OpenShift Dedicated 4

CLI ツール

OpenShift CLI の使用方法について

Last Updated: 2020-05-15

OpenShift Dedicated 4 CLI ツール

OpenShift CLI の使用方法について

法律上の通知

概要

本書では、OpenShift CLI (oc) の設定および使用について説明します。また、CLI コマンドの参照情報、およびそれらの使用方法についての例も記載しています。

目次

第1章 OPENSHIFT CLI (OC)

1.1. CLI の使用方法

1.1.1. CLI について

OpenShift Dedicated のコマンドラインインターフェース (CLI) を使用すると、ターミナルからアプリケーションを作成し、OpenShift Dedicated のプロジェクトを管理できます。CLI の使用は、以下の場合に適しています。

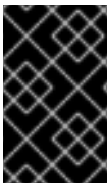
- プロジェクトのソースコードを直接使用している。
- OpenShift Dedicated 操作をスクリプト化する。
- 帯域幅リソースの制限下にあり、Web コンソールを使用できない。

1.1.2. CLI のインストール

OpenShift CLI（**oc**）をインストールするには、バイナリーをダウンロードするか、RPM を使用します。

1.1.2.1. バイナリーのダウンロードによる CLI のインストール

コマンドラインインターフェースを使用して OpenShift Dedicated と対話するために CLI をインストールすることができます。

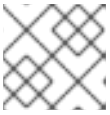


重要

以前のバージョンの **oc** をインストールしている場合、これを使用して OpenShift Dedicated 4.3 のすべてのコマンドを実行することはできません。新規バージョンの **oc**をダウンロードし、インストールします。

手順

1. Red Hat OpenShift Cluster Manager サイトの Infrastructure Provider ページから、選択するインストールタイプのページに移動し、**Download Command-line Tools**をクリックします。
2. オペレーティングシステムおよびアーキテクチャーのフォルダーをクリックしてから、圧縮されたファイルをクリックします。



注記

oc は Linux、Windows、または macOS にインストールできます。

3. ファイルをファイルシステムに保存します。
4. 圧縮ファイルを展開します。
5. これを **PATH** にあるディレクトリーに配置します。

CLI のインストール後は、**oc** コマンドを使用して利用できます。

```
$ oc <command>
```

-

1.1.2.2. RPM を使用した CLI のインストール

Red Hat Enterprise Linux (RHEL) の場合、Red Hat アカウントに有効な OpenShift Dedicated サブスクリプションがある場合は、OpenShift CLI (**oc**) を RPM としてインストールできます。

前提条件

- root または sudo の権限が必要です。

手順

1. Red Hat Subscription Manager に登録します。

```
# subscription-manager register
```

2. 最新のサブスクリプションデータをプルします。

```
# subscription-manager refresh
```

3. 利用可能なサブスクリプションを一覧表示します。

```
# subscription-manager list --available --matches '*OpenShift*'
```

4. 直前のコマンドの出力で、OpenShift Dedicated サブスクリプションのプール ID を見つけ、これを登録されたシステムにアタッチします。

```
# subscription-manager attach --pool=<pool_id>
```

5. OpenShift Dedicated 4.3 で必要なリポジトリーを有効にします。

```
# subscription-manager repos --enable="rhel-7-server-ose-4.3-rpms"
```

6. **openshift-clients** パッケージをインストールします。

```
# yum install openshift-clients
```

CLI のインストール後は、**oc** コマンドを使用して利用できます。

```
$ oc <command>
```

1.1.3. CLI へのログイン

oc CLI にログインしてクラスターにアクセスし、これを管理できます。

前提条件

- OpenShift Dedicated クラスターへのアクセスがあること。
- CLI をインストールしていること。

手順

- **oc login** コマンドを使用して CLI にログインし、プロンプトが出されたら必要な情報を入力します。

```
$ oc login
Server [https://localhost:8443]: https://openshift.example.com:6443 1
The server uses a certificate signed by an unknown authority.
You can bypass the certificate check, but any data you send to the server could be
intercepted by others.
Use insecure connections? (y/n): y 2

Authentication required for https://openshift.example.com:6443 (openshift)
Username: user1 3
Password: 4
Login successful.

You don't have any projects. You can try to create a new project, by running

    oc new-project <projectname>

Welcome! See 'oc help' to get started.
```

1. OpenShift Dedicated サーバーの URL を入力します。
2. 非セキュアな接続を使用するかどうかを入力します。
3. ログインに使用するユーザー名を入力します。
4. ユーザーのパスワードを入力します。

これで、プロジェクトを作成でき、クラスターを管理するための他のコマンドを実行することができます。

1.1.4. CLI の使用

以下のセクションで、CLI を使用して一般的なタスクを実行する方法を確認します。

1.1.4.1. プロジェクトの作成

新規プロジェクトを作成するには、**oc new-project** コマンドを使用します。

```
$ oc new-project my-project
Now using project "my-project" on server "https://openshift.example.com:6443".
```

1.1.4.2. 新しいアプリケーションの作成

新規アプリケーションを作成するには、**oc new-app** コマンドを使用します。

```
$ oc new-app https://github.com/sclorg/cakephp-ex
--> Found image 40de956 (9 days old) in imagestream "openshift/php" under tag "7.2" for "php"
```

```
...

    Run 'oc status' to view your app.
```

1.1.4.3. Pod の表示

現在のプロジェクトの Pod を表示するには、**oc get pods** コマンドを使用します。

```
$ oc get pods -o wide
NAME                READY   STATUS      RESTARTS   AGE     IP           NODE
NOMINATED NODE
cakephp-ex-1-build  0/1     Completed   0          5m45s   10.131.0.10  ip-10-0-141-74.ec2.internal
<none>
cakephp-ex-1-deploy  0/1     Completed   0          3m44s   10.129.2.9   ip-10-0-147-65.ec2.internal
<none>
cakephp-ex-1-ktz97  1/1     Running     0          3m33s   10.128.2.11  ip-10-0-168-105.ec2.internal
<none>
```

1.1.4.4. Pod ログの表示

特定の Pod のログを表示するには、**oc logs** コマンドを使用します。

```
$ oc logs cakephp-ex-1-deploy
--> Scaling cakephp-ex-1 to 1
--> Success
```

1.1.4.5. 現在のプロジェクトの表示

現在のプロジェクトを表示するには、**oc project** コマンドを使用します。

```
$ oc project
Using project "my-project" on server "https://openshift.example.com:6443".
```

1.1.4.6. 現在のプロジェクトのステータスの表示

サービス、DeploymentConfig、および BuildConfig などの現在のプロジェクトについての情報を表示するには、**oc status** コマンドを使用します。

```
$ oc status
In project my-project on server https://openshift.example.com:6443

svc/cakephp-ex - 172.30.236.80 ports 8080, 8443
  dc/cakephp-ex deploys istag/cakephp-ex:latest <-
    bc/cakephp-ex source builds https://github.com/sclorg/cakephp-ex on openshift/php:7.2
    deployment #1 deployed 2 minutes ago - 1 pod

3 infos identified, use 'oc status --suggest' to see details.
```

1.1.4.7. サポートされる API のリソースの一覧表示

サーバー上でサポートされる API リソースの一覧を表示するには、**oc api-resources** コマンドを使用します。

```
$ oc api-resources
NAME                              SHORTNAMES   APIGROUP                       NAMESPACED   KIND
bindings                                                                      true         Binding
componentstatuses                 cs                                          false        ComponentStatus
configmaps                        cm                                          true         ConfigMap
...
```

1.1.5. ヘルプの表示

CLI コマンドおよび OpenShift Dedicated リソースに関するヘルプを以下の方法で表示することができます。

- 利用可能なすべての CLI コマンドの一覧および説明を表示するには、**oc help** を使用します。

  例: **CLI** についての一般的なヘルプの表示

  ```
  $ oc help
  OpenShift Client

  This client helps you develop, build, deploy, and run your applications on any OpenShift or
  Kubernetes compatible
  platform. It also includes the administrative commands for managing a cluster under the 'adm'
  subcommand.

  Usage:
    oc [flags]

  Basic Commands:
    login           Log in to a server
    new-project     Request a new project
    new-app         Create a new application

  ...
  ```

- 特定の CLI コマンドについてのヘルプを表示するには、**--help** フラグを使用します。

  例: **oc create** コマンドについてのヘルプの表示

  ```
  $ oc create --help
  Create a resource by filename or stdin

  JSON and YAML formats are accepted.

  Usage:
    oc create -f FILENAME [flags]

  ...
  ```

- 特定リソースについての説明およびフィールドを表示するには、**oc explain** コマンドを使用します。

  例: **Pod** リソースのドキュメントの表示

  ```
  $ oc explain pods
  ```

```
KIND:     Pod
VERSION:  v1

DESCRIPTION:
     Pod is a collection of containers that can run on a host. This resource is
     created by clients and scheduled onto hosts.

FIELDS:
   apiVersion <string>
     APIVersion defines the versioned schema of this representation of an
     object. Servers should convert recognized schemas to the latest internal
     value, and may reject unrecognized values. More info:
     https://git.k8s.io/community/contributors/devel/api-conventions.md#resources

...
```

1.1.6. CLI からのログアウト

CLI からログアウトし、現在のセッションを終了することができます。

- **oc logout** コマンドを使用します。

```
$ oc logout
Logged "user1" out on "https://openshift.example.com"
```

これにより、サーバーから保存された認証トークンが削除され、設定ファイルから除去されます。

1.2. CLI の設定

1.2.1. タブ補完の有効化

oc CLI ツールをインストールした後に、タブ補完を有効にして **oc** コマンドの自動補完を実行するか、または Tab キーを押す際にオプションの提案が表示されるようにできます。

前提条件

- **oc** CLI ツールをインストールしていること。

手順

以下の手順では、Bash のタブ補完を有効にします。

1. Bash 補完コードをファイルに保存します。

```
$ oc completion bash > oc_bash_completion
```

2. ファイルを **/etc/bash_completion.d/** にコピーします。

```
$ sudo cp oc_bash_completion /etc/bash_completion.d/
```

さらにファイルをローカルディレクトリーに保存した後に、これを **.bashrc** ファイルから取得できるようにすることができます。

タブ補完は、新規ターミナルを開くと有効にされます。

1.3. 開発者の CLI コマンド

1.3.1. 基本的な CLI コマンド

1.3.1.1. explain

特定リソースのドキュメントを表示します。

例: Pod のドキュメントの表示

```
$ oc explain pods
```

1.3.1.2. login

OpenShift Dedicated サーバーにログインし、後続の使用のためにログイン情報を保存します。

例: 対話型ログイン

```
$ oc login
```

例: ユーザー名を指定したログイン

```
$ oc login -u user1
```

1.3.1.3. new-app

ソースコード、テンプレート、またはイメージを指定して新規アプリケーションを作成します。

例: ローカル Git リポジトリーからの新規アプリケーションの作成

```
$ oc new-app .
```

例: リモート Git リポジトリーからの新規アプリケーションの作成

```
$ oc new-app https://github.com/sclorg/cakephp-ex
```

例: プライベートリモートリポジトリーからの新規アプリケーションの作成

```
$ oc new-app https://github.com/youruser/yourprivaterepo --source-secret=yoursecret
```

1.3.1.4. new-project

新規プロジェクトを作成し、設定のデフォルトのプロジェクトとしてこれに切り替えます。

例: 新規プロジェクトの作成

```
$ oc new-project myproject
```

1.3.1.5. project

別のプロジェクトに切り替えて、これを設定でデフォルトにします。

例: 別のプロジェクトへの切り替え

```
$ oc project test-project
```

1.3.1.6. projects

現在のアクティブなプロジェクトおよびサーバー上の既存プロジェクトについての情報を表示します。

例: すべてのプロジェクトの一覧表示

```
$ oc projects
```

1.3.1.7. status

現在のプロジェクトのハイレベルの概要を表示します。

例: 現在のプロジェクトのステータスの表示

```
$ oc status
```

1.3.2. CLI コマンドのビルドおよびデプロイ

1.3.2.1. cancel-build

実行中、保留中、または新規のビルドを取り消します。

例：ビルドの取り消し

```
$ oc cancel-build python-1
```

例: **python BuildConfig** からの保留中のすべてのビルドの取り消し

```
$ oc cancel-build buildconfig/python --state=pending
```

1.3.2.2. import-image

イメージリポジトリーから最新のタグおよびイメージ情報をインポートします。

例: 最新のイメージ情報のインポート

```
$ oc import-image my-ruby
```

1.3.2.3. new-build

ソースコードから新規の **BuildConfig** を作成します。

例: ローカル **Git** リポジトリーからの **BuildConfig** の作成

```
$ oc new-build .
```

例: リモート Git リポジトリーからの BuildConfig の作成

```
$ oc new-build https://github.com/sclorg/cakephp-ex
```

1.3.2.4. rollback

アプリケーションを以前のデプロイメントに戻します。

例: 最後に成功したデプロイメントへのロールバック

```
$ oc rollback php
```

例: 特定バージョンへのロールバック

```
$ oc rollback php --to-version=3
```

1.3.2.5. rollout

新規ロールアウトを開始し、そのステータスまたは履歴を表示するか、またはアプリケーションの以前のバージョンにロールバックします。

例: 最後に成功したデプロイメントへのロールバック

```
$ oc rollout undo deploymentconfig/php
```

例: 最新状態の DeploymentConfig の新規ロールアウトの開始

```
$ oc rollout latest deploymentconfig/php
```

1.3.2.6. start-build

BuildConfig からビルドを開始するか、または既存ビルドをコピーします。

例: 指定された BuildConfig からのビルドの開始

```
$ oc start-build python
```

例: 以前のビルドからのビルドの開始

```
$ oc start-build --from-build=python-1
```

例: 現在のビルドに使用する環境変数の設定

```
$ oc start-build python --env=mykey=myvalue
```

1.3.2.7. tag

既存のイメージをイメージストリームにタグ付けします。

例: **ruby** イメージの **latest** タグを **2.0** タグのイメージを参照するように設定する

```
$ oc tag ruby:latest ruby:2.0
```

