Introduction

Docker Enterprise is the enterprise container platform from Docker Inc to be used across the entire software supply chain. It is a fully-integrated solution for container-based application development, deployment, and management. With integrated end-to-end security, Docker Enterprise enables application portability by abstracting your infrastructure so that applications can move seamlessly from development to production.

What You Will Learn

This reference architecture describes a standard, production grade, Docker Enterprise deployment. It also details the components of Docker Enterprise, how they work, how to automate deployments, how to manage users and teams, how to provide high availability for the platform, and how to manage the infrastructure.

Some environment-specific configuration details are not be provided. For instance, load balancers vary greatly between cloud platforms and on-premises infrastructure platform. For these types of components, general guidelines to environment-specific resources are provided.

Understanding Docker Components

From development to production, Docker Enterprise provides a seamless platform for containerized applications both on-premises and in the cloud. Docker Enterprise include the following components:

- Docker Engine - Enterprise, the commercially supported Docker container runtime
- Universal Control Plane (UCP), the web-based, unified cluster and application management solution
- Docker Kubernetes Service (DKS), a certified Kubernetes distribution (https://www.cncf.io/certification/software-conformance/#logos) with 'sensible secure defaults' out-of-the-box.
- Docker Trusted Registry (DTR), a resilient and secure image management repository
- Docker Desktop Enterprise (DDE), an enterprise friendly, supported version of the popular Docker Desktop application with an extended feature set.

Together they provide an integrated solution with the following design goals:

- Agility — the Docker API is used to interface with the platform so that operational features do not slow down application delivery
- Portability — the platform abstracts details of the infrastructure for applications
- Control — the environment is secure by default, provides robust access control, and logging of all operations

To achieve these goals the platform must be resilient and highly available. This reference architecture demonstrates this robust configuration.

Docker Engine - Enterprise

Docker Engine - Enterprise is responsible for container-level operations, interaction with the OS, providing the Docker API, and running the Swarm cluster. The Engine is also the integration point for infrastructure, including the OS resources, networking, and storage.

Universal Control Plane
UCP extends Docker Engine - Enterprise by providing an integrated application management platform. It is both the main interaction point for users and the integration point for applications. UCP runs an agent on all nodes in the cluster to monitor them and a set of services on the controller nodes. This includes identity services to manage users, Certificate Authorities (CA) for user and cluster PKI, the main controller providing the Web UI and API, data stores for UCP state, and a Classic Swarm service for backward compatibility.

**Docker Kubernetes Service**

At Docker, we recognize that much of Kubernetes' perceived complexity stems from a lack of intuitive security and manageable configurations that most enterprises expect and require for production-grade software. Docker Kubernetes Service (DKS) is a certified Kubernetes distribution that is included with Docker Enterprise and is designed to solve this fundamental challenge. It is the only offering that integrates Kubernetes from the developer desktop to production servers. Simply put, DKS makes Kubernetes easy to use and more secure for the entire organization.

DKS comes hardened out-of-the-box with 'sensible secure defaults' that enterprises expect and require for production-grade deployments. These include out-of-the-box configurations for security, encryption, access control, and lifecycle management — all without having to become a Kubernetes expert. DKS also allows organizations to integrate their existing LDAP and SAML-based authentication solutions with Kubernetes RBAC for simple multi-tenancy.

**Docker Trusted Registry**

DTR is an application managed by, and integrated with UCP, that provides Docker images distribution and security services. DTR uses UCP's identity services to provide Single Sign-On (SSO), and establish a mutual trust to integrate with its PKI. It runs as a set of services on one or several replicas: the registry to store and distribute images, an image signing service, a Web UI, an API, and data stores for image metadata and DTR state.

**Docker Desktop Enterprise**

Docker Desktop Enterprise (DDE) is a desktop offering that is the easiest, fastest and most secure way to create and deliver production-ready containerized applications. Developers can work with frameworks and languages of their choice, while IT can securely configure, deploy and manage development environments that align to corporate standards and practices. This enables organizations to rapidly deliver containerized applications from development to production.

DDE provides a secure way to configure, deploy and manage developer environments while enforcing safe development standards that align to corporate policies and practices. IT teams and application architects can present developers with application templates designed specifically for their team, to bootstrap and standardize the development process and provide a consistent environment all the way to production.

IT desktop admins can securely deploy and manage Docker Desktop Enterprise across distributed development teams with their preferred endpoint management tools using standard MSI and PKG files. No manual intervention or extra configuration from developers is required and desktop administrators can enable or disable particular settings within Docker Desktop Enterprise to meet corporate standards and provide the best developer experience.

**Docker Swarm**

To provide a seamless cluster based on a number of nodes, Docker Enterprise relies on Docker swarm (https://docs.docker.com/engine/swarm/key-concepts/) capability. Docker Swarm divides nodes between workers, nodes running application workloads defined as services, and managers, nodes in charge of maintaining desired state, managing the cluster's internal PKI, and providing an API. Managers can also run workloads. In a Docker Enterprise environment managers run UCP processes and should not run anything else.
The Swarm service model provides a declarative desired state configuration for workloads, scalable to a number of tasks (the service's containers), accessible through a stable resolvable name, and optionally exposing an end-point. Exposed services are accessible from any node on a cluster-wide reserved port, reaching tasks through the routing mesh, a fast routing layer leveraging native high-performance switching in the Linux kernel. This set of features enables routing, internal and external discovery for services, load balancing and enhanced Layer 7 ingress routing based on UCP's Interlock component.

