



Red Hat Fuse 7.9

Fuse on OpenShift ガイド

Red Hat Fuse on OpenShift のインストールおよび管理、OpenShift での Fuse アプリケーションの開発およびデプロイ

Red Hat Fuse 7.9 Fuse on OpenShift ガイド

Red Hat Fuse on OpenShift のインストールおよび管理、OpenShift での Fuse アプリケーションの開発およびデプロイ

法律上の通知

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

Fuse on OpenShift の使用ガイド

目次

多様性を受け入れるオープンソースの強化	5
第1章 作業を始める前に	6
1.1. FUSE スタンドアロンと FUSE ON OPENSIFT の比較	6
第2章 管理者向けの基本情報	8
2.1. コンテナイメージの REGISTRY.REDHAT.IO を使用した認証	8
2.2. OPENSIFT 4.X サーバーでの FUSE イメージストリームおよびテンプレートのインストール	9
2.3. OPENSIFT 4.X への API DESIGNER のインストール	12
2.4. OPENSIFT 4.X での FUSE CONSOLE の設定	16
2.5. OPENSIFT で FUSE アプリケーションを監視するための PROMETHEUS の設定	27
2.6. FUSE ON OPENSIFT のメータリングの使用	31
2.7. カスタム GRAFANA ダッシュボードでの FUSE ON OPENSIFT の監視	33
2.8. OPENSIFT 3.X サーバーでの FUSE イメージストリームおよびテンプレートのインストール	37
第3章 制限された環境での FUSE ON OPENSIFT のインストール	43
3.1. 内部 DOCKER レジストリーの設定	43
3.2. 内部レジストリーのシークレットの設定	44
3.3. 制限された環境での FUSE ON OPENSIFT イメージのインストール	44
3.4. 内部 MAVEN リポジトリの使用	46
第4章 管理者でないユーザーでの FUSE ON OPENSIFT のインストール	48
4.1. 管理者でないユーザーでの FUSE ON OPENSIFT イメージおよびテンプレートのインストール	48
第5章 開発者向けの基本情報	51
5.1. 開発環境の準備	51
5.2. FUSE ON OPENSIFT でのアプリケーションの作成およびデプロイ	53
第6章 SPRING BOOT イメージのアプリケーションの開発	61
6.1. MAVEN ARCHETYPE を使用した SPRING BOOT 2 プロジェクトの作成	61
6.2. CAMEL SPRING BOOT アプリケーションの構造	62
6.3. SPRING BOOT 2 ARCHETYPE カタログ	64
6.4. SPRING BOOT の BOM ファイル	65
6.5. BOM ファイルの組み込み	66
6.6. SPRING BOOT MAVEN プラグイン	67
第7章 SPRING BOOT での APACHE CAMEL アプリケーションの実行	68
7.1. CAMEL SPRING BOOT コンポーネント	68
7.2. CAMEL SPRING BOOT スターターモジュール	68
7.3. スターターモジュールのない CAMEL コンポーネント一覧	69
7.4. CAMEL SPRING BOOT スターターの使用	69
7.5. SPRING BOOT の CAMEL コンテキストの自動設定	70
7.6. SPRING BOOT アプリケーションでの CAMEL ルートの自動検出	71
7.7. CAMEL SPRING BOOT AUTO CONFIGURATION の CAMEL プロパティの設定	72
7.8. カスタム CAMEL コンテキストの設定	72
7.9. 自動設定された CAMELCONTEXT での JMX の無効化	73
7.10. 自動設定されたコンシューマーおよびプロデューサーテンプレートの SPRING 管理 BEAN へのインジェクト	73
7.11. SPRING コンテキストの自動設定された TYPECONVERTER	73
7.12. SPRING タイプコンバージョン API ブリッジ	74
7.13. タイプ変換機能の無効化	74
7.14. 自動設定の XML ルートのクラスパスへの追加	75
7.15. 自動設定の XML REXT-DSL ルートの追加	75

7.16. CAMEL SPRING BOOT でのテスト	76
第8章 FUSE ON OPENSIFT 上での SPRING BOOT 2 用 SOAP TO RESTブリッジクイックスタートの実行	78
第9章 XA トランザクションを使用した SPRING BOOT での CAMEL サービスの実行	84
9.1. STATEFULSET リソース	84
9.2. SPRING BOOT NARAYANA リカバリーコントローラー	84
9.3. SPRING BOOT NARAYANA リカバリーコントローラーの設定	84
9.4. OPENSIFT での CAMEL SPRING BOOT XA クイックスタートの実行	85
9.5. 成功した XA トランザクションのテスト	87
9.6. 失敗した XA トランザクションのテスト	87
第10章 CAMEL アプリケーションの A-MQ ブローカーとの統合	88
10.1. SPRING BOOT CAMEL A-MQ クイックスタートのビルドおよびデプロイ	88
第11章 SPRING BOOT と KUBERNETES の統合	90
11.1. SPRING BOOT の外部化設定	90
11.2. CONFIGMAP プロパティソースのチュートリアルの実行	91
11.3. CONFIGMAP PROPERTYSOURCE の使用	99
11.4. SECRETS PROPERTYSOURCE の使用	100
11.5. PROPERTYSOURCE RELOAD の使用	102
第12章 KARAF イメージのアプリケーションの開発	106
12.1. MAVEN ARCHETYPE を使用した KARAF プロジェクトの作成	106
12.2. CAMEL KARAF アプリケーションの構造	107
12.3. KARAF ARCHETYPE カタログ	108
12.4. FABRIC8 KARAF 機能の使用	108
12.5. FABRIC8 KARAF CONFIG 管理サポートの追加	113
12.6. FABRIC8 KARAF BLUEPRINT サポートの追加	116
12.7. FABRIC8 KARAF ヘルスチェックの有効化	117
12.8. カスタムヘルスチェックの追加	118
第13章 JBOSS EAP イメージのアプリケーションの開発	120
13.1. S2I ソースワークフローを使用した JBOSS EAP プロジェクトの作成	120
13.2. JBOSS EAP アプリケーションの構造	122
13.3. JBOSS EAP クイックスタートテンプレート	122
第14章 FUSE ON OPENSIFT での永続ストレージの使用	124
14.1. ボリュームおよびボリュームタイプ	124
14.2. PERSISTENTVOLUMES	124
14.3. 永続ボリュームの設定	124
14.4. PERSISTENTVOLUMECLAIMS の作成	125
14.5. POD での永続ボリュームの使用	125
第15章 FUSE ON OPENSIFT のパッチ適用	127
15.1. BOM および MAVEN 依存関係に関する重要事項	127
15.2. FUSE ON OPENSIFT イメージのパッチ適用	127
15.3. FUSE ON OPENSIFT テンプレートのパッチ適用	129
15.4. BOM を使用したアプリケーション依存関係のパッチ適用	130
15.5. 利用可能な BOM バージョン	133
付録A SPRING BOOT MAVEN プラグイン	134
A.1. SPRING BOOT MAVEN プラグインのゴール	134
A.2. SPRING BOOT MAVEN プラグインの使用	134
付録B KARAF MAVEN プラグインの使用	137

B.1. MAVEN 依存関係	137
B.2. KARAF MAVEN プラグインの設定	137
B.3. カスタマイズされた KARAF アセンブリー	138
付録C OPENSIFT MAVEN プラグイン	140
C.1. OPENSIFT MAVEN プラグインについて	140
C.2. イメージのビルド	140
C.3. KUBERNETES および OPENSIFT リソース	140
C.4. OPENSIFT MAVEN プラグインのインストール	141
C.5. OPENSIFT MAVEN プラグインのビルドゴールについて	142
C.6. OPENSIFT MAVEN プラグインの開発ゴールについて	142
付録D FABRIC8 MAVEN プラグイン	144
D.1. イメージのビルド	144
D.2. KUBERNETES および OPENSIFT リソース	144
D.3. プラグインのインストール	145
D.4. FABRIC8 MAVEN プラグインのゴールの理解	145
D.5. ジェネレーター	147
付録E FABRIC8 CAMEL MAVEN プラグイン	152
E.1. FABRIC8 CAMEL MAVEN プラグインのゴール	152
E.2. FABRIC8-CAMEL-MAVEN プラグインのプロジェクトへの追加	152
E.3. 任意の MAVEN プロジェクトでのゴール実行	153
E.4. オプション	154
E.5. インクルードテストの検証	155
付録F JVM 環境変数のカスタマイズ	157
F.1. OPENJDK 8 での S2I JAVA ビルダーイメージの使用	157
F.2. OPENJDK 8 での S2I KARAF ビルダーイメージの使用	157
F.3. ビルド時の環境変数	157
F.4. ランタイムの環境変数	158
F.5. JOLOKIA の設定	159
付録G JVM のチューニングによる LINUX コンテナ内での実行	160
G.1. JVM のチューニング	160
G.2. FUSE ON OPENSIFT イメージのデフォルト動作	160
G.3. FUSE ON OPENSIFT イメージのカスタムチューニング	160
G.4. サードパーティーライブラリーのチューニング	161

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、[Red Hat CTO である Chris Wright のメッセージ](#)をご覧ください。

Red Hat Fuse on OpenShift は、OpenShift Container Platform での Fuse アプリケーションのデプロイを可能にします。

第1章 作業を始める前に

リリースノート

本リリースに関する重要な情報は、[リリースノート](#) を参照してください。

バージョンの互換性とサポート

バージョンの互換性とサポートの詳細は、[Red Hat Fuse でサポートされる設定](#) を参照してください。

Windows O/S のサポート

Fuse on OpenShift の開発者ツール (**oc** クライアントおよび Container Development Kit) は、Windows O/S で完全サポートされます。Linux コマンドライン構文で示された例は、Windows コマンドライン構文にしたがって適切に変更すれば、Windows O/S で動作します。

1.1. FUSE スタンドアロンと FUSE ON OPENSIFT の比較

主な機能の違いは次のとおりです。

- Fuse on OpenShift のアプリケーションデプロイメントは、1つのアプリケーションと、Docker イメージ内にパッケージ化された必要なすべてのランタイムコンポーネントで設定されます。アプリケーションはランタイム (実行環境) にデプロイされません。アプリケーションイメージ自体が実行可能なランタイム (実行環境) であり、OpenShift によりデプロイおよび管理されます。
- 各アプリケーションイメージが完全なランタイム環境であるため、OpenShift 環境でのパッチ適用は Fuse スタンドアロンとは異なります。パッチを適用する場合、アプリケーションイメージが再ビルドされ、OpenShift 内に再デプロイされます。主要の OpenShift 管理機能により、ローリングアップグレードやサイドバイサイドデプロイメントを使用して、アップグレード中にアプリケーションの可用性を維持することが可能になります。
- Fuse の Fabric によって提供されるプロビジョニングおよびクラスターリング機能は、Kubernetes および OpenShift の同等の機能に置き換えられました。OpenShift は、アプリケーションのデプロイメントおよびスケーリングの一環として、自動的に子コンテナを個別に作成および設定するため、手作業でこの作業を行う必要はありません。
- Fabric エンドポイントは OpenShift 環境内で使用されません。代わりに、Kubernetes サービスを使用する必要があります。
- メッセージングサービスは、OpenShift イメージの A-MQ を使用して作成および管理され、Karaf コンテナ内に直接含まれません。Fuse on OpenShift は、camel-amq コンポーネントの強化バージョンを提供し、Kubernetes を介して OpenShift のメッセージングサービスへのシームレスな接続を可能にします。
- Karaf を使用して稼働中の Karaf インスタンスをライブ更新しないようにしてください。アプリケーションコンテナを再起動またはスケールアップすると、更新は維持されません。これは、イミュータブルアーキテクチャーの基本原則であり、OpenShift 内でスケーラビリティと柔軟性を実現するために必要となります。
- Red Hat Fuse コンポーネントに直接リンクする Maven 依存関係は Red Hat によってサポートされます。ユーザーが導入したサードパーティー Maven 依存関係はサポートされません。
- SSH エージェントは Apache Karaf マイクロコンテナに含まれないため、bin/client コンソールクライアントを使用して接続できません。

- プロトコル互換性と Fuse on OpenShift アプリケーション内の Camel コンポーネント: HTTP ベースでない通信は、TLS および SNI を使用して、OpenShift 外部から Fuse サービス (Camel コンシューマーエンドポイント) にルーティング可能である必要があります。

第2章 管理者向けの基本情報

OpenShift の管理者は、以下を行って Fuse on OpenShift デプロイメントの OpenShift クラスターを準備できます。

1. **registry.redhat.io** での認証の設定。
2. Fuse on OpenShift イメージおよびテンプレートのインストール。

2.1. コンテナイメージの **REGISTRY.REDHAT.IO** を使用した認証

Fuse コンテナイメージを OpenShift にデプロイする前に、**registry.redhat.io** で認証を設定します。

前提条件

- OpenShift Container Platform クラスターへアクセスできるクラスター管理者権限。
- OpenShift **oc** クライアントツールがインストール済みであること。詳細は、[OpenShift CLI のドキュメント](#) を参照してください。

手順

1. 管理者として OpenShift クラスターにログインします。

```
oc login --user system:admin --token=my-token --server=https://my-cluster.example.com:6443
```

2. Fuse をデプロイするプロジェクトを開きます。

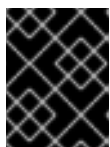
```
oc project myproject
```

3. Red Hat カスタマーポータルアカウントを使用して **docker-registry** シークレットを作成します。**PULL_SECRET_NAME** は作成するシークレットに置き換えます。

```
oc create secret docker-registry PULL_SECRET_NAME \
  --docker-server=registry.redhat.io \
  --docker-username=CUSTOMER_PORTAL_USERNAME \
  --docker-password=CUSTOMER_PORTAL_PASSWORD \
  --docker-email=EMAIL_ADDRESS
```

以下の出力が表示されるはずです。

```
secret/PULL_SECRET_NAME created
```



重要

この **docker-registry** シークレットを、**registry.redhat.io** に対して認証されるすべての OpenShift プロジェクト namespace に作成する必要があります。

4. シークレットをサービスアカウントにリンクして、シークレットをイメージをプルするために使用します。以下は、**default** サービスアカウントを使用する例になります。

```
oc secrets link default PULL_SECRET_NAME --for=pull
```

サービスアカウント名は、OpenShift Pod が使用する名前と一致する必要があります。

5. シークレットを **builder** サービスアカウントにリンクし、ビルドイメージをプッシュおよびプルするためにシークレットを使用します。

```
oc secrets link builder PULL_SECRET_NAME
```



注記

Red Hat のユーザー名とパスワードを使用してプルシークレットを作成したくない場合は、レジストリーサービスアカウントを使用して認証トークンを作成できます。

関連情報

コンテナイメージに対する Red Hat の認証に関する詳細は、以下を参照してください。

- [Red Hat コンテナイメージの認証](#)
- [Red Hat registry service accounts](#)

2.2. OPENSIFT 4.X サーバーでの FUSE イメージストリームおよびテンプレートのインストール

OpenShift Container Platform 4.x は、OpenShift namespace で操作する Samples Operator を使用して、Red Hat Enterprise Linux (RHEL) ベースの OpenShift Container Platform イメージストリームおよびテンプレートをインストールおよび更新します。Fuse on OpenShift イメージストリームおよびテンプレートをインストールするには、以下を行います。

- Samples Operator の再設定
- Fuse イメージストリームおよびテンプレートを **Skipped Imagestreams and Skipped Templates** フィールドに追加します。
 - Skipped Imagestreams (スキップされるイメージストリーム): Samples Operator のインベントリーにあるが、クラスター管理者の希望により Operator が無視または管理しないようにするイメージストリーム。
 - Skipped Templates (スキップされるテンプレート): Samples Operator のインベントリーにあるが、クラスター管理者の希望により Operator が無視または管理しないようにするテンプレート。

前提条件

- OpenShift サーバーにアクセスできる必要があります。
- **registry.redhat.io** への認証が設定されている。
- オプションで、インストール後に、OpenShift ダッシュボードに Fuse テンプレートを表示させる場合には、OpenShift ドキュメント (https://docs.openshift.com/container-platform/4.1/applications/service_brokers/installing-service-catalog.html) に記載されているよ

うに、サービスカタログとテンプレートサービスブローカーを先にインストールする必要があります。

手順

1. OpenShift 4 サーバーを起動します。
2. OpenShift サーバーに管理者としてログインします。

```
oc login -u system:admin
```

3. docker-registry シークレットを作成したプロジェクトを使用していることを確認します。

```
oc project openshift
```

4. Samples Operator の現在の設定を表示します。

```
oc get configs.samples.operator.openshift.io -n openshift-cluster-samples-operator -o yaml
```

5. Samples Operator が、追加された Fuse テンプレートおよびイメージストリームを無視するよう設定します。

```
oc edit configs.samples.operator.openshift.io -n openshift-cluster-samples-operator
```

6. Fuse imagestreams の Skipped Imagestreams セクションを追加し、Fuse および Spring Boot 2 のテンプレートを Skipped Templates セクションに追加します。

```
[...]
spec:
  architectures:
  - x86_64
  managementState: Managed
  skippedImagestreams:
  - fuse-console-rhel8
  - fuse-eap-openshift-jdk8-rhel7
  - fuse-eap-openshift-jdk11-rhel8
  - fuse-java-openshift-rhel8
  - fuse-java-openshift-jdk11-rhel8
  - fuse-karaf-openshift-rhel8
  - fuse-apicurito-generator-rhel8
  - fuse-apicurito-rhel8
  skippedTemplates:
  - s2i-fuse79-eap-camel-amq
  - s2i-fuse79-eap-camel-cdi
  - s2i-fuse79-eap-camel-cxf-jaxrs
  - s2i-fuse79-eap-camel-cxf-jaxws
  - s2i-fuse79-karaf-camel-amq
  - s2i-fuse79-karaf-camel-log
  - s2i-fuse79-karaf-camel-rest-sql
  - s2i-fuse79-karaf-cxf-rest
  - s2i-fuse79-spring-boot-2-camel-amq
  - s2i-fuse79-spring-boot-2-camel-config
  - s2i-fuse79-spring-boot-2-camel-drools
  - s2i-fuse79-spring-boot-2-camel-infinispan
```

```
- s2i-fuse79-spring-boot-2-camel-rest-3scale
- s2i-fuse79-spring-boot-2-camel-rest-sql
- s2i-fuse79-spring-boot-2-camel
- s2i-fuse79-spring-boot-2-camel-xa
- s2i-fuse79-spring-boot-2-camel-xml
- s2i-fuse79-spring-boot-2-cxf-jaxrs
- s2i-fuse79-spring-boot-2-cxf-jaxws
- s2i-fuse79-spring-boot-2-cxf-jaxrs-xml
- s2i-fuse79-spring-boot-2-cxf-jaxws-xml
```

7. Fuse on OpenShift イメージストリームをインストールします。

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005
```

```
oc create -n openshift -f ${BASEURL}/fis-image-streams.json
```



注記

Error from server (AlreadyExists): imagestreams.image.openshift.io
<imagestreamname> already exists というメッセージでエラーが表示された場合、以下のコマンドを使用して、既存のイメージストリームを最新のものに置き換えます。

```
oc replace --force -n openshift -f ${BASEURL}/fis-image-streams.json
```

8. Fuse on OpenShift のクイックスタートテンプレートをインストールします。

```
for template in eap-camel-amq-template.json \
eap-camel-cdi-template.json \
eap-camel-cxf-jaxrs-template.json \
eap-camel-cxf-jaxws-template.json \
karaf-camel-amq-template.json \
karaf-camel-log-template.json \
karaf-camel-rest-sql-template.json \
karaf-cxf-rest-template.json ;
do
oc create -n openshift -f \
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/quickstarts/${template}
done
```

9. Spring Boot 2 のクイックスタートテンプレートをインストールします。

```
for template in spring-boot-2-camel-amq-template.json \
spring-boot-2-camel-config-template.json \
spring-boot-2-camel-drools-template.json \
spring-boot-2-camel-infinispan-template.json \
spring-boot-2-camel-rest-3scale-template.json \
spring-boot-2-camel-rest-sql-template.json \
spring-boot-2-camel-template.json \
spring-boot-2-camel-xa-template.json \
spring-boot-2-camel-xml-template.json \
```

```

spring-boot-2-cxf-jaxrs-template.json \
spring-boot-2-cxf-jaxws-template.json \
spring-boot-2-cxf-jaxrs-xml-template.json \
spring-boot-2-cxf-jaxws-xml-template.json ;
do oc create -n openshift -f \
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-
2.1.0.fuse-sb2-790047-redhat-00005/quickstarts/${template}
done

```

10. (任意手順) インストールされた Fuse on OpenShift テンプレートを表示します。

```
oc get template -n openshift
```

2.3. OPENSIFT 4.X への API DESIGNER のインストール

Red Hat Fuse on OpenShift では、REST API の設計に使用できる Web ベースの API デザイナーツールである API Designer が提供されます。API Designer Operator を使用すると、OpenShift Container Platform 4.x で API Designer を簡単にインストールおよびアップグレードできます。

OpenShift 管理者として、API Designer Operator を OpenShift プロジェクト (namespace) にインストールします。Operator はインストール後に、選択された namespace で実行されます。しかし、API Designer をサービスとして使用できるようにするには、OpenShift 管理者または開発者が API Designer のインスタンスを作成する必要があります。API Designer サービスによって、API Designer Web コンソールにアクセスするための URL が提供されます。

前提条件

- OpenShift クラスターの管理者権限を持っている必要があります。
- **registry.redhat.io** への認証が設定されている。

手順

1. OpenShift 4.x サーバーを起動します。
2. Web ブラウザーで OpenShift コンソールに移動します。ご自分のクレデンシャルでコンソールにログインします。
3. **Operators** をクリックした後、**OperatorHub** をクリックします。
4. 検索フィールドに **API Designer** と入力します。
5. **Red Hat Integration - API Designer** カードをクリックします。**Red Hat Integration - API Designer Operator** のインストールページが表示されます。
6. **Install** をクリックします。**Install Operator** ページが開きます。
 - a. **Update Channel** には、**fuse-console-7.12.x** を選択します。
 - b. **Installation mode** では、クラスターの namespace のリストから namespace (プロジェクト) を1つ選択します。
 - c. **Approval Strategy** では、**Automatic** または **Manual** を選択し、API Designer Operator の更新が OpenShift によってどのように対処されるかを設定します。

- **Automatic** (自動) 更新を選択した場合、新しいバージョンの API Designer Operator が使用できるようになると、人的な介入なしで OpenShift Operator Lifecycle Manager (OLM) によって、API Designer の稼働中のインスタンスが自動的にアップグレードされます。
 - **Manual** (手動) 更新を選択した場合、Operator の新しいバージョンが使用できるようになると、OLM によって更新リクエストが作成されます。クラスター管理者は、更新リクエストを手動で承認して、API Designer Operator を新しいバージョンに更新する必要があります。
7. **Install** をクリックして、指定の namespace (プロジェクト) で API Designer Operator を使用できるようにします。
 8. API Designer がプロジェクトにインストールされていることを確認するには、**Operators** をクリックした後、**Installed Operators** をクリックし、リストに **Red Hat Integration - API Designer** があることを確認します。

次のステップ

API Designer Operator のインストール後、API Designer のインスタンスを作成し、API Designer をサービスとして OpenShift プロジェクトに追加する必要があります。このタスクを実行する方法は2つあります。

- OpenShift 管理者は「[API Designer をサービスとして OpenShift 4.x プロジェクトに追加](#)」の手順を実行できます。
- OpenShift 開発者は、[API の設計](#) に記載されている手順を実行できます。
API Designer サービスによって、API Designer Web コンソールにアクセスするための URL が提供されます。

2.3.1. API Designer をサービスとして OpenShift 4.x プロジェクトに追加

API Designer Operator を OpenShift 4.x プロジェクトにインストールした後、OpenShift 開発者はこれをサービスとして OpenShift プロジェクトに追加できます。開発者が API Designer Web コンソールへアクセスするために使用する URL は API Designer サービスによって提供されます。



注記

OpenShift 開発者が API Designer をサービスとして OpenShift 4.x プロジェクトに追加するための手順は、[API の設計](#) を参照してください。

前提条件

- OpenShift クラスターの管理者権限を持っている必要があります。
- API Designer Operator は現在の OpenShift プロジェクトにインストールされます。

手順

1. OpenShift Web コンソールで、**Operators** をクリックした後、**Installed Operators** をクリックします。
2. **Name** 列で、**Red Hat Integration - API Designer** をクリックします。
3. **Provided APIs** の **Create instance** をクリックします。

API Designer インスタンスの最小限の開始テンプレートがあるデフォルトのフォームが開かれます。デフォルト値を使用するか、任意でデフォルト値を編集します。

4. **Create** をクリックし、新しい **apicurito-service** を作成します。OpenShift は、新しい API Designer サービスの Pod、サービス、およびその他のコンポーネントを起動します。
5. API Designer サービスが使用できることを確認するには、以下を実行します。
 - a. **Operators** をクリックした後、**Installed Operators** をクリックします。
 - b. **Provided APIs** 列で **Apicurito CRD** をクリックします。
Operator Details ページに、**apicurito-service** のリストが表示されます。
6. API Designer を開くには、以下を実行します。
 - a. **Networking > Routes** の順に選択します。
 - b. 正しいプロジェクトが選択されていることを確認してください。
 - c. **apicurito-service-ui** 行の **Location** 列で、URL をクリックします。
API Designer Web コンソールが新規ブラウザタブで開きます。

2.3.2. OpenShift 4.x での API Designer のアップグレード

Red Hat OpenShift 4.x では、Red Hat Fuse Operator などの Operator の更新が処理されます。詳細は、[OpenShift ドキュメントの Operator](#) を参照してください。

また、Operator の更新によってアプリケーションのアップグレードがトリガーされる可能性があります。アプリケーションのアップグレードの実行方法は、アプリケーションの設定によって異なります。

API Designer アプリケーションでは、API Designer Operator をアップグレードすると、OpenShift によってクラスターの API Designer アプリケーションもアップグレードされます。



注記

API Designer 7.8 から API Designer 7.9 にアップグレードするときに、通常の Operator のアップグレードプロセスは動作しません。API Designer を Fuse 7.8 から Fuse 7.9 にアップグレードするには、7.8 API Designer Operator を削除してから 7.9 API Designer Operator をインストールする必要があります。

2.3.3. API Designer のメータリングラベル

OpenShift の Metering Operator を使用すると、インストールされた API Designer Operator、UI コンポーネント、およびコードジェネレーターを分析し、Red Hat サブスクリプションに準拠しているかどうかを判断することができます。詳細は [OpenShift のメータリング](#) を参照してください。

以下の表は、API Designer のメータリングラベルを示しています。

表2.1 API Designer のメータリングラベル

ラベル	使用できる値
com.company	Red_Hat
rht.prod_name	Red_Hat_Integration

ラベル	使用できる値
<code>rht.prod_ver</code>	7.9
<code>rht.comp</code>	Fuse
<code>rht.comp_ver</code>	7.9
<code>rht.subcomp</code>	fuse-apicurito apicurito-service-ui apicurito-service-generator
<code>rht.subcomp_t</code>	infrastructure

例

- API Designer **Operator** の例:

```
apicurito-operator
com.company: Red_Hat
rht.prod_name: Red_Hat_Integration
rht.prod_ver: 7.9
rht.comp: Fuse
rht.comp_ver: 7.9
rht.subcomp: fuse-apicurito
rht.subcomp_t: infrastructure
```

- API Designer **UI** コンポーネントの例:

```
com.company: Red_Hat
rht.prod_name: Red_Hat_Integration
rht.prod_ver: 7.9
rht.comp: Fuse
rht.comp_ver: 7.9
rht.subcomp: apicurito-service-ui
rht.subcomp_t: infrastructure
```

- API Designer **Generator** コンポーネントの例:

```
com.company: Red_Hat
rht.prod_name: Red_Hat_Integration
rht.prod_ver: 7.9
rht.comp: Fuse
rht.comp_ver: 7.9
rht.subcomp: apicurito-service-generator
rht.subcomp_t: infrastructure
```

2.3.4. 制限された環境で API Designer をインストールする場合の注意事項

制限された環境でインストールされる OpenShift クラスターは、デフォルトでは Red Hat が提供する OperatorHub ソースにアクセスできません。これは、OperatorHub ソースのリモートソースに完全なインターネット接続が必要であるためです。このような環境で API Designer Operator をインストールするには、以下の前提条件を満たす必要があります。

- Operator Lifecycle Manager (OLM) のデフォルトのリモート OperatorHub ソースを無効にします。
- 完全なインターネットアクセスのあるワークステーションを使用して、OperatorHub コンテンツのローカルミラーを作成します。
- OLM を、デフォルトのリモートソースからではなくローカルソースから Operator をインストールし、管理するように設定します。

詳細は、OpenShift ドキュメントの [ネットワークが制限された環境での Operator Lifecycle Manager の使用](#) を参照してください。OperatorHub のローカルミラーを作成したら、次のステップを実行できます。

- [OpenShift 4.x への API Designer のインストール](#) の説明にしたがって、ミラーリングされた OperatorHub を使用して API Designer をインストールします。
- [API Designer をサービスとして OpenShift 4.x プロジェクトに追加](#) の説明にしたがって、API Designer をサービスとしてサービスとして追加します。

2.4. OPENSIFT 4.X での FUSE CONSOLE の設定

OpenShift 4.x では、Fuse Console の設定に、インストール、およびデプロイが必要になります。Fuse Console のインストールおよびデプロイには、以下のオプションがあります。

- [「OperatorHub を使用した OpenShift 4.x での Fuse Console のインストールおよびデプロイ」](#)
特定の namespace で Fuse アプリケーションにアクセスするために、Fuse Console Operator を使用して Fuse Console をインストールおよびデプロイします。Operator は、Fuse Console のセキュリティ保護を処理します。
- [「コマンドラインを使用した OpenShift 4.x での Fuse Console のインストールおよびデプロイ」](#)
OpenShift クラスターの複数の namespace または特定の namespace にある Fuse アプリケーションにアクセスするために、コマンドラインと Fuse Console テンプレートの 1 つを使用して Fuse Console をインストールおよびデプロイします。デプロイ前にクライアント証明書を生成して、Fuse Console をセキュアにする必要があります。

任意で、[「OpenShift 4.x 上の Fuse Console のロールベースアクセス制御」](#) の説明通りに、Fuse Console のロールベースアクセス制御 (RBAC) をカスタマイズすることができます。

2.4.1. OperatorHub を使用した OpenShift 4.x での Fuse Console のインストールおよびデプロイ

OpenShift 4.x に Fuse Console をインストールするには、OpenShift OperatorHub で提供される Fuse Console Operator を使用します。Fuse Console をデプロイするには、インストールされた Operator のインスタンスを作成します。

前提条件

- [コンテナイメージの registry.redhat.io を使用した認証](#) で説明されているように、[registry.redhat.io](#) との認証を設定している。

- Fuse Console のロールベースアクセス制御 (RBAC) をカスタマイズする場合は、Fuse Console Operator のインストール先と同じ OpenShift namespace に RBAC 設定マップファイルが必要になる。[OpenShift 4.x の Fuse Console のロールベースアクセス制御](#) の説明にしたがって、デフォルトの RBAC 動作を使用する場合は、設定マップファイルを指定する必要はない。

手順

Fuse Console をインストールおよびデプロイするには、以下を行います。


1. Web ブラウザーで、**cluster admin** 権限を持つユーザーとして OpenShift コンソールにログインします。
2. **Operators** をクリックした後、**OperatorHub** をクリックします。
3. 検索フィールドウィンドウに **Fuse Console** と入力し、Operator のリストを絞り込みます。
4. **Fuse Console Operator** をクリックします。
5. Fuse Console Operator インストールウィンドウで **Install** をクリックします。
Create Operator Subscription フォームが表示されます。
 - **Update Channel** には **7.12.x** を選択します。
 - **Installation Mode** では、デフォルト (クラスターの特定の namespace) を受け入れます。
operator のインストール後に Fuse Console をデプロイする場合、クラスター上のすべての namespace でアプリケーションを監視するか、Fuse Console operator がインストールされた namespace のみでアプリケーションを監視するかを選択できる点に注意してください。
 - **Installed Namespace** には、Fuse Console Operator をインストールする namespace を選択します。
 - **Approval Strategy** では、**Automatic** または **Manual** を選択し、Fuse Console の更新が OpenShift によってどのように対処されるかを設定します。
 - **Automatic** (自動) 更新を選択した場合、新しいバージョンの Fuse Console Operator が使用できるようになると、人的な介入なしで OpenShift Operator Lifecycle Manager (OLM) によって、Fuse Console の稼働中のインスタンスが自動的にアップグレードされます。
 - **Manual** (手動) 更新を選択した場合、Operator の新しいバージョンが使用できるようになると、OLM によって更新リクエストが作成されます。クラスター管理者は、更新リクエストを手動で承認して、Fuse Console Operator を新しいバージョンに更新する必要があります。
6. **Install** をクリックします。
OpenShift によって、Fuse Console Operator が現在の namespace にインストールされます。
7. インストールを確認するには、**Operators** をクリックした後、**Installed Operators** をクリックします。Operator のリストに Fuse Console が表示されます。
8. OpenShift Web コンソールで Fuse Console をデプロイするには、以下を行います。
 - a. **Installed Operators** のリストで、**Name** 列の **Fuse Console** をクリックします。
 - b. **Provided APIs** の **Operator Details** ページで、**Create Instance** をクリックします。
設定のデフォルト値を使用するか、任意でデフォルト値を編集します。

Fuse Console のパフォーマンスを向上する必要がある場合 (高可用性環境など)、**Replicas** で Fuse Console に割り当てられた Pod 数を増やすことができます。

Rbac (ロールベースアクセス制御) では、デフォルトの RBAC 動作をカスタマイズする場合や、Fuse Console Operator をインストールした namespace に ConfigMap ファイルがすでに存在する場合にのみ、**config Map** フィールドに値を指定します。RBAC の詳細は、OpenShift 4.x 上の Fuse Console のロールベースアクセス制御 を参照してください。

- c. **Create** をクリックします。
Fuse Console Operator Details ページが開き、デプロイメントの状態が表示されます。

9. Fuse Console を開くには以下を行います。

- a. **namespace** デプロイメントの場合: OpenShift Web コンソールで、Fuse Console の operator をインストールしたプロジェクトを開き、**Overview** を選択します。**Project Overview** ページで **Launcher** セクションまで下方向にスクロールし、Fuse Console の URL をクリックして開きます。
cluster デプロイメントでは、OpenShift Web コンソールのタイトルバーで、グリッドアイコン () をクリックします。ポップアップメニューの **Red Hat アプリケーション** の下にある、Fuse Console の URL リンクをクリックします。
- b. Fuse Console にログインします。
 ブラウザーに **Authorize Access** ページが表示され、必要なパーミッションが表示されます。
- c. **Allow selected permissions** をクリックします。
 Fuse Console がブラウザーで開き、アクセス権限のある Fuse アプリケーション Pod が表示されます。

10. 表示するアプリケーションの **Connect** をクリックします。
 新しいブラウザーウィンドウが開かれ、Fuse Console にアプリケーションが表示されます。

2.4.2. コマンドラインを使用した OpenShift 4.x での Fuse Console のインストールおよびデプロイ

OpenShift 4.x では、次のデプロイメントオプションの1つを選択して、コマンドラインから Fuse Console をインストールおよびデプロイできます。

- **cluster** - Fuse Console は、OpenShift クラスターで複数の namespace (プロジェクト) 全体にデプロイされた Fuse アプリケーションを検索し、接続することができます。このテンプレートをデプロイするには、OpenShift クラスターの管理者ロールが必要です。
- **ロールベースアクセス制御のあるクラスター** - 設定可能なロールベースアクセス制御 (RBAC) のあるクラスターテンプレート。詳細は、OpenShift 4.x 上の Fuse Console のロールベースアクセス制御 を参照してください。
- **namespace** - Fuse Console は特定の OpenShift プロジェクト (namespace) にアクセスできます。このテンプレートをデプロイするには、OpenShift プロジェクトの管理者ロールが必要です。
- **ロールベースアクセス制御のある namespace** - 設定可能な RBAC のある namespace テンプレート。詳細は、OpenShift 4.x 上の Fuse Console のロールベースアクセス制御 を参照してください。

Fuse Console テンプレートのパラメーターリストを表示するには、以下の OpenShift コマンドを実行します。


```
oc process --parameters -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fuse-console-namespace-os4.json
```

前提条件

- Fuse Console をインストールおよびデプロイする前に、[OpenShift 4.x で Fuse Console をセキュア化するための証明書の生成](#)の説明どおりに、サービス署名認証局で署名されたクライアント証明書を生成する必要があります。
- OpenShift クラスターの **cluster admin** ロールが必要です。
- コンテナイメージの [registry.redhat.io](#) を使用した認証で説明されているように、[registry.redhat.io](#) との認証を設定している。
- [OpenShift 4.x サーバーでの Fuse イメージストリームおよびテンプレートのインストール](#)の説明どおりに Fuse Console イメージストリーム (およびその他の Fuse イメージストリーム) がインストールされている必要があります。

手順

1. 以下のコマンドを実行してすべてのテンプレートのリストを取得し、Fuse Console イメージストリームがインストールされていることを確認します。

```
oc get template -n openshift
```

2. 任意で、すでにインストールされているイメージストリームを新しいリリースタグで更新する場合は、以下のコマンドを使用して **openshift** namespace に Fuse Console イメージをインポートします。

```
oc import-image fuse7/fuse-console-rhel8:1.9 --from=registry.redhat.io/fuse7/fuse-console-rhel8:1.9 --confirm -n openshift
```

3. 以下のコマンドを実行して、Fuse Console の **APP_NAME** の値を取得します。

```
oc process --parameters -f TEMPLATE-FILENAME
```

ここで、**TEMPLATE-FILENAME** は以下のテンプレートのいずれかになります。

- クラスターテンプレート:
<https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fuse-console-cluster-os4.json>
- 設定可能な RBAC のあるクラスターテンプレート:
<https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fuse-console-cluster-rbac.yml>
- namespace テンプレート:
<https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fuse-console-namespace-os4.json>
- 設定可能な RBAC のある namespace テンプレート:
<https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fuse-console-namespace-rbac.yml>

たとえば、設定可能な RBAC のあるクラスターテンプレートの場合は、以下のコマンドを実行します。

```
oc process --parameters -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fuse-console-cluster-rbac.yml
```

- 以下のコマンドを使用して、[OpenShift 4.x で Fuse Console をセキュア化するための証明書の生成](#)で生成した証明書からシークレットを作成し、Fuse Console にマウントします。コマンドの **APP_NAME** は、Fuse Console アプリケーションの名前に置き換えます。

```
oc create secret tls APP_NAME-tls-proxying --cert server.crt --key server.key
```

- 以下のコマンドを実行して、ローカルにコピーされた Fuse Console テンプレートをベースとした新しいアプリケーションを作成します。コマンドの **myproject** は OpenShift プロジェクトの名前、**mytemp** は Fuse Console テンプレートが含まれるローカルディレクトリへのパス、**myhost** は Fuse Console にアクセスするホストの名前に置き換えます。

- クラスターテンプレートの場合:

```
oc new-app -n myproject -f {templates-base-url}/fuse-console-cluster-os4.json -p ROUTE_HOSTNAME=myhost"
```

- RBAC テンプレートのあるクラスターの場合:

```
oc new-app -n myproject -f {templates-base-url}/fuse-console-cluster-rbac.yml -p ROUTE_HOSTNAME=myhost"
```

- Namespace テンプレートの場合:

```
{templates-base-url}/fuse-console-namespace-os4.json
```

- RBAC テンプレートのある namespace の場合:

```
oc new-app -n myproject -f {templates-base-url}/fuse-console-namespace-rbac.yml
```

- OpenShift Web コンソールを開くように Fuse Console を設定するには、以下のコマンドを実行して **OPENSHIFT_WEB_CONSOLE_URL** 環境変数を設定します。

```
oc set env dc/${APP_NAME} OPENSHIFT_WEB_CONSOLE_URL=`oc get -n openshift-config-managed cm console-public -o jsonpath={.data.consoleURL}`
```

- 以下のコマンドを実行して、Fuse Console デプロイメントの状態と URL を取得します。

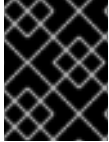
```
oc status
```

- ブラウザーから Fuse Console にアクセスするには、手順 7. で返される URL (例: <https://fuse-console.192.168.64.12.nip.io>) を使用します。

2.4.2.1. OpenShift 4.x で Fuse Console をセキュア化するための証明書の生成

OpenShift 4.x で、Fuse Console プロキシと Jolokia エージェントとの間の接続をセキュアにするには、Fuse Console をデプロイする前にクライアント証明書を生成する必要があります。サービス署名認証局の秘密鍵を使用して、クライアント証明書を署名する必要があります。

この手順は、コマンドラインを使用して Fuse Console をインストールおよびデプロイしている場合にのみ実行する必要があります。Fuse Console Operator を使用している場合は、Fuse Console Operator がこのタスクを処理します。



重要

各 OpenShift クラスターに別のクライアント証明書を生成し、署名する必要があります。複数のクラスターに同じ証明書を使用しないでください。

前提条件

- OpenShift クラスターにアクセス可能な **cluster admin** 権限がある。
- 複数の OpenShift クラスターの証明書を生成し、現在のディレクトリーに別のクラスターの証明書を生成している場合は、以下のいずれかを実行して、現在のクラスターに別の証明書を生成するようにします。
 - 現在のディレクトリーから既存の証明書ファイル (**ca.crt**、**ca.key**、および **ca.srl**) を削除します。
 - 別の作業ディレクトリーに移動します。たとえば、現在の作業ディレクトリーの名前が **cluster1** の場合、新しい **cluster2** ディレクトリーを作成し、作業ディレクトリーをそのディレクトリーに変更します。

```
mkdir ../cluster2
```

```
cd ../cluster2
```

手順

1. cluster admin 権限を持つユーザーとして OpenShift にログインします。

```
oc login -u <user_with_cluster_admin_role>
```

2. 以下のコマンドを実行して、サービス署名認証局のキーを取得します。

- 以下を実行して、証明書を取得します。

```
oc get secrets/signing-key -n openshift-service-ca -o "jsonpath={.data['tls.crt']}" | base64 --decode > ca.crt
```

- 以下を実行して、秘密鍵を取得します。

```
oc get secrets/signing-key -n openshift-service-ca -o "jsonpath={.data['tls.key']}" | base64 --decode > ca.key
```

3. [Kubernetes certificates administration](#) の説明にしたがって、**easysrsa**、**openssl**、または **cfssl** を使用して、クライアント証明書を生成します。

以下に、openssl を使用したコマンドの例を示します。

- a. 秘密鍵を生成します。

```
openssl genrsa -out server.key 2048
```

- b. CSR 設定ファイルを作成します。

```
cat <<EOT >> csr.conf
[ req ]
default_bits = 2048
prompt = no
default_md = sha256
distinguished_name = dn

[ dn ]
CN = fuse-console.fuse.svc

[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
keyUsage=keyEncipherment,dataEncipherment,digitalSignature
extendedKeyUsage=serverAuth,clientAuth
EOT
```

ここでは、**CN** パラメーターの値はアプリケーション名とアプリケーションが使用する namespace を参照します。

- c. CSR を生成します。

```
openssl req -new -key server.key -out server.csr -config csr.conf
```

- d. 署名済みの証明書を発行します。

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out server.crt
-days 10000 -extensions v3_ext -extfile csr.conf
```

次のステップ

[コマンドラインを使用した OpenShift 4.x での Fuse Console のインストールおよびデプロイ](#) で説明されているように、Fuse Console のシークレットを作成するには、この証明書が必要です。

2.4.3. OpenShift 4.x 上の Fuse Console のロールベースアクセス制御

Fuse Console は、OpenShift によって提供されるユーザー承認に応じてアクセスを推測する、ロールベースアクセス制御 (RBAC) を提供します。Fuse Console では、RBAC はユーザーが Pod で MBean 操作を実行できるかどうかを判断します。

OpenShift の承認に関する詳細は、OpenShift ドキュメントの [Using RBAC to define and apply permissions](#) セクションを参照してください。

Operator を使用して OpenShift に Fuse Console をインストールした場合、ロールベースのアクセスはデフォルトで有効になります。

テンプレートでインストールして、Fuse Console のロールベースアクセスを実装する場合は、RBAC で設定可能なテンプレートの 1 つ (**`fuse-console-cluster-rbac.yml`** または **`fuse-console-namespace-rbac.yml`**) を使用して、[Installing and deploying the Fuse Console on OpenShift 4.x by using the command line](#) の説明どおりに Fuse Console をインストールする必要があります。

Fuse Console RBAC は、OpenShift の Pod リソースでユーザーの **verb (動詞)** アクセスを利用して、Fuse Console の Pod の MBean 操作にユーザーがアクセスできるかどうかを判断します。デフォルトでは、Fuse Console には 2 つのユーザーロールがあります。

- **admin**
ユーザーが OpenShift で Pod を **更新** できる場合、ユーザーには Fuse Console の **admin** ロールが付与されます。ユーザーは、Fuse Console で、その Pod に対して **書き込み** の MBean 操作を実行できます。
- **viewer**
ユーザーが OpenShift で Pod を **取得** できる場合、ユーザーには Fuse Console の **viewer** ロールが付与されます。ユーザーは、Fuse Console で、その Pod に対して **読み取り専用** の MBean 操作を実行できます。



注記

RBAC 以外のテンプレートを使用して Fuse Console をインストールした場合、Pod リソースで **update** verb (動詞) が付与された OpenShift ユーザーのみが Fuse Console の MBeans 操作の実行が許可されます。Pod リソースで **get** verb (動詞) を付与されたユーザーは、Pod を **表示** できますが、Fuse Console の操作は実行できません。

関連情報

- [OpenShift 4.x での Fuse Console のアクセ出カルの判断](#)
- [OpenShift 4.x での Fuse Console へのロールベースアクセスのカスタマイズ](#)
- [OpenShift 4.x での Fuse Console のロールベースアクセス制御の無効化](#)

2.4.3.1. OpenShift 4.x での Fuse Console のアクセ出カルの判断

Fuse Console のロールベースアクセス制御は、Pod に対するユーザーの OpenShift パーミッションから推測されます。特定のユーザーに付与された Fuse Console のアクセ出カルを判断するには、Pod に対してユーザーに付与された OpenShift パーミッションを取得します。

前提条件

- ユーザーの名前を知っている必要があります。
- Pod の名前を知っている必要があります。

手順

- ユーザーが Pod に対して Fuse Console の admin ロールを持っているかどうかを確認するには、以下のコマンドを実行してユーザーが OpenShift で Pod を更新できるかどうかを確認します。

```
oc auth can-i update pods/<pod> --as <user>
```

応答が **yes** の場合、ユーザーはその Pod に対して Fuse Console の **admin** ロールを持っています。ユーザーは、Fuse Console で、その Pod に対して **書き込み** の MBean 操作を実行できます。

— ユーザーが Pod に対して Fuse Console の admin ロールを持っているかどうかを確認するには

- ユーザーが Pod に対して Fuse Console の **viewer** ロールを持っているかどうかを確認するには、以下のコマンドを実行してユーザーが OpenShift で Pod を取得できるかどうかを確認します。

```
oc auth can-i get pods/<pod> --as <user>
```

応答が **yes** の場合、ユーザーはその Pod に対して Fuse Console の **viewer** ロールを持っています。ユーザーは、Fuse Console で、その Pod に対して **読み取り専用** の MBean 操作を実行できます。コンテキストによっては、Fuse Console でオプションを無効にしたり、ユーザーによる **書き込み** MBean 操作の試行時にこのユーザーの操作は許可されないという内容のメッセージを表示したりして、**viewer** ロールを持つユーザーによる **書き込み** MBean 操作が実行されないようにします。

応答が **no** の場合、ユーザーは Fuse Console のロールにバインドされず、ユーザーは Fuse Console で Pod を確認できません。

関連情報

- [OpenShift 4.x 上の Fuse Console のロールベースアクセス制御](#)
- [OpenShift 4.x での Fuse Console へのロールベースアクセスのカスタマイズ](#)
- [OpenShift 4.x での Fuse Console のロールベースアクセス制御の無効化](#)

