in the install command above, your username is `admin` and your password is `adminadmin`.
3. Once logged in you will see a prompt screen asking you to upload a license. Click the **Upload License** button and upload your UCP license; if you need a trial license, click the **Get free trial license** button, and follow the prompts.

1.4 Checking SANs

1. On the left sidebar, navigate **Shared Resources** -> **Nodes** -> **ucp-manager-0** and then look on the right sidebar for **SANS** (Subject Alternative Names), and click on it. Confirm that the FQDN and public IP of the `ucp-manager-0` node are in the list. You specified these at install via the `--san` flag to the UCP bootstrapper. **Failing to specify SANs correctly will cause TLS failures when communicating with other nodes if left unaddressed,** since these SANs are added to this node's certificate.

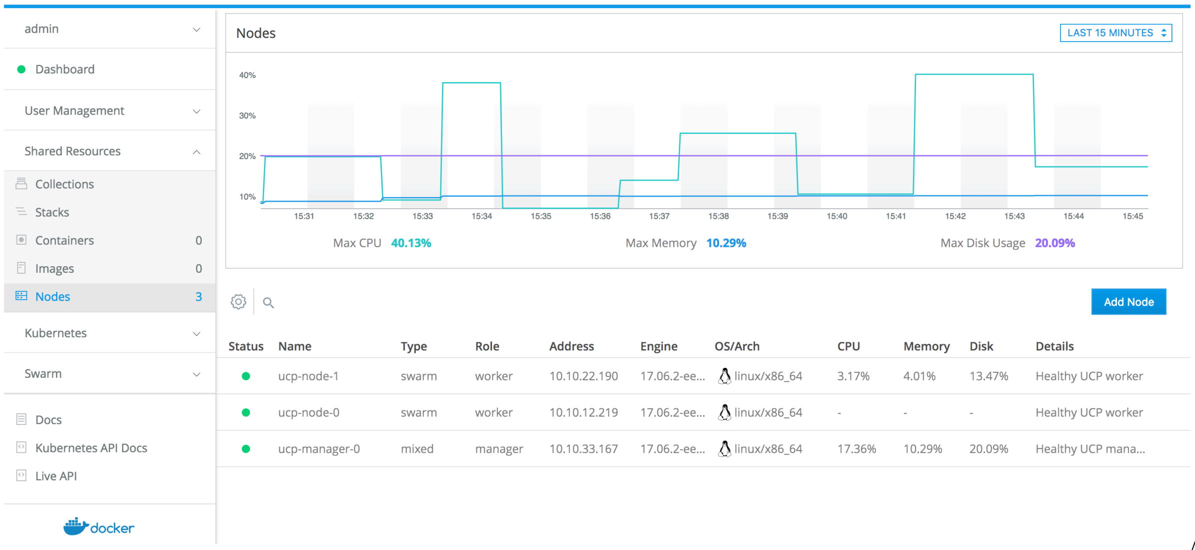


Subject Alternative Name (SAN)

ucp-manager-0
127.0.0.1
172.17.0.1
34.234.78.98
ec2-34-234-78-98.compute-1.amazonaws.com

1.5 Adding additional worker nodes

1. Navigate **Shared Resources** -> **Nodes** -> **Add Node**. A swarm join token is presented which can be used to add workers to your UCP cluster.
2. Use the `docker swarm join...` token presented in the last step on `node-0` and `node-1`. After a moment, they should be visible on the **Nodes** view of UCP as workers.



Max CPU 40.13% Max Memory 10.29% Max Disk Usage 20.09%

Status	Name	Type	Role	Address	Engine	OS/Arch	CPU	Memory	Disk	Details
●	ucp-node-1	swarm	worker	10.10.22.190	17.06.2-ee...	linux/x86_64	3.17%	4.01%	13.47%	Healthy UCP worker
●	ucp-node-0	swarm	worker	10.10.12.219	17.06.2-ee...	linux/x86_64	-	-	-	Healthy UCP worker
●	ucp-manager-0	mixed	manager	10.10.33.167	17.06.2-ee...	linux/x86_64	17.36%	10.29%	20.09%	Healthy UCP mana...

1.6 Conclusion

At this point, UCP is installed with one manager and two worker nodes. Bear in mind that UCP is really just a collection of containers running on a swarm. Alternatively, we could have set up a swarm from the command line first, and installed UCP on top of this if we preferred; UCP would have automatically recognized and integrated all nodes.

2 Adding UCP Manager Nodes

In this exercise, we will add 2 additional manager nodes to our UCP installation in order to make it highly available.

Pre-requisites:

- UCP installed with 2 worker nodes
- Two additional nodes running centos:7.3 or later, and Docker EE 17.06.2-ee or later, named:
- `ucp-manager-1`
- `ucp-manager-2`

2.1 Checking ntp

1. Like we did for our other UCP nodes, make sure `ntp` is running on `ucp-manager-1` and `ucp-manager-2`:

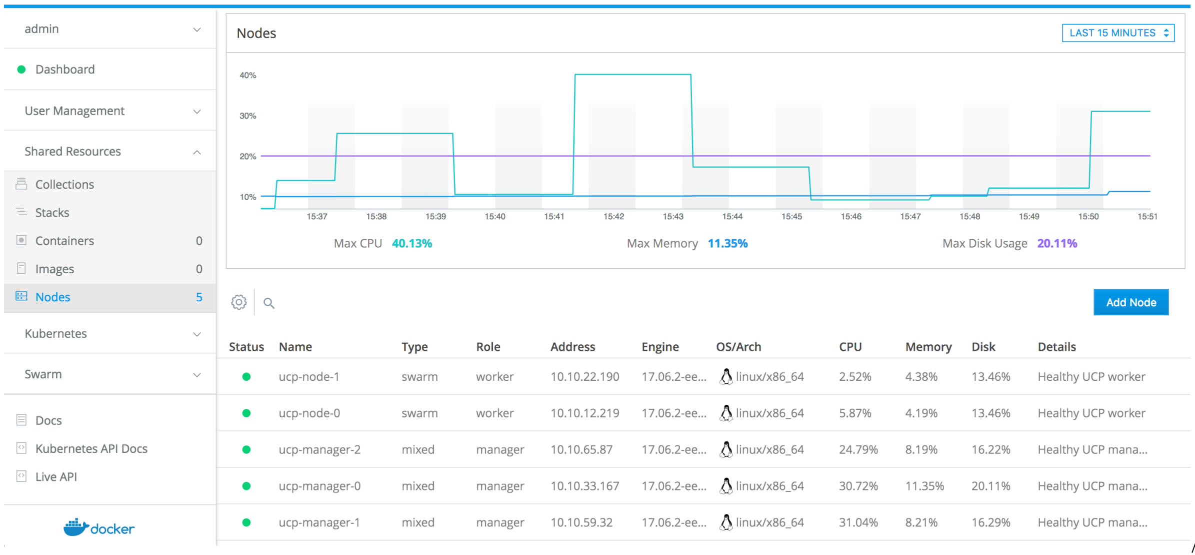
   ```
   $ sudo /bin/systemctl status ntpd.service
   ```

   If it isn't, `sudo /bin/systemctl start ntpd.service` should start the serivce.

2.2 Adding manager nodes

1. Return to the same UCP page visited previously to get a join token for workers.
2. Move the toggle from **Worker** to **Manager**. Note that the join token changes.
3. Open terminal connections to `ucp-manager-1` and `ucp-manager-2`, and use this new join command to add these nodes as managers.

4. Go back to the **Nodes** page on UCP. You should now see 5 nodes, similar to the following:



Max CPU 40.13% Max Memory 11.35% Max Disk Usage 20.11%

Status	Name	Type	Role	Address	Engine	OS/Arch	CPU	Memory	Disk	Details
●	ucp-node-1	swarm	worker	10.10.22.190	17.06.2-ee...	linux/x86_64	2.52%	4.38%	13.46%	Healthy UCP worker
●	ucp-node-0	swarm	worker	10.10.12.219	17.06.2-ee...	linux/x86_64	5.87%	4.19%	13.46%	Healthy UCP worker
●	ucp-manager-2	mixed	manager	10.10.65.87	17.06.2-ee...	linux/x86_64	24.79%	8.19%	16.22%	Healthy UCP mana...
●	ucp-manager-0	mixed	manager	10.10.33.167	17.06.2-ee...	linux/x86_64	30.72%	11.35%	20.11%	Healthy UCP mana...
●	ucp-manager-1	mixed	manager	10.10.59.32	17.06.2-ee...	linux/x86_64	31.04%	8.21%	16.29%	Healthy UCP mana...

Note: It takes a few minutes for a node to be set up as a UCP manager. This is reflected in the **Details** column. At first it will say that the node is Pending, and when finished it will say *Healthy UCP Manager*. Red warning banners at the top of UCP are normal during node configuration, and should resolve themselves once the nodes finish setting up. Be patient! Interrupting the manager join process is a common way to corrupt a cluster.

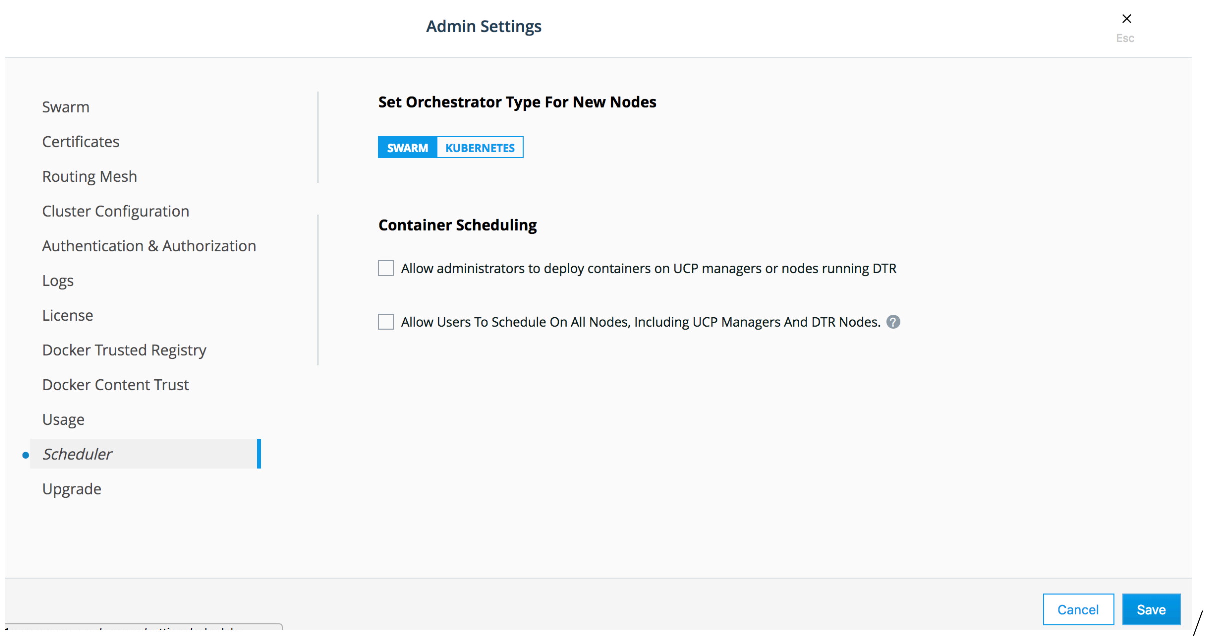
2.3 Testing manager nodes

1. Open a new browser tab and put in the domain or public IP of either your `ucp-manager-1` or `ucp-manager-2` node (with `https://`). You should see the UCP login page.
2. Verify that you can login with the **Admin** account.

2.4 Configuring the Scheduler

Now that we have 3 manager nodes setup, we want to make sure that our manager nodes are dedicated for their purpose and that they do not run any application containers.

1. In UCP, navigate **admin** -> **Admin Settings** -> **Scheduler**.
2. Uncheck both of the checkbox settings on the page as shown below, and be sure to click **Save** afterwards:



Set Orchestrator Type For New Nodes

SWARM KUBERNETES

Container Scheduling

☐ Allow administrators to deploy containers on UCP managers or nodes running DTR

☐ Allow Users To Schedule On All Nodes, Including UCP Managers And DTR Nodes.

This will prevent containers from being scheduled on any of our UCP manager nodes.

2.5 Conclusion

Your UCP cluster now has three manager nodes, preventing your cluster from collapsing if one of them fails. Bear in mind that this manager consensus is exactly a swarm manager consensus; UCP controllers are just swarm managers, with additional containers running on top to serve UCP.

3 UCP API & Client Bundles

In this exercise, you'll manipulate your Universal Control Plane from a remote machine. Try these exercises on your own laptop if you're able to install Docker there; if not, try them from your `elk` node, either directly from the command line or through the remote desktop. Remeber this is an ubuntu node! The username will therefore be `ubuntu`.

Pre-requisites:

- UCP installed and running.

3.1 Establishing an Auth Token

1. In order to authenticate with UCP, we need an auth token. Fetch and save one to `$AUTHTOKEN` with the following commands. These API calls can be made from any machine that can reach your UCP manager; try it on your local laptop first. If the necessary command line utilities can't be installed there, any VM in your cluster will do.

```
$ UCP_FQDN=<UCP FQDN>
$ AUTHTOKEN=$(curl -sk -d '{"username":"admin","password":"adminadmin"}' \
      https://${UCP_FQDN}/auth/login | jq -r .auth_token)
```

   Note you could do this as any user, not just `admin`; your ability to take action through the API is limited by the same RBAC rules as the web client.

2. For convenience, create an alias for issuing API commands:

```
$ alias ucp-api='curl -k -H "Authorization: Bearer $AUTHTOKEN"'
```

3.2 Using Client Bundles

This section requires Docker, `kubectl`, a bash shell, `curl` and `jq` to be installed on your local machine. All of this is pre-configured on your `elk` node if you don't want to run it locally.

One way to control UCP is to issue Docker CLI commands from a remote node to a UCP cluster. Those commands will be governed by the same access control rules imposed on the authenticated user as when they interact with UCP from the web UI or API.

1. Use the UCP API to download the client bundle for your UCP account:

   ```
   $ ucp-api https://${UCP_FQDN}/api/clientbundle -o bundle.zip
   ```

2. Point your local Docker CLI at your remote UCP using the certificates in the `bundle.zip` you just downloaded:

   ```
   $ unzip bundle.zip
   $ eval "$(<env.sh)"
   ```

3. Issue any Docker CLI commands you like; the response will reflect your UCP cluster, rather than your local machine:

   ```
   $ docker container ls
   $ docker volume create bundle-vol
   ```

   Check that that volume got created via your UCP web UI. Which node did it get created on, and why?

4. These same credentials will also allow you to issue `kubectl` commands to the Kubernetes assets in your cluster. For example, put the following in a file `pod.yml`:

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: nginx
   spec:
     containers:
     - name: nginx
       image: nginx:1.7.9
       ports:
       - containerPort: 80
   ```

   Then create this as a pod, and confirm it exists on your UCP web UI:

   ```
   $ kubectl create -f pod.yml
   ```

5. You can also use the client bundle credentials to hit the Kubernetes master API directly:

   ```
   $ curl -s --cacert ./ca.pem --key ./key.pem --cert ./cert.pem \
       https://${UCP_FQDN}:6443/api/v1/pods
   ```

   You should see all the system pods UCP spins up to provide Kubernetes functionality.

6. Point your local CLI back at your local Docker engine, unset all the environment variables `env.sh` set:

   ```
   $ unset DOCKER_HOST
   $ unset DOCKER_CERT_PATH
   $ unset DOCKER_TLS_VERIFY
   $ unset KUBECONFIG
   ```

3.3 Using the API Docs (Optional)

Besides the CLI, authenticated commands can be issued to UCP by posting JSON data to UCP's RESTful API.

1. Make sure a couple of non-system containers are running on your swarm (anything will do). For example, in UCP, navigate **Swarm** -> **Services** -> **Create Service**, name your service `api-demo`, and specify `nginx:latest` as the image, and click **Create**.
2. From UCP's left sidebar, navigate **Live API** -> **GET** under the 'Container' heading. This provides usage examples on this command.
3. Click **Try It Out** -> **Execute** without filling in any of the optional fields. The API call is constructed and sent, showing an example of the syntax as a `curl` command, and the resulting output.
4. Back on your local machine where you set up your auth token, try:

   ```
   $ ucp-api https://${UCP_FQDN}/containers/json | jq
   ```

   The same result as the demo is produced (`jq` here pretty-prints the json returned by the curl. Feel free to substitute any other utility that does so).

3.4 Making a Secret

1. Create a file `mysecret` containing any short piece of text you'd like to distribute as a secret across your swarm.
2. The syntax to create a secret is described in the API docs under the POST request for the route `secrets/create`. But, the secret content must be base 64 encoded. Register your secret in your cluster with the correct encoding:

   ```
   $ ucp-api --data '{"Data":"'$(base64 mysecret)'","Name":"test_secret"}' \
       -X POST https://${UCP_FQDN}/secrets/create
   ```

3. From the web interface, make a service and mount the secret:
   - From the lest sidebar: **Swarm** -> **Service** -> **Create**.
   - Name it `demo`, and specify as an image "nginx".
   - Attach the secret: in the service configuration -> **Environment** -> **Use Secret +** -> Select secret name and confirm.
   - Finally, don't forget to hit **Create**.
4. Now, we want to check that the secret got mounted:
   - To get the id of the container where the service task is running, click on the service, and on the right sidebar click **Inspect Resource** -> **Containers**.
   - Select the container and check on which node it is running from the sidebar.
   - ssh to the node in which the container is running, and open a terminal inside the service's container, as we did before.

     ```
     $ docker container exec -it <container ID> bash
     ```

   - Check that the secret got mounted at `/run/secrets/test_secret`, and has the value you expect.

3.5 Creating a Service

1. Use the API to inspect the `demo` service you created in the last step. Notice that even this simple service has a large JSON object describing it.
2. Use the API to delete your `demo` service. Recall `curl` makes a `DELETE` request when the flag `-X DELETE` is used.
3. A similar JSON object to the one found above is used to create a new service. Entering this at the command line would be a nuisance, so create a basic service definition in a file `myservice.json`:

```
{
  "Name": "demo",
  "TaskTemplate": {
    "ContainerSpec": {
      "Image": "nginx:latest"
    }
  }
}
```

4. Create a service using the definition in `myservice.json`:

   ```
   $  ucp-api --data @myservice.json https://${UCP_FQDN}/services/create
   ```

   Check to see that your service is running in the web UI as you'd expect, and delete it.

5. Modify the content of `myservice.json` so that port 80 internal to the container is exposed on port 8080 external to the container.

   Hint 1: check the 'Try It Out' example in the API docs for the `POST` to `/service/create` for an example of the JSON structure you'll need.

   Hint 2: explore the Docker API docs https://docs.docker.com/engine/api/v1.30/ for detailed explanations of how most Docker client commands map to API endpoints.

   Re-create the service as above, and make sure you can see the nginx landing page at `<UCP FQDN>:8080`.

6. Finally, modify `myservice.json` again to attach the `test_secret` secret to this service. Launch the service and check that the secret was mounted correctly.

3.6 Conclusion

Anything that can be done in the UCP UI can be done through the API illustrated in this exercise. Under the hood, the UCP web app is issuing calls to the exact same API, all reachable on port 443 of any UCP manager node. All the same holds true for DTR's web interface and underlying API. Also note, there's nothing special about `curl` in this context; any tool to issue these requests, like Chrome's postman, will have the same effect.

4 Access Control in UCP

In this exercise, we'll describe a simple set of users and assets at a startup called Whalecorp; at each step, create the entities and permissions described in your UCP instance.

Pre-requisites:

- A working UCP installation with at least two worker nodes, `ucp-node-0` and `ucp-node-1`

4.1 Designing Your RBAC Model

For the following steps, sketch out resource collections, teams and organizations, roles and grants with *pen and paper*. We'll click through UCP later, but first we need to come up with an appropriate plan.

1. Whalecorp has the following org chart:

   - **Development**: Joey and Shaun
   - **QA**: Kelly and Barry
   - **Operations**: Chloe

   Design a UCP organization and team structure for Whalecorp that reflects employee roles.

2. The app Whalecorp is currently developing has the following characteristics:

   - requires a Docker overlay network and a secret

- Development and QA will each run their own separate instance, and should not be able to see or affect the other's.
- Operations staff should be able to access and manipulate both instances with a single grant.

Plan out some resource collections, roles and grants that will enable you to achieve the above requirements.

3. In order to avoid collisions between development and QA, plan your resource collections such that Joey and Shaun can only deploy services to `ucp-node-0`, and Kelly and Barry can only deploy services to `ucp-node-1`. Chloe can deploy on either machine.
4. During development, developers (Joey and Shaun) may need to exec into a running container to examine their application in-flight. But, the QA employees shouldn't be able to exec into a running container, so that secrets aren't exposed to them. Meanwhile, Chloe should have unrestricted access to all running containers. Change the roles you granted above if necessary to ensure that this is the case.

4.2 Configuring RBAC in UCP

Once everyone has had a chance to complete the above plan, the instructor will demo setting up the corresponding teams, organizations, resource collections and grants in UCP:

- *Instructor*: invite the class to call out their answers to each step, and implement what they describe.
- *Participants*: try and click along with the instructor to create the same role based access control model on your own UCP.

4.3 Conclusion

Planning out a role based access control model before touching UCP is a good way to avoid redundancy and difficult-to-manage list of grants. Often, the fewer grants you can make, the better; that way, your list of grants is easy to audit and understand. Leverage the 'trickle down' model of RBAC that UCP uses, when appropriate: grants to organizations apply to all teams and all users in those organization, and grants to parent resource collections apply to child resource collections. This trickle down structure is useful for locking down low- or no-permissions generally, and for granting general permissions to the team leaders or senior staff who need them.

5 User Management with LDAP

Instead of managing users directly from UCP, an LDAP server can define users.

Pre-requisites:

- UCP installation.

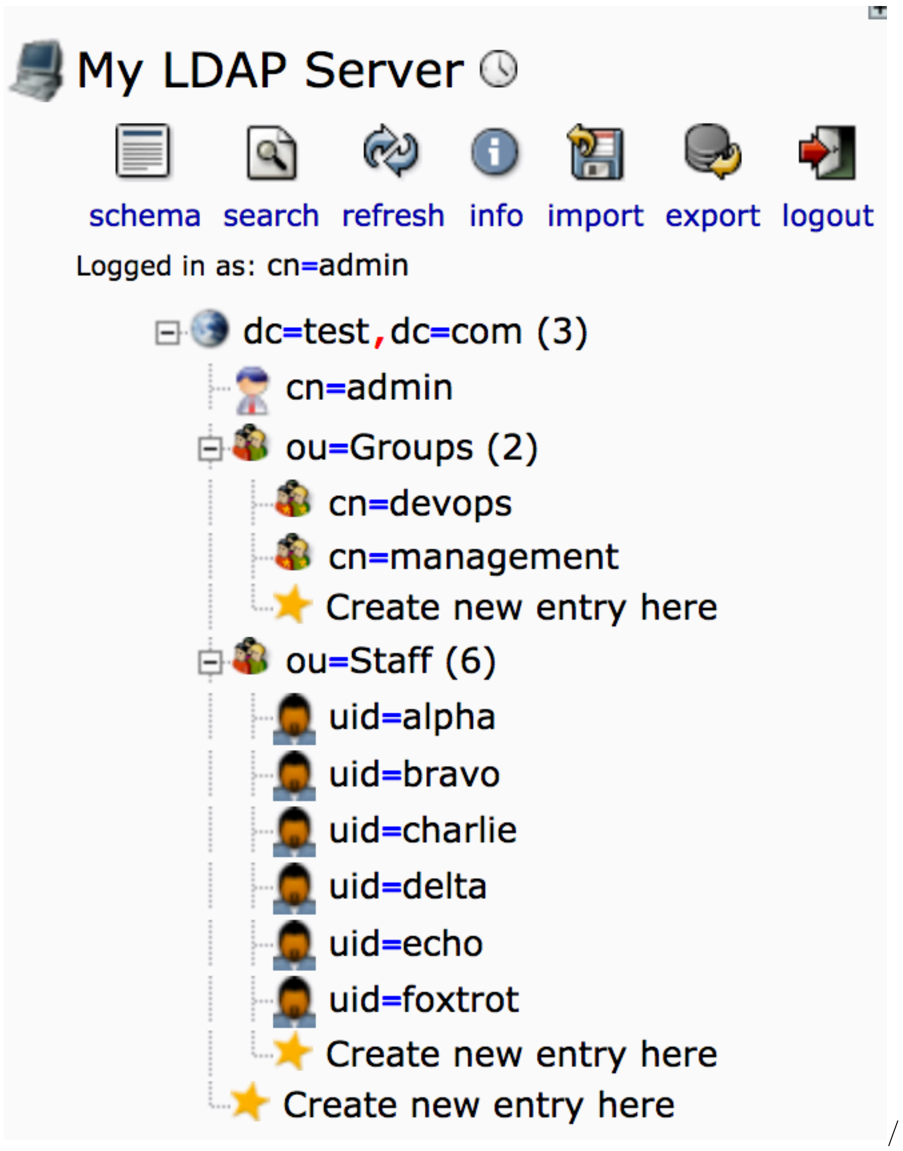
5.1 Setting Up the LDAP Server

1. On your `elk` node, spin up a containerized LDAP demo:

```
$ docker container run -d -p 80:80 -p 389:389 training/ldap:latest
```

2. Open a browser and go to `http://<elk FQDN>/phpldapadmin`. This will open up the GUI for your LDAP server.
3. Login to the server with the following credentials:

   Login DN: `cn=admin,dc=test,dc=com` **Password:** `admin`
4. Expand the left navigation bar and take note of the LDAP entries underneath `ou=Staff`, and `ou=Groups`:


My LDAP Server

Logged in as: cn=admin

dc=test,dc=com (3)
cn=admin
ou=Groups (2)
cn=devops
cn=management
Create new entry here
ou=Staff (6)
uid=alpha
uid=bravo
uid=charlie
uid=delta
uid=echo
uid=foxtrot
Create new entry here
Create new entry here

5.2 Integrate UCP and LDAP

1. Login to UCP as the admin user.
2. Navigate **Admin Settings** -> **Authentication and Authorization**, and click 'yes' under **LDAP Enabled**.
3. In the 'LDAP SERVER' form, fill in:
   - LDAP Server URL: `ldap://<elk FQDN>`
   - Reader DN: `uid=bravo,ou=Staff,dc=test,dc=com`
   - Reader Password: `password`
4. Click **Add LDAP User Search Configurations** + and fill in:
   - Base DN: `dc=test,dc=com`
   - Username Attribute: `uid`
   - Fullname Attribute: `cn`
   - Filter: `objectClass=inetOrgPerson`
   - Check 'Search subtree instead of just one level'
   - Click **Confirm**
5. Under 'LDAP Test Login', fill in:

- Username: bravo
- Password: password

6. Still under 'Test LDAP Login', click **Test**. A successful login test should be reported.
7. Under 'LDAP Sync Configuration', enter 24 for the **Sync Interval (Hours)** field.
8. Click **Save**.

5.3 Testing User Access

1. In the **LDAP Sync Jobs** section, click on the **Sync Now** button.
2. Return to the UCP dashboard, and navigate **User Management** -> **Users**. None of the LDAP users appear listed here. UCP only moves user accounts over when the user first logs in.
3. Logout of UCP as the admin user, and log back in as user `bravo` / password `password`.
4. Log back in as the admin user, and check the **User Management** view again. `bravo` now appears listed here.

5.4 Populating Teams from LDAP Groups

1. Go back to the LDAP configuration page (**admin** -> **Admin Settings** -> **Authentification & Authorization**):
   - uncheck 'Just-in-time User Provisioning'
   - Save the config
   - and click 'Sync Now'.

   This is not strictly necessary for populating teams, but will make it easier to see who is getting put in your teams without having to log in as each one to trigger the JIT provisioning.
2. Create an engineering organization in the **User Management** panel on the left.
3. Make a team `devops` in your engineering organization, and click 'Yes' under 'Enable Sync Team Members' when creating the team.
4. Click 'Match Group Members' to populate this team base on an LDAP group.
5. Fill out the subsequent fields:
   - Group DN: `cn=devops,ou=groups,dc=test,dc=com`
   - Group Member Attribute: `uniqueMember`
6. Click 'Yes' under 'Immediately Sync Team Members', and finally click 'Create'.
7. Navigate back to your new devops team users listing; the same users in the devops team in the LDAP browser should now be members of your devops team in UCP.

5.5 Populating Teams from LDAP Searches

1. Make a team `staff` in your engineering organization, and click 'Yes' under 'Enable Sync Team Members' when creating the team.
2. Click 'Match Search Results' to populate this team based on an LDAP search.
3. Enter `dc=test,dc=com` for 'Search Base DN'
4. Enter `ou=Staff` for 'Search Filter'
5. Check 'Search subtree instead of just one level', and 'Yes' for 'Immediately sync team members'
6. Click 'Create' at the bottom, and navigate to the user list for the `staff` team. The team is populated with users matching your search criteria.

5.6 Cleanup

1. Disable the LDAP integration by selecting **No** under 'LDAP Enabled' on the Authentication and Authorization admin settings page.
2. Click **Save**.

5.7 Conclusion

In this exercise, we imported users from an LDAP server. Users will be populated with just-in-time provisioning on their first login by default, unaffiliated with any team. UCP teams can be populated from LDAP groups or by LDAP searches.

6 Password Recovery

In this exercise you will see how to recover from a scenario where the password to the built-in UCP admin account has been lost.

Pre-requisites:

- Make sure UCP is not using LDAP for its user accounts (see **Admin Settings** -> **Authentication and Authorization**).

6.1 Recovering admin passwords

1. SSH into the `ucp-manager-0` node.
2. If we do a quick `docker container ls` we can see all our UCP containers; the container we want to access is `ucp-auth-api`.
3. Run the following command:

```
$ authapiid=$(docker container ls | grep ucp-auth-api | cut -d' ' -f1)
$ docker container exec -it ${authapiid} enzi \
    "$(docker container inspect --format '{{ index .Args 0 }}' ${authapiid})" \
    passwd --interactive
```

4. You will be prompted for the username. Specify `admin`, then choose a new password. Finally, check that you can log in to the UCP UI with your new credentials.

6.2 Conclusion

Note that node access to a UCP controller and the ability to `exec`-attach as per the above command is all that is required to gain admin control to UCP; make sure access to these nodes is appropriately restricted.

7 Orchestrating Applications

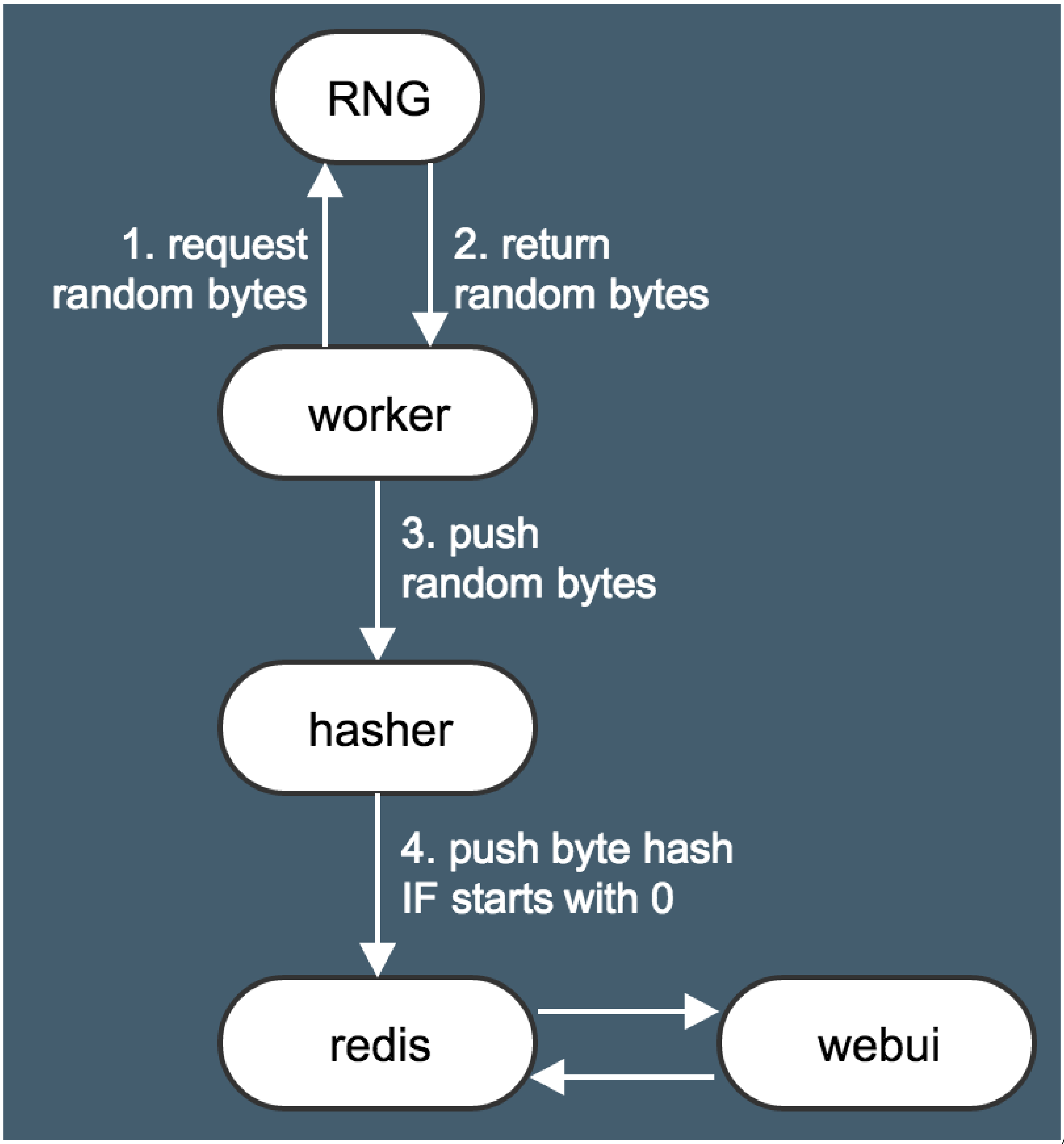
In this exercise you will learn how to deploy an application that consists of multiple services on both Swarm and Kubernetes.

Pre-requisites:

- UCP installed with 2 worker nodes (or more).
- No existing services running. Before beginning this exercise, delete all your previous services that have been deployed, in order to avoid port conflicts.

7.1 Background info

The application that we are going to deploy is a toy Dockercoin miner that hashes random numbers and logs any hash beginning with a 0 as a 'Dockercoin', and reports real-time results in a web UI. The app consists of five services, that interact like this:



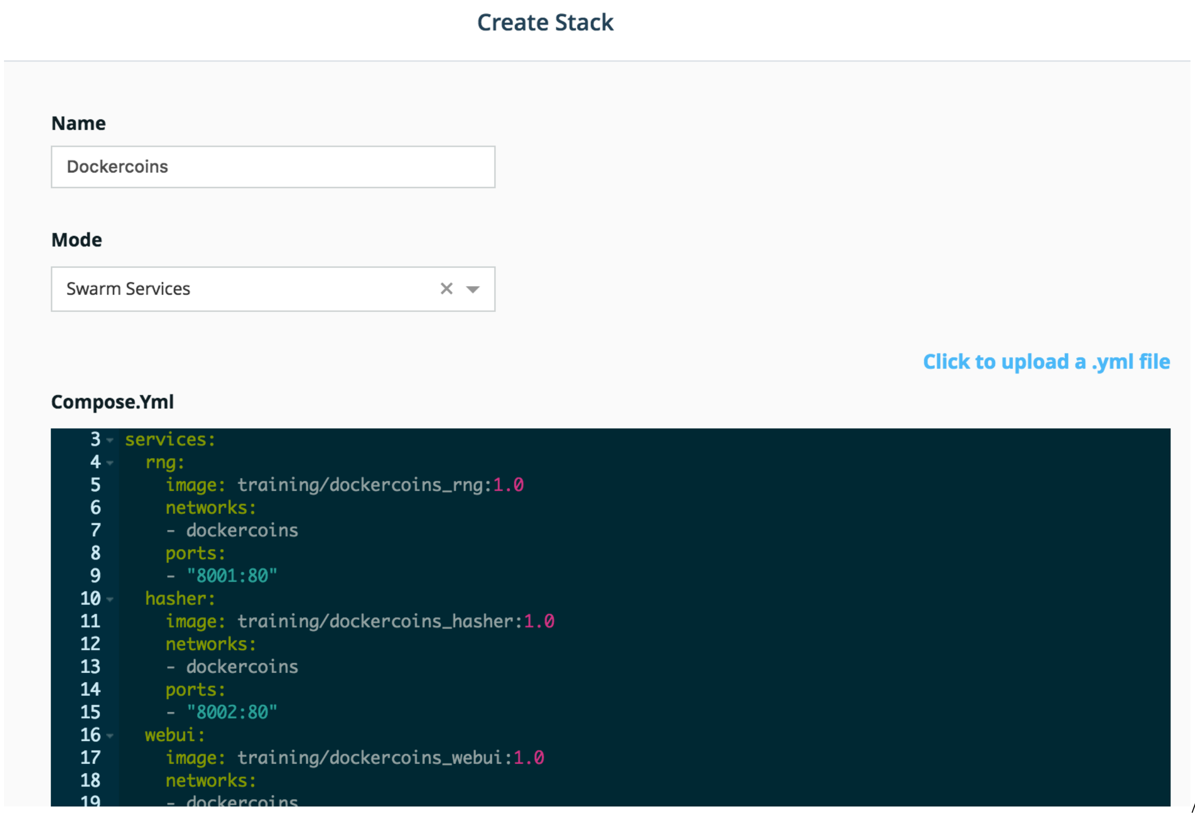
7.2 Deploying to Swarm

The Swarm services that make up our app can be defined in a Compose file and then deployed in UCP via the web UI.

1. In the UCP web UI, navigate **Shared Resources** -> **Stacks** -> **Create Stack**.

2. On the Deploy screen, specify 'Dockercoins' in the **Name** field, 'Swarm Services' in the **Mode** dropdown, and paste the following Compose file into the **Compose.yml** field:

```
version: "3.1"

services:
  rng:
    image: training/dockercoins_rng:1.0
    networks:
    - dockercoins
    ports:
    - "8001:80"
  hasher:
    image: training/dockercoins_hasher:1.0
    networks:
    - dockercoins
    ports:
    - "8002:80"
  webui:
    image: training/dockercoins_webui:1.0
    networks:
    - dockercoins
    ports:
    - "8000:80"
  redis:
    image: redis
    networks:
    - dockercoins
  worker:
    image: training/dockercoins_worker:1.0
    networks:
    - dockercoins

networks:
  dockercoins:
```



Name: Dockercoins

Mode: Swarm Services

Compose.Yml

```
services:
  rng:
    image: training/dockercoins_rng:1.0
    networks:
    - dockercoins
    ports:
    - "8001:80"
  hasher:
    image: training/dockercoins_hasher:1.0
    networks:
    - dockercoins
    ports:
    - "8002:80"
  webui:
    image: training/dockercoins_webui:1.0
    networks:
    - dockercoins
```

3. Click **Create**, wait for the services to deploy and then click **Done**. Dockercoins should now be listed on the **Stacks** page.
4. Navigate **Shared Resources** -> **Stacks** -> **Dockercoins** -> **Inspect Resource** -> **Services**. A list of all the services composing Dockercoins is produced (if this is the first time you've clicked on the 'Inspect Resource' menu, you'll get a popup explaining *filters*; you've just applied a filter that shows only objects associated with the current stack, Dockercoins). Wait for all services to report 1/1 in their Status column.
5. Open a browser tab and see Dockercoins in action by specifying the public IP of any of UCP node, followed by port 8000.

7.3 Scaling the application

We want more Dockercoins! To increase mining speed, we will scale the `worker` service so that we will have more containers to handle the workload.

1. Scale the `worker` service to run 2 containers. Find the option to do this by clicking on the service in UCP, and exploring the right sidebar.
2. Return to the web UI after the new worker container has spun up. Your mining speed should have doubled.
3. Try re-deploying the Dockercoins application exactly as above, with the following modification to the Compose file:

```
worker:
  image: training/dockercoins_worker:1.0
  networks:
  - dockercoins
  deploy:
    replicas: 2
```

   The end result is the same, but the scale of the `worker` service is captured right in the application definition, rather than having to manage it through UCP after deployment. Also notice that re-deploying an existing stack *updates* the stack.

4. Finally, remove your Dockercoins filter, and remove the Dockercoins application via the **Stacks** menu in UCP.

7.4 Deploying to Kubernetes

1. Nodes are configured to run processes orchestrated by Swarm, Kubernetes, or both. Navigate **Shared Resources** -> **Nodes** -> **ucp-node-0** -> **Configure** -> **Details**, choose **Kubernetes** under 'Set Orchestrator Type', and click **Save**.
2. Do the same for **ucp-node-1**.
3. Set up your stack similarly to how you did above when first deploying to Swarm, but this time, select **Kubernetes Workloads** in the Mode dropdown. Another dropdown will appear asking you to choose a Kubernetes namespace; we'll learn more about how to use these later, but for now, choose **default**. Finally, use this slightly modified Compose file:

```
version: "3.1"

services:
  rng:
    image: training/dockercoins_rng:1.0
    networks:
    - dockercoins
    ports:
    - "32769:80"
  hasher:
    image: training/dockercoins_hasher:1.0
    networks:
    - dockercoins
    ports:
    - "32770:80"
  webui:
    image: training/dockercoins_webui:1.0
    networks:
    - dockercoins
    ports:
    - "32768:80"
  redis:
    image: redis
    networks:
    - dockercoins
  worker:
    image: training/dockercoins_worker:1.0
    networks:
    - dockercoins

networks:
  dockercoins:
```

   You'll notice this is exactly the same as the Compose file we used for deploying Dockercoins above, except now we're exposing ports >= 32768, above Kubernetes' reserved range for `nodePort` services. Once configured, click **Create**, and you'll be sent back to the UCP dashboard.

4. Navigate **Kubernetes** -> **Pods**. One pod has been created for each object under the `services` key in your compose.yml.
5. Navigate **Kubernetes** -> **Controllers**. A ReplicaSet has been created for each microservice to perform keep-alive on the pods we saw in the last step, and a Deployment has been created to manage things like rolling

updates and rollback.

6. Navigate **Kubernetes** -> **Load Balancers**. A Kube `ClusterIP` service has been created for every service, to make `ReplicaSets` reachable internally to the cluster by DNS resolution of the service name defined in the compose.yml, exactly analogously to how service name resolution behaved in Swarm.
7. Also in the Load Balancers list, notice that all the microservices that exposed a public port have a `*-published` service, like `webui-published` and `rng-published`. These are Kubernetes `LoadBalancer` services that make `ReplicaSets` reachable from the external network, similar to Swarm's L4 mesh net.
8. Click on the `webui-published` Kube service. A url of the form `<UCP FQDN>:<Node Port>` appears in the SPEC section of the right sidebar; navigate to that address to see Dockercoins up and running.
9. Clean up your Kubernetes deployment by deleting your Dockercoins stack, and resetting your worker nodes to 'Swarm' orchestration.

7.5 Conclusion

In this exercise, you deployed an entire application once as Swarm services, and again as Kubernetes workloads, from a single compose file. Capturing application configuration in compose files is an important best practice for application reproducibility and versioning; the compose file provides a single source of truth that can be distributed via your existing version control strategy, and easily handed off between developers and operations teams.

8 Combining Collections and Kubernetes Namespaces

8.1 Creating Collections

1. Login to UCP as `admin`.
2. Navigate to **Shared Resources** –> **Collections**.
3. On the collection **Shared** click **View Children**.
4. Click **Create Collection** and enter **Development** as **Collection Name**. Then click **Create**.
5. Create another (sibling) collection called **Production**.

8.2 Associating Worker Nodes with Collections

We want to segregate our worker nodes into the two collections we created above.

1. Navigate to **Shared Resources** –> **Nodes**.
2. Select node `ucp-node-0` and in the details pane select **Configure** –> **Details**.
3. Change the value of the label `com.docker.ucp.access.label` to `/Shared/Development` and click **Save**.
4. Move node `ucp-node-1` accordingly to `/Shared/Production`.

8.3 Creating Kubernetes Namespaces

It is a good practice in Kubernetes to segregate applications by namespaces.

1. Login to UCP as `admin`.
2. Navigate to **Kubernetes** –> **Namespaces** and click **Create**.
3. In the **Object YAML** field enter this content to create a `development` namespace:

```
apiVersion: v1
kind: Namespace
metadata:
    name: development
```

4. Click **Create** to have UCP create the namespace.

5. Create another namespace called `production`.

8.4 Preparing Nodes for Kubernetes Workload

In this exercise we want to configure our worker nodes such as they can execute Kubernetes workload.

1. In UCP still logged in as `admin` navigate to **Shared Resources** –> **Nodes**.
2. Select node `ucp-node-0` and the in the details pane select **Configure** –> **Details**.
3. In the dialog under **Role** select **Mixed** as **Orchestrator Type** and click **Save**. This allows us to use this worker node for payload managed by SwarmKit as well as by Kubernetes.

   Note: It is highly recommended that you do not set your nodes to **mixed** mode in production!

4. Repeat the same for node `ucp-node-1`.

8.5 Associating Nodes with Namespaces

1. Navigate to **Kubernetes** –> **Namespaces**. And select the `development` namespace.
2. In the details pane select **Actions** –> **Link Nodes in Collection**.
3. Under **Choose Collection** navigate to the **/Share/Development** collection and select it.
4. Click **Confirm**. Now the Kubernetes namespace `development` is linked with the worker nodes in collection `/Shared/Development`.
5. Similarly link the namespace `production` with the nodes in the `/Shared/Production` collection.

8.6 Creating a Deployment in the "development" namespace

1. Navigate to **Kubernetes** –> **Namespaces** and select the `development` namespace.
2. In the details pane select **Action** –> **Set Context**.

   This will set the context of UCP to the `development` namespace and all resources will be filtered by this namespace.

3. Navigate to **Kubernetes** and click **+ Create**.
4. Under **Namespace** select `development`.
5. In the **Object YAML** field enter this content to create a **deployment**:

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

6. Click **Create**.

   This will create a Kubernetes `Deployment` object as well as a corresponding `ReplicaSet` managing 3 `Pods` containing each an instance of Nginx. All these objects will live in the `development` namespace.

7. Navigate to **Kubernetes Controllers**. You should see the `nginx-deployment` we just created above.
8. Navigate to **Kubernetes Pods** and make sure they are all deployed on node `ucp-node-0` (this is the (only) worker node associated with collection `/Shared/Development` and linked to the `development` namespace.).

8.7 Creating a Deployment in the “production” namespace

Repeat the previous exercise but this time for the namespace `production`. Use the following yaml that defines our production deployment object:

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: tomcat-prod-deployment
spec:
  selector:
    matchLabels:
      app: tomcat
  replicas: 5
  template:
    metadata:
      labels:
        app: tomcat
    spec:
      containers:
      - name: tomcat
        image: tomcat:9.0
        ports:
        - containerPort: 8080
```

8.8 Cleaning up

1. Set the context to namespace `development`.
2. Navigate to **Kubernetes** –> **Controllers**.
3. Select the listed deployment and delete it (**Actions** –> **Remove**).
4. Do the same with the production deployment.
5. Delete the two Kubernetes namespaces `development` and `production`.
6. Move the two worker nodes back into the `/Shared` collection.

8.9 Conclusion

In this exercise we have created multiple Kubernetes resources in their own custom namespaces from within UCP. We have associated worker nodes with different namespaces and as such made sure that the corresponding pods were only deployed to the designated worker node(s). In essence this corresponds to RBAC for worker nodes.

9 Service Mesh (1/3): Service Discovery

9.1 Setup

1. Set both your worker nodes to **Mixed** orchestration, since we'll be jumping back and forth between Swarm and Kubernetes in this exercise.

9.2 Exploring Docker Swarm Service Discovery

Swarm uses the DNS service built into the Docker engine of each node in the swarm. In this section we're going to see that DNS resolution is limited to a Docker network. Networks are like sandboxes and no cross-network name resolution or routing is possible.

You can execute the following exercise by using the Docker client (CLI) with a UCP client bundle to configure remote access from your laptop or your `elk` node to the Docker Swarm (for details check the previous exercise about UCP API & Client Bundle), or you can SSH directly into any of the manager nodes.

1. Create an overlay network `frontend`:

   ```
   $ docker network create \
       --driver overlay \
       --attachable \
       frontend
   ```

   we make this network attachable only to be able to run an individual container on this network that we will use to test service discovery later on.

2. Create an overlay network `backend`:

   ```
   $ docker network create --driver overlay backend
   ```

3. Create a service `web` attached to the network `frontend` using:

   ```
   $ docker service create --name web \
       --network frontend \
       --replicas 3 \
       --publish 3000:8000 \
       training/whoami:latest
   ```

4. Now create a service `worker` attached to the network `backend` using a very similar command:

   ```
   $ docker service create --name worker \
       --network backend \
       --replicas 3 \
       --publish 3100:8000 \
       training/whoami:latest
   ```

5. Finally let's create a container from which we will test that service discovery works. We will attach it to the network `frontend`:

   ```
   $ docker container run --rm -it \
       --network frontend \
       alpine:latest /bin/sh
   ```

6. Let's see if we can access the `web` service which is attached to the same Docker network:

   ```
   $ wget -qO - web:8000
   ```

   the answer should be something similar to this:

   ```
   I'm a93942dfd659
   ```

   This is the proof that service discovery works, for containers and services that share a Docker network.

7. Now let's try to access the `worker` service which is on a different network:

   ```
   $ wget -qO - worker:8000
   ```

   which will fail with a message like this:

   ```
   wget: bad address 'worker:8000'
   ```

   Question: why does it fail?

8. We need to attach the `worker` service to the `frontend` network to make it reachable; exit your test container and do so:

   ```
   $ docker service update --network-add frontend worker
   ```

9. Try to ping the `worker` service from another test container on the `frontend` network:

   ```
   $ docker container run --rm -it \
       --network frontend \
       alpine:latest /bin/sh
   $ wget -qO - worker:8000
   ```

   this time it should work and respond with a message like this:

   ```
   I'm 25120259aac4
   ```

   Conclusion: Service discovery ONLY works for services that are attached to the same network. Docker networks are perfectly firewalled from each other.

9.3 Exploring Kubernetes Service Discovery

Kubernetes uses either environment variables or (highly recommended) the DNS service cluster add-on to provide service discovery. Contrary to Swarm, Kubernetes defines one flat network that spans the whole cluster. Each pod can reach each other pod in the cluster and each cluster node can reach each pod and vice versa. Namespaces in Kubernetes provide a logical boundary but do not prevent pods from reaching each other across namespaces. If we want to sandbox services then we need to use network policies (which is outside of the scope of this exercise).

You can execute the following exercise from within the UI of UCP or alternatively you can use `kubectl` with a client bundle to configure remote access from your laptop or from your `elk` node to the Kubernetes cluster.

1. Create a namespace `frontend` by using this YAML:

   ```
   apiVersion: v1
   kind: Namespace
   metadata:
     name: frontend
   ```

2. Create a namespace `backend` with this YAML:

   ```
   apiVersion: v1
   kind: Namespace
   metadata:
     name: backend
   ```

3. Create a service `web` in the namespace `frontend` using:

   ```
   apiVersion: apps/v1beta2
   kind: Deployment
   metadata:
     name: web-deployment
     namespace: frontend
     labels:
       app: web
   spec:
   ```

```
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: web
        image: training/whoami:latest
        ports:
        - containerPort: 8000
---
kind: Service
apiVersion: v1
metadata:
  name: web
  namespace: frontend
spec:
  selector:
    app: web
  ports:
  - port: 3000
    targetPort: 8000
```

Note how we have a deployment `web-deployment` and a service `web` in the same YAML. The service is of type `ClusterIP` which is good enough for us since we only want to access the service from within the cluster.

4. Now create a service `worker` in the namespace `backend` using a very similar YAML:

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: worker-deployment
  namespace: backend
  labels:
    app: worker
spec:
  replicas: 3
  selector:
    matchLabels:
      app: worker
  template:
    metadata:
      labels:
        app: worker
    spec:
      containers:
      - name: worker
        image: training/whoami:latest
        ports:
        - containerPort: 8000
---
kind: Service
apiVersion: v1
```

```
metadata:
  name: worker
  namespace: backend
spec:
  selector:
    app: worker
  ports:
  - port: 3100
    targetPort: 8000
```

5. Finally let's create a pod `jumper` from which we will test that service discovery works. We will deploy it into namespace `frontend`:

   ```
   apiVersion: apps/v1beta2
   kind: Deployment
   metadata:
     name: jumper-deployment
     namespace: frontend
     labels:
       app: jumper
   spec:
     replicas: 1
     selector:
       matchLabels:
         app: jumper
     template:
       metadata:
         labels:
           app: jumper
       spec:
         containers:
         - name: jumper
           image: nginx:alpine
   ```

   in this case we don't need a service for the test pod since we do not want to access it from elsewhere, and only require one replica for our purposes.

6. Locate the `nginx:alpine` container in the `jumper` pod and open a terminal session to it. Use `/bin/sh` as shell command.

7. Let's see if we can access the `web` service which is in the same namespace:

   ```
   $ wget -qO - web:3000
   ```

   the answer should be something similar to this:

   ```
   I'm web-deployment-76db5b545b-j85kj
   ```

   This is the proof that service discovery works, at least inside the confines of the namespace.

8. To have service discovery work across namespaces we need to append the namespace name to the service name:

   ```
   $ wget -qO - worker.backend:3100
   ```

   which in my case returns:

   ```
   I'm worker-deployment-645cd496fb-cqzpj
   ```

9.4 Conclusion

In this exercise, you explored how Swarm and Kubernetes approach the problem of service discovery for containerized, distributed workloads; both rely heavily on their respective control planes (gossip for Swarm and BGP routing for Calico) to distribute host information, and DNS lookup to map convenient, stable service names onto fungible service VIPs (swarm) or ClusterIP objects (Kube).

10 Service Mesh (2/3): Routing

10.1 Using Swarm's Layer 4 Routing

Prerequisites: You have the `web` and `worker` services running in the swarm from the previous exercise.

1. Open a terminal and try to access the `web` service via its public port (remember, it is 3000):

   ```
   $ curl -4 http://<public IP>:3000
   ```

   where `<public IP>` is the public IP address of any node of your Docker Swarm.

   The response should be similar to this:

   ```
   I'm a93942dfd659
   ```

2. Now try the same with the public IP address of any other of the remaining Swarm nodes. It should work the same way for all nodes.
3. Try the same for the `worker` service. We gave it port 3100.
4. Let's scale down the `worker` service to 1 instance:

   ```
   $ docker service update --replicas 1 worker
   ```

5. Find out on which node the remaining replica is located:

   ```
   $ docker service ps worker
   ```

6. Try to access the worker service from a node which does NOT have an instance of the `worker` service running:

   ```
   $ curl -4 http://<IP address>:3100
   ```

   where `<IP address>` is the public IP address of a node NOT running a `worker` instance. It should be successful; swarm layer 4 routing is provided by its **Routing Mesh**, which provides service reachability across the Swarm, regardless of where service tasks have been scheduled.

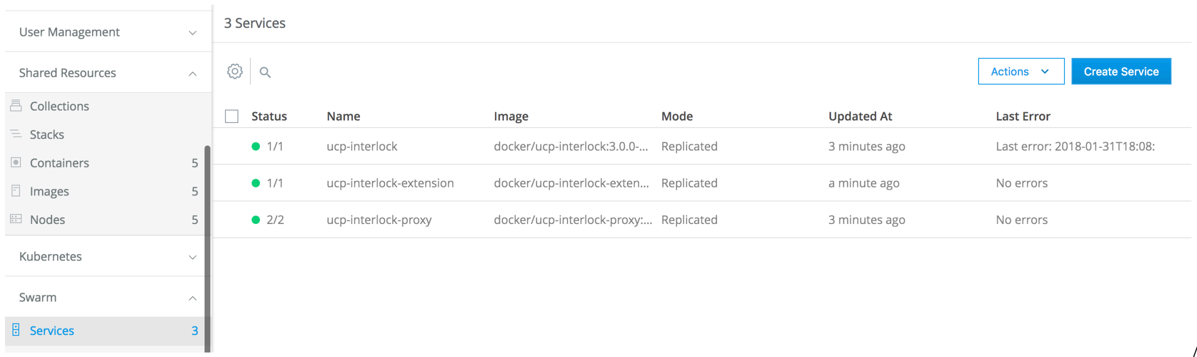
10.2 Using Swarm's Layer 7 Routing

We're going to use **Interlock 2** to enable layer 7 routing in our Docker Swarm.

10.2.1 Enabling Interlock

1. From a UCP admin account, navigate **Admin Settings** -> **Layer 7 Routing**.
2. Leave the default ports alone (80 for **HTTP** and 8443 for **HTTPS**), and tick the checkbox which says **Enable Layer 7 Routing**.
3. Click **Save**, at the bottom.
4. Navigate to the **Networks** tab in the Swarm section of the left navigation bar. There should be a network named `ucp-interlock`.
5. Navigate to the left navigation bar, click into the **Swarm** section, and then **Services** to display all your services. There should be one for `Interlock`, one for `extension` and one for `proxy`:

Note: If you don't see these services, click the gear icon (in the service view page) and check **Show system resources**.



3 Services

Status	Name	Image	Mode	Updated At	Last Error
● 1/1	ucp-interlock	docker/ucp-interlock:3.0.0-...	Replicated	3 minutes ago	Last error: 2018-01-31T18:08:
● 1/1	ucp-interlock-extension	docker/ucp-interlock-exten...	Replicated	a minute ago	No errors
● 2/2	ucp-interlock-proxy	docker/ucp-interlock-proxy:...	Replicated	3 minutes ago	No errors

10.2.2 Deploying Services with Interlock

1. On the **ucp-manager-0** node, create an overlay network just to keep the traffic to Interlock isolated and secure:

```
$ docker network create -d overlay demo
```

2. Create the service and when publishing it, make sure to use the two labels `com.docker.lb.hosts` and `com.docker.lb.port`:

```
$ docker service create --name demo \
    --network demo \
    --label com.docker.lb.hosts=demo.local \
    --label com.docker.lb.port=8000 \
    training/whoami:latest
```

   The `com.docker.lb.hosts` label tells Interlock the URL where this service will be available and `com.docker.lb.port` tells the proxy service to which node port the request should be forwarded, where **Routing Mesh** with its L4 routing then forwards the request to the appropriate container.

3. Make sure the service is up and running:

```
$ curl -4 -H "Host: demo.local" http://localhost
```

   The output should resemble this:

```
I'm 2c206a3a4741
```

   The request has been routed to the container based on the `Host` value in the request header.

4. Scale the service to four replicas:

```
$ docker service update --replicas 4 demo
```

   List the demo tasks by running `docker service ps demo` and all the different replicas should be scheduled across the nodes. This can also be accessed through the UCP UI by clicking on the specific service then scrolling down to the tasks section on the right navigation bar and clicking to view full details.

5. To see that traffic is load balanced across all our new service replicas:

```
$ for N in `seq 1 10`; do curl -4 -H "Host: demo.local" http://localhost; done;
```

   Which produces something akin to:

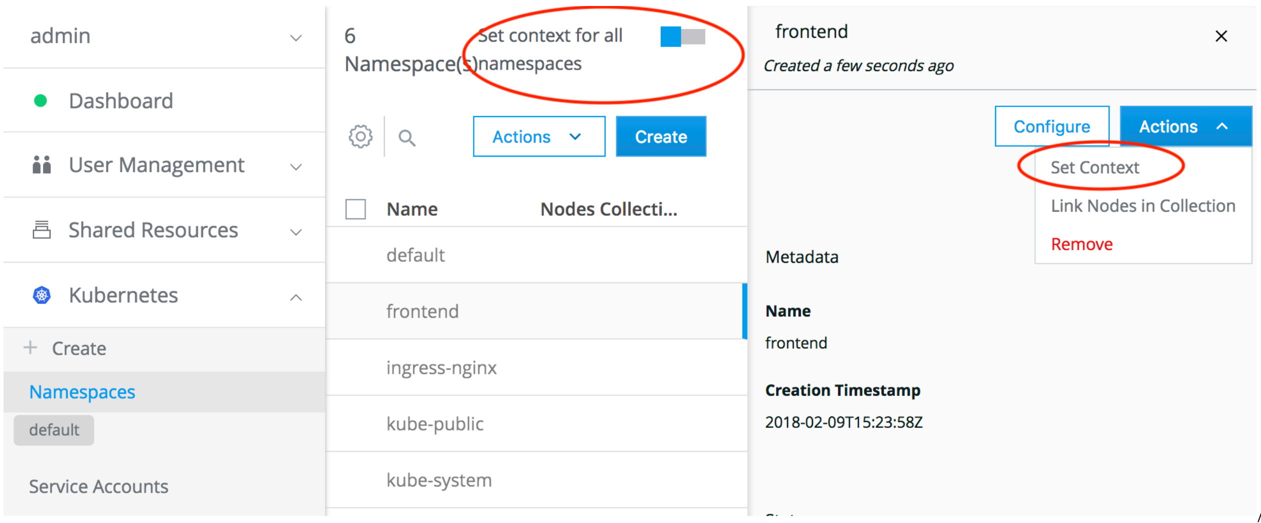
```
I'm 2c206a3a4741
I'm 2c206a3a4741
I'm cf659f3e6093
I'm cf659f3e6093
I'm 89598836e0d1
```

```
I'm 89598836e0d1
I'm d83e3b1acea6
I'm d83e3b1acea6
I'm 2c206a3a4741
I'm 2c206a3a4741
```

As you can see, the instance IDs are being displayed twice before proceeding to the next one. This is because the L7 load balancer round robins across both the replicas of the proxy service, and each of those round robins across the four replicas of the demo service.

10.3 Using Kubernetes Layer 4 Routing

Note: make sure that in the UCP UI you have set the correct context for Kubernetes to be able to see your objects. You can either select the desired namespace (e.g. `frontend`) as current context or select the option **Set context for all namespaces**



1. Delete the old version of the `web` service. In the UCP UI under **Kubernetes –> Load Balancers** locate the `web` service and delete it.

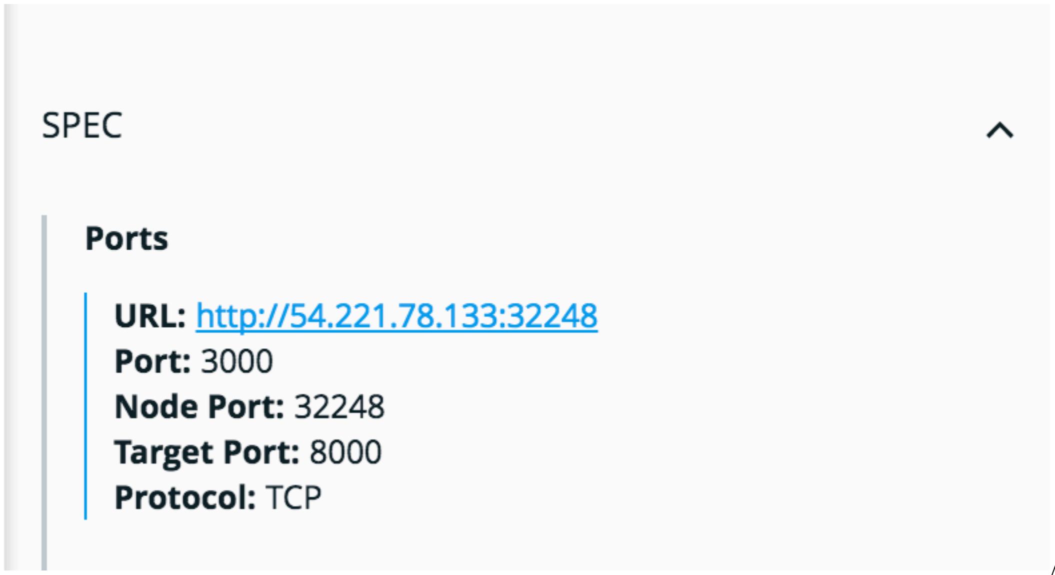
   Note, you can also do this from the CLI if you've set up a client bundle (optional):

   ```
   $ kubectl delete service/web
   ```

2. Now create the new version of the service `web` using:

   ```
   kind: Service
   apiVersion: v1
   metadata:
     name: web
     namespace: frontend
   spec:
     type: NodePort
     selector:
       app: web
     ports:
     - port: 3000
       targetPort: 8000
   ```

3. In the UCP UI locate the new `web` service and select it to display the details of this service.
4. Find the **URL** under **SPEC –> Ports** and click on it:

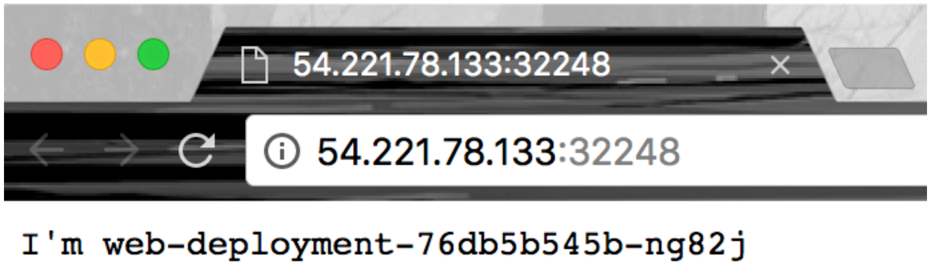


SPEC

Ports

URL: http://54.221.78.133:32248
Port: 3000
Node Port: 32248
Target Port: 8000
Protocol: TCP

Your browser should display something like this:



```
54.221.78.133:32248
I'm web-deployment-76db5b545b-ng82j
```

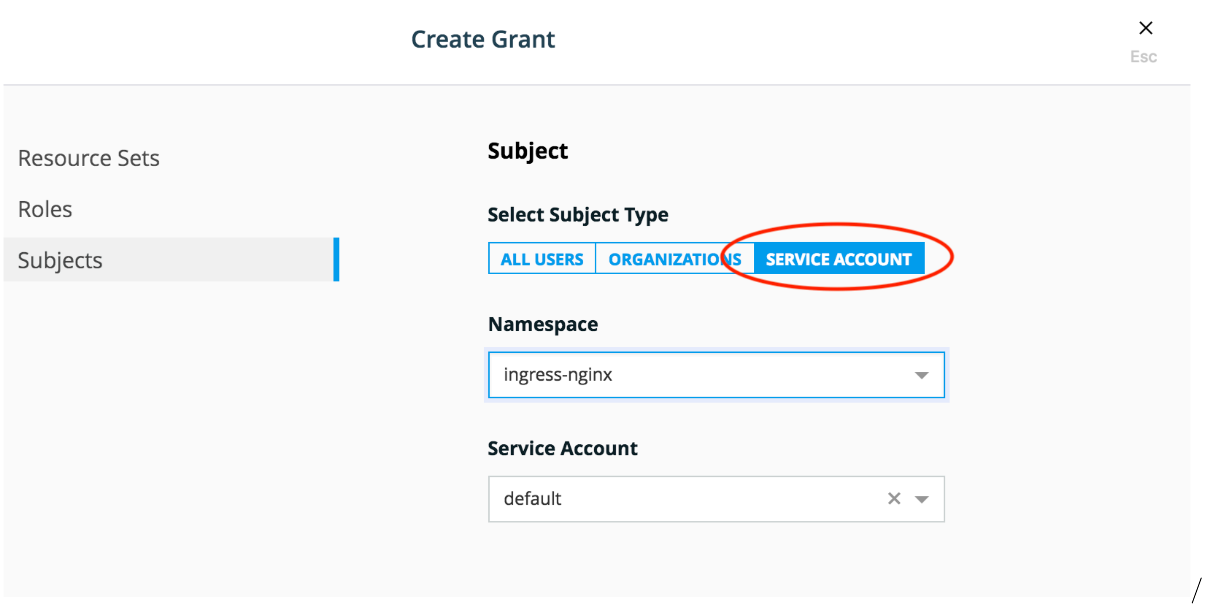
What just happened? As shown in the first image, Kubernetes has published the `web` service on port 32248 on every node of the cluster. This port is now a synonym for the `web` service. An incoming request on any cluster node to this port will be resolved to the **VIP** of the service `web`. From here on layer 3 routing is used to get the request to its final destination.

10.4 Configuring Kubernetes Layer 7 Routing in UCP

1. Login as `admin` to your UCP UI.
2. Create a Kubernetes namespace. In the Kubernetes section click **+ Create** and paste this yaml into the **Object YAML** field:

   ```
   apiVersion: v1
   kind: Namespace
   metadata:
     name: ingress-nginx
   ```

   and click **Create**.
3. Create a new grant that gives the **ServiceAccount** restricted access to all Kubernetes namespaces:
   1. Select **User Management** –> **Grants** and click **Create**
   2. Under **Subject** select **SERVICE ACCOUNT** and then select Namespace **ingress-nginx** and Service Account **default**.
   3. Under **Role** select **Restricted Control**.
   4. Under **Resource Set** select **NAMESPACES** and then turn the switch **Apply grant to all existing and new namespaces** on.
   5. Click **Create**.



Create Grant

Resource Sets

Roles

Subjects

Subject

Select Subject Type

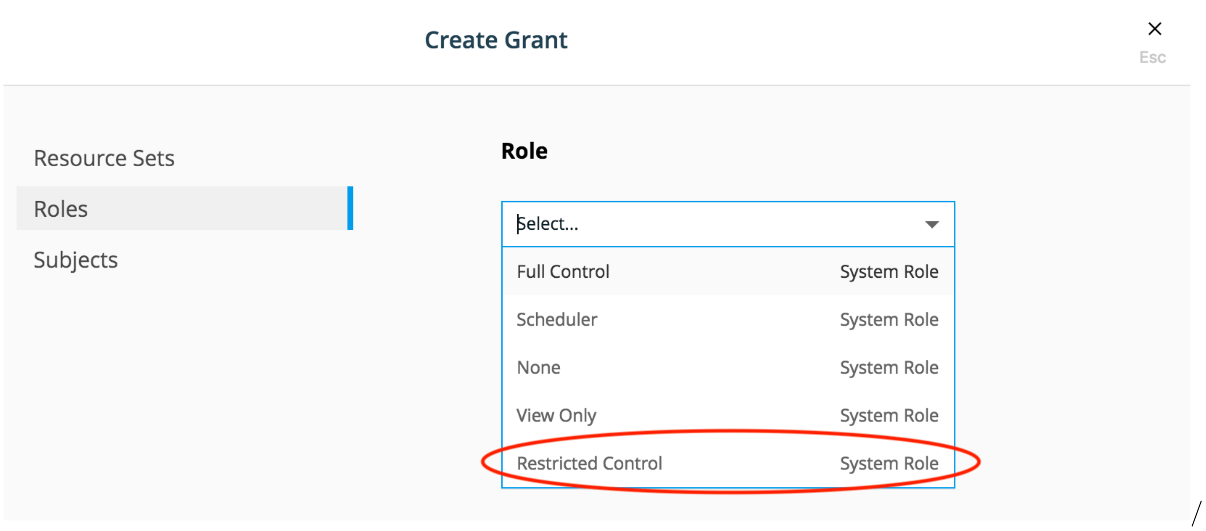
ALL USERS | ORGANIZATIONS | [✓ SERVICE ACCOUNT]

Namespace

ingress-nginx

Service Account

default



Create Grant

Resource Sets

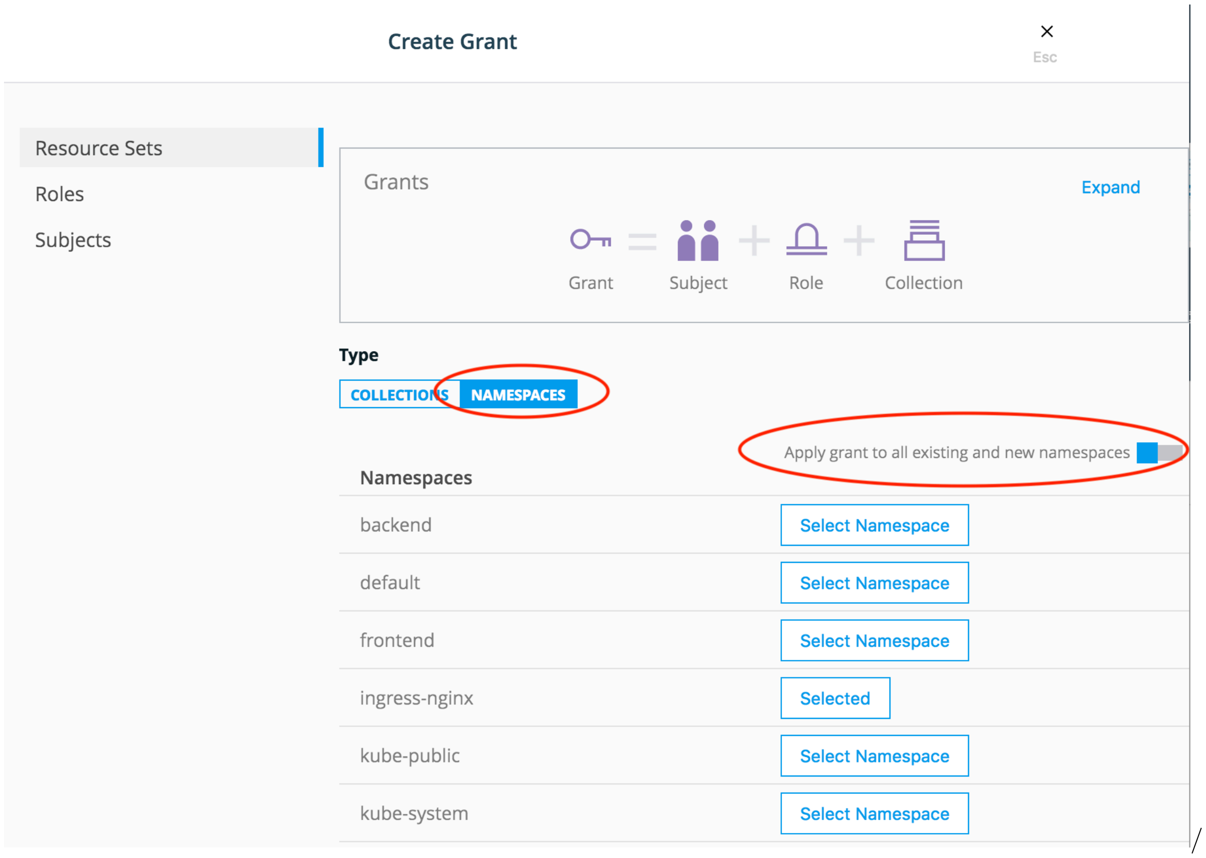
Roles

Subjects

Role

Select...

Full Control	System Role
Scheduler	System Role
None	System Role
View Only	System Role
[✓ Restricted Control]	System Role



Type: COLLECTIONS [✓ NAMESPACES]

Apply grant to all existing and new namespaces

Namespaces	
backend	Select Namespace
default	Select Namespace
frontend	Select Namespace
ingress-nginx	Selected
kube-public	Select Namespace
kube-system	Select Namespace

This grant is needed so that the ingress controller has the necessary access rights. Otherwise it will not start and report access denied error messages.

4. In the Kubernetes section click **+ Create** again and paste this long YAML into the **Object YAML** field:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: default-http-backend
  labels:
    app: default-http-backend
  namespace: ingress-nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: default-http-backend
    spec:
      terminationGracePeriodSeconds: 60
      containers:
      - name: default-http-backend
        # Any image is permissable as long as:
        # 1. It serves a 404 page at /
        # 2. It serves 200 on a /healthz endpoint
        image: gcr.io/google_containers/defaultbackend:1.4
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 30
          timeoutSeconds: 5
        ports:
        - containerPort: 8080
        resources:
          limits:
            cpu: 10m
            memory: 20Mi
          requests:
            cpu: 10m
            memory: 20Mi
---
apiVersion: v1
kind: Service
metadata:
  name: default-http-backend
  namespace: ingress-nginx
  labels:
    app: default-http-backend
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: default-http-backend
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
  namespace: ingress-nginx
  labels:
    app: ingress-nginx
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: tcp-services
  namespace: ingress-nginx
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: udp-services
  namespace: ingress-nginx
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ingress-controller
  namespace: ingress-nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ingress-nginx
  template:
    metadata:
      labels:
        app: ingress-nginx
      annotations:
        prometheus.io/port: '10254'
        prometheus.io/scrape: 'true'
    spec:
      # Beta2 only: Uncomment the following line
      # serviceAccountName: nginx-service-account
      initContainers:
      - command:
        - sh
        - -c
        - sysctl -w net.core.somaxconn=32768; sysctl -w net.ipv4.ip_local_port_range="1024 65535"
        image: alpine:3.6
        imagePullPolicy: IfNotPresent
        name: sysctl
        securityContext:
          privileged: true
      containers:
        - name: nginx-ingress-controller
          image: quay.io/kubernetes-ingress-controller/nginx-ingress-controller:0.10.2
          args:
            - /nginx-ingress-controller
            - --default-backend-service=$(POD_NAMESPACE)/default-http-backend
            - --configmap=$(POD_NAMESPACE)/nginx-configuration
            - --tcp-services-configmap=$(POD_NAMESPACE)/tcp-services
            - --udp-services-configmap=$(POD_NAMESPACE)/udp-services
            - --annotations-prefix=nginx.ingress.kubernetes.io
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: POD_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
          ports:
          - name: http
            containerPort: 80
          - name: https
            containerPort: 443
          livenessProbe:
            failureThreshold: 3
            httpGet:
              path: /healthz
              port: 10254
              scheme: HTTP
            initialDelaySeconds: 10
            periodSeconds: 10
            successThreshold: 1
            timeoutSeconds: 1
          readinessProbe:
            failureThreshold: 3
            httpGet:
              path: /healthz
              port: 10254
              scheme: HTTP
            periodSeconds: 10
            successThreshold: 1
            timeoutSeconds: 1
---
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
  namespace: ingress-nginx
spec:
  type: NodePort
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
  - name: https
    port: 443
    targetPort: 443
    protocol: TCP
  selector:
    app: ingress-nginx
```

