

# VMware Tanzu Build Service 1.1 Documentation

Tanzu Build Service 1.1

You can find the most up-to-date technical documentation on the VMware website at:  
<https://docs.vmware.com/>

**VMware, Inc.**  
3401 Hillview Ave.  
Palo Alto, CA 94304  
[www.vmware.com](http://www.vmware.com)

Copyright © 2022 VMware, Inc. All rights reserved. [Copyright and trademark information.](#)

# Contents

Tanzu Build Service 1.1 Documentation	9
Overview	9
Build Service Concepts	9
Image	9
Builder	9
ClusterStore	9
ClusterStack	9
Build Service Components	10
Build Service Dependencies	10
Buildpacks	10
Stacks	10
Updating Build Service Dependencies	10
Troubleshooting	10
Installing Tanzu Build Service	11
Prerequisites	11
Installing	12
Relocate Images to a Registry	12
Install Tanzu Build Service	13
Install Tanzu Build Service Public Registry	13
Installing with a CA certificate for internal registry	14
Import Tanzu Build Service Dependencies	14
Additional Configuration	15
Configuring TKGI as an OIDC Provider	15
Installation to Air-Gapped Environment	16
Relocate Images to a Registry (Air-Gapped)	16
Installing (Air-Gapped)	17
Additional Configuration	18
Import Tanzu Build Service Dependencies (Air-Gapped)	18
Relocate Tanzu Build Service Dependency Images (Air-Gapped)	18
Import Tanzu Build Service Dependency Resources (Air-Gapped)	19
Verify Installation	19
Upgrading Tanzu Build Service	19
Updating Build Service Dependencies	20

Ensuring Access to Cluster Builders	20
Ensuring the Run Image is Readable	20
Next Steps	20
Kubernetes Permissions for Installation	21
<b>Installing Windows Components (Beta)</b>	<b>23</b>
<b>Getting Started with Tanzu Build Service</b>	<b>24</b>
Prerequisites	24
Assumptions	25
Installation	25
Relocate Images to a Registry	25
Install Tanzu Build Service	25
Import Tanzu Build Service Dependencies	26
Verify Installation	26
Create an Image	26
<b>Accessing Tanzu Build Service</b>	<b>28</b>
<b>Updating Build Service Dependencies</b>	<b>29</b>
Updating Dependencies	29
Bulk Update	29
Cluster Stacks Update	29
Cluster Store Update	30
Updating Buildpacks From Tanzu Network	31
Offline Update of Dependencies	32
<b>Managing Secrets</b>	<b>33</b>
Overview	33
Create Secrets	33
Create a Docker Hub Registry Secret	33
Create a GCR Registry Secret	34
Create an Artifactory, Harbor, or ACR Registry Secret	34
Create a Git SSH Secret	35
Create a Git Basic Auth Secret	35
List Secrets	36
Delete Secrets	36
Encrypting Secrets at Rest	37
<b>Synced-Secrets in Tanzu Build Service</b>	<b>37</b>
When to use Synchronized Secrets	37

Managing Secret Synchronization	37
Create a Synchronized Secret	37
Update a Synchronized Secret	38
Stop Synchronizing a Secret	38
<b>Managing Images and Builds</b>	<b>39</b>
<b>Images</b>	<b>40</b>
Creating Images	40
Source Code	40
Builders	40
Creating an Image With Source Code in a Git Repository	40
Create an Image With Source Code In A Blob Store	41
Creating an Image With Local Source Code	42
Buildpack Configuration	43
Buildpack Configuration Use Cases	43
Buildpack Configuration Documentation	43
Buildpack Configuration in Images	43
Patching Images	44
Saving Images	45
Listing Images	45
Filter Images	46
Image Rebuilds	46
Trigger an Image Rebuild	47
Viewing the Status of an Image	47
Deleting an Image	47
Managing Images with YAML	48
Using Secrets	48
Debugging with Image Status	48
Image Service Bindings	49
Creating an Image with Service Bindings	49
<b>Builds</b>	<b>50</b>
Listing Builds	50
Viewing Build Details for an Image	51
Image Status shows ImagePullBackOff	52
Getting Build Logs	53
Viewing Bill of Materials	55
Offline Builds	56
Image Signing with Notary	56

Generate Notary Signing Keys	56
Create a Secret to be used for TBS Image Signing	57
Create an Image that will be Signed by Notary	57
<b>Managing ClusterStacks</b>	<b>58</b>
Create a ClusterStack	59
Update a ClusterStack	59
Save a ClusterStack	60
Get ClusterStack Status	60
Delete a ClusterStack	60
List all ClusterStacks	60
How to update an Image for Stack updates only?	61
<b>Managing Stores</b>	<b>62</b>
Creating Buildpacks and Buildpackages	62
Listing ClusterStores	62
Creating a ClusterStore	63
Saving a ClusterStore	63
Adding Buildpackages to a ClusterStore	63
Adding Buildpackages to a ClusterStore from Tanzu Network	64
Offline Adding Buildpackages to a ClusterStore from Tanzu Network	64
Removing Buildpackages from a ClusterStore	65
Get ClusterStore Status	65
Migrating Buildpacks	66
Corresponding kpack Resource	66
<b>Managing Builders</b>	<b>67</b>
Creating a Builder	68
Patching a Builder	69
Saving Builders	70
Deleting Builders	71
Retrieving Builder Details	71
Listing Builders	72
Corresponding kpack Resources	72
Pinning Buildpack versions	72
Update Lifecycle	73
<b>Managing Custom Stacks</b>	<b>74</b>
Creating a CustomStack	74
Source Configuration	74

Destination Configuration	75
Example CustomStack from Registry Images	75
Example CustomStack from ClusterStack	76
Debugging CustomStacks	78
<b>RBAC in Tanzu Build Service</b>	<b>79</b>
RBAC using Projects Operator	79
RBAC Support in Tanzu Build Service	79
Build Service User Role	79
Build Service Admin Role	80
<b>Frequently Asked Questions</b>	<b>81</b>
How do Cloud Native Buildpacks (CNBs), kpack, and Tanzu Build Service overlap and differ?	81
Why do I see two images in the image registry after a successful build?	81
How does TBS work in air gapped environments?	81
Is there documentation on supported Tanzu Buildpacks?	81
Why do I get an X509 error from Build Service when trying to create an image in my registry?	82
How do I configure a secret to publish images to Dockerhub?	82
How can I configure an image to pull from a private GitHub repository?	82
Why do some builds fail with "Error: could not read run image: *"?	82
Why don't my image builds appear in my Harbor v1.X.X registry?	83
How do I fix "unsupported status code 500" when creating a builder on my Harbor v2.X.X registry?	83
How do I configure credentials for using gcr as my installation registry?	83
Can I configure a proxy for my Tanzu Build Service?	83
How do I build my app locally using kpack builders?	84
What can I do with the kp --dry-run and --output flags?	84
Does TBS support Azure Devops for git repositories	85
Why do I get a "repository does not exist" error when I use ECR Registry?	85
How do I troubleshoot a failed build?	85
How do I troubleshoot an UNAUTHENTICATED error?	86
<b>Additional resources for Tanzu Build Service</b>	<b>88</b>
Concourse Kpack resource	88
Helpful Articles	88
Helpful Videos	88
Helpful Repositories	88
<b>Release Notes</b>	<b>89</b>
v1.1.4	89

Product Snapshot	89
v1.1.3	89
Product Snapshot	89
v1.1.2	89
Product Snapshot	89
v1.1.1	90
v1.1.0	90
Product Snapshot	90
Product Dependencies	91
Upgrade Path	91

# Tanzu Build Service 1.1 Documentation

This topic provides an overview of Tanzu Build Service.

## Overview

Tanzu Build Service uses the open-source [Cloud Native Buildpacks](#) project to turn application source code into [container images](#). Build Service executes reproducible builds that align with modern container standards, and additionally keeps images up-to-date. It does so by leveraging Kubernetes infrastructure with [kpack](#), a Cloud Native Buildpacks Platform, to orchestrate the image lifecycle. The kpack CLI tool, `kp` can aid in managing kpack resources.

Build Service helps you develop and automate containerized software workflows securely and at scale.

## Build Service Concepts

Build Service reduces operational overhead and improves security by automating the building of application images. It relies on [Image](#), [Builder](#), [ClusterStore](#) and [ClusterStack](#) to achieve these results.

### Image

An [Image](#) defines the source of the application, build time environment and registry destination. This source code could reside in git, a blobstore, or as code on a workstation.

For more information see the [Managing Images and Builds](#) page.

### Builder

A [Builder](#) references the Stack and Buildpacks that are used in the process of building source code. They "provide" the Buildpacks that run against the application and the OS images upon which the application is built and run.

For more information see the [Managing Builders](#) page.

### ClusterStore

A [ClusterStore](#) serves as a repository for Cloud Native Buildpacks available for use in Builders. One can populate a store with Buildpacks they [create](#) and [package](#).

For more information see the [Managing ClusterStores](#) page.

### ClusterStack

A **ClusterStack** defines a pair of build and run OS images. Critical security vulnerabilities are addressed by building apps on the most up-to-date stack. The stacks used by Build Service to build applications are referenced in the Builders.

For more information see the [Managing ClusterStacks](#) page.

## Build Service Components

Tanzu Build Service 1.0.2 ships with the following components:

- [kpack v0.2.0](#)
- [kpack CLI \(kp\) v0.2.0](#)
- [CNB lifecycle v0.10.2](#)

## Build Service Dependencies

### Buildpacks

Tanzu Build Service utilize [Tanzu Buildpacks](#).

### Stacks

Stack Documentation is available on the [Tanzu Buildpacks documentation](#).

The following Stacks and their updates can be found on the [Tanzu Build Service Dependencies](#) page.

Name	ID
tiny	io.paketo.stacks.tiny
base	io.buildpacks.stacks.bionic
full	io.buildpacks.stacks.bionic

## Updating Build Service Dependencies

Build Service allows the user to update Buildpacks and Stacks via the [kp](#) CLI. You can learn more about updating Build Service dependencies [here](#).

## Troubleshooting

For troubleshooting failed builds, check the [FAQ](#) section of our docs.

If you are unable to resolve your problem, please contact [Tanzu VMware Support](#).

# Installing Tanzu Build Service

This topic describes how to install and configure Tanzu Build Service.

## Prerequisites

Before you install Build Service, you must:

- Be on Kubernetes cluster v1.16 or later
- Have access to the Kubernetes cluster satisfying the [minimum required permissions](#).
- Ensure that all worker nodes have at least 50 GB of ephemeral storage allocated to them.
  - ◊ To do this on TKGs, mount a 50GB volume at `/var/lib` to the worker nodes in the `TanzuKubernetesCluster` resource that corresponds to your TKGs cluster. [These instructions](#) show how to configure storage on worker nodes.
- Have access to a container registry to install Tanzu Build Service and store the application images that will be created.
  - ◊ Although the documentation references specific registries for the purpose of providing examples, any registry that adheres to the Docker Registry HTTP API V2 is supported
  - ◊ TBS uses ~5GB of registry space for installation, this does not include the space that will be used for application images.
- Ensure your Kubernetes cluster is configured with default `StorageClass`. Tanzu Build Service will default to using 2G of cache if a default `StorageClass` is defined. Build Service utilizes `PersistentVolumeClaims` to cache build artifacts, which reduces the time of subsequent builds.
 

For more information, see [Persistent Volumes](#) in the Kubernetes documentation. And for information on defining a default StorageClass, see [Changing the default StorageClass](#)
- Download three [Carvel](#) CLIs for your operating system. These tools will facilitate the installation of Tanzu Build Service on your cluster. They can be found on their respective Tanzu Network pages:
  - ◊ [kapp](#) is a deployment tool that allows users to manage Kubernetes resources in bulk.
  - ◊ [ytt](#) is a templating tool that understands YAML structure.
  - ◊ [kblid](#) is tool that builds, pushes, and relocates container images.
- Navigate to the following pages in Tanzu Network and accept all EULAs highlighted in yellow.
  - ◊ [Tanzu Build Service](#)
  - ◊ [Tanzu Build Service Dependencies](#)

- ◊ [Buildpacks for VMware Tanzu](#)
  - ◊ [Stacks for VMware Tanzu](#)
- Download the Build Service Bundle from the [Tanzu Build Service](#) page on Tanzu Network.
  - ◊ Unarchive the Build Service Bundle file:

```
tar xvf build-service-<version>.tar -C /tmp
```

- Download the `kp` CLI for your operating system from the [Tanzu Build Service](#) page on Tanzu Network. The `kp` CLI help text is published [here](#).
  - ◊ These docs assume `kp cli v0.2.*` from TBS release `v1.1.*`. If a feature is not working, you may need to upgrade your cli.
- Download the `docker` CLI to authenticate with registries.
- Download the Dependency Descriptor file (`descriptor-<version>.yaml`) from the latest release on the [Tanzu Build Service Dependencies](#) page on Tanzu Network. This file contains paths to images that contain dependency resources Tanzu Build Service needs to execute image builds.

**Note:** Clusters running with Containerd 1.4.1 are not compatible with TBS. Notably, TKG 1.2.1 uses this version of Containerd, a different TKG version must be used.

**Note:** TKGs clusters running Kubernetes 1.20 are not compatible with TBS. A TKGs patch is expected to fix this. In the meantime, we recommend running k8s 1.19 when using TKGs.

## Installing

Create a kubernetes cluster where you would like to install build service and target the cluster as follows:

```
kubectl config use-context <CONTEXT-NAME>
```

## Relocate Images to a Registry

This procedure relocates images from the Tanzu Network registry to an internal image registry.

1. Log in to the image registry where you want to store the images by running:

```
docker login <IMAGE-REGISTRY>
```

Where `IMAGE-REGISTRY` is the name of the image registry where you want to store the images.

2. Log in to the Tanzu Network registry with your Tanzu Network credentials:

```
docker login registry.pivotal.io
```

3. Relocate the images with the [Carvel](#) tool `kbld` by running:

```
kbld relocate -f /tmp/images.lock --lock-output /tmp/images-relocated.lock --repository <IMAGE-REPOSITORY>
```

Where `IMAGE-REPOSITORY` is the repository in your registry that you want to relocate images to.

**Note:** The flag argument `--lock-output /tmp/images-relocated.lock` creates a file that will be used for installation.

**Note:** When relocating, the `IMAGE-REPOSITORY` must be the `IMAGE-REGISTRY` appended with the destination repository for the images. For example, `IMAGE-REGISTRY/build-service`.

**Exception:** When relocating to Dockerhub, you must provide the Dockerhub repository and an image name that `kbld` will use for relocation. For example, `my-dockerhub-account/build-service`.

For example:

- Dockerhub `kbld relocate -f /tmp/images.lock --lock-output /tmp/images-relocated.lock --repository my-dockerhub-account/build-service`
- GCR `kbld relocate -f /tmp/images.lock --lock-output /tmp/images-relocated.lock --repository gcr.io/my-project/build-service`
- Artifactory `kbld relocate -f /tmp/images.lock --lock-output /tmp/images-relocated.lock --repository artifactory.com/my-project/build-service`
- Harbor `kbld relocate -f /tmp/images.lock --lock-output /tmp/images-relocated.lock --repository harbor.io/my-project/build-service`

## Install Tanzu Build Service

There are two ways to install Tanzu Build Service:

1. Using a public registry (eg. GCR, Dockerhub) or an internal registry that uses a trusted certificate (eg. Let's Encrypt)
2. Using an internal registry that uses a self-signed CA certificate (eg. Harbor, Artifactory)

### Install Tanzu Build Service Public Registry

Use the [Carvel](#) tools `kapp`, `ytt`, and `kbld` to install Build Service and define the required Build Service parameters by running:

```
ytt -f /tmp/values.yaml \
  -f /tmp/manifests/ \
  -v docker_repository="<IMAGE-REPOSITORY>" \
  -v docker_username="<REGISTRY-USERNAME>" \
  -v docker_password="<REGISTRY-PASSWORD>" \
  | kbld -f /tmp/images-relocated.lock -f- \
  | kapp deploy -a tanzu-build-service -f- -y
```

Where:

- `IMAGE-REPOSITORY` is the image repository where Tanzu Build Service images exist.

**Note:** This is identical to the `IMAGE-REPOSITORY` argument provided during `kbld` relocation command.

**Exception:** When using Dockerhub as your registry target, only use your DockerHub account for this value. For example, `my-dockerhub-account` (without `/build-service`).

Otherwise, you will encounter an error similar to: `Error: invalid credentials, ensure registry credentials for 'index.docker.io/my-dockerhub-account/build-service/tanzu-buildpacks_go' are available locally`

- `REGISTRY-USERNAME` is the username you use to access the registry. `gcr.io` expects `_json_key` as the username when using JSON key file authentication.
- `REGISTRY-PASSWORD` is the password you use to access the registry.

**Note:** [Managing Secrets](managing-secrets.html) for more information about how the registry username and password are used in Tanzu Build Service.

## Installing with a CA certificate for internal registry

To install Tanzu Build Service with an internal registry that requires providing a CA certificate such as Harbor, use the normal installation command with the CA certificate file passed in with a `-f` flag:

```
ytt -f /tmp/values.yaml \
  -f /tmp/manifests/ \
  -f <PATH-TO-CA> \
  -v docker_repository="<IMAGE-REPOSITORY>" \
  -v docker_username="<REGISTRY-USERNAME>" \
  -v docker_password="<REGISTRY-PASSWORD>" \
  | kblld -f /tmp/images-relocated.lock -f- \
  | kapp deploy -a tanzu-build-service -f- -y
```

Where:

- `PATH-TO-CA` is the path to the registry root CA. This CA is required to enable Build Service to interact with internally deployed registries. This is the CA that was used while deploying the registry.
- `IMAGE-REPOSITORY` is the image repository where Tanzu Build Service images exist.