**Standard Deployment Architecture**

This section demonstrates a standard, production grade architecture for Docker Enterprise using 10 nodes: 3 UCP managers, 3 workers for DTR, and 4 worker nodes for application workloads. The number of worker nodes is arbitrary, most environments will have more or fewer depending on the needs of the applications hosted. The number or capacity of the worker nodes does not change the architecture or the cluster configuration.

Access to the environment is done through 4 Load Balancers (or 4 load balancer virtual hosts) with corresponding DNS entries for the UCP managers, the DTR replicas, the Kubernetes ingress controller, and Swarm layer 7 routing.

DTR replicas use shared storage (NFS, Cloud, etc.) for images.

**Node Size**

A node is a machine in the cluster (virtual or physical) with Docker Engine running on it. When adding each node to the cluster, it is assigned a role: UCP manager, DTR replicas, or worker node. Typically, only worker nodes are allowed to run application workloads.

To decide what size the node should be in terms of CPU, RAM, and storage resources, consider the following:

1. All UCP manager and worker nodes should fulfill the minimal requirements, the recommended requirements are preferred for a production system, in the [UCP system requirements](https://docs.docker.com/ee/ucp/admin/install/system-requirements/) document.
2. All DTR replica nodes should fulfill the minimal requirements, the recommended requirements are preferred for a production system, in the [DTR system requirements](https://docs.docker.com/ee/dtr/admin/install/system-requirements/) document.
3. Ideal worker node size will vary based on your workloads, so it is impossible to define a universal standard size.
4. Other considerations like target density (average number of containers per node), whether one standard node type or several are preferred, and other operational considerations might also influence sizing.

If possible, node size should be determined by experimentation and testing actual workloads, and they should be refined iteratively. A good starting point is to select a standard or default machine type in your environment and use this size only. If your standard machine type provides more resources than the UCP Controllers need, it makes sense to have a smaller node size for these. Whatever the starting choice, it is important to monitor resource usage and cost to improve the model.

Two example scenarios:

- **Homogeneous:**
  - 4 vCPU
  - 16 GB RAM
  - 50 GB storage
- **Node Type:**
UCP Manager
- 4 vCPU
- 16 GB RAM
- 100 GB storage

DTR Replica
- 4 vCPU
- 32 GB RAM
- 100 GB storage

Worker
- 4 vCPU
- 64 GB RAM
- 100 GB storage

Depending on your OS of choice, storage configuration for Docker Engine might require some planning. Refer to the Docker Enterprise Compatibility Matrix (https://success.docker.com/article/compatibility-matrix) to see what storage drivers are supported for your host OS.

**Load Balancer Configuration**

Load balancers configuration should be done before installation, including the creation of DNS entries. Most load balancers should work with Docker Enterprise. The only requirements are TCP passthrough and the ability to do health checks on an HTTPS endpoint.

In our example architecture, the three UCP managers ensure UCP resiliency in case of node failure or reconfiguration. Access to UCP through the GUI or API is always done using TLS. The load balancer is configured for TCP pass-through on port 443, using a custom HTTPS health check at https://<UCP FQDN>/_ping.

Be sure to create a DNS entry for the UCP host such as ucp.example.com and point it to the load balancer.

The setup for the three DTR replicas is similar to setting up UCP. Again, use TCP passthrough to port 443 on the nodes. The HTTPS health check is also similar to UCP at https://<DTR FQDN>/_ping.

Create a DNS entry for the DTR host such as dtr.example.com and point it to the load balancer. It is important to keep it as concise as possible because it will be part of the full name of images. For example, user_a's webserver image will be named dtr.example.com/user_a/webserver:<tag>.

The Swarm application load balancer provides access to an application's HTTP endpoints exposed through UCP's Layer 7 Routing (Interlock). Layer 7 Routing provides a reverse-proxy to map domain names to services that expose ports. As an example, the voting application exposes the vote service's port 80. Interlock can be leveraged to map http://vote.apps.example.com to this port, and the application LB itself maps *.apps.example.com to nodes in the cluster.

For Kubernetes applications as well, a similar approach is used via an ingress controller which provides Layer 7 / proxy capabilities.

For more details on load balancing Swarm and Kubernetes applications on UCP, see the Layer 7 routing (https://docs.docker.com/ee/ucp/interlock/) documentation and UCP Service Discovery and Load Balancing (https://success.docker.com/article/ucp-service-discovery#supportforkubernetesindockeree) reference architecture.

**DTR Storage**
DTR usually needs to store a large number of images. It uses external storage (NFS, Cloud, etc.), not local node storage so that it can be shared between DTR replicas. The DTR replicates metadata and configuration information between replicas, but not image layers themselves. To determine storage size, start with the size of the existing images used in the environment and increase from there.