1.3.3. アプリケーション管理 CLI コマンド

1.3.3.1. annotate

1 つ以上のリソースでアノテーションを更新します。

例: アノテーションのルートへの追加

```
$ oc annotate route/test-route haproxy.router.openshift.io/ip_whitelist="192.168.1.10"
```

例: ルートからのアノテーションの削除

```
$ oc annotate route/test-route haproxy.router.openshift.io/ip_whitelist-
```

1.3.3.2. apply

JSON または YAML 形式のファイル名または標準入力 (stdin) 別に設定をリソースに適用します。

例: **pod.json** の設定の **Pod** への適用

```
$ oc apply -f pod.json
```

1.3.3.3. autoscale

DeploymentConfig または ReplicationController の自動スケーリングを実行します。

例: 最小の 2 つおよび最大の 5 つの **Pod** への自動スケーリング

```
$ oc autoscale deploymentconfig/parksmap-katacoda --min=2 --max=5
```

1.3.3.4. create

JSON または YAML 形式のファイル名または標準入力 (stdin) 別にリソースを作成します。

例: **pod.json** の内容を使用した **Pod** の作成

```
$ oc create -f pod.json
```

1.3.3.5. delete

リソースを削除します。

例: **parksmap-katacoda-1-qfqz4** という名前の **Pod** の削除

```
$ oc delete pod/parksmap-katacoda-1-qfqz4
```

例: **app=parksmap-katacoda** ラベルの付いたすべての Pod の削除

```
$ oc delete pods -l app=parksmap-katacoda
```

1.3.3.6. describe

特定のオブジェクトに関する詳細情報を返します。

例: **example** という名前のデプロイメントの記述

```
$ oc describe deployment/example
```

例: すべての Pod の記述

```
$ oc describe pods
```

1.3.3.7. edit

リソースを編集します。

例: デフォルトエディターを使用した DeploymentConfig の編集

```
$ oc edit deploymentconfig/parksmap-katacoda
```

例: 異なるエディターを使用した DeploymentConfig の編集

```
$ OC_EDITOR="nano" oc edit deploymentconfig/parksmap-katacoda
```

例: JSON 形式の DeploymentConfig の編集

```
$ oc edit deploymentconfig/parksmap-katacoda -o json
```

1.3.3.8. expose

ルートとしてサービスを外部に公開します。

例: サービスの公開

```
$ oc expose service/parksmap-katacoda
```

例: サービスの公開およびホスト名の指定

```
$ oc expose service/parksmap-katacoda --hostname=www.my-host.com
```

1.3.3.9. get

1 つ以上のリソースを表示します。

例: **default namespace** の **Pod** の一覧表示

```
$ oc get pods -n default
```

例: **JSON** 形式の **python DeploymentConfig** についての詳細の取得

```
$ oc get deploymentconfig/python -o json
```

1.3.3.10. label

1 つ以上のリソースでアノテーションを更新します。

例: **python-1-mz2rf Pod** の **unhealthy** に設定されたラベル **status** での更新

```
$ oc label pod/python-1-mz2rf status=unhealthy
```

1.3.3.11. scale

ReplicationController または DeploymentConfig の必要なレプリカ数を設定します。

例: **ruby-app DeploymentConfig** の 3 つの **Pod** へのスケーリング

```
$ oc scale deploymentconfig/ruby-app --replicas=3
```

1.3.3.12. secrets

プロジェクトのシークレットを管理します。

例: **my-pull-secret** の、**default** サービスアカウントによるイメージプルシークレットとしての使用を許可

```
$ oc secrets link default my-pull-secret --for=pull
```

1.3.3.13. serviceaccounts

サービスアカウントに割り当てられたトークンを取得するか、またはサービスアカウントの新規トークンまたは **kubeconfig** ファイルを作成します。

例: **default** サービスアカウントに割り当てられたトークンの取得

```
$ oc serviceaccounts get-token default
```

1.3.3.14. set

既存のアプリケーションリソースを設定します。

例: **BuildConfig** でのシークレットの名前の設定

```
$ oc set build-secret --source buildconfig/mybc mysecret
```

1.3.4. CLI コマンドのトラブルシューティングおよびデバッグ

1.3.4.1. attach

実行中のコンテナーにシェルを割り当てます。

例: **Pod python-1-mz2rf** の **python** コンテナーからの出力の取得

```
$ oc attach python-1-mz2rf -c python
```

1.3.4.2. cp

ファイルおよびディレクトリーのコンテナーへの/からのコピーを実行します。

例: **python-1-mz2rf Pod** からローカルファイルシステムへのファイルのコピー

```
$ oc cp default/python-1-mz2rf:/opt/app-root/src/README.md ~/mydirectory/.
```

1.3.4.3. debug

コマンドシェルを起動して、実行中のアプリケーションをデバッグします。

例: **python** デプロイメントのデバッグ

```
$ oc debug deploymentconfig/python
```

1.3.4.4. exec

コンテナーでコマンドを実行します。

例: **ls** コマンドの **Pod python-1-mz2rf** の **python** コンテナーでの実行

```
$ oc exec python-1-mz2rf -c python ls
```

1.3.4.5. logs

特定のビルド、BuildConfig、DeploymentConfig、または Pod のログ出力を取得します。

例: **python DeploymentConfig** からの最新ログのストリーミング

```
$ oc logs -f deploymentconfig/python
```

1.3.4.6. port-forward

1 つ以上のポートを Pod に転送します。

例: ポート **8888** でのローカルのリッスンおよび **Pod** のポート **5000** への転送

```
$ oc port-forward python-1-mz2rf 8888:5000
```

1.3.4.7. proxy

Kubernetes API サーバーに対してプロキシーを実行します。

例: **./local/www/** から静的コンテンツを提供するポート **8011** の **API** サーバーに対するプロキシーの実行

```
$ oc proxy --port=8011 --www=./local/www/
```

1.3.4.8. rsh

コンテナーへのリモートシェルセッションを開きます。

例: **python-1-mz2rf Pod** の最初のコンテナーでシェルセッションを開く

```
$ oc rsh python-1-mz2rf
```

1.3.4.9. rsync

ディレクトリーの内容の実行中の Pod コンテナーへの/からのコピーを実行します。変更されたファイルのみが、オペレーティングシステムから **rsync** コマンドを使用してコピーされます。

例: ローカルディレクトリーのファイルの **Pod** ディレクトリーとの同期

```
$ oc rsync ~/mydirectory/ python-1-mz2rf:/opt/app-root/src/
```

1.3.4.10. run

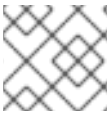
特定のイメージを作成し、実行します。デフォルトでは、これにより作成されたコンテナーを管理するための DeploymentConfig が作成されます。

例: 3 つのレプリカを持つ **perl** イメージのインスタンスの開始

```
$ oc run my-test --image=perl --replicas=3
```

1.3.4.11. wait

1 つ以上のリソースの特定の条件を待機します。



注記

このコマンドは実験的なもので、通知なしに変更される可能性があります。

例: **python-1-mz2rf Pod** の削除の待機

```
$ oc wait --for=delete pod/python-1-mz2rf
```

1.3.5. 上級開発者の CLI コマンド

1.3.5.1. api-resources

サーバーがサポートする API リソースの詳細の一覧を表示します。

例: サポートされている **API** リソースの一覧表示

```
$ oc api-resources
```

1.3.5.2. api-versions

サーバーがサポートする API バージョンの詳細の一覧を表示します。

例: サポートされている **API** バージョンの一覧表示

```
$ oc api-versions
```

1.3.5.3. auth

パーミッションを検査し、RBAC ロールを調整します。

例: 現行ユーザーが **Pod** ログを読み取ることができるかどうかのチェック

```
$ oc auth can-i get pods --subresource=log
```

例: ファイルの **RBAC** ロールおよびパーミッションの調整

```
$ oc auth reconcile -f policy.json
```

1.3.5.4. cluster-info

マスターおよびクラスターサービスのアドレスを表示します。

例: クラスター情報の表示

```
$ oc cluster-info
```

1.3.5.5. convert

YAML または JSON 設定ファイルを異なる API バージョンに変換し、標準出力 (stdout) に出力します。

例: **pod.yaml** の最新バージョンへの変換

```
$ oc convert -f pod.yaml
```

1.3.5.6. extract

ConfigMap またはシークレットの内容を抽出します。ConfigMap またはシークレットのそれぞれのキーがキーの名前を持つ別個のファイルとして作成されます。

例: **ruby-1-ca ConfigMap** の内容の現行ディレクトリーへのダウンロード

```
$ oc extract configmap/ruby-1-ca
```

例: **ruby-1-ca ConfigMap** の内容の標準出力 **(stdout)** への出力

```
$ oc extract configmap/ruby-1-ca --to=-
```

1.3.5.7. idle

スケーラブルなリソースをアイドリングします。アイドリングされたサービスは、トラフィックを受信するとアイドリング解除されます。 これは **oc scale** コマンドを使用して手動でアイドリング解除することもできます。

例: **ruby-app** サービスのアイドリング

```
$ oc idle ruby-app
```

1.3.5.8. image

OpenShift Dedicated クラスターのイメージを管理します。

例: イメージの別のタグへのコピー

```
$ oc image mirror myregistry.com/myimage:latest myregistry.com/myimage:stable
```

1.3.5.9. observe

リソースの変更を監視し、それらの変更に対するアクションを取ります。

例: サービスへの変更の監視

```
$ oc observe services
```

1.3.5.10. patch

JSON または YAML 形式のストテラテジーに基づくマージパッチを使用してオブジェクトの 1 つ以上のフィールドを更新します。

例: ノード **node1** の **spec.unschedulable** フィールドの **true** への更新

```
$ oc patch node/node1 -p '{"spec":{"unschedulable":true}}'
```

注記

カスタムリソース定義 (Custom Resource Definition) のパッチを適用する必要がある場合、コマンドに **--type merge** オプションを含める必要があります。

1.3.5.11. policy

認可ポリシーを管理します。

例: **edit** ロールの現在のプロジェクトの **user1** への追加

```
$ oc policy add-role-to-user edit user1
```

1.3.5.12. process

リソースの一覧に対してテンプレートを処理します。

例: **template.json** をリソース一覧に変換し、 **oc create** に渡す

```
$ oc process -f template.json | oc create -f -
```

1.3.5.13. レジストリー

OpenShift Dedicated で統合レジストリーを管理します。

例: 統合レジストリーについての情報の表示

```
$ oc registry info
```

1.3.5.14. replace

指定された設定ファイルに基づいて既存オブジェクトを変更します。

例: **pod.json** の内容を使用した Pod の更新

```
$ oc replace -f pod.json
```

1.3.6. CLI コマンドの設定

1.3.6.1. completion

指定されたシェルのシェル補完コードを出力します。

例: Bash の補完コードの表示

```
$ oc completion bash
```

1.3.6.2. config

クライアント設定ファイルを管理します。

例: 現在の設定の表示

```
$ oc config view
```

例: 別のコンテキストへの切り替え

```
$ oc config use-context test-context
```

1.3.6.3. logout

現行のセッションからログアウトします。

例: 現行セッションの終了

```
$ oc logout
```

1.3.6.4. whoami

現行セッションに関する情報を表示します。

例: 現行の認証ユーザーの表示

```
$ oc whoami
```

1.3.7. 他の開発者 CLI コマンド

1.3.7.1. help

CLI の一般的なヘルプ情報および利用可能なコマンドの一覧を表示します。

例: 利用可能なコマンドの表示

```
$ oc help
```

例: **new-project** コマンドのヘルプの表示

```
$ oc help new-project
```

1.3.7.2. plugin

ユーザーの **PATH** に利用可能なプラグインを一覧表示します。

例: 利用可能なプラグインの一覧表示

```
$ oc plugin list
```

1.3.7.3. version

oc クライアントおよびサーバーのバージョンを表示します。

例: バージョン情報の表示

```
$ oc version
```

1.4. OC および KUBECTL コマンドの使用

Kubernetes のコマンドラインインターフェース (CLI) **kubectl** は、Kubernetes クラスターに対してコマンドを実行するために使用されます。OpenShift Container Platform は認定 Kubernetes ディストリ

ビューションであるため、OpenShift Dedicated に同梱されるサポート対象の **kubectl** バイナリーを使用するか、または **oc** バイナリーを使用して拡張された機能を取得できます。

1.4.1. oc バイナリー

oc バイナリーは **kubectl** バイナリーと同じ機能を提供しますが、これは、以下を含む OpenShift Dedicated 機能をネイティブにサポートするように拡張されています。

- **OpenShift Dedicated リソースの完全サポート**
  DeploymentConfigs、BuildConfigs、Routes、ImageStreams、および ImageStreamTags などのリソースは OpenShift Dedicated ディストリビューションに固有のリソースであり、標準の Kubernetes プリミティブにビルドされます。

- **認証**
  oc バイナリーは、認証を可能にするビルトインの **login** コマンドを提供し、Kubernetes namespace を認証ユーザーにマップする OpenShift Dedicated プロジェクトを使って作業できるようにします。詳細は、「Understanding authentication」を参照してください。

- **追加コマンド**
  追加コマンドの **oc new-app** などは、既存のソースコードまたは事前にビルドされたイメージを使用して新規アプリケーションを起動することを容易にします。同様に、追加コマンドの **oc new-project** により、デフォルトとして切り替えることができるプロジェクトを簡単に開始できるようになります。

1.4.2. kubectl バイナリー

kubectl バイナリーは、標準の Kubernetes 環境を使用する新規 OpenShift Dedicated ユーザー、または **kubectl** CLI を優先的に使用するユーザーの既存ワークフローおよびスクリプトをサポートする手段として提供されます。**kubectl** の既存ユーザーは引き続きバイナリーを使用し、OpenShift Dedicated クラスターに必要な変更なしに Kubernetes のプリミティブと対話できます。

