# Data Control Tower Home

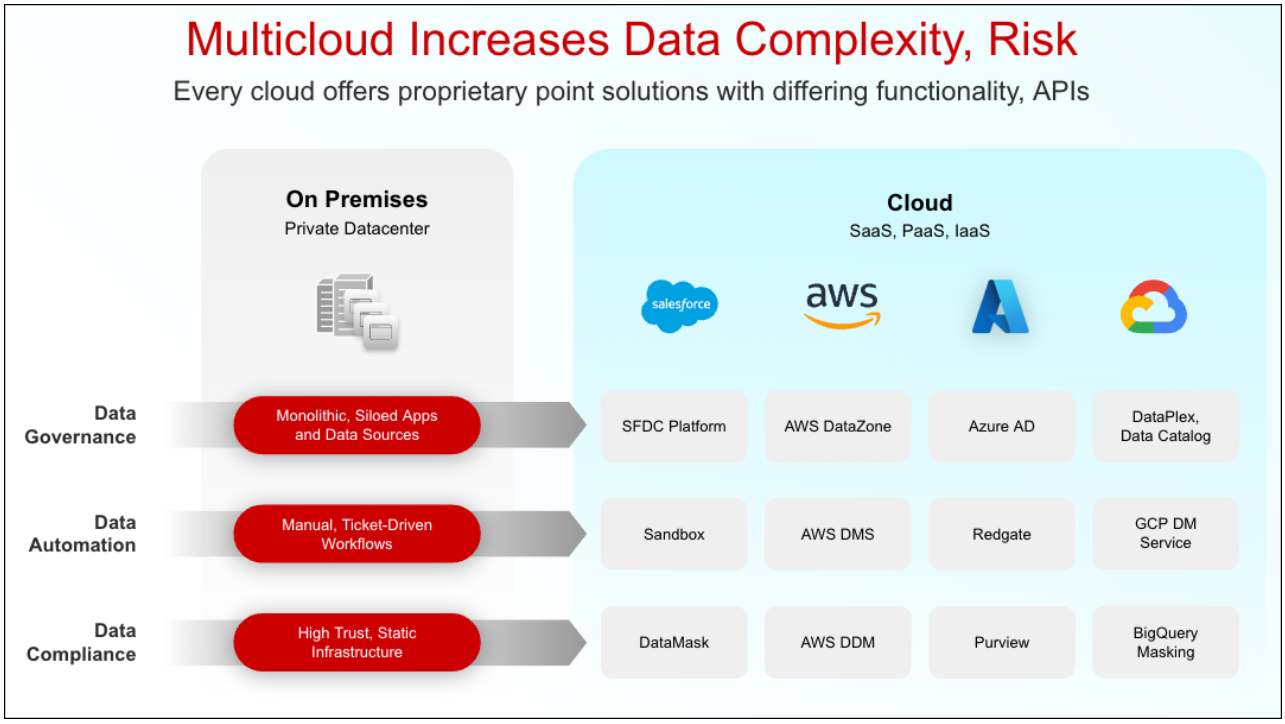
Data Control Tower

Exported on 08/15/2023

# Table of Contents

# 1 What is Data Control Tower (DCT)?

Today's application and data landscape is an increasingly complex ecosystem of hosting architectures, often represented by a multi-cloud landscape coupled with an explosion of different platforms and services. This fragmented picture of heterogeneous silos makes data governance, automation, and compliance a herculean, if not, an impossible task.




Data Control Tower (DCT) is an enabling Delphix platform that introduces a data mesh to unify data governance, automation, and compliance across all applications and cloud platforms.

Data governance is achieved through operational control and visibility of test data across multicloud applications, databases, environments, and releases. DCT brings data cataloging, tagging, and data access controls for central governance of all enterprise data, while providing the right data at the right time to development teams.

Data automation at CI/CD speed and enterprise scale is easier and more powerful, by combining **DCT with Continuous Data**. A unified API gateway, self-service automation tools, and plug-and-play DevOps integrations streamline the initial configuration and day-to-day workflows.

**DCT with Continuous Compliance** provides robust data compliance in lower environments, all while reducing costs and enabling fast, quality software development.