This YAML defines all the necessary Kubernetes resources to enable layer 7 or application level routing using Nginx as a (reverse) proxy.

5. Click **Create**.

   > **Note**: If you specify a namespace for all objects in the YAML file, the namespace dropdown menu will be disabled since it's not needed anymore.

6. To test that our IngressController is working we can first get the description of our IngressController service:

   You can do this either through the UCP interface (**Kubernetes** -> **Load Balancer** -> `ingress-nginx`) or by using the CLI:

```
$ kubectl describe -n ingress-nginx service ingress-nginx
```

which in my case give this result:

```
Name:                     ingress-nginx
Namespace:                ingress-nginx
Labels:                   <none>
Annotations:              <none>
Selector:                 app=ingress-nginx
Type:                     NodePort
IP:                       10.96.76.241
Port:                     http  80/TCP
TargetPort:               80/TCP
NodePort:                 http  31812/TCP
Endpoints:                192.168.85.68:80
Port:                     https  443/TCP
TargetPort:               443/TCP
NodePort:                 https  32156/TCP
Endpoints:                192.168.85.68:443
```

```
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

from there we see the port on which the service is listening, in my case 31812.

7. Now use `curl` to try to access the proxy:

   ```
   $ curl -I http://<IP address>:<port>/healthz
   ```

   where `<IP address>` is the public IP address of any Kubernetes cluster node and `<port>` is the above NodePort.

   The result should look similar to this:

   ```
   HTTP/1.1 200 OK
   Server: nginx/1.13.8
   Date: Fri, 09 Feb 2018 20:45:15 GMT
   Content-Type: text/html
   Content-Length: 0
   Connection: keep-alive
   Strict-Transport-Security: max-age=15724800; includeSubDomains;
   ```

   And evidently the request is handled by the Nginx proxy and the respose status is OK (200). We're all set and good to go.

8. We are now ready to define **Ingress** resources that defines routing rules for the two services. We need to create an Ingress resource for each namespace. Under Kubernetes click **Create** and paste this content into the Object YAML field:

   ```
   apiVersion: extensions/v1beta1
   kind: Ingress
   metadata:
     name: frontend-ingress
     namespace: frontend
     annotations:
       ingress.kubernetes.io/rewrite-target: /
   spec:
     rules:
     - host: example.com
       http:
         paths:
         - path: /web
           backend:
             serviceName: web
             servicePort: 3000
   ```

   and for the `backend` namespace create:

   ```
   apiVersion: extensions/v1beta1
   kind: Ingress
   metadata:
     name: backend-ingress
     namespace: backend
     annotations:
       ingress.kubernetes.io/rewrite-target: /
   spec:
     rules:
     - host: example.com
       http:
         paths:
   ```

```
      - path: /worker
        backend:
          serviceName: worker
          servicePort: 3100
```

and click create. This will define the route mapping:

- example.com/web —> service "web" on port 3000
- example.com/worker —> service "worker" on port 3100

and configure the IngressController with it.

> **Note:** For the curious mind: one can exec into the `ingress-nginx` container and analyze the Nginx configuration file `/etc/nginx/nginx.conf` for the routing information defined by the Ingress resources.

9. In your terminal use `curl` to test the routing:

```
$ curl -H "Host: example.com" <public IP>:<port>/web
I'm web-deployment-67f975fc67-dgf2p

$ curl -H "Host: example.com" <public IP>:<port>/worker
I'm worker-deployment-5995586c4c-8nbfq
```

where `<public IP>` is the public IP address of any cluster node and `<port>` is the NodePort of the Ingress-Controller service as discussed further up (in my example it was 31812).

10.5 Conclusion

In this exercise, we toured layer 4 and layer 7 routing in both Swarm and Kubernetes. Whether you encourage your development teams to build for one or the other depends strongly on how you want to manage your load balancers. If it is convenient for you as an operations team to reconfigure your external load balancers to point at a new service and port every time one is created, then the simplicity of layer 4 routing might be appropriate. But if you want a 'set and forget' external load balancer, then you can push the responsibility of routing traffic to one application or another into your application logic and orchestrator configuration using layer 7 routing.

11 Service Mesh (3/3): Internal Load Balancing

11.1 Experimenting with Swarm Load Balancing

1. Run a container `jumper` attached to the `frontend` network if you have not already done so:

```
$ docker container run --rm -it \
    --network frontend \
    alpine:latest /bin/sh
```

2. Execute the following script inside the session:

```
$ for N in `seq 1 10`; do wget -qO - web:8000; done;
```

the result should look similar to this:

```
I'm e814e1ac47cb
I'm 414c95ade173
I'm a93942dfd659
I'm e814e1ac47cb
I'm 414c95ade173
I'm a93942dfd659
I'm e814e1ac47cb
```

```
I'm 414c95ade173
I'm a93942dfd659
I'm e814e1ac47cb
```

we can see that the requests are load balanced. They are load balanced in a **round robin** way. This is the default behavior of Swarm's **Routing Mesh**, which uses the IPVS service for this purpose.

11.2 Experimenting with Kubernetes Load Balancing

1. Open a terminal session to the `nginx` container in the `jumper` pod if you have not already done so.
2. Execute the following script inside the session:

   ```
   $ for N in `seq 1 10`; do wget -qO - web:3000; done;
   ```

   the result should look similar to this:

   ```
   I'm web-deployment-76db5b545b-j85kj
   I'm web-deployment-76db5b545b-ng82j
   I'm web-deployment-76db5b545b-j85kj
   I'm web-deployment-76db5b545b-j85kj
   I'm web-deployment-76db5b545b-j85kj
   I'm web-deployment-76db5b545b-ng82j
   I'm web-deployment-76db5b545b-6hfmq
   I'm web-deployment-76db5b545b-ng82j
   I'm web-deployment-76db5b545b-ng82j
   I'm web-deployment-76db5b545b-j85kj
   ```

   we can see that the requests are load balanced. They are not load balanced in a round robin way but **randomly**. This is the expected behavior of a Kubernetes service of type `ClusterIP`.
3. Repeat the same for the `worker` service in the `backend` namespace and make sure it is also load balanced.
4. On any node in your cluster, dump your iptables with `sudo iptables-save`. Find the rules that set the random probabilities of routing traffic to any particular pod IP (hint: look for the flags `--mode random --probability ...`).

11.3 Cleanup

Make sure to delete all the stacks, Kubernetes controllers, services, and pods, as well as any Swarm services still remaining on your cluster.

11.4 Conclusion

In this exercise, we saw the fundamental behavior difference between layer 4 load balancing in Swarm and Kubernetes: Swarm balances in round robin fashion by resolving service VIPs to container IPs, while Kubernetes writes iptables chains that randomly route traffic to one of the backend pods.

Note of course that this is strictly *internal* load balancing, once traffic hits the internal VIP (swarm) or clusterIP (kube) of a service. This is entirely distinct from *external* load balancing of traffic coming from outside your cluster; this as always is entirely up to you, but as we learned in the routing exercise, can be made essentially trivial through correct use of your orchestrator's routing technology.

12 Logging

12.1 Installing ELK Stack

In this section we are going to install an ELK stack which serves as a log aggregator and offers a Web UI for the users to drill into the logging data.

1. SSH into the node called `elk` (remember the user for this node is `ubuntu`). Prepare the node to run `elasticsearch` by running the following commands:

   ```
   $ sudo sysctl -w vm.max_map_count=262144
   $ echo 'vm.max_map_count=262144' | sudo tee --append /etc/sysctl.conf
   ```

2. Clone a GitHub repository which contains the code necessary to define and run the ELK stack:

   ```
   $ git clone -b ee2.0 https://github.com/docker-training/elk-dee.git
   $ cd elk-dee
   ```

3. We want our logging stack to be completely independent from our cluster, so that if the cluster goes down, it doesn't take our logging infrastructure with it. As such, make the `elk` node its own swarm:

   ```
   $ docker swarm init
   ```

4. Finally we deploy the ELK stack:

   ```
   $ docker stack deploy -c elk-docker-compose.yml elk
   ```

   Double check that everything runs by using this command:

   ```
   $ watch docker service ls
   ```

   and wait until every service is running. You should see something like this:

   ```
   Every 2.0s: docker service ls

   ID            NAME               MODE        REPLICAS  IMAGE
   13sksh79tim2  elk_elasticsearch  replicated  1/1       elasticsearch:5.2
   cluyclmtaw48  elk_logstash       replicated  1/1       logstash:5.2-alpine
   whni0r8ddm8i  elk_kibana         replicated  1/1       kibana:5.2
   ```

   specifically note the column **Replicas** saying 1/1 for every service. This indicates that we're ready to roll.

12.2 Configuring all Swarm nodes

Now we need to configure the swarm so that every node in it reports all its logs to the ELK stack. We need to set up Docker daemon logging configuration to default to using `journald`. The reason we're doing this is to ensure that we can have a local copy of the logs to be able to use docker logs and docker service logs.

1. SSH into the `ucp-manager-0` node.

2. Add this (recommended) logging configuration to the file `/etc/docker/daemon.json`, alongside the storage driver configuration. The final file should look like:

   ```
   {
       "storage-driver": "overlay2",
       "log-driver": "journald",
       "log-level": "error",
       "log-opts": {
           "tag":"{{.ImageName}}/{{.Name}}/{{.ID}}"
       }
   }
   ```

3. Restart the Docker daemon:

```
$ sudo service docker restart
```

4. Repeat the above steps **every single** node of the swarm.

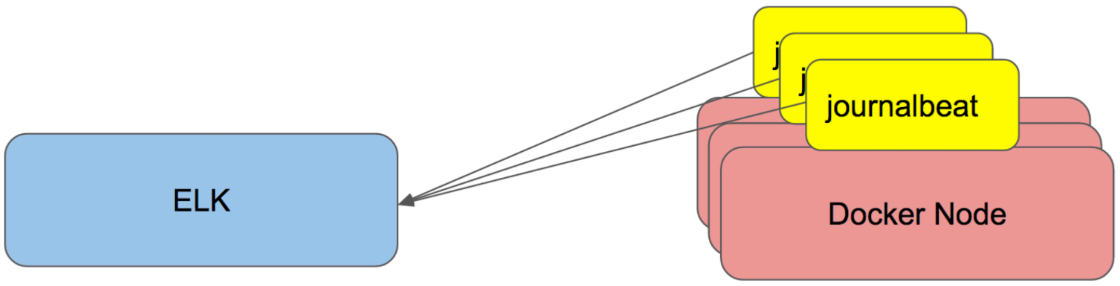
12.3 Stream all Docker logs to ELK

Now that we have configured all nodes to generate their respective logs using the `journald` driver we need to stream the data to our ELK stack.

1. SSH into one of the manager nodes, e.g. our `ucp-manager-0`.
2. Run the following commands on this swarm node:

```
$ git clone -b ee2.0 https://github.com/docker-training/elk-dee.git
$ cd elk-dee
$ export LOGSTASH_HOST=<ELK NODE PRIVATE IP>
$ docker stack deploy -c journalbeat-docker-compose.yml journalbeat
```

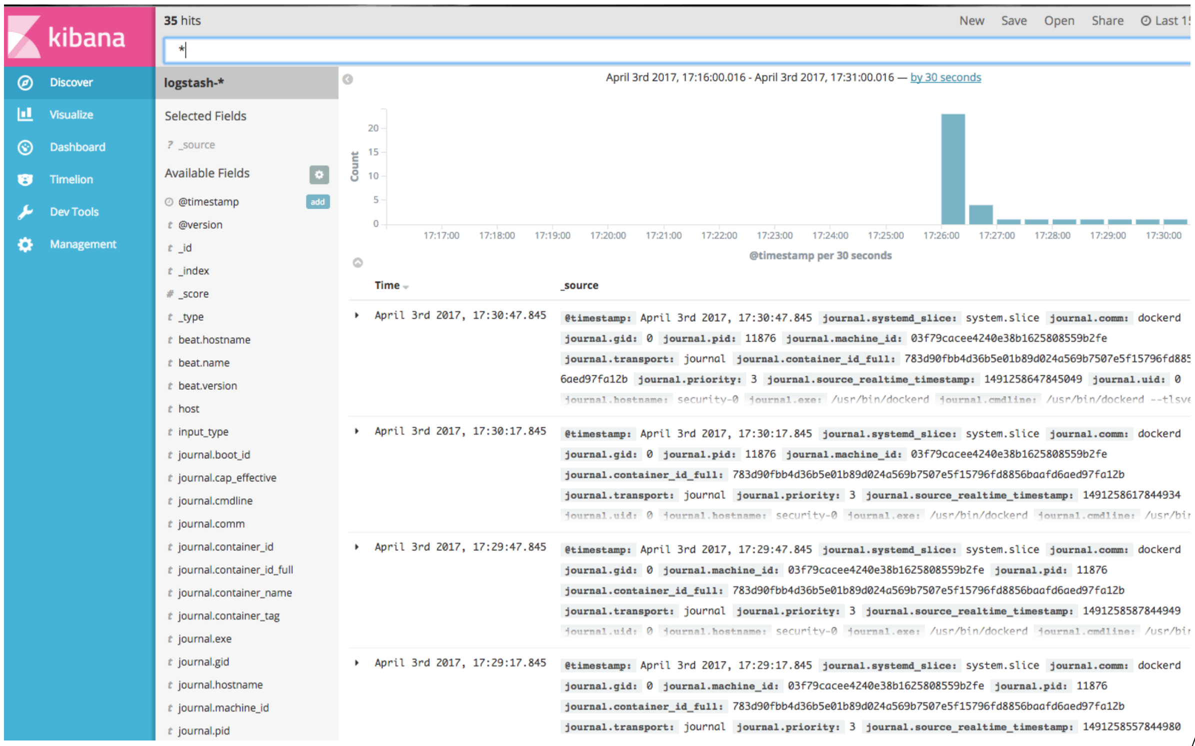
   where `<ELK NODE PRIVATE IP>` corresponds to the **internal** IP of the `elk` node. The result of this can be visualized as follows:



3. Make sure the logs are forwarded correctly to the ELK node by using the `journalctl` tool on the current node:

```
$ journalctl --since "2017-04-03 22:20"
```

   Replace the date/time stamp in the above command with the current date/time minus a few minutes. Watch out for events from `dockerd` and make sure there are no DNS resolution or connection errors visible in the log.
4. Open a browser at `http://<ELK_PUBLIC_IP_ADDRESS>:5601` to access Kibana. Upon first usage you should be asked to create an index for the events before you can proceed. Accept the defaults and click **Create**.
5. In Kibana navigate to **Discover** and view the list of captured events.

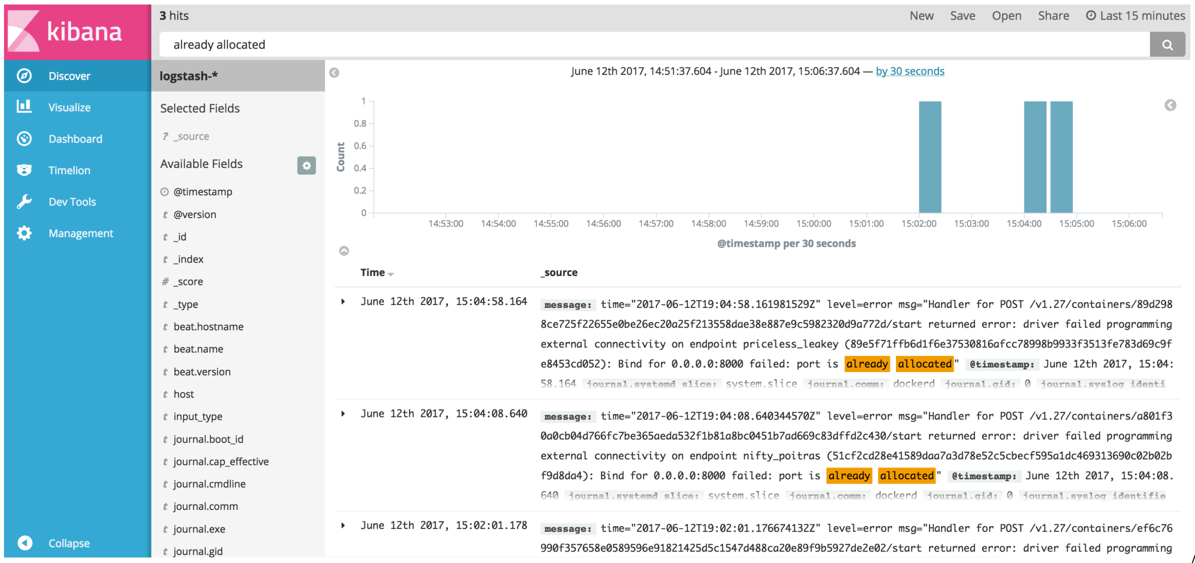


```
April 3rd 2017, 17:30:47.845   @timestamp: April 3rd 2017, 17:30:47.845 journal.systemd_slice: system.slice journal.comm: dockerd
                               journal.gid: 0 journal.pid: 11876 journal.machine_id: 03f79cacee4240e38b1625808559b2fe
                               journal.transport: journal journal.container_id_full: 783d90fbb4d36b5e01b89d024a569b7507e5f15796fd885
                               6aed97fa12b journal.priority: 3 journal.source_realtime_timestamp: 1491258647845049 journal.uid: 0
                               journal.hostname: security-0 journal.exe: /usr/bin/dockerd journal.cmdline: /usr/bin/dockerd --tlsve

April 3rd 2017, 17:30:17.845   @timestamp: April 3rd 2017, 17:30:17.845 journal.systemd_slice: system.slice journal.comm: dockerd
                               journal.gid: 0 journal.pid: 11876 journal.machine_id: 03f79cacee4240e38b1625808559b2fe
                               journal.container_id_full: 783d90fbb4d36b5e01b89d024a569b7507e5f15796fd8856baafd6aed97fa12b
                               journal.transport: journal journal.priority: 3 journal.source_realtime_timestamp: 1491258617844934
                               journal.uid: 0 journal.hostname: security-0 journal.exe: /usr/bin/dockerd journal.cmdline: /usr/bi

April 3rd 2017, 17:29:47.845   @timestamp: April 3rd 2017, 17:29:47.845 journal.systemd_slice: system.slice journal.comm: dockerd
                               journal.gid: 0 journal.machine_id: 03f79cacee4240e38b1625808559b2fe journal.pid: 11876
                               journal.container_id_full: 783d90fbb4d36b5e01b89d024a569b7507e5f15796fd8856baafd6aed97fa12b
                               journal.transport: journal journal.priority: 3 journal.source_realtime_timestamp: 1491258587844949
                               journal.uid: 0 journal.hostname: security-0 journal.exe: /usr/bin/dockerd journal.cmdline: /usr/bi

April 3rd 2017, 17:29:17.845   @timestamp: April 3rd 2017, 17:29:17.845 journal.systemd_slice: system.slice journal.comm: dockerd
                               journal.gid: 0 journal.machine_id: 03f79cacee4240e38b1625808559b2fe journal.pid: 11876
                               journal.container_id_full: 783d90fbb4d36b5e01b89d024a569b7507e5f15796fd8856baafd6aed97fa12b
                               journal.transport: journal journal.priority: 3 journal.source_realtime_timestamp: 1491258557844980
```

6. If you don't see any events yet, try to launch two containers listening on the same port on your UCP cluster, to generate an error:

```
$ docker container run -d -p 8000:80 nginx
$ docker container run -d -p 8000:80 nginx
```

7. In the filter box at the top of the Kibana page enter `*already allocated` and hit enter. Port collision errors should be reported and plotted:



```
June 12th 2017, 15:04:58.164   message: time="2017-06-12T19:04:58.161981529Z" level=error msg="Handler for POST /v1.27/containers/89d298
                               8ce725f22655e0be26ec20a25f213558dae38e887e9c5982320d9a772d/start returned error: driver failed programming
                               external connectivity on endpoint priceless_leakey (89e5f71ffb6d1f6e37530816afcc78998b9933f3513fe783d69c9f
                               e8453cd052): Bind for 0.0.0.0:8000 failed: port is already allocated" @timestamp: June 12th 2017, 15:04:
                               58.164 journal.systemd_slice: system.slice journal.comm: dockerd journal.gid: 0 journal.syslog_identi

June 12th 2017, 15:04:08.640   message: time="2017-06-12T19:04:08.640344570Z" level=error msg="Handler for POST /v1.27/containers/a801f3
                               0a0cb04d766fc7be365aeda532f1b81a8bc0451b7ad669c83dffd2c430/start returned error: driver failed programming
                               external connectivity on endpoint nifty_poitras (51cf2cd28e41589daa7a3d78e52c5cbecf595a1dc469313690c02b02b
                               f9d8da4): Bind for 0.0.0.0:8000 failed: port is already allocated" @timestamp: June 12th 2017, 15:04:08.
                               640 journal.systemd_slice: system.slice journal.comm: dockerd journal.gid: 0 journal.syslog_identifie

June 12th 2017, 15:02:01.178   message: time="2017-06-12T19:02:01.176674132Z" level=error msg="Handler for POST /v1.27/containers/ef6c76
                               990f357658e0589596e91821425d5c1547d488ca20e89f9b5927de2e02/start returned error: driver failed programming
```

12.4 Conclusion

In this exercise, you set up a basic ELK stack. ELK is just one of many options for ingesting and collating Docker's logs, but they all rely on the logging configuration established by `/etc/docker/daemon.json`. For more information on log drivers in Docker, see https://docs.docker.com/engine/admin/logging/overview/.

13 Health Checks

In this exercise we are going to show how the health check mechanism for services works. For this we're going to use a simple service that provides a `/health` endpoint which returns either `OK` (status 200) or `NOT OK` (status 500) depending on an internal setting. Initially the service reports its status as healthy. We can then call the endpoint `/kill` to put the service in an unhealthy state.

Please visit the source code to this service, which can be found here: https://github.com/docker-training/healthcheck.git.

13.1 Analyzing the Dockerfile

1. First let's look at the `Dockerfile` for this service. It looks as follows:

```
FROM ubuntu:16.04

RUN apt-get update && apt-get -y upgrade
RUN apt-get -y install python-pip curl
RUN pip install flask==0.10.1

ADD /app.py /app/app.py
WORKDIR /app

HEALTHCHECK CMD curl --fail http://localhost:5000/health || exit 1

CMD python app.py
```

2. Please specifically note the `HEALTHCHECK` line, which defines the command used to evaluate the health of the application. Exit code 0 is interpreted as healthy, and exit code 1 is interpreted as unhealthy.

3. Have a look at the application code itself; the most important part for healthchecking is the `/health` route:

```
@app.route('/health')
def health():
    global healthy

    if healthy:
        return 'OK', 200
    else:
        return 'NOT OK', 500
```

The app performs some logic to decide if it is healthy or not when `/health` is visited. This toy example just checks a bit, but a real example would have the same structure.

13.2 Deploying a Healthcheck-Enabled Service

1. Deploy a service with healthchecks enabled. This service will perform a healthcheck every two seconds; wait two seconds for a healthy response each time; declare a container failed after three consecutive failures; and wait 10 seconds after container launch to begin the healthchecks:

```
$ docker service create --name app \
    --health-interval 2s \
    --health-timeout 2s \
    --health-retries 3 \
    --health-start-period 10s \
    -p 5000:5000 training/healthcheck:ee2.0
```

2. Open a second terminal and run the following command to observe the `app` service:

```
$ watch docker service ps app
```

you should see something like this:

```
Every 2.0s: docker service ps app

ID            NAME      IMAGE                         NODE        DESIRED STATE  CURRENT STATE
ralw1apn8mgs  app.1     training/healthcheck:17.03   ucp-node-0  Running        Running 25 seconds
```

As we can see, the service is up and running happily.

3. But now we want to disrupt this peaceful state a bit. In your first terminal execute the command to put the service into an unhealthy status:

```
$ curl localhost:5000/kill -4
```

This flips the health bit in our app, so it starts reporting as unhealthy.

4. Observe what's happening in the second terminal where you run the `watch` command. After approximately 8 seconds you should see that the running instance gets killed and a new instance is started instead.
5. Explain the delay between the kill command and the moment the service gets effectively killed.
6. If Kibana is still running from a previous exercise, visit your Kibana UI and search for `unhealthy container` to see the system events logged when a healthcheck killed the container.
7. To get some detailed information about the last five healthchecks on a container, find the node the container is running on and do:

```
$ docker container inspect --format '{{json .State.Health.Log}}' <container id> \
    | python -mjson.tool
```

8. Clean up the system by removing the service:

```
$ docker service rm app
```

13.3 Optional: Running a Pod with a Liveness Probe

Prerequisites: Make sure your local host or remote desktop is configured for remote access to the Kubernetes cluster using a UPC client bundle.

1. Create a file `liveness.yaml` with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: busybox:latest
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

Please note the start command used with `busybox`. It creates a file `/tmp/healthy` and after 30 seconds deletes it. The livenessProbe command tests the existence of this file via `cat /tmp/healthy`.

2. Use `kubectl` to create the pod:

```
$ kubectl create -f ./liveness.yaml
```

3. Now after about 35 seconds execute this command to get more information about the health of the pod:

```
$ kubectl describe pod liveness-exec
```

at the end of the output we will find this:

```
...
Events:
  Type     Reason                 Age                From               Message
  ----     ------                 ----               ----               -------
  Normal   Scheduled              1m                 default-scheduler  Successfully assigned liven
  Normal   SuccessfulMountVolume  1m                 kubelet, minikube  MountVolume.SetUp succeeded
  Normal   Pulled                 1m                 kubelet, minikube  Successfully pulled image "
  Normal   Created                1m                 kubelet, minikube  Created container
  Normal   Started                1m                 kubelet, minikube  Started container
  Warning  Unhealthy              31s (x3 over 41s)  kubelet, minikube  Liveness probe failed: cat:
  Normal   Pulling                0s (x2 over 1m)    kubelet, minikube  pulling image "busybox:late
  Normal   Killing                0s                 kubelet, minikube  Killing container with id
```

The warning is indicating that the liveness probe is failing after 31s. The container is restarted.

4. Remove the pod when done:

```
$ kubectl delete pods liveness-exec
```

13.4 Conclusion

In this exercise we have shown a sample of how we can define a command in our `Dockerfile` which reports the health status of the containerized service. We have created a compose file that defines how frequently the health check should be executed, how many times we should allow the check to fail before the system kills the container running the service and starts a new instance instead, and how long to wait before beginning healthchecks after container initialization.

Each service should implement a health check endpoint. It is advised to come up with a convention in your company or team about how the health check endpoint of each service shall be called and what the possible responses are.

Next to the health check endpoint we need to define a command in the `Dockerfile` which is used to probe the endpoint and finally we can define upon creation of the service (e.g. in the docker-compose file) how frequently the endpoint shall be probed as well as how many failures shall be tolerated until the service is reported as unhealthy.

14 Installing Docker Trusted Registry

In this exercise, we'll walk through installing Docker Trusted Registry with a containerized storage backend.

Pre-requisites:

- UCP installed with 3 manager nodes

14.1 Add additional worker nodes to UCP

DTR is installed on top of UCP nodes. In a real deployment, we would add worker nodes dedicated to running DTR, and install DTR there. However for this demo, we'll just run DTR on the same nodes as our UCP managers, so no extra nodes need to be added at this step. **Do not do this in production!** Always provision dedicated nodes for DTR in a real environment.

14.2 Installing DTR

1. Navigate **Admin Settings** -> **Scheduler**.
2. Tick the box that says **Allow administrators to deploy containers on UCP managers or nodes running DTR** We need to enable this in order to be able to install DTR on our UCP nodes. Make sure to save this config if necessary.
3. By default, DTR uses ports 80 and 443 on whichever host it's installed on, but in our case, we're going to customize this to 81 and 4443 instead, since we'll be installing DTR on the same nodes as our UCP managers. Make sure nothing is listening on tcp/81 or tcp/4443 before continuing.
4. SSH into `ucp-manager-0`, and run the following commands. Note that in our case `DTR_FQDN` is going to be set to the FQDN of `ucp-manager-0`, since we're running DTR on the same nodes as UCP managers.

```
$ DTR_FQDN=<FQDN of ucp-manager-0>
$ UCP_IP=<public IP of ucp-manager-0>
$ docker container run -it --rm docker/dtr:2.5.3 install \
    --ucp-node ucp-manager-0 \
    --ucp-username admin \
    --ucp-password adminadmin \
    --ucp-url https://${UCP_IP} \
    --ucp-insecure-tls \
    --replica-https-port 4443 \
    --replica-http-port 81 \
    --dtr-external-url https://${DTR_FQDN}:4443
```

   Note the `--ucp-password` flag above; if you changed your admin password when you installed UCP or during the Password Recovery exercise, you'll need to use that password here instead of `adminadmin`.

5. Wait for the install to complete, then navigate to **Shared Resources** -> **Stacks** on the UCP left sidebar. DTR should be listed there, with 9 services.
6. Open a new browser tab and navigate to `https://<ucp-manager-0 FQDN>:4443`; after accepting another security exception, and logging in with your UCP admin credentials, the DTR dashboard is presented.

14.3 Setting up a DTR Storage Backend

By default, all images uploaded to DTR will be kept on the local disk of the first DTR replica you just set up. If this node goes down, you will lose access to your images, *even if you have extra DTR replicas still working*. In order to provide true high availability, we must also set up a highly available storage backend for DTR. In this example, we'll set up an Amazon S3 bucket to back our image storage. In production, feel free to use whatever highly available storage you're comfortable working with - but do **not** fail to pick one and set it up. Otherwise, your images will not be highly available.

1. Along with your AWS node IPs, you should have received an S3 bucket name and region; find these now.
2. Navigate to your DTR deployment at `https://<ucp-manager-0 FQDN>:4443`, and click on **System** -> **Storage** -> **Cloud**. Fill in the form with the AWS Region Name and S3 Bucket Name you were provided. All other fields can be left blank.
3. Finally, click **Save** at the bottom; you should get a 'Success' message if all went well. Note that while the form refers to 'AWS' and we did set up an Amazon S3 bucket here, you can actually provide any storage backing

with an S3-compliant API via the same steps, including third-party and on prem solutions that run on your own infrastructure if you prefer.

14.4 Integrate UCP and DTR

By default Docker engine uses TLS when pushing and pulling images to an image registry like Docker Trusted Registry; as such, each Docker engine must be configured to trust DTR.

1. Switch into your `ucp-manager-0` terminal. In ~/.bashrc, add the following line at the bottom:

   ```
   export DTR_FQDN=<DTR FQDN>
   ```

   replacing `<DTR FQDN>` with the FQDN of your DTR instance. Source this file via `source ~/.bashrc` so these changes take effect in the current terminal.

2. Download the DTR certificate:

   ```
   $ sudo curl -k https://${DTR_FQDN}:4443/ca \
       -o /etc/pki/ca-trust/source/anchors/${DTR_FQDN}:4443.crt
   ```

3. Refresh the list of certificates to trust:

   ```
   $ sudo update-ca-trust
   ```

4. Restart the Docker daemon:

   ```
   $ sudo /bin/systemctl restart docker.service
   ```

5. Log in to DTR from the command line:

   ```
   $ docker login ${DTR_FQDN}:4443
   ```

   If your login is successful, you have configured your Docker engine to trust DTR.

6. Repeat steps 1-5 on all your VMs (use the script below); every Docker engine in the Swarm cluster must be configured to trust DTR.

   > **Note:** To speed up the whole process of configuring all nodes of our swarm we can use a bash script like the following:

   ```
   CERT=./<your-pem-name>.pem
   DTR_FQDN=<DTR_FQDN>
   NODE_LIST=<LIST-IP-ADDRESSES-OF-NODES-IN-SWARM>
   # e.g. ("52.23.162.247" "52.3.334.184" "52.205.102.106" ...)
   for NODE in "${NODE_LIST[@]}"; do
     ssh-keyscan -H $NODE >> ~/.ssh/known_hosts
     ssh -t -i $CERT centos@$NODE hostname
     ssh -t -i $CERT centos@$NODE sudo curl -k https://${DTR_FQDN}:4443/ca \
       -o /etc/pki/ca-trust/source/anchors/${DTR_FQDN}:4443.crt
     ssh -t -i $CERT centos@$NODE sudo update-ca-trust
     ssh -t -i $CERT centos@$NODE sudo /bin/systemctl restart docker.service
   done;
   ```

   Replace the placeholders with the respective values of your infrastructure.

14.5 Pushing your first image

1. Via the DTR web UI, click on the **New Repository** button to create a repository called `hello-world`. The repository should go under your **admin** account.
2. On `ucp-manager-0`, pull the `hello-world` image from Docker Store:

```
docker image pull hello-world:latest
```

3. Re-tag the image, and note the correct namespacing for pushing to your DTR; the top level namespace must be the DTR FQDN and port you provided to the `--dtr-external-url` flag during install:

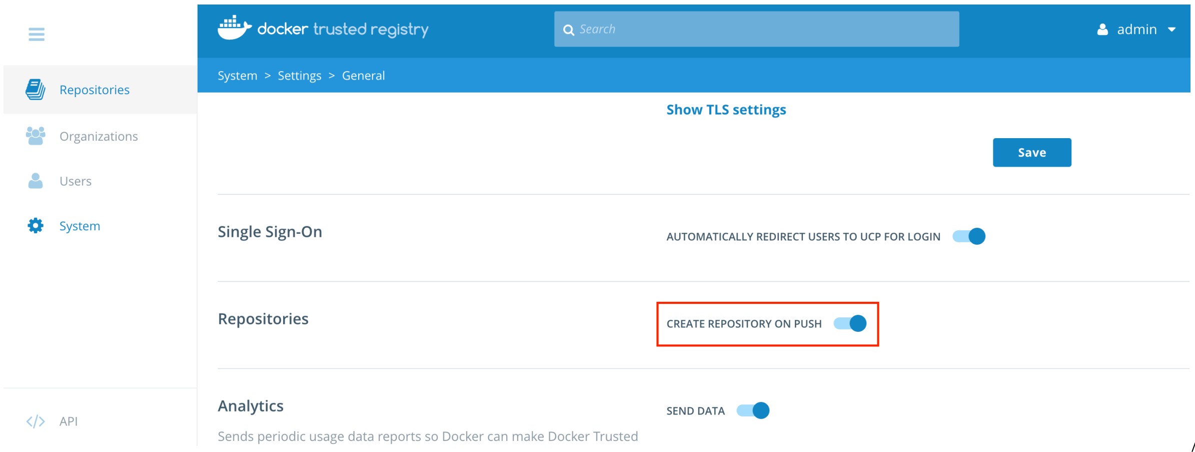
```
$ docker image tag hello-world:latest ${DTR_FQDN}:4443/admin/hello-world:1.0
```

4. Push to DTR:

```
$ docker image push ${DTR_FQDN}:4443/admin/hello-world:1.0
```

   and verify that you can see your image in DTR by navigating **Repositories** -> **admin/hello-world** -> **Images**.

5. Now, we will configure the option to auto create a repository on push. In the DTR web UI, navigate to the **System** -> **General**, and enable **Create repository on push**:



Single Sign-On — AUTOMATICALLY REDIRECT USERS TO UCP FOR LOGIN (on)

Repositories — CREATE REPOSITORY ON PUSH (on)

Analytics — SEND DATA (on)

Sends periodic usage data reports so Docker can make Docker Trusted

6. Back on `ucp-manager-0`, re-tag the `hello-world` image for a repository that has not yet been created:

```
$ docker image tag hello-world:latest ${DTR_FQDN}:4443/admin/hellohello
```

7. Push to DTR:

```
$ docker image push ${DTR_FQDN}:4443/admin/hellohello
```

   and verify that a new repository has been created and that you can see your image.

14.6 Conclusion

In this exercise we have installed Docker Trusted Registry, which is now running on top of UCP, with an S3 backing for your images. Note that DTR installation expects ports tcp/443 and tcp/80 to be available by default (4443 and 81 in our example above); if they aren't, double check whether one of those ports are occupied by a service in UCP (including the HTTP routing mesh, which defaults to tcp/80). Furthermore, `x509` errors when logging into DTR are usually a symptom of not correctly setting up trust between UCP and DTR; note that the FQDN provided in `--dtr-external-url` when installing DTR is explicitly written to DTR's certificate to allow this trust. This is also the FQDN (and port if not the default 443) that is the required top level namespace for all images to be pushed to this DTR.

15 (Optional) Install DTR Replicas

In this exercise, you'll set up 2 additional DTR replicas in order to implement high availability for your registry.

Pre-requisites:

- UCP and DTR set up as described in the 'Installing DTR' exercise.

Note: in this exercise, we'll continue setting up DTR replicas on our UCP managers. As noted in the last exercise, this is fine for a demo, but should not be done in a business critical environment! In practice, DTR should be installed on dedicated UCP workers.

15.1 Installing Replicas

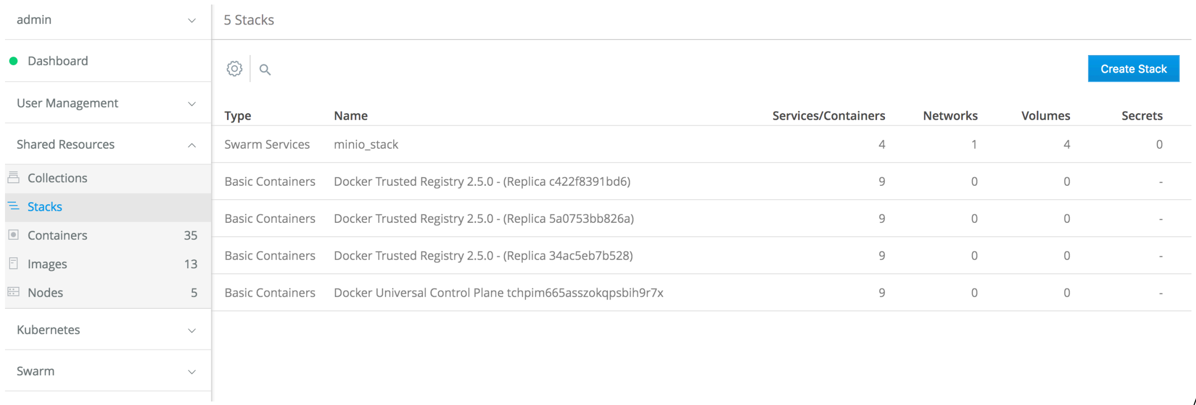
1. On your `ucp-manager-0` node, run the `docker/dtr` bootstrapper to configure `ucp-manager-1` as a DTR replica:

```
$ UCP_FQDN=<FQDN of UCP manager leader>
$ docker container run -it --rm docker/dtr:2.5.3 join \
    --ucp-node ucp-manager-1 \
    --ucp-username admin \
    --ucp-password adminadmin \
    --ucp-url https://${UCP_FQDN} \
    --ucp-insecure-tls \
    --replica-https-port 4443 \
    --replica-http-port 81
```

   You'll be asked to 'Choose a replica to join'; press return to accept the default (the only currently available option). Take note of this replica ID. **Note** the `--ucp-password` flag above; if you changed your admin password when you installed UCP or during the Password Recovery exercise, you'll need to use that password here instead of `adminadmin`.

2. Run the bootstrapper again but this time, change the `--ucp-node` flag to `ucp-manager-2`. When asked which replica to join, make sure to choose the same one you did in the last step (won't be the default this time).

3. Check the **Stacks** page inside the UCP web UI. You should now see three instances of DTR.



Type	Name	Services/Containers	Networks	Volumes	Secrets
Swarm Services	minio_stack	4	1	4	0
Basic Containers	Docker Trusted Registry 2.5.0 - (Replica c422f8391bd6)	9	0	0	-
Basic Containers	Docker Trusted Registry 2.5.0 - (Replica 5a0753bb826a)	9	0	0	-
Basic Containers	Docker Trusted Registry 2.5.0 - (Replica 34ac5eb7b528)	9	0	0	-
Basic Containers	Docker Universal Control Plane tchpim665asszokqpsbih9r7x	9	0	0	-

4. Verify that you can visit the DTR dashboard via the public IP of both `ucp-manager-1` and `ucp-manager-2`.

15.2 Cleanup

1. Navigate **Admin Settings** -> **Scheduler** in UCP
2. Untick both checkboxes under **Container Scheduling**. This is so that later on, when we run containers, we do not accidentally get them scheduled on our DTR nodes or on our UCP manager nodes.

15.3 Conclusion

After completing this exercise, your DTR instance is running in high availability mode. Just like the UCP manager consensus, the DTR replicas maintain their state in a database, and elect a leader via a Raft consensus.

16 Working with Organizations and Teams

Pre-requisites:

- DTR set up as per the 'Install DTR' exercise above.

16.1 Creating Organizations and Teams

1. Create two new user accounts, for users **Chandrasekhar** and **Suzuki**. You must be logged in as an admin to do this.
2. Create a new organization called **myorg** in DTR (navigate **Organizations** -> **New Organization**).
3. Create two teams under myorg, **database** and **web** (navigate **Organizations** -> **myorg** -> **+**).
4. Add Chandrasekhar to the database team, and Suzuki to the web team (navigate **Organizations** -> **myorg** -> **database** -> **Add User**).
5. Create a repository **myorg/db** owned by myorg: navigate **Organizations** -> **myorg** -> **Repositories** -> **Add Repository** -> **New**. Do the same again for another repo **myorg/ui**.
6. Give the database team R/W access to **myorg/db**: navigate **Organizations** -> **myorg** -> **database** -> **Repositories** -> **New Repository** -> **Existing**, and pick the appropriate choices from the dropdown lists.
7. Finally, assign the following permissions in the same fashion:
   - database team, R/O access to **myorg/ui**
   - web team, R/W access to **myorg/ui**
   - web team, R/O access to **myorg/db**

16.2 R/W in Org Repos

1. Log in to the DTR web UI as **Suzuki**, and click on Repositories on the left sidebar to see a list of all repos you have read access to. Use the dropdown to filter only repos that belong to the myorg org.
2. On `ucp-manager-0`, tag any arbitrary image as `${DTR_FQDN}:4443/myorg/ui:1.0`
3. Log in to DTR from the bash shell as Suzuki, and push the ui image you just tagged. Everything should proceed as normal.

   ```
   $ docker login ${DTR_FQDN}:4443
   ```

4. Tag another image as `${DTR_FQDN}:4443/myorg/ui:1.1`, log in to DTR from the shell as Chandrasekhar this time, and push the image. The image push should fail - why?
5. Log into the DTR web UI as admin, and set both the `db` and `ui` repositories to **Private** (look under the **Settings** tab for each). A 'private' icon should appear next to the repo name when this has been set correctly. Then log in to the DTR web UI as Suzuki and check if that user can still see `myorg/ui` and `myorg/db`. Why or why not?
6. Finally, revoke the **database** team's read only permission to the `ui` repo, and the **web** team's read only permission to the `db` repo. Can Suzuki see both repositories now? Why or why not?

16.3 Establishing Access Tokens

1. Log into the DTR web UI as admin, and navigate **Users** -> **Suzuki** -> **Access Tokens**.
2. Click **New Access Token**, give the token a name, and click **create**. An access token is presented *only this once*! Copy it, don't lose it, and click **Done**.
3. Try logging into DTR from the cli as Suzuki, using this new access token instead of Suzuki's usual password.

16.4 Conclusion

In this exercise, we saw how organization-owned repositories appear on DTR. Users can read and write to repos based on the permissions afforded them by their team; private org-owned repos become invisible to any user without at least explicit read access to that repo.

Access tokens can also be granted to users to obfuscate upstream passwords; if an access token is compromised, it can be rotated out without revealing the user's root password.

17 Content Trust

In this exercise you will install and use Notary to sign and push a trusted image.

Pre-requisites:

- DTR set up as per the 'Install DTR' exercise above.

17.1 Install the Notary CLI

1. SSH into the **ucp-manager-0** VM.
2. Use `wget` to download Notary:

   ```
   $ wget https://github.com/theupdateframework/notary/releases/download/v0.4.3/notary-Linux-amd64
   ```

   > **Note:** at this time Notary 0.4.3 is the latest Notary compatible with Docker 17.06.x.

3. Copy the downloaded binary file into the `/usr/local/bin` folder and rename it to just `notary`:

   ```
   $ sudo cp notary-Linux-amd64 /usr/local/bin/notary
   ```

4. Set the permissions on the `notary` client:

   ```
   $ sudo chmod +x /usr/local/bin/notary
   ```

17.2 Configure the Notary CLI

Notary commands require pointers to the Notary server, and DTR certificates; after acquiring the Notary binary, it's convenient to alias the `notary` command to handle this automatically:

1. Get a copy of the DTR CA certificate:

   ```
   $ DTR_FQDN=<DTR_FQDN>
   $ curl https://${DTR_FQDN}:4443/ca > dtr-ca.pem
   ```

2. In your `~/.bashrc` file, add the following alias:

   ```
   DTR_FQDN=<DTR_FQDN>
   alias notary="notary -s https://${DTR_FQDN}:4443 -d ~/.docker/trust --tlscacert ~/dtr-ca.pem"
   ```

3. Source your new `.bashrc` so the changes take effect:

   ```
   $ source ~/.bashrc
   ```

17.3 Set Up Content Trust for a repository

1. Login to the DTR web client as your admin user.
2. Create a new repository under the **engineering** organization called `enterprise-app`.

3. Check if Notary has content trust data for your `enterprise-app` repository by running, back at the command line:

   ```
   $ notary list ${DTR_FQDN}:4443/engineering/enterprise-app
   ```

   When prompted for a username and password, enter your admin DTR credentials. Notice the error message since we do not have any trust data at the moment.

4. Initialise the content trust collection by running:

   ```
   $ notary init -p ${DTR_FQDN}:4443/engineering/enterprise-app
   ```

   You will be prompted to pick a passphrase for a **root key, targets key** and **snapshot key**. Do not lose these.

   > **Note:** If you get an error like this: `fatal: you are not authorized to perform this operation: server returned 401.`, make sure you're not using `https://` in `${DTR_FQDN}` but only the public IP address or DNS name of your assigned DTR.

17.4 Push a signed image

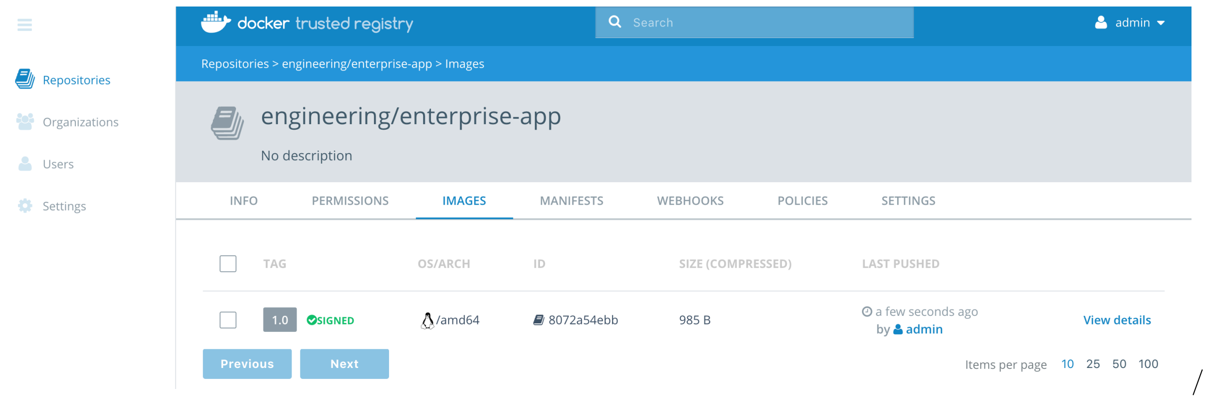
1. Pull the `hello-world` image into your **ucp-manager-0** node.
2. Tag the image with your DTR enterprise-app repo

   ```
   $ docker image tag hello-world ${DTR_FQDN}:4443/engineering/enterprise-app:1.0
   ```

3. Enable content trust by running:

   ```
   $ export DOCKER_CONTENT_TRUST=1
   ```

4. Login to DTR as the admin user on the Docker CLI if you haven't already.
5. Push the `${DTR_FQDN}:4443/engineering/enterprise-app:1.0` image to DTR. You will be prompted for your repository key passphrase.
6. Go to the visit your `enterprise-app` repository in DTR via the web client, and click on the **Images** tab. You should be able to see the 1.0 image you pushed. It should be marked as signed.



engineering/enterprise-app

No description

TAG	OS/ARCH	ID	SIZE (COMPRESSED)	LAST PUSHED	
1.0 SIGNED	/amd64	8072a54ebb	985 B	a few seconds ago by admin	View details

17.5 Signing Delegation

Instead of transferring repo keys to developers or our CI/CD pipeline, we can issue **delegation keys** that allow third parties to sign images as above, without risking compromising our snapshot or target keys. The snapshot key can be managed by DTR's Notary server, eliminating the need to share this with users.

1. On the node with notary installed from above, rotate the repository **snapshot** key:

   ```
   $ notary key rotate ${DTR_FQDN}:4443/engineering/enterprise-app \
       snapshot --server-managed
   ```

   You'll be required to provide your admin DTR login credentials, as well as your root key password.

17.6 Delegate image signing

For each user that you want to delegate image signing to, you need to get their `cert.pem` file. This is the `cert.pem` file that comes with each user's **client bundle**. We'll delegate our image signing to the user **Chloe**.

1. Login to the UCP web UI as **Chloe**.
2. Download the client bundle to your local machine or remote desktop (**Chloe** -> **My Profile** -> **Client Bundles** -> **+ New Client Bundle**).
3. Upload the zip archive to the node with the notary CLI set up:

   ```
   $ scp ucp-bundle-chloe.zip centos@<node-IP>:~/.
   ```

4. Unzip the archive (you may need to do `sudo yum install -y unzip` to get the unzip util first). Run the following command to create a new Notary delegation role, using the user certificate:

   ```
   $ notary delegation add -p ${DTR_FQDN}:4443/engineering/enterprise-app \
       targets/releases --all-paths cert.pem
   ```

   When prompted for authentication details, specify your admin DTR credentials.

17.7 Push a delegated signed image

For this step, we will pretend that we are the user **Chloe**.

1. Open a new ssh session into the same node you've been working on (in reality, this would be Chole's machine which they have also installed Notary CLI on, but we'll use the same one for expedience).
2. In the folder you unzipped Chloe's client bundle, run the command:

   ```
   $ notary key import key.pem
   ```

   This is needed because before delegated users can sign content with Notary, they must import the private key associated with the user certificate used to make the delegation role above. You will be prompted for a passphrase for the delegation key. Pick a passphrase and make sure to remember it.
3. Pull the `hello-world` image into the node

   ```
   $ docker image pull hello-world
   ```

4. Re-tag the `hello-world` image with the `engineering/enterprise-app` repository and use a 2.0 tag.

   ```
   $ docker image tag hello-world \
       ${DTR_FQDN}:4443/engineering/enterprise-app:2.0
   ```

5. Enable content trust

   ```
   $ export DOCKER_CONTENT_TRUST=1
   ```

6. Login to DTR on the Docker CLI as Chloe, and push the `${DTR_FQDN}:4443/engineering/enterprise-app:2.0` image to DTR.

   What do you notice? You should see a **denied** error message. This is because Chloe does not have permissions to push to the repository.
7. Login to the DTR web UI.
8. Navigate **Organizations** -> **engineering**.
9. From here, create a new team called **qa**, add Chloe to it, and make sure that team has R/W permission to the `engineering/enterprise-app` repository.
10. Try and push the `engineering/enterprise-app` repo to DTR again, as Chloe. This time, you will be prompted for your delegation key passphrase and the push should succeed.