詳細は kubectl ドキュメントを参照してください。

第2章 OPENSHIFT DO DEVELOPER CLI (ODO)

2.1. OPENSHIFT DO について

OpenShift Do (**odo**) は、アプリケーションを OpenShift Dedicated で作成するための高速で使いやすい CLI ツールです。**odo** を使用する開発者は、 OpenShift Dedicated クラスター自体を管理する必要なしにアプリケーションの作成に集中することができます。デプロイメント設定、ビルド設定、サービスルートおよび他の OpenShift Dedicated 要素の作成は、すべて **odo** によって自動化されます。

oc などの既存ツールは操作により重点が置かれ、Kubernetes および OpenShift Dedicated の概念のより深い理解が必要です。 **odo** は Kubernetes および OpenShift 概念の複雑な部分を取り除き、開発者にとって最も重要な「コード」にフォーカスできるようにします。

2.1.1. 主な特長

odo は、以下の主な特長によって単純化および簡潔化されるように設計されています。

- プロジェクト、アプリケーションおよびコンポーネントなどの開発者にとって馴染みのある概念を中心とした単純な構文および設計。
- 完全にクライアントベースである。デプロイにあたって OpenShift Dedicated クラスター内のサーバーは不要です。
- Node.js および Java コンポーネントの正式なサポート。
- Ruby、Perl、PHP、Python などの言語およびフレームワークとの部分的な互換性。
- ローカルコードの変更を検出し、これをクラスターに自動的にデプロイ。これにより、変更を検証するためのインスタントフィードバックがリアルタイムに提供されます。
- OpenShift Dedicated クラスターのすべての利用可能なコンポーネントおよびサービスを一覧表示。

2.1.2. コアとなる概念

Project

Project (プロジェクト) は、別個の単一の単位で編成されるソースコード、テスト、ライブラリーです。

Application

Application (アプリケーション) は、エンドユーザー向けに設計されたプログラムです。アプリケーションは、アプリケーション全体を構築するために個別に動作する複数のマイクロサービスまたはコンポーネントで構成されます。アプリケーションの例: ビデオゲーム、メディアプレイヤー、Web ブラウザー。

Component

コンポーネントとは、コードまたはデータをホストする Kubernetes リソースのセットです。各コンポーネントは個別に実行され、デプロイできます。コンポーネントの例: Node.js、Perl、PHP、Python、Ruby

サービス

Service (サービス) は、コンポーネントのリンク先となるか、またはコンポーネントが依存するソフトウェアです。サービスの例: MariaDB、Jenkins、MySQL**odo** では、サービスは OpenShift Service Catalog からプロビジョニングされ、クラスター内で有効にされる必要があります。

2.1.2.1. 正式にサポートされる言語と対応するコンテナーイメージ

表2.1 サポートされる言語、コンテナーイメージ、およびパッケージマネージャー

言語	コンテナーイメージ	パッケージマネージャー
Node.js	centos/nodejs-8-centos7	NPM
	rhoar-nodejs/nodejs-8	NPM
	bucharestgold/centos7-s2i-nodejs	NPM
	rhscl/nodejs-8-rhel7	NPM
	rhscl/nodejs-10-rhel7	NPM
Java	redhat-openjdk-18/openjdk18-openshift	Maven、Gradle
	openjdk/openjdk-11-rhel8	Maven、Gradle
	openjdk/openjdk-11-rhel7	Maven、Gradle

2.1.2.1.1. 利用可能なコンテナーイメージの一覧表示

注記

利用可能なコンテナーイメージの一覧は、クラスターの内部コンテナーレジストリーおよびクラスターに関連付けられた外部レジストリーから取得されます。

利用可能なコンポーネントおよびクラスターの関連付けられたコンテナーイメージを一覧表示するには、以下を実行します。

1. **odo** を使用して OpenShift Dedicated クラスターにログインします。

```
$ odo login -u developer -p developer
```

2. 利用可能な **odo** がサポートするコンポーネントとサポートしないコンポーネント、および対応するコンテナーイメージを一覧表示します。

```
$ odo catalog list components
Odo Supported OpenShift Components:
NAME        PROJECT      TAGS
java        openshift     8,latest
nodejs      openshift     10,8,8-RHOAR,latest

Odo Unsupported OpenShift Components:
NAME                    PROJECT       TAGS
dotnet                  openshift     1.0,1.1,2.1,2.2,latest
fuse7-eap-openshift     openshift     1.3
```

-

TAGS コラムは利用可能なイメージバージョンを表します (例: **10** は **rhoar-nodejs/nodejs-10** コンテナーイメージを表します）。

2.2. ODO アーキテクチャー

このセクションでは、**odo** アーキテクチャーについて説明し、**odo** による OpenShift Dedicated リソースのクラスターでの管理方法について説明します。

2.2.1. 開発者の設定

odo を使用すると、ターミナルを使って OpenShift Dedicated クラスターでアプリケーションを作成し、デプロイできます。コードエディタープラグインは、ユーザーがそれぞれの IDE ターミナルから OpenShift Dedicated クラスターと対話することを可能にする odo を使用します。odo を使用するプラグインの例: VS Code Openshift Connector、Openshift Connector for Intellij、Codewind for Eclipse Che。

odo は Windows、macOS、および Linux のオペレーティングシステムで機能し、すべてのターミナルから使用できます。odo は bash および zsh コマンドラインシェルの自動補完を提供します。

odo 1.1.0 は Node.js および Java コンポーネントをサポートします。

2.2.2. OpenShift Source-to-Image (S2I)

OpenShift Source-to-Image (S2I) はオープンソースプロジェクトであり、ソースコードからアーティファクトをビルドし、これらをコンテナーイメージに挿入するのに役立ちます。S2I は、Dockerfile なしにソースコードをビルドすることで、実行可能なイメージを生成します。odo は、コンテナー内で開発者ソースコードを実行するために S2I ビルダーイメージを使用します。

2.2.3. OpenShift クラスターオブジェクト

2.2.3.1. Init コンテナー

init コンテナーはアプリケーションコンテナーが起動する前に実行される特殊なコンテナーであり、アプリケーションコンテナーの実行に必要な環境を設定します。init コンテナーには、アプリケーションイメージにないファイル (設定スクリプトなど) を含めることができます。Init コンテナーは常に完了するまで実行され、Init コンテナーのいずれかに障害が発生した場合にはアプリケーションコンテナーは起動しません。

odo によって作成された Pod は 2 つの Init コンテナーを実行します。

- **copy-supervisord** Init コンテナー。
- **copy-files-to-volume** Init コンテナー。

2.2.3.1.1. copy-supervisord

copy-supervisord Init コンテナーは必要なファイルを **emptyDir** ボリュームにコピーします。メインのアプリケーションコンテナーはこれらのファイルを **emptyDir** ボリュームから使用します。

emptyDir ボリュームにコピーされるファイル:

- バイナリー:

- **go-init** は最小限の init システムです。アプリケーションコンテナー内の最初のプロセス (PID 1) として実行されます。go-init は、開発者コードを実行する **SupervisorD** デーモンを起動します。go-init は、孤立したプロセスを処理するために必要です。

- **SupervisorD** はプロセス制御システムです。これは設定されたプロセスを監視し、それらが実行中であることを確認します。また、必要に応じてサービスを再起動します。odo の場合、**SupervisorD** は開発者コードを実行し、監視します。

- 設定ファイル:

  - **supervisor.conf** は、SupervisorD デーモンの起動に必要な設定ファイルです。

- スクリプト:

  - **assemble-and-restart** は、ユーザーソースコードをビルドし、デプロイするための OpenShift S2I の概念です。assemble-and-restart スクリプトは、まずアプリケーションコンテナー内でユーザーソースコードをアセンブルしてから、ユーザーの変更を有効にするために SupervisorD を再起動します。

  - **Run** は、アセンブルされたソースコードを実行することに関連した OpenShift S2I の概念です。**run** スクリプトは **assemble-and-restart** スクリプトで作成されたアセンブルされたコードを実行します。

  - **s2i-setup** は、**assemble-and-restart** および run スクリプトが正常に実行されるために必要なファイルおよびディレクトリーを作成するスクリプトです。このスクリプトは、アプリケーションのコンテナーが起動されるたびに実行されます。

- ディレクトリー:

  - **language-scripts**: OpenShift S2I はカスタムの **assemble** および **run** スクリプトを許可します。**language-scripts** ディレクトリーにいくつかの言語固有のカスタムスクリプトがあります。カスタムスクリプトは、odo のデバッグを機能させる追加の設定を提供します。

emtpyDir Volume は、Init コンテナーとアプリケーションコンテナーの両方の **/opt/odo** マウントポイントにマウントされます。

2.2.3.1.2. copy-files-to-volume

copy-files-to-volume Init コンテナーは、S2I ビルダーイメージの **/opt/app-root** にあるファイルを永続ボリュームにコピーします。次に、ボリュームはアプリケーションコンテナーの同じ場所 (**/opt/app-root**) にマウントされます。

PersistentVolume が **/opt/app-root** にないと、このディレクトリーのデータは、**PersistentVolumeClaim** が同じ場所にマウントされる際に失われます。

PVC は、Init コンテナー内の **/mnt** マウントポイントにマウントされます。

2.2.3.2. アプリケーションコンテナー

アプリケーションコンテナーは、ユーザーソースコードが実行されるメインコンテナーです。

アプリケーションコンテナーは、以下の 2 つのボリュームでマウントされます。

- **emptyDir** ボリュームは **/opt/odo** にマウントされます。

- **PersistentVolume** は **/opt/app-root** にマウントされます。

go-init はアプリケーションコンテナー内の最初のプロセスとして実行されます。次に、**go-init** プロセスは **SupervisorD** を起動します。

SupervisorD は、ユーザーのアセンブルされたソースコードを実行し、監視します。ユーザープロセスがクラッシュすると、**SupervisorD** がこれを再起動します。

2.2.3.3. PersistentVolume および PersistentVolumeClaim

PersistentVolumeClaim (**PVC**) は、**PersistentVolume** をプロビジョニングする Kubernetes のボリュームタイプです。**PersistentVolume** のライフサイクルは Pod ライフサイクルとは異なります。**PersistentVolume** のデータは Pod の再起動後も永続します。

copy-files-to-volume Init コンテナーは、必要なファイルを **PersistentVolume** にコピーします。メインアプリケーションコンテナーは、実行時にこれらのファイルを使用します。

PersistentVolume の命名規則は <component-name>-s2idata です。

Container	PVC のマウント先
copy-files-to-volume	**/mnt**
アプリケーションコンテナー	**/opt/app-root**

2.2.3.4. emptyDir ボリューム

emptyDir ボリュームは、Pod がノードに割り当てられている際に作成され、Pod がノードで実行されている限り存在します。コンテナーが再起動または移動すると、**emptyDir** の内容が削除され、Init コンテナーはデータを **emptyDir** に復元します。**emptyDir** の初期状態は空です。

copy-supervisord Init コンテナーは必要なファイルを **emptyDir** ボリュームにコピーします。これらのファイルは、実行時にメインアプリケーションコンテナーによって使用されます。

Container	emptyDir Volume のマウント先
copy-supervisord	**/opt/odo**
アプリケーションコンテナー	**/opt/odo**

2.2.3.5. サービス

サービスは、一連の Pod と通信する方法を抽象化する Kubernetes の概念です。

odo はすべてのアプリケーション Pod についてサービスを作成し、これが通信用にアクセス可能にします。

2.2.4. odo push のワークフロー

このセクションでは、**odo push** ワークフローについて説明します。odo push は必要なすべての OpenShift Dedicated リソースを使って OpenShift Dedicated クラスターにユーザーコードをデプロイします。

1. リソースの作成
   まだ作成されていない場合には、**odo push** は以下の OpenShift Dedicated リソースを作成します。

   - デプロイメント設定 (DC):
     - 2 つの init コンテナー **copy-supervisord** および **copy-files-to-volume** が実行されます。init コンテナーはファイルを **emptyDir** と **PersistentVolume** タイプのボリュームのそれぞれにコピーします。
     - アプリケーションコンテナーが起動します。アプリケーションコンテナーの最初のプロセスは、PID=1 の **go-init** プロセスです。
     - **go-init** プロセスは SupervisorD デーモンを起動します。

     注記

     ユーザーアプリケーションコードはアプリケーションコンテナーにコピーされていないため、**SupervisorD** デーモンは **run** スクリプトを実行しません。

   - サービス
   - シークレット
   - **PersistentVolumeClaim**

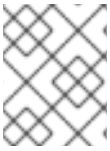
2. ファイルのインデックス設定
   - ファイルインデックサーは、ソースコードディレクトリーのファイルをインデックス化します。インデックサーはソースコードディレクトリー間を再帰的に移動し、作成、削除、または名前が変更されたファイルを検出します。
   - ファイルインデックサーは、**.odo** ディレクトリー内の odo インデックスファイルにインデックス化された情報を維持します。
   - odo インデックスファイルが存在しない場合、ファイルインデックサーの初回の実行時であることを意味し、新規の odo インデックス JSON ファイルが作成されます。odo index JSON ファイルにはファイルマップが含まれます。移動したファイルの相対パスと、変更され、削除されたファイルの絶対パスが含まれます。

3. コードのプッシュ
   ローカルコードは、通常は **/tmp/src** の下にあるアプリケーションコンテナーにコピーされます。

4. **assemble-and-restart** の実行
   ソースコードのコピーに成功すると、**assemble-and-restart** スクリプトは実行中のアプリケーションコンテナー内で実行されます。

2.3. ODO のインストール

以下のセクションでは、各種の異なるプラットフォームに **odo** をインストールする方法を説明します。



注記

現時点では、**odo** はネットワークが制限された環境でのインストールをサポートしていません。

2.3.1. odo の Linux へのインストール

2.3.1.1. バイナリーインストール

```
# curl -L https://mirror.openshift.com/pub/openshift-v4/clients/odo/latest/odo-linux-amd64 -o
/usr/local/bin/odo
# chmod +x /usr/local/bin/odo
```