# 2 DCT concepts

## 2.1 Introduction

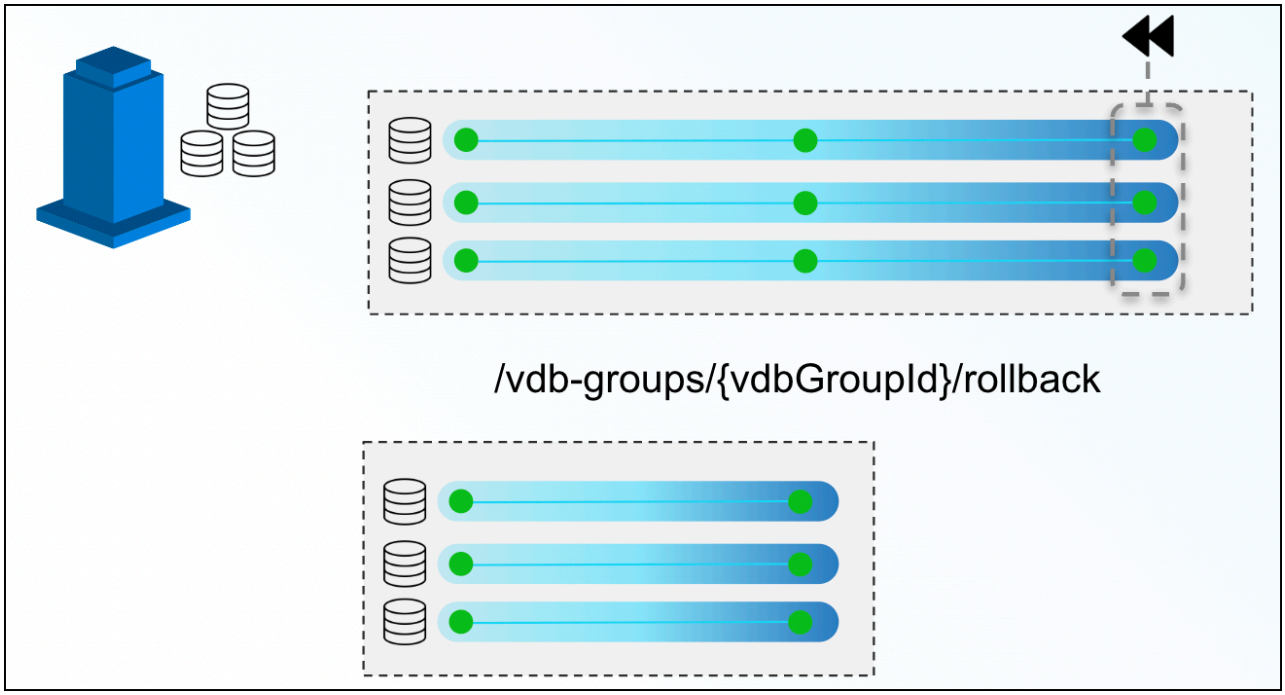
Data Control Tower (DCT) provides new and novel approaches to general Delphix workflows, delivering a more streamlined developer experience. This article will introduce these concepts to Delphix and how they work with DCT.

## 2.2 Concepts

### 2.2.1 Virtual Database (VDB) groups

Virtual Database (VDB) groups are a new concept to Delphix, which enable the association of one or more VDBs as a single VDB group. This allows for bulk operations to be performed on the grouped VDBs, such as bookmark, provision, refresh, rewind, and others. This will assist in complex application testing scenarios (e.g. integration and functional testing) that require multiple data sources to properly complete testing.

With VDB groups, developers can now maintain data synchronicity between all grouped VDBs, which is particularly useful for complex timeflow operations. For example, updating VDBs to reflect a series of schema changes across data sources, or to reflect an interesting event in all grouped datasets. In order to maintain synchronicity among grouped datasets, timeflow operations (refresh, rewind, etc.) must use a bookmark reference.




In the above example, a VDB Group reference is created for three VDBs. At the end of the above timeline group, a developer decides to rollback those VDBs to a previous snapshot. By issuing a single command via the VDB groups endpoint, DCT will move all three back, ensuring that they all maintain referential synchronicity.

Bookmarks and VDB groups are loosely related; a VDB group can exist in the absence of any bookmarks, and a bookmark can exist without any VDB group. It is important to note that the bookmark represents data, while the VDB group represents the databases to make this data available.

DCT will automatically stop an operation from executing if one or more objects are incompatible (e.g. provisioning a VDB group into a set of environments, where one of the VDBs is incompatible, such as an Oracle on Linux VDB provisioned onto a Windows environment).

VDB groups based operations will return a single job to monitor the overall status of the series of individual VDB operations. If one of those individual operations is unable to complete, DCT will report a "fail", but any individual operations that are able to successfully complete will still do so.

### 2.2.2 Comparing Self-Service containers to VDB groups

As mentioned above, VDB groups are a crucial DCT concept that enable Self-Service functionality outside of the Self-Service application. Consider VDB groups acting similarly to Self-Service containers, in that it provides grouping and synchronization among VDBs, but VDB groups can provide a more flexible approach for users. Here are some additional points for example:

- The same VDB can be included in multiple VDB groups
- Including a VDB in a VDB group does not prevent operations on the VDB individually
- VDBs can be added to or removed from VDB groups
- VDB groups do not have their own timeline

### 2.2.3 Bookmarks

DCT Bookmarks are a new concept that represents a human-readable snapshot reference that is maintained within DCT. This is not to be confused with Self-Service bookmarks, maintained separately within the Self-Service application. With DCT Bookmarks, developers can now reference meaningful data (e.g. capturing a schema version reference to pair with an associated code version, capturing test failure data so that developers can reproduce the error in a developer environment, etc.) and use those references for any number of use-cases (e.g. versioning data as code, quickly provisioning a break/fix environment with relevant data, etc.). DCT Bookmarks are compatible with both VDBs and VDB groups, and can be used as a reference for common timeflow operations such as:

- Provisioning a VDB or VDB group from a bookmark
- Refreshing a VDB or VDB group to a bookmark
- Rewinding a VDB or VDB group to a bookmark

DCT Bookmarks have associated retention policies, the default value is 30 days, but policies can be customized anywhere from a day to an infinite amount of time. Once the Bookmark expires, DCT will delete the bookmark.
Bookmarks are compatible with individual VDBs and VDB groups. Bookmark Sharing is only available for engines on version 6.0.13 and above.
DCT Bookmarks, when created, initiate a snapshot operation on each and every VDB in order to maintain synchronicity between each VDB. In that same vein, bookmark-based VDB group operations will have each VDB-specific sub-process run in parallel (as opposed to sequentially) to reduce drift between grouped VDBs.