**Note:** This is identical to the `IMAGE-REPOSITORY` argument provided during `kblld` relocation command.

**Exception:** When using Dockerhub as your registry target, only use your DockerHub account for this value. For example, `my-dockerhub-account` (without `/build-service`). Otherwise, you will encounter an error similar to: `Error: invalid credentials, ensure registry credentials for 'index.docker.io/my-dockerhub-account/build-service/tanzu-buildpacks_go' are available locally`

- `REGISTRY-USERNAME` is the username you use to access the registry. `gcr.io` expects `_json_key` as the username when using JSON key file authentication.
- `REGISTRY-PASSWORD` is the password you use to access the registry.

**Note:** [Managing Secrets](managing-secrets.html) for more information about how the registry username and password are used in Tanzu Build Service.

## Import Tanzu Build Service Dependencies

The Tanzu Build Service Dependencies (Stacks, Buildpacks, Builders, etc.) are used to build applications and keep them patched.

These must be imported with the `kp` cli and the Dependency Descriptor (`descriptor-  
<version>.yaml`) file from the [Tanzu Build Service Dependencies](#) page:

```
kp import -f /tmp/descriptor-<version>.yaml
```

When importing to a registry that uses a self-signed CA certificate:

```
kp import -f /tmp/descriptor-<version>.yaml --registry-ca-cert-path <path-to-ca-cert>
```

Using the `--show-changes` flag will give a summary of the resource changes for the import. You will also be asked to confirm the import. Confirmation can be skipped with `--force`.

Successfully performing a `kp import` command requires that your Tanzu Network account has access to the images specified in the Dependency Descriptor file. Users can only access these images if they agree to the dependency EULAs.

Users must navigate to the following dependencies pages in Tanzu Network and accept all EULAs highlighted in yellow.

1. [Tanzu Build Service Dependencies](#)
2. [Buildpacks for VMware Tanzu](#)
3. [Stacks for VMware Tanzu](#)

**Note:** `kp import` will fail if it cannot access the images in all of the above Tanzu Network pages.

**Note:** You must be logged in locally to the registry used for `IMAGE-REGISTRY` during relocation and the Tanzu Network registry `registry.pivotal.io`.

## Additional Configuration

Other optional parameters can be added using the `-v` flag:

- `admin_users` is a comma separated list of users who will be granted admin privileges on Build Service.
- `admin_groups`: a comma separated list of groups that will be granted admin privileges on Build Service.
- `http_proxy`: The HTTP proxy to use for network traffic.
- `https_proxy`: The HTTPS proxy to use for network traffic.
- `no_proxy`: A comma-separated list of hostnames, IP addresses, or IP ranges in CIDR format that should not use a proxy.

**Note:** When proxy server is enabled using `http_proxy` and/or `https_proxy`, traffic to the kubernetes API server will also flow through the proxy server. This is a known limitation and can be circumvented by using `no_proxy` to specify the kubernetes API server.

## Configuring TKGI as an OIDC Provider

The authentication and authorization processes for Build Service use a combination of RBAC rules and third-party authentication, including OpenID Connect (OIDC). You may configure UAA as an OIDC provider for your TKGI deployment to provide authentication for Build Service.

To configure UAA as an OIDC provider for your TKGI deployment:

1. Navigate to the OpsManager Installation Dashboard.
2. Click the TKGI tile.
3. Select **UAA**.
4. Under **Configure created clusters to use UAA as the OIDC provider**, select **Enable**.
5. Ensure the values in the **UAA OIDC Groups Prefix** and **UAA OIDC Username Prefix** fields are the same and record them. For example, "`oidc:`". You will need these values during the installation of Build Service.

**Note:** Ensure you add a `:` at the end of the desired prefix.

6. Click **Save**.
7. In the OpsManager Installation Dashboard, click **Review Pending Changes**, then **Apply Changes**.

## Installation to Air-Gapped Environment

Tanzu Build Service can be installed to a Kubernetes Cluster and registry that are air-gapped from external traffic.

An air-gapped environment will often use an internal registry with a self-signed CA certificate and you will need access to this CA certificate file to install TBS.

**Note:** If you are using a CA certificate that is trusted (eg. Let's Encrypt) you will not need the CA certificate file.

## Relocate Images to a Registry (Air-Gapped)

This procedure relocates images from the Tanzu Network registry to an internal image registry via a local machine.

The local machine must have write access to the internal registry.

1. Log in to the image registry where you want to store the images by running:

```
docker login <IMAGE-REGISTRY>
```

Where `IMAGE-REGISTRY` is the name of the image registry where you want to store the images.

2. Log in to the Tanzu Network registry with your Tanzu Network credentials:

```
docker login registry.pivotal.io
```

3. Package the images in a file on your local machine with the [Carvel](#) tool `kbld` by running:

```
kbld package -f /tmp/images.lock --output /tmp/packaged-images.tar
```

4. Move the output file `packaged-images.tar` to a machine that has access to the air-gapped environment.
5. Unpackage the images from your local machine to the internal registry:

```

kblld unpackage -f /tmp/images.lock \
  --input /tmp/packaged-images.tar \
  --repository <IMAGE-REPOSITORY> \
  --lock-output /tmp/images-relocated.lock \
  --registry-ca-cert-path <PATH-TO-CA>

```

Where:

- `IMAGE-REPOSITORY` is the repository in your registry that you want to relocate images to.
- `PATH-TO-CA` is the path to the registry CA certificate file.

**Note:** The flag argument `--lock-output /tmp/images-relocated.lock` creates a file that will be used for installation.

**Note:** When relocating to a registry that is not Dockerhub, the `IMAGE-REPOSITORY` must be the `IMAGE-REGISTRY` appended with the destination repository for the images. For example, `IMAGE-REGISTRY/build-service`.

**Exception:** When relocating to Dockerhub, you must provide the Dockerhub repository and an image name that `kblld` will use for relocation. For example, `my-dockerhub-account/build-service`.

For example:

- Dockerhub `kblld unpackage -f /tmp/images.lock --input /tmp/packaged-images.tar --lock-output /tmp/images-relocated.lock --repository my-dockerhub-account/build-service --registry-ca-cert-path ca.crt`
- GCR `kblld unpackage -f /tmp/images.lock --input /tmp/packaged-images.tar --lock-output /tmp/images-relocated.lock --repository gcr.io/my-project/build-service --registry-ca-cert-path ca.crt`
- Artifactory `kblld unpackage -f /tmp/images.lock --input /tmp/packaged-images.tar --lock-output /tmp/images-relocated.lock --repository artifactory.com/my-project/build-service --registry-ca-cert-path ca.crt`
- Harbor `kblld unpackage -f /tmp/images.lock --input /tmp/packaged-images.tar --lock-output /tmp/images-relocated.lock --repository harbor.io/my-project/build-service --registry-ca-cert-path ca.crt`

## Installing (Air-Gapped)

Use the `Carvel` tools `kapp`, `ytt`, and `kblld` to install Build Service and define the required Build Service parameters by running:

```

ytt -f /tmp/values.yaml \
  -f /tmp/manifests/ \
  -f <PATH-TO-CA> \
  -v docker_repository="<IMAGE-REPOSITORY>" \
  -v docker_username="<REGISTRY-USERNAME>" \
  -v docker_password="<REGISTRY-PASSWORD>" \
  | kblld -f /tmp/images-relocated.lock -f- \
  | kapp deploy -a tanzu-build-service -f- -y

```

Where:

- `PATH-TO-CA` is the path to the registry root CA. This CA is required to enable Build Service to interact with internally deployed registries. This is the CA that was used while deploying the registry.
- `IMAGE-REPOSITORY` is the image repository where Tanzu Build Service images exist.

**Note:** This is identical to the `IMAGE-REPOSITORY` argument provided during `kbld` relocation command.

**Exception:** When using Dockerhub as your registry target, only use your DockerHub account for this value. For example, `my-dockerhub-account` (without `/build-service`). Otherwise, you will encounter an error similar to: `Error: invalid credentials, ensure registry credentials for 'index.docker.io/my-dockerhub-account/build-service/tanzu-buildpacks_go' are available locally`

- `REGISTRY-USERNAME` is the username you use to access the registry. `gcr.io` expects `_json_key` as the username when using JSON key file authentication.
- `REGISTRY-PASSWORD` is the password you use to access the registry.

**Note:** [Managing Secrets](managing-secrets.html) for more information about how the registry username and password are used in Tanzu Build Service.

## Additional Configuration

Other optional parameters can be added using the `-v` flag:

- `admin_users` is a comma separated list of users who will be granted admin privileges on Build Service.
- `admin_groups`: a comma separated list of groups that will be granted admin privileges on Build Service.

## Import Tanzu Build Service Dependencies (Air-Gapped)

The Tanzu Build Service Dependencies (Stacks, Buildpacks, Builders, etc.) are used to build applications and keep them patched.

These must be imported with the `kp` cli and the Dependency Descriptor (`descriptor-<version>.yaml`) file from the [Tanzu Build Service Dependencies](#) page.

### Relocate Tanzu Build Service Dependency Images (Air-Gapped)

To import these dependencies into an air-gapped environment, they must first be relocated to the internal registry. Use `kbld` to perform this relocation similarly to installation:

1. Download the dependency images locally:

```
kbld package -f descriptor-<version>.yaml \
  --output /tmp/packaged-dependencies.tar
```

**Note:** You must be logged in locally to the Tanzu Network registry.

1. Move the output file `packaged-dependencies.tar` to a machine that has access to the air-gapped environment.
2. Upload the dependency images to the Tanzu Build Service registry:

```
kbld unpackage -f descriptor-<version>.yaml \
  --input /tmp/packaged-dependencies.tar \
  --repository <IMAGE-REPOSITORY> \
  --lock-output /tmp/dependencies-relocated.lock \
  --registry-ca-cert-path <PATH-TO-CA>
```

Where:

- `IMAGE-REPOSITORY` is the internal image repository where dependency images will be relocated.
- `PATH-TO-CA` is the path to the registry CA certificate file.

**Note:** You must be logged in locally to the registry used for `IMAGE-REGISTRY`.

### Import Tanzu Build Service Dependency Resources (Air-Gapped)

After the dependency images are uploaded to the internal registry, you can successfully import these images and create the corresponding Tanzu Build Service resources.

Use the following command with `kbld` and the `kp` CLI:

```
kbld -f descriptor-<version>.yaml -f /tmp/dependencies-relocated.lock | kp import -f --registry-ca-cert-path <path-to-ca-cert>
```

## Verify Installation

Verify your Build Service installation by first targeting the cluster Build Service has been installed on.

To verify your Build Service installation:

1. Download the `kp` binary from the [Tanzu Build Service](#) page on Tanzu Network.
2. List the cluster builders available in your installation:

```
kp clusterbuilder list
```

You should see an output that looks as follows:

NAME	READY	STACK	IMAGE
base	true	io.buildpacks.stacks.bionic	<image@sha256:digest>
default	true	io.buildpacks.stacks.bionic	<image@sha256:digest>
full	true	io.buildpacks.stacks.bionic	<image@sha256:digest>
tiny	true	io.paketo.stacks.tiny	<image@sha256:digest>

## Upgrading Tanzu Build Service

To upgrade Tanzu Build Service to a newer version, run the same commands as installation. Re-importing dependencies is not required for upgrading TBS.

Relocate the images with `kbld`:

```
kbld relocate -f /tmp/images.lock --lock-output /tmp/images-relocated.lock --repository <IMAGE-REPOSITORY>
```

Run the `ytt/kapp` command to install:

```
ytt -f /tmp/values.yaml \
  -f /tmp/manifests/ \
  -v docker_repository="<IMAGE-REPOSITORY>" \
  -v docker_username="<REGISTRY-USERNAME>" \
  -v docker_password="<REGISTRY-PASSWORD>" \
  | kbld -f /tmp/images-relocated.lock -f- \
  | kapp deploy -a tanzu-build-service -f- -y
```

## Updating Build Service Dependencies

Use the following documentation to keep applications patched and up-to-date with Tanzu Build Service:

To keep dependencies up-to-date, see [Updating Build Service Dependencies](#)

To manage Stacks, see [Managing Stacks](#)

To manage Buildpack Stores, see [Managing Stores](#)

## Ensuring Access to Cluster Builders

In order to use Cluster Builders, such as the ones installed with Tanzu Build Service, we suggest to install Tanzu Build Service to a repository that is accessible by the nodes in the kubernetes cluster without credentials.

If this is not desired, see [When to use Synchronized Secrets](#).

## Ensuring the Run Image is Readable

Build Service relies on the run-image being publicly readable or readable with the registry credentials configured in a project/namespace for the builds to be executed successfully.

The location of the run image can be identified by running the following command:

```
kp clusterstack status <stack-name>
```

If the cluster stack run image is not public, you may need to create a registry secret in any namespace where Images or Builds will be used. For more details on secrets in Tanzu Build Service, see [Managing Secrets](#)

This can be done with the `kp` CLI:

```
kp secret create my-registry-creds --registry example-registry.io --registry-user my-registry-user --namespace build-namespace
```

## Next Steps

Visit the [Managing Images and Builds](#) page to learn how to create and manage a new image.

## Kubernetes Permissions for Installation

The minimum Kubernetes RBAC permissions required to install Tanzu Build Service are as follows. This includes the namespaces required for the Kubernetes Roles:

```

---
apiVersion: v1
kind: Namespace
metadata:
  name: build-service
---
apiVersion: v1
kind: Namespace
metadata:
  name: kpack
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: build-service-install-cluster-role
rules:
- apiGroups:
  - "admissionregistration.k8s.io"
  resources:
  - mutatingwebhookconfigurations
  - validatingwebhookconfigurations
  verbs:
  - '*'
- apiGroups:
  - "rbac.authorization.k8s.io"
  resources:
  - clusterroles
  - clusterrolebindings
  verbs:
  - '*'
- apiGroups:
  - "apiextensions.k8s.io"
  resources:
  - customresourcedefinitions
  verbs:
  - '*'
- apiGroups:
  - "storage.k8s.io"
  resources:
  - storageclasses
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - kpack.io
  resources:
  - builds
  - builds/status
  - builds/finalizers

```

```

- images
- images/status
- images/finalizers
- builders
- builders/status
- clusterbuilders
- clusterbuilders/status
- clusterstores
- clusterstores/status
- clusterstacks
- clusterstacks/status
- sourceresolvers
- sourceresolvers/status
verbs:
- '*'
- apiGroups:
- "projects.vmware.com"
resources:
- projects
verbs:
- '*'
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: build-service-install-role
  namespace: build-service
rules:
- apiGroups:
- ""
resources:
- configmaps
- secrets
- serviceaccounts
- services
- namespaces
verbs:
- '*'
- apiGroups:
- "rbac.authorization.k8s.io"
resources:
- roles
- rolebindings
verbs:
- '*'
- apiGroups:
- apps
resources:
- deployments
- daemonsets
verbs:
- '*'
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: kpack-install-role
  namespace: kpack
rules:

```

```

- apiGroups:
  - ""
  resources:
    - services
    - serviceaccounts
    - namespaces
    - secrets
    - configmaps
  verbs:
    - '*'
- apiGroups:
  - "rbac.authorization.k8s.io"
  resources:
    - roles
    - rolebindings
  verbs:
    - '*'
- apiGroups:
  - apps
  resources:
    - deployments
    - daemonsets
  verbs:
    - '*'

```

The `kapp` command used to install Tanzu Build Service requires access to ConfigMaps in the namespace that will be used to run `kapp`:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: kapp-role
  namespace: <my-kapp-namespace>
rules:
- apiGroups:
  - ""
  resources:
    - configmaps
  verbs:
    - '*'

```

Where the namespace `<my-kapp-namespace>` must be the namespace of the Kubernetes context that `kapp` will be run in.

## Installing Windows Components (Beta)

**Warning:** This feature is in Beta.

**Note:** TBS on Windows does not currently support self-signed registry certificates. Please use a public registry or a non-self-signed cert.

Tanzu Build Service supports building [.NET Framework](#) application images. Building .NET Framework images will require a Kubernetes Cluster with windows nodes provisioned.

After the windows nodes are provisioned, the Tanzu Build Service Windows Dependencies (Stacks, Buildpacks, Builders, etc.) can be used to build .NET Framework applications and keep them patched. These must be imported with the `kp` cli and the Dependency Descriptor (`windows-`

`descriptor-<version>.yaml`) file from the [Tanzu Build Service Dependencies for Microsoft Windows](#) page:

```
kp import -f /tmp/windows-descriptor-<version>.yaml
```

The following features are not yet supported on windows nodes of Tanzu Build Service

- Caching of build artifacts (which reduces the time of subsequent builds)
- Preloading of ClusterBuilder images
- Self-signed registry certificate

## Getting Started with Tanzu Build Service

This topic describes how to get started with a typical installation of Tanzu Build Service and create an Image.

This page is meant to serve as a quick-start guide and may not include some configurations required for your specific environment. For more details on installation, see [Installing Tanzu Build Service](#).

## Prerequisites

Before you install Build Service, you must:

- Have access to the Kubernetes cluster satisfying the [minimum required permissions](#).
- Ensure your Kubernetes cluster is configured with default `StorageClass`. Tanzu Build Service will default to using 2G of cache if a default `StorageClass` is defined. Build Service utilizes `PersistentVolumeClaims` to cache build artifacts, which reduces the time of subsequent builds.

For more information, see [Persistent Volumes](#) in the Kubernetes documentation. And for information on defining a default StorageClass, see [Changing the default StorageClass](#)

- Download three [Carvel](#) CLIs for your operating system. These tools will facilitate the installation of Tanzu Build Service on your cluster. They can be found on their respective Tanzu Network pages:
  - ◊ [kapp](#) is a deployment tool that allows users to manage Kubernetes resources in bulk.
  - ◊ [ytt](#) is a templating tool that understands YAML structure.
  - ◊ [kblid](#) is tool that builds, pushes, and relocates container images.
- Download the Build Service Bundle from the [Tanzu Build Service](#) page on Tanzu Network.
  - ◊ Unarchive the Build Service Bundle file:

```
tar xvf build-service-<version>.tar -C /tmp
```

- Download the `kp` CLI for your operating system from the [Tanzu Build Service](#) page on Tanzu Network. The `kp` CLI help text is published [here](#).
  - ◊ These docs assume `kp cli v0.2.*` from TBS release `v1.1.*`. If a feature is not working, you may need to upgrade your cli.

- Download the `docker` cli to authenticate with registries.
- Download the Dependency Descriptor file (`descriptor-<version>.yaml`) from the latest release on the [Tanzu Build Service Dependencies](#) page on Tanzu Network. This file contains paths to images that contain dependency resources Tanzu Build Service needs to execute image builds.

## Assumptions

For this example setup, we will make the following assumptions:

- You are using a registry named `my.registry.io` with credentials
  - ◊ Username: `my-user`
  - ◊ Password: `my-password`
- Your registry uses a self-signed CA certificate and you have access to the cert in a file `/tmp/ca.crt`
  - ◊ The nodes on your cluster must also be configured to trust this CA certificate so they can pull in images. Configuration for this depends on the cluster provider
- You are using an "online" environment that has access to the internet

## Installation

### Relocate Images to a Registry

This procedure relocates images from the Tanzu Network registry to your registry.

1. Log in to your image registry:

```
docker login my.registry.io --tlscacert /tmp/ca.crt
```

2. Log in to the Tanzu Network registry with your Tanzu Network credentials:

```
docker login registry.pivotal.io
```

3. Relocate the images with the [Carvel](#) tool `kbld` by running:

```
kbld relocate -f /tmp/images.lock --lock-output /tmp/images-relocated.lock --repository my.registry.io/tbs --registry-ca-cert-path /tmp/ca.crt
```

**Note:** The flag argument `--lock-output /tmp/images-relocated.lock` creates a file that will be used for installation.

### Install Tanzu Build Service

Use the [Carvel](#) tools `kapp`, `ytt`, and `kbld` to install Build Service and define the required Build Service parameters by running:

```
ytt -f /tmp/values.yaml \
  -f /tmp/manifests/ \
  -f /tmp/ca.crt \
```

```
-v docker_repository="my.registry.io/tbs" \
-v docker_username="my-user" \
-v docker_password="my-password" \
| kblid -f /tmp/images-relocated.lock -f- \
| kapp deploy -a tanzu-build-service -f- -y
```

## Import Tanzu Build Service Dependencies

The Tanzu Build Service Dependencies (Stacks, Buildpacks, Builders, etc.) are used to build applications and keep them patched.

These must be imported with the `kp` cli and the Dependency Descriptor (`descriptor-<version>.yaml`) file from the [Tanzu Build Service Dependencies](#) page:

```
kp import -f /tmp/descriptor-<version>.yaml --registry-ca-cert-path /tmp/ca.crt
```

Successfully performing a `kp import` command requires that your Tanzu Network account has access to the images specified in the Dependency Descriptor file. Users can only access these images if they agree to the dependency EULAs.

Users must navigate to the following dependencies pages in Tanzu Network and accept all EULAs highlighted in yellow.

1. [Tanzu Build Service Dependencies](#)
2. [Buildpacks for VMware Tanzu](#)
3. [Stacks for VMware Tanzu](#)

**Note:** `kp import` will fail if it cannot access the images in all of the above Tanzu Network pages.

## Verify Installation

To verify your Build Service installation:

List the cluster builders available in your installation:

```
kp clusterbuilder list
```

You should see an output that looks as follows:

NAME	READY	STACK	IMAGE
base	true	io.buildpacks.stacks.bionic	<image@sha256:digest>
default	true	io.buildpacks.stacks.bionic	<image@sha256:digest>
full	true	io.buildpacks.stacks.bionic	<image@sha256:digest>
tiny	true	io.paketo.stacks.tiny	<image@sha256:digest>

## Create an Image

You can now create a Tanzu Build Service Image to start building your app and keep it patched with the latest Stack and Buildpack Dependencies.

We will assume you are using the `default` namespace, use `-n` when using `kp` to set a specific namespace.

1. Create a Kubernetes Secret that will allow your Builds to push to the desired registry with the `kp` cli:

```
kp secret create my-registry-creds --registry my.registry.io --registry-user my-user
```

You will be prompted for your password (`my-password`).

2. Create the Tanzu Build Service Image:

We will use a [sample java-maven app](#):

```
kp image create my-image --tag my.registry.io/tbs/test-app --git https://github.com/buildpacks/samples --sub-path ./apps/java-maven --wait
```

# Accessing Tanzu Build Service

To use your Build Service installation, gain kubeconfig access to the Kubernetes Cluster that has the Build Service installed. For example, if you are using TKGI (formerly PKS):

```
tkgi login -a <tkg-api-url> -u <username> -p <password>
tkgi get-credentials <clustername> -a <tkg-api-url> -u <username> -p <password>
```

You can use the `kp` CLI, downloaded as part of the [installation](#) to interact with Build Service. The `kp` CLI uses the local `KUBECONFIG` utilized by `kubectl`. All operations will be performed on kubernetes current-context namespace.

The `kp` CLI help text is published [here](#).

```
$ kp
kp controls the kpack installation on Kubernetes.

kpack extends Kubernetes and utilizes unprivileged kubernetes primitives to provide
builds of OCI images as a platform implementation of Cloud Native Buildpacks (CNB).
Learn more about kpack @ https://github.com/pivotal/kpack

Usage:
  kp [command]

Available Commands:
  build           Build Commands
  builder        Builder Commands
  clusterbuilder Cluster Builder Commands
  clusterstack   Cluster Stack Commands
  clusterstore   Cluster Store Commands
  completion     Generate completion script
  help           Help about any command
  image          Image commands
  import         Import dependencies for stores, stacks, and cluster builders
  secret         Secret Commands
  version        Display kp version

Flags:
  -h, --help    help for kp

Use "kp [command] --help" for more information about a command.
```

**Note:** These docs assume `kp` cli v0.2.\* from TBS release v1.1.\*. If a feature is not working, you may need to upgrade your cli.

# Updating Build Service Dependencies

Keeping applications up-to-date with the latest dependency patches is a core feature of Tanzu Build Service. Updates to dependencies will be propagated to application images. The resources that account for these patches are:

- **ClusterStacks** - Update a ClusterStack to patch operating system packages.
- **ClusterStores** - Update a ClusterStore to patch the Cloud Native Buildpacks used to build your applications.

You can use the `kp` CLI to update any resource. The help text is published [here](#).

**Note:** These docs assume `kp cli v0.2.*` from TBS release `v1.1.*`. If a feature is not working, you may need to upgrade your cli.

## Updating Dependencies

### Bulk Update

**Note:** If you want to be alerted when a new descriptor file is published, we recommend using an RSS reader and watching the Tanzu Build Service Dependencies TanzuNet feed for updates <https://network.pivotal.io/rss>

The Bulk Update workflow can update all dependencies (ClusterStacks, ClusterStores and ClusterBuilders) in Tanzu Build Service using the `kp import` command.

1. Download the Dependency Descriptor file (`descriptor-<version>.yaml`) from the latest release on the [Tanzu Build Service Dependencies](#) page on Tanzu Network.

**Note:** You can see all of the buildpackages versions that will be imported by looking at the `'buildpackage-versions-.yaml'` file from the [Tanzu Build Service Dependencies] (<https://network.pivotal.io/products/tbs-dependencies/>) release.

2. Use the `kp` CLI

```
kp import -f descriptor-<version>.yaml
```

The following ClusterStacks will be updated with the latest Operating System patches: `base`, `default`, `full`, and `tiny`.

The following ClusterStore will be updated with the latest Cloud Native Buildpacks: `default`

Using the `--show-changes` flag will give a summary of the resource changes for the import. You will also be asked to confirm the import. Confirmation can be skipped with `--force`.

### Cluster Stacks Update

This section described how to update individual cluster stacks. This provides a more fine-grained way to patch operating system packages.

New stack versions will be provided on the [Tanzu Build Service Dependencies](#) page on Tanzu Network.

To update specific cluster stacks, go to the latest release of the [Tanzu Build Service Dependencies](#) page on Tanzu Network to find the image references and their `<sha256>` sums. Example commands will be provided on this page.

Use the following `kp` CLI commands to update the desired stack:

```
kp clusterstack update base \
  --build-image registry.pivotal.io/tbs-dependencies/build-base@<sha256> \
  --run-image registry.pivotal.io/tbs-dependencies/run-base@<sha256>

kp clusterstack update default \
  --build-image registry.pivotal.io/tbs-dependencies/build-full@<sha256> \
  --run-image registry.pivotal.io/tbs-dependencies/run-full@<sha256>

kp clusterstack update full \
  --build-image registry.pivotal.io/tbs-dependencies/build-full@<sha256> \
  --run-image registry.pivotal.io/tbs-dependencies/run-full@<sha256>

kp clusterstack update tiny \
  --build-image registry.pivotal.io/tbs-dependencies/build-tiny@<sha256> \
  --run-image registry.pivotal.io/tbs-dependencies/run-tiny@<sha256>
```

**Note:** Both build and run images need to be provided to update the stack.

The updated ClusterStack can be viewed with the following command:

```
kp clusterstack status <stack-name>
```

Example output

```
$ kp clusterstack status tiny
Status:      Ready
Id:          io.paketo.stacks.tiny
Run Image:   gcr.io/build-service-dev/test/run@sha256:34b01fd9a3745fcaa345f89939382
91c931f7977cc2bee78ed377da2edc55e3d
Build Image: gcr.io/build-service-dev/test/build@sha256:5288d9c5b7cf7068d07b5a184f3
ec2f124fbc5842401b8b23c74485c4d2ba23a
```

## Cluster Store Update

ClusterStores contain all of the buildpackages (one or more packaged Cloud Native Buildpacks) to be used by Builders to build application images.

You can update Cloud Native Buildpacks in Tanzu Build Service by adding new buildpackage versions to the store.

To list the buildpackages available in a store:

```
kp clusterstore status <store-name>
```

## Example output

```
$ kp clusterstore status default
Status:    Ready

BUILDPACKAGE ID          VERSION    HOMEPAGE
paketo-buildpacks/procfile  1.4.0     https://github.com/paketo-buildpacks/procfile
tanzu-buildpacks/dotnet-core  0.0.3
tanzu-buildpacks/go        1.0.5
tanzu-buildpacks/httpd     0.0.38
tanzu-buildpacks/java      2.5.0     https://github.com/pivotal-cf/tanzu-java
tanzu-buildpacks/nginx     0.0.45
tanzu-buildpacks/nodejs    1.1.0
tanzu-buildpacks/php       0.0.3
```

To show a complete list of all buildpacks available in a store:

```
kp clusterstore status <store-name> --verbose
```

Update a store with one or more buildpackages with:

```
kp clusterstore add <store-name> -b <buildpackage-image1> -b <buildpackage-image2>
```

**Note:** Any number of buildpackages can be added to a store at a time with multiple `-b` flags.`

## Updating Buildpacks From Tanzu Network

New Cloud Native Buildpacks (packaged as buildpackages) will be available on [Tanzu Network](#) and should be uploaded to a Tanzu Build Service to keep application images patched.

New versions of the Java, NodeJS, and Go buildpacks will be released on their respective Tanzu Network pages: [Java](#), [NodeJS](#) and [Go](#). New versions of all other buildpacks will be released on the [Tanzu Build Service Dependencies](#) page.

Here is a list of how to update each buildpack that is included with Tanzu Build Service by default:

```
kp clusterstore add default -b registry.pivotal.io/tanzu-java-buildpack/java:<version>
kp clusterstore add default -b registry.pivotal.io/tanzu-nodejs-buildpack/nodejs:<version>
kp clusterstore add default -b registry.pivotal.io/tanzu-go-buildpack/go:<version>
kp clusterstore add default -b registry.pivotal.io/tbs-dependencies/tanzu-buildpacks_dotnet-core:<version>
kp clusterstore add default -b registry.pivotal.io/tbs-dependencies/tanzu-buildpacks_php:<version>
kp clusterstore add default -b registry.pivotal.io/tbs-dependencies/tanzu-buildpacks_nginx:<version>
kp clusterstore add default -b registry.pivotal.io/tbs-dependencies/tanzu-buildpacks_httpd:<version>
kp clusterstore add default -b registry.pivotal.io/tbs-dependencies/paketo-buildpacks_procfile:<version>
```

Additionally, multiple buildpackages can be added to Build Service by passing multiple image references:

```
kp clusterstore add <store-name> \
  -b registry.pivotal.io/buildpackage-1 \
  -b registry.pivotal.io/buildpackage-2 \
  -b registry.pivotal.io/buildpackage-3
```

## Offline Update of Dependencies

The stack images and buildpacks used by build service can be updated by first downloading those images and saving them as a .tar file. This file can be provided to the `kp` CLI to import to Tanzu Build Service.

1. Download the Dependency Descriptor file (`descriptor-<version>.yaml`) from the latest release on the [Tanzu Build Service Dependencies](#) page on Tanzu Network.
2. Download the `kp` CLI for your operating system from the latest release on the [Tanzu Build Service](#) page.
3. Download the `kbld` CLI for your operating system from the latest release on the [kbld](#) page.
4. Download the dependency images for Tanzu Build Service to your local machine with `kbld`:

```
docker login registry.pivotal.io

kbld package -f descriptor-<version>.yaml \
  --output /tmp/packaged-dependencies.tar
```

5. Move the output file `packaged-dependencies.tar` to a machine that has access to the "offline" environment
6. Upload the dependency images to the internal registry used to deploy Tanzu Build Service:

```
docker login <build-service-registry>

kbld unpack -f descriptor-<version>.yaml \
  --input /tmp/packaged-dependencies.tar \
  --repository <IMAGE-REPOSITORY> \
  --lock-output /tmp/dependencies-relocated.lock
```

Where `IMAGE-REPOSITORY` is the repository used to install Tanzu Build Service. This should be the same value as `IMAGE-REPOSITORY` used in the [Installation Steps](#).

7. Now that dependencies are relocated to the internal registry, you can use the following command to update the necessary resources:

```
kbld -f descriptor-<version>.yaml -f /tmp/dependencies-relocated.lock | kp import -f -
```

# Managing Secrets

## Overview

VMware Tanzu Build Service uses Kubernetes secrets to manage credentials.

- To publish images to a Registry, you must use a Registry secret.
- To use source code stored in a private Git repository, you must use a Git secret.

Secrets are namespaced and can only be used for image configurations that exist in the same namespace. For more information about Kubernetes secrets, see [Secrets](#) in the Kubernetes documentation.

For more information about secret synchronization, see the [Secret Synchronization](#) page.

You can manage secrets with the `kp` CLI. The help text is published [here](#).

```
$ kp secret
Secret Commands

Usage:
  kp secret [command]

Available Commands:
  create      Create a secret configuration
  delete      Delete secret
  list        List secrets

Flags:
  -h, --help  help for secret

Use "kp secret [command] --help" for more information about a command.
```

**Note:** These docs assume `kp cli v0.2.*` from TBS release `v1.1.*`. If a feature is not working, you may need to upgrade your cli.

## Create Secrets

You can create secrets using the `kp` CLI and script them with environment variables.

Secrets are created in the Kubernetes `current-context` namespace, unless you specify a different namespace using the `--namespace` or `-n` flag. Kubernetes automatically adds these secrets to the `default` service account in the same namespace.

**Note:** The `kp` CLI does not validate the secret against the specified registry or Git at the time of secret creation. Incorrect credentials will be reported as they are used during an image build.

## Create a Docker Hub Registry Secret

You can create a Docker Hub registry secret using the `--dockerhub` flag.

```
kp secret create SECRET-NAME --dockerhub DOCKER-HUB-ID
```

Where:

- `SECRET-NAME` is the name you give your secret.
- `DOCKER-HUB-ID` is your Docker Hub user ID.

When prompted, enter your Docker Hub password. Alternatively, you can use the `DOCKER_PASSWORD` environment variable to bypass the password prompt.

The Docker Hub registry secret is stored as a `kubernetes.io/dockerconfigjson` secret.

Examples:

```
$ kp secret create secret1 --dockerhub my-dockerhub-id
dockerhub password:
"secret1" created

$ DOCKER_PASSWORD="my-password" kp secret create secret2 --dockerhub my-dockerhub-id
"secret2" created
```

## Create a GCR Registry Secret

You can create a GCR registry secret using the `--gcr` flag.

```
kp secret create SECRET-NAME --gcr GCR-SERVICE-ACCOUNT-PATH
```

Where:

- `SECRET-NAME` is the name you give your secret.
- `GCR-SERVICE-ACCOUNT-PATH` is the path to your GCR service account json file.

Alternatively use the `GCR_SERVICE_ACCOUNT_PATH` environment variable instead of the `--gcr` flag.

The GCR registry secret is stored as a `kubernetes.io/dockerconfigjson` secret.

Examples:

```
$ kp secret create secret1 --gcr /tmp/my-gcr-service-account.json
"secret1" created

$ GCR_SERVICE_ACCOUNT_PATH="/tmp/my-gcr-service-account.json" kp secret create secret2
"secret2" created
```

## Create an Artifactory, Harbor, or ACR Registry Secret

You can create an Artifactory, Harbor, or ACR secret using the `--registry` and `--registry-user` flags.

