OpenShift Dedicated 4

イメージ

OpenShift Dedicated 4 でのイメージおよびイメージストリームの作成および管理

Last Updated: 2020-05-15

OpenShift Dedicated 4 イメージ

OpenShift Dedicated 4 でのイメージおよびイメージストリームの作成および管理

法律上の通知

概要

本書では、OpenShift Dedicated 4 でイメージおよびイメージストリームを作成し、管理する方法を説明します。さらに、テンプレートの使用方法についても説明します。

目次

第1章 コンテナー、イメージおよびイメージストリームについて

コンテナー、イメージ、およびイメージストリームは、コンテナー化されたソフトウェアを作成し、管理する際に理解しておくべき重要な概念です。イメージは、コンテナーがコンテナーイメージの実行中のインスタンスである場合に、実行の準備ができている一連のソフトウェアを保持します。イメージストリームは、同一の基本的なイメージの異なるバージョンを保存する 1 つの方法です。それらの異なるバージョンは、同じイメージ名の異なるタグによって表されます。

1.1. イメージ

OpenShift Dedicated のコンテナーは OCI または Docker 形式のコンテナーのイメージをベースにしています。イメージは、単一コンテナーを実行するためのすべての要件、およびそのニーズおよび機能を記述するメタデータを含むバイナリーです。

これはパッケージ化テクノロジーとして考えることができます。コンテナーには、作成時にコンテナーに追加のアクセスを付与しない限り、イメージで定義されるリソースにのみアクセスできます。同じイメージを複数のホストにまたがって複数のコンテナーにデプロイし、それらの間で負荷を分散することにより、OpenShift Dedicated はイメージにパッケージ化されたサービスの冗長性および水平的なスケーリングを提供できます。

イメージをビルドするために podman または **docker** CLI を直接使用することはできますが、OpenShift Dedicated は、コードまたは設定を既存イメージに追加して新規イメージの作成を支援するビルダーイメージも提供します。

アプリケーションは一定期間をかけて開発されるため、単一のイメージ名が同じイメージの数多くの異なるバージョンを参照する場合があります。異なるイメージはそれぞれ、そのハッシュ (長い 16 進数、例: **fd44297e2ddb050ec4f...**) で一意に参照され、通常は 12 文字 (例: **fd44297e2ddb**) に短縮されます。

1.2. コンテナー

OpenShift Dedicated アプリケーションの基本的な単位はコンテナーと呼ばれています。Linux コンテナーテクノロジーは、指定されたリソースのみとの対話に制限されるように、実行中のプロセスを分離する軽量なメカニズムです。このコンテナーという用語は、コンテナーイメージの実行中または一時停止している特定のインスタンスとして定義されています。

数多くのアプリケーションインスタンスは、相互のプロセス、ファイル、ネットワークなどを可視化せずに単一ホストのコンテナーで実行される可能性があります。通常、コンテナーは任意のワークロードに使用されますが、各コンテナーは Web サーバーまたはデータベースなどの (通常は「マイクロサービス」と呼ばれることの多い) 単一サービスを提供します。

Linux カーネルは数年にわたりコンテナーテクノロジーの各種機能を統合してきました。Docker プロジェクトはホスト上の Linux コンテナーの便利な管理インターフェースを開発しました。さらに最近では、Open Container Initiative により、コンテナー形式およびコンテナーランタイムのオープン標準が策定されています。OpenShift Dedicated および Kubernetes は複数ホストのインストール間で OCI および Docker 形式のコンテナーのオーケストレーションを実行する機能を追加しています。

OpenShift Dedicated を使用する際にコンテナーランタイムと直接対話することはありませんが、それらの OpenShift Dedicated における役割やコンテナー内でのアプリケーションの機能を理解する上で、それらの機能および用語を理解しておくことは重要です。

podman などのツールは、コンテナーを直接実行し、管理するための **docker** コマンドラインツールを置き換えるために使用できます。**podman** を使用すると、OpenShift Dedicated と切り離してコンテナーの実験を行うことができます。

1.3. イメージレジストリー

イメージレジストリー は、コンテナーイメージを保管し、提供するコンテナーサーバーです。以下は例になります。

```
registry.redhat.io
```

レジストリーには、1 つ以上のタグ付けされたイメージを持つ 1 つ以上のイメージリポジトリーのコレクションが含まれます。Red Hat は、サブスクリプションをお持ちのお客様に対して **registry.redhat.io** でレジストリーを提供しています。また、OpenShift Dedicated はカスタムコンテナーイメージを管理するための独自の内部レジストリーも提供しています。

1.4. イメージリポジトリー

イメージリポジトリー は、関連するコンテナーイメージおよびそれらを特定するタグのコレクションです。たとえば、OpenShift Jenkins イメージはリポジトリーにあります。

```
docker.io/openshift/jenkins-2-centos7
```

1.5. イメージタグ

イメージタグは、イメージストリーム内の他のイメージから特定のイメージを識別するリポジトリーのコンテナーイメージに適用されるラベルです。通常、タグはある種のバージョン番号を表します。たとえば、ここでは v3.11.59-2 がタグになります。

```
registry.access.redhat.com/openshift3/jenkins-2-rhel7:v3.11.59-2
```

イメージにタグを追加することができます。たとえば、イメージには :v3.11.59-2 および :latest というタグが割り当てられる可能性があります。

OpenShift Dedicated は **docker tag** コマンドに似ている **oc tag** コマンドを提供しますが、イメージ上で直接動作するのではなくイメージストリーム上で動作します。

1.6. イメージ ID

イメージ ID は、イメージをプルするために使用できる SHA (Secure Hash Algorithm) コードです。SHA イメージ ID は変更できません。特定の SHA ID は同一のコンテナーイメージコンテンツを常に参照します。以下は例になります。

```
docker.io/openshift/jenkins-2-centos7@sha256:ab312bda324
```

1.7. イメージストリームの使用

イメージストリームおよびその関連付けられたタグは、OpenShift Dedicated 内でコンテナーイメージを参照するための抽象化を提供します。イメージストリームとそのタグを使用して、利用可能なイメージを確認し、リポジトリーのイメージが変更される場合でも必要な特定のイメージを使用していることを確認できます。

イメージストリームには実際のイメージデータは含まれませんが、イメージリポジトリーと同様に、関連するイメージの単一の仮想ビューを提示します。

ビルドおよびデプロイメントは、イメージストリームで新規イメージの追加時の通知の有無を監視するように設定し、ビルドまたはデプロイメントをそれぞれ実行してこれに対応するように設定できます。

たとえば、デプロイメントで特定のイメージを使用していて、そのイメージの新規バージョンが作成される場合、デプロイメントを、そのイメージの新規バージョンを選択できるように自動的に実行きます。

ただし、デプロイメントまたはビルドで使用されるイメージストリームタグが更新されない場合は、コンテナーイメージレジストリーのコンテナーイメージが更新されても、ビルドまたはデプロイメントは以前のおそらくは既知の正常なイメージをそのまま使用します。

ソースイメージは以下のいずれかに保存できます。

- OpenShift Dedicated の統合レジストリー。
- **registry.redhat.io** または **hub.docker.com** などの外部レジストリー
- OpenShift Dedicated クラスターの他のイメージストリーム。

(ビルドまたはデプロイメント設定などの) イメージストリームタグを参照するオブジェクトを定義する場合は、Docker リポジトリーではなく、イメージストリームタグを参照します。アプリケーションのビルドまたはデプロイ時に、OpenShift Dedicated はイメージストリームタグを使用して Docker リポジトリーにクエリーを送信し、イメージの関連付けられた ID を特定し、正確なイメージを使用します。

イメージストリームメタデータは他のクラスター情報と共に etcd インスタンスに保存されます。

イメージストリームの使用には、いくつかの大きな利点があります。

- コマンドラインを使用して再プッシュすることなく、タグ付けや、タグのロールバック、およびイメージの迅速な処理を実行できます。
- 新規イメージがレジストリーにプッシュされると、ビルドおよびデプロイメントをトリガーできます。また、OpenShift Dedicated には他のリソースの汎用トリガーがあります (Kubernetes オブジェクトなど)。
- 定期的な再インポートを実行するためにタグにマークを付けることができます。ソースイメージが変更された場合、その変更は認識され、イメージストリームに反映されます。これにより、ビルドまたはデプロイメント設定のいずれかに応じて、ビルドおよび/またはデプロイメントフローがトリガーされます。
- 詳細なアクセス制御を使用してイメージを共有し、チーム間でイメージを迅速に分散できます。
- ソースイメージが変更されても、イメージストリームタグはイメージの既知の正常なバージョンを参照したままになり、アプリケーションに予期しない障害が発生しないようにします。
- イメージストリームオブジェクトのパーミッションを使用して、イメージを表示し、使用できるユーザーについてのセキュリティー設定を行うことができます。
- クラスターレベルでイメージを読み込んだり、一覧表示するパーミッションのないユーザーでも、イメージストリームを使用してプロジェクトでタグ付けされたイメージを取得できます。

1.7.1. イメージストリームタグ

イメージストリームタグは、イメージストリームのイメージに対する名前付きポインターです。イメージストリームタグはコンテナーイメージタグに似ています。

1.8. イメージストリームイメージ

イメージストリームイメージは、これがタグ付けされている特定のイメージストリームから特定のコンテナーイメージを取得できるようにします。イメージストリームイメージは、特定のイメージの SHA ID についてのメタデータをプルする API リソースオブジェクトです。

1.9. イメージストリームトリガー

イメージストリームトリガーは、イメージストリームタグの変更時に特定のアクションを生じさせます。たとえば、インポートにより、タグの値が変更され、これによりデプロイメント、ビルドまたはそれらをリッスンする他のリソースがある場合にトリガーが実行されます。

1.10. 追加リソース

第2章 イメージの作成

使用可能な事前にビルドされたイメージを使用して独自のコンテナーイメージを作成する方法について確認します。このプロセスには、イメージの作成、イメージのメタデータの定義、イメージのテストおよびカスタムビルダーワークフローを使用した OpenShift Dedicated で使用できるイメージの作成のベストプラクティスを理解することが含まれます。

2.1. コンテナーのベストプラクティスについて

OpenShift Dedicated で実行するコンテナーイメージを作成する場合には、イメージの作成者は、イメージの使いやすさの点で数多くのベストプラクティスを考慮する必要があります。イメージは変更不可で、そのままの状態で使用されることが意図されているため、以下のガイドラインは、イメージを使用しやすく、OpenShift Dedicated で簡単に使用できるようにするのに役立ちます。

2.1.1. コンテナーイメージの一般的なガイドライン

以下のガイドラインは、イメージが OpenShift Dedicated で使用されるかどうかにかかわらず、コンテナーイメージの作成時に一般的に適用されます。

イメージの再利用
可能な場合は、**FROM** ステートメントを使用し、適切なアップストリームイメージをベースとしてイメージを設定することを推奨します。これにより、依存関係を直接更新する必要なく、イメージが更新時にアップストリームイメージからセキュリティー修正を簡単に取得できるようになります。

さらに、**FROM** 命令 (例: **rhel:rhel7**) のタグを使用して、お使いのイメージがどのバージョンのイメージをベースとしているかを明確にします。アップストリームイメージの **latest** バージョンを使用すると互換性に影響のある変更が組み込まれる可能性があるため、**latest** 以外のタグを使用することができます。

タグ内の互換性の維持
独自のイメージにタグを付ける場合には、タグ内で後方互換性が維持されるようにすることを推奨します。たとえば、**foo** という名前のイメージがあり、現時点でバージョン 1.0 が含まれている場合には、タグに **foo:v1** を指定します。イメージの更新時には、元のイメージとの互換性がある限り、新しいイメージに **foo:v1** のタグを付けることができ、このタグのダウンストリームのコンシューマーは、互換性に関する影響を被ることなく更新を取得できるようになります。