### 2.2.4 Jobs

Jobs in DCT are the primary means of providing operation feedback (PENDING, STARTED, TIMEDOUT, RUNNING, CANCELED, FAILED, SUSPENDED, WAITING, COMPLETED, ABANDONED) for top-level operations that are run on DCT. Top-level operations represent the parent operation that may have one or more child-based jobs (e.g. refreshing a

VDB group is the parent job to all of the individual refresh jobs for the grouped VDBs under the VDB group reference).

> Top-level jobs will report a “FAILED” status if one or more child jobs fail. For child jobs that can complete, DCT will continue to complete those jobs even if a parent job reports a failure.

### 2.2.5 Tags

DCT Tags enable a new business metadata layer for users and consumers to filter, sort, and identify common Delphix objects, to power any number of business-driven workflows. A tag is comprised of a (Key:Value) pair that associates business-level data (e.g. location, application, owner, etc.) with supported objects. DCT 2.0 and above support the following Tags:

- Continuous Data Engines
- Environments
- dSources
- VDBs

Developers and administrators add and remove tags using tag-specific object endpoints (e.g. `/vdbs/{vdbId}/tags`) and can leverage tags as search criteria when using the object-specific search endpoints (e.g. using filtering language to narrow results).

Some sample tag-based use-cases include:

- Refreshing all the VDBs owned by a specific App Team using an “Application: Payment Processing” tag. This would be accomplished by querying “what VDBs have the (Application: Payment Processing) tag" and feeding those VDB IDs into the refresh endpoint.
- Driving accountability for VDB ownership by tagging primary and secondary owners for each VDB (e.g. (primary_owner: John Smith), (secondary_owner: Jane Brown)). That way, if a VDB is overdue for a refresh, tracking down an owner is a simple tag query.

> Tags are registered as an attribute that is specific to an object as opposed to a central tagging service. As a result, tag-based querying can only be done on a per-object type basis.
>
> A supported object can contain any number of tags.

### 2.2.6 Tag-based filtering

All taggable objects support tag-based filtering for API queries that adhere to the search standards documented in API References(see page 28). A few examples of how tag-based filtering can be used are as follows:

List all VDBs of type `'Oracle'`, of which IP address contains the `'10.1.100'` string and which have been tagged with the `'team'` tag, `'app-dev-1'`.

```
database_type EQ 'Oracle' AND ip_address CONTAINS '10.1.100' and tags CONTAINS { key
EQ 'team' AND value EQ 'app-dev-1'}
```

## 2.3 Nuances

### 2.3.1 Stateful APIs

All applicable DCT APIs are stateful so that running complex queries against a large Delphix deployment can be done rapidly and efficiently. DCT accomplishes this by periodically gathering and hosting telemetry-based Delphix metadata from each engine.

### 2.3.2 Local data availability

DCT currently relies on existing Continuous Data and Compliance constructs around data-environment-engine relationships. This means that DCT operations require VDBs to live on the engine where the parent dSource lives and so on.

### 2.3.3 Engine-to-DCT API mapping

Wherever possible, DCT has looked to provide an easier-to-consume developer experience. This means that in some cases, an API on DCT could have an identical API on an engine. However, there are many instances of providing a higher level abstraction for ease of consumption; one example is the data inventory APIs on DCT (sources, dSources, VDBs), which are a simplified representation of data represented by the source, sourceconfig, and repository endpoints on the local engine (source, dSource, and VDB detail are all combined under those three endpoints).

### 2.3.4 Local references to global UUIDs

In order to avoid collision of identically-named and referenced objects, DCT generates Universally Unique IDentifiers (UUID) for all objects. For existing objects on engines like dSources and VDBs, DCT will concatenate the local engine reference with the engine UUID (e.g. `'Oracle-1'` on engine `'3cec810a-ee0f-11ec-8ea0-0242ac120002'` will be represented as `'Oracle-1-3cec810a-ee0f-11ec-8ea0-0242ac120002'` on DCT).

### 2.3.5 Environment representations

Environments within Delphix serve as a reference for the combination of a host and instance. This is coupled with the fact that environments can be leveraged by multiple engines at the same time and that engines often have a specific context to some of the elements that comprise an environment. For example, an environment could have both an Oracle and ASE instance installed and that Engine A leverages an Oracle-based workflow and Engine B leverages an ASE workflow. DCT will create two identifiers to represent the specific host and instance combinations. Thus, in DCT, Engine A will be connected to a different uniquely identified Environment than Engine B.

As mentioned earlier with Engine-to-DCT API mapping, DCT aims to simplify the user experience with Delphix APIs by combining different Continuous Data endpoints into a simplified DCT API. The Environment API does this by combining environment, repository, and host endpoints so that writing queries against Delphix data is a much simpler process. One example would be identifying all environments that have a compatible Oracle home for provisioning:

```
repositories CONTAINS { database_type EQ 'Oracle' and allow_provisioning EQ true AND
version CONTAINS '19.2.3'}
```

### 2.3.6 Supported data sources/configurations

DCT is compatible with all Delphix-supported data sources and configurations.