2.4.3.2. OpenShift 4.x での Fuse Console へのロールベースアクセスのカスタマイズ

OperatorHub を使用して Fuse Console をインストールした場合、[OpenShift 4.x 上の Fuse Console のロールベースアクセス制御](#)の説明どおりに、ロールベースアクセス制御 (RBAC) はデフォルトで有効になります。Fuse Console の RBAC 動作をカスタマイズする場合、Fuse Console をデプロイする前に ConfigMap ファイル (カスタムの RBAC 動作を定義する) を指定する必要があります。カスタム ConfigMap ファイルは、Fuse Console Operator をインストールした namespace に配置する必要があります。

コマンドラインテンプレートを使用して Fuse Console をインストールする場合、**deployment-cluster-rbac.yml** および **deployment-namespace-rbac.yml** テンプレートによって、設定ファイル (**ACL.yml**) が含まれる ConfigMap が作成されます。設定ファイルでは、MBean 操作に許可されるロールが定義されます。

前提条件

- OperatorHub または Fuse Console の RBAC テンプレート (**deployment-cluster-rbac.yml** または **deployment-namespace-rbac.yml**) の 1 つを使用して、Fuse Console がインストール済みである必要があります。

手順

Fuse Console の RBAC ロールをカスタマイズする場合は、以下を行います。

1. コマンドラインを使用して Fuse Console をインストールした場合、インストールテンプレートにはデフォルトの ConfigMap ファイルが含まれるため、次のステップを省略できます。OperatorHub を使用して Fuse Console をインストールした場合、Fuse Console をデプロイする前に RBAC ConfigMap を作成します。
 - a. 現在の OpenShift プロジェクトが Fuse Console をインストールするプロジェクトであることを確認します。たとえば、Fuse Console を **fusetest** プロジェクトにインストールするには、以下のコマンドを実行します。

```
oc project fusetest
```

- b. テンプレートから Fuse Console RBAC ConfigMap ファイルを作成するには、以下のコマンドを実行します。

```
oc process -f https://raw.githubusercontent.com/jboss-fuse/application-templates/2.1.x.sb2.redhat-7-8-x/fuse-console-operator-rbac.yml -p APP_NAME=fuse-console | oc create -f -
```

2. 以下のコマンドを実行してエディターで ConfigMap を開きます。

```
oc edit cm $APP_NAME-rbac
```

以下に例を示します。

```
oc edit cm fuse-console-rbac
```

3. ファイルを編集します。
4. ファイルを保存して変更を適用します。Fuse Console の Pod は OpenShift によって自動的に再起動されます。

関連情報

- [OpenShift 4.x 上の Fuse Console のロールベースアクセス制御](#)
- [OpenShift 4.x での Fuse Console のアクセ出カルの判断](#)
- [OpenShift 4.x での Fuse Console のロールベースアクセス制御の無効化](#)

2.4.3.3. OpenShift 4.x での Fuse Console のロールベースアクセス制御の無効化

コマンドラインを使用して Fuse Console をインストールし、Fuse Console の RBAC テンプレートのいずれかを指定した場合、Fuse Console の **HAWTIO_ONLINE_RBAC_ACL** 環境変数は、ロールベースアクセス制御 (RBAC) の ConfigMap 設定ファイルのパスを OpenShift サーバーに渡します。**HAWTIO_ONLINE_RBAC_ACL** 環境変数が指定されていない場合、RBAC のサポートは無効になり、Pod リソース (OpenShift の) で **update** verb (動詞) が付与されたユーザーのみが Fuse Console の Pod で MBean 操作を呼び出すことが承認されます。

OperatorHub を使用して Fuse Console をインストールする場合は、ロールベースのアクセスはデフォルトで有効になり、**HAWTIO_ONLINE_RBAC_ACL** 環境変数は適用されません。

前提条件

コマンドラインを使用して Fuse Console がインストール済みで、Fuse Console の RBAC テンプレート (**deployment-cluster-rbac.yml** または **deployment-namespace-rbac.yml**) のいずれかを指定している。

手順

Fuse Console のロールベースのアクセスを無効にするには、以下を実行します。

1. OpenShift で、Fuse Console の **Deployment Config** リソースを編集します。
2. **HAWTIO_ONLINE_RBAC_ACL** 環境変数の定義をすべて削除します。

(値を消去するだけでは不十分です)。

3. ファイルを保存して変更を適用します。Fuse Console の Pod は OpenShift によって自動的に再起動されます。

関連情報

- [OpenShift 4.x 上の Fuse Console のロールベースアクセス制御](#)
- [OpenShift 4.x での Fuse Console のアクセス出力の判断](#)
- [OpenShift 4.x での Fuse Console へのロールベースアクセスのカスタマイズ](#)

2.4.4. OpenShift 4.x での Fuse Console のアップグレード

Red Hat OpenShift 4.x では、Red Hat Fuse Operator などの Operator の更新が処理されます。詳細は、[OpenShift ドキュメントの Operator](#) を参照してください。

その後、アプリケーションの設定方法によっては、Operator の更新でアプリケーションのアップグレードをトリガーできるようになります。

Fuse Console アプリケーションでは、アプリケーションカスタムリソース定義の **.spec.version** フィールドを編集して、アプリケーションのアップグレードをトリガーすることもできます。

前提条件

- OpenShift クラスターの管理者権限が必要です。

手順

Fuse Console アプリケーションをアップグレードするには、以下を行います。

1. ターミナルウィンドウで、以下のコマンドを使用してアプリケーションカスタムリソース定義の **.spec.version** フィールドを変更します。

```
oc patch <project-name> <custom-resource-name> --type='merge' -p '{"spec": {"version": "1.7.1"}}'
```

以下に例を示します。

```
oc patch myproject example-fuseconsole --type='merge' -p '{"spec":{"version": "1.7.1"}}'
```

2. アプリケーションの状態が更新されたことを確認します。

```
oc get myproject
```

応答には、バージョン番号などのアプリケーションに関する情報が表示されます。

```
NAME          AGE  URL
example-fuseconsole 1m   https://fuseconsole.192.168.64.38.nip.io
docker.io/fuseconsole/online:1.7.1
```

.spec.version フィールドの値を変更すると、OpenShift によってアプリケーションが自動的に再デプロイされます。

- バージョンの変更によってトリガーされた再デプロイの状態をチェックするには、以下を実行します。

```
oc rollout status deployment.v1.apps/example-fuseconsole
```

正常にデプロイされた場合は以下の応答が表示されます。

```
deployment "example-fuseconsole" successfully rolled out
```

2.5. OPENSIFT で FUSE アプリケーションを監視するための PROMETHEUS の設定

2.5.1. Prometheus の概要

[Prometheus](#) は、Red Hat OpenShift 環境にデプロイされたサービスの監視に使用できる、システムおよびサービスのオープンソースの監視およびアラートツールキットです。Prometheus は、指定の間隔で設定されたサービスからメトリクスを収集および保存します。さらに、ルール式の評価や結果の表示を行い、指定の条件が true になるとアラートをトリガーできます。



重要

Prometheus に対する Red Hat のサポートは、Red Hat 製品ドキュメントに記載されているセットアップと推奨設定に限定されます。

OpenShift サービスを監視するには、エンドポイントを Prometheus 形式に公開するように各サービスを設定する必要があります。このエンドポイントは、メトリクスのリストとメトリクスの現在の値を提供する HTTP インターフェイスです。Prometheus は定期的にターゲット定義の各エンドポイントをスクレイピングし、収集したデータをそのデータベースに書き込みます。Prometheus は、現在実行中のセッションだけでなく、長期間にわたってデータを収集します。Prometheus は、データ上でクエリーをグラフィカルに可視化し、実行できるように、データを格納します。

2.5.1.1. Prometheus クエリー

Prometheus Web インターフェイスでは、[Prometheus Query Language \(PromQL\)](#) でクエリーを作成して、収集したデータを選択し、集約することができます。

たとえば、以下のクエリーを使用すると、Prometheus が直近の 5 分間に記録したすべての時系列データから、メトリクス名が **http_requests_total** である値をすべて選択することができます。

```
http_requests_total[5m]
```

クエリーの結果をさらに定義または絞り込むには、メトリクスのラベル (**key:value** ペア) を指定します。たとえば、以下のクエリーを使用すると、Prometheus が直近の 5 分間に記録したすべての時系列データから、メトリクス名が **http_requests_total** でかつジョブラベルが **integration** に設定されている値をすべて選択することができます。

```
http_requests_total{job="integration"}[5m]
```

2.5.1.2. Prometheus データの表示オプション

Prometheus がクエリーの結果を処理する方法を指定できます。

- Prometheus データを表データとして Prometheus の式ブラウザーで表示します。
- [Prometheus HTTP API](#) 経由で外部システムで Prometheus データを消費します。
- Prometheus データをグラフで表示します。
Prometheus は、収集するデータのデフォルトグラフィカルビューを提供します。Prometheus メトリクスを表示するより堅牢なグラフィカルダッシュボードが必要な場合は、Grafana を選択するのが一般的です。



注記

Grafana はコミュニティがサポートする機能です。Grafana をデプロイして Red Hat の製品を監視する設定は、Red Hat の実稼働環境におけるサービスレベルアグリーメント (SLA) の対象外です。

PromQL 言語を使用して、[Prometheus の Alertmanager ツール](#) で通知を設定することもできます。

2.5.2. Prometheus の設定

Prometheus を設定するには、クラスターに Prometheus Operator のカスタムリソース定義をインストールし、Prometheus を Fuse アプリケーションが含まれる OpenShift プロジェクトに追加します。

前提条件

- OpenShift クラスターにアクセス可能な **cluster admin** 権限がある。
- [Fuse on OpenShift ガイド](#) の説明にしたがって、Fuse on OpenShift イメージおよびテンプレートをインストールして OpenShift クラスターが準備されている必要があります。
- クラスターで OpenShift プロジェクトを作成し、Fuse アプリケーションをそのプロジェクトに追加されている必要があります。

手順

1. 管理者権限で OpenShift にログインします。

```
oc login -u system:admin
```

2. Prometheus operator の実行に必要なカスタムリソース定義をインストールします。

```
oc create -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/openshift3/fuse-prometheus-crd.yml
```

Prometheus Operator がクラスターのすべての namespace で利用できるようになります。

3. 以下のコマンド構文を使用して、Prometheus Operator を namespace にインストールします。

```
oc process -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/openshift3/fuse-prometheus-operator.yml -p NAMESPACE=<YOUR NAMESPACE> | oc create -f -
```

たとえば、**myproject** という名前のプロジェクト (namespace) には、以下のコマンドを使用します。


```
oc process -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/openshift3/fuse-prometheus-operator.yml -p NAMESPACE=myproject | oc create -f -
```



注記

Prometheus Operator を namespace に初めてインストールする場合、Prometheus リソース Pod が起動するまでに数分の時間がかかる場合があります。その後、Prometheus Operator をクラスターの他の namespace にインストールすると、Prometheus リソース Pod の起動ははるかに速くなります。

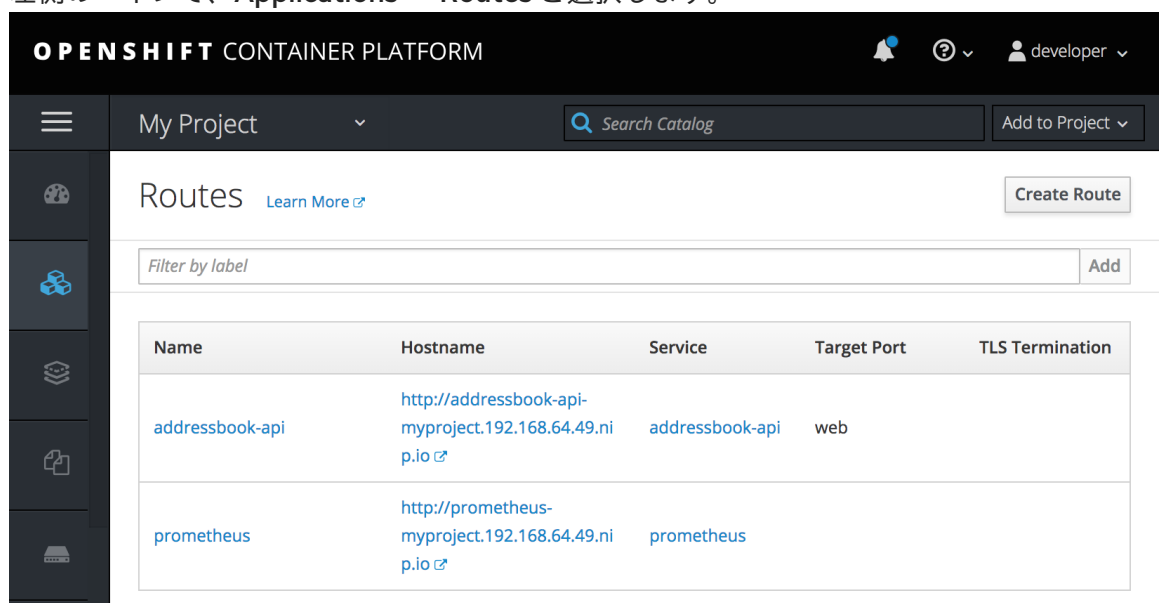
4. 以下のコマンド構文を使用して、Prometheus Operator を指示し、プロジェクトの Fuse アプリケーションを監視します。

```
oc process -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/openshift3/fuse-servicemonitor.yml -p NAMESPACE=<YOUR NAMESPACE> -p FUSE_SERVICE_NAME=<YOUR FUSE SERVICE> | oc apply -f -
```

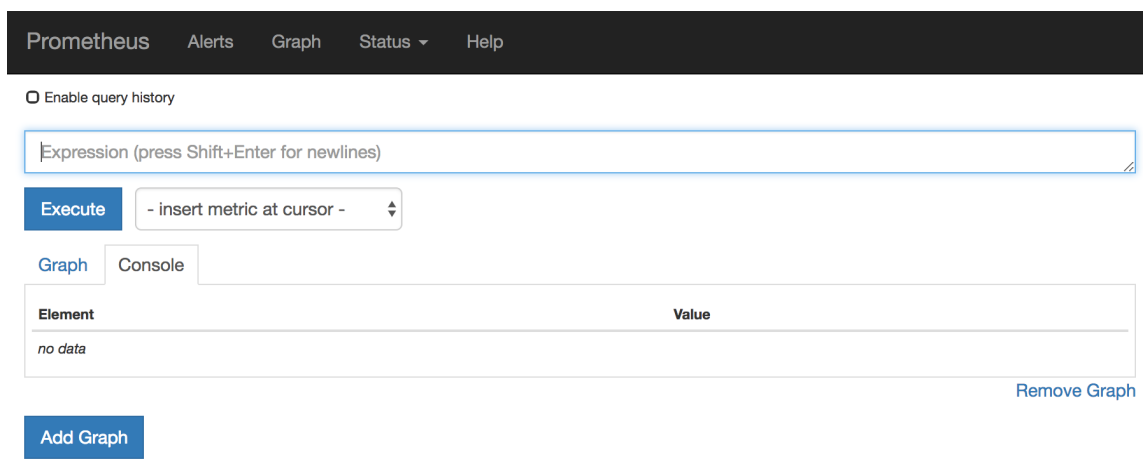
たとえば、**myfuseapp** という名前の Fuse アプリケーションが含まれる **myproject** という名前の OpenShift プロジェクト (namespace) には、次のコマンドを使用します。

```
oc process -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/openshift3/fuse-servicemonitor.yml -p NAMESPACE=myproject -p FUSE_SERVICE_NAME=myfuseapp | oc apply -f -
```

5. Prometheus ダッシュボードを開くには、以下を実行します。
 - a. OpenShift コンソールにログインします。
 - b. Prometheus を追加したプロジェクトを開きます。
 - c. 左側のペインで、**Applications -> Routes** と選択します。



- d. Prometheus Hostname URL をクリックし、新しいブラウザータブまたはウィンドウで Prometheus ダッシュボードを開きます。



- e. Prometheus を初めて使用する場合は、https://prometheus.io/docs/prometheus/latest/getting_started/ を参照してください。

2.5.3. OpenShift 環境変数

アプリケーションの Prometheus インスタンスを設定するには、表2.2「Prometheus 環境変数」にリストされている OpenShift 環境変数を設定します。

表2.2 Prometheus 環境変数

環境変数	説明	デフォルト
AB_PROMETHEUS_HOST	バインドするホストアドレス。	0.0.0.0
AB_PROMETHEUS_OFF	設定されている場合、Prometheus のアクティベートを無効にします (空の値をエコーします)。	Prometheus が有効。
AB_PROMETHEUS_PORT	使用するポート。	9779
AB_JMX_EXPORTER_CONFIG	ファイル (パスを含む) を Prometheus 設定ファイルとして使用します。	Camel メトリクスが含まれる /opt/prometheus/prometheus-config.yml ファイル。
AB_JMX_EXPORTER_OPTS	JMX エクスポート設定に追加する追加オプション。	該当なし

関連情報

Pod の環境変数の設定に関する詳細は、**OpenShift 開発者ガイド** (https://access.redhat.com/documentation/ja-jp/openshift_container_platform/3.11/html/developer_guide/index) を参照してください。

2.5.4. Prometheus が監視および収集するメトリクスの制御

デフォルトでは、Prometheus は Camel によって公開される可能性のあるすべてのメトリクスが含まれる設定ファイル (<https://raw.githubusercontent.com/jboss-fuse/application-templates/master/prometheus/prometheus-config.yml>) を使用します。

Prometheus が監視および収集するアプリケーション内にカスタムメトリクスがある場合 (アプリケーションプロセスを実行するオーダーの数など)、独自の設定ファイルを使用できます。識別できるメトリクスは、JMX で提供されるメトリクスに限定されるため注意してください。

手順

カスタム設定ファイルを使用して、デフォルトの Prometheus 設定では対応されない JMX Bean を公開するには、以下の手順に従います。

1. カスタム Prometheus 設定ファイルを作成します。デフォルトファイルのコンテンツ (**prometheus-config.yml** <https://raw.githubusercontent.com/jboss-fuse/application-templates/master/prometheus/prometheus-config.yml>) を形式の目安として使用できます。カスタム設定ファイルには任意の名前を使用できます (例: **my-prometheus-config.yml** など)。
2. prometheus 設定ファイル (例:**my-prometheus-config.yml**) をアプリケーションの **src/main/jkube-includes** ディレクトリーに追加します。
3. アプリケーション内に **src/main/jkube/deployment.xml** ファイルを作成し、設定ファイルに設定された値で **AB_JMX_EXPORTER_CONFIG** 環境変数のエントリーを追加します。以下に例を示します。

```
spec:
  template:
    spec:
      containers:
      -
        resources:
          requests:
            cpu: "0.2"
          limits:
            cpu: "1.0"
        env:
        - name: SPRING_APPLICATION_JSON
          value: '{"server":{"tomcat":{"max-threads":1}}}'
        - name: AB_JMX_EXPORTER_CONFIG
          value: "my-prometheus-config.yml"
```

この環境変数は、Pod レベルでアプリケーションに適用されます。

4. アプリケーションの再構築およびデプロイ

2.6. FUSE ON OPENSIFT のメータリングの使用

OCP 4 で利用可能なメータリングツールを使用して、異なるデータソースからメータリングレポートを生成できます。クラスター管理者として、メータリングを使用してクラスターの内容を分析できます。独自のクエリーを作成するか、または事前定義 SQL クエリーを使用して、利用可能な異なるデータソースからデータを処理する方法を定義できます。Prometheus をデフォルトのデータソースとして使用すると、Pod、namespace、およびその他ほとんどの Kubernetes リソースのレポートを生成できます。メータリングツールを使用するには、最初にメータリング Operator を OpenShift Container Platform 4.x にインストールし、設定する必要があります。メータリングに関する詳細は [メータリング](#) を参照してください。

2.6.1. メータリングリソース

メータリングには、メータリングのデプロイメントやインストール、およびメータリングが提供するレポート機能を管理するために使用できるリソースが多数含まれています。メータリングは以下の CustomResourceDefinition (CRD) を使用して管理されます。

表2.3 メータリングリソース

名前	説明
MeteringConfig	デプロイメントのメータリングスタックを設定します。メータリングスタックを設定する各コンポーネントを制御するカスタマイズおよび設定オプションが含まれます。
Report	使用するクエリー、クエリーを実行するタイミングおよび頻度、および結果を保存する場所を制御します。
ReportQuery	ReportDataSource 内に含まれるデータに対して分析を実行するために使用される SQL クエリーが含まれます。
ReportDataSource	ReportQuery および Report で利用可能なデータを制御します。メータリング内で使用できるように複数の異なるデータベースへのアクセスの設定を可能にします。

2.6.2. Fuse on OpenShift のメータリングラベル

表2.4 メータリングラベル

ラベル	使用できる値
com.company	Red_Hat
rht.prod_name	Red_Hat_Integration
rht.prod_ver	7.9
rht.comp	Fuse
rht.comp_ver	7.9
rht.subcomp	fuse7-java-openshift fuse7-eap-openshift fuse7-karaf-openshift
rht.subcomp_t	infrastructure

2.7. カスタム GRAFANA ダッシュボードでの FUSE ON OPENSIFT の監視

OpenShift Container Platform 4.6 は、クラスターコンポーネントおよびユーザー定義のワークロードの状態を理解するのに役立つモニタリングダッシュボードを提供します。

前提条件

- クラスターに Prometheus をインストールおよびデプロイしておく必要があります。OpenShift 4 に Grafana をインストールする方法の詳細は <https://github.com/jboss-fuse/application-templates/blob/master/monitoring/prometheus.md> を参照してください。
- Grafana がインストールおよび設定されている必要があります。

Fuse on OpenShift のカスタムダッシュボード

Fuse on OpenShift に使用できるカスタムダッシュボードは 2 つあります。これらのダッシュボードを使用するには、クラスターに Grafana と Prometheus をインストールおよび設定しておく必要があります。Fuse on OpenShift には 2 種類のダッシュボードサンプルが用意されています。これらのダッシュボードは [Fuse Grafana dashboards](#) からインポートできます。

- Fuse Pod / インスタンスメトリクスダッシュボード
このダッシュボードは、単一の Fuse アプリケーション Pod / インスタンスからメトリクスを収集します。**`fuse-grafana-dashboard.yml`** を使用してダッシュボードをインポートできます。OpenShift の Fuse Pod メトリクスダッシュボードのパネルには以下が含まれます。

表2.5 Fuse Pod メトリクスダッシュボード

タイトル	凡例	クエリー	説明
Process Start Time	-	<code>process_start_time_seconds{pod="\$pod"}*1000</code>	処理が開始された時間
Current Memory HEAP	-	<code>sum(jvm_memory_bytes_used{pod="\$pod", area="heap"})*100/sum(jvm_memory_bytes_max{pod="\$pod", area="heap"})</code>	Fuse によって現在使用されているメモリー
Memory Usage	committed	<code>sum(jvm_memory_bytes_committed{pod="\$pod"})</code>	コミットされたメモリー
	used	<code>sum(jvm_memory_bytes_used{pod="\$pod"})</code>	使用されているメモリー
	max	<code>sum(jvm_memory_bytes_max{pod="\$pod"})</code>	最大メモリー

タイトル	凡例	クエリー	説明
Threads	current?	jvm_threads_current{pod="\$pod"}	現在のスレッド数
	daemon	jvm_threads_daemon{pod="\$pod"}	デーモンスレッドの数
	peak	jvm_threads_peak{pod="\$pod"}	ピークスレッドの数
Camel Exchanges / 1m	Exchanges Completed / 1m	sum(increase(org_apache_camel_Exchanges Completed{pod="\$pod"}[1m]))	1分あたりの完了した Camel エクスチェンジ
	Exchanges Failed / 1m	sum(increase(org_apache_camel_Exchanges Failed{pod="\$pod"}[1m]))	1分あたりの失敗した Camel エクスチェンジ
	Exchanges Total / 1m	sum(increase(org_apache_camel_Exchanges Total{pod="\$pod"}[1m]))	1分あたりの Camel エクスチェンジの合計
	Exchanges Inflight	sum(org_apache_camel_ExchangesInflight{pod="\$pod"})	現在処理中の Camel エクスチェンジ
Camel Processing Time	Delta Processing Time	sum(org_apache_camel_DeltaProcessingTime{pod="\$pod"})	Camel 処理時間のデルタ
	Last Processing Time	sum(org_apache_camel_LastProcessingTime{pod="\$pod"})	最後の Camel 処理時間
	Max Processing Time	sum(org_apache_camel_MaxProcessingTime{pod="\$pod"})	Camel の最大処理時間
	Min Processing Time	sum(org_apache_camel_MinProcessingTime{pod="\$pod"})	Camel の最小処理時間
	Mean Processing Time	sum(org_apache_camel_MeanProcessingTime{pod="\$pod"})	Camel の平均処理時間