As long as it is compatible with DTR, it is a good option to use an existing storage solution in your environment. That way image storage can benefit from existing operational experience. If opting for a new solution, consider using object storage, which maps more closely to image registry operations.

Refer to An Introduction to Storage Solutions for Docker CaaS (https://success.docker.com/article/an-introduction-to-storage-solutions-for-docker-caas) for more information about selecting storage solutions.

Recommendations for the Docker Enterprise Installation

This section details the installation process for the architecture and provide a checklist. It is not a substitute for the documentation, which provides more details and is authoritative in any case. The goal is to help you define a repeatable (and ideally automated) process to deploy, configure, upgrade and expand your Docker Enterprise environment.

The three main stages of a Docker Enterprise installation are as follows:

1. Deploying and configuring the infrastructure (hosts, network, storage)
2. Installing and configuring the Docker Engine, running as an application on the hosts
3. Installing and configuring UCP and DTR, delivered as Docker containers running on the hosts

Infrastructure Considerations

The installation documentation (https://docs.docker.com/ee/end-to-end-install/) details infrastructure requirements for Docker Enterprise. It is recommended to use existing or platform specific tools in your environment to provide standardized and repeatable configuration for infrastructure components.

Network

Docker components need to communicate over the network, and the systems requirements documentation lists the ports used (https://docs.docker.com/ee/ucp/admin/install/system-requirements/#ports-used) for communication. Misconfiguration of the cluster's internal network can lead to issues that might be difficult to track down. It is better to start with a relatively simple environment. This reference architecture assumes a single subnet for all nodes and the default settings (https://docs.docker.com/ee/ucp/admin/install/plan-installation/#avoid-ip-range-conflicts) for all other configuration.

To get more details and evaluate options, consult the Designing Scalable, Portable Docker Container Networks (https://success.docker.com/article/networking) reference architecture.

Firewall

Access to Docker Enterprise is done using port 443 and 6443. This makes external firewall configuration simple. In most cases you only need to open ports 443 and 6443. Access to applications is through a load balancer using HTTPS. If you expose other TCP services to the outside world, open those ports on the firewall. As explained in the previous section, several ports need to be open for communication inside the cluster. If you have a firewall between some nodes in the cluster, for example, to separate manager from worker nodes, open the relevant port there as well.

For a full list of ports used see the UCP System Requirements (https://docs.docker.com/ee/ucp/admin/install/system-requirements/#ports-used) and DTR System Requirements (https://docs.docker.com/ee/dtr/admin/install/system-requirements/#ports-used) documentation.
If encrypted overlay networks are used within the applications, then ESP (Encapsulating Security Payload) or IP Protocol 50 traffic should also be allowed. ESP is not based on TCP or UDP protocols, and it will be used for end to end encapsulation of security payloads / data.

**Load Balancers**

Load balancers are detailed in the previous section. They must be in place before installation and must be provisioned with the domain names. External (load balancer) domain names are used for HA and also for TLS certificates. Having everything in place prior to installation simplifies the process as it avoids the need to reconfigure components after the installation process.

Refer to the Load Balancer Configuration (https://success.docker.com/article/docker-enterprise-best-practices#loadbalancerconfiguration) section for more details.

**Shared Storage**

DTR shared storage, used for images in the registry, must be ready and accessible from the DTR nodes. Test it works using a command line client to avoid having to debug DTR storage configuration.

**Host Configuration**

Host configuration varies based on the OS and existing configuration standards, but there are some important steps that must be followed after OS installation:

1. **Clock synchronization** using NTP or similar service. Clock skew can produce hard to debug errors, especially where the Raft algorithm is used (UCP and the DTR).
2. **Static IPs** are required for all hosts.
3. **Hostnames** are used for node identification in the cluster. The hostname must be set in an non-ephemeral way.
4. **Host firewalls** must allow intra-cluster traffic on all the ports specified in the installation docs.
5. **Storage Driver** prerequisite configuration must be completed if required for the selected driver.

**Docker Engine - Enterprise Installation Considerations**

Detailed instructions for the Docker Engine - Enterprise installation are available on the documentation site (https://docs.docker.com/ee/supported-platforms/). To install on nodes that do not have internet access, add the package to your internal package repository or follow the *install from package* section of the document for your OS. After installing the package, make sure the `docker` service is configured to start on system boot.

The best way to change parameters for Docker Engine - Enterprise is to use the `daemon.json` configuration file. This ensures that the configuration can be reused across different systems and OS in a consistent way. See the `dockerd` documentation (https://docs.docker.com/engine/reference/commandline/dockerd/) for a full list of options (https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-configuration-file) for the `daemon.json` configuration file.

Make sure the engine is configured correctly by starting the `docker` service and verifying the parameters with `docker info`.

**UCP Installation Considerations**

The UCP installer creates a functional cluster from a set of machines running Docker Engine - Enterprise. That includes creating a Swarm cluster and installing the UCP controllers. The default installation mode as described in the Install UCP for production (https://docs.docker.com/ee/ucp/admin/install/) document is interactive.
To perform fully-automated, repeatable deployments, provide more information to the installer. The full list of install parameters is provided in the docker/ucp install (https://docs.docker.com/reference/ucp/3.2/cli/) documentation.