### 2.3.7 Process feedback

Whenever a DCT request completes, it will return a JOB ID as its response. This Job ID can be used in conjunction with the jobs endpoint to query the operation status.

# 3 Supported versions

Data Control Tower has minimum engine versions that are actively tested against to ensure optimal interoperability. Please ensure that all connected engines meet the version requirements:

| **Delphix Engine** | **Version** |
|---|---|
| Continuous Data | 6.0.0.1 or higher |
| Continuous Compliance | 6.0.13.0 or higher |

# 4 Installation and setup

> ⚠ Docker Compose should only be used to deploy DCT in an evaluation/testing capacity, and production DCT workloads in Docker Compose are not fully supported. Installations starting on Docker Compose may be migrated to Kubernetes or OpenShift by using the steps in the technical documentation. In-place upgrades from Docker Compose to Kubernetes or OpenShift are not supported.

## 4.1 Hardware requirements

The hardware requirements for Data Control Tower are listed below. In addition to these requirements, inbound port 443 must be open for API clients, and outbound port 443 to engines.

**CPU**: 4-Core
**Memory**: 2GB
**Storage**: 50GB
**Port**: 443

## 4.2 Installation requirements (Docker Compose)

DCT **requires** Docker and Docker Compose to run, thus, Linux versions and distributions that have been verified to work with Docker are supported. To see a list of supported distributions, please reference this Docker article[1].

This example uses a Docker installation[2] and is completed on an Ubuntu 20.04 VM.

To begin, uninstall any old versions of Docker.

```
sudo apt-get remove docker docker-engine docker.io containerd runc
```

Next, update the package lists and install Docker.

```
sudo apt-get update
sudo apt-get install docker.io
```

Last, install Docker Compose[3].

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.1/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

> Docker-Compose is packaged with Docker engine version 20.10.15 and up.

1 https://docs.docker.com/engine/install/#server
2 https://docs.docker.com/engine/install/
3 https://docs.docker.com/compose/install/

### 4.2.1 Running Docker as non-root (optional)

To avoid prefacing the Docker command with sudo, create a Unix group called docker and add users to it. When the Docker daemon starts, it creates a Unix socket accessible by members of the Docker group. See Docker Post Installation[4] documentation for details.

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

## 4.3 Unpack and install DCT

Once Docker and Docker Compose are installed, DCT can be installed. Begin by downloading the latest version of the tarball from the Delphix Download site[5]. Next, transfer the file to the Linux machine where Docker is installed. Run the following commands to extract the containers and load them into Docker:

```
tar -xzf delphix-dct*.tar.gz
for image in *.tar; do sudo docker load --input $image; done
```

## 4.4 Run DCT

To run DCT, navigate to the location of the extracted **docker-compose.yaml** file from the tarball and run the following command. Using `-d` in the command will start up the application in the background.

```
sudo docker-compose up -d
```

Running `docker ps` should show 9 containers up and running:

```
sudo docker ps
CONTAINER ID   IMAGE                                  COMMAND                  CREATED
STATUS         PORTS                     NAMES
75a9df0cae07   delphix-dct-proxy:6.0.0                "/sbin/tini -- /boot…"   7 seconds
ago   Up 4 seconds   0.0.0.0:443->8443/tcp   delphix-dct-proxy:3.0.0
a23f4fbe0220   delphix-dct-app:6.0.0                  "java -jar /opt/delp…"   7 seconds
ago   Up 5 seconds                           delphix-dct-app:6.0.0
96ba8018fa03   delphix-dct-data-library:6.0.0         "/usr/bin/tini -- ./…"   7 seconds
ago   Up 5 seconds                           delphix-dct-data-library:6.0.0
8e5b1e671acc   delphix-dct-jobs:6.0.0                 "/usr/bin/tini -- ./…"   7 seconds
ago   Up 5 seconds                           delphix-dct-jobs:6.0.0
96049058f025   delphix-dct-data-bookmarks:6.0.0       "/usr/bin/tini -- ./…"   7 seconds
ago   Up 5 seconds                           delphix-dct-data-bookmarks:6.0.0
```

4 https://docs.docker.com/engine/install/linux-postinstall/
5 https://download.delphix.com/folder

```
20d1782cb3bb   delphix-dct-ui:6.0.0                "node ./index.js"        7 seconds
ago   Up 5 seconds                              delphix-dct-ui:6.0.0
4fae43c79e8d   delphix-dct-virtualization:6.0.0   "/usr/bin/tini -- ./…"   7 seconds
ago   Up 5 seconds                              delphix-dct-virtualization:6.0.0
83d7d661d8a0   delphix-dct-graphql:6.0.0          "/bin/sh -c 'BASE_UR…"   7 seconds
ago   Up 6 seconds                              delphix-dct-graphql:6.0.0
3dded474e28b   delphix-dct-postgres:6.0.0         "docker-entrypoint.s…"   7 seconds
ago   Up 6 seconds   5432/tcp                   delphix-dct-postgres:6.0.0
```

# 5 Custom configuration

Docker Compose should only be used to deploy DCT in an evaluation/testing capacity.

## 5.1 Introduction

DCT was designed for users to configure Delphix applications in a way that would meet their security requirements, which handled with a custom configuration. This article provides background information on performing custom configurations, which are referenced throughout DCT articles and sections.