互換性のない更新を後にリリースした場合には、**foo:v2** などの新しいタグに切り替える必要があります。これにより、ダウンストリームのコンシューマーはいつでも新しいバージョンに移行できますが、意図せずにこの互換性のない新規イメージによる影響を受けることはありません。**foo:latest** を使用するダウンストリームコンシューマーには、互換性のない変更が導入されるリスクがあります。

複数プロセスの回避
データベースや **SSHD** など複数のサービスを 1 つのコンテナー内で起動しないようにしてください。コンテナーは軽量で、複数のプロセスをオーケストレーションするために簡単にリンクできるので、複数プロセスの実行は不要です。OpenShift Dedicated では、関連のあるイメージを 1 つの Pod にグループ化して、簡単に共存させ、共同管理することができます。

このように共存させることで、コンテナーはネットワークの namespace とストレージを通信用に共有できるようになります。また、イメージの更新頻度が低く、個別に更新されるので、更新による中断の可能性が低くなります。シグナル処理フローは、複数の起動したプロセスへのルーティングシグナルを管理する必要がないので、単一プロセスによって明確になります。

ラッパースクリプトでの **exec** の使用
多くのイメージはラッパースクリプトを使用して、実行されるソフトウェアのプロセスを開始する前にいくつかの設定を行います。イメージがこのようなスクリプトを使用する場合、そのスクリプトは、ス

クリプトのプロセスがソフトウェアによって置き換えられるように **exec** を使用するはずです。**exec** を使用しない場合、コンテナーランタイムによって送信されるシグナルが、ソフトウェアのプロセスではなくラッパースクリプトに送られます。これは、以下で説明するように望ましいアクションではありません。

たとえば、一部のサーバーのプロセスを開始するラッパースクリプトがあるとします。コンテナーを起動する (たとえば、**podman run -i** を使用する) と、それによりラッパースクリプトが実行され、次にプロセスが開始されます。ここで、**CTRL+C** でコンテナーを強制終了する必要があるとします。ラッパースクリプトが exec を使用してサーバープロセスを開始した場合、**podman** は SIGINT をサーバープロセスに送信し、すべてが予想通りに機能します。ラッパースクリプトで exec を使用しなかった場合、**podman** はラッパースクリプトのプロセスに SIGINT を送信し、何も生じなかったかのようにプロセスは実行し続けます。

また、コンテナー内で実行すると、プロセスは PID 1 として実行される点に留意してください。つまり、主なプロセスが中断された場合には、コンテナー全体が停止され、PID 1 プロセスから起動した子プロセスが終了します。

その他の影響については、Docker and the PID 1 zombie reaping problem のブログ記事を参照してください。また、PID 1 および **init** システムの詳細については、Demystifying the init system (PID 1) のブログ記事も参照してください。

一時ファイルの消去
ビルドプロセスで作成される一時ファイルはすべて削除する必要があります。これには、**ADD** コマンドで追加したファイルも含まれます。たとえば、**yum install** の操作を実行してから、**yum clean** コマンドを実行することを強く推奨します。

yum キャッシュがイメージレイヤーに残らないように、以下のように **RUN** ステートメントを作成します。

```
RUN yum -y install mypackage && yum -y install myotherpackage && yum clean all -y
```

以下のように記述した場合には注意してください。

```
RUN yum -y install mypackage
RUN yum -y install myotherpackage && yum clean all -y
```

上記のように記述すると、最初の **yum** 呼び出しにより、対象のレイヤーに追加のファイルが残り、**yum clean** 操作を後に実行してもこれらのファイルは削除できません。これらの追加ファイルは最終イメージでは確認できませんが、下位レイヤーには存在します。

現在のコンテナービルドプロセスでは、前のレイヤーで何かが削除された場合でも、後のレイヤーでコマンドを実行してイメージが使用する容量を縮小できません。ただし、これについては今後変更される可能性はあります。後のレイヤーでファイルが表示されていなくても **rm** コマンドを実行したとしても、ダウンロードするイメージの全体のサイズを縮小することになりません。そのため、**yum clean** の場合のように、可能な場合は後にレイヤーに書き込まれないように、ファイルの作成に使用したのと同じコマンドでファイルを削除することが最も適切と言えます。

また、単一の **RUN** ステートメントで複数のコマンドを実行すると、イメージのレイヤー数が減り、ダウンロードと実行時間が短縮されます。

正しい順序での命令の指定
コンテナービルダーは **Dockerfile** を読み取り、トップダウンで命令を実行します。命令が正常に実行されると、同じイメージが次回ビルドされるときや、別のイメージがビルドされる時に再利用することができるレイヤーが作成されます。**Dockerfile** の上部にほとんど変更されない命令を配置することは非常に重要です。こうすることで、上位レイヤーで加えられた変更によってキャッシュが無効にならないので、同じイメージの次回のビルドをすばやく実行できます。

たとえば、反復するファイルをインストールするための **ADD** コマンドと、パッケージを **yum install** する **RUN** コマンドが含まれる **Dockerfile** で作業を行う場合には、**ADD** コマンドを最後に配置することが最善の方法です。

```
FROM foo
RUN yum -y install mypackage && yum clean all -y
ADD myfile /test/myfile
```

これにより、**myfile** を編集して **podman build** または **docker build** を返すたびに、システムは **yum** コマンドのキャッシュされたレイヤーを再利用し、**ADD** 操作に対してのみ新規レイヤーを生成します。

代わりに **Dockerfile** を以下のように作成した場合:

```
FROM foo
ADD myfile /test/myfile
RUN yum -y install mypackage && yum clean all -y
```

myfile を変更して、**podman build** または **docker build** を再実行するたびに、**ADD** 操作は **RUN** レイヤーのキャッシュを無効にするので、**yum** 操作も再実行する必要があります。

重要なポートのマーク付け
EXPOSE 命令は、ホストシステムで利用できるコンテナーおよび他のコンテナーにポートを作成します。ポートを **podman run** の起動で公開されるように指定できますが、**Dockerfile** で EXPOSE 命令を使用すると、ソフトウェアが実行する必要のあるポートを明示的に宣言することで、人間とソフトウェアの両方がイメージをより簡単に使用できるようになります。

- 公開されるポートは、イメージから作成されるコンテナーに関連付けられて **podman ps** の下に表示されます。
- 公開されるポートは、**podman inspect** によって返されるイメージのメタデータにも表示されます。
- 公開されるポートは、1つのコンテナーを別のコンテナーにリンクする際にリンクされます。

環境変数の設定
ENV 命令で環境変数を設定することが適切です。一例として、プロジェクトのバージョンを設定するなどが挙げられます。バージョンを設定することで、**Dockerfile** を確認せずにバージョンを簡単に見つけ出すことができます。別の例としては、**JAVA_HOME** など、別のプロセスで使用可能なシステムでパスを公開する場合などです。

デフォルトのパスワードの回避
デフォルトのパスワードは設定しないことが最善の策です。イメージを拡張して、デフォルトのパスワードを削除または変更するのを忘れることが多くあります。これは、実稼働環境で使用するユーザーに誰でも知っているパスワードが割り当てられると、セキュリティーの問題に発展する可能性があります。パスワードは、環境変数を使用して設定できるようにする必要があります。

デフォルトのパスワードを設定することにした場合には、コンテナーの起動時に適切な警告メッセージが表示されるようにしてください。メッセージはデフォルトパスワードの値をユーザーに通知し、環境変数の設定など、パスワードの変更方法を説明するものである必要があります。

SSHD の回避
イメージで **sshd** を実行しないようにしてください。ローカルホストで実行中のコンテナーにアクセスするには、**podman exec** または **docker exec** コマンドを使用できます。または、**oc exec** コマンドまたは **oc rsh** コマンドを使用して、OpenShift Dedicated クラスターで実行中のコンテナーにアクセス

できます。イメージで **sshd** をインストールし、実行すると、攻撃の経路が増え、セキュリティー修正が必要になります。

永続データ向けのボリュームの使用
イメージは、永続データ用にボリュームを使用する必要があります。こうすることで、OpenShift Dedicated により、コンテナーを実行するノードにネットワークストレージがマウントされ、コンテナーが新しいノードに移動した場合に、ストレージはそのノードに再度割り当てられます。永続ストレージのすべての要件に対応するようにボリュームを使用することで、コンテナーが再起動されたり、移動されたりしても、コンテンツは保存されます。イメージがコンテナー内の任意の場所にデータを書き込む場合には、コンテンツは保存されない可能性があります。

コンテナーが破棄された後も保存する必要のあるデータはすべて、ボリュームに書き込む必要があります。コンテナーエンジンはコンテナーの **readonly** フラグをサポートしており、このフラグを使用して、コンテナーの一時ストレージにデータが決して記述されないようにすることができます。イメージをこの機能に基づいて設計すると、この機能を後に利用することがより簡単になります。

さらに、**Dockerfile** でボリュームを明示的に定義すると、イメージの利用者がイメージの実行時に定義する必要のあるボリュームがどれかを簡単に理解できるようになります。

OpenShift Dedicated でのボリュームの使用方法についての詳細は、Kubernetes ドキュメントを参照してください。

注記

永続ボリュームでも、イメージの各インスタンスには独自のボリュームがあり、ファイルシステムはインスタンス間で共有されません。つまり、ボリュームを使用してクラスターの状態を共有できません。

追加リソース

- Docker ドキュメント - Best practices for writing Dockerfiles
- Project Atomic ドキュメント - Guidance for Container Image Authors

2.1.2. OpenShift Dedicated 固有のガイドライン

以下は、OpenShift Dedicated で使用するためのコンテナーイメージの作成時に適用されるガイドラインです。

Source-To-Image (S2I) 向けのイメージの有効化
開発者が提供した Ruby コードを実行するように設計された Ruby イメージなど、サードパーティー提供のアプリケーションコードを実行することが目的のイメージの場合には、イメージを Source-to-Image (S2I) ビルドツールと連携できるようにすることができます。S2I は、インプットとして、アプリケーションのソースコードを受け入れるイメージを簡単に記述でき、アセンブルされたアプリケーションをアウトプットとして実行する新規イメージを簡単に生成することができるフレームワークです。

たとえば、この Python イメージは S2I スクリプトを定義して、Python アプリケーションのさまざまなバージョンをビルドします。

任意のユーザー **ID** のサポート
デフォルトでは OpenShift Dedicated は、任意に割り当てられたユーザー ID を使用してコンテナーを実行します。こうすることで、コンテナーエンジンの脆弱性が原因でコンテナーから出ていくプロセスに対して追加のセキュリティーを設定でき、ホストノードでパーミッションのエスカレーションが可能になります。

イメージが任意ユーザーとしての実行をサポートできるように、イメージ内のプロセスで記述される可

能性のあるディレクトリーやファイルは、root グループが所有し、このグループに対して読み取り/書き込みの権限を割り当てる必要があります。実行予定のファイルには、グループの実行権限も必要です。

以下を Dockerfile に追加すると、root グループのユーザーがビルドされたイメージでアクセスできるように、ディレクトリーおよびファイルのパーミッションが設定されます。

```
RUN chgrp -R 0 /some/directory &&\
    chmod -R g=u /some/directory
```

コンテナーユーザーは常に root グループのメンバーであるため、コンテナーユーザーはこれらのファイルに対する読み取り、書き込みが可能です。

警告

コンテナーの慎重に扱うべき分野のディレクトリーおよびファイルパーミッションを変更する場合には注意が必要です（通常のシステムの扱い方と同様です)。

/etc/passwd などの慎重に扱うべき分野に適用されると、意図しないユーザーによるこのようなファイルの変更が可能となり、コンテナーやホストにセキュリティー上のリスクが生じる可能性があります。CRI-O は、ランダムユーザー ID のコンテナーの **/etc/passwd** への挿入をサポートするため、そのパーミッションを変更する必要はありません。

さらに、コンテナーで実行中のプロセスは、特権のあるユーザーとして実行されていないので、特権のあるポート (1024 未満のポート) をリッスンできません。

イメージ間通信でのサービスの使用
データの保存や取得のためにデータベースイメージにアクセスする必要のある Web フロントエンドイメージなど、別のイメージが提供するサービスとイメージが通信する場合には、イメージは OpenShift Dedicated サービスを使用する必要があります。サービスは、コンテナーが停止、開始、または移動しても変更されない静的アクセスエンドポイントを提供します。さらに、サービスにより、要求が負荷分散されます。