## Adding Nodes

Once the installation has finished for the first manager node, in order to enable HA, two additional managers must be installed by joining them to the cluster. UCP configures a full replica on each manager node in the cluster, so the only command needed on the other managers is a `docker swarm join` with the manager token. The exact command can be obtained by running `docker swarm join-token manager` on the first manager.

To join the worker nodes, the equivalent command can be obtained with `docker swarm join-token worker` on any manager:

```
docker swarm join-token worker
```

To add a worker to this swarm, run the following command (an example):

```
docker swarm join
--token SWMTKN-1-00gqkzjo87dxcxb53qs4brml51vm6ca2e8fjnd6dds8lyn9ng1-092vhgjxz3jixvf081sdge3p
192.168.0.2:2377
```

To make sure everything is running correctly, log into UCP at https://ucp.example.com.

## DTR Installation Considerations

Installation of DTR is similar to that of UCP. Install and configure one node, and then join replicas to form a full, highly-available setup. For installation of the first instance as well as the replicas, point the installer to the node in the cluster it will install on.

Certificates and image storage must be configured after installation. Once shared storage is configured, the two replicas can be added with the `join` command.

## Validating the Deployment

When installation of everything has finished, tests can be performed to validate the deployment. Disable scheduling of workloads on UCP manager nodes and the DTR nodes.

Basic tests to consider:

1. Log in through https://ucp.example.com as well as directly to a manager node, eg. https://manager1.example.com. Make sure the cluster and all nodes are healthy.
2. Test that you can deploy an application following the example in the documentation (https://docs.docker.com/ee/ucp/swarm/).
3. Test that users can download a bundle and access the cluster via CLI (https://docs.docker.com/ee/ucp/user-access/cli/). Test that they can use compose (https://docs.docker.com/compose/overview/).
4. Test DTR with a full image workflow. Make sure storage is appropriately configured and images are stored in the right place.

Consider building a standard automated test suite to validate new environments and updates. Just testing standard functionality should hit most configuration issues. Make sure you run these tests with a non-admin user, the test user should have similar rights as users of the platform. Measuring time taken by each test can also pinpoint issues with underlying infrastructure configuration. Fully deploying an actual application from your organization should be part of this test suite.
High Availability in Docker Enterprise

In a production environment, it is vital that critical services have minimal downtime. It is important to understand how high availability (HA) is achieved in UCP and DTR, and what to do when it fails. UCP and DTR use the same principles to provide HA, but UCP is more directly tied to Swarm's features. The general principle is to have core services replicated in a cluster, which allows another node to take over when one fails. Load balancers make that transparent to the user by providing a stable hostname, independent of the actual node processing the request. It is the underlying clustering mechanism that provides HA.

Swarm

The foundation of UCP HA is provided by Swarm, the clustering functionality of Docker Engine. As detailed in the Docker Engine documentation (https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/), there are two algorithms involved in managing a Swarm cluster: a Gossip protocol for worker nodes and the Raft consensus algorithm for managers. Gossip protocols are eventually consistent, which means that different parts of the cluster might have different versions of a value while new information spreads in the cluster (they are also called epidemic protocols because information spreads like a virus). This allows for very large scale cluster because it is not necessary to wait for the whole cluster to agree on a value, while still allowing fast propagation of information to reach consistency in an acceptable time. Managers handle tasks that need to be based on highly consistent information because they need to make decisions based on global cluster and services state.

In practice, high consistency can be difficult to achieve without impeding availability because each write needs to be acknowledged by all participants, and a participant being unavailable or slow to respond will impact the whole cluster. This is explained by the CAP Theorem (https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/), which (to simplify) states that in the presence of partitions (P) in a distributed system, we have to choose between consistency (C) or availability (A). Consensus algorithms like Raft address this trade-off using a quorum: if a majority of participant agree on a value, it is good enough, the minority participant eventually get the new value. That means that a write needs only acknowledgement from 2 out of 3, 3 out of 5, or 4 out of 7 nodes.

Because of the way consensus works, an odd number of nodes is recommended when configuring Swarm. With 3 manager nodes in Swarm, the cluster can temporarily lose 1 and still have a functional cluster, with 5 you can lose 2, and so on. Conversely, you need 2 managers to acknowledge a write in a 3 manager cluster, but 3 with 5 managers, so more managers do not provide more performance or scalability — you are actually replicating more data. Having 4 managers does not add any benefits since you still can only lose 1 (majority is 3), and more data is replicated than with just 3. In practice, it is more fragile.

If you have 3 managers and lose 2, your cluster is non-functional. Existing services and containers keep running, but new requests are not processed. A single remaining manager in a cluster does not "switch" to single manager mode. It is just a minority node. You also cannot just promote worker nodes to manager to regain quorum. The failed nodes are still members of the consensus group and need to come back online.

UCP

UCP runs a global service across all cluster nodes called ucp-agent. This agent installs a UCP controller on all Swarm manager nodes. There is a one-to-one correspondence between Swarm managers and UCP controllers, but they have different roles. Using its agent, UCP relies on Swarm for HA, but it also includes replicated data stores that rely on their own raft consensus groups that are distinct from Swarm: ucp-auth-store, a replicated database for identity management data, and ucp-kv, a replicated key-value store for UCP configuration data.