2.3.1.2. tarball インストール

```
# sh -c 'curl -L https://mirror.openshift.com/pub/openshift-v4/clients/odo/latest/odo-linux-amd64.tar.gz
| gzip -d > /usr/local/bin/odo'
# chmod +x /usr/local/bin/odo
```

2.3.2. odo の Windows へののインストール

2.3.2.1. バイナリーインストール

1. 最新の **odo.exe** ファイルをダウンロードします。
2. **odo.exe** の場所を **GOPATH/bin** ディレクトリーに追加します。

Windows 7/8 の PATH 変数の設定
以下の例は、パス変数の設定方法を示しています。バイナリーは任意の場所に配置することができますが、この例では C:\go-bin を場所に使用します。

1. **C:\go-bin**にフォルダーを作成します。
2. **Start** を右クリックし、**Control Panel** をクリックします。
3. **System and Security** を選択してから **System** をクリックします。
4. 左側のメニューから、**Advanced systems settings** を選択し、下部にある **Environment Variables** ボタンをクリックします。
5. **Variable** セクションから **Path** を選択し、**Edit** をクリックします。
6. **New** をクリックしてフィールドに **C:\go-bin** を入力するか、または **Browse** をクリックしてディレクトリーを選択してから **OK** をクリックします。

Windows 10 の PATH 変数の設定
検索機能を使用して**環境変数**を編集します。

1. **Search** クリックして、**env** または **environment** を入力します。
2. **Edit environment variables for your account**を選択します。
3. **Variable** セクションから **Path** を選択し、**Edit** をクリックします。

4. **New** をクリックしてフィールドに **C:\go-bin** を入力するか、または **Browse** をクリックしてディレクトリーを選択してから **OK** をクリックします。

2.3.3. odo の macOS へのインストール

2.3.3.1. バイナリーインストール

```
# curl -L https://mirror.openshift.com/pub/openshift-v4/clients/odo/latest/odo-darwin-amd64 -o
/usr/local/bin/odo
# chmod +x /usr/local/bin/odo
```

2.3.3.2. tarball インストール

```
# sh -c 'curl -L https://mirror.openshift.com/pub/openshift-v4/clients/odo/latest/odo-darwin-
amd64.tar.gz | gzip -d > /usr/local/bin/odo'
# chmod +x /usr/local/bin/odo
```

2.4. 制限された環境での ODO の使用

2.4.1. 制限された環境での odo について

odo を非接続の OpenShift Dedicated クラスター、または制限された環境でプロビジョニングされたクラスターで実行するには、クラスター管理者がミラーリングされたレジストリーでクラスターを作成していることを確認する必要があります。

非接続クラスターで作業を開始するには、まず **odo init** イメージをクラスターのレジストリーにプッシュし、**ODO_BOOTSTRAPPER_IMAGE** 環境変数を使用して **odo init** イメージパスを上書きする必要があります。

odo init image のプッシュ後に、レジストリーからサポートされているビルダーイメージをミラーリングし、ミラーレジストリーを上書きした後にアプリケーションを作成する必要があります。ビルダーイメージは、アプリケーションのランタイム環境を設定するために必要であり、これにはアプリケーションのビルドに必要なビルドツールが含まれます (例: Node.js の場合は npm、Java の場合は Maven)。ミラーレジストリーには、アプリケーションに必要なすべての依存関係が含まれます。

追加リソース

- レジストリーへのアクセス

2.4.2. odo init イメージの制限されたクラスターレジストリーへのプッシュ

クラスターおよびオペレーティングシステムの設定に応じて、**odo init** イメージをミラーレジストリーにプッシュするか、または内部レジストリーに直接プッシュできます。

前提条件

- クライアントオペレーティングシステムに **oc** をインストールします。
- **odo** をクライアントオペレーティングシステムにインストールします。
- 内部レジストリーまたはミラーレジストリーが設定された OpenShift Dedicated の制限付きクラスターへのアクセス。

2.4.2.1. odo init イメージのミラーレジストリーへのプッシュ

オペレーティングシステムによっては、以下のように **odo init** イメージをミラーレジストリーを持つクラスターにプッシュできます。

2.4.2.1.1. init イメージを Linux のミラーレジストリーにプッシュする

手順

1. **base64** を使用してミラーレジストリーのルート認証局 (CA) コンテンツをエンコードします。

```
$ echo <content_of_additional_ca> | base64 -d > disconnect-ca.crt
```

2. エンコーディングされたルート CA 証明書を適切な場所にコピーします。

```
$ sudo cp ./disconnect-ca.crt /etc/pki/ca-trust/source/anchors/<mirror-registry>.crt
```

3. クライアントプラットフォームで CA を信頼し、OpenShift Dedicated ミラーレジストリーにログインします。

```
$ sudo update-ca-trust enable && sudo systemctl daemon-reload && sudo systemctl restart /
docker && docker login <mirror-registry>:5000 -u <username> -p <password>
```

4. **odo init** イメージをミラーリングします。

```
$ oc image mirror registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
<mirror-registry>:5000/openshiftdo/odo-init-image-rhel7:<tag>
```

5. **ODO_BOOTSTRAPPER_IMAGE** 環境変数を設定してデフォルトの **odo init** イメージパスを上書きします。

```
$ export ODO_BOOTSTRAPPER_IMAGE=<mirror-registry>:5000/openshiftdo/odo-init-
image-rhel7:<tag>
```

2.4.2.1.2. init イメージを MacOS のミラーレジストリーにプッシュする

手順

1. **base64** を使用してミラーレジストリーのルート認証局 (CA) コンテンツをエンコードします。

```
$ echo <content_of_additional_ca> | base64 -d > disconnect-ca.crt
```

2. エンコーディングされたルート CA 証明書を適切な場所にコピーします。
   a. Docker UI を使用して Docker を再起動します。
   b. 以下のコマンドを実行します。

```
$ docker login <mirror-registry>:5000 -u <username> -p <password>
```

3. **odo init** イメージをミラーリングします。

```
$ oc image mirror registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
<mirror-registry>:5000/openshiftdo/odo-init-image-rhel7:<tag>
```

4. **ODO_BOOTSTRAPPER_IMAGE** 環境変数を設定してデフォルトの **odo init** イメージパスを上書きします。

```
$ export ODO_BOOTSTRAPPER_IMAGE=<mirror-registry>:5000/openshiftdo/odo-init-
image-rhel7:<tag>
```

2.4.2.1.3. Windows のミラーレジストリーに init イメージをプッシュする

手順

1. **base64** を使用してミラーレジストリーのルート認証局 (CA) コンテンツをエンコードします。

```
PS C:\> echo <content_of_additional_ca> | base64 -d > disconnect-ca.crt
```

2. 管理者として、以下のコマンドを実行して、エンコーディングされたルート CA 証明書を適切な場所にコピーします。

```
PS C:\WINDOWS\system32> certutil -addstore -f "ROOT" disconnect-ca.crt
```

3. クライアントプラットフォームで CA を信頼し、OpenShift Dedicated ミラーレジストリーにログインします。

   a. Docker UI を使用して Docker を再起動します。

   b. 以下のコマンドを実行します。

```
PS C:\WINDOWS\system32> docker login <mirror-registry>:5000 -u <username> -p
<password>
```

4. **odo init** イメージをミラーリングします。

```
PS C:\> oc image mirror registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
<mirror-registry>:5000/openshiftdo/odo-init-image-rhel7:<tag>
```

5. **ODO_BOOTSTRAPPER_IMAGE** 環境変数を設定してデフォルトの **odo init** イメージパスを上書きします。

```
PS C:\> $env:ODO_BOOTSTRAPPER_IMAGE="<mirror-registry>:5000/openshiftdo/odo-
init-image-rhel7:<tag>"
```

2.4.2.2. odo init イメージを内部レジストリーに直接プッシュする

クラスターでイメージを内部レジストリーに直接プッシュできる場合、以下のように **odo init** イメージをレジストリーにプッシュします。

2.4.2.2.1. init イメージを Linux 上で直接プッシュする

手順

1. デフォルトのルートを有効にします。

```
$ oc patch configs.imageregistry.operator.openshift.io cluster -p '{"spec":
{"defaultRoute":true}}' --type='merge' -n openshift-image-registry
```

2. ワイルドカードルート CA を取得します。

```
$ oc get secret router-certs-default -n openshift-ingress -o yaml
apiVersion: v1
data:
  tls.crt: **************************
  tls.key: ##################
kind: Secret
metadata:
  [...]
type: kubernetes.io/tls
```

3. **base64** を使用してミラーレジストリーのルート認証局 (CA) コンテンツをエンコードします。

```
$ echo <tls.crt> | base64 -d > ca.crt
```

4. クライアントプラットフォームで CA を信頼します。

```
$ sudo cp ca.crt  /etc/pki/ca-trust/source/anchors/externalroute.crt && sudo update-ca-trust
enable && sudo systemctl daemon-reload && sudo systemctl restart docker
```

5. 内部レジストリーにログインします。

```
$ oc get route -n openshift-image-registry
NAME      HOST/PORT    PATH   SERVICES     PORT  TERMINATION   WILDCARD
default-route   <registry_path>          image-registry   <all>   reencrypt     None

$ docker login <registry_path> -u kubeadmin -p $(oc whoami -t)
```

6. **odo init** イメージをプッシュします。

```
$ docker pull registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>

$ docker tag registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
<registry_path>/openshiftdo/odo-init-image-rhel7:<tag>

$ docker push <registry_path>/openshiftdo/odo-init-image-rhel7:<tag>
```

7. **ODO_BOOTSTRAPPER_IMAGE** 環境変数を設定してデフォルトの **odo init** イメージパスを上書きします。

```
$ export ODO_BOOTSTRAPPER_IMAGE=<registry_path>/openshiftdo/odo-init-image-
rhel7:1.0.1
```

2.4.2.2.2. init イメージを MacOS 上で直接プッシュする

手順

1. デフォルトのルートを有効にします。

```
$ oc patch configs.imageregistry.operator.openshift.io cluster -p '{"spec":
{"defaultRoute":true}}' --type='merge' -n openshift-image-registry
```

2. ワイルドカードルート CA を取得します。

```
$ oc get secret router-certs-default -n openshift-ingress -o yaml
apiVersion: v1
data:
  tls.crt: **************************
  tls.key: ##################
kind: Secret
metadata:
  [...]
type: kubernetes.io/tls
```

3. **base64** を使用してミラーレジストリーのルート認証局 (CA) コンテンツをエンコードします。

```
$ echo <tls.crt> | base64 -d > ca.crt
```

4. クライアントプラットフォームで CA を信頼します。

```
$ sudo security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain ca.crt
```

5. 内部レジストリーにログインします。

```
$ oc get route -n openshift-image-registry
NAME      HOST/PORT    PATH   SERVICES     PORT  TERMINATION   WILDCARD
default-route   <registry_path>          image-registry   <all>   reencrypt     None

$ docker login <registry_path> -u kubeadmin -p $(oc whoami -t)
```

6. **odo init** イメージをプッシュします。

```
$ docker pull registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>

$ docker tag registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
<registry_path>/openshiftdo/odo-init-image-rhel7:<tag>

$ docker push <registry_path>/openshiftdo/odo-init-image-rhel7:<tag>
```

7. **ODO_BOOTSTRAPPER_IMAGE** 環境変数を設定してデフォルトの **odo init** イメージパスを上書きします。

```
$ export ODO_BOOTSTRAPPER_IMAGE=<registry_path>/openshiftdo/odo-init-image-
rhel7:1.0.1
```

2.4.2.2.3. initイメージを Windows 上で直接プッシュする

手順

1. デフォルトのルートを有効にします。

```
PS C:\> oc patch configs.imageregistry.operator.openshift.io cluster -p '{"spec":
{"defaultRoute":true}}' --type='merge' -n openshift-image-registry
```

2. ワイルドカードルート CA を取得します。

```
PS C:\> oc get secret router-certs-default -n openshift-ingress -o yaml
apiVersion: v1
data:
  tls.crt: **************************
  tls.key: ##################
kind: Secret
metadata:
  [...]
type: kubernetes.io/tls
```

3. **base64** を使用してミラーレジストリーのルート認証局 (CA) コンテンツをエンコードします。

```
PS C:\> echo <tls.crt> | base64 -d > ca.crt
```

4. 管理者として、以下のコマンドを実行して、クライアントプラットフォームの CA を信頼します。

```
PS C:\WINDOWS\system32> certutil -addstore -f "ROOT" ca.crt
```

5. 内部レジストリーにログインします。

```
PS C:\> oc get route -n openshift-image-registry
NAME      HOST/PORT    PATH   SERVICES     PORT  TERMINATION   WILDCARD
default-route   <registry_path>          image-registry   <all>   reencrypt     None

PS C:\> docker login <registry_path> -u kubeadmin -p $(oc whoami -t)
```

6. **odo init** イメージをプッシュします。

```
PS C:\> docker pull registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>

PS C:\> docker tag registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
<registry_path>/openshiftdo/odo-init-image-rhel7:<tag>

PS C:\> docker push <registry_path>/openshiftdo/odo-init-image-rhel7:<tag>
```

7. **ODO_BOOTSTRAPPER_IMAGE** 環境変数を設定してデフォルトの **odo init** イメージパスを上書きします。

```
PS C:\> $env:ODO_BOOTSTRAPPER_IMAGE="<registry_path>/openshiftdo/odo-init-
image-rhel7:<tag>"
```

2.4.3. コンポーネントの作成および非接続クラスターへのデプロイ

ミラーリングされたレジストリーを持つクラスターに **init** イメージをプッシュした後に、アプリケーションでサポートされるビルダーイメージを **oc** ツールでミラーリングし、環境変数を使用してミラーレジストリーを上書きし、コンポーネントを作成する必要があります。