タイトル	凡例	クエリー	説明
Camel Service Durations	Maximum Duration	sum(org_apache_camel_MaxDuration{pod="\$pod"})	Camel のサービス期間
	Minimum Duration	sum(org_apache_camel_MinDuration{pod="\$pod"})	Camel の最小サービス期間
	Mean Duration	sum(org_apache_camel_MeanDuration{pod="\$pod"})	Camel の平均サービス期間
Camel Failures & Redeliveries	Redeliveries	sum(org_apache_camel_Redeliveries{pod="\$pod"})	再配信の数
	Last Processing Time	sum(org_apache_camel_LastProcessingTime{pod="\$pod"})	最後の Camel 処理時間
	External Redeliveries	sum(org_apache_camel_ExternalRedeliveries{pod="\$pod"})	外部再配信の数

- Fuse Camel ルートメトリクスダッシュボード
このダッシュボードは、Fuse アプリケーションで単一の Camel ルートからメトリクスを収集します。**fuse-grafana-dashboard-routes.yml** を使用してダッシュボードをインポートできます。OpenShift の Fuse Camel Route メトリクスダッシュボードのパネルには以下が含まれます。

表2.6 Fuse Camel ルートメトリクスダッシュボード

タイトル	凡例	クエリー	説明
Exchanges per second	-	rate(org_apache_camel_ExchangesTotal{route="\$route"}[5m])	1 秒あたりの Camel エクステンジの合計
Exchanges Inflight	-	max(org_apache_camel_ExchangesInflight{route="\$route"})	現在処理中の Camel エクステンジの数

タイトル	凡例	クエリー	説明
Exchanges failure rate	-	<code>sum(org_apache_camel_ExchangesFailed{route=\"\$route\"}) / sum(org_apache_camel_ExchangesTotal{route=\"\$route\"})</code>	失敗した Camel エクスチェンジの割合 (パーセント)
Mean processing time	-	<code>org_apache_camel_MeanProcessingTime{route=\"\$route\"}</code>	Camel の平均処理時間
Exchanges per second	Failed	<code>rate(org_apache_camel_ExchangesFailed{route=\"\$route\"} [5m])</code>	1 秒あたりの失敗した エクスチェンジ
	Completed	<code>rate(org_apache_camel_ExchangesCompleted{route=\"\$route\"} [5m])</code>	1 秒あたりの完了した エクスチェンジ
Exchanges Inflight	Exchanges Inflight	<code>org_apache_camel_ExchangesInflight{route=\"\$route\"}</code>	現在処理中の Camel エクスチェンジ
Processing time	Max	<code>org_apache_camel_MaxProcessingTime{route=\"\$route\"}</code>	Camel の最大処理時間
	Mean	<code>org_apache_camel_MeanProcessingTime{route=\"\$route\"}</code>	Camel の平均処理時間
	Min	<code>org_apache_camel_MinProcessingTime{route=\"\$route\"}</code>	Camel の最小処理時間
External Redeliveries per second	-	<code>rate(org_apache_camel_ExternalRedeliveries{route=\"\$route\"} [5m])</code>	1 秒あたりの外部再配信
Redeliveries per second	-	<code>rate(org_apache_camel_Redeliveries{route=\"\$route\"} [5m])</code>	1 秒あたりの再配信

タイトル	凡例	クエリー	説明
Failures handled per second	-	rate(org_apache_camel_FailuresHandled{route=\"\$route\"}[5m])	1秒あたりに処理された失敗

2.8. OPENSIFT 3.X サーバーでの FUSE イメージストリームおよびテンプレートのインストール

registry.redhat.io への認証を設定した後、Red Hat Fuse on OpenShift イメージストリームおよびテンプレートをインポートおよび使用します。

手順

1. OpenShift サーバーを起動します。
2. OpenShift サーバーに管理者としてログインします。

```
oc login -u system:admin
```

3. docker-registry シークレットを作成したプロジェクトを使用していることを確認します。

```
oc project openshift
```

4. Fuse on OpenShift イメージストリームをインストールします。

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005
```

```
oc create -n openshift -f ${BASEURL}/fis-image-streams.json
```

5. クイックスタートテンプレートをインストールします。

```
for template in eap-camel-amq-template.json \
eap-camel-cdi-template.json \
eap-camel-cxf-jaxrs-template.json \
eap-camel-cxf-jaxws-template.json \
karaf-camel-amq-template.json \
karaf-camel-log-template.json \
karaf-camel-rest-sql-template.json \
karaf-cxf-rest-template.json ;
do
oc create -n openshift -f \
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/quickstarts/${template}
done
```

6. Spring Boot 2 のクイックスタートテンプレートをインストールします。

```
for template in spring-boot-2-camel-amq-template.json \
```

```

spring-boot-2-camel-config-template.json \
spring-boot-2-camel-drools-template.json \
spring-boot-2-camel-infinispan-template.json \
spring-boot-2-camel-rest-3scale-template.json \
spring-boot-2-camel-rest-sql-template.json \
spring-boot-2-camel-template.json \
spring-boot-2-camel-xa-template.json \
spring-boot-2-camel-xml-template.json \
spring-boot-2-cxf-jaxrs-template.json \
spring-boot-2-cxf-jaxws-template.json \
spring-boot-2-cxf-jaxrs-xml-template.json \
spring-boot-2-cxf-jaxws-xml-template.json ;
do oc create -n openshift -f \
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-
2.1.0.fuse-sb2-790047-redhat-00005/quickstarts/${template}
done

```

7. Fuse Console のテンプレートをインストールします。

```

oc create -n openshift -f https://raw.githubusercontent.com/jboss-fuse/application-
templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fis-console-cluster-
template.json
oc create -n openshift -f https://raw.githubusercontent.com/jboss-fuse/application-
templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fis-console-
namespace-template.json

```



注記

Fuse Console のデプロイに関する詳細は、[OpenShift 4.x での Fuse Console の設定](#) を参照してください。

8. Apicurito テンプレートをインストールします。

```
oc create -n openshift -f ${BASEURL}/fuse-apicurito.yml
```

9. (任意手順): インストールされた Fuse on OpenShift イメージおよびテンプレートを表示します。

```
oc get template -n openshift
```

2.8.1. OpenShift 3.11 での Fuse Console の設定

OpenShift 3.11 では、以下のように Fuse Console にアクセスできます。

- プロジェクトで実行しているすべての Fuse コンテナを監視できるように、Fuse Console を OpenShift プロジェクトに追加する。
- クラスター上のすべてのプロジェクトで稼働中のすべての Fuse コンテナを監視できるように、Fuse Console を OpenShift クラスターに追加する。
- 実行している単一の Fuse コンテナを監視できるように、特定の Fuse Pod から Fuse Console を開く。

コマンドラインから Fuse Console テンプレートをデプロイします。



注記

Minishift または CDK ベースの環境変数に Fuse Console をインストールするには、以下の KCS の記事で説明されている手順に従います。

- Minishift または CDK ベースの環境に Fuse Console をインストールするには、[KCS 4998441](#) を参照してください。
- Jolokia 認証を無効にする必要がある場合は、[KCS 3988671](#) で説明されている回避策を参照してください。

前提条件

- [Fuse on OpenShift ガイド](#) の説明にしたがって、Fuse Console の Fuse on OpenShift イメージストリームおよびテンプレートをインストールする必要があります。



注記

- Fuse Console のユーザー管理は、OpenShift によって処理されます。
- ロールベースアクセス制御 (デプロイ後に Fuse Console にアクセスするユーザーの場合) は現在 Fuse on OpenShift 3.11 では使用できません。

[「OpenShift 3.11 での Fuse Console のデプロイ」](#)

[「OpenShift 3.11 の Fuse Console から単一の Fuse Pod を監視」](#)

2.8.1.1. OpenShift 3.11 での Fuse Console のデプロイ

[表2.7 「Fuse Console テンプレート」](#) は、Fuse アプリケーションのデプロイメントのタイプに応じて、コマンドラインから Fuse Console をデプロイするために使用する OpenShift 3.11 テンプレートについて説明しています。

表2.7 Fuse Console テンプレート

タイプ	説明
<code>fis-console-cluster-template.json</code>	Fuse Console は、複数の namespace またはプロジェクトにまたがってデプロイされた Fuse アプリケーションを検出し、接続することができます。このテンプレートをデプロイするには、OpenShift の cluster-admin ロールが必要です。
<code>fis-console-namespace-template.json</code>	このテンプレートは、Fuse Console の現在の OpenShift プロジェクト (namespace) へのアクセスを制限するため、単一のテナントデプロイメントとして動作します。このテンプレートをデプロイするには、現在の OpenShift プロジェクトの admin ロールが必要です。

任意で以下のコマンドを実行すると、すべてのテンプレートのパラメーターの一覧を表示できます。

```
oc process --parameters -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fis-console-namespace-template.json
```



注記

Fuse Console のテンプレートは、デフォルトでエンドツーエンド暗号化を設定するため、Fuse Console のリクエストはブラウザからクラスター内のサービスまでエンドツーエンドでセキュア化されます。

前提条件

- OpenShift 3.11 のクラスターモードでは、cluster admin ロールとクラスターモードテンプレートが必要です。以下のコマンドを実行します。

```
oc adm policy add-cluster-role-to-user cluster-admin system:serviceaccount:openshift-infra:template-instance-controller
```

手順

コマンドラインから Fuse Console をデプロイするには、以下を行います。

1. 以下のコマンドの1つを実行して、Fuse Console テンプレートをベースとした新しいアプリケーションを作成します。コマンドの **myproject** はプロジェクトの名前に置き換えます。

- Fuse Console の **cluster** テンプレートの場合は、以下のようになります。**myhost** は Fuse Console にアクセスするホストの名前になります。

```
oc new-app -n myproject -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fis-console-cluster-template.json -p ROUTE_HOSTNAME=myhost
```

- Fuse Console の **namespace** テンプレートの場合は以下のようになります。

```
oc new-app -n myproject -f https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/fis-console-namespace-template.json
```



注記

namespace テンプレートの `route_hostname` パラメーターは OpenShift によって自動的に生成されるため、省略することが可能です。

2. 以下のコマンドを実行して、Fuse Console デプロイメントの状態と URL を取得します。

```
oc status
```

3. ブラウザーから Fuse Console にアクセスするには、提供される URL (例: <https://fuse-console.192.168.64.12.nip.io>) を使用します。

2.8.1.2. OpenShift 3.11 の Fuse Console から単一の Fuse Pod を監視

OpenShift 3.11 で実行している Fuse Pod の Fuse Console を開きます。

前提条件

- Pod ビューで Fuse Console へのリンクを表示するよう OpenShift を設定するには、Fuse on OpenShift イメージを実行している Pod が **jolokia** に設定された name 属性内で TCP ポートを宣言する必要があります。

```
{
  "kind": "Pod",
  [...]
  "spec": {
    "containers": [
      {
        [...]
        "ports": [
          {
            "name": "jolokia",
            "containerPort": 8778,
            "protocol": "TCP"
          }
        ]
      }
    ]
  }
}
```

手順

- OpenShift プロジェクトの **Applications → Pods** ビューで、Pod 名をクリックし、実行している Fuse Pod の詳細を表示します。このページの右側に、コンテナテンプレートの概要が表示されます。

Template

Containers

CONTAINER: SPRING-BOOT

 **Image:** [test/fuse70-spring-boot](#) eda527f 193.1 MiB

 **Build:** [fuse70-spring-boot-s2i, #2](#)

 **Source:** Binary

 **Ports:** 8080/TCP (http), 8778/TCP (jolokia), 9779/TCP (prometheus)

 **Mount:** default-token-p4zsn → /var/run/secrets/kubernetes.io/serviceaccount
read-only

 **CPU:** 200 millicores to 1 core

 **Readiness Probe:** GET /health on port 8081 (HTTP) 10s delay, 1s timeout

 **Liveness Probe:** GET /health on port 8081 (HTTP) 180s delay, 1s timeout

 [Open Java Console](#)

- このビューの **Open Java Console** リンクをクリックし、Fuse Console を開きます。

OPENSIFT CONTAINER PLATFORM

Connected to spring-boot

Back

JMXThreadsCamel

AttributesOperationsChartsRoute DiagramSourceInflightBlockedEndpoints (in/out)Type Converters

Camel Contexts

camel

Routes

simple-route

Endpoints

Components

MBeans

StartPauseStopDelete

Filter...

	State	Context	Route	Completed	Failed	Inflight	Mean Time	Min Time	Max Time
	●	camel	simple-route	185	0	0	1	0	10

第3章 制限された環境での FUSE ON OPENSIFT のインストール

制限のない環境で Fuse on OpenShift をインストールするには、イメージストリームおよびテンプレートを **registry.redhat.io** からプルします。インターネットアクセスがないか、またはこれに制限がある実稼働環境では、それが不可能です。ここでは、制限される環境で Fuse on OpenShift をインストールする方法について説明します。

前提条件

- 制限された環境で実行できるように OpenShift サーバーをインストールし、設定している。

3.1. 内部 DOCKER レジストリーの設定

ここでは、イメージのプッシュまたはプルに使用できる、内部 docker レジストリーを設定する方法を説明します。イメージをプルまたはプッシュできる内部 docker レジストリーを設定する必要があります。

手順

1. 内部 ROOT CA をインストールします。

```
cd /etc/pki/ca-trust/source/anchors
sudo curl -O https://password.corp.redhat.com/RH-IT-Root-CA.crt
sudo update-ca-trust extract
sudo update-ca-trust update
```

この証明書により、システムがレジストリーに対して自己認証できるようになります。

2. **registry.redhat.io** にログインします。

```
docker login -u USERNAME -p PASSWORD registry.redhat.io
```

3. **registry.redhat.io** から Fuse on OpenShift イメージをプルします。

```
docker pull registry.redhat.io/fuse7/fuse-java-openshift-rhel8:1.9
docker pull registry.redhat.io/fuse7/fuse-java-openshift-jdk11-rhel8:1.9
docker pull registry.redhat.io/fuse7/fuse-karaf-openshift-rhel8:1.9
docker pull registry.redhat.io/fuse7/fuse-console-rhel8:1.9
docker pull registry.redhat.io/fuse7/fuse-apicurito-rhel8:1.9
docker pull registry.redhat.io/fuse7/fuse-apicurito-generator-rhel8:1.9
```

4. プルしたイメージストリームにタグを付けます。

```
docker tag registry.redhat.io/fuse7/fuse-java-openshift-rhel8:1.9 docker-
registry.upshift.redhat.com/fuse7/fuse-java-openshift-rhel8:1.9
docker tag registry.redhat.io/fuse7/fuse-java-openshift-jdk11-rhel8:1.9 docker-
registry.upshift.redhat.com/fuse7/fuse-java-openshift-jdk11-rhel8:1.9
docker tag registry.redhat.io/fuse7/fuse-karaf-openshift-rhel8:1.9 docker-
registry.upshift.redhat.com/fuse-karaf-openshift-rhel8:1.9
docker tag registry.redhat.io/fuse7/fuse-console-rhel8:1.9 docker-
registry.upshift.redhat.com/fuse7-fuse-console-rhel8:1.9
docker tag registry.redhat.io/fuse7/fuse-apicurito-rhel8:1.9 docker-
```

```
registry.upshift.redhat.com/fuse7-fuse-apicurito-rhel8:1.9
docker tag registry.redhat.io/fuse7/fuse-apicurito-generator-rhel8:1.9 docker-
registry.upshift.redhat.com/fuse7-fuse-apicurito-generator-rhel8:1.9
```

5. タグを付けたイメージストリームを内部 docker レジストリーにプッシュします。

```
docker push docker-registry.upshift.redhat.com/fuse7/fuse-java-openshift-rhel8:1.9
docker push docker-registry.upshift.redhat.com/fuse7/fuse-java-openshift-jdk11-rhel8:1.9
docker push docker-registry.upshift.redhat.com/fuse-karaf-openshift-rhel8:1.9
docker push docker-registry.upshift.redhat.com/fuse7-fuse-console-rhel8:1.9
docker push docker-registry.upshift.redhat.com/fuse7-fuse-apicurito-rhel8:1.9
docker push docker-registry.upshift.redhat.com/fuse7-fuse-apicurito-generator-rhel8:1.9
```

3.2. 内部レジストリーのシークレットの設定

制限された docker レジストリーを設定し、すべてのイメージをプッシュした後、内部レジストリーと通信できるように、制限された OpenShift サーバーを設定する必要があります。

手順

1. OpenShift サーバーに管理者としてログインします。

```
oc login -u system:admin
```

2. Red Hat カスタマーポータルアカウントまたは Red Hat Developer Program アカウントのクレデンシャルを使用して、docker-registry シークレットを作成します。<pull_secret_name> は作成するシークレットの名前に置き換えてください。

```
oc create secret docker-registry psi-internal-registry <pull_secret_name> \
--docker-server=docker-registry.upshift.redhat.com \
--docker-username=CUSTOMER_PORTAL_USERNAME \
--docker-password=CUSTOMER_PORTAL_PASSWORD \
--docker-email=EMAIL_ADDRESS
```

3. Pod のイメージをプルするため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します。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と一致する必要があります。

```
oc secrets add serviceaccount/builder secrets/psi-internal-registry
oc secrets add serviceaccount/default secrets/psi-internal-registry --for=pull
oc secrets add serviceaccount/builder secrets/psi-internal-registry
```

4. ビルドイメージをプッシュまたはプルするためにシークレットを使用するには、シークレットを Pod の内部にマウントする必要があります。シークレットをマウントするには、以下のコマンドを使用します。

```
oc secrets link default psi-internal-registry
oc secrets link default psi-internal-registry --for=pull
oc secrets link builder psi-internal-registry
```

3.3. 制限された環境での FUSE ON OPENSIFT イメージのインストール

fis-image-streams.json ファイルには、Red Hat Fuse on OpenShift の `imageStream` 定義が含まれます。ただし、すべてのイメージストリームは **registry.redhat.io** を参照します。**psi-internal-registry** URL へのすべての **registry.redhat.io** を変更する必要があります。

手順

1. Red Hat Fuse on OpenShift イメージストリームの json ファイルをダウンロードします。

```
curl -o fis-image-streams.json {BASEURL}
```

2. **fis-image-streams.json** ファイルを開き、**registry.redhat.io** への参照をすべて特定します。以下に例を示します。

```
{
  "name": "1.9",
  "annotations": {
    "description": "Red Hat Fuse 7.9 Karaf S2I images.",
    "openshift.io/display-name": "Red Hat Fuse 7.9 Karaf",
    "iconClass": "icon-rh-integration",
    "tags": "builder,jboss-fuse,java,karaf,xpaas,hidden",
    "supports": "jboss-fuse:7.9.0,java:8,xpaas:1.2",
    "version": "1.9"
  },
  "referencePolicy": {
    "type": "Local"
  },
  "from": {
    "kind": "DockerImage",
    "name": "registry.redhat.io/fuse7/fuse-karaf-openshift-rhel8:1.9"
  }
},
```

3. ファイル内のすべての **registry.redhat.io** 参照を **psi-internal-registry** の名前に置き換えます。以下に例を示します。

```
{
  "name": "1.9",
  "annotations": {
    "description": "Red Hat Fuse 7.9 Karaf S2I images.",
    "openshift.io/display-name": "Red Hat Fuse 7.9 Karaf",
    "iconClass": "icon-rh-integration",
    "tags": "builder,jboss-fuse,java,karaf,xpaas,hidden",
    "supports": "jboss-fuse:7.9.0,java:8,xpaas:1.2",
    "version": "1.9"
  },
  "referencePolicy": {
    "type": "Local"
  },
  "from": {
    "kind": "DockerImage",
    "name": "docker-registry.upshift.redhat.com/fuse7/fuse-karaf-openshift-rhel8:1.9"
  }
},
```

- すべての参照が置き換えられたら、以下のコマンドを実行して Fuse on OpenShift イメージストリームをインストールします。

```
oc create -f fis-image-streams.json -n {namespace}
```

3.4. 内部 MAVEN リポジトリの使用

制限された環境では、別の Maven リポジトリを使用する必要があります。それは、**MAVEN_MIRROR_URL** という名前のテンプレートパラメーターを使用して指定できます。この **MAVEN_MIRROR_URL** パラメーターを使用して、コマンドラインから新規アプリケーションを作成することができます。

3.4.1. MAVEN_MIRROR_URL を使用した Spring Boot アプリケーションの実行

この例では、MAVEN_MIRROR_URL を使用して Spring Boot アプリケーションをデプロイおよび実行する方法について説明します。

手順

1. Spring Boot Camel XML クイックスタートをダウンロードします。

```
oc create -f {BASEURL}/quickstarts/spring-boot-2-camel-xml-template.json
```

2. 以下のコマンドを入力し、**MAVEN_MIRROR_URL** を使用して Spring Boot クイックスタートテンプレートの実行に必要なリソースを作成します。これにより、クイックスタートのデプロイメント設定およびビルド設定が作成されます。クイックスタートのデフォルトパラメーターや作成されたリソースの情報はターミナルに表示されます。

```
oc new-app s2i-fuse79-spring-boot-2-camel-xml -n {namespace} -p  
IMAGE_STREAM_NAMESPACE={namespace} -p MAVEN_MIRROR_URL={Maven mirror  
URL}
```

3.4.2. OpenShift Maven プラグインを使用した Spring Boot アプリケーションの実行

この例では、内部 Maven リポジトリを使用して OpenShift Maven プラグインで Spring Boot アプリケーションをデプロイおよび実行する方法を説明します。

手順

1. OpenShift Maven プラグインでクイックスタートを実行するには、ローカルリポジトリから Spring Boot 2 camel archetype をダウンロードし、クイックスタートをデプロイします。**{Maven Mirror URL}** を Maven ミラーリポジトリの URL に置き換えます。

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
  -DarchetypeCatalog={Maven Mirror URL}/archetypes/archetypes-catalog/2.2.0.fuse-sb2-  
790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-redhat-00004-archetype-  
catalog.xml \
  -DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
  -DarchetypeArtifactId=spring-boot-camel-xml-archetype  
-DarchetypeVersion=2.2.0.fuse-sb2-790047-redhat-00004
```

2. archetype プラグインが対話モードに切り替わり、残りのフィールドの入力を要求されます。

```
Define value for property 'groupid': : org.example.fis
Define value for property 'artifactId': : fuse79-spring-boot2
Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': org.example.fis: :
Confirm properties configuration:
groupid: org.example.fis
artifactId: fuse79-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y
```

3. 上記のコマンドが BUILD SUCCESS 状態で終了した場合、**fuse79-spring-boot2** サブディレクトリー内に新しい Fuse on OpenShift プロジェクトが作成されているはずです。
4. これで、**fuse79-spring-boot2** プロジェクトをビルドおよびデプロイできるようになりました。OpenShift にログインしている状態で、**fuse79-spring-boot2** プロジェクトのディレクトリーに移動し、以下のようにプロジェクトをビルドおよびデプロイします。

```
cd fuse79-spring-boot2
mvn oc:deploy -Popenshift
```

第4章 管理者でないユーザーでの FUSE ON OPENSIFT のインストール

アプリケーションを作成し、OpenShift にデプロイして Fuse on OpenShift の使用を開始することができます。最初に、Fuse on OpenShift イメージおよびテンプレートをインストールする必要があります。

4.1. 管理者でないユーザーでの FUSE ON OPENSIFT イメージおよびテンプレートのインストール

前提条件

- OpenShift サーバーへアクセスできる必要があります。CDK による仮想 OpenShift サーバーまたはリモート OpenShift サーバーのいずれかにアクセスできる必要があります。
- **registry.redhat.io** で認証を設定している。

詳細は、以下のドキュメントを参照してください。

- [コンテナイメージの registry.redhat.io を使用した認証](#)
- [Red Hat CDK 3.17 Getting Started Guide](#)

手順

1. Fuse on OpenShift プロジェクトのビルドおよびデプロイを準備するため、以下のように OpenShift サーバーにログインします。

```
oc login -u developer -p developer https://OPENSIFT_IP_ADDR:8443
```