DTR

The DTR has a replication model that is similar to how UCP works, but it does not synchronize with Swarm. It has one replicated component, its datastore, which might also have a lot of state to replicate at one time. It relies on raft consensus.
Both UCP controllers and DTR replicas may have a lot more state to replicate when (re)joining the cluster. Some reconfiguration operations can make a cluster member temporary unavailable. With 3 members, it is good practice to wait for the one you reconfigured to get back in sync before reconfiguring a second one, or they could lose quorum. Temporary losses in quorum are easily recoverable, but it still means the cluster is in an unhealthy state. Monitoring the state of controllers to ensure the cluster does not stay in that state is critical.

Backup and Restore

The HA setup using multiple nodes works well to provide continuous availability in the case of temporary failure, including planned node downtime for maintenance. For other cases, including the loss of the full cluster, permanent loss of quorum, and data loss due to storage faults, restoring from backup is necessary.

UCP Backup

A backup of UCP is obtained by running the `docker/ucp backup` command on a manager node. It stops the UCP containers on the node and performs a full backup of the configuration and state of UCP. Some of this information is sensitive, therefore it is recommended to use the `--passphrase` option to encrypt the backup. The backup also includes information about organizations, teams and users used by UCP as well as DTR. It is highly recommended to schedule regular backups. Here is an example showing how to run the `docker/ucp backup` command without user interaction:

```
UCPID=$(docker run --rm -i --name ucp -v /var/run/docker.sock:/var/run/docker.sock docker/ucp id)
docker run --rm -i --name ucp -v /var/run/docker.sock:/var/run/docker.sock docker/ucp backup --id $UCPID --passphrase "secret" > /tmp/backup.tar
```

There are two ways to use the backup: - To restore a controller using the `docker/ucp restore` command (only the backup from that controller can be used) - To install a new cluster using the `docker/ucp install --from-backup` command (preserves users and configuration)

DTR Backup

A DTR backup includes configuration information, images metadata, and certificates. Images themselves need to be backed up separately directly from storage. Remember that users and organizations within DTR are managed and backed up by UCP.

The backup can only be used to create a new DTR, using the `docker/dtr restore` command.

Identity Management

Accessing resources (images, containers, volumes, networks etc) and functionality within the components of Docker Enterprise (UCP & DTR) require at a minimum, an account and a corresponding password to be accessed. Accounts within Docker Enterprise are identities stored within an internal database, but the source of creating those accounts and the associated access control can be manual (managed or internal) or external through a connection to a directory server (LDAP) or Active Directory (AD). Managing the authorization for these accounts is an extension of coarse and fine grained permissions that are described in the sections below.

RBAC and Managing Access to Resources

UCP provides powerful role based access control features which can be seamlessly integrated with enterprise identity management tool sets and address enterprise security requirements. Besides facilitating both coarse-grained and fine-grained security access controls, this feature can be used as an enabler of multi-tenancy
within a single UCP cluster sharing a wide range of resources grouped into collections.

Access permissions in UCP are managed through grants of roles to subjects over collections of those resources. Access permissions are what define what a user can or cannot do within the system.

The default roles in UCP are None, View Only, Restricted Control, Scheduler, and Full Control. The description about these roles and how they relate to each other are detailed in the Reference Architecture Securing Docker Enterprise and Security Best Practices (https://success.docker.com/article/security-best-practices). Each of these roles have a set of operations that define the permissions associated with the role. Additional custom roles can be defined by combining a unique set of permissions. Custom roles can be leveraged to accommodate fine-grained access control as required for certain organizations and security controls.

Subjects are individual users or teams within an organization. Teams are typically backed by an LDAP/AD group or search filter. It is also possible to add users manually. But it is not possible to have a hybrid composition of users. In other words, the list of users within a team should be derived from a directory server (e.g. AD) or should be added manually, not both.

Collections are groupings of objects within UCP. A collection can be made up of one or many of nodes, stacks, containers, services, volumes, networks, secrets, or configs — or it can hold other collections. To associate a node or a stack or any resource with a collection, that resource should share the label com.docker.ucp.access.label with the collection. A resource can be associated with zero or multiple collections, and a collection can have zero or multiple resources or other child collections in it. Collections within collections allow the structuring of resource objects in a hierarchical nature and can significantly simplify access control. Access provided at a top level collection is inherited by all its children, including any child collections.

Consider a very simple use case for this approach. Suppose you define a top level collection called Prod and additional child collections corresponding to each application within Prod. These child collections contain the actual resource objects for the application like stacks, services, containers, volumes, networks, secrets, etc. Now suppose that all members of the IT Operations team require access to all Prod resources. With this setup, even if there are a high number of applications (and by extension, child collections within the Prod collection), the team IT Operations within UCP can be granted access to the Full Control role over the Prod collection alone. The access trickles down to every collection contained within the Prod collection. At the same time, members of a specific application development team can be provided fine-grained access to just the application collection. This model implements a traditional Role Based Access Control (RBAC), where the teams are assigned roles over specific collections of resources.