```
kp secret create SECRET-NAME --registry REGISTRY-URL --registry-user REGISTRY-USER-ID
```

Where:

- `SECRET-NAME` is the name you give your secret.
- `REGISTRY-URL` is the URL of the registry. This should only be the domain for the registry and should not contain folders or projects. Example: `registry.io` and not `registry.io/project`.
- `REGISTRY-USER-ID` is your registry user ID.

When prompted, enter your registry password. Alternatively, you can use the `REGISTRY_PASSWORD` environment variable to bypass the password prompt.

The Artifactory, Harbor, or ACR registry secret is stored as a `kubernetes.io/dockerconfigjson` secret.

Examples:

```
$ kp secret create secret1 \
  --registry registry.pivotal.io \
  --registry-user someuser@pivotal.io
registry password:
"secret1" created

$ REGISTRY_PASSWORD="my-password" kp secret create secret2 \
  --registry registry.pivotal.io \
  --registry-user someuser@pivotal.io
"secret2" created
```

## Create a Git SSH Secret

You can create a Git SSH secret by specifying a Git SSH URL and private SSH key.

```
kp secret create SECRET-NAME --git-url GIT-SSH-URL --git-ssh-key PRIVATE-SSH-KEY-PATH
```

Where:

- `SECRET-NAME` is the name you give your secret.
- `GIT-SSH-URL` is the Git SSH domain URL. This is not the full repository URL. For example, value should be `git@github.com` for GitHub.
- `PRIVATE-SSH-KEY-PATH` is the path to your private SSH key.

Alternatively, use the `GIT_SSH_KEY_PATH` environment variable instead of the `--git-ssh-key` flag.

The Git SSH secret is stored as a `kubernetes.io/ssh-auth` secret.

Examples:

```
$ kp secret create secret1 \
  --git-url git@github.com \
  --git-ssh-key /tmp/private-repo-git-deploy-key
"secret1" created

$ GIT_SSH_KEY_PATH="/tmp/private-repo-git-deploy-key" kp secret create secret2 \
  --git-url git@github.com \
  "secret2" created
```

## Create a Git Basic Auth Secret

You can create a Git basic auth secret by providing your Git username and password

```
kp secret create SECRET-NAME --git-url GIT-DOMAIN-URL --git-user GIT-USERNAME
```

Where:

- `SECRET-NAME` is the name you give your secret.
- `GIT-DOMAIN-URL` is the Git domain url. This is not the full repository url. For example, value should be `https://github.com` for GitHub.
- `GIT-USERNAME` is your Git username.

When prompted, enter your Git password. Alternatively, you can use the `GIT_PASSWORD` environment variable to bypass the password prompt.

The Git basic auth secret is stored as a `kubernetes.io/basic-auth` secret.

Examples:

```
$ kp secret create secret1 \
  --git-url https://github.com \
  --git-user someone@vmware.com
git password:
"secret1" created

$ GIT_PASSWORD="my-password" kp secret create secret2 \
  --git-url https://github.com \
  --git-user someone@vmware.com
"secret2" created
```

## List Secrets

To list the names and the targets for your secrets:

```
kp secret list
```

Unless you specify a namespace using the `--namespace` or `-n` flag, running the `kp secret list` command lists secrets for the Kubernetes `current-context` namespace.

Example:

```
$ kp secret list
NAME                TARGET
default-token-qrdbr
docker-hub-creds    https://index.docker.io/v1/
gcr-creds           gcr.io
git-creds           https://github.com
git-ssh-creds       git@github.com
harbor-creds        registry.pivotal.io
```

The `default-token-xxxxxx` secret is automatically added to the `default` service account by Kubernetes

## Delete Secrets

To delete secrets:

```
kp secret delete SECRET-NAME
```

Where `SECRET-NAME` is the name of the secret you want to delete.

Unless you specify a namespace using the `--namespace` or `-n` flag, secrets are deleted from the Kubernetes `current-context` namespace. There is no confirmation required from the user.

## Encrypting Secrets at Rest

Because Tanzu Build Service uses standard Kubernetes secrets, administrators may configure the cluster to encrypt secrets at rest. For more information, see the following link:

<https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/>

## Synced-Secrets in Tanzu Build Service

### When to use Synchronized Secrets

To enable the use of Cluster Builders from private registries, a Secret with registry credentials must exist in the namespace of the Image using that Cluster Builder.

You can configure this secret manually for each namespace, but Tanzu Build Service provides functionality to synchronize secrets across namespaces to simplify this process.

This feature is applicable in the following cases:

- You have installed Tanzu Build Service to a private registry and do not wish to make Cluster Builders imported by `kp` publicly readable.
- You have used `kp` to create a Cluster Builder in a private registry and do not wish to make it publicly readable.

Synchronized secrets are attached to build pods as `imagePullSecrets` so that the Cluster Builder Image can be pulled at build time.

**Note:** Synchronizing secrets with write access is not recommended. Instead, create and synchronize read-only secrets. A synced secret is not created during installation because the credentials provided for installation must be writable.

## Managing Secret Synchronization

Currently, the `kp` CLI does not support adding and removing synchronized secrets. However, this may be achieved by using the `kubectl` CLI.

### Create a Synchronized Secret

To start synchronizing a secret to all namespaces with builds, use `kubectl` to create a `docker-registry` (Dockercfg or DockerConfigJson) secret in the `build-service` namespace with the following label: `com.vmware.tanzu.buildservice.sync=true`.

### Example Secret:

```

apiVersion: v1
data:
  .dockerconfigjson: <SECRET DATA>
kind: Secret
metadata:
  labels:
    com.vmware.tanzu.buildservice.sync: "true"
  name: my-synced-secret
  namespace: build-service
type: kubernetes.io/dockerconfigjson

```

### Example configuration steps:

```

# Remove current local docker config
rm ~/.docker/config.json

# Login locally with READ-ONLY creds
docker login my-registry.io -u <read-only-user> -p <read-only-password>

# Create kubernetes Secret
cat <<EOF | kubectl apply -f-
apiVersion: v1
data:
  .dockerconfigjson: $(cat ~/.docker/config.json | base64)
kind: Secret
metadata:
  labels:
    com.vmware.tanzu.buildservice.sync: "true"
  name: my-synced-secret
  namespace: build-service
type: kubernetes.io/dockerconfigjson
EOF

```

## Update a Synchronized Secret

To update a secret and roll-out those changes to all namespaces that use Builds, simply update the secret(s) with the `com.vmware.tanzu.buildservice.sync=true` label located in the `build-service` namespace.

## Stop Synchronizing a Secret

To stop synchronizing a secret, delete the secret from the `build-service` namespace or remove the `com.vmware.tanzu.buildservice.sync=true` label from the secret located in the `build-service` namespace.

# Managing Images and Builds

This topic contains the following sections:

- [Images](#)
- [Builds](#)

The `kp` CLI can be used to manage images and builds. The help text is published [here](#).

```
$ kp image
Image commands

Usage:
  kp image [command]

Aliases:
  image, images, imgs, img

Available Commands:
  create      Create an image configuration
  delete      Delete an image
  list        List images
  patch       Patch an existing image configuration
  save        Create or patch an image configuration
  status      Display status of an image
  trigger     Trigger an image build

Flags:
  -h, --help  help for image

Use "kp image [command] --help" for more information about a command.
```

```
$ kp build
Build Commands

Usage:
  kp build [command]

Aliases:
  build, builds, blds, bld

Available Commands:
  list        List builds for an image
  logs        Tails logs for an image build
  status      Display status for an image build

Flags:
  -h, --help  help for build
```

```
Use "kp build [command] --help" for more information about a command.
```

**Note:** These docs assume kp cli v0.2.\* from TBS release v1.1.\*. If a feature is not working, you may need to upgrade your cli.

## Images

Images provide a configuration for Tanzu Build Service to build and maintain a Docker image utilizing Tanzu, Paketo, and custom [Cloud Native Buildpacks](#).

Build Service will monitor the inputs to the image configuration to rebuild the image when the underlying source or buildpacks have changed.

The following procedures describe how to create and manage images in Build Service with the `kp` CLI.

### Creating Images

Prerequisites:

- [Access to a cluster](#) running Build Service.
- Configured write [secrets](#) for your Docker registry.

#### Source Code

The `kp` CLI supports creating Images using source code from the following locations:

- [Git based source](#)
- [Blob store](#)
- [Local machine](#)

You can specify only one location for app source code.

#### Builders

Users can select a Builder (namespaced-scoped) or a Cluster Builder (cluster-scoped) to be used to create image builds. You can use any of the available Builders or Cluster Builders with any of the source types (git, blob, or local).

If you do not use the `--builder` or `--cluster-builder` flags, the `default` Cluster Builder will be used.

For more information on Builders, see [Managing Builders](#).

### Creating an Image With Source Code in a Git Repository

To create an image using source code from a git repository run:

```

kp image create <name> \
  --tag <tag> \
  [--builder <builder> or --cluster-builder <cluster-builder>] \
  --namespace <namespace> \
  --env <env> \
  --sub-path <sub-path> \
  --wait \
  --git <git-repo> \
  --git-revision <git-revision>

```

Where:

- **name**: The name of the image.
- **tag**: The registry location where the image will be created.
- **builder**: (optional) Builder name to be used in the image. Cannot be used with **cluster-builder**.
- **cluster-builder**: (optional) Cluster Builder name to be used in the image. Defaults to **default** when **builder** is not set. Cannot be used with **builder**.
- **namespace**: (optional) The Kubernetes namespace for the image. Defaults to the local Kubernetes current-context namespace.
- **env** (optional): Image environment variable configuration as key=val pairs (**env\_var=env\_val**). The **--env** flag can be specified multiple times.
- **sub-path** (optional): Build code at the sub path located within the source code directory.
- **cache-size** (optional): The cache size used for subsequent builds. Must be a valid kubernetes quantity (default 2G).
- **wait** flag (optional) Waits for image create to be reconciled and tails resulting build logs.
- **git-repo** Git repository URL of the source code.
- **git-revision** (optional) The Git revision of the code that the image is built against. Can be either a **branch**, **tag** or a commit **sha**. When you target the image against a branch, Build Service triggers a build for every new commit. Defaults to **master**.

**Note:** If the **git-repo** is a private repository, you must configure the git credentials. For more information, see [Create Secrets](#).

## Create an Image With Source Code In A Blob Store

Users can specify source code in a blob store saved as a compressed file (**zip**, **tar.gz**, **.tar**) or a **.jar** file.

To create an image using source code from blob store:

```

kp image create <name> \
  --tag <tag> \
  [--builder <builder> or --cluster-builder <cluster-builder>] \
  --namespace <namespace> \
  --env <env> \
  --sub-path <sub-path> \
  --wait \

```

```
--blob <blob-url>
```

Where:

- `name`: The name of the image.
- `tag`: The registry location where the image will be created.
- `builder`: (optional) Builder name to be used in the image. Cannot be used with `cluster-builder`.
- `cluster-builder`: (optional) Cluster Builder name to be used in the image. Defaults to `default` when `builder` is not set. Cannot be used with `builder`.
- `namespace`: (optional) The Kubernetes namespace for the image. Defaults to the local Kubernetes current-context namespace.
- `env` (optional): Image environment variable configuration as key=val pairs (`env_var=env_val`). The `--env` flag can be specified multiple times.
- `sub-path` (optional): Build code at the sub path located within the source code directory.
- `cache-size` (optional): The cache size used for subsequent builds. Must be a valid kubernetes quantity (default 2G).
- `wait` flag (optional) Waits for image create to be reconciled and tails resulting build logs.
- `blob-url` URL of the source code blob file.

**Note:** The source code file in the blob store must be publicly viewable or the `blob-url` must contain the basic authentication credentials.

## Creating an Image With Local Source Code

Users can apply local source code from a directory, compressed source code (`zip`, `tar.gz`, `.tar`), or a `.jar` file.

To create an image using source code from a local machine run:

```
kp image create <name> \
  --tag <tag> \
  --local-path <source-path> \
  [--builder <builder> or --cluster-builder <cluster-builder>] \
  --namespace <namespace> \
  --env <env> \
  --cache \
  --registry-ca-cert-path <path-to-ca-cert> \
  --registry-verify-certs
```

Where:

- `name`: The name of the image.
- `tag`: The registry location where the image will be created.
- `source-path` Path to local source code.
- `builder`: (optional) Builder name to be used in the image. Cannot be used with `cluster-builder`.

- `cluster-builder`: (optional) Cluster Builder name to be used in the image. Defaults to `default` when `builder` is not set. Cannot be used with `builder`.
- `namespace`: (optional) The Kubernetes namespace for the image. Defaults to the local Kubernetes current-context namespace.
- `env` (optional): Image environment variable configuration as key=val pairs (`env_var=env_val`). The `--env` flag can be specified multiple times.
- `cache-size` (optional): The cache size used for subsequent builds. Must be a valid kubernetes quantity (default 2G).
- `--wait` flag (optional) Waits for image create to be reconciled and tails resulting build logs.
- `registry-ca-cert-path` (optional) Add CA certificate for registry API
- `registry-verify-certs` (optional) Set whether to verify server's certificate chain and host name (default true)

## Buildpack Configuration

Images use buildpacks to build application images in a registry. The buildpacks contain the dependencies needed for these builds and you can add buildpack configuration to Tanzu Build Service Images.

### Buildpack Configuration Use Cases

Common use cases for setting buildpack configuration include:

- Selecting a specific version or version line of a dependency (Go 1.15.\*, Java 1.8)
- Language-specific configuration (Go build target)
- Buildpack-specific configuration

### Buildpack Configuration Documentation

Buildpack configuration details can be found in the documentation for that specific buildpack.

Use `kp clusterstore status <store-name> --verbose` to find the homepage of the desired buildpack.

### Buildpack Configuration in Images

Buildpack configuration – including manually selecting buildpacks to use – can be set in two ways in Tanzu Build Service Images. The configuration depends on the specific buildpack, find buildpack details in [Buildpack Configuration Documentation](#).

1. Creating a `buildpack.yml` file at the root of the application source code.

Example `buildpack.yml` for a Go app to use the latest Go 1.15 version and build with the path `./cmd/package`:

```
go:
  version: 1.15.*
```

```
targets: ["/cmd/package"]
```

## 2. Setting environment variables on an Image.

Tanzu Build Service Images can have environment variables configured which will be set in all Builds and in the final exported registry image. These can be used for buildpack configuration.

Example `kp` command to create an image for a Go app to build with the path `./cmd/package`:

```
kp image create my-image \
  --tag registry.io/my-repo \
  --git https://github.com/my-go-app \
  --env BP_GO_TARGETS="./cmd/package"
```

## Patching Images

Users can patch their existing images with the `kp` CLI. Running a patch will trigger a new build of the image if any of the build inputs are changed.

**Note:**For handling source code changes in the Tanzu Build Service process, we recommend utilizing the `kp image save --wait` command within a CI/CD pipeline to update the source code referenced in the image configuration.

This can be accomplished by updating the `--git-revision` field with a new commit ID. For many TBS customers this commit ID references source code that has undergone unit testing, so that they can be confident that the resulting image can be deployed or promoted to higher level environments.

Patch images with the following commands:

- With Source Code in a Git Repository

```
kp image patch <name> \
  [--builder <builder> or --cluster-builder <cluster-builder>] \
  --namespace <namespace> \
  --env <env> \
  --wait \
  --git <git-repo> \
  --git-revision <git-revision>
```

- With Source Code In A Blob Store

```
kp image patch <name> \
  [--builder <builder> or --cluster-builder <cluster-builder>] \
  --namespace <namespace> \
  --env <env> \
  --wait \
  --blob <blob-url>
```

- With Local Source Code

```
kp image patch <name> \
  [--builder <builder> or --cluster-builder <cluster-builder>] \
  --namespace <namespace> \
  --env <env> \
  --wait \
  --local-path <source-path>
```

Where:

- `name`: The name of the image to patch.
- `namespace`: (optional) The Kubernetes namespace for the image. Defaults to the local Kubernetes current-context namespace.
- `env` (optional): Image environment variable configuration as key=val pairs (`env_var=env_val`). The `--env` flag can be specified multiple times.
- `cache-size` (optional): The cache size used for subsequent builds. Must be a valid kubernetes quantity (default 2G).
- `git-repo` Git repository URL of the source code. Must select one of `git-repo`, `blob-url`, or `source-path`
- `git-revision` (optional) The Git revision of the code that the image is built against. Can be either a `branch`, `tag` or a commit `sha`. When you target the image against a branch, Build Service triggers a build for every new commit. Defaults to `master`.
- `blob-url` URL of the source code blob file. Must select one of `git-repo`, `blob-url`, or `source-path`
- `source-path` Path to local source code. Must select one of `git-repo`, `blob-url`, or `source-path`

**Note:** If the `git-repo` is a private repository, you must configure the git credentials. For more information, see [Create Secrets](#).

**Note:** The `tag` location in a registry and `name` of an image cannot be modified. To change these fields, you must create a new image.

## Saving Images

Users can create or patch an Image using the `save` command. The `kp image save` command is used exactly the same as `kp image create` or `kp image patch`, but it will determine if a image needs to be created or patched.

```
kp image save my-image \
  --tag my-registry.com/my-repo \
  --git https://my-repo.com/my-app.git \
  --git-revision my-branch
```

## Listing Images

To list all the image configurations in a Kubernetes namespace:

```
kp image list --namespace <namespace>
```

Example

```
$ kp image list -n example1
```

NAME	READY	LATEST REASON	LATEST IMAGE	NAMESPACE
test-image1	True	CONFIG	first/image:sha	example1
test-image2	False	BUILDPACK	second/image:sha	example1

To list all the image configurations across all Kubernetes namespaces:

```
kp image list --all-namespaces
```

### Example

```
$ kp image list -A
```

NAME	READY	LATEST REASON	LATEST IMAGE	NAMESPACE
test-image1	True	CONFIG	first/image:sha	example1
test-image2	True	BUILDPACK	second/image:sha	example1
test-image3	True	BUILDPACK	third/image:sha	example2
test-image4	False	CONFIG	fourth/image:sha	example2

## Filter Images

Users can further filter the list of Images by applying the `--filter` flag and specifying a filter and value. This command is useful for traversing large number of Image configurations by narrowing the list to only display Images that possess certain attributes.