前提条件

- クライアントオペレーティングシステムに **oc** をインストールします。
- **odo** をクライアントオペレーティングシステムにインストールします。
- 内部レジストリーまたはミラーレジストリーが設定された OpenShift Dedicated の制限付きクラスターへのアクセス。
- **odo init** イメージをクラスターレジストリーにプッシュします。

2.4.3.1. サポートされるビルダーイメージのミラーリング

Node.js の依存関係に npm パッケージを使用し、Java の依存関係に Maven パッケージを使用し、アプリケーションのランタイム環境を設定するには、ミラーレジストリーから適切なビルダーイメージをミラーリングする必要があります。

手順

1. 必要なイメージタグがインポートされていないことを確認します。

```
$ oc describe is nodejs -n openshift
Name:                   nodejs
Namespace:              openshift
[...]

10
  tagged from <mirror-registry>:<port>/rhoar-nodejs/nodejs-10
    prefer registry pullthrough when referencing this tag

  Build and run Node.js 10 applications on RHEL 7. For more information about using this
builder image, including OpenShift considerations, see https://github.com/nodeshift/centos7-
s2i-nodejs.
  Tags: builder, nodejs, hidden
  Example Repo: https://github.com/sclorg/nodejs-ex.git

  ! error: Import failed (NotFound): dockerimage.image.openshift.io "<mirror-registry>:
<port>/rhoar-nodejs/nodejs-10:latest" not found
      About an hour ago

10-SCL (latest)
  tagged from <mirror-registry>:<port>/rhscl/nodejs-10-rhel7
    prefer registry pullthrough when referencing this tag

  Build and run Node.js 10 applications on RHEL 7. For more information about using this
builder image, including OpenShift considerations, see https://github.com/nodeshift/centos7-
s2i-nodejs.
  Tags: builder, nodejs
  Example Repo: https://github.com/sclorg/nodejs-ex.git

  ! error: Import failed (NotFound): dockerimage.image.openshift.io "<mirror-registry>:
<port>/rhscl/nodejs-10-rhel7:latest" not found
```

```
      About an hour ago

[...]
```

2. サポートされるイメージタグをプライベートレジストリーに対してミラーリングします。

```
$ oc image mirror registry.access.redhat.com/rhscl/nodejs-10-rhel7:<tag>
<private_registry>/rhscl/nodejs-10-rhel7:<tag>
```

3. イメージをインポートします。

```
$ oc tag <mirror-registry>:<port>/rhscl/nodejs-10-rhel7:<tag> nodejs-10-rhel7:latest --
scheduled
```

イメージを定期的に再インポートする必要があります。**--scheduled** フラグは、イメージの自動再インポートを有効にします。

4. 指定されたタグを持つイメージがインポートされていることを確認します。

```
$ oc describe is nodejs -n openshift
Name:                   nodejs
[...]
10-SCL (latest)
  tagged from <mirror-registry>:<port>/rhscl/nodejs-10-rhel7
    prefer registry pullthrough when referencing this tag

  Build and run Node.js 10 applications on RHEL 7. For more information about using this
builder image, including OpenShift considerations, see https://github.com/nodeshift/centos7-
s2i-nodejs.
  Tags: builder, nodejs
  Example Repo: https://github.com/sclorg/nodejs-ex.git

  * <mirror-registry>:<port>/rhscl/nodejs-10-
rhel7@sha256:d669ecbc11ac88293de50219dae8619832c6a0f5b04883b480e073590fab7c54

      3 minutes ago

[...]
```

2.4.3.2. ミラーレジストリーの上書き

Node.js の依存関係用の npm パッケージおよび Java の依存関係用の Maven パッケージをプライベートミラーレジストリーからダウンロードするには、クラスター上にミラー npm または Maven レジストリーを作成し、設定する必要があります。その後、既存のコンポーネントで、または新規コンポーネントの作成時にミラーレジストリーを上書きできます。

手順

- 既存のコンポーネントでミラーレジストリーを上書きするには、以下を実行します。

```
$ odo config set --env NPM_MIRROR=<npm_mirror_registry>
```

- コンポーネントの作成時にミラーレジストリーを上書きするには、以下を実行します。

```
$ odo component create nodejs --env NPM_MIRROR=<npm_mirror_registry>
```

2.4.3.3. odo を使用した Node.js アプリケーションの作成

Node.js コンポーネントを作成するには、Node.js アプリケーションをダウンロードし、**odo**でソースコードをクラスターにプッシュします。

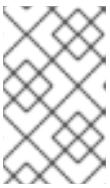
手順

1. 現在のディレクトリーをアプリケーションのあるディレクトリーに切り替えます。

```
$ cd <directory name>
```

2. Node.js タイプのコンポーネントをアプリケーションに追加します。

```
$ odo create nodejs
```



注記

デフォルトで、最新イメージが使用されます。また、**odo create openshift/nodejs:8** を使用してイメージのバージョンを明示的に指定できます。

3. 初期ソースコードをコンポーネントにプッシュします。

```
$ odo push
```

これで、コンポーネントは OpenShift Dedicated にデプロイされます。

4. URL を作成し、以下のようにローカル設定ファイルにエントリーを追加します。

```
$ odo url create --port 8080
```

5. 変更をプッシュします。これにより、URL がクラスターに作成されます。

```
$ odo push
```

6. コンポーネントに必要な URL を確認するために URL を一覧表示します。

```
$ odo url list
```

7. 生成された URL を使用してデプロイされたアプリケーションを表示します。

```
$ curl <URL>
```

2.5. ODO を使用した単一コンポーネントアプリケーションの作成

odo を使用すると、OpenShift Dedicated クラスターでアプリケーションを作成し、デプロイできます。

前提条件

- **odo** がインストールされている。
- OpenShift Dedicated クラスターが実行中であること。CodeReady Containers (CRC) を使用して、OpenShift Dedicated のローカルクラスターを迅速にデプロイできます。

2.5.1. プロジェクトの作成

プロジェクトを作成し、別個の単一の単位で編成されるソースコード、テスト、ライブラリーを維持します。

手順

1. OpenShift Dedicated クラスターにログインします。

```
$ odo login -u developer -p developer
```

2. プロジェクトを作成します。

```
$ odo project create myproject
 ✓  Project 'myproject' is ready for use
 ✓  New project created and now using project : myproject
```

2.5.2. odo を使用した Node.js アプリケーションの作成

Node.js コンポーネントを作成するには、Node.js アプリケーションをダウンロードし、**odo**でソースコードをクラスターにプッシュします。

手順

1. コンポーネントの新規ディレクトリーを作成します。

```
$ mkdir my_components $$ cd my_components
```

2. Node.js アプリケーションのサンプルをダウンロードします。

```
$ git clone https://github.com/openshift/nodejs-ex
```

3. 現在のディレクトリーをアプリケーションのあるディレクトリーに切り替えます。

```
$ cd <directory name>
```

4. Node.js タイプのコンポーネントをアプリケーションに追加します。

```
$ odo create nodejs
```

注記

デフォルトで、最新イメージが使用されます。また、**odo create openshift/nodejs:8** を使用してイメージのバージョンを明示的に指定できます。

5. 初期ソースコードをコンポーネントにプッシュします。

```
$ odo push
```

これで、コンポーネントは OpenShift Dedicated にデプロイされます。

6. URL を作成し、以下のようにローカル設定ファイルにエントリーを追加します。

```
$ odo url create --port 8080
```

7. 変更をプッシュします。これにより、URL がクラスターに作成されます。

```
$ odo push
```

8. コンポーネントに必要な URL を確認するために URL を一覧表示します。

```
$ odo url list
```

9. 生成された URL を使用してデプロイされたアプリケーションを表示します。

```
$ curl <URL>
```

2.5.3. アプリケーションコードの変更

アプリケーションコードを変更し、それらの変更を OpenShift Dedicated のアプリケーションに適用します。

1. 選択するテキストエディターで、Node.js ディレクトリー内のレイアウトファイルのいずれかを編集します。
2. コンポーネントを更新します。

```
$ odo push
```

3. ブラウザーでアプリケーションを更新し、変更を確認します。

2.5.4. ストレージのアプリケーションコンポーネントへの追加

永続ストレージは、odo を再起動してもデータを利用可能な状態に維持します。**odo storage** コマンドを使用して、ストレージをコンポーネントに追加できます。

手順

- ストレージをコンポーネントに追加します。

```
$ odo storage create nodestorage --path=/opt/app-root/src/storage/ --size=1Gi
```

コンポーネントには 1GB のストレージがあります。

2.5.5. ビルドイメージを指定するためのカスタムビルダーの追加

OpenShift Dedicated では、カスタムイメージの作成ごとに発生する差を埋めるカスタムイメージを追加できます。

以下の例は、**redhat-openjdk-18** イメージの正常なインポートおよび使用方法について示しています。

前提条件

- OpenShift CLI (oc) がインストールされている。

手順

1. イメージを OpenShift Dedicated にインポートします。

```
$ oc import-image openjdk18 \
--from=registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift \
--confirm
```

2. イメージにタグを付け、odo からアクセスできるようにします。

```
$ oc annotate istag/openjdk18:latest tags=builder
```

3. odo でイメージをデプロイします。

```
$ odo create openjdk18 --git \
https://github.com/openshift-evangelists/Wild-West-Backend
```

2.5.6. OpenShift Service Catalog を使用したアプリケーションの複数サービスへの接続

OpenShift サービスカタログは、Kubernetes 用の Open Service Broker API (OSB API) の実装です。これを使用して、OpenShift Dedicated にデプロイされているアプリケーションをさまざまなサービスに接続できます。

前提条件

- OpenShift Dedicated クラスターが実行中であること。
- サービスカタログがクラスターにインストールされ、有効にされている。

手順

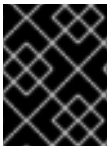
- サービスを一覧表示するには、以下を使用します。

```
$ odo catalog list services
```

- サービスカタログ関連の操作を使用するには、以下を実行します。

```
$ odo service <verb> <servicename>
```

2.5.7. アプリケーションの削除



重要

アプリケーションを削除すると、アプリケーションに関連付けられたすべてのコンポーネントが削除されます。

手順

1. 現在のプロジェクトのアプリケーションを一覧表示します。

```
$ odo app list
  The project '<project_name>' has the following applications:
  NAME
  app
```

2. アプリケーションに関連付けられたコンポーネントを一覧表示します。これらのコンポーネントはアプリケーションと共に削除されます。

```
$ odo component list
  APP     NAME                    TYPE       SOURCE       STATE
  app     nodejs-nodejs-ex-elyf     nodejs     file:///.     Pushed
```

3. アプリケーションを削除します。

```
$ odo app delete <application_name>
  ? Are you sure you want to delete the application: <application_name> from project:
<project_name>
```

4. **Y** で削除を確定します。**-f** フラグを使用すると、確認プロンプトを非表示にできます。

2.6. ODO を使用したマルチコンポーネントアプリケーションの作成

odo を使用すると、簡単かつ自動化された方法でマルチコンポーネントアプリケーションを作成し、変更し、そのコンポーネントをリンクすることができます。

この例では、マルチコンポーネントアプリケーション (シューティングゲーム) をデプロイする方法について説明します。アプリケーションはフロントエンド Node.js コンポーネントとバックエンド Java コンポーネントで構成されます。

前提条件

- **odo** がインストールされている。
- OpenShift Dedicated クラスターが実行中であること。開発者は CodeReady Containers (CRC) を使用して、OpenShift Dedicated のローカルクラスターを迅速にデプロイできます。
- Maven がインストールされている。

2.6.1. プロジェクトの作成

プロジェクトを作成し、別個の単一の単位で編成されるソースコード、テスト、ライブラリーを維持します。

手順

1. OpenShift Dedicated クラスターにログインします。

```
$ odo login -u developer -p developer
```

2. プロジェクトを作成します。

```
$ odo project create myproject
 ✓  Project 'myproject' is ready for use
 ✓  New project created and now using project : myproject
```

2.6.2. バックエンドコンポーネントのデプロイ

Java コンポーネントを作成するには、Java ビルダーイメージをインポートし、Java アプリケーションをダウンロードし、**odo** でソースコードをクラスターにプッシュします。

手順

1. **openjdk18** をクラスターにインポートします。

```
$ oc import-image openjdk18 \
--from=registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift --confirm
```

2. イメージに **builder** のタグを付け、イメージが odo でアクセスできるようにします。

```
$ oc annotate istag/openjdk18:latest tags=builder
```

3. **odo catalog list components** を実行し、作成されたイメージを表示します。

```
$ odo catalog list components
Odo Supported OpenShift Components:
NAME          PROJECT       TAGS
nodejs        openshift     10,8,8-RHOAR,latest
openjdk18     myproject     latest
```

4. コンポーネントの新規ディレクトリーを作成します。

```
$ mkdir my_components $$ cd my_components
```

5. バックエンドアプリケーションのサンプルをダウンロードします。

```
$ git clone https://github.com/openshift-evangelists/Wild-West-Backend backend
```

6. ディレクトリーをバックエンドソースディレクトリーに切り替え、そのディレクトリーに正しいファイルが含まれることを確認します。

```
$ cd backend
$ ls
debug.sh  pom.xml  src
```

7. バックエンドのソースファイルを Maven でビルドし、JAR ファイルを作成します。

```
$ mvn package
```

```
...
[INFO] ------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------
[INFO] Total time: 2.635 s
[INFO] Finished at: 2019-09-30T16:11:11-04:00
[INFO] Final Memory: 30M/91M
[INFO] ------------------------------------
```

8. **backend** という Java コンポーネントタイプのコンポーネント設定を作成します。

```
$ odo create openjdk18 backend --binary target/wildwest-1.0.jar
 ✓  Validating component [1ms]
Please use `odo push` command to create the component with source deployed
```

設定ファイルの **config.yaml** は、デプロイ用のコンポーネントについての情報が含まれるバックエンドコンポーネントのローカルディレクトリーに置かれます。