共通のライブラリーの提供
サードパーティーが提供するアプリケーションコードの実行を目的とするイメージの場合は、プラットフォーム用として共通に使用されるライブラリーをイメージに含めるようにしてください。とくに、プラットフォームで使用する共通のデータベース用のデータベースドライバーを設定してください。たとえば、Java フレームワークイメージを作成する場合に、MySQL や PostgreSQL には JDBC ドライバーを設定します。このように設定することで、アプリケーションのアセンブリー時に共通の依存関係をダウンロードする必要がなくなり、アプリケーションイメージのビルドがスピードアップします。また、すべての依存関係の要件を満たすためのアプリケーション開発者の作業が簡素化されます。

設定での環境変数の使用
イメージのユーザーは、ダウンストリームイメージをイメージに基づいて作成しなくても、イメージの設定が行えるようにしてください。つまり、ランタイム設定は環境変数を使用して処理される必要があります。単純な設定の場合、実行中のプロセスは環境変数を直接使用できます。より複雑な設定や、これをサポートしないランタイムの場合、起動時に処理されるテンプレート設定ファイルを定義してランタイムを設定します。このプロセス時に、環境変数を使用して渡される値は設定ファイルで置き換えることも、この値を使用して、設定ファイルに指定するオプションを決定することもできます。

環境変数を使用して、コンテナーに証明書やキーなどのシークレットを渡すこともでき、これは推奨されています。環境変数を使用することで、シークレット値がイメージにコミットされたり、コンテナーイメージレジストリーに漏洩されることはありません。

環境変数を指定することで、イメージの利用者は、イメージ上に新しいレイヤーを作成することなく、データベースの設定、パスワード、パフォーマンスチューニングなどの動作をカスタマイズできます。Pod の定義時に環境変数の値を定義するだけで、イメージの再ビルドなしに設定を変更できます。

非常に複雑なシナリオの場合、ランタイム時にコンテナーにマウントされるボリュームを使用して設定を指定することも可能です。ただし、この方法を使用する場合には、必要なボリュームや設定が存在しない場合に明確なエラーメッセージが起動時に表示されるように、イメージが設定されている必要があります。

サービスエンドポイントの情報を渡す環境変数としてデータソースなどの設定を定義する必要がある点で、これはイメージ間の通信でのサービスの使用についてのトピックと関連しています。これにより、アプリケーションは、アプリケーションイメージを変更せずに、OpenShift Dedicated 環境に定義されているデータソースサービスを動的に使用できます。

さらに、コンテナーの **cgroups** 設定を確認して、調整を行う必要があります。これにより、イメージは利用可能なメモリー、CPU、他のリソースに合わせてチューニングが可能になります。たとえば、Java ベースのイメージは、制限を超えず、メモリー不足のエラーが表示されないように、**cgroup** の最大メモリーパラメーターを基にヒープをチューニングする必要があります。

コンテナーの **cgroup** クォータを管理する方法については、以下の資料を参照してください。

- ブログ記事 - Resource management in Docker
- Docker ドキュメント - Runtime metrics
- ブログ記事 - Memory inside Linux containers

イメージのメタデータ設定
イメージのメタデータを定義することで、OpenShift Dedicated によるコンテナーイメージの使用が改善され、開発者が OpenShift Dedicated でイメージを使用しやすくなります。たとえば、メタデータを追加して、イメージに関する役立つ情報を提供したり、必要とされる可能性のある他のイメージを提案したりできます。

クラスタリング
イメージの複数のインスタンスを実行するとはどういうことかを十分に理解しておく必要があります。最も単純な例では、サービスの負荷分散機能は、イメージのすべてのインスタンスにトラフィックをルーティングします。ただし、セッションの複製などで、リーダーの選択やフェイルオーバーの状態を実行するには、多くのフレームワークが情報を共有する必要があります。

OpenShift Dedicated での実行時に、インスタンスでこのような通信を実現する方法を検討します。Pod 同士は直接通信できますが、Pod が起動、停止、移動するたびに IP アドレスが変更されます。そのため、クラスタリングスキームを動的にしておくことが重要です。

ロギング
すべてのロギングを標準出力に送信することが推奨されます。OpenShift Dedicated はコンテナーから標準出力を収集し、表示が可能な中央ロギングサービスに送信します。ログコンテンツを分離する必要がある場合には、出力のプレフィックスに適切なキーワードを指定して、メッセージをフィルタリングできるようにしてください。

お使いのイメージがファイルにロギングをする場合には、手動で実行中のコンテナーに入り、ログファイルを取得または表示する必要があります。

Liveness および **Readiness** プローブ

イメージで使用可能な liveness および readiness プローブの例をまとめます。これらのプローブにより、処理の準備ができるまでトラフィックがコンテナーにルーティングされず、プロセスが正常でない状態になる場合にコンテナーが再起動されるので、ユーザーはイメージを安全にデプロイできます。

テンプレート
イメージと共にテンプレートサンプルを提供することも検討してください。テンプレートがあると、ユーザーは、正しく機能する設定を指定してイメージをすばやく簡単にデプロイ できるようになります。完全を期すため、テンプレートには、イメージに関連して記述した liveness および readiness プローブ を含めるようにしてください。

追加リソース

- Docker basics
- Dockerfile reference
- Project Atomic Guidance for Container Image Authors

2.2. イメージへのメタデータの組み込み

イメージのメタデータを定義することで、OpenShift Dedicated によるコンテナーイメージの使用が改善され、開発者が OpenShift Dedicated でイメージを使用しやすくなります。たとえば、メタデータを追加して、イメージに関する役立つ情報を提供したり、必要とされる可能性のある他のイメージを提案したりできます。

このトピックでは、現在の一連のユースケースに必要なメタデータのみを定義します。他のメタデータまたはユースケースは、今後追加される可能性があります。

2.2.1. イメージメタデータの定義

Dockerfile で **LABEL** 命令を使用して、イメージのメタデータを定義することができます。ラベルは、イメージやコンテナーに割り当てるキーと値のペアである点で環境変数と似ています。ただし、ラベルは、実行中のアプリケーションに表示されず、イメージやコンテナーをすばやく検索する場合にも使用できる点で、環境変数とは異なります。

LABEL 命令に関する詳細は、Docker ドキュメントを参照してください。

ラベル名には、通常 namespace を指定する必要があります。namespace は、対象のラベルを選択して使用するプロジェクトを反映するように設定してください。OpenShift Dedicated の場合は、namespace は **io.openshift** に、Kubernetes の場合は、namespace は **io.k8s** に設定してください。

形式に関する詳細は、Docker のカスタムメタデータに関するドキュメントを参照してください。

表2.1 サポートされるメタデータ

変数	説明
io.openshift.tags	このラベルには、カンマ区切りの文字列値の一覧として表現されているタグの一覧が含まれます。タグを使用して、コンテナーイメージを幅広い機能エリアに分類します。タグを使用すると、UI および生成ツールがアプリケーションの作成プロセスで適切なコンテナーイメージを提案しやすくなります。 `LABEL io.openshift.tags mongodb,mongodb24,nosql`

変数	説明
io.openshift.wants	コンテナーイメージにすでにタグが指定されていない場合に、生成ツールと UI が適切な提案を行うのに使用するタグの一覧を指定します。たとえば、コンテナーイメージに **mysql** と **redis** が必要で、コンテナーイメージに **redis** タグが指定されていない場合には、UI はこのイメージをデプロイメントに追加するように提案する可能性があります。 LABEL io.openshift.wants mongodb,redis
io.k8s.description	このラベルは、コンテナーイメージの利用者に、このイメージが提供するサービスや機能に関する詳細情報を提供するのに使用できます。UI は、この説明とコンテナーイメージ名を使用して、人間が解読しやすい情報をエンドユーザーに提供します。 LABEL io.k8s.description The MySQL 5.5 Server with master-slave replication support
io.openshift.non-scalable	イメージは、この変数を使用して、スケーリングがサポートされていない旨を示します。その後、UI はこれをそのイメージのコンシューマーに通知します。スケーリング不可にした場合は基本的に **replicas** の値を最初に 1 よりも大きい値に設定することはできません。 LABEL io.openshift.non-scalable true
io.openshift.min-memory および **io.openshift.min-cpu**	このラベルは、コンテナーイメージが正しく機能するにはどの程度リソースが必要かを提案します。UI でユーザーに対し、このコンテナーイメージをデプロイすると、ユーザークォータを超過する可能性があることを警告します。この値は、Kubernetes の数量と互換性がある必要があります。 LABEL io.openshift.min-memory 8Gi LABEL io.openshift.min-cpu 4

2.3. S2I イメージのテスト

Source-to-Image (S2I) ビルダーイメージの作成者は、S2I イメージをローカルでテストして、自動テストや継続的な統合に OpenShift Dedicated ビルドシステムを使用できます。

S2I ビルドを正常に実行するには、S2I に **assemble** と **run** スクリプトが必要です。S2I 外のコンテナーイメージを実行した場合に、**save-artifacts** スクリプトがあると、ビルドのアーティファクトが再利用され、**usage** スクリプトがあると、使用についての情報がコンソールに出力されるようになります。

S2I イメージのテストは、ベースのコンテナーイメージを変更したり、コマンドが使用するツールが更新されたりした場合でも、上記のコマンドが正しく機能することを確認するのが目的です。

2.3.1. テスト要件について

test スクリプトは、基本的に **test/run** に配置されます。このスクリプトは、OpenShift Dedicated S2I イメージビルダーが呼び出し、単純な Bash スクリプトか静的な Go バイナリーのいずれかの形式を取ることができます。

test/run スクリプトは S2I ビルドを実行するので、S2I バイナリーを **$PATH** で利用可能にしておく必要があります。必要に応じて、S2I README のインストール手順に従います。

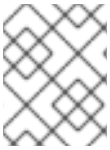
S2I は、アプリケーションのソースコードおよびビルダーイメージを統合します。これをテストするには、ソースが実行可能なコンテナーイメージに変換されることを検証するためのサンプルアプリケーションのソースが必要です。サンプルアプリケーションは単純なものである必要がありますが、**assemble** および **run** スクリプトの重要な手順を実行できる必要があります。

2.3.2. スクリプトおよびツールの生成

S2I ツールは、新しい S2I イメージの作成プロセスを加速化する強力な生成ツールと共に提供されます。**s2i create** コマンドでは、**Makefile** 以外に、必要とされる S2I スクリプトとテストツールすべてが生成されます。

```
$ s2i create _<image name>_ _<destination directory>_
```

生成された **test/run** スクリプトは、より使いやすくするために調整する必要がありますが、このスクリプトを開発の開始段階で使用することが推奨されます。



注記

s2i create コマンドで生成した **test/run** スクリプトでは、サンプルアプリケーションのソースを **test/test-app** ディレクトリーに配置しておく必要があります。

2.3.3. ローカルでのテスト

S2I イメージテストをローカルに実行する最も簡単な方法として、生成した **Makefile** を使用することができます。

s2i create コマンドを使用しない場合には、以下の **Makefile** テンプレートをコピーして、**IMAGE_NAME** パラメーターをお使いのイメージ名に置き換えることができます。

Makefile の例

```
IMAGE_NAME = openshift/ruby-20-centos7
CONTAINER_ENGINE := $(shell command -v podman 2> /dev/null | echo docker)

build:
 ${CONTAINER_ENGINE} build -t $(IMAGE_NAME) .

.PHONY: test
test:
 ${CONTAINER_ENGINE} build -t $(IMAGE_NAME)-candidate .
 IMAGE_NAME=$(IMAGE_NAME)-candidate test/run
```