```
$ kp image list --filter ready=false -A
```

NAME	READY	LATEST REASON	LATEST IMAGE	NAMESPACE
test-image2	False	BUILDPACK	second/image:sha	example1
test-image4	False	CONFIG	fourth/image:sha	example2

See below for the current supported filters and values:

```
builder=string
clusterbuilder=string
latest-reason=commit,trigger,config,stack,buildpack
ready=true,false,unknown
```

## Image Rebuilds

Rebuilds happen in three ways:

1. An imperative rebuild occurs when you patch an image with `kp image patch`.
2. An automatic rebuild occurs when build inputs change (source code, stack, or buildpacks).
3. A user can trigger a rebuild manually.

An imperative rebuild will be initiated if any of the following changes are made to an image:

- An update to the commit, branch, Git repository, or other arguments to `kp image patch`.
- You upload a new copy of the local source code by running `kp image patch --local-path <source-path>`, where `<source-path>` is the source code path.

For more information, see [Patching Images](#).

Build Service auto-rebuilds images when one or more of the following build inputs change:

- New buildpack versions are made available via updates to a Cluster Store.
  - ◊ New Buildpack versions are made available on [Tanzu Network](#).
  - ◊ To update buildpacks, you must add new buildpack versions from Tanzu Network to a Cluster Store. See [Updating Build Service Dependencies](#) for more details.
- There is a new commit on a branch or tag Tanzu Build Service is tracking.
- There is a new Cluster Stack (ie. base OS image) available, such as `full`, `tiny`, or `base`.
  - ◊ New Stack versions are made available on the [Tanzu Build Service Dependencies](#) page on Tanzu Network.
  - ◊ You can get updates to Stacks from the Tanzu Network Registry by using the `kp` CLI. See [Updating Build Service Dependencies](#) for more details.

## Trigger an Image Rebuild

You can initiate a manual rebuild using `kp`:

```
kp image trigger <image-name> --namespace <namespace>
```

This is useful for debugging image builds.

## Viewing the Status of an Image

When a user creates an image using the above workflow, they are configuring Tanzu Build Service to start creating builds of the image which create container images to be pushed to a registry.

If a particular build associated with an image fails, check the status of the image by running:

```
kp image status <image-name> --namespace <namespace>
```

Where `image-name` is the name of the image. See [Listing Images](#) to get image names.

The following is an example output of this command:

```
Status:          Not Ready
Message:         --
LatestImage:     gcr.io/myapp@sha256:9d7b1fbf7f5cb0f8efe797f30e598b5e38bb1c08ada143d4c
96e4f78111a9239

Last Successful Build
Id:             1
Reason:        CONFIG

Last Failed Build
Id:             2
Reason:        COMMIT
```

## Deleting an Image

This procedure describes how to delete a Build Service image with the `kp` CLI.

**Warning:** Deleting an image deletes the image resource and all the builds that the image resource owns. It does not delete the app images generated by those builds from the registry.

To delete an image:

```
kp image delete <image> --namespace <namespace>
```

Where `image` is the name of the image.

When you successfully delete an image, you will see this message:

```
"<image>" deleted
```

## Managing Images with YAML

Build Services images can be created by applying the [kpack image resources](#) to cluster via `kubectl`.

Use the `default` service account for Build Service registry and git secrets.

### Using Secrets

Use the `default` service account for Build Service registry and git secrets. `kpack` will default to the `default` service account if no service account is specified.

### Debugging with Image Status

Using `kubectl` is a good way to debug Images.

When an image has successfully built with its current configuration, its status will report the up to date fully qualified built image reference.

This information is available with `kubectl get image <image-name> -o yaml`.

```
status:
  conditions:
  - lastTransitionTime: "2020-01-17T16:16:36Z"
    status: "True"
    type: Succeeded
  - lastTransitionTime: "2020-01-17T16:16:36Z"
    status: "True"
    type: BuilderReady
  latestImage: index.docker.io/sample/image@sha256:d3eb15a6fd25cb79039594294419de2328f
  14b443fa0546fa9e16f5214d61686
  ...
```

When a build fails the image status will report the condition `Succeeded=False`. The Image status also includes the status of the builder being used by the image. If the builder is not ready, you may want to inspect that builder. More details in [Managing Builders](#).

```
status:
  conditions:
  - lastTransitionTime: "2020-01-17T16:13:48Z"
    status: "False"
    type: Succeeded
```

```

message: "Some error occurred"
- lastTransitionTime: "2020-01-17T16:16:36Z"
  status: "False"
  type: BuilderReady
  message: "Some builder error occurred"
...

```

If further debugging is required, inspect the image's latest Build status discussed in [Viewing Build Details for an Image](#).

## Image Service Bindings

Tanzu Build Service supports application service bindings as described in the [Cloud Native Buildpack Service Bindings specification](#).

The `kp` CLI does not currently support creating service bindings, you should use `kubectl`.

## Creating an Image with Service Bindings

To create a service binding in your application image, you must create the following:

- A Kubernetes Secret containing the service binding data
  - The Secret `stringData` field must contain key-value pairs of `<binding file name>: <binding data>`. For each key-value pair, a file will be created that is accessible during build.
- A Kubernetes ConfigMap containing the metadata for the service binding
  - The ConfigMap must have the fields `data.kind` and `data.provider` populated. The buildpacks used to build the image will handle the service bindings based on these fields.
- A Tanzu Build Service Image referencing that Secret and ConfigMap in the `spec.build.bindings` field.

**Note:** Check the desired buildpack documentation for details on the service bindings it supports. To find buildpack docs, see [\[Store Status\]\(managing-stores.html#show-buildpackages-in-store\)](#).

The following is an example that can be used with `kubectl apply`. It creates a `settings.xml` service binding for a maven app.

Example:

```

apiVersion: kpack.io/v1alpha1
kind: Image
metadata:
  name: sample-binding-with-secret
spec:
  tag: my-registry.com/repo
  builder:
    kind: ClusterBuilder
    name: default
  source:
    git:
      url: https://github.com/buildpack/sample-java-app.git
      revision: 0eccc6c2f01d9f055087ebbf03526ed0623e014a
  build:

```

```

bindings:
  - name: settings
    secretRef:
      name: settings-xml
    metadataRef:
      name: settings-binding-metadata
---
apiVersion: v1
kind: Secret
metadata:
  name: settings-xml
type: Opaque
stringData:
  settings.xml: <settings>...</settings>
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: settings-binding-metadata
data:
  kind: maven
  provider: sample

```

## Builds

The procedures in this section describe how to view information and logs for image builds using the `kp` CLI.

### Listing Builds

Build Service stores the ten most recent successful builds and the ten most recent failed builds.

To see a the list of builds for an image run:

```
kp build list <image-name> --namespace <namespace>
```

If the `namespace` is not specified, it defaults to the kubernetes current-context namespace. And if the `image-name` is not specified, the builds for all the images in your namespace are listed.

The following is an example of the output for this command:

BUILD	STATUS	IMAGE	REASON
1	SUCCESS	gcr.io/myapp@sha256:some-sha1	CONFIG
2	SUCCESS	gcr.io/myapp@sha256:some-sha2	COMMIT
3	SUCCESS	gcr.io/myapp@sha256:some-sha3	STACK
4	FAILURE	gcr.io/myapp@sha256:some-sha4	CONFIG+
5	BUILDING	gcr.io/myapp@sha256:some-sha5	BUILDPACK

The following describes the fields in the example output:

- **BUILD**: Describes the index of builds in the order that they were built.
- **STATUS**: Describes the status of a previous build image.
- **IMAGE**: The full image reference for the app image produced by the build.
- **REASON**: Describes why an image rebuild occurred. These reasons include:
  - ◊ **CONFIG**: Occurs when a change is made to commit, branch, Git repository, or build fields on the image's configuration file and you run `kp image apply`.
  - ◊ **COMMIT**: Occurs when new source code is committed to a branch or tag that Build Service is monitoring for changes.
  - ◊ **BUILDPACK**: Occurs when new buildpack versions are made available through an updated builder.
  - ◊ **STACK**: Occurs when a new base OS image, called a `run image`, is available.
  - ◊ **TRIGGER**: Occurs when a new build is manually triggered.

**Note:** A rebuild can occur for more than one reason. When there are multiple reasons for a rebuild, the `kp` CLI output shows the primary `Reason` and appends a `+` sign to the `Reason` field. The priority rank for the `Reason`, from highest to lowest, is `CONFIG`, `COMMIT`, `BUILDPACK`, `STACK`, and `TRIGGER`.

## Viewing Build Details for an Image

To display retrieve a detailed Bill of Materials for a particular build:

```
kp build status <image> -b <build-number>
```

Where:

- `image-name` is the name of the image the build is associated with
- `build-name` (optional) is the index of the build from [listing builds](#). Defaults to latest build.

The following is an example of the output for this command:

```
Image:      gcr.io/myapp@sha256:f87b614257af05c3301c1554c4f15131793caec3adf55e45d2c612
e90445765a
Status:     SUCCESS
Reason:     CONFIG
            resources
            - source: {}
            + source:
            +   git:
            +     revision: 948b2eff6a21580a44a0f4d8c609a2af45359d41
            +     url: https://github.com/paketo-buildpacks/samples
            +     subPath: go/mod

Started:    2021-02-02 18:34:33
Finished:   2021-02-02 18:41:03

Pod Name:   build-pod-xyz

Builder:    gcr.io/my-builder:base@sha256:grteiwads0asdvf09asdf
Run Image:  gcr.io/base-image:run@sha256:asdas098asdas
```

```

Source:      Git
Url:         http://github.com/myapp
Revision:    ad123ad

BUILDPACK ID      BUILDPACK VERSION
io.java.etc       123
io.kotlin.etc     321

```

The following describes the fields in the example output:

- **Image:** The full image reference for the app image produced by the build.
- **Status:** Describes the status of a previous build image.
- **Reason:** Describes why an image build occurred and the change diff. The reason could be one or more of these:
  - ◆ **CONFIG:** Occurs when a change is made to commit, branch, Git repository, or build fields on the image's configuration file and you run `kp image apply`.
  - ◆ **COMMIT:** Occurs when new source code is committed to a branch or tag that Build Service is monitoring for changes.
  - ◆ **BUILDPACK:** Occurs when new buildpack versions are made available through an updated builder.
  - ◆ **STACK:** Occurs when a new base OS image (called a `run image`) is available.
  - ◆ **TRIGGER:** Occurs when a new build is manually triggered.
- **Started:** When a build started.
- **Finished:** When a build finished.
- **Pod Name:** The name of the Pod being used for the Build.
- **Builder:** The full image tag for the builder image used by the build.
- **Run Image:** The full image tag for the run image used by the app.
- **Source:** Describes where the source code used to build the image is coming from. Can be `Git`, `Blob`, or `Local Source`.
- **Url:** The Git repository URL for `Git` source, the Blob file URL for `Blob` source. Unset for `Local Source`.
- **Revision:** The Git commit sha of the source code used to create the build for `Git` source.
- **BUILDPACK ID:** A list of buildpack ids the build used.
- **BUILDPACK VERSION:** A list of buildpack versions the build used.

### Image Status shows ImagePullBackOff

If the Build is currently waiting for a container, the Build status will show details in the output of `kp build status`.

Here is an example output:

---

```

Image:      --
Status:     BUILDING
Reason:     CONFIG
Status Reason:  ImagePullBackOff
Status Message:  A container image currently cannot be pulled: Back-off pulling image
                "gcr.io/my-builder:base@sha256:grtewwads0asdvf09asdf"

Pod Name:   build-pod-xyz

Builder:    gcr.io/my-builder:base@sha256:grtewwads0asdvf09asdf
Run Image:  gcr.io/base-image:run@sha256:asd098asdas

Source:     Git
Url:        http://github.com/myapp
Revision:   ad123ad

BUILDPACK ID      BUILDPACK VERSION

```

If you are seeing this error and you are using a Cluster Builder, you may need to configure a Synced Secret. See [When to use Synchronized Secrets](#).

## Getting Build Logs

An image that a user creates will cause builds to be initiated for that image. Builds are where Cloud Native Buildpacks are run and apps get built into images.

Build logs are a good way to debug issues and to get information about how your app is being built.

If you get logs of a build in progress, the logs will be tailed and will terminate when the build completes.

To get logs from a build run:

```
kp build logs <image> --build <build-number> --namespace <namespace>
```

Where:

- `image-name` is the name of the image the build is associated with
- `build-name` (optional) is the index of the build from [listing builds](#). Defaults to latest build.

The following is an example of the output of the command:

```

===> PREPARE
Build reason(s): CONFIG
CONFIG:
  resources: {}
  - source: {}
  + source:
    + git:
    +   revision: 446dbda043ca103d33e2cad389d43f289e63f647
    +   url: https://github.com/some-org/some-repo
Loading secret for "gcr.io" from secret "gcr" at location "/var/build-secrets/gcr"
Cloning "https://github.com/some-org/some-repo" @ "446dbda043ca103d33e2cad389d43f289e63f647"...
Successfully cloned "https://github.com/some-org/some-repo" @ "446dbda043ca103d33e2cad389d43f289e63f647" in path "/workspace"
===> DETECT

```

```

tanzu-buildpacks/node-engine 0.1.2
tanzu-buildpacks/npm-install 0.1.1
tanzu-buildpacks/npm-start 0.0.2
===> ANALYZE
Previous image with name "gcr.io/test-app" not found
===> RESTORE
===> BUILD
Tanzu Node Engine Buildpack 0.1.2
  Resolving Node Engine version
    Candidate version sources (in priority order):
      -> ""
      <unknown> -> "*"

    Selected Node Engine version (using ): 14.15.1

  Executing build process
    Installing Node Engine 14.15.1
      Completed in 2.495s

  Configuring environment
    NODE_ENV      -> "production"
    NODE_HOME     -> "/layers/tanzu-buildpacks_node-engine/node"
    NODE_VERBOSE  -> "false"

  Writing profile.d/0_memory_available.sh
    Calculates available memory based on container limits at launch time.
    Made available in the MEMORY_AVAILABLE environment variable.

Tanzu NPM Install Buildpack 0.1.1
  Resolving installation process
    Process inputs:
      node_modules      -> "Not found"
      npm-cache         -> "Not found"
      package-lock.json -> "Not found"

    Selected NPM build process: 'npm install'

  Executing build process
    Running 'npm install --unsafe-perm --cache /layers/tanzu-buildpacks_npm-install/npm-cache'
      Completed in 3.591s

  Configuring environment
    NPM_CONFIG_LOGLEVEL -> "error"
    NPM_CONFIG_PRODUCTION -> "true"
    PATH                -> "$PATH:/layers/tanzu-buildpacks_npm-install/modules/node_modules/.bin"

Tanzu NPM Start Buildpack 0.0.2
  Assigning launch processes
    web: node server.js
===> EXPORT
Adding layer 'tanzu-buildpacks/node-engine:node'
Adding layer 'tanzu-buildpacks/npm-install:modules'
Adding layer 'tanzu-buildpacks/npm-install:npm-cache'
Adding 1/1 app layer(s)
Adding layer 'launcher'
Adding layer 'config'
Adding label 'io.buildpacks.lifecycle.metadata'

```

```

Adding label 'io.buildpacks.build.metadata'
Adding label 'io.buildpacks.project.metadata'
*** Images (sha256:0abdbaf1f25c3c13cdb918d06906670b84dd531bc7301177b11284dac68bdb9c) :
    gcr.io/test-app
    gcr.io/test-app:b1.20210203.225422
Adding cache layer 'tanzu-buildpacks/node-engine:node'
Adding cache layer 'tanzu-buildpacks/npm-install:modules'
Adding cache layer 'tanzu-buildpacks/npm-install:npm-cache'
===> COMPLETION
Build successful

```

## Viewing Bill of Materials

The `kp cli` allows you to view the bill of materials in an image built by a Build.