IP アドレスは常に同じではないため、**OPENSIFT_IP_ADDR** は OpenShift サーバーの IP アドレスのプレースホルダーになります。この値を実際の IP アドレスに置き換えます。



注記

開発者パスワードを持つ開発者ユーザーは、CDK による仮想 OpenShift サーバーで自動作成される標準のアカウントです。リモートサーバーにアクセスする場合は、OpenShift 管理者が提供する URL とクレデンシャルを使用します。

2. **test** という名前のプロジェクト namespace が存在しない場合は作成します。

```
oc new-project test
```

test プロジェクト namespace がすでに存在する場合は、以下のコマンドを使用して切り替えます。

```
oc project test
```

3. Fuse on OpenShift イメージストリームをインストールします。

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005
```

```
oc create -n test -f ${BASEURL}/fis-image-streams.json
```

コマンド出力に、Fuse on OpenShift プロジェクトで使用できるようになった Fuse イメージストリームが表示されます。

4. クイックスタートテンプレートをインストールします。

```
for template in eap-camel-amq-template.json \
eap-camel-cdi-template.json \
eap-camel-cxf-jaxrs-template.json \
eap-camel-cxf-jaxws-template.json \
karaf-camel-amq-template.json \
karaf-camel-log-template.json \
karaf-camel-rest-sql-template.json \
karaf-cxf-rest-template.json ;
do
oc create -n test -f \
${BASEURL}/quickstarts/${template}
done
```

5. Spring Boot 2 のクイックスタートテンプレートをインストールします。

```
for template in spring-boot-2-camel-amq-template.json \
spring-boot-2-camel-config-template.json \
spring-boot-2-camel-drools-template.json \
spring-boot-2-camel-infinispan-template.json \
spring-boot-2-camel-rest-3scale-template.json \
spring-boot-2-camel-rest-sql-template.json \
spring-boot-2-camel-template.json \
spring-boot-2-camel-xa-template.json \
spring-boot-2-camel-xml-template.json \
spring-boot-2-cxf-jaxrs-template.json \
spring-boot-2-cxf-jaxws-template.json \
spring-boot-2-cxf-jaxrs-xml-template.json \
spring-boot-2-cxf-jaxws-xml-template.json ;
do oc create -n openshift -f \
https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005/quickstarts/${template}
done
```

6. Fuse Console のテンプレートをインストールします。

```
oc create -n test -f ${BASEURL}/fis-console-cluster-template.json
oc create -n test -f ${BASEURL}/fis-console-namespace-template.json
```



注記

Fuse Console のデプロイに関する詳細は、[OpenShift 4.x での Fuse Console の設定](#) を参照してください。

7. (任意手順): インストールされた Fuse on OpenShift イメージおよびテンプレートを表示します。

```
oc get template -n test
```

8. ブラウザーで OpenShift コンソールに移動します。
 - a. https://OPENSIFT_IP_ADDR:8443 を使用します。**OPENSIFT_IP_ADDR** は OpenShift サーバーの実際の IP アドレスに置き換えます。
 - b. クレデンシャル (例: ユーザー名 **developer** とパスワード **developer**) を使用して OpenShift コンソールにログインします。

第5章 開発者向けの基本情報

5.1. 開発環境の準備

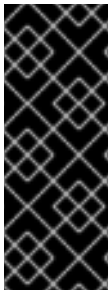
OpenShift サーバーにアクセスできることが、Fuse on OpenShift プロジェクトを開発およびテストするための基本要件になります。基本的な方法は次のとおりです。

- [ローカルマシンでの CDK \(Container Development Kit\) のインストール](#)
- [既存の OpenShift サーバーへのリモートアクセスを取得](#)

5.1.1. ローカルマシンでの CDK (Container Development Kit) のインストール

開発者は、すぐに開始したい場合は、[Red Hat CDK](#) をローカルマシンにインストールすることが最も実用的な方法です。CDK を使用すると、Red Hat Enterprise Linux (RHEL) 7 上の OpenShift のイメージを実行する仮想マシン (VM) インスタンスを起動できます。CDK のインストールは、次の主要コンポーネントで設定されます。

- 仮想マシン (libvirt、VirtualBox、または Hyper-V)
- コンテナ開発環境を開始および管理する minishift



重要

Red Hat CDK は **開発での使用のみ** を目的としています。本番環境などの他の環境での使用を目的としておらず、既知のセキュリティー脆弱性に対応しない場合があります。Docker 形式コンテナ内部でミッションクリティカルなアプリケーションを実行するためのフルサポートを受けるには、アクティブな RHEL 7 または RHEL Atomic サブスクリプションが必要です。詳細は、[Red Hat Container Development Kit \(CDK\) のサポート](#) を参照してください。

前提条件

- Java バージョン
開発者のマシンに、Fuse 7.12 がサポートするバージョンの Java がインストールされているようにしてください。サポートされる Java バージョンの詳細は [Red Hat Fuse でサポートされる設定](#) を参照してください。

手順

ローカルマシンに CDK をインストールするには、以下を行います。

1. Fuse on OpenShift では、バージョン 3.17 の CDK をインストールすることが推奨されます。CDK 3.17 をインストールおよび使用するための詳細手順は [Red Hat CDK 3.17 Getting Started Guide](#) を参照してください。
2. [コンテナイメージの registry.redhat.io を使用した認証](#) の手順に従って、Red Hat Ecosystem Catalog へのアクセスを取得できるように、OpenShift クレデンシャルを設定します。
3. [2章 管理者向けの基本情報](#) の説明にしたがって、Fuse on OpenShift イメージおよびテンプレートを手作業でインストールします。



注記

CDK のバージョンには Fuse on OpenShift イメージおよびテンプレートが事前にインストールされている可能性があります。ただし、OpenShift クレデンシャルの設定後に、Fuse on OpenShift イメージおよびテンプレートをインストール (または更新) する必要があります。

4. 本章の例を参照する前に、[Red Hat CDK 3.17 Getting Started Guide](#) の内容を読み、理解するようにしてください。

5.1.2. 既存の OpenShift サーバーへのリモートアクセスの取得

IT 部門によってサーバーマシン上に OpenShift クラスターがすでに設定されている可能性があります。この場合、Fuse on OpenShift を使用するには以下の要件を満たしている必要があります。

- サーバーマシンがサポートされるバージョンの OpenShift Container Platform を実行している必要があります ([Red Hat Fuse でサポートされる設定](#) を参照)。本ガイドの例は、バージョン 3.11 に対してテストされています。
- OpenShift 管理者に、最新の Fuse on OpenShift コンテナベースイメージと Fuse on OpenShift テンプレートを OpenShift サーバーにインストールするよう依頼します。
- OpenShift 管理者に、通常の開発者権限 (OpenShift プロジェクトを作成、デプロイ、および実行できる権限) を持つユーザーアカウントを作成してもらいます。
- 管理者に、OpenShift サーバーの URL (OpenShift コンソールの閲覧または **oc** コマンドラインクライアントを使用した OpenShift への接続に使用) と、アカウントのログインクレデンシャルを提供するよう依頼します。

5.1.3. クライアント側ツールのインストール

開発者のマシンに以下のツールをインストールすることが推奨されます。

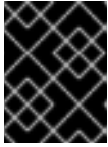
- Apache Maven 3.6.x: OpenShift プロジェクトのローカルビルドに必要です。[Apache Maven download](#) ページから適切なパッケージをダウンロードします。バージョン 3.6.x 以上がインストールされているようにしてください。3.6.x 未満のバージョンを使用すると、プロジェクトのビルド時に依存関係の解決に関する問題が発生する可能性があります。
- Git: OpenShift S2I ソースワークフローに必要で、通常 Fuse on OpenShift プロジェクトのソース制御用に推奨されます。[Git Downloads](#) ページから適切なパッケージをダウンロードします。
- OpenShift client: CDK を使用している場合、シェルに入力する必要があるコマンドを表示する **minishift oc-env** を使用して、**oc** バイナリーを PATH に追加します。**oc-env** の出力は、OS とシェルのタイプによって異なります。

```
$ minishift oc-env
export PATH="/Users/john/.minishift/cache/oc/v1.5.0:$PATH"
# Run this command to configure your shell:
# eval $(minishift oc-env)
```

詳細は、CDK 3.17 [Getting Started Guide](#) の [Using the OpenShift Client Binary](#) を参照してください。

CDK を使用しない場合は、[CLI Reference](#) の手順にしたがって、**oc** クライアントツールをインストールします。

- (任意) Docker client: 上級ユーザーは Docker クライアントツールがインストールされていると便利である可能性があります (OpenShift サーバー上で実行されている docker デーモンとの通信に使用)。オペレーティングシステム用の特定のバイナリーインストールに関する詳細は、[Docker インストール](#) サイトを参照してください。
詳細は、CDK 3.17 **Getting Started Guide** の [Reusing the docker Daemon](#) を参照してください。



重要

OpenShift サーバーで稼働しているバージョンの OpenShift と互換性のあるバージョンの **oc** および **docker** ツールをインストールするようにしてください。

その他のリソース

(任意) Red Hat JBoss CodeReady Studio: **Red Hat JBoss CodeReady Studio** は、Fuse on OpenShift アプリケーションの開発サポートが含まれる Eclipse ベースの開発環境です。この開発環境のインストール方法に関する詳細は、[Red Hat JBoss CodeReady Studio のインストールガイド](#) を参照してください。

5.1.4. Maven リポジトリの設定

ローカルマシンで Fuse on OpenShift プロジェクトをビルドするのに必要な archetype とアーティファクトを保持する Maven リポジトリを設定します。

手順

1. Maven の **settings.xml** ファイルを開きます。このファイルは通常、Linux または MacOS の場合は `~/.m2/settings.xml`、Windows の場合は **Documents and Settings\<USER_NAME>\.m2\settings.xml** にあります。
2. 以下の Maven リポジトリを追加します。
 - Maven central: <https://repo1.maven.org/maven2>
 - Red Hat GA リポジトリ: <https://maven.repository.redhat.com/ga>
 - Red Hat EA リポジトリ: <https://maven.repository.redhat.com/earlyaccess/all>
上記のリポジトリは、**settings.xml** ファイルの依存関係リポジトリセクションと、プラグインリポジトリセクションの両方に追加する必要があります。

5.2. FUSE ON OPENSIFT でのアプリケーションの作成およびデプロイ

以下の OpenShift Source-to-Image (S2I) アプリケーション開発ワークフローの1つを使用してアプリケーションを作成し、OpenShift にデプロイして、Fuse on OpenShift の使用を開始することができます。

S2I バイナリーワークフロー

ビルド入力が **バイナリーソース** からの S2I です。このワークフローの特徴は、ビルドの一部が開発者自身のマシンで実行されることです。このワークフローはバイナリーパッケージをローカルでビルドした後、バイナリーパッケージを OpenShift に渡します。詳細は、OpenShift Container Platform **Builds** ガイドの [Binary Source](#) を参照してください。

S2I ソースワークフロー

ビルド入力が **Git** ソース からの S2I です。このワークフローの特徴は、ビルドがすべて OpenShift サーバー上でビルドされることです。詳細は、OpenShift Container Platform Builds ガイドの [Git Source](#) を参照してください。

5.2.1. S2I バイナリーを使用したアプリケーションの作成およびデプロイ

ここでは、OpenShift S2I バイナリーワークフローを使用して、Fuse on OpenShift プロジェクトを作成、ビルド、およびデプロイします。



注記

JDK11 を使用したクイックスタートの実行

ランタイム時に JDK11 ベースのイメージを使用する場合は、コンパイル時に正しい JDK11 プロファイルを使用します。JDK11 を使用してクイックスタートをビルドおよびデプロイする場合は、ビルドマシンに JDK11 をインストールし、正しい JDK11 プロファイルを使用してクイックスタートをビルドするようにしてください。

手順

1. Maven の archetype を使用して、新しい Fuse on OpenShift プロジェクトを作成します。この例では、サンプル Spring Boot Camel プロジェクトを作成する archetype を使用します。新しいシェルプロンプトを開き、以下の Maven コマンドを入力します。

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
-
DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabric8/archetypes/archetypes-
catalog/2.2.0.fuse-sb2-790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-
redhat-00004-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-xml-archetype \
-DarchetypeVersion=2.2.0.fuse-sb2-790047-redhat-00004
```

archetype プラグインが対話モードに切り替わり、残りのフィールドの入力を要求されます。

```
Define value for property 'groupId': : org.example.fis
Define value for property 'artifactId': : fuse79-spring-boot
Define value for property 'version': : 1.0-SNAPSHOT: :
Define value for property 'package': : org.example.fis: :
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse79-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y
```

プロンプトが表示されたら、**org.example.fis** を **groupId** の値として入力し、**fuse79-spring-boot** を **artifactId** の値として入力します。残りのフィールドにはデフォルト値を使用します。

2. 前のコマンドが **BUILD SUCCESS** 状態で終了した場合、**fuse79-spring-boot** サブディレクトリ内に新しい Fuse on OpenShift プロジェクトが作成されているはずです。**fuse79-spring-boot/src/main/resources/spring/camel-context.xml** ファイルの XML DSL コードを確認します。デモンストラーションコードは、乱数が含まれるメッセージを継続的にログに送信する、シンプルな Camel ルートを定義します。

3. Fuse on OpenShift プロジェクトのビルドおよびデプロイを準備するため、以下のように OpenShift サーバーにログインします。

```
oc login -u developer -p developer https://OPENSHIFT_IP_ADDR:8443
```

IP アドレスは常に同じではないため、**OPENSHIFT_IP_ADDR** は OpenShift サーバーの IP アドレスのプレースホルダーになります。この値を実際の IP アドレスに置き換えます。



注記

developer パスワードを持つ **developer** ユーザーは、CDK による仮想 OpenShift サーバーで自動作成される標準のアカウントです。リモートサーバーにアクセスする場合は、OpenShift 管理者が提供する URL とクレデンシャルを使用します。

4. 次のように、**openshift** プロジェクトに切り替えます (まだ **openshift** プロジェクトにない場合)。

```
oc project openshift
```

5. 以下のコマンドを実行し、Fuse on OpenShift イメージおよびテンプレートがすでにインストールされ、アクセス可能であることを確認します。

```
oc get template -n openshift
```

イメージおよびテンプレートが事前にインストールされていない場合や、提供されたバージョンが古い場合は、Fuse on OpenShift イメージおよびテンプレートを手作業でインストール (または更新) します。Fuse on OpenShift イメージのインストール方法に関する詳細は、[2章 管理者向けの基本情報](#) を参照してください。

6. これで、**fuse79-spring-boot** プロジェクトをビルドおよびデプロイできるようになりました。OpenShift にログインしている状態で、**fuse79-spring-boot** プロジェクトのディレクトリーに移動し、以下のようにプロジェクトをビルドおよびデプロイします。

```
cd fuse79-spring-boot
mvn oc:deploy -Popenshift
```

ビルドに成功すると、以下のような出力が表示されます。

```
...
[INFO] OpenShift platform detected
[INFO] Using project: openshift
[INFO] Creating a Service from openshift.yml namespace openshift name fuse79-spring-boot
[INFO] Created Service: target/jkube/applyJson/openshift/service-fuse79-spring-boot.json
[INFO] Using project: openshift
[INFO] Creating a DeploymentConfig from openshift.yml namespace openshift name fuse79-spring-boot
[INFO] Created DeploymentConfig: target/jkube/applyJson/openshift/deploymentconfig-fuse79-spring-boot.json
[INFO] Creating Route openshift:fuse79-spring-boot host: null
[INFO] F8: HINT: Use the command `oc get pods -w` to watch your pods start up
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

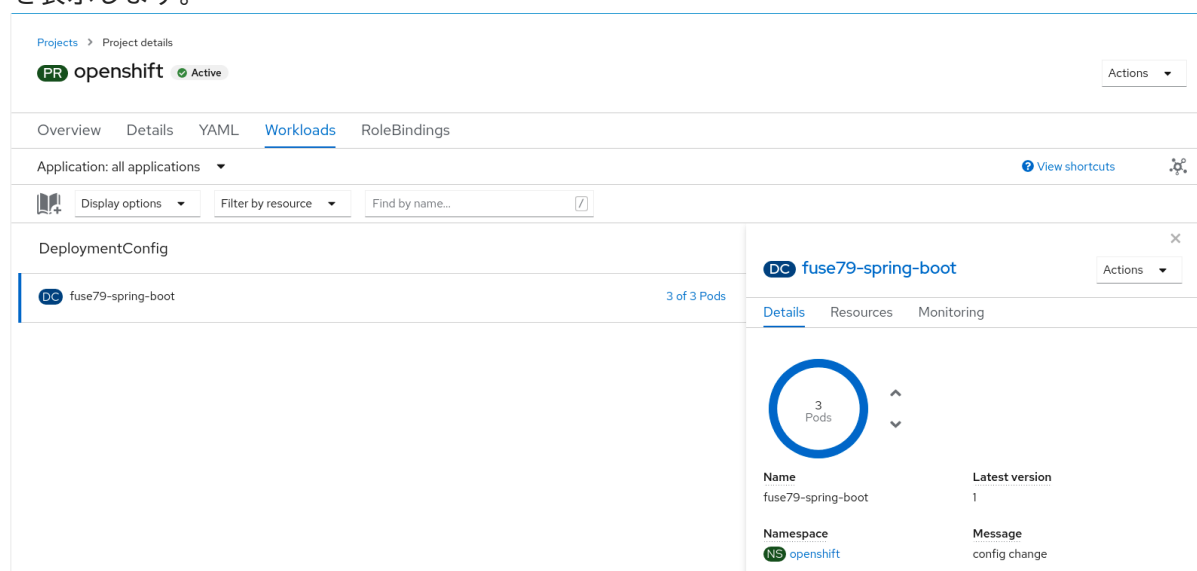
```
[INFO] Total time: 05:38 min
[INFO] Finished at: 2020-12-04T12:15:06+05:30
[INFO] Final Memory: 63M/688M
[INFO] -----
```



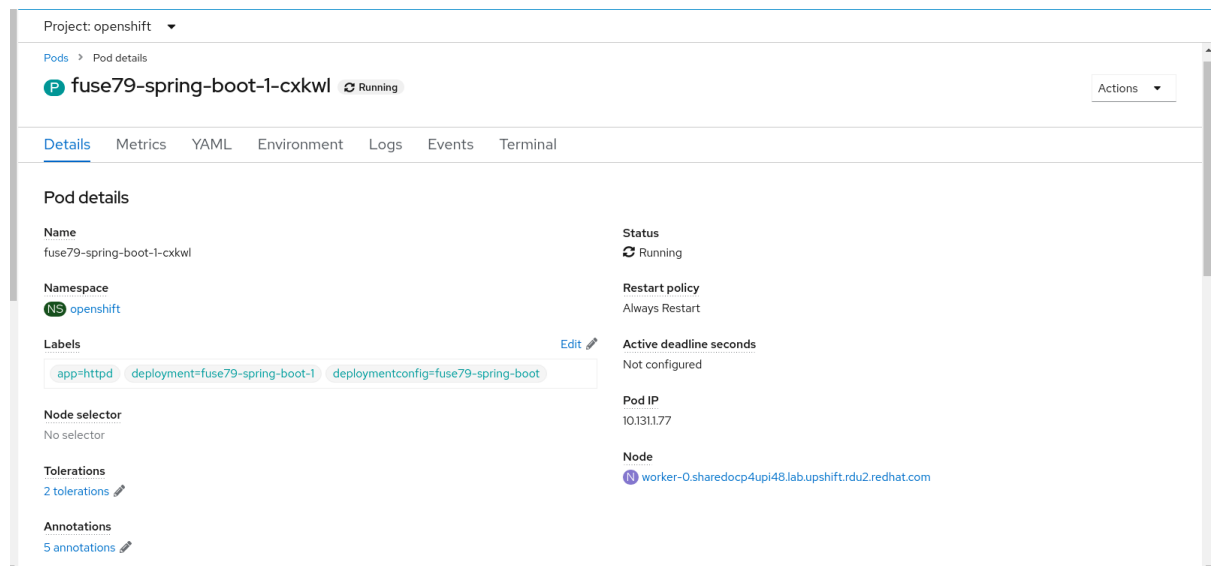
注記

このコマンドを初めて実行する場合、Maven は多くの依存関係をダウンロードする必要があるため、数分かかることがあります。2 回目からはビルドが速くなります。

7. ブラウザーで OpenShift コンソールに移動し、クレデンシャル (例: ユーザー名 **developer** およびパスワード **developer**) を使用してコンソールにログインします。
8. 左側のパネルで **Home** を展開し、**Status** をクリックして **openshift** プロジェクトの Project Status ページを表示します。
9. **fuse75-spring-boot** をクリックし、**fuse75-spring-boot** アプリケーションの概要情報ページを表示します。




10. 左側のパネルで **Workloads** を展開します。
11. **Pods** をクリックします。**openshift** プロジェクトで稼働している Pod すべてが表示されます。
12. Pod の **Name** (この例では **fuse79-spring-boot-xxxxx**) をクリックし、稼働中の Pod の詳細を表示します。



13. **Logs** タブをクリックしてアプリケーションログを表示し、ログを下方方向にスクロールして、Camel アプリケーションによって生成された乱数のログメッセージを見つけます。

```
...
06:45:54.311 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 130
06:45:56.265 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 898
06:45:58.265 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 414
06:46:00.265 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 486
06:46:02.265 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 093
06:46:04.265 [Camel (MyCamel) thread #1 - timer://foo] INFO simple-route - >>> 080
```

14. 稼働中の Pod を終了するには以下を行います。
- openshift** プロジェクトの Project Status ページで、**fuse79-spring-boot** アプリケーションをクリックします。
 - Overview** タブをクリックし、アプリケーションの概要情報ページを表示します。
 - Desired Count の横にある  アイコンをクリックします。Edit Count ウィンドウが表示されます。
 - 下矢印を使用して値をゼロにし、Pod を停止します。

5.2.2. プロジェクトのアンデプロイおよび再デプロイ

以下のようにプロジェクトをアンデプロイまたは再デプロイできます。

手順

- プロジェクトをアンデプロイするには、以下のコマンドを入力します。

```
mvn oc:undeploy
```

- プロジェクトを再デプロイするには、以下のコマンドを入力します。

```
mvn oc:undeploy
mvn oc:deploy -Popenshift
```

5.2.3. S2I ソースワークフローを使用したアプリケーションの作成およびデプロイ

ここでは、OpenShift S2I ソースワークフローを使用して、テンプレートをベースとした Fuse on OpenShift アプリケーションをビルドおよびデプロイします。最初に、リモート Git リポジトリに保存されたクイックスタートプロジェクトを使用します。OpenShift コンソールを使用すると、このクイックスタートプロジェクトを OpenShift サーバーでダウンロード、ビルド、およびデプロイできます。

手順

1. 次のように、OpenShift サーバーにログインします。

```
oc login -u developer -p developer https://OPENSIFT_IP_ADDR:8443
```

IP アドレスは常に同じではないため、**OPENSIFT_IP_ADDR** は OpenShift サーバーの IP アドレスのプレースホルダーになります。この値を実際の IP アドレスに置き換えます。



注記

developer パスワードを持つ **developer** ユーザーは、CDK による仮想 OpenShift サーバーで自動作成される標準のアカウントです。リモートサーバーにアクセスする場合は、OpenShift 管理者が提供する URL とクレデンシャルを使用します。

2. 次のように、**openshift** プロジェクトに切り替えます (まだ **openshift** プロジェクトにない場合)。

```
oc project openshift
```

3. 以下のコマンドを実行し、Fuse on OpenShift テンプレートがすでにインストールされ、アクセス可能であることを確認します。