2.3.4. テストの基本的なワークフロー

test スクリプトは、テストするイメージをすでにビルドしていることが前提です。必要に応じて、以下のコマンドで S2I イメージを先にビルドしてください。以下のいずれかのコマンドを実行してください。

- Podman を使用する場合は、以下のコマンドを実行します。

```
$ podman build -t _<BUILDER_IMAGE_NAME>_
```

- Docker を使用する場合は、以下のコマンドを実行します。

```
$ docker build -t _<BUILDER_IMAGE_NAME>_
```

以下の手順では、S2I イメージビルダーをテストするデフォルトのワークフローを説明しています。

1. **usage** スクリプトが機能していることを確認します。
   - Podman を使用する場合は、以下のコマンドを実行します。

```
$ podman run _<BUILDER_IMAGE_NAME>_ .
```

   - Docker を使用する場合は、以下のコマンドを実行します。

```
$ docker run _<BUILDER_IMAGE_NAME>_ .
```

2. イメージをビルドします。

```
$ s2i build file:///path-to-sample-app _<BUILDER_IMAGE_NAME>_
_<OUTPUT_APPLICATION_IMAGE_NAME>_
```

3. オプション: **save-artifacts** をサポートする場合には、再度手順 2 を実行して、保存して復元するアーティファクトが正しく機能することを確認します。
4. コンテナーを実行します。
   - Podman を使用する場合は、以下のコマンドを実行します。

```
$ podman run _<OUTPUT_APPLICATION_IMAGE_NAME>_
```

   - Docker を使用する場合は、以下のコマンドを実行します。

```
$ docker run _<OUTPUT_APPLICATION_IMAGE_NAME>_
```

5. コンテナーが実行され、アプリケーションが応答していることを確認します。

これらの手順を実行すると、通常はビルダーイメージが予想通りに機能しているかどうかが分かります。

2.3.5. イメージのビルドでの OpenShift Dedicated の使用

新しい S2I ビルダーイメージを構成する **Dockerfile** と他のアーティファクトが準備できたら、それらを git リポジトリーに配置して、OpenShift Dedicated を使用し、イメージをビルドしてプッシュします。その後は、お使いのリポジトリーを参照する Docker ビルドを定義することのみが必要になります。

OpenShift Dedicated インスタンスが公開 IP アドレスでホストされる場合、ビルドは、S2I ビルダーイメージ GitHub リポジトリーにプッシュするたびにトリガーされます。

ImageChangeTrigger を使用して、更新した S2I ビルダーイメージに基づくアプリケーションの再ビルドをトリガーすることもできます。

第3章 イメージの管理

3.1. イメージの管理の概要

OpenShift Dedicated では、イメージのレジストリーが置かれる場所やレジストリー関連の認証要件、およびビルドとデプロイメントで必要とされる動作に応じてイメージと対話し、イメージストリームをセットアップできます。

3.1.1. イメージの概要

イメージストリームは、タグで識別される数多くのコンテナーイメージで構成されます。これはコンテナーイメージリポジトリーのように関連イメージの単一仮想ビューを提供します。

イメージストリームを監視することにより、ビルドおよびデプロイメントは新規イメージの追加または変更時に通知を受信し、それぞれビルドまたはデプロイメントを実行してこれに対応します。

3.2. イメージのタグ付け

以下のセクションでは、OpenShift Dedicated イメージストリームおよびそれらのタグを操作するためにコンテナーイメージのコンテキストでイメージタグを使用する概要および方法について説明します。

3.2.1. イメージタグ

イメージタグは、イメージストリーム内の他のイメージから特定のイメージを識別するリポジトリーのコンテナーイメージに適用されるラベルです。通常、タグはある種のバージョン番号を表します。たとえば、ここでは v3.11.59-2 がタグになります。

```
registry.access.redhat.com/openshift3/jenkins-2-rhel7:v3.11.59-2
```

イメージにタグを追加することができます。たとえば、イメージには :v3.11.59-2 および :latest というタグが割り当てられる可能性があります。

OpenShift Dedicated は **docker tag** コマンドに似ている **oc tag** コマンドを提供しますが、イメージ上で直接動作するのではなくイメージストリーム上で動作します。

3.2.2. イメージタグの規則

イメージは時間の経過と共に変化するもので、それらのタグはその変化を反映します。ほとんどの場合、イメージタグはビルドされる最新イメージを常に参照します。

v2.0.1-may-2019 のように、タグ名に非常に多くの情報が組み込まれる場合、タグはイメージの単一のリビジョンのみを参照し、更新されることはありません。デフォルトのイメージのプルーニングオプションを使用しても、このようなイメージは削除されません。

タグの名前が **v2.0** である場合はイメージのリビジョンの数が多くなることが予想されます。これによりタグ履歴が長くなるため、イメージプルーナーが古くなり使われなくなったイメージを削除する可能性が高くなります。

タグの名前付け規則は各自で定めることができますが、ここでは **<image_name>:<image_tag>** 形式のいくつかの例を見てみましょう。

表3.1 イメージタグの名前付け規則

説明	例
リビジョン	**myimage:v2.0.1**
アーキテクチャー	**myimage:v2.0-x86_64**
ベースイメージ	**myimage:v1.2-centos7**
最新 (不安定な可能性がある)	**myimage:latest**
最新 (安定性がある)	**myimage:stable**

タグ名に日付を含める必要がある場合、古くなり使用されなくなったイメージおよび **istags** を定期的に検査し、これらを削除してください。そうしないと、古いイメージを保持して、リソースの使用量が増大する可能性があります。

3.2.3. タグのイメージストリームへの追加

OpenShift Dedicated のイメージストリームは、タグで識別される 0 個以上のコンテナーイメージで構成されます。

各種のタグを利用できます。デフォルト動作では、特定の時点の特定のイメージを参照する永続タグを使用します。_permanent_tag が使用され、ソースが変更される場合、タグは宛先について変更されません。

追跡 タグの場合は、宛先タグのメタデータがソースタグのインポート時に更新されます。

手順

- **oc tag** コマンドを使用して、タグをイメージストリームに追加できます。

```
$ oc tag <source> <destination>
```

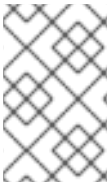
  たとえば、**ruby** イメージストリームの **static-2.0** タグを **ruby** イメージストリーム **2.0** タグの現行のイメージを常に参照するように設定するには、以下を実行します。

```
$ oc tag ruby:2.0 ruby:static-2.0
```

  これにより、**ruby** イメージストリームに **static-2.0** という名前のイメージストリームタグが新たに作成されます。この新規タグは、**oc tag** の実行時に **ruby:2.0** イメージストリームタグが参照したイメージ ID を直接参照し、これが参照するイメージが変更されることはありません。

- 宛先タグがソースタグの変更時に常に更新されるようにするには、**--alias=true** フラグを使用します。

```
$ oc tag --alias=true <source> <destination>
```



注記

永続的なエイリアス (**latest** または **stable** など) を作成するには、追跡タグを使用します。このタグは単一イメージストリーム内でのみ適切に機能します。複数のイメージストリーム間で使用されるエイリアスを作成しようとするとエラーが生じます。

- また、**--scheduled=true** フラグを追加して、宛先タグが定期的に更新 (再インポート) されるようにもできます。期間はシステムレベルでグローバルに設定できます。
- **--reference** フラグは、インポートされていないイメージストリームタグを作成します。このタグはソースの場所を参照しますが、これを永続的に参照します。
  統合レジストリーのタグ付けされたイメージを常にフェッチするように OpenShift に指示するには、**--reference-policy=local** を使用します。レジストリーはプルスルー (pull-through) 機能を使用してイメージをクライアントに提供します。デフォルトで、イメージ Blob はレジストリーによってローカルにミラーリングされます。その結果、それらが次回必要となる場合により迅速にプルされます。また、このフラグは **--insecure-registry** をコンテナーランタイムに指定しなくても、イメージストリームに非セキュアなアノテーションがあるか、またはタグに非セキュアなインポートポリシーがある限り、非セキュアなレジストリーからのプルを許可します。

3.2.4. タグのイメージストリームからの削除

タグをイメージストリームから削除できます。

手順

タグをイメージストリームから完全に削除するには、以下を実行します。

```
$ oc delete istag/ruby:latest
```

または、以下を実行します。

```
$ oc tag -d ruby:latest
```

3.2.5. イメージストリームでのイメージの参照

タグを使用してイメージストリームのイメージを参照するには、以下の参照タイプを使用します。

表3.2 イメージストリームの参照タイプ

参照タイプ	説明
ImageStreamTag	**ImageStreamTag** は、所定のイメージストリームおよびタグのイメージを参照または取得するために使用されます。
ImageStreamImage	**ImageStreamImage** は、所定のイメージストリームおよびイメージの **sha** ID のイメージを参照または取得するために使用されます。

参照タイプ	説明
DockerImage	**DockerImage** は、所定の外部レジストリーのイメージを参照または取得するために使用されます。この名前は、標準の Docker **プル仕様**に基づいて付けられます。

イメージストリーム定義のサンプルを表示すると、これらには **ImageStreamTag** の定義と **DockerImage** の参照が含まれていますが、**ImageStreamImage** に関連するものは何も含まれていないことに気づくでしょう。

これは、**ImageStreamImage** オブジェクトが、イメージをイメージストリームにインポートしたり、タグ付けしたりする際に OpenShift Dedicated に自動的に作成されるためです。イメージストリームを作成するために使用するイメージストリーム定義に **ImageStreamImage** オブジェクトを明示的に定義する必要はありません。

手順

- 所定のメージストリームおよびタグのイメージを参照するには、**ImageStreamTag** を使用します。

```
<image_stream_name>:<tag>
```

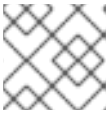
- 所定のイメージストリームおよびイメージの **sha** ID のイメージを参照するには、**ImageStreamImage** を使用します。

```
<image_stream_name>@<id>
```

<id> は、ダイジェストとも呼ばれる特定イメージのイミュータブルな ID です。

- 所定の外部レジストリーのイメージを参照または取得するには、**DockerImage** を使用します。

```
openshift/ruby-20-centos7:2.0
```



注記

タグが指定されていない場合、**latest** タグが使用されることが想定されます。

サードパーティーのレジストリーを参照することもできます。

```
registry.redhat.io/rhel7:latest
```

またはダイジェストでイメージを参照できます。

```
centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b2
8e
```

3.2.6. 追加情報

- CentOS イメージストリームのイメージストリーム定義の例。

3.3. イメージプルポリシー

Pod のそれぞれのコンテナーにはコンテナーイメージが含まれます。イメージを作成してレジストリーにプッシュすると、このイメージを Pod で参照できます。

3.3.1. イメージプルポリシーの概要

OpenShift Dedicated はコンテナーを作成する際に、コンテナーの **imagePullPolicy** を使用して、コンテナーの起動前にイメージをプルする必要があるかどうかを判別します。**imagePullPolicy** には以下の 3 つの値があります。

表3.3 **imagePullPolicy** の値

値	説明
Always	常にイメージをプルします。
IfNotPresent	イメージがノード上にない場合にのみイメージをプルします。
Never	イメージをプルしません。

コンテナーの **imagePullPolicy** パラメーターが指定されていない場合、OpenShift Dedicated はイメージのタグに基づいてこれを設定します。

1. タグが **latest** の場合、OpenShift Dedicated は **imagePullPolicy** を **Always** にデフォルト設定します。
2. それ以外の場合に、OpenShift Dedicated は **imagePullPolicy** を **IfNotPresent** にデフォルト設定します。

3.4. イメージプルシークレットの使用

OpenShift Dedicated の内部レジストリーを使用し、同じプロジェクトにあるイメージストリームからプルしている場合は、Pod のサービスアカウントには適切なパーミッションがすでに設定されているため、追加のアクションは不要です。

ただし、OpenShift Dedicated プロジェクト全体でイメージを参照する場合や、セキュリティー保護されたレジストリーからイメージを参照するなどの他のシナリオでは、追加の設定手順が必要になります。