## 5.2 Bind mounts

Configuration of DCT is achieved through a combination of API calls and the use of Docker bind mounts[6]. A bind mount is a directory or file on the host machine that will be mounted inside the container. Changes made to the files on the host machine will be reflected inside the container. It does not matter where the files live on the host machine, but the files must be mounted to specific locations inside the container so that the application can find them.

The DCT and proxy containers can both be configured via separate bind mounted directories. Each container requires all configuration files to be mounted to the `/etc/config` directory inside the container. Therefore, it is recommended to create a directory for each container on the host machine to store all of the configuration files and mount them to `/etc/config`. This is done by editing the `docker-compose.yaml`. Under **proxy services**, add a **volumes** section if one does not already exist; this is used to mount the configuration directory on the host to `/etc/config`. For example, if `/my/proxy/config` is the directory on the host that contains the configuration files, then the relevant part of the compose file would look like this:

```
services:
    proxy:
        volumes:
            - /my/proxy/config:/etc/config
```

To change the configuration of the DCT container, make a similar change under its service section, the only difference being the directory on the host. After making this change, the application will need to be stopped and restarted.

The structure of `/my/proxy/config` will need to match the required layout in `/etc/config`. When each container starts, it will create default versions of each file and place them in the expected location. It is highly recommended to start from the default version of these files. For example, if `/my/proxy/config` is the bind mount directory on the host, it could be populated with all the default configuration files by running the following commands.

First, create an `nginx` directory inside `/my/proxy/config` on the host.

```
cd /my/proxy/config
mkdir nginx
```

6 https://docs.docker.com/storage/bind-mounts/

Find the **id** of the proxy container with **docker ps**. Look for the container with a **delphix-dct-proxy** image name. To determine the user and group ownership for any configuration files, start the containers and open a shell to the relevant one (nginx in this example), then examine the current user/group IDs associated with the files.

```
$ docker ps

CONTAINER ID   IMAGE                                COMMAND                  CREATED
STATUS         PORTS                                     NAMES
ac343412492a   delphix-dct-proxy:3.0.0   "/bootstrap.sh"          8 minutes ago   Up
8 minutes   0.0.0.0:443->443/tcp, :::443->443/tcp   dct-packaged_proxy_1
```

In the above example, ac343412492a is the **id**. Run the following command to copy the default files to the bind mount.

```
docker cp <container id>:/etc/config/nginx /my/proxy/config/nginx
```

One can always go back to the original configuration by removing the bind-mount and restarting the container or using docker cp as in the previous example to overwrite the custom files with the default versions.

# 6 Authentication

## 6.1 Introduction

DCT uses Nginx /OpenResty[7] as an HTTP server and a reverse proxy for the application. Using the default configuration, all connections to DCT are over HTTPS and require the user to authenticate. There are two supported methods for authentication; **API keys** and **OpenID Connect**. The Nginx/OpenResty configuration files can be edited via `/etc/config` bind mounts for the proxy container to customize the HTTP server and change options such as TLS versions.

## 6.2 API Keys

API keys are the default method to authenticate with DCT. This is done by including the key in the HTTP Authorization request header[8] with type **apk**. A cURL example using an example key of 1.0p9PMkZO4Hgy0ezwjhX0Fi4lEKrD4pflejgqjd0pfKtywlSWR9G0fIaWajuKcBT3 would appear as:

```
curl --header 'Authorization: apk
1.0p9PMkZO4Hgy0ezwjhX0Fi4lEKrD4pflejgqjd0pfKtywlSWR9G0fIaWajuKcBT3'
```

cURL (like web browsers and other HTTP clients) will not connect to DCT over HTTPS unless a valid TLS certificate has been configured for the Nginx server. If you haven't performed this configuration step[9] yet, and understand the risk, you may disable the check in the HTTP client. For instance this can done with cURL using the --insecure flag.

### 6.2.1 Bootstrap First API Key

There is a special process to bootstrap the creation of the first API key. This first API key should only be used to create another key and then promptly deleted, since the bootstrap API will appear in the logs. T his process can be repeated as many times as needed, for example, in a case where existing API keys are lost or have been deleted. It also means that the Linux users with permissions to edit the **docker-compose** file implicitly have the ability to get an API key at any time. There is no mechanism to lock this down after the first bootstrap key is created.

Begin by stopping the application with the following command:

```
sudo docker - compose stop
```

Once the application is stopped, edit the **docker-compose.yaml** file and modify the following lines to the DCT section, to set the *API_KEY_CREATE* to the string value *"true"* :

```
services:
  gateway:
    environment:
```

7 https://openresty.org/en
8 https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization
9 https://portal.document360.io/#replace-https-certificate-for-apigw

```
      API_KEY_CREATE: "true"
```

Start DCT again with `sudo docker-compose up` . You will see the following output in the logs for the *app* container (the key will be different from this example):

```
NEWLY GENERATED API KEY: 1.0p9PMkZO4Hgy0ezwjhX0Fi4lEKrD4pflejgqjd0pfKtywlSWR9G0fIaWaj
uKcBT3
```

Copy the API Key and shut down the DCT app. The API key can now be used to authenticate with DCT. Remember that the API Key value must be prefixed with **apk**. An example cURL command with the above API Key appears as follows:

```
curl --header 'Authorization: apk
1.0p9PMkZO4Hgy0ezwjhX0Fi4lEKrD4pflejgqjd0pfKtywlSWR9G0fIaWajuKcBT3'
```

Edit the **docker-compose.yaml** file to set the *API_KEY_CREATE* environment variable value back to "false" and restart DCT again with `sudo docker-compose up -d` .