```
oc get template -n openshift
```

イメージおよびテンプレートが事前にインストールされていない場合や、提供されたバージョンが古い場合は、Fuse on OpenShift イメージおよびテンプレートを手作業でインストール (または更新) します。Fuse on OpenShift イメージのインストール方法に関する詳細は、[2章 管理者向けの基本情報](#) を参照してください。

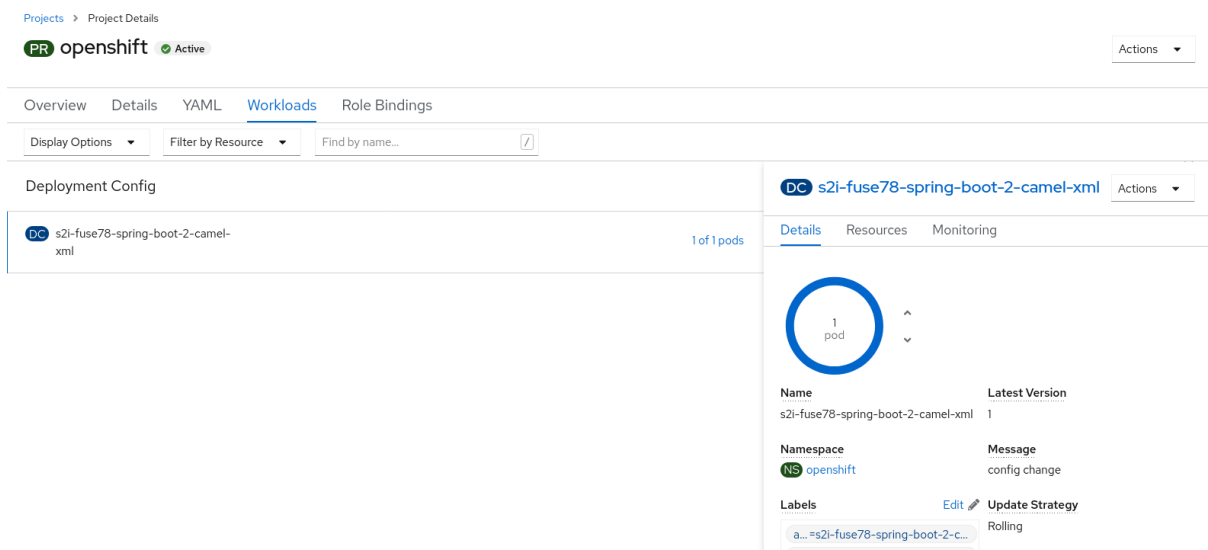
4. 以下のコマンドを入力し、**Red Hat Fuse 7.12 Camel XML DSL with Spring Boot** クイックスタートテンプレートの実行に必要なリソースを作成します。これにより、クイックスタートのデプロイメント設定およびビルド設定が作成されます。クイックスタートのデフォルトパラメーターや作成されたリソースの情報はターミナルに表示されます。

```
oc new-app s2i-fuse7-spring-boot-camel-xml

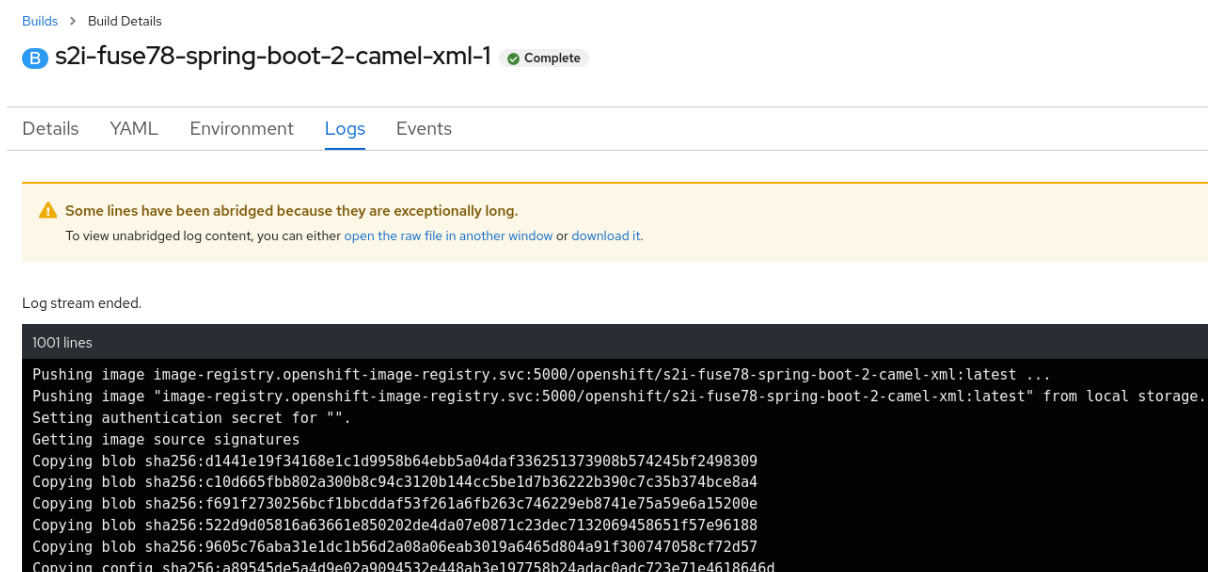
--> Deploying template "openshift/s2i-fuse7-spring-boot-camel-xml" to project openshift
...
--> Creating resources ...
    imagestream.image.openshift.io "s2i-fuse7-spring-boot-camel-xml" created
```

```
buildconfig.build.openshift.io "s2i-fuse7-spring-boot-camel-xml" created
deploymentconfig.apps.openshift.io "s2i-fuse7-spring-boot-camel-xml" created
--> Success
Build scheduled, use 'oc logs -f bc/s2i-fuse7-spring-boot-camel-xml' to track its progress.
Run 'oc status' to view your app.
```

5. ブラウザーで https://OPENSIFT_IP_ADDR の OpenShift Web コンソールに移動します (OPENSIFT_IP_ADDR はクラスターの IP アドレスに置き換えます)。クレデンシャル (例: ユーザー名 **developer**、パスワード **developer**) を使用して、コンソールにログインします。
6. 左側のパネルで **Home** を展開します。**Status** をクリックして **Project Status** ページを表示します。選択された namespace (例: **openshift**) の既存のアプリケーションがすべて表示されます。
7. **s2i-fuse7-spring-boot-camel-xml** をクリックし、クイックスタートの **Overview** 情報ページを表示します。




8. **Resources** タブをクリックした後に **View logs** をクリックし、アプリケーションのビルドログを表示します。



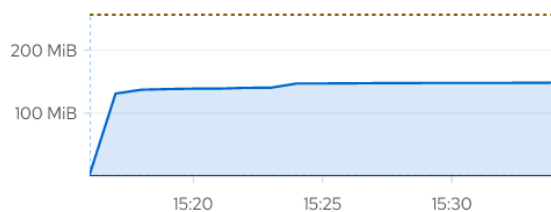
9. 左側のパネルで **Workloads** を展開します。
10. **Pods** をクリックした後、**s2i-fuse7-spring-boot-camel-xml-xxxx** をクリックします。アプリケーションの Pod の詳細が表示されます。

Pods > Pod Details

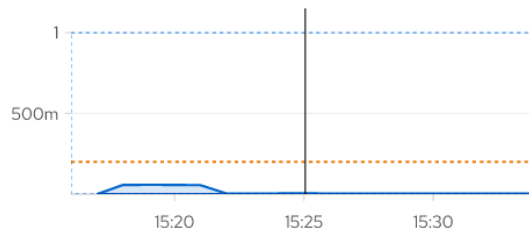
P s2i-fuse78-spring-boot-2-camel-xml-1-czrx2  Running[Details](#) [YAML](#) [Environment](#) [Logs](#) [Events](#) [Terminal](#)

Pod Details

Memory Usage



CPU Usage



Network In



Network Out



11. 稼働中の Pod を終了するには以下を行います。

- a. **openshift** プロジェクトの Project Status ページで、**s2i-fuse7-spring-boot-camel-xml-xxxx** アプリケーション をクリックします。
- b. **Overview** タブをクリックし、アプリケーションの概要情報ページを表示します。
- c. Desired Count の横にある  アイコンをクリックします。Edit Count ウィンドウが表示されます。
- d. 下矢印を使用して値をゼロにし、Pod を停止します。

第6章 SPRING BOOT イメージのアプリケーションの開発

本章では、Spring Boot イメージのアプリケーションを開発する方法を説明します。

6.1. MAVEN ARCHETYPE を使用した SPRING BOOT 2 プロジェクトの作成

このクイックスタートでは、Maven archetype を使用して Spring Boot 2 プロジェクトを作成する方法を実証します。

手順

1. システムの適切なディレクトリーに移動します。
2. Shell プロンプトに以下の **mvn** コマンドを入力し、Spring Boot 2 プロジェクトを作成します。

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
-
DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabric8/archetypes/archetypes-
catalog/2.2.0.fuse-sb2-790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-
redhat-00004-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-xml-archetype \
-DarchetypeVersion=2.2.0.fuse-sb2-790047-redhat-00004
```

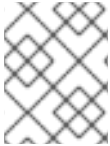
archetype プラグインが対話モードに切り替わり、残りのフィールドの入力を要求されます。

```
Define value for property 'groupId': : org.example.fis
Define value for property 'artifactId': : fuse79-spring-boot
Define value for property 'version': : 1.0-SNAPSHOT: :
Define value for property 'package': : org.example.fis: :
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse79-spring-boot
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y
```

プロンプトが表示されたら、**org.example.fis** を **groupId** の値として入力し、**fuse79-spring-boot** を **artifactId** の値として入力します。残りのフィールドにはデフォルト値を使用します。

3. 上記のコマンドが BUILD SUCCESS 状態で終了した場合、**fuse79-spring-boot** サブディレクトリー内に新しい Fuse on OpenShift プロジェクトが作成されているはずです。
4. これで、**fuse79-spring-boot** プロジェクトをビルドおよびデプロイできるようになりました。OpenShift にログインしている状態で、**fuse79-spring-boot** プロジェクトのディレクトリーに移動し、以下のようにプロジェクトをビルドおよびデプロイします。

```
cd fuse79-spring-boot
mvn oc:deploy -Popenshift
```

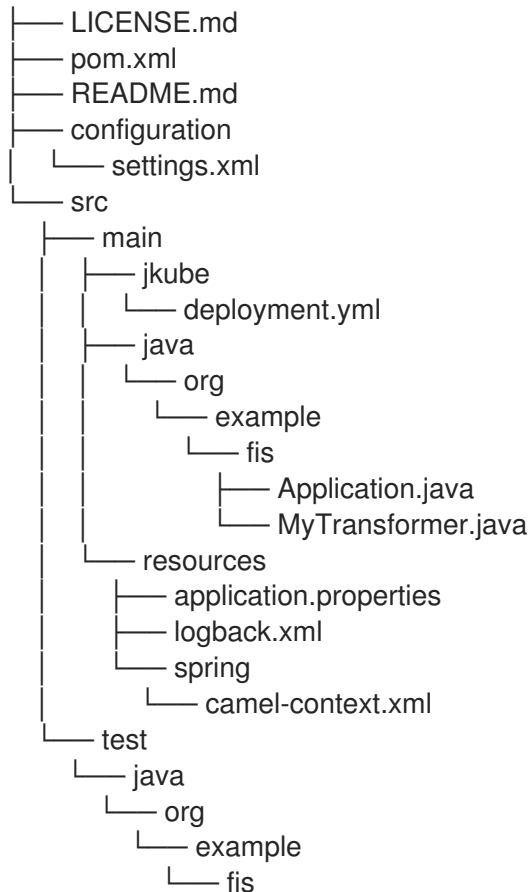


注記

使用できる Spring Boot 2 archetype の完全リストは、[Spring Boot 2 archetype カタログ](#) を参照してください。

6.2. CAMEL SPRING BOOT アプリケーションの構造

Camel Spring Boot アプリケーションのディレクトリー構造は以下のようになります。



このうち、アプリケーションの開発に重要なファイルは次のとおりです。

pom.xml

追加の依存関係が含まれます。Spring Boot と互換性のある Camel コンポーネントは、**camel-jdbc-starter** や **camel-infinispan-starter** などのスターターバージョンを利用できます。**pom.xml** にスターターが含まれると、スターターは自動的に設定され、起動時に Camel コンテンツと登録されます。**application.properties** ファイルを使用すると、コンポーネントのプロパティーを設定できます。

application.properties

設定の外部化を可能にし、異なる環境で同じアプリケーションコードを使って作業できるようになります。詳細は [Externalized Configuration](#) を参照してください。

たとえば、この Camel アプリケーションでアプリケーションの名前や IP アドレスなどの特定のプロパティーを設定できます。

application.properties

```
#spring.main.sources=org.example.fos

logging.config=classpath:logback.xml
```

```
# the options from org.apache.camel.spring.boot.CamelConfigurationProperties can be configured
here
camel.springboot.name=MyCamel

# lets listen on all ports to ensure we can be invoked from the pod IP
server.address=0.0.0.0
management.address=0.0.0.0

# lets use a different management port in case you need to listen to HTTP requests on 8080
management.server.port=8081

# disable all management endpoints except health
endpoints.enabled = false
endpoints.health.enabled = true
```

Application.java

これはアプリケーションを実行するために重要なファイルです。ユーザーとして **camel-context.xml** ファイルをインポートし、Spring DSL を使用してルートを設定します。

Application.java file は **@SpringBootApplication** アノテーションを指定します。このアノテーションは、デフォルトの属性を持つ **@Configuration**、**@EnableAutoConfiguration**、および **@ComponentScan** と同等です。

Application.java

```
@SpringBootApplication
// load regular Spring XML file from the classpath that contains the Camel XML DSL
@ImportResource({"classpath:spring/camel-context.xml"})
```

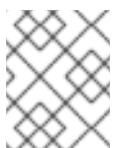
Spring Boot アプリケーションを実行するには **main** メソッドが必要です。

Application.java

```
public class Application {
    /**
     * A main method to start this application.
     */
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

camel-context.xml

src/main/resources/spring/camel-context.xml は、Camel ルートが含まれる、アプリケーションの開発に重要なファイルです。



注記

Spring Boot アプリケーションの開発に関する詳細は、[Developing your first Spring Boot Application](#) を参照してください。

src/main/jkube/deployment.yml

openshift-maven-plugin によって生成されるデフォルトの OpenShift 設定ファイルにマージされる追加設定を提供します。



注記

このファイルは Spring Boot アプリケーションの一部として使用されませんが、CPU やメモリー使用量などのリソースを制限するためにすべてのクイックスタートで使用されます。

6.3. SPRING BOOT 2 ARCHETYPE カタログ

Spring Boot 2 archetype カタログには、以下の例が含まれます。

表6.1 Spring Boot 2 Maven archetype

名前	説明
spring-boot-camel-archetype	fabric8 Java ベースイメージを基にして Apache Camel を Spring Boot と使用する方法を実証します。
spring-boot-camel-amq-archetype	Spring Boot アプリケーションを ActiveMQ ブローカーに接続する方法と、Kubernetes または OpenShift を使用して 2 つの Camel ルートの間で JMS メッセージングを使用する用法を実証します。
spring-boot-camel-drools-archetype	Apache Camel を使用して、リモート Kie Server で Kubernetes または OpenShift で実行している Spring Boot アプリケーションを統合する方法を実証します。
spring-boot-camel-infinispan-archetype	Hot Rod プロトコルを使用して Spring Boot アプリケーションを JBoss Data Grid または Infinispan サーバーに接続する方法を実証します。
spring-boot-camel-rest-3scale-archetype	Camel の REST DSL を使用して RESTful API を公開し、それを 3scale に公開する方法を実証します。
spring-boot-camel-rest-sql-archetype	Camel の REST DSL とともに JDBC を介して SQL を使用し、RESTful API を公開する方法を実証します。
spring-boot-camel-xml-archetype	Spring XML 設定ファイルより、Spring Boot の Camel ルートを設定する方法を実証します。
spring-boot-cxf-jaxrs-archetype	fabric8 Java ベースイメージを基にして Spring Boot で Apache CXF を使用する方法を実証します。クイックスタートは Spring Boot を使用して、Swagger が有効な CXF JAXRS エンドポイントが含まれるアプリケーションを設定します。

名前	説明
spring-boot-cxf-jaxws-archetype	fabric8 Java ベースイメージを基にして Spring Boot で Apache CXF を使用する方法を実証します。クイックスタートは Spring Boot を使用して、CXF JAXWS エンドポイントが含まれるアプリケーションを設定します。
spring-boot-cxf-jaxrs-xml-archetype	OpenShift 上の Spring Boot 2 と Apache CXF JAX-RS を使用する方法を実証します。このクイックスタートは Spring Boot2 を使用して、Swagger が有効な CXF JAXRS エンドポイントが含まれる Spring 設定ファイルベースの CXF アプリケーションを起動します。
spring-boot-cxf-jaxws-xml-archetype	OpenShift 上の Spring Boot 2 と Apache CXF JAX-WS を使用する方法を実証します。このクイックスタートは Spring Boot2 を使用して、CXF JAXWS エンドポイントが含まれる Spring 設定ファイルベースの CXF アプリケーションを起動します。

注記

以下の Spring Boot 2 の Maven archetype は、OpenShift でビルドおよびデプロイできません。詳細は [リリースノート](#) を参照してください。

- **spring-boot-camel-archetype**
- **spring-boot-camel-infinispan-archetype**
- **spring-boot-cxf-jaxrs-archetype**
- **spring-boot-cxf-jaxws-archetype**

この問題を回避するには、これらのクイックスタートの1つに Maven プロジェクトを生成した後に、プロジェクトの Maven **pom.xml** ファイルを編集し、以下の依存関係を追加します。

```
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <version>2.4.1</version>
  <scope>test</scope>
</dependency>
```

6.4. SPRING BOOT の BOM ファイル

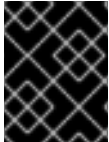
Maven BOM (Bill of Materials) ファイルの目的は、正常に動作する Maven 依存関係バージョンのセットを提供し、各 Maven アーティファクトに対して個別にバージョンを定義する必要をなくすることです。

重要

使用している Spring Boot のバージョンに適した Fuse BOM を使用するようにしてください。

Spring Boot の Fuse BOM には以下の利点があります。

- Maven 依存関係のバージョンを定義するため、依存関係を POM に追加するときにバージョンを指定する必要がありません。
- 特定バージョンの Fuse に対して完全にテストされ、完全にサポートする依存関係のセットを定義します。
- Fuse のアップグレードを簡素化します。



重要

Fuse BOM によって定義される依存関係のセットのみが Red Hat によってサポートされます。

6.5. BOM ファイルの組み込み

Maven プロジェクトに BOM ファイルを組み込むには、以下の Spring Boot 2 の例のように、プロジェクトの **pom.xml** ファイル (または親 POM ファイル内の) **dependencyManagement** 要素を指定します。

- [Spring Boot 2 の BOM](#)

Spring Boot 2 の BOM

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project ...>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <!-- configure the versions you want to use here -->
    <fuse.version>7.9.0.fuse-sb2-790065-redhat-00001</fuse.version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>fuse-springboot-bom</artifactId>
        <version>${fuse.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

依存関係管理のメカニズムを使用して BOM を指定した後、アーティファクトのバージョンを指定しなくても、Maven 依存関係を POM に追加できるようになります。たとえば、**camel-hystrix** コンポーネントの依存関係を追加するには、以下の XML フラグメントを POM の **dependencies** 要素に追加します。

```
<dependency>
```

```
<groupId>org.apache.camel</groupId>
<artifactId>camel-hystrix-starter</artifactId>
</dependency>
```

Camel アーティファクト ID が **-starter** 接尾辞とともに追加されていることに注意してください。つまり、Camel Hystrix コンポーネントを **camel-hystrix** ではなく **camel-hystrix-starter** として指定します。Camel スターターコンポーネントは、Spring Boot 環境に対して最適化されるようにパッケージ化されています。

6.6. SPRING BOOT MAVEN プラグイン

Spring Boot Maven プラグインは Spring Boot によって提供されます。これは、Spring Boot プロジェクトをビルドおよび実行するための開発者ユーティリティです。

- **ビルド**: プロジェクトディレクトリーでコマンド **mvn package** を入力し、Spring Boot アプリケーションの実行可能な Jar パッケージを作成します。ビルドの出力は、Maven プロジェクトの **target/** サブディレクトリーに格納されます。
- **実行**: 新規ビルドされたアプリケーションは **mvn spring-boot:start** コマンドで実行することができます。

Spring Boot Maven プラグインをプロジェクトの POM ファイルに組み込むには、以下の例のように、プラグイン設定を **pom.xml** ファイルの **project/build/plugins** セクションに追加します。

例

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project ...>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <!-- configure the versions you want to use here -->
    <fuse.version>7.9.0.fuse-sb2-790065-redhat-00001</fuse.version>

  </properties>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>${fuse.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

第7章 SPRING BOOT での APACHE CAMEL アプリケーションの実行

Apache Camel Spring Boot コンポーネントは、自動的に Camel コンテキストを Spring Boot に設定します。Camel コンテキストの自動設定によって、Spring コンテキストで使用できる Camel ルートが自動検出され、プロデューサーテンプレート、コンシューマーテンプレート、タイプコンバーターなどの主な Camel ユーティリティーが Bean として登録されます。Apache Camel コンポーネントには、スターターを使用して Spring Boot アプリケーションを開発できるようにする Spring Boot スターターモジュールが含まれます。

7.1. CAMEL SPRING BOOT コンポーネント

Camel Spring Boot アプリケーションはすべてプロジェクトの **pom.xml** にある **dependencyManagement** を使用して、依存関係の製品化バージョンを指定する必要があります。これらの依存関係は Red Hat Fuse BOM で定義され、特定バージョンの Red Hat Fuse でサポートされます。BOM からのバージョンをオーバーライドしないようにするため、追加のスターターのバージョン番号を省略することができます。詳細は、[quickstart pom](#) を参照してください。

例

```
<dependencyManagement>
<dependencies>
<dependency>
<groupId>org.jboss.redhat-fuse</groupId>
<artifactId>fuse-springboot-bom</artifactId>
<version>${fuse.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```



注記

camel-spring-boot jar には **spring.factories** ファイルが含まれています。このファイルにより、依存関係をクラスパスに追加できるため、Spring Boot によって Camel コンテキストが自動的に設定されます。

7.2. CAMEL SPRING BOOT スターターモジュール

スターターは、Spring Boot アプリケーションでの使用を目的とする Apache Camel モジュールです。「[スターターモジュールのない Camel コンポーネント一覧](#)」に記載されている一部の例外を除き、**camel-xxx-starter** モジュールは各 Camel コンポーネントにあります。

スターターは以下の要件を満たしています。

- IDE ツールと互換性のあるネイティブ Spring Boot 設定システムを使用して、コンポーネントの自動設定を可能にします。
- データ形式および言語の自動設定を可能にします。
- 推移的なログの依存関係を管理し、Spring Boot ロギングシステムと統合します。

- 追加の依存関係を含め、推移的な依存関係を合わせることで Spring Boot アプリケーションを作成するための労力を最小限にします。

各スターターの **tests/camel-itest-spring-boot** に独自の統合テストがあり、現在のリリースの Spring Boot との互換性を検証します。



注記

詳細は、[Apache Camel Spring-Boot examples](#) を参照してください。

7.3. スターターモジュールのない CAMEL コンポーネント一覧

互換性の問題があるため、以下のコンポーネントにはスターターモジュールがありません。

- **camel-blueprint** (OSGi のみを対象)
- **camel-cdi** (CDI のみを対象)
- **camel-core-osgi** (OSGi のみを対象)
- **camel-ejb** (JEE のみを対象)
- **camel-eventadmin** (OSGi のみを対象)
- **camel-ibatis** (**camel-mybatis-starter** が含まれます)
- **camel-jclouds**
- **camel-mina** (**camel-mina2-starter** が含まれます)
- **camel-paxlogging** (OSGi のみを対象)
- **camel-quartz** (**camel-quartz2-starter** が含まれます)
- **camel-spark-rest**
- **camel-openapi-java** (**camel-openapi-java-starter** が含まれます)

7.4. CAMEL SPRING BOOT スターターの使用

Apache Camel では、Spring Boot アプリケーションをすぐに開発できるようにするスターターモジュールが提供されます。

手順

1. 以下の依存関係を Spring Boot の pom.xml に追加します。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-boot-starter</artifactId>
</dependency>
```

2. 以下のスニペットのように、Camel ルートでクラスを追加します。これらのルートがクラスパスに追加されると、ルートは自動的に開始されます。

```
package com.example;

import org.apache.camel.builder.RouteBuilder;
import org.springframework.stereotype.Component;

@Component
public class MyRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("timer:foo")
            .to("log:bar");
    }
}
```

3. 任意手順:Camel の稼働を維持するためにメインスレッドがブロックされた状態を維持するには、以下の1つを行います。
 - a. **spring-boot-starter-web** 関係が含まれるようにします。
 - b. または、**camel.springboot.main-run-controller=true** を **application.properties** または **application.yml** ファイルに追加します。
application.properties または **application.yml** ファイルで **camel.springboot.*properties** を使用すると Camel アプリケーションをカスタマイズできます。
4. 任意手順:Bean の ID 名を使用してカスタム Bean を参照するには、**src/main/resources/application.properties** または **application.yml** ファイルのオプションを設定します。以下の例は、Bean ID を使用して xslt コンポーネントがカスタム Bean を参照する方法を示しています。
 - a. ID **myExtensionFactory** でカスタム Bean を参照します。

```
camel.component.xslt.saxon-extension-functions=myExtensionFactory
```

- b. 次に、Spring Boot の @Bean アノテーションを使用してカスタム Bean を作成します。

```
@Bean(name = "myExtensionFactory")
public ExtensionFunctionDefinition myExtensionFactory() {
}
```

または、Jackson ObjectMapper の場合は、**camel-jackson** データ形式を以下のようにします。

```
camel.dataformat.json-jackson.object-mapper=myJacksonMapper
```

7.5. SPRING BOOT の CAMEL コンテキストの自動設定

Camel Spring Boot auto-configuration は、**CamelContext** インスタンスを提供し、**SpringCamelContext** を作成します。また、コンテキストの初期化およびシャットダウンを実行します。この Camel コンテキストは、**camelContext** Bean 名で Spring アプリケーションコンテキストに登録され、他の Spring Bean と同様にアクセスできます。**camelContext** には次のようにアクセスできます。

例

```

@Configuration
public class MyAppConfig {

    @Autowired
    CamelContext camelContext;

    @Bean
    MyService myService() {
        return new DefaultMyService(camelContext);
    }

}

```

7.6. SPRING BOOT アプリケーションでの CAMEL ルートの自動検出

Camel auto-configuration は、Spring コンテキストからすべての **RouteBuilder** インスタンスを収集し、自動的に **CamelContext** にインジェクトします。これにより、Spring Boot スターターで新しい Camel ルートを作成する処理が簡単になります。以下のようにルートを作成できます。

例

@Component アノテーションが付けられたクラスをクラスパスに追加します。

```

@Component
public class MyRouter extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("jms:invoices").to("file:/invoices");
    }

}

```

または、新しいルート **RouteBuilder** Bean を **@Configuration** クラスで作成します。

```

@Configuration
public class MyRouterConfiguration {

    @Bean
    RoutesBuilder myRouter() {
        return new RouteBuilder() {

            @Override
            public void configure() throws Exception {
                from("jms:invoices").to("file:/invoices");
            }

        };
    }

}

```

7.7. CAMEL SPRING BOOT AUTO CONFIGURATION の CAMEL プロパティの設定

Spring Boot auto-configuration は、プロパティのプレースホルダー、OS 環境変数、Camel プロパティがサポートされるシステムプロパティなどの Spring Boot 外部設定に接続します。

手順

1. **application.properties** ファイルにプロパティを定義します。

```
route.from = jms:invoices
```

または、以下の例のように Camel プロパティをシステムプロパティとして設定します。

```
java -Droute.to=jms:processed.invoices -jar mySpringApp.jar
```

2. 次のように、設定されたプロパティを Camel ルートのプレースホルダーとして使用します。

```
@Component
public class MyRouter extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("#{route.from}").to("#{route.to}");
    }
}
```

7.8. カスタム CAMEL コンテキストの設定

Camel Spring Boot auto-configuration によって作成された **CamelContext** Bean でオペレーションを実行するには、Spring コンテキストで **CamelContextConfiguration** インスタンスを登録します。

手順

- 以下のように、Spring コンテキストで **CamelContextConfiguration** のインスタンスを登録します。

```
@Configuration
public class MyAppConfig {

    ...