3.4.1. Pod が複数のプロジェクト間でイメージを参照できるようにする設定

内部レジストリーを使用している場合で **project-a** の Pod が **project-b** のイメージを参照できるようにするには、**project-a** のサービスアカウントが **project-b** の **system:image-puller** ロールにバインドされている必要があります。

手順

1. **project-a** の Pod が **project-b** のイメージを参照できるようにするには、**project-a** のサービスアカウントを **project-b** の **system:image-puller** ロールにバインドします。

```
$ oc policy add-role-to-user \
    system:image-puller system:serviceaccount:project-a:default \
    --namespace=project-b
```

このロールを追加すると、デフォルトのサービスアカウントを参照する **project-a** の Pod は **project-b** からイメージをプルできるようになります。

2. **project-a** のすべてのサービスアカウントにアクセスを許可するには、グループを使用します。

```
$ oc policy add-role-to-group \
    system:image-puller system:serviceaccounts:project-a \
    --namespace=project-b
```

3.4.2. Pod が他のセキュリティー保護されたレジストリーからイメージを参照できるようにする設定

Docker クライアントの **.dockercfg $HOME/.docker/config.json** ファイルは、セキュア/非セキュアなレジストリーに事前にログインしている場合に認証情報を保存する Docker 認証情報ファイルです。

OpenShift Dedicated の内部レジストリーにないセキュリティー保護されたコンテナーイメージをプルするには、Docker 認証情報でプルシークレットを作成し、これをサービスアカウントに追加する必要があります。

手順

- セキュリティー保護されたレジストリーの **.dockercfg** ファイルがすでにある場合は、以下を実行してそのファイルからシークレットを作成できます。

```
$ oc create secret generic <pull_secret_name> \
    --from-file=.dockercfg=<path/to/.dockercfg> \
    --type=kubernetes.io/dockercfg
```

- または、**$HOME/.docker/config.json** ファイルがある場合は以下を実行します。

```
$ oc create secret generic <pull_secret_name> \
    --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
    --type=kubernetes.io/dockerconfigjson
```

- セキュリティー保護されたレジストリーの Docker 認証情報がない場合は、以下を実行してシークレットを作成できます。

```
$ oc create secret docker-registry <pull_secret_name> \
    --docker-server=<registry_server> \
    --docker-username=<user_name> \
    --docker-password=<password> \
    --docker-email=<email>
```

- Pod のイメージのプルにシークレットを使用するには、シークレットをサービスアカウントに追加する必要があります。この例では、サービスアカウントの名前は、Pod が使用するサービスアカウントの名前に一致している必要があります。**default** はデフォルトのサービスアカウントです。

```
$ oc secrets link default <pull_secret_name> --for=pull
```

- ビルドイメージのプッシュおよびプルにシークレットを使用するには、シークレットは Pod 内でマウント可能である必要があります。以下でこれを実行できます。

```
$ oc secrets link builder <pull_secret_name>
```

3.4.2.1. 委任された認証を使用したプライベートレジストリーからのプル

プライベートレジストリーは認証を別個のサービスに委任できます。この場合、イメージプルシークレットは認証およびレジストリーのエンドポイントの両方に対して定義される必要があります。

手順

1. 委任された認証サーバーのシークレットを作成します。

```
$ oc create secret docker-registry \
    --docker-server=sso.redhat.com \
    --docker-username=developer@example.com \
    --docker-password=******** \
    --docker-email=unused \
    redhat-connect-sso

secret/redhat-connect-sso
```

2. プライベートレジストリーのシークレットを作成します。

```
$ oc create secret docker-registry \
    --docker-server=privateregistry.example.com \
    --docker-username=developer@example.com \
    --docker-password=******** \
    --docker-email=unused \
    private-registry

secret/private-registry
```

3.4.3. グローバルクラスターのプルシークレットの更新

クラスターのグローバルプルシークレットを更新できます。

警告

クラスターリソースは新規のプルシークレットに合わせて調整する必要がありますが、これにより、クラスターのユーザビリティーが一時的に制限される可能性があります。

前提条件

- アップロードする新規または変更されたプルシークレットファイルがある。
- **cluster-admin** ロールを持つユーザーとしてクラスターにアクセスできる。

手順

- 以下のコマンドを実行して、クラスターのグローバルプルシークレットを更新します。

```
$ oc set data secret/pull-secret -n openshift-config --from-file=.dockerconfigjson=<pull-secret-location> 1
```

1 新規プルシークレットファイルへのパスを指定します。

この更新はすべてのノードにロールアウトされます。これには、クラスターのサイズに応じて多少時間がかかる場合があります。この間に、ノードがドレイン (解放) され、Pod は残りのノードで再スケジュールされます。

第4章 テンプレートの使用

以下のセクションでは、テンプレートの概要と共に、それらを使用し、作成する方法についての概要を説明します。

4.1. テンプレートについて

テンプレートでは、パラメーター化や処理が可能な一連のオブジェクトを記述し、OpenShift Dedicated で作成するためのオブジェクトの一覧を生成します。テンプレートは、サービス、ビルド設定およびデプロイメント設定など、プロジェクト内で作成パーミッションがあるすべてのものを作成するために処理できます。また、テンプレートではラベルのセットを定義して、これをテンプレート内に定義されたすべてのオブジェクトに適用できます。

オブジェクトの一覧は CLI を使用してテンプレートから作成することも、テンプレートがプロジェクトまたはグローバルテンプレートライブラリーにアップロードされている場合、Web コンソールを使用することもできます。

4.2. テンプレートのアップロード

テンプレートを定義する JSON または YAML ファイルがある場合は、この例にあるように、CLI を使用してプロジェクトにテンプレートをアップロードできます。こうすることで、プロジェクトにテンプレートが保存され、対象のプロジェクトに対して適切なアクセス権があるユーザーがこれを繰り返し使用できます。独自のテンプレートの記述については、このトピックで後ほど説明します。

手順

- 現在のプロジェクトのテンプレートライブラリーにテンプレートをアップロードするには、JSON または YAML ファイルを以下のコマンドで渡します。

```
$ oc create -f <filename>
```

- **-n** オプションを使用してプロジェクト名を指定することで、別のプロジェクトにテンプレートをアップロードできます。

```
$ oc create -f <filename> -n <project>
```

テンプレートは、Web コンソールまたは CLI を使用して選択できるようになりました。

4.3. WEB コンソールを使用したアプリケーションの作成

Web コンソールを使用して、テンプレートからアプリケーションを作成することができます。

手順

1. 必要なプロジェクトで **Add to Project** をクリックします。
2. プロジェクト内にあるイメージの一覧またはサービスカタログからビルダーイメージを選択します。

注記

以下に示すように、**builder** タグがアノテーションに列挙されるイメージストリームタグのみがこの一覧に表示されます。

```
kind: "ImageStream"
apiVersion: "v1"
metadata:
  name: "ruby"
  creationTimestamp: null
spec:
  dockerImageRepository: "registry.redhat.io/openshift3/ruby-20-rhel7"
  tags:
    -
      name: "2.0"
      annotations:
        description: "Build and run Ruby 2.0 applications"
        iconClass: "icon-ruby"
        tags: "builder,ruby" (1)
        supports: "ruby:2.0,ruby"
        version: "2.0"
```

(1) ここに **builder** を含めると、この **ImageStreamTag** がビルダーとして Web コンソールに表示されます。

3. 新規アプリケーション画面で設定を変更し、オブジェクトをアプリケーションをサポートするように設定します。

4.4. CLI を使用してテンプレートからオブジェクトを作成する手順

CLI を使用して、テンプレートを処理し、オブジェクトを作成するために生成された設定を使用できます。

4.4.1. ラベルの追加

ラベルは、Pod などの生成されたオブジェクトを管理し、整理するために使用されます。テンプレートで指定されるラベルは、テンプレートから生成されるすべてのオブジェクトに適用されます。

手順

- コマンドラインからテンプレートにラベルを追加します。

```
$ oc process -f <filename> -l name=otherLabel
```

4.4.2. パラメーターの一覧表示

上書きできるパラメーターの一覧は、テンプレートの**parameters** セクションに表示されます。

手順

1. CLI で以下のコマンドを使用し、使用するファイルを指定して、パラメーターを一覧表示することができます。

```
$ oc process --parameters -f <filename>
```

または、テンプレートがすでにアップロードされている場合には、以下を実行します。

```
$ oc process --parameters -n <project> <template_name>
```

たとえば、デフォルトの **openshift** プロジェクトにあるクイックスタートテンプレートのいずれかに対してパラメーターを一覧表示する場合に、以下のような出力が表示されます。

```
$ oc process --parameters -n openshift rails-postgresql-example
NAME                     DESCRIPTION
GENERATOR          VALUE
SOURCE_REPOSITORY_URL        The URL of the repository with your application source
code                                    https://github.com/sclorg/rails-ex.git
SOURCE_REPOSITORY_REF        Set this to a branch name, tag or other ref of your
repository if you are not using the default branch
CONTEXT_DIR                  Set this to the relative path to your project if it is not in the root of
your repository
APPLICATION_DOMAIN           The exposed hostname that will route to the Rails service
rails-postgresql-example.openshiftapps.com
GITHUB_WEBHOOK_SECRET        A secret string used to configure the GitHub webhook
expression          [a-zA-Z0-9]{40}
SECRET_KEY_BASE              Your secret key for verifying the integrity of signed cookies
expression          [a-z0-9]{127}
APPLICATION_USER             The application user that is used within the sample application
to authorize access on pages                    openshift
APPLICATION_PASSWORD         The application password that is used within the sample
application to authorize access on pages                secret
DATABASE_SERVICE_NAME        Database service name
postgresql
POSTGRESQL_USER              database username
expression          user[A-Z0-9]{3}
POSTGRESQL_PASSWORD          database password
expression          [a-zA-Z0-9]{8}
POSTGRESQL_DATABASE          database name
root
POSTGRESQL_MAX_CONNECTIONS   database max connections
10
POSTGRESQL_SHARED_BUFFERS    database shared buffers
12MB
```

この出力から、テンプレートの処理時に正規表現のようなジェネレーターで生成された複数のパラメーターを特定できます。

4.4.3. オブジェクト一覧の生成

CLI を使用して、標準出力にオブジェクト一覧を返すテンプレートを定義するファイルを処理できます。

手順

1. 標準出力にオブジェクト一覧を返すテンプレートを定義するファイルを処理します。

```
$ oc process -f <filename>
```

または、テンプレートがすでに現在のプロジェクトにアップロードされている場合には以下を実行します。

```
$ oc process <template_name>
```

2. テンプレートを処理し、**oc create** の出力をパイプして、テンプレートからオブジェクトを作成します。

```
$ oc process -f <filename> | oc create -f -
```

または、テンプレートがすでに現在のプロジェクトにアップロードされている場合には以下を実行します。

```
$ oc process <template> | oc create -f -
```

3. 上書きする **<name>=<value>** の各ペアに、**-p** オプションを追加することで、ファイルに定義されたパラメーターの値を上書きできます。パラメーター参照は、テンプレートアイテム内のテキストフィールドに表示される場合があります。
たとえば、テンプレートの以下の **POSTGRESQL_USER** および **POSTGRESQL_DATABASE** パラメーターを上書きし、カスタマイズされた環境変数の設定を出力します。

a. テンプレートからのオブジェクト一覧の作成

```
$ oc process -f my-rails-postgresql \
    -p POSTGRESQL_USER=bob \
    -p POSTGRESQL_DATABASE=mydatabase
```

b. JSON ファイルは、ファイルにリダイレクトすることも、**oc create** コマンドで処理済みの出力をパイプして、テンプレートをアップロードせずに直接適用することも可能です。

```
$ oc process -f my-rails-postgresql \
    -p POSTGRESQL_USER=bob \
    -p POSTGRESQL_DATABASE=mydatabase \
    | oc create -f -
```

c. 多数のパラメーターがある場合は、それらをファイルに保存してからそのファイルを **oc process** に渡すことができます。

```
$ cat postgres.env
POSTGRESQL_USER=bob
POSTGRESQL_DATABASE=mydatabase
$ oc process -f my-rails-postgresql --param-file=postgres.env
```

d. **--param-file** の引数として **"-"** を使用して、標準入力から環境を読み込むこともできます。

```
$ sed s/bob/alice/ postgres.env | oc process -f my-rails-postgresql --param-file=-
```