### 6.2.2 Create and manage API Keys

The initial API key created should be used to create a new admin secure key. This is done by creating a new *Api Client* entity and setting the generate_api_key. The " name" attribute should be the desired name to uniquely identify the user of this key.

```
curl --location --request POST 'https://<hostname>/v2/management/api-clients' \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
--header 'Authorization: apk
1.0p9PMkZO4Hgy0ezwjhX0Fi4lEKrD4pflejgqjd0pfKtywlSWR9G0fIaWajuKcBT3' \
--data-raw '{
   "name": "secure-key",
   "generate_api_key": true
}'
```

A response should be received similar to the lines below:

```
{
   "api_key_id": 5,
   "token": "5.vCfC0MnpySYZLshuxap2aZ7xqBKAnQvV7hFnobe7xuNlHS9AF2NQnV9XXw4UyET6"
}
```

Now that the new and secure API key is created, the old one must be deleted for security reasons since the key appeared in the logs. To do this make the following request:

```
curl --location --request DELETE 'https://<hostname>/v2/management/api-clients/<id>'
 \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
--header 'Authorization: apk
5.vCfC0MnpySYZLshuxap2aZ7xqBKAnQvV7hFnobe7xuNlHS9AF2NQnV9XXw4UyET6'
```

The `id` referenced above is the numeric id of the API client. It is the integer before the period in the token. For example, the id of 1.0p9PMkZO4Hgy0ezwjhX0Fi4lEKrD4pflejgqjd0pfKtywlSWR9G0fIaWajuKcBT3 is 1.

Finally, to list all of the current API clients, make the following request:

```
curl --location --request GET 'https://<hostname>/v2/management/api-clients/' \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
--header 'Authorization: apk <your API key>'
```

## 6.3 OAuth 2.0

### 6.3.1 Enable OpenID connect

DCT supports OAuth2 authentication via the OpenID Connect Discovery[10] specification. To add support for OAuth 2.0, first configure Nginx to communicate with your OAuth2 server: a bind mount[11] must be used . The file needs to be named **default.conf** and appear at `config/nginx/conf.d/default.conf` related to the root of the bind mounted directory. See the bind mount section about how to use docker cp to begin with the default version of the file.

There are three important sections of the file that need to be updated. First, the **open_id_connect_enabled** variable must be set to true and API keys need to be disabled like so:

```
local api_keys_enabled = false
local open_id_connect_enabled = true
```

The other two important configuration options are the discovery URL of the OAuth2 server and the specific attribute names of the JWT, to provide a unique ID and name for the user.

```
-- OpenID Connect implementation
if open_id_connect_enabled then
  local opts = {
    -- Replace the discovery URL with the discovery endpoint of your own OAuth2
server.
    discovery = "https://delphix.okta.com/oauth2/default/.well-known/oauth-
authorization-server",
```

10 https://openid.net/specs/openid-connect-discovery-1_0.html
11 https://portal.document360.io/dct-2-0-0/docs/dct-2-0-0-0-custom-configuration#bind-mounts

```
    ssl_verify = "yes",
    accept_unsupported_alg = false,
    accept_none_alg = false,
    redirect_uri = "",
  }

  local jwt, err, token = require("resty.openidc").bearer_jwt_verify(opts)
  if err then
      ngx.header["X-jwt-error"] = err
      ngx.status = 401
      ngx.log(ngx.ERR, "Invalid token: " .. err)
      ngx.exit(ngx.HTTP_UNAUTHORIZED)
      return
  end

  -- Replace "sub" with the attribute which is meant to be used
  -- as client_id or client_name in the JWT.
  ngx.var.client_id = jwt.sub
  ngx.var.client_name = jwt.sub

end
```

Once requests are authenticated, they must be matched with an existing API client in DCT. To do so, one of the claims of the JWT (Json Web Token) must correspond to the *client_id* of an API client. For instance, imagine someone is using a JWT with a *sub* claim with value *abc123* and the configuration above, which extracts the *sub* claim out of the JWT and sets it as *client_id* . We can create a corresponding admin API client with the following request:

```
curl -k --location --request POST 'https://<hostname>/v2/management/api-clients' \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
--header 'Authorization: apk
1.0p9PMkZO4Hgy0ezwjhX0Fi4lEKrD4pflejgqjd0pfKtywlSWR9G0fIaWajuKcBT3' \
--data-raw '{
   "name": "oauth2-test-api-client",
   "api_client_id": "abc123",
   "generate_api_key": false
}'
```

Set the *name* to a logical name corresponding to the application or person using the JWT. *is_admin* denotes whether this API client has admin access to DCT. *api_client_id* must be set to the exact string value found in the JWT claim extracted in the Nginx config above. *generate_api_key* is disabled in this example as the API client will use exclusively OAuth2 (JWT) to authenticate and not API keys.