**Managed (Internal) Authentication**

Managed mode for authentication and authorization is the default mode in Docker Enterprise. In this mode, accounts are directly created using the Docker Enterprise API. User accounts can be created manually by accessing the User Management —> Users —> Create User form in the UCP UI. Accounts can also be created and managed in an automated fashion by making HTTP requests to the authentication and authorization RESTful service known as eNZi.

User management using the "Managed" mode is recommended only for demo purposes or where the number of users needing to access Docker Enterprise is very small.

Pros:
- Easy and quick to setup
- Simple to troubleshoot
- Appropriate for a small set of users with static roles
- Managed without leaving the UCP interface

Cons:
- User Account management gets cumbersome for larger numbers or when roles need to be managed for
several applications.

- All lifecycle changes such as adding / removing permissions of users need to be accomplished manually user by user.
- Users must be deleted manually, meaning access may not get cleaned up quickly, making the system less secure.
- Sophisticated setups of integrating application creation and deployment through LDAP or external systems is not possible.

**LDAP / AD Integration**

The LDAP method of user account authentication can be turned on to manage user access. As the name suggests, this mode enables automatic synchronization of user accounts from a directory server such as Active Directory or OpenLDAP.

This method is particularly applicable for enterprise use cases where organizations have a large set of users, typically maintained in a centralized identity store that manages both authentication and authorization. Most of these stores are based on a directory server such as Microsoft's Active Directory or a system that supports the LDAP protocol. Additionally, such enterprises already have mature processes for on-boarding, off-boarding, and managing the lifecycle of changes to user and system accounts. All these can be leveraged to provide a seamless and efficient process of access control within Docker Enterprise.

**Pros:**

- Ability to leverage already-established access control processes to grant and revoke permissions
- Ability to continue managing users and permissions from a centralized system based on LDAP
- Ability to take advantage of self-cleaning nature of this mode, where non-existent LDAP users are automatically removed from Docker Enterprise on the next sync, thereby increasing security
- Ability to configure complex, upstream systems such as flat files, database tables using an LDAP proxy, and the automatic time-based de-provisioning of access through AD/LDAP groups

**Cons:**

- Increased complexity compared to Managed mode
- Higher admin requirements since knowledge on an external system (LDAP) is needed
- Greater time needed to troubleshoot issues with an extra component (LDAP) in the mix
- Unexpected changes to Docker Enterprise due to changes made to upstream LDAP/AD systems, which can have unexpected effects on Docker Enterprise

A recommended best practice is to use group membership to control the access of user accounts to resources. Ideally, the management of such group membership is achieved through a centralized Identity Management or a Role Based Access Control system. This provides a standard, flexible, and scalable model to control the authentication and authorization rules within Docker Enterprise through a centralized directory server. Through the Identity Management system, this directory server is kept in sync with user on-boarding, off-boarding, and changes in roles and responsibilities.

To change the mode of authentication, use the form at [Admin Settings ➔ Authentication & Authorization](https://docs.docker.com/ee/ucp/admin/configure/external-auth/) in the UCP UI. In this form, set the **LDAP Enabled** toggle to **Yes**.

Accounts that are discovered and subsequently synced from the directory server can be automatically assigned default permissions on their own private collections. To assign additional permissions on non-private collection, those users need to be added to appropriate teams that have the required role(s) assigned.

For details about the LDAP configuration options, refer to the Integrate with an LDAP directory documentation.
The following list highlights important configuration options to consider when setting up LDAP authentication:

- In LDAP auth mode, when accounts are discovered, they may not get created within UCP until after the users log into the system. This is controlled by the **Just-in-time User Provisioning** setting in the LDAP configuration. It is recommended to turn on this setting.

- A user account on the directory server has to be configured to discover and import accounts from the directory server. This user account is recommended to have read-only to view the necessary organizationalUnits (ou) and query for group memberships. The details for this account are configured using the fields **Reader DN** and **Reader Password**. The **Reader DN** must be in the distinguishedName format.

- Use secure LDAP if at all possible.

- Use the **LDAP Test Login** section of the LDAP configuration to make sure you can connect before switching to LDAP authentication.

- Once the form is filled out and the test connection succeeds, the sync button provides an option to run a sync immediately without waiting for the next interval. Doing so initiates an LDAP connection and runs the filters to import the users.

- After the configuration is complete and saved, it should be possible to log in using a valid LDAP/AD account that matched the sync criteria. The account should be active within the LDAP/AD system for the login to be successful within Docker Enterprise.

- The progress / status of the sync and any errors that occur can be viewed and analyzed in the **LDAP Sync Jobs** section.

### Organizations & Teams

User accounts that exist within Docker Enterprise, either through a LDAP sync or manually managed, can be organized into teams. Teams need to be contained within an Organization. Each team created can be granted a role on collections that will allow the members of the team to operate within the associated collection.

Consider this example of creating an organization called **enterprise-applications** with three teams **Developers**, **Testers** and **Operations**.

To create a team, an organization needs to be created first. An organization can be created under **Access Control —> Orgs & Teams** by clicking the **Create** button.