    @Bean
    CamelContextConfiguration contextConfiguration() {
        return new CamelContextConfiguration() {
            @Override
            void beforeApplicationStart(CamelContext context) {
                // your custom configuration goes here
            }
        };
    }
}
```

```

    }
}

```

Spring コンテキストの開始前に **CamelContextConfiguration** および **beforeApplicationStart(CamelContext)** メソッドが呼び出され、このコールバックに渡された **CamelContext** インスタンスは完全に自動設定されます。複数のインスタンスの **CamelContextConfiguration** を Spring コンテキストに追加でき、すべてが実行されます。

7.9. 自動設定された CAMELCONTEXT での JMX の無効化

自動設定された **CamelContext** で JMX を無効にするには、**camel.springboot.jmxEnabled** プロパティを使用できます。JMX はデフォルトで有効になっています。

手順

- 以下のプロパティを **application.properties** ファイルに追加し、**false** に設定します。

```

camel.springboot.jmxEnabled = false

```

7.10. 自動設定されたコンシューマーおよびプロデューサーテンプレートの SPRING 管理 BEAN へのインジェクト

Camel auto configuration によって、事前設定された **ConsumerTemplate** および **ProducerTemplate** インスタンスが提供されます。これらを Spring 管理の Bean にインジェクトすることができます。

例

```

@Component
public class InvoiceProcessor {

    @Autowired
    private ProducerTemplate producerTemplate;

    @Autowired
    private ConsumerTemplate consumerTemplate;
    public void processNextInvoice() {
        Invoice invoice = consumerTemplate.receiveBody("jms:invoices", Invoice.class);
        ...
        producerTemplate.sendBody("netty-http:http://invoicing.com/received/" + invoice.id());
    }
}

```

デフォルトでは、コンシューマーテンプレートとプロデューサーテンプレートのエンドポイントキャッシュサイズは 1000 に設定されています。これらの値を変更するには、以下の Spring プロパティを希望するキャッシュサイズに設定します。例を以下に示します。

```

camel.springboot.consumerTemplateCacheSize = 100
camel.springboot.producerTemplateCacheSize = 200

```

7.11. SPRING コンテキストの自動設定された TYPECONVERTER

Camel auto configuration は、Spring コンテキストの **typeConverter** という名前の **TypeConverter** インスタンスを登録します。

例

```
@Component
public class InvoiceProcessor {

    @Autowired
    private TypeConverter typeConverter;

    public long parseInvoiceValue(Invoice invoice) {
        String invoiceValue = invoice.grossValue();
        return typeConverter.convertTo(Long.class, invoiceValue);
    }
}
```

7.12. SPRING タイプコンバージョン API ブリッジ

Spring は、強力な [タイプコンバージョン API](#) で設定されます。Spring API は Camel の [タイプコンバーター API](#) と似ています。これらの API は似ているため、Camel Spring Boot は Spring コンバージョン API に委譲するブリッジコンバーター (**SpringTypeConverter**) を自動的に登録します。つまり、追加設定のない Camel は Spring コンバーターを Camel と同様に扱います。

これにより、以下のように Camel **TypeConverter** API を使用して、Camel および Spring コンバーターの両方にアクセスできます。

例

```
@Component
public class InvoiceProcessor {

    @Autowired
    private TypeConverter typeConverter;

    public UUID parseInvoiceId(Invoice invoice) {
        // Using Spring's StringToUUIDConverter
        UUID id = invoice.typeConverter.convertTo(UUID.class, invoice.getId());
    }
}
```

ここでは、Spring Boot はアプリケーションコンテキストで使用できる Spring の **ConversionService** インスタンスに変換を委譲します。**ConversionService** インスタンスが利用できない場合、Camel Spring Boot auto-configuration は **ConversionService** のインスタンスを作成します。

7.13. タイプ変換機能の無効化

Camel Spring Boot のタイプ変換機能を無効にするには、**camel.springboot.typeConversion** プロパティを **false** に設定します。このプロパティが **false** に設定されると、auto-configuration によってタイプコンバーターインスタンスが登録されず、Spring Boot タイプコンバージョン API へのタイプ変換の委譲が有効になりません。

手順

- Camel Spring Boot コンポーネントのタイプ変換機能を無効にするには、以下のように **camel.springboot.typeConversion** プロパティを **false** に設定します。

```
camel.springboot.typeConversion = false
```

7.14. 自動設定の XML ルートのクラスパスへの追加

デフォルトでは、**camel** ディレクトリーのクラスパスにある Camel XML ルートは Camel Spring Boot コンポーネントによって自動検出され、含まれます。設定オプションを使用すると、ディレクトリー名を設定でき、設定オプションを使用してこの機能を無効化できます。

手順

- 以下のようにクラスパスの Camel Spring Boot XML ルートを設定します。

```
// turn off
camel.springboot.xmlRoutes = false
// scan in the com/foo/routes classpath
camel.springboot.xmlRoutes = classpath:com/foo/routes/*.xml
```



注記

XML ファイルによって Camel XML ルート要素が定義され、**CamelContext** 要素は定義されないはずです。例を以下に示します。

```
<routes xmlns="http://camel.apache.org/schema/spring">
  <route id="test">
    <from uri="timer://trigger"/>
    <transform>
      <simple>ref:myBean</simple>
    </transform>
    <to uri="log:out"/>
  </route>
</routes>
```

Spring XML ファイルの使用

<camelContext> で Spring XML ファイルを使用するには、Spring XML ファイルまたは **application.properties** ファイルの Camel コンテキストを設定します。Camel コンテキストの名前を設定し、ストリームキャッシングを有効にするには、以下を **application.properties** ファイルに追加します。

```
camel.springboot.name = MyCamel
camel.springboot.stream-caching-enabled=true
```

7.15. 自動設定の XML REXT-DSL ルートの追加

Camel Spring Boot コンポーネントによって、**camel-rest** ディレクトリー以下のクラスパスに追加される Camel Rest-DSL XML ルートが自動検出され、組み込まれます。設定オプションを使用すると、ディレクトリー名を設定でき、設定オプションを使用してこの機能を無効化できます。

手順

- 以下のように、クラスパスの Camel Spring Boot Rest-DSL XML ルートを設定します。

```
// turn off
camel.springboot.xmlRests = false
// scan in the com/foo/routes classpath
camel.springboot.xmlRests = classpath:com/foo/rests/*.xml
```



注記

Rest-DSL ファイルによって、Camel XML REST 要素が定義され、**CamelContext** 要素は定義されないはずです。例を以下に示します。

```
<rests xmlns="http://camel.apache.org/schema/spring">
  <rest>
    <post uri="/persons">
      <to uri="direct:postPersons"/>
    </post>
    <get uri="/persons">
      <to uri="direct:getPersons"/>
    </get>
    <get uri="/persons/{personId}">
      <to uri="direct:getPersionId"/>
    </get>
    <put uri="/persons/{personId}">
      <to uri="direct:putPersionId"/>
    </put>
    <delete uri="/persons/{personId}">
      <to uri="direct:deletePersionId"/>
    </delete>
  </rest>
</rests>
```

7.16. CAMEL SPRING BOOT でのテスト

Camel を Spring Boot で実行すると、Spring Boot は自動的に Camel と **@Component** アノテーションが付けられたそのルートを組み込みます。Spring Boot でテストする場合、**@ContextConfiguration** ではなく **@SpringBootTest** を使用して、使用する設定クラスを指定します。

異なる RouteBuilder クラスに複数の Camel ルートがある場合、アプリケーションの実行時に Camel Spring Boot コンポーネントによってこれらのルートがすべて自動的に組み込まれます。1つの RouteBuilder クラスのみからルートをテストする場合は、以下のパターンを使用して、有効にする RouteBuilder を include (含める) または exclude (除外) することができます。

- java-routes-include-pattern: パターンに一致する RouteBuilder クラスを include (含める) ために使用されます。
- java-routes-exclude-pattern: パターンに一致する RouteBuilder クラスを exclude (除外) するために使用されます。exclude は include よりも優先されます。

手順

1. 以下のように、ユニットテストクラスの **include** または **exclude** パターンを **@SpringBootTest** アノテーションへのプロパティとして指定します。

```
@RunWith(CamelSpringBootRunner.class)
@SpringBootTest(classes = {MyApplication.class};
    properties = {"camel.springboot.java-routes-include-pattern=**/Foo*"})
public class FooTest {
```

FooTest クラスの include パターンは Ant スタイルパターンを表す ****/Foo*** です。このパターンは、すべてのパッケージ名と一致する 2 つのアスタリスクで始まります。**/Foo*** は、FooRoute のようにクラス名が Foo で始まる必要があることを意味します。

2. 以下の maven コマンドを使用してテストを実行します。

```
mvn test -Dtest=FooTest
```

その他のリソース

- [Configuring Camel \(Camel の設定\)](#)
- [Component \(コンポーネント\)](#)
- [Endpoint \(エンドポイント\)](#)
- [スタートガイド](#)

第8章 FUSE ON OPENSIFT 上での SPRING BOOT 2 用 SOAP TO REST ブリッジクイックスタートの実行

このクイックスタートは、Camel の REST DSL を使用してバックエンド SOAP API を公開する方法を示しています。簡単な camel ルートは REST 呼び出しをレガシー SOAP サービスにブリッジできます。RH SSO がサポートする REST エンドポイントと SOAP エンドポイントの両方に対してセキュリティが関与します。OAuth および OpenID Connect によって保護されるフロントエンド REST API ならびにクライアントは、リソースオーナーパスワードクレデンシャル OAuth2 モードを使用して RH SSO から JWT アクセストークンを取得し、このトークンを使用して REST エンドポイントにアクセスします。

前提条件

- OCP 3.11 以降のバージョンをインストールし、設定している。
- RH SSO 7.4 以降のバージョンをインストールしている。
- 3Scale 2.8 以降のバージョンをインストールしている。
- [registry.redhat.io](#) への認証が設定されている。詳細は、[Configuring Red Hat Container Registry authentication](#) を参照してください。

手順

以下のセクションでは、Fuse on OpenShift で SOAP to REST ブリッジクイックスタートを実行し、デプロイする方法を説明します。

1. OpenShift サーバーを起動します。このクイックスタートの前提条件として RH SSO イメージ (2 Pod) および 3Scale イメージ (15 Pod) をインストールする必要があるため、**--memory 8GB --cpus 4** オプションを使用して、強力なマシンで OpenShift サーバーを起動する必要があります。有効期限のあるセキュリティトークンを発行する必要もあるため、タイムゾーンオプションも追加する必要があります。Openshift クラスターがローカルマシンと同じタイムゾーンを使用するようにします (デフォルトでは UTC タイムゾーンを使用します)。
2. **cluster-admin** ロールをユーザー **developer** に追加します。

```
$ oc login -u system:admin
$ oc adm policy add-cluster-role-to-user cluster-admin developer
$ oc login -u developer
$ oc project openshift
```

このクイックスタートは RH SSO イメージと同様に **openshift** namespace にデプロイされ (これは関係するテンプレートのデフォルト設定の要件です)、**cluster-admin** ロールをユーザー **developer** に追加する必要があります。

3. シークレットを作成し、これを **serviceaccounts** にリンクします。

```
$ oc create secret docker-registry camel-bridge --docker-server=registry.redhat.io \
--docker-username=USERNAME \
--docker-password=PASSWORD \
--docker-email=EMAIL_ADDRESS
$ oc secrets link default camel-bridge --for=pull
$ oc secrets link builder camel-bridge
```

4. RH SSO イメージストリームを追加し、テンプレート **sso74-x509-postgresql-persistent** で RH SSO をインストールします。

```
$ for resource in sso74-image-stream.json \
  sso74-https.json \
  sso74-postgresql.json \
  sso74-postgresql-persistent.json \
  sso74-x509-https.json \
  sso74-x509-postgresql-persistent.json
do
  oc create -f \
    https://raw.githubusercontent.com/jboss-container-images/redhat-sso-7-openshift-
image/sso74-dev/templates/${resource}
done

$ oc policy add-role-to-user view system:serviceaccount:$(oc project -q):default

$ oc new-app --template=sso74-x509-postgresql-persistent
```

RH SSO イメージが **openshift** namespace からアクセスできることを確認し、テンプレート **sso74-x509-postgresql-persistent** で RH SSO をインストールします。このテンプレートは、RH SSO 設定を永続的に保存できるため、Openshift サーバーの再起動後に設定が保持されます。

5. RH SSO イメージがサーバーに正常にインストールされたら、コンソールに以下のような出力が表示されます。

A new persistent RH-SSO service (using PostgreSQL) has been created in your project. The admin username/password for accessing the master realm via the RH-SSO console is tprYtXP1/nEjf7fojv11FmhJ5eaqadoh0SI2gvlls. The username/password for accessing the PostgreSQL database "root" is userqxe/XNYRjL74CrJEWW7HiSYEdH5FMKVSDytx. The HTTPS keystore used for serving secure content, the JGroups keystore used for securing JGroups communications, and server truststore used for securing RH-SSO requests were automatically created via OpenShift's service serving x509 certificate secrets.

- * With parameters:
 - * Application Name=sso
 - * Custom RH-SSO Server Hostname=
 - * JGroups Cluster Password=1whGRnsAWu162u0e4P6jNpLn5ysJLWjg # generated
 - * Database JNDI Name=java:jboss/datasources/KeycloakDS
 - * Database Name=root
 - * Datasource Minimum Pool Size=
 - * Datasource Maximum Pool Size=
 - * Datasource Transaction Isolation=
 - * PostgreSQL Maximum number of connections=
 - * PostgreSQL Shared Buffers=
 - * Database Username=userqxe # generated
 - * Database Password=XNYRjL74CrJEWW7HiSYEdH5FMKVSDytx # generated
 - * Database Volume Capacity=1Gi
 - * ImageStream Namespace=openshift
 - * RH-SSO Administrator Username=tprYtXP1 # generated
 - * RH-SSO Administrator Password=nEjf7fojv11FmhJ5eaqadoh0SI2gvlls # generated
 - * RH-SSO Realm=
 - * RH-SSO Service Username=

- * RH-SSO Service Password=
- * PostgreSQL Image Stream Tag=10
- * Container Memory Limit=1Gi

6. RH SSO 管理コンソールへのアクセスに使用される Username/Password をメモします。以下に例を示します。

- * RH-SSO Administrator Username=tpYtXP1 # generated
- * RH-SSO Administrator Password=nEjf7fojv11FmhJ5eaqadoh0Sl2gvlls # generated

7. 3scale プロジェクトに 3scale テンプレートをインストールします。

```
$ oc new-project 3scale
$ oc create secret docker-registry threescale-registry-auth --docker-server=registry.redhat.io
--docker-server=registry.redhat.io \
--docker-username=USERNAME \
--docker-password=PASSWORD \
--docker-email=EMAIL_ADDRESS
$ oc secrets link default threescale-registry-auth --for=pull
$ oc secrets link builder threescale-registry-auth
$ oc new-app --param WILDCARD_DOMAIN="OPENSIFT_IP_ADDR.nip.io" -f
https://raw.githubusercontent.com/3scale/3scale-amp-openshift-
templates/2.8.0.GA/amp/amp-eval-tech-preview.yml
```

Openshift への 3scale のインストールは 15 の Pod を起動するため、3scale 用に専用の新規プロジェクトを作成する必要があります。また、3scale の新たな **threescale-registry-auth** (3scale テンプレートで記述されているようにこの名前を使ってシークレットを作成します) シークレットも作成する必要があります。camel-bridge シークレットからの USERNAME/PASSWORD を再利用できます。ここでは意図的に **amp-eval-tech-preview.yml** テンプレートを使用します。これは、ハードウェアリソースを明示的に指定していないため、ローカルマシン/ラップトップで簡単に実行できるためです。

8. 3scale テンプレートが Openshift に正常にインストールされたら、コンソールに以下のような出力が表示されます。

3scale API Management

3scale API Management main system (Evaluation)

Login on <https://3scale-admin.192.168.64.33.nip.io> as admin/b6t784nt

- * With parameters:
 - * AMP_RELEASE=2.8
 - * APP_LABEL=3scale-api-management
 - * TENANT_NAME=3scale
 - * RWX_STORAGE_CLASS=null
 - * AMP_BACKEND_IMAGE=registry.redhat.io/3scale-amp2/backend-rhel7:3scale2.8
 - * AMP_ZYNC_IMAGE=registry.redhat.io/3scale-amp2/zync-rhel7:3scale2.8
 - * AMP_APICAST_IMAGE=registry.redhat.io/3scale-amp2/apicast-gateway-rhel8:3scale2.8
 - * AMP_SYSTEM_IMAGE=registry.redhat.io/3scale-amp2/system-rhel7:3scale2.8
 - * ZYNC_DATABASE_IMAGE=registry.redhat.io/rhsc1/postgresql-10-rhel7
 - * MEMCACHED_IMAGE=registry.redhat.io/3scale-amp2/memcached-rhel7:3scale2.8
 - * IMAGESTREAM_TAG_IMPORT_INSECURE=false
 - * SYSTEM_DATABASE_IMAGE=registry.redhat.io/rhsc1/mysql-57-rhel7:5.7

```
* REDIS_IMAGE=registry.redhat.io/rhscsl/redis-32-rhel7:3.2
* System MySQL User=mysql
* System MySQL Password=mrscfh4h # generated
* System MySQL Database Name=system
* System MySQL Root password.=xbi0ch3i # generated
* WILDCARD_DOMAIN=192.168.64.33.nip.io
* SYSTEM_BACKEND_USERNAME=3scale_api_user
* SYSTEM_BACKEND_PASSWORD=kraji167 # generated
* SYSTEM_BACKEND_SHARED_SECRET=8af5m6gb # generated
*
```

```
SYSTEM_APP_SECRET_KEY_BASE=726e63427173e58cbb68a63bdc60c7315565d6acd037c
aedeeb0050ecc0e6e41c3c7ec4aba01c17d8d8b7b7e3a28d6166d351a6238608bb84aa5d5b2d
c02ae60 # generated
```

```
* ADMIN_PASSWORD=b6t784nt # generated
* ADMIN_USERNAME=admin
* ADMIN_EMAIL=
* ADMIN_ACCESS_TOKEN=k055jof4itblvwwn # generated
* MASTER_NAME=master
* MASTER_USER=master
* MASTER_PASSWORD=buikudum # generated
* MASTER_ACCESS_TOKEN=xa7wkt16 # generated
* RECAPTCHA_PUBLIC_KEY=
* RECAPTCHA_PRIVATE_KEY=
* SYSTEM_REDIS_URL=redis://system-redis:6379/1
* SYSTEM_MESSAGE_BUS_REDIS_URL=
* SYSTEM_REDIS_NAMESPACE=
* SYSTEM_MESSAGE_BUS_REDIS_NAMESPACE=
* Zync Database PostgreSQL Connection Password=efyJdRccBbYcWtWI # generated
* ZYNC_SECRET_KEY_BASE=dcmNGWtrjCReuJIQ # generated
* ZYNC_AUTHENTICATION_TOKEN=3FKMAije3V3RWQQ8 # generated
* APICAST_ACCESS_TOKEN=2ql8txu4 # generated
* APICAST_MANAGEMENT_API=status
* APICAST_OPENSSL_VERIFY=false
* APICAST_RESPONSE_CODES=true
* APICAST_REGISTRY_URL=http://apicast-staging:8090/policies
```

9. 3scale 管理コンソールにアクセスできる Username/Password をメモします。

```
* ADMIN_PASSWORD=b6t784nt # generated
* ADMIN_USERNAME=admin
```

10. RH SSO を設定します。

- a. RH SSO をインストールした後にコンソールに表示されるユーザー名/パスワードを使用して https://sso-openshift.OPENSIFT_IP_ADDR.nip.io/auth から RH SSO 管理コンソールにログインします。
- b. ページの左上隅にある **Add Realm** ボタンをクリックします。
- c. **Add Realm** ページで **Import Select file** ボタンを選択します。
- d. ディレクトリーから `./src/main/resources/keycloak-config/realm-export-new.json` を選択します。これは、事前定義されたこのサンプルに必要な `realm/client/user/role` をインポートします。

11. 3Scale API ゲートウェイを設定します。

- a. 3Scale をインストールした後にコンソールに表示されるユーザー名/パスワードを使用して https://3scale-admin.OPENSIFT_IP_ADDR.nip.io/p/admin/dashboard から 3Scale 管理コンソールにログインします。
- b. 新規製品の作成時に、**Define manually** を選択し、**Name** と **System name** の両方に **camel-security-bridge** を使用します。
- c. 新規バックエンドを作成する場合は、**Name** と **System name** の両方に **camel-security-bridge** を使用し、プライベートベース URL は http://spring-boot-camel-soap-rest-bridge-openshift.OPENSIFT_IP_ADDR.nip.io/ にする必要があります。
- d. 新規作成したバックエンドを新たに作成したプロダクトに追加します。
- e. マッピングルール **Verb:POST Pattern:/** を追加します。
- f. アプリケーションプランを作成する場合には、**Name** と **System name** の両方に **camel-security-bridge** を使用します。
- g. アプリケーションの作成時に、新しい作成した **camel-security-bridge** アプリケーションプランを選択します。アプリケーションを作成したら、API クレデンシャルをメモします。これらのクレデンシャルを使用して 3scale ゲートウェイにアクセスします。以下に例を示します。