```
$ docker push ${DTR_FQDN}:4443/engineering/enterprise-app:2.0
The push refers to a repository [<DTR_FQDN>:4443/engineering/enterprise-app]
98c944e98de8: Layer already exists
2.0: digest: sha256:2075ac87b043415d35bb6351b4a59df19b8ad154e578f7048335feeb02d0f759 size: 524
Signing and pushing trust metadata
Enter passphrase for delegation key with ID 0491cf9:
Successfully signed "<DTR_FQDN>:4443/engineering/enterprise-app":2.0
```

11. Go to the enterprise-app repository on your web browser and check that the 2.0 tag reports being signed and pushed by Chloe.

17.8 Conclusion

Now that Notary is installed and content trust is activated, the Notary server integrated in DTR maintains signed metadata that describes the contents of a repository (targets), how old the content is (timestamp), and what combinations of those two are valid (snapshot). A client's Notary server will sync this metadata locally, and use it to defeat downgrade, mix-and-match, and other man-in-the-middle attacks when fetching a signed image tag from the registry.

Furthermore, by allowing the appropriate keys to be managed by the notary server, you were able to delegate signing authority to another user on your team, without ever having to transmit your content trust keys.

18 Image Scanning in DTR

In this exercise, you'll walk through enabling image scanning in DTR, and inspecting the results of a scan.

18.1 Pre-requisites

- DTR set up as per the 'Install DTR' exercise above.

18.2 Enabling Image Scanning

1. Log into your DTR web ui as an admin, and navigate **System** -> **Security** and click the 'Enable Scanning' toggle. Click 'Enable' on the popup, without changing the default behavior.
2. Notice that a yellow warning appears saying that the 'Vulnerability Database is updating', the CVE database version number is 0, and the button below this is disabled, has a spinner, and says 'Database Updating'. All these things indicate that DTR is downloading its first CVE database. You can proceed with the exercise, but **do not push** until this database download is complete!

18.3 Creating a repo

1. Create a new user-owned repository called `admin/whale-test`. On the repo creation page, click **Show Advanced Settings**, and enable the 'SCAN ON PUSH' toggle. Click **Create** when done.
2. Pull the `ubuntu:16.04` image or any image that you would like to perform image scanning on:

```
$ docker image pull ubuntu:16.04
```

3. Re-tag the image with your DTR repository name and a tag of 1.0.

```
$ docker image tag ubuntu:16.04 \
    ${DTR_FQDN}:4443/admin/whale-test:1.0
```

4. **Make sure the CVE database download is complete**, and then log in and push the image to DTR:

   ```
   $ docker login ${DTR_FQDN}:4443
   $ docker image push ${DTR_FQDN}:4443/admin/whale-test:1.0
   ```

18.4 Investigating layers & components

1. Go to the repository on the DTR web UI and verify that you now have a 1.0 tag on the repository. At the moment, this tag probably reports 'Scanning' with a spinner under the Vulnerabilities column. Wait for this to finish (it can take several minutes).
2. Click 'View Details' link of your image, it will take you to 'Layers' view. Dockerfile entries are listed on the left; clicking on them will display the report for that layer.
3. Click on 'Components'; the same vulnerabilities are listed by component instead of layer, with links to the relevant CVE reports.
4. In the Components view, try clicking on **Hide** next to one of the CVE summaries; this CVE will now not be counted towards vulnerability totals for this *or any other* image in DTR. In this way, you can mark a vulnerability as not a concern.

18.5 Conclusion

At this point, layers pushed to the repo you created in this exercise will be scanned when they are received by DTR, and when the local CVE database is updated. These scans in turn can be set to trigger webhooks on scan completion or vulnerability detection, allowing users to integrate image scanning into CI/CD pipelines.

19 Image Promotion & Webhooks

In this exercise, we'll set up a simple image pipeline using image promotion and webhooks. Once complete, you will have a repository to push new development versions of images to, which will promote those images to different repositories based on the results of a security scan. Finally, you'll fire a webhook when an image is ready to go to QA.

19.1 Creating a Promotion Pipeline

1. Create three repositories: **`admin/whale-dev`**, **`admin/whale-qa`**, and **`admin/whale-vulnerable`**.
2. Navigate **Repositories** -> **`admin/whale-dev`** -> **Promotions** -> **New Promotion Policy**, and create a policy that promotes images to **`admin/whale-qa`** if they have zero critical vulnerabilities. Also make sure the promoted image changes the tag to `qa-*`, where the asterisk is whatever the original tag of the image was.
3. In the same repository, create another policy that promotes images that *do* have a critical vulnerability to **`admin/whale-vulnerable`**. Retag these images with the original tag, plus the current date.
4. Visit **`admin/whale-qa`** and **`admin/whale-vulnerable`**'s Promotions tab, and click on 'Is Target'; you should see the policies you just created that push images to each repository.

19.2 Setting up a Webhook

1. On `ucp-manager-0`, create a service that will catch and display the webhook's POST payload (in reality, this would be some tooling like a build manager or notification system that needs to take action when the webhook fires, but for the sake of simplicity we'll just create a service that logs the POST information):

   ```
   $ docker service create --name whale-webhook -p 8000:8000 training/whale-server
   ```

2. Back in DTR, navigate **Repositories** -> **`admin/whale-dev`** -> **Webhooks** -> **New Webhook**.
3. Choose 'Tag Pushed to Repository' for the notification to receive, and provide `http://${UCP_FQDN}:8000` as the webhook URL. Now whenever an image is pushed to this repository, DTR will POST some JSON describing the newly pushed image to the URL you provided.
4. Save this webhook.

19.3 Triggering the Pipeline

1. Find an image on Docker Store that has some critical vulnerabilities, eg `ubuntu:16.04`. Pull it down, tag it as **`${DTR_FQDN}:4443/admin/whale-dev:0.1`**, and push it to DTR. Make sure the promotion to **`admin/whale-vulnerable`** worked as expected.
2. Find an image on Docker Store that has *no* critical vulnerabilities, eg `busybox:latest` or `alpine:latest`, tag it as **`${DTR_FQDN}:4443/admin/whale-dev:0.2`**, and push it to DTR. Check that it was promoted as expected, and look at the container logs of the service you started above to see the results of the POST request from triggering the webhook.

19.4 Optional: Image Mirroring

DTR can promote an image not only to another repository it controls, but to a repository in an entirely separate deployment of DTR; we refer to this as 'image mirroring'. If feasible, try this with a partner; use one partner's DTR as 'DTR 1', and the other's deployment as 'DTR 2'.

1. Login as `admin` to **DTR 1** and to **DTR 2**.
2. On **DTR 1** create a public repository `admin/mirror-origin`.
3. On **DTR 2** create a public repository `admin/mirror-target`.
4. Back on **DTR 1** add a new **Mirroring Policy** for `admin/mirror-origin` with **REGISTRY URL** pointing to `admin/mirror-target` on **DTR 2**.
5. Under **Show advanced settings** select **SKIP TLS**.
6. Do not add any triggers at this time. This will cause all images pushed to the repo on DTR 1 to be mirrored to the target on DTR 2.
7. Click **Connect** and then **Save and Apply** on the bottom of the view.
8. In the CLI e.g. tag an alpine image to `<DTR_1_FQDN>:4443/admin/mirror-origin:1.0`.
9. Push the above image.
10. Demonstrate that the image is available in **DTR 1**, and has been mirrored to **DTR 2**.

19.5 Conclusion

In this exercise, you set up a basic DTR pipeline. Images can be promoted from one repository to another based on conditions that reflect progress through your CI/CD, review, and approval process, allowing you to create a clear picture of how an image moves from development to production, and understand precisely where a given image is in that process. Furthermore, webhooks can be fired at any point in this process to integrate with the rest of your tooling.

20 The Software Supply Chain

Please organize yourself in teams of 2 or maximum 3 persons.

20.1 Goal

Create a simple CI/CD pipeline with GitHub, UCP, DTR and Jenkins

20.2 Areas Covered

This final project is supposed to cover the following areas:

20.2.1 UCP

- Orchestration with Swarm (and Kubernetes)
- Orgs, teams and users
- Collections, node sets (Kube) and grants
- Node RBAC
- Rolling updates and rollbacks

20.2.2 DTR

- Repositories
- Orgs, teams and users
- Image promotion and mirroring
- Webhooks
- Content trust

20.3 Tasks

1. Person 1 contributes their lab as `Dev` environment. Person 2 contributes their lab as `Prod` environment.
2. If your team consists of 3 people then person 3 contributes their lab as `CI` environment environment where the CI server will run (affects point 5 below).
3. Fork repository /docker-training/dops-final-project on GitHub.
4. Analyze the code of this simple web project. It is a photo album showing some animals and will be used as the application that we will build, test and deploy.
5. In the `Dev` environment use the `elk` node as CI server. If your group consists of 2 people, use the `elk` node from the Dev environment as your CI server; clean the node and then add it to the UCP cluster. If your group consists of 3 people, use the third person's UCP as your CI environment instead.
6. Deploy a containerized version of Jenkins to your CI environment. >**Note:** If you're not familiar with Jenkins you can get inspired by the optional exercise "Creating a Build Server" in your exercise book. This exercise walks through a sample setup of Jenkins step-by-step. The exercise is taken "as is" from another Docker course and thus you might need to adjust the paths and node names so that they match with your environment.
7. Configure Jenkins to pull the latest code from GitHub whenever someone commits changes to the code.
8. Have Jenkins build the image.
9. Have Jenkins run unit tests against the image.
10. If the build is successful and the unit tests pass, then have Jenkins push the image to the `Dev` DTR.
11. If the image scan is successful and no vulnerabilities are detected, the image is automatically promoted to the "QA" repository in `Dev` DTR.
12. The "QA" repo should have a webhook that triggers Jenkins to deploy the app into the Dev environment (if you're a 2 person group) or the CI environment (if you're a three person group) using SwarmKit as orchestration engine.
13. Person 1 which is also a QA manually signs the image.
14. QA person manually pushes signed image from QA repo to `Prod` DTR (unfortunately signing is lost during mirroring at this time…)
15. Define a webhook on the prod repo to trigger Jenkins to:
    - Sign the image in `prod` DTR
    - Deploy the application to the `Prod` environment

16. Update something in the application code to trigger the pipeline

20.3.1 Optional

1. Configure all worker nodes to accept mixed workloads
2. Migrate the application to Kubernetes.
3. Define health checks (liveness and readiness probes) for the application.
4. Update the application code again, this time with an error to trigger a rollback during deployment.

21 Appendix: Build Server

In a typical Docker pipeline, a build agent will be responsible for pulling code and base images from their respective registries, building developers' images, and pushing them to DTR to trigger a CI/CD pipeline. In this exercise, you'll configure a containerized Jenkins to watch a GitHub repo, build images out of code it finds there, and push those images to DTR.

A lot of information has been distilled from here: https://github.com/yongshin/leroy-jenkins.

21.1 Prerequisites

1. Make sure all your UCP swarm nodes trust DTR. As a reminder, here is the commands that you would use to do so for CentOS:

```
# Specify the fully qualified domain name (FQDN) of DTR (e.g. dtr.example.com)
$ DTR_FQDN=<DTR FQDN>
# Download the DTR CA certificate
$ sudo curl -k https://$DTR_FQDN:4443/ca -o /etc/pki/ca-trust/source/anchors/$DTR_FQDN:4443.crt
# Refresh the list of certificates to trust
$ sudo update-ca-trust
# Restart the Docker daemon
$ sudo /bin/systemctl restart docker.service
```

21.2 Creating and Shipping the Jenkins Image

1. On one of your UCP nodes, create a folder `build-server/jenkins`.
2. Add a `Dockerfile` to the folder with this content:

```
FROM jenkins:2.60.3
USER root
RUN apt-get update \
 && apt-get install -y \
        apt-transport-https \
        ca-certificates \
        curl \
        software-properties-common \
 && curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add - \
 && add-apt-repository \
        "deb [arch=amd64] https://download.docker.com/linux/debian \
        $(lsb_release -cs) \
        stable" \
 && apt-get update \
 && apt-get install -y docker-ce sudo \
 && rm -rf /var/lib/apt/lists/*
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers
```

```
COPY entrypoint.sh /
ENTRYPOINT /entrypoint.sh
```

we're using the base Jenkins image, and add Docker CE and a few other libraries to it.

3. Add a file entrypoint.sh to the same folder with the following content:

```
openssl s_client -connect ${DTR_IP}:443 -showcerts \
    </dev/null 2>/dev/null | openssl x509 -outform PEM | sudo tee \
    /usr/local/share/ca-certificates/${DTR_IP}.crt
sudo update-ca-certificates

/bin/tini -- /usr/local/bin/jenkins.sh
```

4. Make the above file executable:

```
$ chmod +x ./entrypoint.sh
```

5. Build the image, make the corresponding repo in DTR, and push:

```
$ docker image build -t ${DTR_FQDN}:4443/admin/my-jenkins:1.0 .
$ docker image push ${DTR_FQDN}:4443/admin/my-jenkins:1.0
```

21.3 Preparing the Repository in DTR

1. Login to your DTR as `admin`.
2. Create a public repo called `admin/jenkins-demo`. This is where Jenkins will push built images to.

21.4 Preparing a Source Repo

1. Log in to GitHub, and create a public repo called `jenkins-demo`. This is where the code descibing your image will be controlled.
2. On your local machine or remote desktop, make a fresh folder called `jenkins-demo`, and add a Dockerfile to it describing an image you're developing (feel free to just use the Dockerfile and entrypoint.sh defined above for the Jenkins image itself if you like).
3. Set this code up to push, but *don't* push yet:

```
$ cd jenkins-demo
$ git init
$ git remote add origin <your jenkins-demo GitHub repo url>
$ git add *
$ git commit -m 'code ready for pipeline testing'
```

21.5 Running Jenkins in the Swarm

21.5.1 Preparing the Swarm Node

1. SSH into the swarm node `ucp-node-1` where we'll run Jenkins.
2. Create a directory `jenkins` and a directory `ucp-bundle-admin`:

```
$ mkdir jenkins
$ mkdir ucp-bundle-admin
```

3. Create an auth token to use UCP's API from `ucp-node-1`:

```
$ UCP_IP=<UCP IP>
$ AUTHTOKEN=$(curl -sk -d '{"username":"admin","password":"adminadmin"}' \
    https://${UCP_IP}/auth/login | jq -r .auth_token)
```

4. Download and unzip a client bundle for your admin account:

```
$ cd ucp-bundle-admin
$ curl -k -H "Authorization: Bearer $AUTHTOKEN" \
    https://${UCP_IP}/api/clientbundle -o bundle.zip
$ unzip bundle.zip
```

5. Run the shell configuration script which is part of the bundle:

```
$ source env.sh
```

21.5.2 Creating the Jenkins Service

1. SSH into one of your UCP manager nodes.
2. Label the target node for Jenkins such that we can use constraints with the Jenkins service to place the master on this node:

```
$ docker node update --label-add jenkins=master ucp-node-1
```

3. Run a service for Jenkins:

```
$ export DTR_IP=<public IP of ucp-manager-0>
$ docker service create --name my-jenkins --publish 8080:8080 \
    --mount type=bind,source=/var/run/docker.sock,destination=/var/run/docker.sock \
    --mount type=bind,source=/home/centos/jenkins,destination=/var/jenkins_home \
    --mount \
    type=bind,source=/home/centos/ucp-bundle-admin,destination=/home/jenkins/ucp-bundle-admin \
    --constraint 'node.labels.jenkins == master' \
    --detach=false \
    -e DTR_IP=${DTR_IP} \
    ${DTR_FQDN}:4443/admin/my-jenkins:1.0
```

4. Verify that the service is up and running using e.g.:

```
$ docker service ps my-jenkins
```

21.5.3 Finalizing the Jenkins Configuration

1. Open a browser window and navigate to Jenkins at `http://<UCP_FQDN>:8080`
2. Enter the initial password that you can get on the node where you had installed Jenkins by entering this command:

```
$ sudo cat ~/jenkins/secrets/initialAdminPassword
```

3. Install the suggested plugins from the inital popup window, create your admin account and click **Continue as Admin** -> **Start Using Jenkins**.

21.6 Configuring Jenkins Jobs

1. Login to Jenkins if not already done.
2. Create a Jenkins Job of type **Free Style** and call it **docker build and push**.

3. In **Source Code Management** -> **Git** - set repository to the url of the `jenkins-demo` GitHub repo you created above.
4. Set Build Triggers -> Poll SCM, and enter `* * * * *` in the 'Schedule' field, to poll for code changes every minute. See https://en.wikipedia.org/wiki/Cron for an explanation of this scheduling syntax.
5. Add a Build Step of type **Execute Shell** and add the following script (remember to replace `<DTR_FQDN>` with the FQDN of your DTR deployment):

```
#!/bin/bash
export DTR_FQDN=<DTR FQDN>
docker image build -t ${DTR_FQDN}:4443/admin/jenkins-demo .
docker image tag ${DTR_FQDN}:4443/admin/jenkins-demo ${DTR_FQDN}:4443/admin/jenkins-demo:1.${BUILD_
docker login -u admin -p adminadmin ${DTR_FQDN}:4443
docker image push ${DTR_FQDN}:4443/admin/jenkins-demo
docker image push ${DTR_FQDN}:4443/admin/jenkins-demo:1.${BUILD_NUMBER}
```

6. Save the job.
7. Back on your development machine, push your `jenkins-demo` code to GitHub (`git push origin master` in the `jenkins-demo` folder). Wait a minute, and SCM polling in Jenkins will trigger a build and push your new image to DTR when complete.

21.7 Conclusion

In this exercise we set up a containerized Jenkins build server that ingests code from a version control repository, and pushes built images into Docker Trusted Registry. From this point, you can configure DTR to scan images for vulnerabilities, promote them to new repositories, and trigger webhooks to external services at each step.