4.5. アップロードしたテンプレートの変更

すでにプロジェクトにアップロードされているテンプレートを編集できます。

手順

- すでにアップロードされているテンプレートを変更します。

```
$ oc edit template <template>
```

4.6. インスタントアプリおよびクイックスタートテンプレートの使用

OpenShift Dedicated では、デフォルトで、インスタントアプリとクイックスタートテンプレートを複数提供しており、各種言語で簡単に新規アプリの構築を開始できます。Rails (Ruby)、Django (Python)、Node.js、CakePHP (PHP) および Dancer (Perl) 用のテンプレートを利用できます。クラスター管理者は、これらのテンプレートを利用できるようにデフォルトのグローバル **openshift** プロジェクトにこれらのテンプレートを作成している必要があります。

デフォルトで、テンプレートビルドは 必要なアプリケーションコードが含まれる GitHub の公開ソースリポジトリーを使用して行われます。

手順

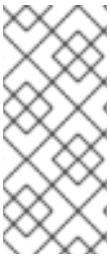
1. 以下のように、利用可能なデフォルトのインスタントアプリとクイックスタートテンプレートを一覧表示できます。

```
$ oc get templates -n openshift
```

2. ソースを変更し、アプリケーションの独自のバージョンをビルドするには、以下を実行します。

   a. テンプレートのデフォルト **SOURCE_REPOSITORY_URL** パラメーターが参照するリポジトリーをフォークします。

   b. テンプレートから作成する場合には、**SOURCE_REPOSITORY_URL** パラメーターの値を上書きします。 デフォルト値ではなく、フォークを指定してください。
   これにより、テンプレートで作成したビルド設定はアプリケーションコードのフォークを参照するようになり、コードを変更し、アプリケーションを自由に再ビルドできます。



注記

一部のインスタンスアプリおよびクイックスタートのテンプレートで、データベースのデプロイメント設定を定義します。テンプレートが定義する設定では、データベースコンテンツ用に一時ストレージを使用します。データベース Pod が何らかの理由で再起動されると、データベースの全データが失われてしまうので、これらのテンプレートはデモ目的でのみ使用する必要があります。

4.6.1. クイックスタートのテンプレート

クイックスタートは、OpenShift Dedicated で実行するアプリケーションの基本的なサンプルです。クイックスタートではさまざまな言語やフレームワークを使用でき、サービスのセット、ビルド設定およびデプロイメント設定などで構成されるテンプレートで定義されています。このテンプレートは、必要なイメージやソースリポジトリーを参照して、アプリケーションをビルドし、デプロイします。

クイックスタートを確認するには、テンプレートからアプリケーションを作成します。管理者がこれらのテンプレートを OpenShift Dedicated クラスターにすでにインストールしている可能性がありますが、その場合には、Web コンソールからこれを簡単に選択できます。

クイックスタートは、アプリケーションのソースコードを含むソースリポジトリーを参照します。クイックスタートをカスタマイズするには、リポジトリーをフォークし、テンプレートからアプリケーションを作成する時に、デフォルトのソースリポジトリー名をフォークしたリポジトリーに置き換えま

す。これにより、提供されたサンプルのソースではなく、独自のソースコードを使用してビルドが実行されます。ソースリポジトリーでコードを更新し、新しいビルドを起動して、デプロイされたアプリケーションで変更が反映されていることを確認できます。

4.6.1.1. Web フレームワーククイックスタートのテンプレート

以下のクイックスタートテンプレートでは、指定のフレームワークおよび言語の基本アプリケーションを提供します。

- CakePHP: PHP Web フレームワーク (MySQL データベースを含む)
- Dancer: Perl Web フレームワーク (MySQL データベースを含む)
- Django: Python Web フレームワーク (PostgreSQL データベースを含む)
- NodeJS: NodeJS web アプリケーション (MongoDB データベースを含む)
- Rails: Ruby Web フレームワーク (PostgreSQL データベースを含む)

4.7. テンプレートの作成

アプリケーションの全オブジェクトを簡単に再作成するために、新規テンプレートを定義できます。テンプレートでは、作成するオブジェクトと、これらのオブジェクトの作成をガイドするメタデータを定義します。

以下は、単純なテンプレートオブジェクト定義 (YAML) の例です。

```
apiVersion: v1
kind: Template
metadata:
  name: redis-template
  annotations:
    description: "Description"
    iconClass: "icon-redis"
    tags: "database,nosql"
objects:
- apiVersion: v1
  kind: Pod
  metadata:
    name: redis-master
  spec:
    containers:
    - env:
      - name: REDIS_PASSWORD
        value: ${REDIS_PASSWORD}
      image: dockerfile/redis
      name: master
      ports:
      - containerPort: 6379
        protocol: TCP
parameters:
- description: Password used for Redis authentication
  from: '[A-Z0-9]{8}'
  generate: expression
```

```
    name: REDIS_PASSWORD
  labels:
    redis: master
```

4.7.1. テンプレート記述の作成

テンプレートの記述により、テンプレートの内容に関する情報を提供でき、Web コンソールでの検索時に役立ちます。テンプレート名以外のメタデータは任意ですが、使用できると便利です。メタデータには、一般的な説明などの情報以外にタグのセットも含まれます。便利なタグにはテンプレートで使用する言語名などがあります (例: **java**、**php**、**ruby**)。

以下は、テンプレート記述メタデータの例です。

```
kind: Template
apiVersion: v1
metadata:
  name: cakephp-mysql-example (1)
  annotations:
    openshift.io/display-name: "CakePHP MySQL Example (Ephemeral)" (2)
    description: >-
      An example CakePHP application with a MySQL database. For more information
      about using this template, including OpenShift considerations, see
      https://github.com/sclorg/cakephp-ex/blob/master/README.md.


      WARNING: Any data stored will be lost upon pod destruction. Only use this
      template for testing." (3)
    openshift.io/long-description: >-
      This template defines resources needed to develop a CakePHP application,
      including a build configuration, application DeploymentConfig, and
      database DeploymentConfig.  The database is stored in
      non-persistent storage, so this configuration should be used for
      experimental purposes only. (4)
    tags: "quickstart,php,cakephp" (5)
    iconClass: icon-php (6)
    openshift.io/provider-display-name: "Red Hat, Inc." (7)
    openshift.io/documentation-url: "https://github.com/sclorg/cakephp-ex" (8)
    openshift.io/support-url: "https://access.redhat.com" (9)
  message: "Your admin credentials are ${ADMIN_USERNAME}:${ADMIN_PASSWORD}" (10)
```

(1) テンプレートの一意の名前。

(2) ユーザーインターフェースで利用できるように、ユーザーに分かりやすく、簡単な名前。

(3) テンプレートの説明。デプロイされる内容、デプロイ前に知っておく必要のある注意点をユーザーが理解できるように詳細を追加します。**README** ファイルなど、追加情報へのリンクも追加できます。パラグラフを作成するには、改行を追加できます。

(4) 追加の説明。たとえば、サービスカタログに表示されます。

(5) 検索およびグループ化を実行するためにテンプレートに関連付けられるタグ。これを指定されるカタログカテゴリーのいずれかに組み込むタグを追加します。コンソールの定数ファイルの **CATALOG_CATEGORIES** で **id** および **categoryAliases** を参照してください。

6 Web コンソールでテンプレートと一緒に表示されるアイコン。可能な場合は、既存のロゴアイコンから選択します。また、FontAwesome からもアイコンを使用できます。

7 テンプレートを提供する人または組織の名前

8 テンプレートに関する他のドキュメントを参照する URL

9 テンプレートに関するサポートを取得できる URL

10 テンプレートがインスタンス化された時に表示される説明メッセージ。このフィールドで、新規作成されたリソースの使用方法をユーザーに通知します。生成された認証情報や他のパラメーターを出力に追加できるように、メッセージの表示前にパラメーターの置換が行われます。ユーザーが従うべき次の手順が記載されたドキュメントへのリンクを追加してください。

4.7.2. テンプレートラベルの作成

テンプレートにはラベルのセットを追加できます。これらのラベルは、テンプレートがインスタンス化される時に作成されるオブジェクトごとに追加します。このようにラベルを定義すると、特定のテンプレートから作成された全オブジェクトの検索、管理が簡単になります。

以下は、テンプレートオブジェクトのラベルの例です。

```
kind: "Template"
apiVersion: "v1"
...
labels:
  template: "cakephp-mysql-example" 1
  app: "${NAME}" 2
```

1 このテンプレートから作成する全オブジェクトに適用されるラベル

2 パラメーター化されたラベル。このラベルは、このテンプレートを基に作成された全オブジェクトに適用されます。パラメーターは、ラベルキーおよび値の両方で拡張されます。

4.7.3. テンプレートパラメーターの作成

パラメーターにより、テンプレートがインスタンス化される時に値を生成するか、ユーザーが値を指定できるようになります。パラメーターが参照されると、値が置換されます。参照は、オブジェクト一覧フィールドであればどこでも定義できます。これは、無作為にパスワードを作成したり、テンプレートのカスタマイズに必要なユーザー固有の値やホスト名を指定したりできるので便利です。 パラメーターは、2 種類の方法で参照可能です。

- 文字列の値として、テンプレートの文字列フィールドに **${PARAMETER_NAME}** の形式で配置する
- json/yaml の値として、テンプレートのフィールドに **${{PARAMETER_NAME}}** の形式で配置する

${PARAMETER_NAME}** 構文を使用すると、複数のパラメーター参照を 1 つのフィールドに統合でき、**"http://${PARAMETER_1}${PARAMETER_2}" などのように、参照を固定データ内に埋め込むことができます。どちらのパラメーター値も置換されて、引用された文字列が最終的な値になります。

${{PARAMETER_NAME}} 構文のみを使用する場合は、単一のパラメーター参照のみが許可され、先頭文字や終了文字は使用できません。結果の値は、置換後に結果が有効な json オブジェクトの場合は引用

されません。結果が有効な json 値でない場合に、結果の値は引用され、標準の文字列として処理されます。

単一のパラメーターは、テンプレート内で複数回参照でき、1 つのテンプレート内で両方の置換構文を使用して参照することができます。

デフォルト値を指定でき、ユーザーが別の値を指定していない場合に使用されます。

以下は、明示的な値をデフォルト値として設定する例です。

```
parameters:
  - name: USERNAME
    description: "The user name for Joe"
    value: joe
```

パラメーター値は、パラメーター定義に指定したルールを基に生成することも可能です。 以下は、パラメーター値の生成例です。

```
parameters:
  - name: PASSWORD
    description: "The random user password"
    generate: expression
    from: "[a-zA-Z0-9]{12}"
```

上記の例では、処理後に、英字の大文字、小文字、数字すべてを含む 12 文字長のパスワードが無作為に作成されます。

利用可能な構文は、完全な正規表現構文ではありません。ただし、**\w**、**\d**、および **\a** 修飾子を使用できます。

- **[\w]{10}** は、10 桁の英字、数字、およびアンダースコアを生成します。これは PCRE 標準に準拠し、**[a-zA-Z0-9_]{10}** に相当します。
- **[\d]{10}** は 10 桁の数字を生成します。これは **[0-9]{10}** に相当します。
- **[\a]{10}** は 10 桁の英字を生成します。これは **[a-zA-Z]{10}** に相当します。