Teams can be created in the UCP UI by clicking on an organization and then clicking on + on the upper right side of the page. Once a team is created, members can be added to the team manually or synchronized via LDAP groups. This is based on an automatic sync of discovered accounts from the directory server that was configured to enable the LDAP Auth mode. Finer filters can be applied here which determine which discovered accounts are placed into which teams. A team can have multiple users, and a user can be a member of zero to multiple teams. Below is an example of creating three teams **Developers**, **Testers** and **Operations** inside the **enterprise-applications** organization.

First, create the **enterprise-applications** organization:

Select the **enterprise-applications** organization:

Create the **Developers** team:

Create the **Testers** team:
Create the **Operations** team:

---

## Resource Sets

To control user access, cluster resources are grouped into Kubernetes namespaces or Docker Swarm collections.

**Kubernetes namespaces:** A namespace is a logical area for a Kubernetes cluster. Kubernetes comes with a default namespace for your cluster objects, plus two more namespaces for system and public resources. You can create custom namespaces, but unlike Swarm collections, namespaces cannot be nested. Resource types that users can access in a Kubernetes namespace include pods, deployments, network policies, nodes, services, secrets, and many more.

**Swarm collections:** A collection has a directory-like structure that holds Swarm resources. You can create collections in UCP by defining a directory path and moving resources into it. Also, you can create the path in UCP and use labels in your YAML file to assign application resources to the path. Resource types that users can access in a Swarm collection include containers, networks, nodes, services, secrets, and volumes.

Together, namespaces and collections are named resource sets. For more information, see the [Resource Set](https://docs.docker.com/ee/ucp/authorization/#resource-sets) documentation.

### Kubernetes Namespaces

A namespace is a scope for Kubernetes resources within a cluster. Kubernetes comes with a default namespace for your cluster objects, plus two more namespaces for system and public resources. You can create custom namespaces, but unlike Swarm collections, namespaces cannot be nested. Resource types that users can access in a Kubernetes namespace include pods, deployments, network policies, nodes, services, secrets, and many more.

Namespaces can be found in UCP under **Kubernetes** —> **Namespaces.** To create a new namespace click the **Create** button which will bring up the **Create Kubernetes Object** panel. On the **Create Kubernetes Object** panel you can either enter the YAML directly or upload an existing YAML file. For more information about Kubernetes namespace see the [Share a Cluster with Namespaces](https://kubernetes.io/docs/tasks/administer-cluster/namespaces/) documentation. Below is an example of creating a namespace called `prod-billing-application`.

---

Once a namespace has been created, a Role Binding can be created to assign a role to users, teams, or service accounts based on the functions they need to perform.

### Swarm Collections

A collection is a logical construct that can be used to group a set of resources. Collection are found under **Shared Resources** —> **Collections.** The **Swarm** collection is the root collections and all collections must be created as a child of the **Swarm** collection. To create a collections first click **View Children** next to the **Swarm** collection. Then click the **Create Collections** on the upper right of the page. Below is an example of creating a collection called `Production`. 

---
The **Production** collection in this example is used to hold other application collections. One such collection is the **Billing Application** which can be created as a child of the **Production** collection. To Create the **Billing Application** navigate to **Shared Resources** —> **Collections** —> click **View Children** next to the **Swarm** collection —> click **View Children** next to the **Production** collection and then click the **Create Collections** on the upper right of the page.

At this point, a grant can be created to assign a role to the teams based on their functions within the collection(s).

**Grants**

Docker Enterprise administrators can create grants to control how users and organizations access resource sets.

A grant defines *who* has *how much* access to *what* resources. Each grant is a 1:1:1 mapping of subject, role, and resource set. A common workflow for creating grants has four steps:

- Add and configure subjects (users, teams, and service accounts).
- Define custom roles (or use defaults) by adding permitted API operations per type of resource.
- Group cluster resources into a resource set.
- Create grants by combining subject + role + resource set.

It is easier to explain with a real world example:

Suppose you have a simple application called **www**, which is a web server based on the **nginx** official image. Also suppose that the **www** application is one of the billing applications deployed into production. There are three teams that need access to this application — Developers, Testers, and Operations. Typically, Testers need view only access and nothing more, while the **Operations** team would need full control to manage and maintain the environment. The Developers team needs access to troubleshoot, restart, and control the lifecycle of the application but should be forbidden from any other activity involving the need to access the host file systems or starting up privileged containers. This follows a typical use case that uses the principles of "least privilege / permission" as well as "separation of duties."

The following sections will address the access requirements of the example application **www** described above for each orchestrator.

**Kubernetes Role Bindings**

Kubernetes Role Bindings can be created in UCP using the wizard by navigating to **Access Control** —> **Grants** —> **Swarm** tab —> **Create Role Binding**.

First create a role binding to provide the **enterprise-applications Operations** team with the **admin** role on the **prod-billing-application** namespace.

Select the organization **enterprise-applications** and team **Operations** for the Subject:

Click **Next**.

Select the **prod-billing-application** namespace for the Resource Set:
Select the **Cluster Role** for the Role Type and **admin** for the Cluster Role:

Click **Create**.

This will create a role binding **enterprise-applications-Operations:admin** on the **prod-billing-application** namespace.