9. 以下を使用して **config.yaml** ファイルでバックエンドコンポーネントの設定内容を確認します。

```
$ odo config view
COMPONENT SETTINGS
------------------------------------------------
PARAMETER       CURRENT_VALUE
Type            openjdk18
Application     app
Project         myproject
SourceType      binary
Ref
SourceLocation  target/wildwest-1.0.jar
Ports           8080/TCP,8443/TCP,8778/TCP
Name            backend
MinMemory
MaxMemory
DebugPort
Ignore
MinCPU
MaxCPU
```

10. コンポーネントを OpenShift Dedicated クラスターにプッシュします。

```
$ odo push
Validation
 ✓  Checking component [6ms]

Configuration changes
 ✓  Initializing component
 ✓  Creating component [124ms]

Pushing to component backend of type binary
 ✓  Checking files for pushing [1ms]
 ✓  Waiting for component to start [48s]
 ✓  Syncing files to the component [811ms]
 ✓  Building component [3s]
```

odo push を使用すると、OpenShift Dedicated はバックエンドコンポーネントをホストするためのコンテナーを作成し、そのコンテナーを OpenShift Dedicated クラスターで実行されている Pod にデプロイし、 **backend** コンポーネントを起動します。

11. 以下を検証します。

    - odo でのアクションのステータス

```
odo log -f
2019-09-30 20:14:19.738  INFO 444 --- [           main] c.o.wildwest.WildWestApplication
: Starting WildWestApplication v1.0 onbackend-app-1-9tnhc with PID 444
(/deployments/wildwest-1.0.jar started by jboss in /deployments)
```

    - バックエンドコンポーネントのステータス

```
$ odo list
APP     NAME        TYPE          SOURCE                          STATE
app     backend     openjdk18     file://target/wildwest-1.0.jar     Pushed
```

2.6.3. フロントエンドコンポーネントのデプロイ

フロントエンドコンポーネントを作成およびデプロイするには、Node.js アプリケーションをダウンロードし、ソースコードを **odo**でクラスターにプッシュします。

手順

1. フロントエンドアプリケーションのサンプルをダウンロードします。

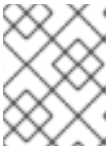
```
$ git clone https://github.com/openshift/nodejs-ex
```

2. 現在のディレクトリーをフロントエンドディレクトリーに切り替えます。

```
$ cd <directory-name>
```

3. フロントエンドが Node.js アプリケーションであることを確認するために、ディレクトリーの内容を一覧表示します。

```
$ ls
assets  bin  index.html  kwww-frontend.iml  package.json  package-lock.json  playfield.png
README.md  server.js
```



注記

フロントエンドコンポーネントはインタプリター型言語で記述され (Node.js)、ビルドされる必要はありません。

4. **frontend**という名前の Node.js コンポーネントタイプのコンポーネント設定を作成します。

```
$ odo create nodejs frontend
 ✓  Validating component [5ms]
Please use `odo push` command to create the component with source deployed
```

5. コンポーネントを実行中のコンテナーにプッシュします。

```
$ odo push
Validation
 ✓  Checking component [8ms]

Configuration changes
 ✓  Initializing component
 ✓  Creating component [83ms]

Pushing to component frontend of type local
 ✓  Checking files for pushing [2ms]
 ✓  Waiting for component to start [45s]
 ✓  Syncing files to the component [3s]
 ✓  Building component [18s]
 ✓  Changes successfully pushed to component
```

2.6.4. 2 つのコンポーネントのリンク

クラスターで実行されるコンポーネントは、対話するために接続される必要があります。OpenShift Dedicated は、リンクの仕組みを提供し、プログラムからクライアントへの通信バインディングを公開します。

手順

1. クラスターで実行されるすべてのコンポーネントの一覧を表示します。

```
$ odo list
APP     NAME         TYPE          SOURCE                          STATE
app     backend      openjdk18     file://target/wildwest-1.0.jar     Pushed
app     frontend     nodejs        file://./                       Pushed
```

2. 現在のフロントエンドコンポーネントをバックエンドにリンクします。

```
$ odo link backend --port 8080
 ✓  Component backend has been successfully linked from the component frontend

Following environment variables were added to frontend component:
- COMPONENT_BACKEND_HOST
- COMPONENT_BACKEND_PORT
```

バックエンドコンポーネントの設定情報がフロントエンドコンポーネントに追加され、フロントエンドコンポーネントが再起動します。

2.6.5. コンポーネントの公開

手順

1. アプリケーションの外部 URL を作成します。

```
$ cd frontend
$ odo url create frontend --port 8080
 ✓  URL frontend created for component: frontend
```

```
To create URL on the OpenShift  cluster, use `odo push`
```

2. 変更を適用します。

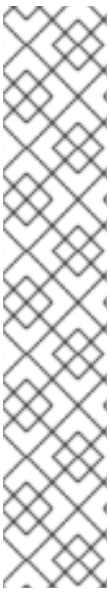
```
$ odo push
Validation
 ✓  Checking component [21ms]

Configuration changes
 ✓  Retrieving component data [35ms]
 ✓  Applying configuration [29ms]

Applying URL changes
 ✓  URL frontend: http://frontend-app-myproject.192.168.42.79.nip.io created

Pushing to component frontend of type local
 ✓  Checking file changes for pushing [1ms]
 ✓  No file changes detected, skipping build. Use the '-f' flag to force the build.
```

3. ブラウザーで URL を開き、アプリケーションを表示します。



注記

アプリケーションに OpenShift Dedicated namespace にアクセスし、アクティブな Pod を削除するのに有効なサービスアカウントのパーミッションが必要な場合、バックエンドコンポーネントから **odo log** を参照すると以下のエラーが発生する場合があります。

Message: Forbidden!Configured service account doesn't have access.Service account may have been revoked

このエラーを解決するには、サービスアカウントロールのパーミッションを追加します。

```
$ oc policy add-role-to-group view system:serviceaccounts -n <project>
$ oc policy add-role-to-group edit system:serviceaccounts -n <project>
```

これは実稼働クラスターでは実行しないでください。

2.6.6. 実行中のアプリケーションの変更

手順

1. ローカルディレクトリーをフロントエンドディレクトリーに切り替えます。

```
$ cd ~/frontend
```

2. 以下のコマンドを実行して、ファイルシステムで変更を監視します。

```
$ odo watch
```

3. **index.html** ファイルを編集して、ゲームの表示される名前を変更します。

注記

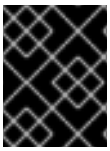
odo が変更を認識するまでに若干の遅延が発生する場合があります。

odo は変更をフロントエンドコンポーネントにプッシュし、そのステータスをターミナルに印刷します。

```
File /root/frontend/index.html changed
File  changed
Pushing files...
 ✓  Waiting for component to start
 ✓  Copying files to component
 ✓  Building component
```

4. Web ブラウザーでアプリケーションページを更新します。これで新しい名前が表示されます。

2.6.7. アプリケーションの削除



重要

アプリケーションを削除すると、アプリケーションに関連付けられたすべてのコンポーネントが削除されます。

手順

1. 現在のプロジェクトのアプリケーションを一覧表示します。

```
$ odo app list
   The project '<project_name>' has the following applications:
   NAME
   app
```

2. アプリケーションに関連付けられたコンポーネントを一覧表示します。これらのコンポーネントはアプリケーションと共に削除されます。

```
$ odo component list
   APP     NAME                    TYPE       SOURCE       STATE
   app     nodejs-nodejs-ex-elyf     nodejs     file://./     Pushed
```

3. アプリケーションを削除します。

```
$ odo app delete <application_name>
   ? Are you sure you want to delete the application: <application_name> from project:
<project_name>
```

4. **Y** で削除を確定します。**-f** フラグを使用すると、確認プロンプトを非表示にできます。

2.7. データベースと共にアプリケーションを作成する

以下の例では、データベースをフロントエンドアプリケーションにデプロイし、接続する方法を説明します。

前提条件

前提条件

- **odo** がインストールされている。
- **oc** クライアントがインストールされている。
- OpenShift Dedicated クラスターが実行中であること。開発者は CodeReady Containers (CRC) を使用して、OpenShift Dedicated のローカルクラスターを迅速にデプロイできます。
- サービスカタログが有効にされている。

2.7.1. プロジェクトの作成

プロジェクトを作成し、別個の単一の単位で編成されるソースコード、テスト、ライブラリーを維持します。

手順

1. OpenShift Dedicated クラスターにログインします。

```
$ odo login -u developer -p developer
```

2. プロジェクトを作成します。

```
$ odo project create myproject
 ✓  Project 'myproject' is ready for use
 ✓  New project created and now using project : myproject
```

2.7.2. フロントエンドコンポーネントのデプロイ

フロントエンドコンポーネントを作成およびデプロイするには、Node.js アプリケーションをダウンロードし、ソースコードを **odo**でクラスターにプッシュします。

手順

1. フロントエンドアプリケーションのサンプルをダウンロードします。

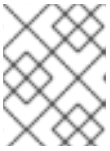
```
$ git clone https://github.com/openshift/nodejs-ex
```

2. 現在のディレクトリーをフロントエンドディレクトリーに切り替えます。

```
$ cd <directory-name>
```

3. フロントエンドが Node.js アプリケーションであることを確認するために、ディレクトリーの内容を一覧表示します。

```
$ ls
assets  bin  index.html  kwww-frontend.iml  package.json  package-lock.json  playfield.png
README.md  server.js
```



注記

フロントエンドコンポーネントはインタプリター型言語で記述され (Node.js)、ビルドされる必要はありません。

4. **frontend**という名前の Node.js コンポーネントタイプのコンポーネント設定を作成します。

```
$ odo create nodejs frontend
 ✓  Validating component [5ms]
Please use `odo push` command to create the component with source deployed
```

5. フロントエンドインターフェースにアクセスするための URL を作成します。

```
$ odo url create myurl
 ✓  URL myurl created for component: nodejs-nodejs-ex-pmdp
```

6. コンポーネントを OpenShift Dedicated クラスターにプッシュします。

```
$ odo push
Validation
 ✓  Checking component [7ms]

Configuration changes
 ✓  Initializing component
 ✓  Creating component [134ms]

Applying URL changes
 ✓  URL myurl: http://myurl-app-myproject.192.168.42.79.nip.io created

Pushing to component nodejs-nodejs-ex-mhbb of type local
 ✓  Checking files for pushing [657850ns]
 ✓  Waiting for component to start [6s]
 ✓  Syncing files to the component [408ms]
 ✓  Building component [7s]
 ✓  Changes successfully pushed to component
```

2.7.3. 対話モードでデータベースをデプロイする

odo は、デプロイをシンプルにするコマンドラインの対話モードを提供します。

手順

- 対話モードを実行し、プロンプトに対応します。

```
$ odo service create
? Which kind of service do you wish to create database
? Which database service class should we use mongodb-persistent
? Enter a value for string property DATABASE_SERVICE_NAME (Database Service Name):
mongodb
? Enter a value for string property MEMORY_LIMIT (Memory Limit): 512Mi
? Enter a value for string property MONGODB_DATABASE (MongoDB Database Name):
sampledb
? Enter a value for string property MONGODB_VERSION (Version of MongoDB Image): 3.2
? Enter a value for string property VOLUME_CAPACITY (Volume Capacity): 1Gi
```

```
? Provide values for non-required properties No
? How should we name your service  mongodb-persistent
? Output the non-interactive version of the selected options No
? Wait for the service to be ready No
 ✓  Creating service [32ms]
 ✓  Service 'mongodb-persistent' was created
Progress of the provisioning will not be reported and might take a long time.
You can see the current status by executing 'odo service list'
```

注記

パスワードまたはユーザー名がフロントエンドアプリケーションに環境変数として渡されます。

2.7.4. データベースの手動デプロイ

1. 利用可能なサービスを一覧表示します。

```
$ odo catalog list services
NAME                        PLANS
django-psql-persistent      default
jenkins-ephemeral           default
jenkins-pipeline-example    default
mariadb-persistent          default
mongodb-persistent          default
mysql-persistent            default
nodejs-mongo-persistent     default
postgresql-persistent       default
rails-pgsql-persistent      default
```

2. サービスの **mongodb-persistent** タイプを選択し、必要なパラメーターを確認します。

```
$ odo catalog describe service mongodb-persistent
 **********************        | ******************************************************
 Name                          | default
 ----------------              | ----------------
 Display Name                  |
 ----------------              | ----------------
 Short Description             | Default plan
 ----------------              | ----------------
 Required Params without a     |
 default value                 |
 ----------------              | ----------------
 Required Params with a default | DATABASE_SERVICE_NAME
 value                         | (default: 'mongodb'),
                               | MEMORY_LIMIT (default:
                               | '512Mi'), MONGODB_VERSION
                               | (default: '3.2'),
                               | MONGODB_DATABASE (default:
                               | 'sampledb'), VOLUME_CAPACITY
                               | (default: '1Gi')
 ----------------              | ----------------
 Optional Params               | MONGODB_ADMIN_PASSWORD,
                               | NAMESPACE, MONGODB_PASSWORD,
                               | MONGODB_USER
```

3. 必須のパラメーターをフラグとして渡し、データベースのデプロイを待機します。

```
$ odo service create mongodb-persistent --plan default --wait -p
DATABASE_SERVICE_NAME=mongodb -p MEMORY_LIMIT=512Mi -p
MONGODB_DATABASE=sampledb -p VOLUME_CAPACITY=1Gi
```

2.7.5. データベースのフロントエンドアプリケーションへの接続

1. データベースをフロントエンドサービスにリンクします。

```
$ odo link mongodb-persistent
 ✓  Service mongodb-persistent has been successfully linked from the component nodejs-
nodejs-ex-mhbb

Following environment variables were added to nodejs-nodejs-ex-mhbb component:
- database_name
- password
- uri
- username
- admin_password
```