## 6.4 Replace HTTPS Certificate for APIGW

By default to enable HTTPS, DCT creates a unique self-signed certificate when starting up for the first time. To replace this certificate and private key, use a bind mount[12]. The configuration needs to be bind mounted to a `/etc/config/nginx/ssl` directory inside the container. The default location for the configuration is `/etc/config/nginx/ssl/ssl.conf` and therefore should be placed in `nginx/ssl/ssl.conf` - inside the configuration directory on the host. The default configuration can be retrieved by using docker cp[13]to copy `/etc/config/nginx/ssl/ssl.conf` out of the proxy container and used as a starting point.

To replace the certificate and corresponding private key used by Nginx, place the updated certificate at `nginx/ssl/nginx.crt` and the private key at `nginx/ssl/nginx.key` inside the bind mounted directory on the host. Doing this could require the assistance of someone from your security or IT departments. A new key pair (public and private key) will need to be created, in addition to a certificate signing request (CSR) for that key pair. IT should be able to determine the correct certificate authority (CA) to sign this CSR and produce the new certificate. The common name of the certificate should match the fully qualified-domain name (FQDN) of the host as well include the FQDN as a Subject Alternative Name (SAN). The supported versions of TLS can be changed by altering the ssl_protocols line and the list of ciphers by altering the ssl_ciphers line. After this is done, restart DCT application with docker-compose.

12 https://portal.document360.io/dct-2-0-0/docs/dct-2-0-0-0-custom-configuration#bind-mounts
13 https://docs.docker.com/engine/reference/commandline/cp

# 7 Engines: connecting/authenticating

## 7.1 Introduction

After DCT Authentication is complete, the HTTPS should be securely configured on DCT and able to be authenticated against. The next step is to register an engine with DCT so that it can fetch results. DCT connects to all engines over HTTPS, thus some configurations might be required to ensure it can communicate successfully.

## 7.2 Truststore for HTTPS

If the CA certificate that signed the engine's HTTPS certificate is not a trusted root CA certificate present in the JDK, then custom CA certificates can be provided to DCT. If these certificates are not provided, a secure HTTPS connection cannot be established and registering the engine will fail. The insecure_ssl engine registration parameter can be used to bypass the check, however, this should not be used unless the risks are understood.

Get the public certificate of the CA that signed the engine's HTTPS certificate in PEM format. IT team help may be required to get the correct certificates. Base64 encode the certificate with:

```
cat mycertfile.pem | base64 -w 0
```

Copy the Base64 encoded value from the previous step and configure in values.yaml file under truststoreCertificates section. e.g. section will look like this:

```
truststoreCertificates:
<certificate_name>.crt: <base64 encode certificate string value in single line>
```

**<certificate_name>** can be any logically valid string value for e.g. "engine".

All the certificates configured in truststoreCertificates section will be read and included in the trustStore which would be then used for SSL/TLS communication between DCT and Delphix Engine.

## 7.3 Authentication with engine

All authentication with the Delphix Engine is done with the username and password of a domain admin engine user. There are two methods of storing these credentials with DCT. They can either be stored and encrypted on DCT itself or retrieved from a password vault. We recommend fetching the credentials from a vault. Currently only the HashiCorp vault is supported.

## 7.4 HashiCorp vault

There are two high-level steps to configuring a HashiCorp vault. The first is to set up authentication with the vault and register the vault. The second is to tell DCT how to get the specific engine credentials needed from that registered vault. A single vault can be used for multiple different Delphix Engines.

### 7.4.1 Vault authentication and registration

First, DCT needs to be able to authenticate with the vault. DCT supports the Token[14], AppRole[15], and TLS Certificates[16] authentication methods. This is done by passing a command to the HashiCorp CLI[17]. It is recommended to first ensure that successful authentication is done and one can retrieve the credentials with the HashiCorp CLI directly to ensure the correct commands are passed to DCT.

Adding a vault to DCT is done through API calls to the /v2/management/vaults/hashicorp endpoint. All authentication methods requires the location of the vault is provided through the env_variables property in the POST body like so:

```
"env_variables": {
       "VAULT_ADDR": "https://10.119.132.40:8200"
   }
```

### 7.4.2 Token

To use the token authentication method, this needs to be included as part of the env_variables field. The full example to register the vault would appear as:

```
curl --location --request POST 'https://<hostname>/v2/management/vaults/hashicorp' \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
--header 'Authorization: apk <your API key>' \
--data-raw '{
   "env_variables": {
       "VAULT_TOKEN": "<your token>"
       "VAULT_ADDR": "https://10.119.132.40:8200"
   }
}'
```

A response should be received similar to the lines below:

```
{
   "id": 2,
   "env_variables": {
       "VAULT_TOKEN": "<your token>"
       "VAULT_ADDR": "https://10.119.132.40:8200"
   }
}
```

Note the id of the vault, this will be needed in the next step to register the engine.

14 https://www.vaultproject.io/docs/auth/token
15 https://www.vaultproject.io/docs/auth/approle
16 https://www.vaultproject.io/docs/auth/cert
17 https://www.vaultproject.io/docs/commands

### 7.4.3 AppRole

To use the AppRole authentication method, this needs to be included as part the login_command_args field, as shown below.

```
"login_command_args":
       [ "write", "auth/approle/login", "role_id=1", "secret_id=123"]
```

The full example to register the vault would appear as:

```
curl --location --request POST 'https://<hostname>/v2/management/vaults/hashicorp' \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
--header 'Authorization: apk <your API key>' \
--data-raw '{
   "env_variables": {
       "VAULT_ADDR": "https://10.119.132.40:8200"
   },
   "login_command_args":
       [ "write", "auth/approle/login", "role_id=1", "secret_id=123"]
}'
```

A response should be received similar to the lines below:

```
{
   "id": 2,
   "env_variables": {
       "VAULT_TOKEN": "<your token>"
       "VAULT_ADDR": "https://10.119.132.40:8200"
   }
}
```

## 7.5 TLS certificates

The configuration of mutual TLS authentication requires an additional step. This feature currently is NOT supported for Kubernetes deployment of DCT. This will be covered in later releases.