```
kp build status <image-name> --bom
```

For generating the bill of materials, the `kp CLI` will read metadata from the image (generated by the build) in the registry.

**Note:** You must have credentials to access the image registry on your machine.

As an example:

```

$ kp build status --bom my-app-image | jq
[
  {
    "buildpack": {
      "id": "tanzu-buildpacks/node-engine",
      "version": "0.1.2"
    },
    "metadata": {
      "licenses": [],
      "name": "Node Engine",
      "sha256": "b981046a0ea3d5594a7f04fae3afdfa1983bc65f4e26e768b38a2d67057ac75c",
      "stacks": [
        "io.buildpacks.stacks.bionic",
        "org.cloudfoundry.stacks.cflinuxfs3"
      ],
      "uri": "file:///dependencies/b981046a0ea3d5594a7f04fae3afdfa1983bc65f4e26e768b38a2d67057ac75c",
      "version": "14.15.1"
    },
    "name": "node",
    "version": "14.15.1"
  },
  {
    "buildpack": {
      "id": "tanzu-buildpacks/npm-install",
      "version": "0.1.1"
    },
    "metadata": {
      "launch": true
    },
    "name": "node_modules"
  }
]

```

## Offline Builds

Tanzu Build Service supports offline/air-gapped builds with Tanzu Buildpacks. Offline builds use pre-packaged dependencies and do not need to download from anywhere off-cluster to create application images.

When using Tanzu Buildpacks the build will execute as an offline build. For details on how to configure buildpacks, see [Buildpack Configuration in Images](#).

**Note:** Offline builds only ensure buildpack dependencies are offline. The application build and custom configuration must also not reach off-cluster to be completely offline.

## Image Signing with Notary

Tanzu Build Service supports [Notary](#) image signing.

Images signed with Notary require using `kubect1` instead of `kp`.

Prerequisites:

- [notary cli](#)
- A notary server accessible by the Tanzu Build Service cluster

## Generate Notary Signing Keys

Only one root signing key is required. Each image that will be signed by notary will require a target and snapshot signing key.

Run the following commands. You will be asked to provide the registry credentials for the `<image-repository>`:

```
export NOTARY_ROOT_PASSPHRASE=<notary-root-passphrase>
export NOTARY_SNAPSHOT_PASSPHRASE=<notary-snapshot-passphrase>
notary -s <notary-server-url> init <image-repository>
notary -s <notary-server-url> key rotate <image-repository> snapshot -r
notary -s <notary-server-url> publish <image-repository>
```

You will be prompted to enter a `<notary-targets-passphrase>`:

```
Enter passphrase for targets key with ID <target-hash>:
```

Where:

- `<notary-root-passphrase>` is a secure passphrase (this is the root passphrase and should be secure and stored. This is used to generate and rotate other keys.)
- `<notary-snapshot-passphrase>` is a secure passphrase (one per image)
- `<notary-targets-passphrase>` is a secure passphrase for the targets (one per image)
- `<notary-server-url>` is the notary server url
- `<image-repository>` is the repository for the image that will be built by Tanzu Build Service.
- `<target-hash>` is the hash for the target signing key.

**Note:** All passphrases can be entered manually via prompts. Some are set as env vars in the example command for simplicity.

## Create a Secret to be used for TBS Image Signing

Run the following:

```
kubectl create secret generic <notary-secret-name> \
  --from-literal=password=<notary-targets-passphrase> \
  --from-file=~/.notary/private/<target-hash>.key \
  --namespace <image-namespace>
```

Where:

- `<notary-secret-name>` is the arbitrary name of the notary secret
- `<notary-targets-passphrase>` is the passphrase used in the previous step
- `<target-hash>` is the hash from the previous step
- `<image-namespace>` is the namespace where the TBS image will be created

## Create an Image that will be Signed by Notary

TBS will sign images when the `spec.notary` key is populated.

This configuration cannot be set by `kp cli`, `kubectl` must be used to create the Image.

Example image.yaml:

```
apiVersion: kpack.io/v1alpha1
kind: Image
metadata:
  name: my-notary-image
  namespace: <image-namespace>
spec:
  notary:
    v1:
      url: <notary-server-url>
      secretRef:
        name: <notary-secret-name>
  serviceAccount: default
  source:
    git:
      url: github.com/my-git-repo
  tag: <image-repository>
  status: {}
```

Where:

- `<image-namespace>` is the namespace used for the secret created in the above step
- `<notary-server-url>` is the notary server url used in the first step
- `<notary-secret-name>` is the secret name created in the previous step
- `<image-repository>` is the image repository used in the first step

# Managing ClusterStacks

A ClusterStack is a cluster scoped resource that provides the build and run images for the [Cloud Native Buildpack stack](#) that will be used in a [Builder](#).

Most users automatically configure three ClusterStack resources via the TBS installation process. These ClusterStacks are referenced in three corresponding ClusterBuilder resources.

Additional information about security and patching cadence for these stacks and their ideal use cases can be found [here](#). More detailed release notes for the stacks can be accessed by following the links in the table below.

Name	ID
<a href="#">tiny</a>	io.paketo.stacks.tiny
<a href="#">base</a>	io.buildpacks.stacks.bionic
<a href="#">full</a>	io.buildpacks.stacks.bionic

The `kp` CLI can be used to manage clusterstack. The help text is published [here](#).

```

$ kp clusterstack
Cluster Stack Commands

Usage:
  kp clusterstack [command]

Aliases:
  clusterstack, csk

Available Commands:
  create      Create a cluster stack
  delete     Delete a cluster stack
  list       List cluster stacks
  save       Create or update a cluster stack
  status     Display cluster stack status
  update     Update a cluster stack

Flags:
  -h, --help  help for clusterstack

Use "kp clusterstack [command] --help" for more information about a command.
```

**Note:** These docs assume `kp cli v0.2.*` from TBS release `v1.1.*`. If a feature is not working, you may need to upgrade your cli.

**Note:** Only Build Service Admins (i.e. users with the `pb-admin-role` kubernetes ClusterRole) can perform clusterstack commands.

## Create a ClusterStack

Users can create a clusterstack using build and run images from a Docker registry or the local machine. The run and build images provided during clusterstack creation will be uploaded to the `canonical repository`, which is the `docker-repository` specified during TBS install.

- If using a Docker registry for the stack images:

```
kp clusterstack create <clusterstack-name> \
  --build-image <location of build-image> \
  --run-image <location of run-image>
```

**Note:** The user must have read access to the source Docker registry and write access to the canonical registry on the local machine.

Example:

```
kp csk create my-clusterstack \
  -b gcr.io/test/stack/run:latest
  -r gcr.io/test/stack/build:latest
```

- If using local stack images created with `docker save`:

```
kp clusterstack create <clusterstack-name> \
  --build-image <path to build-image>.tar \
  --run-image <path to run-image>.tar
```

**Note:** The user must have write access to the canonical registry on the local machine.

Example:

```
kp csk create my-clusterstack \
  -b ./local-build-image.tar \
  -r ./local-run-image.tar
```

## Update a ClusterStack

Users can update a stack using build and run images from a Docker registry or the local machine. The run and build images provided during clusterstack update will be uploaded to the `canonical repository`, which is the `docker-repository` specified during TBS install.

- If using a Docker registry:

```
kp clusterstack update <stack-name> \
  --build-image <location of build-image> \
  --run-image <location of run-image>
```

**Note:** The user must have read access to the source Docker registry and write access to the canonical registry on the local machine.

Example:

```
kp csk update my-clusterstack \
  -b gcr.io/test/stack/run:latest
```

```
-r gcr.io/test/stack/build:latest
```

- If using local stack images created with `docker save`:

```
kp clusterstack update <stack-name> \
  --build-image <path to build-image>.tar \
  --run-image <path to run-image>.tar
```

**Note:** The user must have write access to the canonical registry on the local machine.

Example:

```
kp csk update my-clusterstack \
  -b ./local-build-image.tar \
  -r ./local-run-image.tar
```

## Save a ClusterStack

Users can create or update a ClusterStack using the `save` command. The `kp clusterstack save` command is used exactly the same as `kp clusterstack create` and `kp clusterstack update`, but it will determine if a clusterstack needs to be created or updated.

## Get ClusterStack Status

Users can get the current status of a clusterstack:

```
kp clusterstack status <stack-name>
```

The following is an example of the output for this command:

```
Status:      Ready
ID:          org.cloudfoundry.stacks.cflinuxfs3
Run Image:   paketo/run:full-cnb
Build Image: paketo/build:full-cnb
```

## Delete a ClusterStack

Users can delete an existing clusterstack:

```
kp clusterstack delete <stack-name>
```

**Note:** User will not be asked for a confirmation before deletion.

## List all ClusterStacks

Users can view the list of all ClusterStacks created:

The following is an example of the output for this command:

NAME	READY	ID
base	True	io.buildpacks.stacks.bionic
default	True	io.buildpacks.stacks.bionic

```
full      True      org.cloudfoundry.stacks.cflinuxfs3
tiny     True      io.paketo.stacks.tiny
```

## How to update an Image for Stack updates only?

To achieve Stack only updates for an Image, you can [pin the Buildpack versions](#) in the Builder used for creating the Image.

# Managing Stores

A Store is a cluster level resource that provides a collection of buildpacks that can be utilized by Builders. Buildpacks are distributed and added to a store in buildpackages which are docker images containing one or more buildpacks.

Build Service ships with a curated collection of Tanzu buildpacks for Java, Nodejs, Go, PHP, nginx, and httpd and Paketo buildpacks for profile, and .NET Core. Detailed documentation about the buildpacks that are installed with TBS can be found [here](#). It is important to keep these buildpacks up-to-date. Updates to these buildpacks are provided on [Tanzu Network](#).

In addition to supported Tanzu and Paketo buildpacks, custom buildpackages can be uploaded to Build Service stores.

The `kp` CLI can be used to manage clusterstores. The help text is published [here](#).

```
$ kp clusterstore
ClusterStore Commands

Usage:
  kp clusterstore [command]

Aliases:
  clusterstore, clusterstores, clstrcsrs, clstrcsr, csrs, csr

Available Commands:
  add          Add buildpackage(s) to cluster store
  create       Create a cluster store
  delete       Delete a cluster store
  list         List cluster stores
  remove       Remove buildpackage(s) from cluster store
  save         Create or update a cluster store
  status       Display cluster store status

Flags:
  -h, --help  help for clusterstore
```

**Note:** These docs assume `kp cli v0.2.*` from TBS release `v1.1.*`. If a feature is not working, you may need to upgrade your cli.

## Creating Buildpacks and Buildpackages

Documentation for creating buildpacks is available [here](#).

Documentation for creating buildpackages is available [here](#).

**Note:** Only Build Service Admins can perform store commands.

## Listing ClusterStores

Users can view the existing stores with:

```
kp clusterstore list
```

## Creating a ClusterStore

Tanzu Build Service ships with a `default` store containing all of the supported buildpacks. Users can create additional stores with:

```
kp clusterstore create <store-name> -b <buildpackage-1> -b <buildpackage-2>
```

Examples:

```
kp clusterstore create my-store -b my-registry.com/my-buildpackage
kp clusterstore create my-store -b my-registry.com/my-buildpackage -b my-registry.com/
my-other-buildpackage
kp clusterstore create my-store -b ../path/to/my-local-buildpackage.cnb
```

Buildpackages will be uploaded to the registry used during installation.

**Note:** The user must have read access to the source Docker registry and write access to the registry used for installation on the local machine.

## Saving a ClusterStore

Users can create or update a ClusterStore using the `save` command. The `kp clusterstore save` command is used exactly the same as `kp clusterstore create`, but it will determine if a clusterstore needs to be created or updated.

```
kp clusterstore save <store-name> -b <buildpackage-1> -b <buildpackage-2>
```

## Adding Buildpackages to a ClusterStore

Users can add multiple buildpackages at a time from a registry or from a file on the local machine.

This command is useful for users that want to only consume certain buildpacks rather than update all dependencies with `kp import`.

- If using a Docker registry:

```
kp clusterstore add <store-name> -b <buildpackage-1> -b <buildpackage-2> ...
```

**Note:** The user must have read access to the source Docker registry and write access to the registry used for installation on the local machine.

- If using local `.cnb` buildpackage files created as described in the [buildpackages docs](#):

```
kp clusterstore add <store-name> -b <path-to-buildpackage-1>.cnb -b <path-to-buildpackage-2>.cnb ...
```

## Adding Buildpackages to a ClusterStore from Tanzu Network

Updated versions of all supported Buildpacks will be available on [Tanzu Network](#) as registry images. Updated Buildpacks will be found in the following locations:

- [Java](#)
- [NodeJS](#)
- [Go](#)
- [PHP, .NET Core, nginx, httpd, procfile](#)

Here is a list of how to update each buildpack that is included with Tanzu Build Service by default:

```
kp clusterstore add default registry.pivotal.io/tanzu-java-buildpack/java:<version>
kp clusterstore add default registry.pivotal.io/tanzu-nodejs-buildpack/nodejs:<version>
kp clusterstore add default registry.pivotal.io/tanzu-go-buildpack/go:<version>
kp clusterstore add default registry.pivotal.io/tbs-dependencies/paketo-buildpacks_dotnet-core:<version>
kp clusterstore add default registry.pivotal.io/tbs-dependencies/tanzu-buildpacks_php:<version>
kp clusterstore add default registry.pivotal.io/tbs-dependencies/tanzu-buildpacks_nginx:<version>
kp clusterstore add default registry.pivotal.io/tbs-dependencies/tanzu-buildpacks_httpd:<version>
kp clusterstore add default registry.pivotal.io/tbs-dependencies/paketo-buildpacks_procfile:<version>
```

## Offline Adding Buildpackages to a ClusterStore from Tanzu Network

If your Tanzu Build Service installation is in an offline/air-gapped environment, you can update stores with the following offline workflow:

1. Download the Dependency Descriptor file (`descriptor-<version>.yaml`) from the latest release on the [Tanzu Build Service Dependencies](#) page on Tanzu Network.
2. Download the `kp` CLI for your operating system from the latest release on the [Tanzu Build Service](#) page.
3. Download the `kbld` CLI for your operating system from the latest release on the [kbld](#) page.
4. Download the dependency images for Tanzu Build Service to your local machine with `kbld`:

```
docker login registry.pivotal.io

kbld package -f descriptor-<version>.yaml \
  --output /tmp/packaged-dependencies.tar
```

5. Move the output file `packaged-dependencies.tar` to a machine that has access to the "offline" environment
6. Upload the dependency images to the registry used to deploy Tanzu Build Service:

```
docker login <build-service-registry>
```

```

kblld unpackage -f descriptor-<version>.yaml \
  --input /tmp/packaged-dependencies.tar \
  --repository <IMAGE-REPOSITORY> \
  --lock-output /tmp/dependencies-relocated.lock

```

Where `IMAGE-REPOSITORY` is the repository used to install Tanzu Build Service. This should be the same value as `IMAGE-REPOSITORY` used in the [Installation Steps](#).

- Now that dependencies are relocated to the internal registry, you can use the following command to update the necessary resources:

```

kblld -f descriptor-<version>.yaml -f /tmp/dependencies-relocated.lock | kp import -f -

```

## Removing Buildpackages from a ClusterStore

Users can remove a buildpackage from a ClusterStore by referencing the buildpackage Id and version.

```

kp clusterstore remove <store> -b <buildpackage-id>@<buildpackage-version>

```

Examples:

```

kp clusterstore remove my-store -b buildpackage@1.0.0
kp clusterstore remove my-store -b buildpackage@1.0.0 -b other-buildpackage@2.0.0

```

The ClusterStore status shows the list of buildpackage Id and version

## Get ClusterStore Status

Users can use the `kp` CLI to get details about a store including buildpackages and their buildpacks, as well as meta-buildpacks. [Meta-buildpacks](#) are buildpacks that indicate the order that other buildpacks run:

To view the buildpackages in a store:

```

kp clusterstore status <store-name>

```

Example:

```

$kp clusterstore status default

Status:    Ready

BUILDPACKAGE ID          VERSION    HOMEPAGE
paketo-buildpacks/go     0.1.3     https://github.com/paketo-buildpacks/
go
paketo-buildpacks/procfile 2.0.2     https://github.com/paketo-buildpacks/
procfile
paketo-buildpacks/procfile 3.0.0     https://github.com/paketo-buildpacks/
procfile
tanzu-buildpacks/dotnet-core 0.0.4
tanzu-buildpacks/dotnet-core 0.0.7
tanzu-buildpacks/dotnet-core 0.0.6
tanzu-buildpacks/go      1.0.6

```

tanzu-buildpacks/go	1.0.7	
tanzu-buildpacks/go	1.0.9	
tanzu-buildpacks/go	1.0.5	
tanzu-buildpacks/httpd	0.0.38	
tanzu-buildpacks/httpd	0.0.39	
tanzu-buildpacks/httpd	0.0.40	
tanzu-buildpacks/java ava	3.8.0	<a href="https://github.com/pivotal-cf/tanzu-java">https://github.com/pivotal-cf/tanzu-java</a>
tanzu-buildpacks/java ava	3.5.0	<a href="https://github.com/pivotal-cf/tanzu-java">https://github.com/pivotal-cf/tanzu-java</a>
tanzu-buildpacks/java ava	4.1.0	<a href="https://github.com/pivotal-cf/tanzu-java">https://github.com/pivotal-cf/tanzu-java</a>
tanzu-buildpacks/java ava	4.0.0	<a href="https://github.com/pivotal-cf/tanzu-java">https://github.com/pivotal-cf/tanzu-java</a>
tanzu-buildpacks/java-native-image ava-native-image	3.6.0	<a href="https://github.com/pivotal-cf/tanzu-java-native-image">https://github.com/pivotal-cf/tanzu-java-native-image</a>
tanzu-buildpacks/java-native-image ava-native-image	3.9.0	<a href="https://github.com/pivotal-cf/tanzu-java-native-image">https://github.com/pivotal-cf/tanzu-java-native-image</a>
tanzu-buildpacks/java-native-image ava-native-image	3.4.2	<a href="https://github.com/pivotal-cf/tanzu-java-native-image">https://github.com/pivotal-cf/tanzu-java-native-image</a>
tanzu-buildpacks/java-native-image ava-native-image	3.10.0	<a href="https://github.com/pivotal-cf/tanzu-java-native-image">https://github.com/pivotal-cf/tanzu-java-native-image</a>
tanzu-buildpacks/nginx	0.0.48	
tanzu-buildpacks/nginx	0.0.46	
tanzu-buildpacks/nodejs	1.1.0	
tanzu-buildpacks/nodejs	1.2.3	
tanzu-buildpacks/nodejs	1.2.2	
tanzu-buildpacks/php	0.0.3	
tanzu-buildpacks/php	0.0.5	

To view buildpackages & their individual buildpacks as well as display the order of meta-buildpacks use the `--verbose` flag

```
kp clusterstore status <store-name> --verbose
```

## Migrating Buildpacks

Build Service will never automatically remove buildpackages from the store unless you explicitly remove them. In this way, users can continue to use older buildpacks until the operator is ready to migrate them.

How you migrate is entirely dependent on the configuration of your Builder resources: \* Builders that do not provide a buildpack version will automatically update to the latest buildpack version if it is available. \* Builders that explicitly specify a buildpack version will not update automatically.

With the above in mind, migrating buildpackages in the store is as simple as `kp clusterstore adding` newer buildpackages and `kp clusterstore removeing` older buildpackages as necessary.

If you'd like fine-grained control over buildpack updates, you can create multiple stores to manage buildpack versions. Then, you can point individual builders at the desired store. Each store can be updated as needed without affecting other builders or fanning out large, sweeping changes.

## Corresponding kpack Resource

All Build Service builders utilize cluster scoped [Store Resources](#).

# Managing Builders

A Builder is a Tanzu Build Service resource used to manage [Cloud Native Buildpack builders](#).

Builders contain a set of buildpacks and a stack that will be used to create images.

There are two types of Builders:

- Cluster Builders: Cluster-scoped Builders
- Builders: Namespace-scoped Builders

**Note:** Only Build Service Admins can manage Cluster Builders.

The `kp` CLI can be used to manage builders and clusterbuilders. The help text is published [here](#).