以下は、パラメーター定義と参照を含む完全なテンプレートの例です。

```
kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
  - kind: BuildConfig
    apiVersion: v1
    metadata:
      name: cakephp-mysql-example
      annotations:
        description: Defines how to build the application
    spec:
      source:
        type: Git
        git:
          uri: "${SOURCE_REPOSITORY_URL}" 1
```

```
        ref: "${SOURCE_REPOSITORY_REF}"
      contextDir: "${CONTEXT_DIR}"
  - kind: DeploymentConfig
    apiVersion: v1
    metadata:
      name: frontend
    spec:
      replicas: "${{REPLICA_COUNT}}" 2
parameters:
  - name: SOURCE_REPOSITORY_URL 3
    displayName: Source Repository URL 4
    description: The URL of the repository with your application source code 5
    value: https://github.com/sclorg/cakephp-ex.git 6
    required: true 7
  - name: GITHUB_WEBHOOK_SECRET
    description: A secret string used to configure the GitHub webhook
    generate: expression 8
    from: "[a-zA-Z0-9]{40}" 9
  - name: REPLICA_COUNT
    description: Number of replicas to run
    value: "2"
    required: true
message: "... The GitHub webhook secret is ${GITHUB_WEBHOOK_SECRET} ..." 10
```

1. この値は、テンプレートがインスタンス化された時点で **SOURCE_REPOSITORY_URL** パラメーターに置き換えられます。
2. この値は、テンプレートがインスタンス化された時点で、**REPLICA_COUNT** パラメーターの引用なしの値に置き換えられます。
3. パラメーター名。この値は、テンプレート内でパラメーターを参照するのに使用します。
4. 分かりやすいパラメーターの名前。これは、ユーザーに表示されます。
5. パラメーターの説明。期待値に対する制約など、パラメーターの目的を詳細にわたり説明します。説明には、コンソールのテキスト標準に従い、完結した文章を使用するようにしてください。表示名と同じ内容を使用しないでください。
6. テンプレートをインスタンス化する時に、ユーザーにより値が上書きされない場合に使用されるパラメーターのデフォルト値。パスワードなどのデフォルト値の使用を避けるようにしてください。シークレットと組み合わせた生成パラメーターを使用するようにしてください。
7. このパラメーターが必須であることを示します。つまり、ユーザーは空の値で上書きできません。パラメーターでデフォルト値または生成値が指定されていない場合には、ユーザーは値を指定する必要があります。
8. 値が生成されるパラメーター
9. ジェネレーターへの入力。この場合、ジェネレーターは、大文字、小文字を含む 40 桁の英数字の値を生成します。
10. パラメーターはテンプレートメッセージに含めることができます。これにより、生成された値がユーザーに通知されます。

4.7.4. テンプレートオブジェクト一覧の作成

テンプレートの主な部分は、テンプレートがインスタンス化される時に作成されるオブジェクトの一覧です。これには、**BuildConfig**、**DeploymentConfig**、**Service** など、有効な API オブジェクトを使用できます。オブジェクトはここで定義された通りに作成され、パラメーターの値は作成前に置換されます。これらのオブジェクトの定義では、以前に定義したパラメーターを参照できます。

以下は、オブジェクト一覧の例です。

```
kind: "Template"
apiVersion: "v1"
metadata:
  name: my-template
objects:
  - kind: "Service" 1
    apiVersion: "v1"
    metadata:
      name: "cakephp-mysql-example"
      annotations:
        description: "Exposes and load balances the application pods"
    spec:
      ports:
        - name: "web"
          port: 8080
          targetPort: 8080
      selector:
        name: "cakephp-mysql-example"
```

1 **Service** の定義。 このテンプレートにより作成されます。

注記

オブジェクト定義のメタデータに **namespace** フィールドの固定値が含まれる場合、フィールドはテンプレートのインスタンス化の際に定義から取り除かれます。**namespace** フィールドにパラメーター参照が含まれる場合には、通常のパラメーター置換が行われ、パラメーターの置換による値の解決が実行された namespace で、オブジェクトが作成されます。 この場合、ユーザーは対象の namespace でオブジェクトを作成するパーミッションがあることが前提になります。

4.7.5. テンプレートをバインド可能としてマーキングする

テンプレートサービスブローカーは、認識されているテンプレートオブジェクトごとに、カタログ内にサービスを 1 つ公開します。デフォルトでは、これらのサービスはそれぞれ「バインド可能」として公開され、エンドユーザーがプロビジョニングしたサービスに対してバインドできるようにします。

手順

テンプレートの作成者は、エンドユーザーが指定テンプレートからプロビジョニングされたサービスに対してバインディングすることを防ぐことができます。

- **template.openshift.io/bindable: "false"** のアノテーションをテンプレートに追加して、エンドユーザーが指定のテンプレートからプロビジョニングされるサービスをバインドできないようにできます。

4.7.6. テンプレートオブジェクトフィールドの公開

テンプレートの作成者は、テンプレートに含まれる特定のオブジェクトのフィールドを公開すべきかどうかを指定できます。テンプレートサービスブローカーは、ConfigMap、Secret、Service、Route オブジェクトに公開されたフィールドを認識し、ユーザーがブローカーでサポートされているサービスをバインドする際に公開されたフィールドの値を返します。

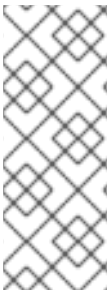
オブジェクトのフィールドを 1 つまたは複数公開するには、テンプレート内のオブジェクトに、プレフィックスが **template.openshift.io/expose-** または **template.openshift.io/base64-expose-** のアノテーションを追加します。

各アノテーションキーは、**bind** 応答のキーになるように、プレフィックスが削除されてパススルーされます。

各アノテーションの値は Kubernetes JSONPath 式の値であり、バインド時に解決され、**bind** 応答で返される値が含まれるオブジェクトフィールドを指定します。

注記

Bind 応答のキー/値のペアは、環境変数として、システムの他の場所で使用できます。そのため、アノテーションキーでプレフィックスを取り除いた値を有効な環境変数名として使用することが推奨されます。先頭に**A-Z**、**a-z** または _ を指定して、その後に、ゼロか、他の文字 **A-Z**、**a-z**、**0-9** または _ を指定してください。



注記

バックスラッシュでエスケープしない限り、Kubernetes の JSONPath 実装は表現内のどの場所に使用されていても、**.**、**@** などはメタ文字として解釈します。そのため、たとえば、**my.key** という名前の **ConfigMap** のデータを参照するには、JSONPath 式は **{.data['my\.key']}** とする必要があります。JSONPath 式が YAML でどのように記述されているかによって、**"{.data['my\\.key']}"** などのように、追加でバックスラッシュが必要になる場合があります。

以下は、公開されるさまざまなオブジェクトのフィールドの例です。

```
kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
- kind: ConfigMap
  apiVersion: v1
  metadata:
    name: my-template-config
    annotations:
      template.openshift.io/expose-username: "{.data['my\\.username']}"
  data:
    my.username: foo
- kind: Secret
  apiVersion: v1
  metadata:
    name: my-template-config-secret
    annotations:
      template.openshift.io/base64-expose-password: "{.data['password']}"
  stringData:
```

```
    password: bar
- kind: Service
  apiVersion: v1
  metadata:
    name: my-template-service
    annotations:
      template.openshift.io/expose-service_ip_port: "{.spec.clusterIP}:{.spec.ports[?
(.name==\"web\")].port}"
  spec:
    ports:
    - name: "web"
      port: 8080
- kind: Route
  apiVersion: v1
  metadata:
    name: my-template-route
    annotations:
      template.openshift.io/expose-uri: "http://{.spec.host}{.spec.path}"
  spec:
    path: mypath
```

上記の部分的なテンプレートでの **bind** 操作に対する応答例は以下のようになります。

```
{
  "credentials": {
    "username": "foo",
    "password": "YmFy",
    "service_ip_port": "172.30.12.34:8080",
    "uri": "http://route-test.router.default.svc.cluster.local/mypath"
  }
}
```

手順

- **template.openshift.io/expose-** アノテーションを使用して、値を文字列として返します。これは、任意のバイナリーデータを処理しないものの、便利な方法です。
- バイナリーデータを返す必要がある場合、**template.openshift.io/base64-expose-** アノテーションを使用して、データが返される前にデータを base64 でエンコードします。

4.7.7. テンプレートの準備ができるまで待機する

テンプレートの作成者は、テンプレート内の特定のオブジェクトがサービスカタログ、Template Service Broker または TemplateInstance API によるテンプレートのインスタンス化が完了したとされるまで待機する必要があるかを指定できます。

この機能を使用するには、テンプレート内の **Build**、**BuildConfig**、**Deployment**、**DeploymentConfig**、**Job** または **StatefulSet** のオブジェクト 1 つ以上に、次のアノテーションでマークを付けてください。

```
"template.alpha.openshift.io/wait-for-ready": "true"
```

テンプレートのインスタンス化は、アノテーションのマークが付けられたすべてのオブジェクトが準備できたと報告されるまで、完了しません。同様に、アノテーションが付けられたオブジェクトが失敗したと報告されるか、固定タイムアウトである 1 時間以内にテンプレートの準備が整わなかった場合に、

テンプレートのインスタンス化は失敗します。

インスタンス化の目的で、各オブジェクトの種類の準備状態および失敗は以下のように定義されます。

種類	準備状態 (Readines)	失敗 (Failure)
Build	オブジェクトが Complete (完了) フェーズを報告する	オブジェクトが Canceled (キャンセル)、Error (エラー)、または Failed (失敗) を報告する
BuildConfig	関連付けられた最新のビルドオブジェクトが Complete (完了) フェーズを報告する	関連付けられた最新のビルドオブジェクトが Canceled (キャンセル)、Error (エラー)、または Failed (失敗) を報告する
Deployment	オブジェクトが新しい ReplicaSet やデプロイメントが利用可能であることを報告する (これはオブジェクトに定義された readiness プローブに従います)	オブジェクトで、Progressing (進捗中) の状態が false であると報告される
DeploymentConfig	オブジェクトが新しい ReplicaController やデプロイメントが利用可能であると報告する (これはオブジェクトに定義された readiness プローブに従います)	オブジェクトで、Progressing (進捗中) の状態が false であると報告される
Job	オブジェクトが完了 (completion) を報告する	オブジェクトが 1 つ以上の失敗が発生したことを報告する
StatefulSet	オブジェクトがすべてのレプリカが準備状態にあることを報告する (これはオブジェクトに定義された readiness プローブに従います)	該当なし

以下は、テンプレートサンプルを一部抜粋したものです。この例では、**wait-for-ready** アノテーションが使用されています。他のサンプルは、OpenShift クイックスタートテンプレートにあります。

```
kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
- kind: BuildConfig
  apiVersion: v1
  metadata:
    name: ...
    annotations:
      # wait-for-ready used on BuildConfig ensures that template instantiation
      # will fail immediately if build fails
      template.alpha.openshift.io/wait-for-ready: "true"
  spec:
    ...
- kind: DeploymentConfig
  apiVersion: v1
```

```
  metadata:
    name: ...
    annotations:
      template.alpha.openshift.io/wait-for-ready: "true"
  spec:
    ...
- kind: Service
  apiVersion: v1
  metadata:
    name: ...
  spec:
    ...
```

その他の推奨事項

- アプリケーションにスムーズに実行するのに十分なリソースが提供されるようにメモリー、CPU、およびストレージのデフォルトサイズを設定します。
- **latest** タグが複数のメジャーバージョンで使用されている場合には、イメージからこのタグを参照しないようにします。新規イメージがそのタグにプッシュされると、実行中のアプリケーションが破損してしまう可能性があります。
- 適切なテンプレートの場合、テンプレートのデプロイ後に変更する必要なしに、ビルドおよびデプロイが正常に行われます。

4.7.8. 既存オブジェクトからのテンプレートの作成

テンプレートをゼロから作成するのではなく、プロジェクトから既存のオブジェクトを YAML 形式でエクスポートして、パラメーターを追加したり、テンプレート形式としてカスタマイズしたりして、YAML 形式を変更することもできます。

手順

1. オブジェクトを YAML 形式でプロジェクトにエクスポートします。

   ```
   $ oc get -o yaml --export all > <yaml_filename>
   ```

   all ではなく、特定のリソースタイプや複数のリソースを置き換えることも可能です。他の例については、**oc get -h** を実行してください。

   以下は、**oc get --export all** に含まれるオブジェクトタイプです。

   - BuildConfig
   - Build
   - DeploymentConfig
   - ImageStream
   - Pod
   - ReplicationController
   - Route