2. Pod のアプリケーションおよびデータベースの環境変数を確認します。

```
$ oc get pods
NAME                              READY     STATUS    RESTARTS   AGE
mongodb-1-gsznc                   1/1       Running   0          28m
nodejs-nodejs-ex-mhbb-app-4-vkn9l 1/1       Running   0          1m

$ oc rsh nodejs-nodejs-ex-mhbb-app-4-vkn9l
sh-4.3$ env
uri=mongodb://172.30.126.3:27017
password=dHIOpYneSkX3rTLn
database_name=sampledb
username=user43U
admin_password=NCn41tqmx7RIqmfv
sh-4.3$
```

3. ブラウザーで URL を開き、右下に表示されるデータベース設定を確認します。

```
$ odo url list
```

```
Request information
Page view count: 24

DB Connection Info:
Type: MongoDB
URL: mongodb://172.30.126.3:27017/sampledb
```

2.7.6. アプリケーションの削除

重要

アプリケーションを削除すると、アプリケーションに関連付けられたすべてのコンポーネントが削除されます。

手順

1. 現在のプロジェクトのアプリケーションを一覧表示します。

```
$ odo app list
   The project '<project_name>' has the following applications:
   NAME
   app
```

2. アプリケーションに関連付けられたコンポーネントを一覧表示します。これらのコンポーネントはアプリケーションと共に削除されます。

```
$ odo component list
   APP     NAME                    TYPE       SOURCE       STATE
   app     nodejs-nodejs-ex-elyf     nodejs     file:///.     Pushed
```

3. アプリケーションを削除します。

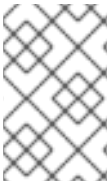
```
$ odo app delete <application_name>
   ? Are you sure you want to delete the application: <application_name> from project:
<project_name>
```

4. **Y** で削除を確定します。**-f** フラグを使用すると、確認プロンプトを非表示にできます。

2.8. サンプルアプリケーションの使用

odo は、OpenShift カタログのコンポーネントタイプ内の言語またはランタイムとの部分的な互換性を提供します。以下は例になります。

```
NAME        PROJECT       TAGS
dotnet      openshift     2.0,latest
httpd       openshift     2.4,latest
java        openshift     8,latest
nginx       openshift     1.10,1.12,1.8,latest
nodejs      openshift     0.10,4,6,8,latest
perl        openshift     5.16,5.20,5.24,latest
php         openshift     5.5,5.6,7.0,7.1,latest
python      openshift     2.7,3.3,3.4,3.5,3.6,latest
ruby        openshift     2.0,2.2,2.3,2.4,latest
wildfly     openshift     10.0,10.1,8.1,9.0,latest
```



注記

odo 1.1.0 については、Java および Node.js は正式にサポートされているコンポーネントタイプです。**odo catalog list components** を実行して、正式にサポートされているコンポーネントタイプを確認します。

Web 経由でコンポーネントにアクセスするには、 **odo url create** を使用して URL を作成します。

2.8.1. Git リポジトリーの例

2.8.1.1. httpd

この例は、CentOS 7 で httpd を使用して静的コンテンツをビルドし、提供するのに役立ちます。OpenShift Dedicated の考慮点を含む、このビルダーイメージの使用方法についての詳細は、「Apache HTTP Server container image repository」を参照してください。

```
$ odo create httpd --git https://github.com/openshift/httpd-ex.git
```

2.8.1.2. java

この例は、CentOS 7 で Fat JAR Java アプリケーションをビルドし、実行するのに役立ちます。OpenShift Dedicated の考慮点を含む、このビルダーイメージを使用する方法についての詳細は、「Java S2I Builder image 」を参照してください。

```
$ odo create java --git https://github.com/spring-projects/spring-petclinic.git
```

2.8.1.3. nodejs

CentOS 7 で Node.js アプリケーションをビルドし、実行します。OpenShift Dedicated の考慮点を含む、このビルダーイメージを使用する方法についての詳細は、「Node.js 8 container image 」を参照してください。

```
$ odo create nodejs --git https://github.com/openshift/nodejs-ex.git
```

2.8.1.4. perl

この例は、CentOS 7 で Perl アプリケーションのビルドし、実行するのに役立ちます。OpenShift Dedicated の考慮点を含む、このビルダーイメージを使用する方法についての詳細は、「Perl 5.26 container image」を参照してください。

```
$ odo create perl --git https://github.com/openshift/dancer-ex.git
```

2.8.1.5. php

この例は、CentOS 7 で PHP アプリケーションのビルドし、実行するのに役立ちます。OpenShift Dedicated の考慮点を含む、このビルダーイメージを使用する方法についての詳細は、「PHP 7.1 Docker image」を参照してください。

```
$ odo create php --git https://github.com/openshift/cakephp-ex.git
```

2.8.1.6. python

この例は、CentOS 7 で Python アプリケーションをビルドし、実行するのに役立ちます。OpenShift Dedicated の考慮点を含む、このビルダーイメージを使用する方法についての詳細は、「Python 3.6 container image」を参照してください。

```
$ odo create python --git https://github.com/openshift/django-ex.git
```

2.8.1.7. ruby

この例は、CentOS 7 で Ruby アプリケーションをビルドし、実行するのに役立ちます。OpenShift Dedicated の考慮点を含む、このビルダーイメージを使用する方法についての詳細は、「Ruby 2.5 container image」を参照してください。

```
$ odo create ruby --git https://github.com/openshift/ruby-ex.git
```

2.8.1.8. wildfly

この例は、CentOS 7 で WildFly アプリケーションをビルドし、実行するのに役立ちます。OpenShift Dedicated の考慮点を含む、このビルダーイメージを使用する方法についての詳細は、「Wildfly - CentOS Docker images for OpenShift」を参照してください。

```
$ odo create wildfly --git https://github.com/openshift/openshift-jee-sample.git
```

2.8.2. バイナリーのサンプル

2.8.2.1. java

Java を使用すると、以下のようにバイナリーアーティファクトをデプロイすることができます。

```
$ git clone https://github.com/spring-projects/spring-petclinic.git
$ cd spring-petclinic
$ mvn package
$ odo create java test3 --binary target/*.jar
$ odo push
```

2.8.2.2. wildfly

WildFly を使用すると、以下のようにバイナリーアプリケーションをデプロイすることができます。

```
$ git clone https://github.com/openshiftdemos/os-sample-java-web.git
$ cd os-sample-java-web
$ mvn package
$ cd ..
$ mkdir example && cd example
$ mv ../os-sample-java-web/target/ROOT.war example.war
$ odo create wildfly --binary example.war
```

2.9. ODO でのアプリケーションのデバッグ

重要

odo でのインタラクティブなデバッグはテクノロジープレビュー機能です。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。Red Hat は実稼働環境でこれらを使用することを推奨していません。これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、お客様は機能性をテストし、開発プロセス中にフィードバックをお寄せいただくことができます。

Red Hat のテクノロジープレビュー機能のサポート範囲についての詳細は、https://access.redhat.com/ja/support/offerings/techpreview/ を参照してください。

odoを使用すると、デバッガーを割り当て、アプリケーションをリモートでデバッグできます。この機能は NodeJS および Java コンポーネントでのみサポートされます。

odo で作成されたコンポーネントは、デフォルトでデバッグモードで実行されます。デバッガーのエージェントは、特定のポートでコンポーネントに対して実行されます。アプリケーションのデバッグを開始するには、ポート転送を開始して、統合開発環境 (IDE) にバンドルされたローカルのデバッガーを割り当てる必要があります。

2.9.1. アプリケーションのデバッグ

odo debug コマンドを使用して、**odo** でアプリケーションをデバッグできます。

手順

1. アプリケーションがデプロイされた後に、コンポーネントのポート転送を開始して、アプリケーションのデバッグを行います。

```
$ odo debug port-forward
```

2. IDE にバンドルされたデバッガーをコンポーネントに割り当てます。手順は、IDE によって異なります。

2.9.2. デバッグパラメーターの設定

odo config コマンドでリモートポートを指定し、**odo debug** コマンドでローカルポートを指定できます。

手順

- デバッグエージェントを実行するリモートポートを設定するには、以下を実行します。

```
$ odo config set DebugPort 9292
```

注記

この値のコンポーネントをコンポーネントに反映させるには、コンポーネントを再デプロイする必要があります。

- ローカルポートをポート転送に設定するには、以下を実行します。

```
$ odo debug port-forward --local-port 9292
```

注記

ローカルポートの値は永続化されません。ポートを変更する必要がある場合は毎回これを指定する必要があります。

2.10. 環境変数の管理

odo はコンポーネント固有の設定および環境変数を **config** ファイルに保存します。**odo config** コマンドを使用すると、**config** ファイルを変更せずに、コンポーネントの環境変数の設定、設定解除、および一覧表示を実行できます。

2.10.1. 環境変数の設定および設定解除

手順

- コンポーネントで環境変数を設定するには、以下を実行します。

```
$ odo config set --env <variable>=<value>
```

- コンポーネントの環境変数の設定を解除するには、以下を実行します。

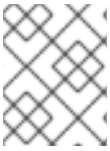
```
$ odo config unset --env <variable>
```

- コンポーネント内のすべての環境変数を一覧表示するには、以下を実行します。

```
$ odo config view
```

2.11. ODO CLI の設定

2.11.1. コマンド補完の使用



注記

現時点で、コマンドの補完は bash、zsh、および fish シェルでのみサポートされています。

odo は、ユーザー入力に基づくコマンドパラメーターのスマート補完を提供します。これを機能させるには、odo は実行中のシェルと統合する必要があります。

手順

- コマンド補完を自動的にインストールするには、以下を実行します。

1. 以下を実行します。

```
$ odo --complete
```

2. 補完フックのインストールを求めるプロンプトが出されたら、**y** を押します。

- 補完フックを手動でインストールするには、**complete -o nospace -C <full path to your odo binary> odo** をシェル設定ファイルに追加します。シェル設定ファイルを変更したら、シェルを再起動します。
- 補完を無効にするには、以下を実行します。

  1. 以下を実行します。

     ```
     $ odo --uncomplete
     ```

  2. 補完フックをアンインストールするようプロンプトされたら **y** を押します。

注記

odo 実行可能ファイルの名前を変更した場合や、これを別のディレクトリーに移動する場合、コマンド補完を再度有効にします。

2.11.2. ファイルまたはパターンを無視する

アプリケーションのルートディレクトリーにある **.odoignore** ファイルを変更して、無視するファイルまたはパターンの一覧を設定できます。これは、**odo push** および **odo watch** の両方に適用されます。

.odoignore ファイルが存在 **しない** 場合、特定のファイルおよびフォルダーを無視するように **.gitignore** ファイルが代わりに使用されます。

.git ファイル、**.js** 拡張子のあるファイルおよびフォルダー **tests** を無視するには、以下を **.odoignore** または **.gitignore** ファイルのいずれかに追加します。

```
.git
*.js
tests/
```

.odoignore ファイルはすべての glob 表現を許可します。

2.12. ODO CLI リファレンス

2.12.1. 基本的な odo CLI コマンド

2.12.1.1. app

OpenShift Dedicated プロジェクトに関連するアプリケーション操作を実行します。

app の使用例

```
# Delete the application
odo app delete myapp

# Describe 'webapp' application,
odo app describe webapp

# List all applications in the current project
odo app list
```

```
# List all applications in the specified project
odo app list --project myproject
```

2.12.1.2. catalog

カタログ関連の操作を実行します。

catalog の使用例

```
# Get the supported components
odo catalog list components

# Get the supported services from service catalog
odo catalog list services

# Search for a component
odo catalog search component python

# Search for a service
odo catalog search service mysql

# Describe a service
odo catalog describe service mysql-persistent
```

2.12.1.3. component

アプリケーションのコンポーネントを管理します。

component の使用例

```
# Create a new component
odo component create

# Create a local configuration and create all objects on the cluster
odo component create --now
```

2.12.1.4. config

config ファイル内で **odo** 固有の設定を変更します。

config の使用例

```
# For viewing the current local configuration
odo config view

# Set a configuration value in the local configuration
odo config set Type java
odo config set Name test
odo config set MinMemory 50M
odo config set MaxMemory 500M
odo config set Memory 250M
odo config set Ignore false
```

```
odo config set MinCPU 0.5
odo config set MaxCPU 2
odo config set CPU 1

# Set an environment variable in the local configuration
odo config set --env KAFKA_HOST=kafka --env KAFKA_PORT=6639

# Create a local configuration and apply the changes to the cluster
odo config set --now

# Unset a configuration value in the local config
odo config unset Type
odo config unset Name
odo config unset MinMemory
odo config unset MaxMemory
odo config unset Memory
odo config unset Ignore
odo config unset MinCPU
odo config unset MaxCPU
odo config unset CPU

# Unset an env variable in the local config
odo config unset --env KAFKA_HOST --env KAFKA_PORT
```

Application	Application は、コンポーネントを含める必要のあるアプリケーションの名前になります。
CPU	コンポーネントが使用できる CPU の最小数と最大数
Ignore	プッシュと監視に関連して .odoignore ファイルを考慮します。

表2.2 利用可能なローカルパラメーター:

Application	コンポーネントを含める必要のあるアプリケーションの名前
CPU	コンポーネントが使用できる CPU の最小数と最大数
Ignore	プッシュおよび監視に関連して **.odoignore** ファイルを考慮するかどうか
MaxCPU	コンポーネントで使用可能な最大 CPU
MaxMemory	コンポーネントで使用可能な最大メモリー
Memory	コンポーネントで使用できる最小および最大メモリー
MinCPU	コンポーネントで使用できる最小 CPU

MinMemory	コンポーネントに指定される最小メモリー
Name	コンポーネントの名前
Ports	コンポーネントで開くポート
Project	コンポーネントを含めるプロジェクトの名前
Ref	git ソースからコンポーネントを作成するために使用する Git ref
SourceLocation	パスはバイナリーファイルまたは git ソースの場所を示します。
SourceType	コンポーネントソースのタイプ: git/binary/local
Storage	コンポーネントのストレージ
Type	コンポーネントのタイプ
Url	コンポーネントにアクセスするために使用する URL