```
$ kp builder
Builder Commands

Usage:
  kp builder [command]

Aliases:
  builder, builders, bldrs, bldr

Available Commands:
  create      Create a builder
  delete      Delete a builder
  list        List available builders
  patch       Patch an existing builder configuration
  save        Create or patch a builder
  status      Display status of a builder

Flags:
  -h, --help  help for builder

Use "kp builder [command] --help" for more information about a command.
```

```
$ kp clusterbuilder
ClusterBuilder Commands

Usage:
  kp clusterbuilder [command]

Aliases:
  clusterbuilder, clusterbuilders, clstrbldrs, clstrbldr, cbldrs, cbldr, cbs, cb

Available Commands:
  create      Create a cluster builder
  delete      Delete a cluster builder
  list        List available cluster builders
```

```

patch      Patch an existing cluster builder configuration
save       Create or patch a cluster builder
status     Display cluster builder status

```

## Flags:

```
-h, --help  help for clusterbuilder
```

**Note:** These docs assume kp cli v0.2.\* from TBS release v1.1.\*. If a feature is not working, you may need to upgrade your cli.

## Creating a Builder

Use the `kp` cli to create a Builder:

- Cluster Builder:

```
kp clusterbuilder create <name> --tag <tag> --order <order> --stack <stack> --store <store>
```

```
kp clusterbuilder create <name> --tag <tag> --stack <stack> --store <store> --buildpack <buildpack>
```

- Builder:

```
kp builder create <name> --tag <tag> --order <order> --stack <stack> --store <store> --namespace <namespace>
```

```
kp builder create <name> --tag <tag> --stack <stack> --store <store> --namespace <namespace> --buildpack <buildpack>
```

Where:

- `name`: The name of the builder.
- `tag`: The registry location where the builder will be created.
- `stack`: The name of the stack to be used by the builder.
- `store`: The name of the store containing the buildpacks that will be used by the builder.
- `namespace`: The kubernetes namespace for the builder (for Builders only)
- `order`: The local path to the buildpack order YAML that the builder will use. Sample order YAML files will be available on the [VMware Tanzu Build Service Dependencies](#) page on Tanzu Network. For more information about listing buildpacks in groups in the order YAML, see [builder.toml](#) in the Buildpacks.io documentation.

Example order YAML file that would be used by a builder designed to build NodeJS and Java apps:

```

- group:
  - id: tanzu-buildpacks/nodejs
- group:
  - id: tanzu-buildpacks/java

```

- **buildpack**: Buildpack id and optional version in the form of either '@' or ''. Repeat for each buildpack in order, or supply once with comma-separated list. This cannot be combined with `--order`. All supplied buildpacks will be in the same group.

## Patching a Builder

You can update a Builder resource using the `kp` cli. To update a builder given a `name`, run:

- Cluster Builder:

```
kp clusterbuilder patch <name> --order <order> --stack <stack> --store <store>
```

```
kp clusterbuilder patch <name> --stack <stack> --store <store> --buildpack <buildpack>
```

- Builder:

```
kp builder patch <name> --order <order> --stack <stack> --store <store> --namespace <namespace>
```

```
kp builder patch <name> --stack <stack> --store <store> --namespace <namespace> --buildpack <buildpack>
```

`kp ccb patch` and `kp cb patch` are respective aliases.

Where:

- **name**: The name of the builder.
- **stack**: The name of the stack to be used by the builder.
- **store**: The name of the store containing the buildpacks that will be used by the builder.
- **namespace**: The kubernetes namespace for the builder (for Builders only)
- **order**: The local path to the buildpack order YAML that the builder will use. Sample order YAML files will be available on the [VMware Tanzu Build Service Dependencies](#) page on Tanzu Network. For more information about listing buildpacks in groups in the order YAML, see `builder.toml` in the Buildpacks.io documentation.

Example order YAML file that would be used by a builder designed to build NodeJS and Java apps:

```
---
- group:
  - id: paketo-buildpacks/bellsoft-liberica
  - id: paketo-buildpacks/gradle
- group:
  - id: paketo-buildpacks/nodejs
```

- **buildpack**: Buildpack id and optional version in the form of either '@' or ''. Repeat for each buildpack in order, or supply once with comma-separated list. This cannot be combined with `--order`. All supplied buildpacks will be in the same group.

**Note:** The `tag` (location in a registry) of a builder cannot be modified. To change this field, you must

create a new builder.

## Saving Builders

Users can create or update a Builder/ClusterBuilder using the `save` command. The `kp builder/clusterbuilder save` command is used exactly the same as `kp builder/clusterbuilder create` and `kp builder/clusterbuilder update`, but it will determine if a builder/clusterbuilder needs to be created or updated.

To save a Builder/ClusterBuilder:

- Cluster Builder:

```
kp clusterbuilder save <name> --tag <tag> --order <order> --stack <stack> --store <store>
```

```
kp clusterbuilder save <name> --tag <tag> --stack <stack> --store <store> --buildpack <buildpack>
```

- Builder:

```
kp builder save <name> --tag <tag> --order <order> --stack <stack> --store <store> --namespace <namespace>
```

```
kp builder save <name> --tag <tag> --stack <stack> --store <store> --namespace <namespace> --buildpack <buildpack>
```

Where:

- `name`: The name of the builder.
- `tag`: The registry location where the builder will be created.
- `stack`: The name of the stack to be used by the builder.
- `store`: The name of the store containing the buildpacks that will be used by the builder.
- `namespace`: The kubernetes namespace for the builder (for Builders only)
- `order`: The local path to the buildpack order YAML that the builder will use. Sample order YAML files will be available on the [VMware Tanzu Build Service Dependencies](#) page on Tanzu Network. For more information about listing buildpacks in groups in the order YAML, see `builder.toml` in the Buildpacks.io documentation.

Example order YAML file that would be used by a builder designed to build NodeJS and Java apps:

```
---
- group:
  - id: paketo-buildpacks/bellsoft-liberica
  - id: paketo-buildpacks/gradle
- group:
  - id: paketo-buildpacks/nodejs
```

- `buildpack`: Buildpack id and optional version in the form of either '@' or '. Repeat for each

buildpack in order, or supply once with comma-separated list. This cannot be combined with `--order`. All supplied buildpacks will be in the same group.

## Deleting Builders

To delete a Builder:

- Cluster Builder:

```
kp clusterbuilder delete <builder name>
```

- Builder:

```
kp builder delete <builder name> --namespace <namespace>
```

**Warning:** Deleting a builder will prevent image configs that reference that builder from successfully building again.

## Retrieving Builder Details

To get builder details:

- Cluster Builder:

```
kp clusterbuilder status <builder-name>
```

- Builder:

```
kp builder status <builder-name> --namespace <namespace>
```

Example:

```
$ kp clusterbuilder status tiny

Status:      Ready
Image:       gcr.io/my-repo/tiny@sha256:07d94db2e3e9f43cba67c389f1c83e4eac821aa83084a88136ed8d431b37f008
Stack:       io.paketo.stacks.tiny
Run Image:   gcr.io/cf-build-service-dev-219913/ssuresh/install/run@sha256:e9159f0ef23c28b943cfb1b5d5be9638b67211f6ff0bd3fae35ff4b499136152

BUILDPACK ID          VERSION    HOMEPAGE
paketo-buildpacks/graalvm 4.0.0     https://github.com/paketo-buildpacks/graalvm
tanzu-buildpacks/go-dist 0.1.3
paketo-buildpacks/gradle 3.5.0     https://github.com/paketo-buildpacks/gradle
paketo-buildpacks/sbt     3.6.0     https://github.com/paketo-buildpacks/sbt
paketo-buildpacks/maven  3.2.1     https://github.com/paketo-buildpacks/maven
tanzu-buildpacks/dep      0.0.10
paketo-buildpacks/spring-boot 3.5.0     https://github.com/paketo-buildpacks/spring-boot
paketo-buildpacks/leiningen 1.2.1     https://github.com/paketo-buildpacks/leiningen
```

```

ldpacks/leiningen
paketo-buildpacks/spring-boot-native-image      2.0.0      https://github.com/paketo-bui
ldpacks/spring-boot-native-image
paketo-buildpacks/executable-jar                3.1.3      https://github.com/paketo-bui
ldpacks/executable-jar
tanzu-buildpacks/go-build                       0.0.23
paketo-buildpacks/environment-variables         2.1.2      https://github.com/paketo-bui
ldpacks/environment-variables
paketo-buildpacks/procfile                     3.0.0      https://github.com/paketo-bui
ldpacks/procfile
paketo-buildpacks/image-labels                 2.0.6      https://github.com/paketo-bui
ldpacks/image-labels
tanzu-buildpacks/dep-ensure                    0.0.29
tanzu-buildpacks/go-mod-vendor                 0.0.26
tanzu-buildpacks/java-native-image             3.10.0     https://github.com/pivotal-cf
/tanzu-java-native-image
tanzu-buildpacks/go                            1.0.9

DETECTION ORDER
Group #1
  tanzu-buildpacks/go@1.0.9
Group #2
  tanzu-buildpacks/java-native-image@3.10.0
Group #3
  paketo-buildpacks/procfile@3.0.0

```

## Listing Builders

To list all builders available to the current user:

- Cluster Builder:

```
kp clusterbuilder list
```

- Builder:

```
kp builder list --namespace <namespace>
```

## Corresponding kpack Resources

All Build Service Builders are represented as kpack resources.

- [Builder](#)
- [ClusterBuilder](#)

## Pinning Buildpack versions

You can pin buildpack versions by specifying the version for buildpacks in the order file.

As an **example**, consider the clusterbuilder created below:

```
kp cb create pinned \
  --tag my-registry.io/example/pinned \
```

```
--order order.yaml
```

where the contents of order.yaml file is

```
- group:
  - id: tanzu-buildpacks/php
    version: 0.0.5
- group:
  - id: tanzu-buildpacks/nodejs
    version: 1.3.0
```

**Note:** When a buildpack version is pinned, Images that use the Builder will not initiate new Builds due to new Buildpack versions. For best practice, only pin a buildpack version when necessary.

## Update Lifecycle

All builders make use of a lifecycle. A lifecycle orchestrates buildpack execution, then assembles the resulting artifacts into a final app image. Within Build Service, it will be uploaded to the canonical registry, which is the docker-repository specified during TBS install. More information on lifecycles can be found [here](#).

To update the lifecycle that will be used by builders:

```
...
kp lifecycle update --image <image-tag>
...
```

**Note:** You must have credentials to access the registry on your machine.

# Managing Custom Stacks

A CustomStack is a resource that allows users to create a customized [ClusterStack](#) from Ubuntu 18.04 (Bionic Beaver) based OCI images.

CustomStacks can be used to:

- Convert a pre-existing base image that you'd like to use with TBS into a ClusterStack resource.
- Add required stack metadata to base images.
- Add CA certificates to build and/or run image.
- Add packages and [mixin labels](#) to build and/or run image.
- Set CNB user and group IDs.

## Creating a CustomStack

A CustomStack is created by running `kubectl apply` with a resource configuration file. The following defines the relevant fields of the CustomStack resource spec in more detail:

- `source`: The location of base images used for building the stack. See more info in [Source Configuration](#).
- `destination`: The location to publish built images and optional ClusterStack. See more info in [Destination Configuration](#).
- `caCerts`: References to config maps of CA certificates to add to one or both of the stack images.
- `packages`: List of packages to install on one or both of the stack images. A list of all available packages can be found [here](#).
- `mixins`: List of mixin labels to add to one or both of the stack images. Information on the mixins concept can be found [here](#).
- `service-account-name`: Name of service account with secret containing credentials to push to registry.
- `user`: User and group ID of the CNB user
  - ◊ Not required if the user is already present in metadata.
  - ◊ If the user and/or group ID do not exist on the image, they will be created.

## Source Configuration

The `source` field describes the base images for the CustomStack. It can be configured in exactly one of the following ways:

- Registry Images

```
source:
  registryImages:
    build:
      image: <build-base-image>
    run:
      image: <run-base-image>
```

- ◆ `build-base-image`: The fully qualified reference of the build base image.
- ◆ `run-base-image`: The fully qualified reference of the run base image.

- Stack

```
stack:
  name: <cluster-stack-name>
  apiVersion: kpack.io/v1alpha1
  kind: ClusterStack
```

- ◆ `cluster-stack-name`: Name of ClusterStack to base CustomStack images on.

## Destination Configuration

The `destination` field describes where the built images will be published and if a ClusterStack should be created.

```
destination:
  build:
    tag: <output-build-image-tag>
  run:
    tag: <output-run-image-tag>
  stack: # Optional
    name: <output-cluster-stack-name>
    apiVersion: kpack.io/v1alpha1
    kind: ClusterStack
```

- ◆ `output-build-image-tag`: The registry location where the build image will be created.
- ◆ `output-run-image-tag`: The registry location where the run image will be created.
- ◆ `output-cluster-stack-name`: Name of ClusterStack to create with CustomStack images

## Example CustomStack from Registry Images

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: build-ca-certs
data:
  cert-1: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
  cert-2: |
    -----BEGIN CERTIFICATE-----
```

```

...
-----END CERTIFICATE-----
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: run-ca-certs
data:
  cert-3: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
---
apiVersion: stacks.stacks-operator.tanzu.vmware.com/v1alpha1
kind: CustomStack
metadata:
  name: stack-sample
spec:
  source:
    registryImages:
      build:
        image: paketobuildpacks/build@sha256:ae88191cc5bfd0dcd2938954f20d5df5060a562af
8e3d65a92a815612054537c
        run:
          image: paketobuildpacks/run@sha256:48f67dcb3f2b27403de80193e34abd3172b3fbdfdd8
7e452721aba90ea68fc66
      destination:
        build:
          tag: my.registry.io/final-build-image
        run:
          tag: my.registry.io/final-run-image
    stack: # Optional
      name: stack-sample-cluster-stack
      apiVersion: kpack.io/v1alpha1
      kind: ClusterStack
    caCerts: # Optional
      buildRef: # Optional
        name: build-ca-certs
      runRef: # Optional
        name: run-ca-certs
    packages: # Optional
      - name: cowsay
      - name: cowsay-off
      - name: fortune
        phase: build
      - name: rolldice
        phase: run
    mixins: # Optional
      - name: set=build-utils
        phase: build
      - name: set=run-utils
        phase: run
      - name: set=shared-utils
serviceAccountName: default
user: # Optional
  userID: 1000
  groupID: 1000

```

## Example CustomStack from ClusterStack

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: build-ca-certs
data:
  cert-1: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
  cert-2: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: run-ca-certs
data:
  cert-3: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
---
apiVersion: stacks.stacks-operator.tanzu.vmware.com/v1alpha1
kind: CustomStack
metadata:
  name: stack-sample
spec:
  source:
    stack:
      name: stack-sample-cluster-stack
      apiVersion: kpack.io/v1alpha1
      kind: ClusterStack
  destination:
    build:
      tag: my.registry.io/final-build-image
    run:
      tag: my.registry.io/final-run-image
    stack: # Optional
      name: final-stack-sample-cluster-stack
      apiVersion: kpack.io/v1alpha1
      kind: ClusterStack
  caCerts: # Optional
    buildRef: # Optional
      name: build-ca-certs
    runRef: # Optional
      name: run-ca-certs
  packages: # Optional
    - name: cowsay
    - name: cowsay-off
    - name: fortune
      phase: build # Optional
    - name: rolldice
      phase: run # Optional
  mixins: # Optional
    - name: set=build-utils

```

```

    phase: build # Optional
  - name: set=run-utils
    phase: run # Optional
  - name: set=shared-utils
serviceAccountName: default
user: # Optional
  userID: 1000 # Optional
  groupID: 1000 # Optional

```

## Debugging CustomStacks

When a CustomStack is created, a pod is created in the same namespace which will modify the base image and push the resulting stack image to the registry. The pod will be named `stack-pod-<customstack-name>-<number>`, where:

- `customstack-name`: The name of your CustomStack
- `number`: The revision of your CustomStack. This will be incremented by one each time a new spec is applied.

The ten latest pods are kept around for debugging purposes. To debug a failing CustomStack, check the logs of the corresponding pod: `kubectl logs <pod-name> -c<create-build-image/create-run-image>`, where:

- `pod-name`: The name of the pod
- `create-build-image/create-run-image`: The container whose logs you would like to see.
  - ◊ `create-build-image` for logs related to creating the build image.
  - ◊ `create-run-image` for logs related to create the run image.

# RBAC in Tanzu Build Service

Given that Tanzu Build Service supports functionality most customers would likely want to restrict to only certain users, we encourage utilization of RBAC as a best practice if Tanzu Build Service is to be broadly deployed for usage by many users.

## RBAC using Projects Operator

[Projects Operator](#) can be installed on the cluster to simplify RBAC management.

Projects Operator extends kubernetes with a [Project](#) CRD and corresponding controller. Projects are intended to provide isolation of kubernetes resources on a single kubernetes cluster. A [Project](#) is essentially a kubernetes namespace along with a corresponding set of RBAC rules.

As part of the [Projects Operator installation](#), you can specify the ClusterRole to apply for each [Project](#) using the `CLUSTER_ROLE_REF` environment variable. The TBS installation comes with a ClusterRole called `build-service-user-role` which can be used for this purpose.

## RBAC Support in Tanzu Build Service

Tanzu Build Service is installed with 2 Kubernetes [ClusterRoles](#) that can be used for RBAC for Build Service users and admins:

- `build-service-user-role`
- `build-service-admin-role`

### Build Service User Role

This should be used for users that will create Images and Builds.

To view the configuration for this role:

```
kubectl get clusterrole build-service-user-role -o yaml
```

To use this ClusterRole you should create a [RoleBinding](#) with an existing user.

Example:

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: my-build-service-user-role-binding
  namespace: my-build-namespace
roleRef:
  apiGroup: rbac.authorization.k8s.io
```

```
kind: ClusterRole
name: build-service-user-role
subjects:
- kind: User
  name: my-user
```

## Build Service Admin Role

This should be used for admin users that will operate Tanzu Build Service.