- Service

第5章 RUBY ON RAILS の使用

Ruby on Rails は Ruby で記述される Web フレームワークです。本書では、OpenShift Dedicated での Rails 4 の使用について説明します。

警告

チュートリアル全体をチェックして、OpenShift Dedicated でアプリケーションを実行するために必要なすべての手順を概観してください。問題に直面した場合には、チュートリアル全体を振り返り、もう一度問題に対応してください。またチュートリアルは、実行済みの手順を確認し、すべての手順が適切に実行されていることを確認するのに役立ちます。

前提条件

- Ruby および Rails の基本知識
- Ruby 2.0.0+、Rubygems、Bundler のローカルにインストールされたバージョン
- Git の基本知識
- OpenShift Dedicated 4 の実行中のインスタンス
- OpenShift Dedicated のインスタンスが実行中であり、利用可能であることを確認してください。さらに、**oc** CLI クライアントがインストールされており、コマンドがコマンドシェルからアクセスできることを確認し、メールアドレスおよびパスワードを使用してログインする際にこれを使用できるようにします。

5.1. データベースの設定

Rails アプリケーションはほぼ常にデータベースと併用されます。ローカル開発の場合は、PostgreSQL データベースを使用します。

手順

1. データベースをインストールします。

```
$ sudo yum install -y postgresql postgresql-server postgresql-devel
```

2. データベースを初期化します。

```
$ sudo postgresql-setup initdb
```

このコマンドで **/var/lib/pgsql/data** ディレクトリーが作成され、このディレクトリーにデータが保存されます。

3. データベースを起動します。

```
$ sudo systemctl start postgresql.service
```

4. データベースが実行されたら、**rails** ユーザーを作成します。

```
$ sudo -u postgres createuser -s rails
```

作成をしたユーザーのパスワードは作成されていない点に留意してください。

5.2. アプリケーションの作成

Rails アプリケーションをゼロからビルドするには、Rails gem を先にインストールする必要があります。その後に、アプリケーションを作成することができます。

手順

1. Rails gem をインストールします。

```
$ gem install rails
Successfully installed rails-4.3.0
1 gem installed
```

2. Rails gem のインストール後に、PostgreSQL をデータベースとして 指定して新規アプリケーションを作成します。

```
$ rails new rails-app --database=postgresql
```

3. 新規アプリケーションディレクトリーに切り替えます。

```
$ cd rails-app
```

4. アプリケーションがすでにある場合には **pg** (postgresql) gem が **Gemfile** に配置されていることを確認します。配置されていない場合には、gem を追加して **Gemfile** を編集します。

```
gem 'pg'
```

5. すべての依存関係を含む **Gemfile.lock** を新たに生成します。

```
$ bundle install
```

6. **pg** gem で **postgresql** データベースを使用するほか、**config/database.yml** が **postgresql** アダプターを使用していることを確認する必要があります。
config/database.yml ファイルの **default** セクションを以下のように更新するようにしてください。

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: 5
  host: localhost
  username: rails
  password:
```

7. アプリケーションの開発およびテスト用のデータベースを作成します。

```
$ rake db:create
```

これで PostgreSQL サーバーに **development** および **test** データベースが作成されます。

5.2.1. Welcome ページの作成

Rails 4 では静的な **public/index.html** ページが実稼働環境で提供されなくなったので、新たに root ページを作成する必要があります。

Welcome ページをカスタマイズするには、以下の手順を実行する必要があります。

- index アクションで コントローラー を作成します。
- **welcome** コントローラー **index** アクションの ビュー ページを作成します。
- 作成した コントローラー と ビュー と共にアプリケーションの root ページを提供する ルート を作成します。

Rails には、これらの必要な手順をすべて実行するジェネレーターがあります。

手順

1. Rails ジェネレーターを実行します。

```
$ rails generate controller welcome index
```

すべての必要なファイルが作成されます。

2. 以下のように **config/routes.rb** ファイルの 2 行目を編集します。

```
root 'welcome#index'
```

3. rails server を実行して、ページが利用できることを確認します。

```
$ rails server
```

ブラウザーで http://localhost:3000 に移動してページを表示してください。このページが表示されない場合は、サーバーに出力されるログを確認してデバッグを行ってください。

5.2.2. OpenShift Dedicated のアプリケーションの設定

アプリケーションが OpenShift Dedicated Platform で実行中の PostgreSQL データベースサービスと通信できるようにするには、データベースサービスの作成時に定義する環境変数を使用できるように **config/database.yml** の **default** セクションを編集する必要があります。

手順

- 以下のように事前に定義した変数で、**config/database.yml** の **default** セクションを編集します。

```
<% user = ENV.key?("POSTGRESQL_ADMIN_PASSWORD") ? "root" :
ENV["POSTGRESQL_USER"] %>
<% password = ENV.key?("POSTGRESQL_ADMIN_PASSWORD") ?
ENV["POSTGRESQL_ADMIN_PASSWORD"] : ENV["POSTGRESQL_PASSWORD"] %>
```

```
<% db_service = ENV.fetch("DATABASE_SERVICE_NAME","").upcase %>

default: &default
  adapter: postgresql
  encoding: unicode
  # For details on connection pooling, see rails configuration guide
  # http://guides.rubyonrails.org/configuring.html#database-pooling
  pool: <%= ENV["POSTGRESQL_MAX_CONNECTIONS"] || 5 %>
  username: <%= user %>
  password: <%= password %>
  host: <%= ENV["#{db_service}_SERVICE_HOST"] %>
  port: <%= ENV["#{db_service}_SERVICE_PORT"] %>
  database: <%= ENV["POSTGRESQL_DATABASE"] %>
```

5.2.3. アプリケーションの Git への保存

通常 OpenShift Dedicated でアプリケーションをビルドする場合、ソースコードを git リポジトリーに保存する必要があるため、**git** がない場合にはインストールしてください。

前提条件

- git をインストールします。

手順

1. **ls -1** コマンドを実行して、Rails アプリケーションのディレクトリーで操作を行っていることを確認します。コマンドの出力は以下のようになります。

```
$ ls -1
app
bin
config
config.ru
db
Gemfile
Gemfile.lock
lib
log
public
Rakefile
README.rdoc
test
tmp
vendor
```

2. Rails app ディレクトリーで以下のコマンドを実行して、コードを初期化し、git にコミットします。

```
$ git init
$ git add .
$ git commit -m "initial commit"
```

+ アプリケーションがコミットされたら、これをリモートリポジトリーにプッシュする必要があります。新規リポジトリーを作成する GitHub アカウントです。

1. お使いの **git** リポジトリーを参照するリモートを設定します。

```
$ git remote add origin git@github.com:<namespace/repository-name>.git
```

2. アプリケーションをリモートの git リポジトリーにプッシュします。

```
$ git push
```

5.3. アプリケーションの OPENSHIFT DEDICATED へのデプロイ

OpenShift Dedicated にアプリケーションをデプロイすることができます。

rails-app プロジェクトの作成後、新規プロジェクトの namespace に自動的に切り替えられます。

OpenShift Dedicated へのアプリケーションのデプロイでは 3 つの手順を実行します。

- OpenShift Dedicated の PostgreSQL イメージからデータベースサービスを作成します。
- データベースサービスと連動する OpenShift Dedicated の Ruby 2.0 ビルダーイメージおよび Ruby on Rails ソースコードのフロントエンドサービスを作成します。
- アプリケーションのルートを作成します。

手順

- Ruby on Rails アプリケーションをデプロイするには、アプリケーション用に新規のプロジェクトを作成します。

```
$ oc new-project rails-app --description="My Rails application" --display-name="Rails
Application"
```

5.3.1. データベースサービスの作成

Rails アプリケーションには実行中のデータベースサービスが必要です。このサービスには、PostgeSQL データベースイメージを使用します。

データベースサービスを作成するために、**oc new-app** コマンドを使用します。このコマンドには、必要な環境変数を渡す必要があります。この環境変数は、データベースコンテナー内で使用します。これらの環境変数は、ユーザー名、パスワード、およびデータベースの名前を設定するために必要です。これらの環境変数の値を任意の値に変更できます。変数は以下のようになります。

- POSTGRESQL_DATABASE
- POSTGRESQL_USER
- POSTGRESQL_PASSWORD

これらの変数を設定すると、以下を確認できます。

- 指定の名前のデータベースが存在する
- 指定の名前のユーザーが存在する
- ユーザーは指定のパスワードで指定のデータベースにアクセスできる

手順

1. データベースサービスを作成します。

```
$ oc new-app postgresql -e POSTGRESQL_DATABASE=db_name -e
POSTGRESQL_USER=username -e POSTGRESQL_PASSWORD=password
```

データベース管理者のパスワードを設定するには、直前のコマンドに以下を追加します。

```
-e POSTGRESQL_ADMIN_PASSWORD=admin_pw
```

2. 進行状況を確認します。

```
$ oc get pods --watch
```

5.3.2. フロントエンドサービスの作成

アプリケーションを OpenShift Dedicated にデプロイするには、アプリケーションが置かれるリポジトリーを指定する必要があります。

手順

1. フロントエンドサービスを作成し、データベースサービスの作成時に設定されたデータベース関連の環境変数を指定します。

```
$ oc new-app path/to/source/code --name=rails-app -e POSTGRESQL_USER=username -e
POSTGRESQL_PASSWORD=password -e POSTGRESQL_DATABASE=db_name -e
DATABASE_SERVICE_NAME=postgresql
```

このコマンドでは、OpenShift Dedicated は指定された環境変数を使用してソースコードの取得、ビルドのセットアップ、アプリケーションイメージのビルド、新規に作成されたイメージのデプロイを実行します。このアプリケーションには **rails-app** という名前を指定します。

2. **rails-app** DeploymentConfig の JSON ドキュメントを参照して、環境変数が追加されたかどうかを確認できます。

```
$ oc get dc rails-app -o json
```

以下のセクションが表示されるはずです。

```
env": [
    {
        "name": "POSTGRESQL_USER",
        "value": "username"
    },
    {
        "name": "POSTGRESQL_PASSWORD",
        "value": "password"
    },
    {
        "name": "POSTGRESQL_DATABASE",
        "value": "db_name"
    },
    {
```

```
        "name": "DATABASE_SERVICE_NAME",
        "value": "postgresql"
    }

],
```

3. ビルドプロセスを確認します。

```
$ oc logs -f build/rails-app-1
```

4. ビルドが完了すると、OpenShift Dedicated で Pod が実行されていることを確認します。

```
$ oc get pods
```

myapp-<number>-<hash> で始まる行が表示されますが、これは OpenShift Dedicated で実行中のアプリケーションです。

5. データベースの移行スクリプトを実行してデータベースを初期化してからでないと、アプリケーションは機能しません。これを実行する 2 種類の方法があります。

- 実行中のフロントエンドコンテナーから手動で実行する
  - **rsh** コマンドでフロントエンドコンテナーに exec を実行します。

    ```
    $ oc rsh <FRONTEND_POD_ID>
    ```

  - コンテナー内から移行を実行します。

    ```
    $ RAILS_ENV=production bundle exec rake db:migrate
    ```

    development または **test** 環境で Rails アプリケーションを実行する場合には、**RAILS_ENV** の環境変数を指定する必要はありません。

- デプロイメント前のライフサイクルフックをテンプレートに追する

5.3.3. アプリケーションのルートの作成

アプリケーションのルートを作成するためにサービスを公開できます。

手順

- **www.example.com** などの外部からアクセスできるホスト名を指定してサービスを公開するには、OpenShift Dedicated のルートを使用します。この場合は、以下を入力してフロントエンドサービスを公開する必要があります。

```
$ oc expose service rails-app --hostname=www.example.com
```

警告

指定するホスト名がルーターの IP アドレスに解決することを確認します。