Create two more role binding, using the same steps above, for the remaining teams:

1. Select the **enterprise-applications Developers** team for the subject, select the **prod-billing-application** namespace for the resource set, select **Cluster Role** for the Role Type and **edit** for the Cluster Role.

2. Select the **enterprise-applications Testers** team for the subject, select the **prod-billing-application** namespace for the resource set, select **Cluster Role** for the Role Type and **view** for the Cluster Role.

With these role bindings, the teams would have appropriate levels of access based on their functions to any resources within the **prod-billing-application** namespace.

To associate the **www** resources with the **prod-billing-application** namespace, the resources are created in the usual manner, except the namespace is selected before creating the resources as show below:

**Swarm Grants**

Swarm grants can be created in UCP using the wizard by navigating to **Access Control** —> **Grants** —> **Swarm** tab —> **Create Grant**.

First create a grant to provide the **enterprise-applications Operations** team with **Full Control** of the **Production** collection.

Select the organization **enterprise-applications** and team **Operations** for the Subject:

Click **Next**.

Select the **Production** collection for the Resource Set:

Select the **Full Control** role type for the Role:

Click **Create**.

This will create a grant for **Team - Operations, Full Control, /Production**

Create two more grant, using the same steps above, for the remaining teams:

1. Select the **enterprise-applications Developers** team for the subject, select the **Billing Application** collection for the resource set, and select the **Restricted Control** role type for the role.

2. Select the **enterprise-applications Testers** team for the subject, select the **Billing Application** collection.
for the resource set, and select the View Only role type for the role.

With these grants, the teams would have appropriate levels of access based on their functions to any application within the Billing Application collection.

To associate the www application with the Billing Application collection, the service is created in the usual manner, except the collection is selected before creating the service as show below:

Select the collection:

Strategy for Using LDAP Filters

The users needing access to UCP are all sourced from the corporate Directory Server system. These users are the admin users needed to manage the Docker Enterprise infrastructure as well as all members of each of the teams configured within UCP. Also assume that the total universe of users needing access to Docker Enterprise (includes admins, developers, testers, and operations) is a subset of the gamut of users within the Directory Server.

A recommended strategy to use when organizing users is to create an overarching membership group that identifies all users of Docker Enterprise, irrespective of which team they are a part of. Let us call this group Docker_Users. No user should be made a member of this group directly. Instead, the Docker_Users group should contain other groups and only those other groups as its members. Per our example, let us call these groups dev, test, and ops. In our example, these groups are part of what is known as a nested group structure within the directory server. Nested groups allow the inheritance of permissions from one group to each of its sub-groups.

NOTE: Some directory servers do not support the feature of nested groups or even the memberOf attribute by default. If so, then they would need to be enabled. If the choice of directory server does not support these features at all, then alternate means of organizing users and querying them should be used. Microsoft Active Directory supports both these features out of the box.

User accounts should be added as members of these sub-groups in the directory server. This should not impact any existing layout in the organization units or pre-existing group membership for these users. The sub-group should be used as the value of the Group DN in the defining of the teams.

Finally, if and when it becomes necessary to terminate all access for any user account, removing the group membership of the account from just the one group Docker_Users would remove all access for the user. Due to the nature of how nested groups work, all additional access within Docker is automatically cleaned — the user account is removed from any and all team memberships at the time of the next sync without need for manual intervention or additional steps. This step can be integrated into a standard on-boarding / off-boarding automated provisioning step within a corporate Identity Management system.

Authentication API (eNZi)

The AuthN API or eNZi (as it is known internally and pronounced N-Z) is a centralized authentication and authorization service and framework for Docker Enterprise. This API is completely integrated and configured into Docker Enterprise and works seamlessly with UCP as well as DTR. This is the component and service under the hood that manages accounts, teams and organizations, user sessions, permissions and access control through labels, Single-Sign-On (Web SSO) through OpenID Connect, and synchronization of account details from an external LDAP-based system into Docker Enterprise.
For regular day-to-day activities, users and operators need not be concerned with the AuthN API and how it works. However, its features can be leveraged to automate many common functions and/or bypass the UCP UI altogether to manage and manipulate the data directly.

Interaction with AuthN can be accomplished in two ways: via the exposed RESTful AuthN API over HTTP or via the `enzi` command.

For example, the command below uses `curl` and `jq` to fetch all user accounts in Docker Enterprise via the AuthN API over HTTP:

```
$ curl --silent --insecure --header "Authorization: Bearer $(curl --silent --insecure \
   --data '{"username":"<admin-username>","password":"<admin-password>"}' \n   https://<UCP-domain-name>/auth/login | jq --raw-output .auth_token)" \n   https://<UCP-domain-name>/enzi/v0/accounts | jq .
```

The AuthN service can also be invoked on the CLI on a UCP controller. To connect into it, run the following on a UCP controller:

```
$ docker exec -it ucp-auth-api sh
```

At the resulting prompt (#), type the `enzi` command with a sub-command such as the one below to list the database table status:

```
# enzi db-status
```

**Note:** Refer to Recovering the Admin Password for Docker Enterprise (https://success.docker.com/article/recovering-the-admin-password-for-docker-ee-standard-and-advanced) for a detailed example.