To view the configuration for this role:

```
kubectl get clusterrole build-service-admin-role -o yaml
```

To use this ClusterRole you should create a [RoleBinding](#) or [ClusterRoleBinding](#) with an existing user.

Example:

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-build-service-admin-role-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: build-service-admin-role
subjects:
- kind: User
  name: my-cluster-wide-admin-user
```

# Frequently Asked Questions

## How do Cloud Native Buildpacks (CNBs), kpack, and Tanzu Build Service overlap and differ?

**CNBs** are build tools that adhere to the [CNB v3 Specification](#) and transform source code into an OCI compliant runnable image. The v3 specification, lifecycle, and local CLI ([pack](#)) are governed by the open source [Cloud Native Buildpacks project](#).

[kpack](#) is a collection of open source resource controllers that together function as a Kubernetes native build service. The product provides a declarative image type that builds an image and schedules image rebuilds when dependencies of the image change. kpack is a platform implementation of CNBs in that it utilizes CNBs and the v3 lifecycle to execute image builds.

**Tanzu Build Service** is a commercial product owned and operated by VMware that utilizes kpack and CNBs. Build Service provides additional abstractions intended to ease the use of the above technologies in Enterprise settings. These abstractions are covered in detail throughout the documentation on this site. Additionally, customers of Build Service are entitled to support and VMware Tanzu buildpacks.

---

## Why do I see two images in the image registry after a successful build?

By default Build Service will tag each built image twice. The first tag will be the configured image tag. The second tag will be a unique tag with the build number and build timestamp. The second tag is added to ensure that previous images are not deleted on registries that garbage collect untagged images.

---

## How does TBS work in air gapped environments?

Build Service is installed and deployed using [Carvel](#) tools. Therefore, the [kbuild package](#) command can create a `.tar` file composed of the kubernetes manifests and images required to successfully install Build Service. The [kbuild unpack](#) command ensures that all the images can be relocated to air-gapped registries, and by providing the credentials to the air-gapped registry when executing the [install](#) command, Build Service can then use that secret to pull images from said registry, hence working in air-gapped environments.

For more details on air-gapped installation, see [Installation to Air-Gapped Environment](#).

For more details on air-gapped builds, see [Offline Builds](#).

---

## Is there documentation on supported Tanzu Buildpacks?

Yes, documentation is available on [Tanzu Buildpacks Documentation](#).

---

## Why do I get an X509 error from Build Service when trying to create an image in my registry?

When interacting with a registry or a git repo that has been deployed using a self signed certificate, Build Service must be provided with the certificate during install time. Unfortunately, you will either need to target a registry that does not have self signed certificates or re-install Build Service to work with this registry.

---

## How do I configure a secret to publish images to Dockerhub?

1. Create a dockerhub secret with the `kp` cli:

```
kp secret create my-dockerhub-creds --dockerhub DOCKERHUB-USERNAME
```

Where `DOCKERHUB-USERNAME` is your dockerhub username You will be prompted for your dockerhub password

---

## How can I configure an image to pull from a private GitHub repository?

1. Create a github secret with the `kp` cli:

Using a [git ssh key](#)

```
kp secret create my-git-ssh-cred --git git@github.com --git-ssh-key PATH-TO-GIT
HUB-PRIVATE-KEY
```

Where `PATH-TO-GITHUB-PRIVATE-KEY` is the absolute local path to the github ssh private key

Or with a basic auth github username and password

```
kp secret create my-git-cred --git https://github.com --git-user GITHUB-USERNA
ME
```

Where `GITHUB-USERNAME` is your github username You will be prompted for your github password

---

## Why do some builds fail with "Error: could not read run image: \*"?

The run image must be publicly readable or readable with the registry credentials configured in a project/namespace.

To see where the build service run image is located run: `kp stack status STACK-NAME`.

If you cannot make the run image publicly readable, you must `kp` to create a registry secret within the namespace where your builds reside. This can be accomplished using `kp secret create`.

---

## Why don't my image builds appear in my Harbor v1.X.X registry?

There is a known bug in Harbor that, at times, prevents the UI from showing images. If you are unable to see a recently built image in the Harbor UI, try pulling it using the `docker` CLI to verify that it exists.

---

## How do I fix "unsupported status code 500" when creating a builder on my Harbor v2.X.X registry?

Some builders are very large and can overwhelm Harbor's default database connection. You can remediate this issue by increasing the `database.maxOpenConns` setting in the helm `values.yaml` file. Increase this value from 100 to 300. The exact setting can be found [here](#).

---

## How do I configure credentials for using gcr as my installation registry?

You can use Google Container Registry for your Tanzu Build Service installation registry.

If you have trouble configuring the registry credentials for gcr when following the [install docs](#), use the following to set the gcr credentials:

```
registry_name="_json_key"
registry_password="$(cat /path/to/gcp/service/account/key.json)"
ytt -f /tmp/values.yaml \
  -f /tmp/manifests/ \
  -v docker_repository="" \
  -v docker_username="$registry_name" \
  -v docker_password="$registry_password" \
  | kbld -f /tmp/images-relocated.lock -f- \
  | kapp deploy -a tanzu-build-service -f- -y
```

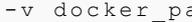
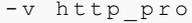
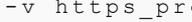
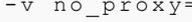
---

## Can I configure a proxy for my Tanzu Build Service?

TBS can be configured with a proxy at [installation](#) time by specifying additional parameters:

- `http_proxy`: The HTTP proxy to use for network traffic.
- `https_proxy`: The HTTPS proxy to use for network traffic.
- `no_proxy`: A comma-separated list of hostnames, IP addresses, or IP ranges in CIDR format that should not use a proxy.

**Note:** When proxy server is enabled using `http_proxy` and/or `https_proxy`, traffic to the kubernetes API server will also flow through the proxy server. This is a known limitation and can be circumvented by using `no_proxy` to specify the kubernetes API server.

```
ytt -f /tmp/values.yaml \
  -f /tmp/manifests/ \
  -v docker_repository="" \
  -v docker_username="" \
  -v docker_password="" \
  -v http_proxy="" \
  -v https_proxy="" \
  -v no_proxy="" \
  | kbuild -f /tmp/images-relocated.lock -f- \
  | kapp deploy -a tanzu-build-service -f- -y
```

## How do I build my app locally using kpack builders?

You can use the [pack cli](#) with your kpack builders to test them locally before checking in your code. By using your kpack builder locally, you can guarantee that the buildpacks, stacks, and lifecycle used to build the image config will also be used by the pack CLI, resulting in a container image that is the exact same, whether it is built by [kpack](#) or [pack](#).

**Note:** Make sure that you `docker login` to the image repository containing your kpack builder.

```
pack build my-app --path ~/workspace/my-app --builder gcr.io/my-project/my-image:latest --trust-builder
```

## What can I do with the kp --dry-run and --output flags?

From kp CLI v1.0.3+ the [--dry-run](#) and [--output](#) flags are made available to kp commands that create or update any kpack Kubernetes resources.

The [--dry-run](#) flag lets you perform a quick validation with no side-effects as no objects are sent to the server. And the [--output](#) flag lets you view the resource in yaml or json format.

The [--dry-run-with-image-upload](#) flag is similar to the [--dry-run](#) flag in that no kpack Kubernetes resources are updated. This flag is provided as a convenience for kp commands that can output Kubernetes resource with generated container image references.

For example, consider the command below

```
$ kp clusterstack create test-stack \
  --dry-run \
  --output yaml \
  --build-image gcr.io/paketo-buildpacks/build@sha256:f550ab24b72586cb26215817b874b9e9e
c2ca615ede03206833286934779ab5d \
  --run-image gcr.io/paketo-buildpacks/run@sha256:21c1fb65033ae5a765a1fb44bfefdea37024c
eac86ac6098202b891d27b8671f

Creating ClusterStack... (dry run)
Uploading to 'gcr.io/my-project/my-repo'... (dry run)
  Skipping 'gcr.io/my-project/my-repo/build@sha256:f550ab24b72586cb26215817b874b
9e9ec2ca615ede03206833286934779ab5d'
  Skipping 'gcr.io/my-project/my-repo/run@sha256:21c1fb65033ae5a765a1fb44bfefdea
37024ceac86ac6098202b891d27b8671f'
apiVersion: kpack.io/v1alpha1
```

```

kind: ClusterStack
metadata:
  creationTimestamp: null
  name: test-stack
spec:
  buildImage:
    image: gcr.io/my-project/my-repo/build@sha256:f550ab24b72586cb26215817b874b9e9ec2c
a615ede03206833286934779ab5d
    id: io.buildpacks.stacks.bionic
  runImage:
    image: gcr.io/my-project/my-repo/run@sha256:21c1fb65033ae5a765a1fb44bfefdea37024ce
ac86ac6098202b891d27b8671f
status:
  buildImage: {}
  runImage: {}

```

The resource yaml output above has the relocated build and run image urls. However, the images were never uploaded.

If you now apply the resource output using `kubectl apply -f` as shown below, then the resource will be created but will be faulty since the referenced images do not exist.

```

$ kp clusterstack create test-stack \
  --dry-run \
  --output yaml \
  --build-image gcr.io/paketo-buildpacks/build@sha256:f550ab24b72586cb26215817b874b9e9e
c2ca615ede03206833286934779ab5d \
  --run-image gcr.io/paketo-buildpacks/run@sha256:21c1fb65033ae5a765a1fb44bfefdea37024c
eac86ac6098202b891d27b8671f \
  | kubectl apply -f -
Creating ClusterStack... (dry run)
Uploading to 'gcr.io/my-project/my-repo'... (dry run)
  Skipping 'gcr.io/my-project/my-repo/build@sha256:f550ab24b72586cb26215817b874b
9e9ec2ca615ede03206833286934779ab5d'
  Skipping 'gcr.io/my-project/my-repo/run@sha256:21c1fb65033ae5a765a1fb44bfefdea
37024ceac86ac6098202b891d27b8671f'
clusterstack.kpack.io/test-stack created

```

Running the same command above with the `--dry-run-with-image-upload` flag (instead of `--dry-run`) ensures the created resource refers to images exist.

## Does TBS support Azure Devops for git repositories

Azure Devops cannot be used as a git repository with TBS. This can be worked-around by using the `--local-path` source type with the `kp` cli which will use your registry to store the source code.

This is expected to be fixed in upcoming releases.

## Why do I get a "repository does not exist" error when I use ECR Registry?

ECR is supported but requires manually creating each repository that TBS will use. With other registries, the repositories will be created automatically.

## How do I troubleshoot a failed build?

Like many Kubernetes native products, operating TBS involves orchestrating resources that depend on each other to function. If a resource is in a "not ready" state it is likely that there is a problem with one of the resources it depends on.

If you are encountering a not ready `Image`, check and see which builder it uses and then check the status of that builder for additional information that could help you troubleshoot the problem.

```
$ kp image status <image-name>

$ kp clusterbuilder status <clusterbuilder-name>
```

Similarly, if a builder resource is in a "not ready" state, it is possible that there is a problem with the `clusterstack` or `clusterstore` resources it is referencing.

```
$ kp clusterstack status <clusterstack-name> --verbose

$ kp clusterstore status <clusterstore-name> --verbose
```

All Build Service concepts are also Kubernetes resources. Therefore, customers can interact with them using the `kubectl` CLI to see all the information that can be provided by the Kubernetes API.

```
$ kubectl describe image <image-name>

$ kubectl describe clusterbuilder <clusterbuilder-name>
```

## How do I troubleshoot an UNAUTHENTICATED error?

During `kbld relocate`

1. Ensure you are logged in locally to both registries with:

```
docker logout registry.pivotal.io && docker login registry.pivotal.io
docker logout <tbs-registry> && docker login <tbs-registry>
```

2. On linux, if you have installed `docker` with `snap` you will need to copy `/root/snap/docker/471/.docker/config.json` to `~/.docker/config.json` which is where `kbld` is looking for the docker credentials
3. Ensure your credentials have write access to your registry with `docker push <registry>/<build-service-repository>` this is the same repository used during install with the `ytt/kapp` command

During `kp import`

1. Ensure you are logged in locally to both registries with:

```
docker logout registry.pivotal.io && docker login registry.pivotal.io
docker logout <tbs-registry> && docker login <tbs-registry>
```

2. Ensure the credentials used to install TBS have write access to your registry as they sometimes differ from local credentials

- Use `docker login <tbs-registry>` using the credentials used to install TBS with `ytt/kapp`
- Try to `docker push <tbs-registry>/<build-service-repository>` this is the same repository used during install with the `ytt/kapp` command

# Additional resources for Tanzu Build Service

## Concourse Kpack resource

The [Concourse Kpack resource](#) helps in the integration of Kpack in a Concourse based CI/CD pipeline. This Concourse resource is capable of triggering Image builds based on a commit SHA. The [Git repo](#) for the Concourse Kpack resource provides guidance on usage within a pipeline.

**Note:** The Kpack Image must be created within a TBS cluster before referring to it within a pipeline using the Concourse Kpack resource

**Note:** The Concourse Kpack resource currently only supports GKE and TKGI clusters

## Helpful Articles

- [Getting Started with VMware Tanzu Build Service 1.0](#)

(September 03, 2020 - Tony Vetter)

This covers installation of Tanzu Build Service on local Kubernetes cluster (using Docker Desktop) and demonstrates the auto build of app images for Code and OS updates.

- [VMware Tanzu Build Service, a Kubernetes-Native Way to Build Containers, Is Now GA](#)

(September 03, 2020 - Brad Bock)

A big picture overview of Tanzu Build Service, integration with CI/CD and links on getting started.

## Helpful Videos

- [Introduction to Tanzu Build Service 1.0](#)

(September 22, 2020 - Tony Vetter)

This covers the different components of TBS, the benefits it offers, and a demo of how TBS can auto update your application images for different reasons - Code update, Config change or Stack update.

## Helpful Repositories

- kpdemo - <https://github.com/matthewmcnew/kpdemo>

A tool to visualize and demo kpack.

Demos include auto Image creation for Stack and Buildpack updates.

# Release Notes

## v1.1.4

**Release Date: March 10, 2021**

- Fixed issue that prevented "smart-warmer" pods from correctly starting in the Build Service Namespace.

## Product Snapshot

Tanzu Build Service 1.1.4 ships with the following components:

- [kpack 0.2.2](#)
- [kpack cli v0.2.0](#)
- [CNB lifecycle v0.10.2](#)
- [Stacks Operator v0.0.4](#)

## v1.1.3

**Release Date: March 08, 2021**

- Build pod pod security context will match user:group of the builder config. Adds support for more restrictive pod security policies.

## Product Snapshot

Tanzu Build Service 1.1.3 ships with the following components:

- [kpack 0.2.2](#)
- [kpack cli v0.2.0](#)
- [CNB lifecycle v0.10.2](#)
- [Stacks Operator v0.0.4](#)

## v1.1.2

**Release Date: March 03, 2021**

- kpack controller resources updated to be more accurate and avoid limit issues.
- stacks operator controller resources updated to be more accurate and avoid limit issues.
- CustomResourceDefinitions migrated to [apiextensions.k8s.io/v1](#).

## Product Snapshot

Tanzu Build Service 1.1.2 ships with the following components:

- [kpack 0.2.1](#)
- [kpack cli v0.2.0](#)
- [CNB lifecycle v0.10.2](#)
- [Stacks Operator v0.0.4](#)

### v1.1.1

**Release Date: February 08, 2021**

**1.1.1** provides a patch of 1.1.0, which fixes an install issue where relocated images locations were not being properly written.

### v1.1.0

**Release Date: February 05, 2021**

**NOTE: v1.1.0** has an install issue which will prevent kbuild from writing the correct image location. Please use **v1.1.1**.

**1.1.0** represents the second minor GA release of Tanzu Build Service.

This release includes a few notable new features:

- Windows .NET Framework images can be built on clusters with windows nodes.
- Image resources can be configured to sign images with Notary.
- The kpack cli can upgrade the version of the lifecycle.
- Improved logging for Image builds
  - ◊ Display Build reason change diff
  - ◊ Log fetching source
  - ◊ More descriptive errors

This release includes the 0.2.0 release of kpack.

- Image resources can be configured to sign images with Notary.
- Build logs will display a diff explaining why the build was scheduled.
- The lifecycle used by kpack can be updated without a controller restart.

This release includes the 0.2.0 release of kpack-cli

- `kp import` supports v1alpha3 descriptors
- kpack lifecycle image can be updated.
- `kp import` will wait for relevant resources to resolve before exiting.
- `kp image list` Image list can be filtered.

## Product Snapshot

Tanzu Build Service 1.1.0 ships with the following components:

- [kpack 0.2.0](#)
- [kpack cli v0.2.0](#)
- [CNB lifecycle v0.10.2](#)
- [\[Stacks Operator v0.0.3\]](#)

Tanzu Build Service supports and utilizes [Tanzu Buildpacks](#).

## Product Dependencies

Build Service can be installed on any Kubernetes cluster (v1.16 or later).

## Upgrade Path

v1.0.\* can be upgraded to v1.1.\*. Please follow the [install](#) process to upgrade.