2.12.1.5. create

OpenShift Dedicated にデプロイするコンポーネントを記述する設定を作成します。コンポーネント名が指定されていない場合、これは自動的に生成されます。

デフォルトで、ビルダーイメージは現在の namespace から使用されます。namespace を明示的に指定するには、**odo create namespace/name:version** を使用します。バージョンが指定されていない場合、デフォルトは **latest**に設定されます。

odo catalog list を使用してデプロイできるコンポーネントタイプの詳細一覧を表示します。

create の使用例

```
# Create new Node.js component with the source in current directory.
odo create nodejs

# A specific image version may also be specified
odo create nodejs:latest

# Create new Node.js component named 'frontend' with the source in './frontend' directory
odo create nodejs frontend --context ./frontend

# Create a new Node.js component of version 6 from the 'openshift' namespace
odo create openshift/nodejs:6 --context /nodejs-ex

# Create new Wildfly component with binary named sample.war in './downloads' directory
odo create wildfly wildfly --binary ./downloads/sample.war

# Create new Node.js component with source from remote git repository
```

```
  odo create nodejs --git https://github.com/openshift/nodejs-ex.git

  # Create new Node.js git component while specifying a branch, tag or commit ref
  odo create nodejs --git https://github.com/openshift/nodejs-ex.git --ref master

  # Create new Node.js git component while specifying a tag
  odo create nodejs --git https://github.com/openshift/nodejs-ex.git --ref v1.0.1

  # Create new Node.js component with the source in current directory and ports 8080-tcp,8100-tcp
and 9100-udp exposed
  odo create nodejs --port 8080,8100/tcp,9100/udp

  # Create new Node.js component with the source in current directory and env variables key=value
and key1=value1 exposed
  odo create nodejs --env key=value,key1=value1

  # For more examples, visit: https://github.com/openshift/odo/blob/master/docs/examples.adoc
  odo create python --git https://github.com/openshift/django-ex.git

  # Passing memory limits
  odo create nodejs --memory 150Mi
  odo create nodejs --min-memory 150Mi --max-memory 300 Mi

  # Passing cpu limits
  odo create nodejs --cpu 2
  odo create nodejs --min-cpu 200m --max-cpu 2
```

2.12.1.6. delete

既存のコンポーネントを削除します。

delete の使用例

```
  # Delete component named 'frontend'.
  odo delete frontend
  odo delete frontend --all
```

2.12.1.7. describe

指定のコンポーネントについて説明します。

describe の使用例

```
  # Describe nodejs component
  odo describe nodejs
```

2.12.1.8. link

サービスまたはコンポーネントにコンポーネントをリンクします。

link の使用例

```
  # Link the current component to the 'my-postgresql' service
```

```
odo link my-postgresql

# Link component 'nodejs' to the 'my-postgresql' service
odo link my-postgresql --component nodejs

# Link current component to the 'backend' component (backend must have a single exposed port)
odo link backend

# Link component 'nodejs' to the 'backend' component
odo link backend --component nodejs

# Link current component to port 8080 of the 'backend' component (backend must have port 8080
exposed)
odo link backend --port 8080
```

リンクにより、適切なシークレットがソースコンポーネントの環境に追加されます。ソースコンポーネントは、シークレットのエントリーを環境変数として使用できます。ソースコンポーネントが指定されない場合、現在のアクティブなコンポーネントが使用されます。

2.12.1.9. list

現在のアプリケーションのすべてのコンポーネントを一覧表示します。

list の使用例

```
# List all components in the application
odo list
```

2.12.1.10. log

指定のコンポーネントのログを取得します。

log の使用例

```
# Get the logs for the nodejs component
odo log nodejs
```

2.12.1.11. login

クラスターにログインします。

login の使用例

```
# Log in interactively
odo login

# Log in to the given server with the given certificate authority file
odo login localhost:8443 --certificate-authority=/path/to/cert.crt

# Log in to the given server with the given credentials (basic auth)
odo login localhost:8443 --username=myuser --password=mypass
```

```
# Log in to the given server with the given credentials (token)
odo login localhost:8443 --token=xxxxxxxxxxxxxxxxxxxxxxxx
```

2.12.1.12. logout

現在の OpenShift Dedicated セッションからログアウトします。

logout の使用例

```
# Log out
odo logout
```

2.12.1.13. preference

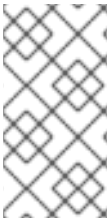
グローバル設定ファイル内の **odo** 固有の設定内容を変更します。

preference の使用例

```
# For viewing the current preferences
odo preference view

# Set a preference value in the global preference
odo preference set UpdateNotification false
odo preference set NamePrefix "app"
odo preference set Timeout 20

# Unset a preference value in the global preference
odo preference unset  UpdateNotification
odo preference unset  NamePrefix
odo preference unset  Timeout
```



注記

デフォルトで、グローバル設定ファイルへのパスは ~/**.odo/preferece.yaml** であり、これは環境変数 **GLOBALODOCONFIG** に保存されます。環境変数の値を新規の設定パスに設定し、カスタムパスをセットアップできます (例: **GLOBALODOCONFIG="new_path/preference.yaml"**)。

表2.3 利用可能なパラメーター:

NamePrefix	デフォルトのプレフィックスは、現在のディレクトリー名です。この値を使用して、デフォルトの名前のプレフィックスを設定します。
Timeout	OpenShift Dedicated サーバー接続チェックのタイムアウト (秒単位) です。
UpdateNotification	更新通知が表示されるかどうかを制御します。

2.12.1.14. project

プロジェクト操作を実行します。

project の使用例

```
# Set the active project
odo project set

# Create a new project
odo project create myproject

# List all the projects
odo project list

# Delete a project
odo project delete myproject

# Get the active project
odo project get
```

2.12.1.15. push

ソースコードをコンポーネントにプッシュします。

push の使用例

```
# Push source code to the current component
odo push

# Push data to the current component from the original source.
odo push

# Push source code in ~/mycode to component called my-component
odo push my-component --context ~/mycode
```

2.12.1.16. service

サービスカタログ操作を実行します。

service の使用例

```
# Create new postgresql service from service catalog using dev plan and name my-postgresql-db.
odo service create dh-postgresql-apb my-postgresql-db --plan dev -p postgresql_user=luke -p
postgresql_password=secret

# Delete the service named 'mysql-persistent'
odo service delete mysql-persistent

# List all services in the application
odo service list
```

2.12.1.17. storage

ストレージ操作を実行します。

storage の使用例

```
  # Create storage of size 1Gb to a component
  odo storage create mystorage --path=/opt/app-root/src/storage/ --size=1Gi
  # Delete storage mystorage from the currently active component
  odo storage delete mystorage

  # Delete storage mystorage from component 'mongodb'
  odo storage delete mystorage --component mongodb
  # List all storage attached or mounted to the current component and
  # all unattached or unmounted storage in the current application
  odo storage list
```

2.12.1.18. unlink

コンポーネントまたはサービスのリンクを解除します。

このコマンドが正常に実行されるには、サービスまたはコンポーネントが呼び出し前に **odo link** を使用してリンクされている必要があります。

unlink の使用例

```
  # Unlink the 'my-postgresql' service from the current component
  odo unlink my-postgresql

  # Unlink the 'my-postgresql' service  from the 'nodejs' component
  odo unlink my-postgresql --component nodejs

  # Unlink the 'backend' component from the current component (backend must have a single
exposed port)
  odo unlink backend

  # Unlink the 'backend' service  from the 'nodejs' component
  odo unlink backend --component nodejs

  # Unlink the backend's 8080 port from the current component
  odo unlink backend --port 8080
```

2.12.1.19. update

コンポーネントのソースコードパスを更新します。

update の使用例

```
  # Change the source code path of a currently active component to local (use the current directory as
a source)
  odo update --local

  # Change the source code path of the frontend component to local with source in ./frontend directory
  odo update frontend --local ./frontend

  # Change the source code path of a currently active component to git
  odo update --git https://github.com/openshift/nodejs-ex.git
```

```
  # Change the source code path of the component named node-ex to git
  odo update node-ex --git https://github.com/openshift/nodejs-ex.git

  # Change the source code path of the component named wildfly to a binary named sample.war in
./downloads directory
  odo update wildfly --binary ./downloads/sample.war
```

2.12.1.20. url

コンポーネントを外部に公開します。

url の使用例

```
  # Create a URL for the current component with a specific port
  odo url create --port 8080

  # Create a URL with a specific name and port
  odo url create example --port 8080

  # Create a URL with a specific name by automatic detection of port (only for components which
expose only one service port)
  odo url create example

  # Create a URL with a specific name and port for component frontend
  odo url create example --port 8080 --component frontend

  # Delete a URL to a component
  odo url delete myurl

  # List the available URLs
  odo url list

  # Create a URL in the configuration and apply the changes to the cluster
  odo url create --now
```

このコマンドを使用して生成される URL は、クラスター外からデプロイされたコンポーネントにアクセスするために使用できます。

2.12.1.21. utils

ターミナルコマンドのユーティリティーおよび odo 設定の変更

utils の使用例

```
  # Bash terminal PS1 support
  source <(odo utils terminal bash)

  # Zsh terminal PS1 support
  source <(odo utils terminal zsh)
```

2.12.1.22. version

クライアントバージョンの情報を出力します。

version の使用例

```
# Print the client version of odo
odo version
```

2.12.1.23. watch

odo は変更の有無の監視を開始し、変更時にコンポーネントを自動的に更新します。

watch の使用例

```
# Watch for changes in directory for current component
odo watch

# Watch for changes in directory for component called frontend
odo watch frontend
```

2.13. ODO 1.1.0 リリースノート

2.13.1. odo 1.1.0 の主な改善点

- IBM Z および PowerPC アーキテクチャーバイナリーが利用できるようになりました。
- **odo catalog** が改善され、より有用な出力情報が提供されるようになりました。
- **odo service create** コマンドに情報プロンプトが追加され、次のステップについてのシナリオが提供されるようになりました。
- ログの冗長性は、環境変数 **ODO_LOG_LEVEL** を使用して設定できるようになりました。
- **odo preference set PushTimeout <seconds>** を使用して、コマンドが失敗する前に、**odo** がコンポーネント Pod のデプロイを待機する秒数を指定します。
- ドキュメントが全体的に改善されました。本書には、odo の内部アーキテクチャーについての説明が含まれています。

2.13.2. サポート

ドキュメント

ドキュメントのエラーが見つかったか、またはドキュメントの改善に関する提案をお寄せいただける場合は、Bugzilla に報告してください。**OpenShift Dedicated** の製品タイプおよび **Documentation** コンポーネントタイプを選択します。

製品

エラーを見つけた場合や、**odo** の機能に関するバグが見つかった場合やこれに関する改善案をお寄せいただける場合は、Bugzilla に報告してください。製品タイプとして **Red Hat odo for OpenShift Dedicated** を選択します。

問題の詳細情報をできる限り多く入力します。

2.13.3. 修正された問題

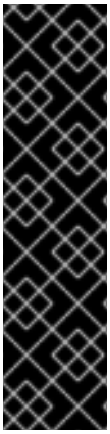
- Bug 1760573: プロジェクトの削除後に、アクティブなポインターが現在のアクティブなプロジェトに切り替わりません。
- Bug 1760578: **odo watch** コマンドは Git ベースのコマンドについてのエラーメッセージを出して失敗するはずであるのに失敗しません。
- Bug 1760583: **odo config unset** コマンドは環境変数の設定を解除しませんが、設定を解除したことを示唆します。
- Bug 1760585: **odo delete --all** は、**$HOME** から実行されると **$HOME/.odo** フォルダーを削除します。
- Bug 1760589: 自動補完は **odo push** の **--context** フラグでは機能しません。
- Bug 1761442: **component create** コマンドは、バイナリーが一時フォルダーにある場合、**--context** フラグおよび **--binary** フラグと共に使用すると失敗します。
- Bug 1783179: 環境変数を設定し、URL ルートを作成し、コンポーネントのソースコードを変更すると、URL ルートにはアクセスできなくなります。

2.13.4. 既知の問題

- Bug 1760574: 削除された namespace が **odo project get** コマンドで一覧表示されます。
- Bug 1760575: **odo app delete** コマンドはアプリケーションコンポーネントを削除しますが、サービスを削除しません。
- Bug 1760577: **odo push** コマンドは、コンポーネント名が変更されると OpenShift オブジェクトを削除しません。
- Bug 1760586: **odo delete** コマンドは、プロジェクトが削除され、コンポ ーネント名が設定されると無限ループを開始します。
- Bug 1760588: **odo service create** コマンドは Cygwin で実行されるとクラッシュします。
- Bug 1760590: Windows 用の Git BASH では、 **odo login -u developer** は要求された場合も入力されたパスワードを非表示にしません。
- Bug 1783188: 非接続クラスターでは、コンポーネントがカタログリストに一覧表示されているとしても、**odo component create** コマンドがエラー **...tag not found...** をスローします。
- Bug 1761440: 1 つのオブジェクトに同じタイプの 2 つのサービスを作成することができません。

2.13.5. odo 1.1.0 のテクノロジープレビュー機能

odo debug は、ユーザーが OpenShift Dedicated の Pod で実行されているコンポーネントにローカルデバッガーを割り当てることを可能にする機能です。



重要

odo debug は現時点ではテクノロジープレビュー機能です。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。Red Hat は実稼働環境でこれらを使用することを推奨していません。これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、お客様は機能性をテストし、開発プロセス中にフィードバックをお寄せいただくことができます。

Red Hat のテクノロジープレビュー機能のサポート範囲についての詳細は、https://access.redhat.com/ja/support/offerings/techpreview/ を参照してください。

- **odo debug port-forward** コマンドを使用してポート転送を開始します。
- **odo config set DebugPort 9292** コマンドを使用して、デバッグエージェントが実行すべきリモートポートを指定します。
- **odo debug port-forward --local-port 9292** コマンドを使用すると、ポート転送のローカルポートを指定できます。