### 7.5.1 Retrieving engine credentials

Once DCT can authenticate with the vault, it needs to know how to fetch the relevant engine credentials. When registering an engine, the user will need to provide the HashiCorp CLI commands through the `hashicorp_vault_username_command_args` and `hashicorp_vault_password_command_args` parameters.

The relevant part of the engine registration payload will look like the following:

```
'{
    "hashicorp_vault_id": 1
    "hashicorp_vault_username_command_args": ["kv", "get", "-field=username",  "kv-
v2/delphix-engine-secrets/engineUser"]
,
    "hashicorp_vault_password_command_args": ["kv", "get", "-field=password",  "kv-
v2/delphix-engine-secrets/engineUser"]
}'
```

The hashicorp_vault_id will be the ID that was returned as part of the previous step. Note that the exact paths to fetch the username and password will vary depending on the exact configuration of the vault.

# 8 Backup DCT

This article discusses how to backup DCT. The data that needs to be backed up is the **Docker volumes** used by the DCT container, gwdatabase container, and the **configuration directories** on the host that are bind mounted to the containers.

The Docker volumes named **orbital-api-gateway_gateway-data** and **orbital-api-gateway_gwdatabase-data** should be backed up to prevent data loss. This Docker article[18] explains how to backup a data volume.

The bind mount directories containing the configuration files are standard directories that can be backed up as desired. A simple approach would be to create a tar file of the contents. If /my/config is the bind mount directory on the host, then this can be done with the following command: `tar -czf gateway-backup.tgz /my/config`

18 https://docs.docker.com/storage/volumes/#backup-restore-or-migrate-data-volumes

# 9 DCT Logs

DCT leverages the Docker logging[19] infrastructure. All containers log to stdout and stderr so that their logs are processed by Docker. Docker supports logging drivers for a variety of tools such as fluentd, Amazon CloudWatch, and Splunk to name a few. See docker documentation here[20] on how to configure them. These changes will need to be made to the docker-composee.yaml file. This link[21] explains how to alter the compose file to adjust the logging driver. For example, if you want to use syslog for the proxy container then it would look like this:

```
services:
    proxy:
        logging:
            driver: syslog
            options:
                syslog-address: "tcp://192.123.1.23:123"
```

19 https://docs.docker.com/config/containers/logging
20 https://docs.docker.com/config/containers/logging/configure
21 https://docs.docker.com/compose/compose-file/compose-file-v3/#logging

# 10 Developer resources

## 10.1 API requests and reporting

### 10.1.1 Introduction

This article showcases example requests to the various data APIs supported by DCT.

DCT provides interactive API documentation that allows users to experiment with the APIs in their web browser. The interactive API documentation can be accessed by entering the hostname for DCT and the **/api** path into a browser's address bar. For example, if DCT is running on host gateway.example.com[22], then enter https://gateway.example.com/api into the browser's address bar.

To simplify development, Python and Go programming libraries are available. The **Python** bindings can be found on PyPi here[23]. The latest version can be installed with the following command:

```
pip install delphix-dct
```

The **Go** bindings can be found on go.dev here[24].

### 10.1.2 Engines

This section showcases some examples of querying the Engines endpoint for information about connected Delphix Virtualization Engines. These examples leverage the generated Python bindings:

```
import delphix.api.gateway
import delphix.api.gateway.configuration
import delphix.api.gateway.api.management_api
cfg = delphix.api.gateway.configuration.Configuration()
cfg.host = "https://localhost/v2"

# For example purposes

cfg.verify_ssl = False

# Replace the string with your own API key

cfg.api_key['ApiKeyAuth'] = 'apk 3.tEd4DXFce'
api_client = delphix.api.gateway.ApiClient(configuration=cfg)
engines_api =  delphix.api.gateway.api.management_api.ManagementApi(api_client)
print(engines_api.get_registered_engines())
```

The result should appear similar to the following:

22 http://gateway.example.com/
23 https://pypi.org/project/delphix-dct-api/
24 https://pkg.go.dev/github.com/delphix/dct-sdk-go

```
{'items': [{'connection_status': 'ONLINE',
            'cpu_core_count': 2,
            'data_storage_capacity': 23404216320,
            'data_storage_used': 11589626880,
            'hostname': 'avm.delphix.com',
            'id': 1,
            'insecure_ssl': True,
            'memory_size': 8589934592,
            'name': 'vmname',
            'password': '******',
            'status': 'CREATED',
            'tags': [],
            'type': 'UNSET',
            'unsafe_ssl_hostname_check': False,
            'username': 'admin',
            'uuid': 'ec2fbfea-928b-07f8-94c4-29fea614624f',
            'version': '6.1.0.0'}]}
```

## 10.2 API references

To access the API list for DCT version 2.2.0, click the link below and the .html file with the API content will download.


(see page 28)