```
User Key bdfb53fe9b426fbf21428fd116035798
```

- h. 新たに作成した **camel-security-bridge** プロジェクトを編集し、Dashboard の **camel-security-bridge** から公開します。
 - i. Integration > Settings の順に移動します。**Credentials location** として **HTTP Headers** を選択します。
 - j. Dashboard の **camel-security-bridge** から Integration > Configuration の順に移動し、**Staging APIcast** と **Production APIcast**の両方を昇格します。
12. 展開したクイックスタートアプリケーションが含まれるディレクトリーに移動します (例:my_openshift/spring-boot-camel-soap-rest-bridge)。

```
$ cd my_openshift/spring-boot-camel-soap-rest-bridge
```

13. プロジェクトを OpenShift クラスタにビルドし、デプロイします。

```
$ mvn clean oc:deploy -Popenshift -DJAVA_OPTIONS="-Dsso.server=https://sso-openshift.OPENSIFT_IP_ADDR.nip.io -Dweather.service.host=${your local ip}"
```

Openshift の **camel-soap-rest-bridge** イメージに 2 つの属性を渡す必要があります。1 つは Openshift の RH SSO サーバーアドレスで、これは https://sso-openshift.OPENSIFT_IP_ADDR.nip.io です。もう 1 つは、バックエンド soap サーバーです。このクイックスタートでは、ローカルマシンでバックエンド soap サーバーを実行するので、マシンのローカル IP アドレスを **-Dweather.service.host** として渡します(ローカルホストまたは 127.0.0.1 以外の IP アドレスでなければなりません)。

14. ブラウザーで OpenShift コンソールの **openshift** プロジェクトに移動します。**spring-boot-camel-soap-rest-bridge** の Pod が起動していることを確認できるまで待機します。
15. プロジェクトの Overview ページで、**spring-boot-camel-soap-rest-bridge** アプリケーションの詳細ページデプロイメント

https://OPENSIFT_IP_ADDR:8443/console/project/openshift/browse/pods/spring-boot-camel-soap-rest-bridge-NUMBER_OF_DEPLOYMENT?tab=details に移動します。

16. **Logs** タブに切り替えて、Camel からのログを表示します。

17. OpenApi API にアクセスします。

この例では、context-path camelcxf/openapi を使用して openapi を使用するサービスの API ドキュメントを提供します。Web ブラウザーから API ドキュメント (http://spring-boot-camel-soap-rest-bridge-openshift.OPENSIFT_IP_ADDR.nip.io/camelcxf/openapi/openapi.jsonn) にアクセスできます。

第9章 XA トランザクションを使用した SPRING BOOT での CAMEL サービスの実行

Spring Boot Camel XA トランザクションクイックスタートは、2つの外部トランザクションリソースである JMS リソース (A-MQ) およびデータベース (PostgreSQL) にて XA トランザクションをサポートする Spring Boot で Camel サービスを実行する方法を実証します。これらの外部リソースは OpenShift によって提供され、このクイックスタートを実行する前に起動する必要があります。

9.1. STATEFULSET リソース

このクイックスタートは OpenShift **StatefulSet** リソースを使用してトランザクションマネージャーの一意性を保証します。トランザクションログの保存には PersistentVolume が必要になります。アプリケーションは、StatefulSet リソースのスケールリングをサポートします。各インスタンスには独自の **in-process** リカバリーマネージャーがあります。特別なコントローラーは、アプリケーションがスケールダウンした場合に、終了されたすべてのインスタンスが作業をすべて適切に完了するようにし、保留中のトランザクションが残されないようにします。リカバリーマネージャーが保留中の作業を終了前にすべてフラッシュできない場合、スケールダウン操作はコントローラーによってロールバックされます。このクイックスタートは Spring Boot Narayana リカバリーコントローラーを使用します。

9.2. SPRING BOOT NARAYANA リカバリーコントローラー

Spring Boot Narayana リカバリーコントローラーは、終了前に保留中のトランザクションをクリーンアップして、StatefulSet のスケールダウンフェーズを正常に処理できるようにします。スケールダウン操作が実行されても、終了後に Pod がクリーンアップされない場合は、以前のレプリカが復元されるため、スケールダウン操作が効果的にキャンセルされます。

StatefulSet のすべての Pod は、StatefulSet に属する各 Pod の終了状態を保存するために使用される共有ボリュームにアクセスする必要があります。StatefulSet の pod-0 は状態を定期的にチェックし、StatefulSet のサイズが合わない場合に適切なサイズにスケールリングします。

リカバリーコントローラーが動作するには、現在の namespace のパーミッションを編集する必要があります (ロールバインディングは OpenShift に公開されたリソースセットに含まれています)。リカバリーコントローラーを無効にするには、**CLUSTER_RECOVERY_ENABLED** 環境変数を使用します。この場合、サービスアカウントに特別なパーミッションは必要ありませんが、スケールダウン操作によって、保留中のトランザクションが通知のないまま終了された Pod に残される可能性があります。

9.3. SPRING BOOT NARAYANA リカバリーコントローラーの設定

以下の例は、リカバリーコントローラーを使用して OpenShift で Narayana が動作するよう設定する方法を示しています。

手順

1. これは **application.properties** ファイルの例です。Kubernetes yaml 記述子の以下のオプションを置き換えます。

```
# Cluster
cluster.nodename=1
cluster.base-dir=./target/tx

# Transaction Data
spring.jta.transaction-manager-id=${cluster.nodename}
spring.jta.log-dir=${cluster.base-dir}/store/${cluster.nodename}
```



```
# Narayana recovery settings
snowdrop.narayana.openshift.recovery.enabled=true
snowdrop.narayana.openshift.recovery.current-pod-name=${cluster.nodename}
# You must enable resource filtering in order to inject the Maven artifactId
snowdrop.narayana.openshift.recovery.statefulset=${project.artifactId}
snowdrop.narayana.openshift.recovery.status-dir=${cluster.base-dir}/status
```

- 共有ボリュームに、終了に関連するトランザクションと情報の両方を格納する必要があります。以下のように StatefulSet yaml 記述子にマウントすることができます。

```
apiVersion: apps/v1beta1
kind: StatefulSet
#...
spec:
#...
  template:
#...
    spec:
      containers:
      - env:
        - name: CLUSTER_BASE_DIR
          value: /var/transaction/data
          # Override CLUSTER_NODENAME with Kubernetes Downward API (to use `pod-0`,
          `pod-1` etc. as tx manager id)
        - name: CLUSTER_NODENAME
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.name
#...
      volumeMounts:
      - mountPath: /var/transaction/data
        name: the-name-of-the-shared-volume
#...
```

Spring Boot Narayana リカバリーコントローラーの Camel エクステンション

Spring Boot アプリケーションコンテキストで Camel が見つかった場合、保留中のトランザクションをすべてフラッシュする前に Camel コンテキストは自動的に停止されます。

9.4. OPENSIFT での CAMEL SPRING BOOT XA クイックスタートの実行

この手順では、実行中の単一ノードの OpenShift クラスタでクイックスタートを実行する方法を説明します。

手順

1. Camel Spring Boot XA プロジェクトをダウンロードします。

```
git clone --branch spring-boot-camel-xa-7.9.0.fuse-sb2-790055-redhat-00002
https://github.com/jboss-fuse/spring-boot-camel-xa
```

2. **spring-boot-camel-xa** ディレクトリーに移動し、以下のコマンドを実行します。

■

```
mvn clean install
```

- OpenShift サーバーにログインします。

```
oc login -u developer -p developer
```

- test** という名前のプロジェクト namespace が存在しない場合は作成します。

```
oc new-project test
```

test プロジェクト namespace がすでに存在する場合は、以下のコマンドを使用して切り替えます。

```
oc project test
```

- 依存関係をインストールします。

- ユーザー名 **theuser** とパスワード **Thepassword1!** を使用して、**postgresql** をインストールします。

```
oc new-app --param=POSTGRESQL_USER=theuser --
param=POSTGRESQL_PASSWORD='Thepassword1!' --
env=POSTGRESQL_MAX_PREPARED_TRANSACTIONS=100 --template=postgresql-
persistent
```

- ユーザー名 **theuser** とパスワード **Thepassword1!** を使用して、**A-MQ** ブローカーをインストールします。

```
oc new-app --param=MQ_USERNAME=theuser --
param=MQ_PASSWORD='Thepassword1!' --template=amq63-persistent
```

- トランザクションログの永続ボリュームクレームを作成します。

```
oc create -f persistent-volume-claim.yml
```

- クイックスタートをビルドおよびデプロイします。

```
mvn oc:deploy -Popenshift
```

- レプリカの数が必要に合わせてスケールアップします。

```
oc scale statefulset spring-boot-camel-xa --replicas 3
```

注記: Pod 名はトランザクションマネージャー ID として使用されます (spring.jta.transaction-manager-id プロパティ)。また、現在の実装によってトランザクションマネージャー ID の長さも制限されます。よって、以下の点に注意してください。

- StatefulSet の名前はトランザクションシステムの識別子であるため、変更しないでください。
- Pod の名前が 23 文字以下になるように StatefulSet の名前を付ける必要があります。Pod 名は、<statefulset-name>-0, <statefulset-name>-1 のような慣例を使用して、OpenShift によって作成されます。Narayana はできる限り、同じ ID を持つリカバリーマネージャーが

複数存在しないようにするため、Pod 名が制限よりも長い場合は、最後の 23 バイトがトランザクション ID として適用されます (-などの一部の文字の削除後)。

9. クイックスタートが実行されたら、以下のコマンドを使用してベースサービス URL を取得します。

```
NARAYANA_HOST=$(oc get route spring-boot-camel-xa -o jsonpath={.spec.host})
```

9.5. 成功した XA トランザクションのテスト

以下のワークフローは、成功した XA トランザクションのテスト方法を示しています。

手順

1. audit_log テーブルのメッセージのリストを取得します。

```
curl -w "\n" http://$NARAYANA_HOST/api/
```

2. 当初、このリストは空の状態です。最初の要素を追加します。

```
curl -w "\n" -X POST http://$NARAYANA_HOST/api/?entry=hello
```

しばらく待ってから、新しいリストを取得します。

```
curl -w "\n" http://$NARAYANA_HOST/api/
```

3. 新しいリストには、**hello** と **hello-ok** の 2 つのメッセージが含まれます。**hello-ok** は、メッセージが送信キューに送られ、ログに記録されたことを確認します。複数のメッセージを追加してログを確認することができます。

9.6. 失敗した XA トランザクションのテスト

以下のワークフローは、失敗した XA トランザクションのテスト方法を示しています。

手順

1. **fail** という名前のメッセージを送信します。

```
curl -w "\n" -X POST http://$NARAYANA_HOST/api/?entry=fail
```

2. しばらく待ってから、新しいリストを取得します。

```
curl -w "\n" http://$NARAYANA_HOST/api/
```

3. このメッセージは、ルートの最後で例外を発生するため、トランザクションは常にロールバックされます。audit_log テーブルにはメッセージのトレースが記録されないはずです。

第10章 CAMEL アプリケーションの A-MQ ブローカーとの統合

このチュートリアルでは、A-MQ イメージを使用してクイックスタートをデプロイする方法を説明します。

10.1. SPRING BOOT CAMEL A-MQ クイックスタートのビルドおよびデプロイ

このクイックスタートでは、Spring Boot アプリケーションを AMQ Broker に接続する方法と、Fuse on OpenShift を使用して 2 つの Camel ルートの間で JMS メッセージングを使用する方法を実証します。

前提条件

- [Deploying AMQ Broker on OpenShift](#) の説明どおりに、AMQ Broker がインストールされ、稼働していることを確認します。
- OpenShift が適切に稼働し、Fuse イメージストリームがすでに OpenShift にインストールされているようにしてください。[管理者向けの基本情報](#) を参照してください。
- Maven リポジトリが Fuse に対して設定されているようにしてください。詳細は [Maven リポジトリの設定](#) を参照してください。

手順

1. 開発者として OpenShift サーバーにログインします。

```
oc login -u developer -p developer
```

2. 以下のように、クイックスタートの新規プロジェクトを作成します。

```
oc new-project quickstart
```

3. Maven archetype を使用してクイックスタートプロジェクトを取得します。

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate -
DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabric8/archetypes/archetypes-
catalog/2.2.0.fuse-sb2-790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-
redhat-00004-archetype-catalog.xml -DarchetypeGroupId=org.jboss.fuse.fis.archetypes -
DarchetypeArtifactId=spring-boot-camel-amq-archetype -DarchetypeVersion=2.2.0.fuse-sb2-
790047-redhat-00004
```

4. クイックスタートディレクトリー **fuse79-spring-boot-camel-amq** に移動します。

```
cd fuse79-spring-boot-camel-amq
```

5. 以下のコマンドを実行し、設定ファイルを AMQ Broker に適用します。これらの設定ファイルによって、管理者権限を持つ AMQ Broker ユーザーおよびキューが作成されます。

```
oc login -u admin -p admin
```

```
oc apply -f src/main/resources/k8s
```

6. 以下のように、アプリケーションの ConfigMap を作成します。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: spring-boot-camel-amq-config
  namespace: quickstarts
data:
  service.host: 'fuse-broker-amqps-0-svc'
  service.port.amqp: '5672'
  service.port.amqps: '5671'
```

7. ステップ 3 の ImageStream を使用して、**mvn** コマンドを実行してクイックスタートを OpenShift サーバーにデプロイします。

```
mvn oc:deploy -Popenshift -Djkube.generator.fromMode=istag -
Djkube.generator.from=openshift/fuse-java-openshift:1.9
```

8. クイックスタートが正常に実行されていることを確認するには、以下を行います。
- ブラウザーで https://OPENSIFT_IP_ADDR の OpenShift Web コンソールに移動します。OPENSIFT_IP_ADDR はクラスターの IP アドレスに置き換えます。クレデンシャル (例: ユーザー名 developer、パスワード developer) を使用して、コンソールにログインします。
 - 左側のパネルで **Home** を展開し、**Status** をクリックして **openshift** プロジェクトの Project Status ページを表示します。
 - fuse79-spring-boot-camel-amq** をクリックし、クイックスタートの概要情報ページを表示します。
 - 左側のパネルで **Workloads** を展開します。
 - Pods** をクリックした後、**fuse79-spring-boot-camel-amq-xxxxx** をクリックします。クイックスタートの Pod の詳細が表示されます。
 - Logs** をクリックし、アプリケーションのログを確認します。出力にはメッセージが正常に送信されたことが表示されます。

```
10:17:59.825 [Camel (camel) thread #10 - timer://order] INFO generate-order-route -
Generating order order1379.xml
10:17:59.829 [Camel (camel) thread #8 - JmsConsumer[incomingOrders]] INFO jms-
cbr-route - Sending order order1379.xml to the UK
10:17:59.829 [Camel (camel) thread #8 - JmsConsumer[incomingOrders]] INFO jms-
cbr-route - Done processing order1379.xml
10:18:02.825 [Camel (camel) thread #10 - timer://order] INFO generate-order-route -
Generating order order1380.xml
10:18:02.829 [Camel (camel) thread #7 - JmsConsumer[incomingOrders]] INFO jms-
cbr-route - Sending order order1380.xml to another country
10:18:02.829 [Camel (camel) thread #7 - JmsConsumer[incomingOrders]] INFO jms-cbr-
route - Done processing order1380.xml
```

9. Web インターフェイスでルートを表示するには、**Open Java Console** をクリックし、AMQ キューのメッセージをチェックします。

第11章 SPRING BOOT と KUBERNETES の統合

現在、Spring Cloud Kubernetes プラグインは Spring Boot と Kubernetes の以下の機能を統合できます。

- [Spring Boot の外部化設定](#)
- [Kubernetes の ConfigMap](#)
- [Kubernetes の Secret](#)

11.1. SPRING BOOT の外部化設定

Spring Boot の [外部化設定 \(externalized configuration\)](#) は、外部ソースの設定値を Java コードにインジェクトできるメカニズムです。Java コードでインジェクションを有効にするには、通常 **@Value** アノテーション (単一のフィールドにインジェクトする場合) または **@ConfigurationProperties** アノテーション (Java Bean クラスの複数のプロパティにインジェクトする場合) をつけます。

設定データはさまざまなソース (**プロパティソース**) から取得できます。特に、設定プロパティは多くの場合でプロジェクトの **application.properties** ファイル (または **application.yaml** ファイル) で設定されます。

11.1.1. Kubernetes の ConfigMap

Kubernetes の [ConfigMap](#) は、設定データをデプロイされたアプリケーションに提供できるメカニズムです。ConfigMap オブジェクトは、通常 YAML ファイルに定義され、Kubernetes クラスタにアップロードされるため、デプロイされたアプリケーションで設定データを利用できるようになります。

11.1.2. Kubernetes の Secret

Kubernetes の [シークレット](#) は、機密データ (パスワード、証明書など) をデプロイされたアプリケーションに提供するメカニズムです。

11.1.3. Spring Cloud Kubernetes プラグイン

[Spring Cloud Kubernetes](#) プラグインは、Kubernetes と Spring Boot 間のインテグレーションを実装します。原則では、Kubernetes API を使用して ConfigMap から設定データにアクセスできます。しかし、Kubernetes の ConfigMap が Spring Boot 設定の代替プロパティソースとして動作するように Kubernetes の ConfigMap を直接 Spring Boot の外部化設定メカニズムで統合した方がはるかに便利です。Spring Cloud Kubernetes プラグインは基本的にこの機能を提供します。

11.1.4. Kubernetes インテグレーションでの Spring Boot の有効化

Kubernetes インテグレーションを有効にするには、そのインテグレーションを Maven 依存関係として **pom.xml** ファイルに追加します。

手順

1. 以下の Maven 依存関係を Spring Boot Maven プロジェクトの pom.xml ファイルに追加して、Kubernetes インテグレーションを有効にします。

```
<project ...>
...
<dependencies>
```

```

...
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-kubernetes-config</artifactId>
</dependency>
...
</dependencies>
...
</project>

```

2. インテグレーションを完了するには以下を行います。

- アノテーションを Java ソースコードに追加します。
- Kubernetes ConfigMap オブジェクトを作成します。
- アプリケーションが ConfigMap オブジェクトを読み取りできるようにするため、OpenShift サービスアカウントのパーミッションを編集します。

関連情報

- 詳細は、[ConfigMap プロパティソースのチュートリアルの実行](#) を参照してください。

11.2. CONFIGMAP プロパティソースのチュートリアルの実行

以下のチュートリアルでは、Kubernetes の Secret および ConfigMap の設定を試すことができます。[Kubernetes インテグレーションでの Spring Boot の有効化](#) の説明どおりに、Spring Cloud Kubernetes プラグインを有効にして、Kubernetes 設定オブジェクトを Spring Boot 外部化設定と統合します。

11.2.1. Spring Boot Camel Config クイックスタートの実行

以下のチュートリアルは、Kubernetes の Secret と ConfigMap の設定を可能にする **spring-boot-camel-config-archetype** Maven archetype を基にしています。

手順

1. 新しいシェルプロンプトを開き、以下の Maven コマンドを入力して簡単な Camel Spring Boot プロジェクトを作成します。

```

mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
-
DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabric8/archetypes/archetypes-
catalog/2.2.0.fuse-sb2-790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-
redhat-00004-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-config-archetype \
-DarchetypeVersion=2.2.0.fuse-sb2-790047-redhat-00004

```

archetype プラグインは対話モードに切り替わり、残りのフィールドを追加するよう要求します。

```

Define value for property 'groupId': : org.example.fis
Define value for property 'artifactId': : fuse79-configmap

```

```

Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': org.example.fis: :
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse79-configmap
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y

```

プロンプトが表示されたら、**org.example.fis** を **groupId** の値として入力し、**fuse79-configmap** を **artifactId** の値として入力します。残りのフィールドにはデフォルト値を使用します。

2. OpenShift にログインし、アプリケーションをデプロイする OpenShift プロジェクトに切り替えます。たとえば、**developer** ユーザーとしてログインし、**openshift** プロジェクトにデプロイする場合は、以下のコマンドを入力します。

```

oc login -u developer -p developer
oc project openshift

```

3. コマンドラインで、新しい **fuse79-configmap** プロジェクトのディレクトリーに移動し、このアプリケーションの Secret オブジェクトを作成します。

```

cd fuse79-configmap
oc create -f sample-secret.yml

```



注記

アプリケーションをデプロイする **前** に Secret オブジェクトを作成する必要があります。そうでないと、Secret が利用できるようになるまでデプロイされたコンテナが待機状態になります。続けて Secret を作成すると、コンテナは待機状態ではなくなります。Secret オブジェクトの設定方法に関する詳細は **Secret の設定** を参照してください。

4. クイックスタートアプリケーションをビルドおよびデプロイします。**fuse79-configmap** プロジェクトのトップレベルで以下を入力します。

```

mvn oc:deploy -Popenshift

```

5. 次のようにアプリケーションログを確認します。
 - a. ブラウザーで https://OPENSIFT_IP_ADDR の OpenShift Web コンソールに移動します (**OPENSIFT_IP_ADDR** はクラスターの IP アドレスに置き換えます)。クレデンシャル (例: ユーザー名 **developer**、パスワード **developer**) を使用して、コンソールにログインします。
 - b. 左側のパネルで **Home** を展開します。**Status** をクリックして **Project Status** ページを表示します。選択された namespace (例: **openshift**) の既存のアプリケーションがすべて表示されます。
 - c. **fuse79-configmap** をクリックし、クイックスタートの **Overview** 情報ページを表示します。
 - d. 左側のパネルで **Workloads** を展開します。

- e. **Pods** をクリックした後、**fuse79-configmap-xxxx** をクリックします。アプリケーションの Pod の詳細が表示されます。
 - f. **Logs** タブをクリックし、アプリケーションのログを表示します。
6. **src/main/resources/application.properties** で設定されるデフォルトの受信者リストは、生成されたメッセージを2つのダミーエンドポイントである **direct:async-queue** および **direct:file** に送信します。これにより、以下のようなメッセージがアプリケーションログに書き込まれます。

```
5:44:57.377 [Camel (camel) thread #0 - timer://order] INFO generate-order-route -
Generating message message-44, sending to the recipient list
15:44:57.378 [Camel (camel) thread #0 - timer://order] INFO target-route-queue - ---->
message-44 pushed to an async queue (simulation)
15:44:57.379 [Camel (camel) thread #0 - timer://order] INFO target-route-queue - ----> Using
username 'myuser' for the async queue
15:44:57.380 [Camel (camel) thread #0 - timer://order] INFO target-route--file - ---->
message-44 written to a file
```

7. ConfigMap オブジェクトを使用して **fuse79-configmap** アプリケーションの設定を更新する前に、OpenShift ApiServer からデータを確認するパーミッションを **fuse79-configmap** アプリケーションに付与する必要があります。以下のコマンドを入力し、**fuse79-configmap** アプリケーションのサービスアカウントに **view** パーミッションを付与します。

```
oc policy add-role-to-user view system:serviceaccount:openshift:qs-camel-config
```



注記

構文 **system:serviceaccount:PROJECT_NAME:SERVICE_ACCOUNT_NAME** を使用してサービスアカウントが指定されます。**fis-config** デプロイメント記述子は、**SERVICE_ACCOUNT_NAME** を **qs-camel-config** に定義します。

8. ライブリロード機能の動作を確認するには、以下のように ConfigMap オブジェクトを作成します。

```
oc create -f sample-configmap.yml
```

新しい ConfigMap は、実行中のアプリケーションにある Camel ルートの受信者リストをオーバーライドし、生成されたメッセージを3つのダミーエンドポイントである **direct:async-queue**、**direct:file**、および **direct:mail** に送信するように設定します。ConfigMap オブジェクトの詳細は、ConfigMap の設定を参照してください。これにより、以下のようなメッセージがアプリケーションログに書き込まれます。

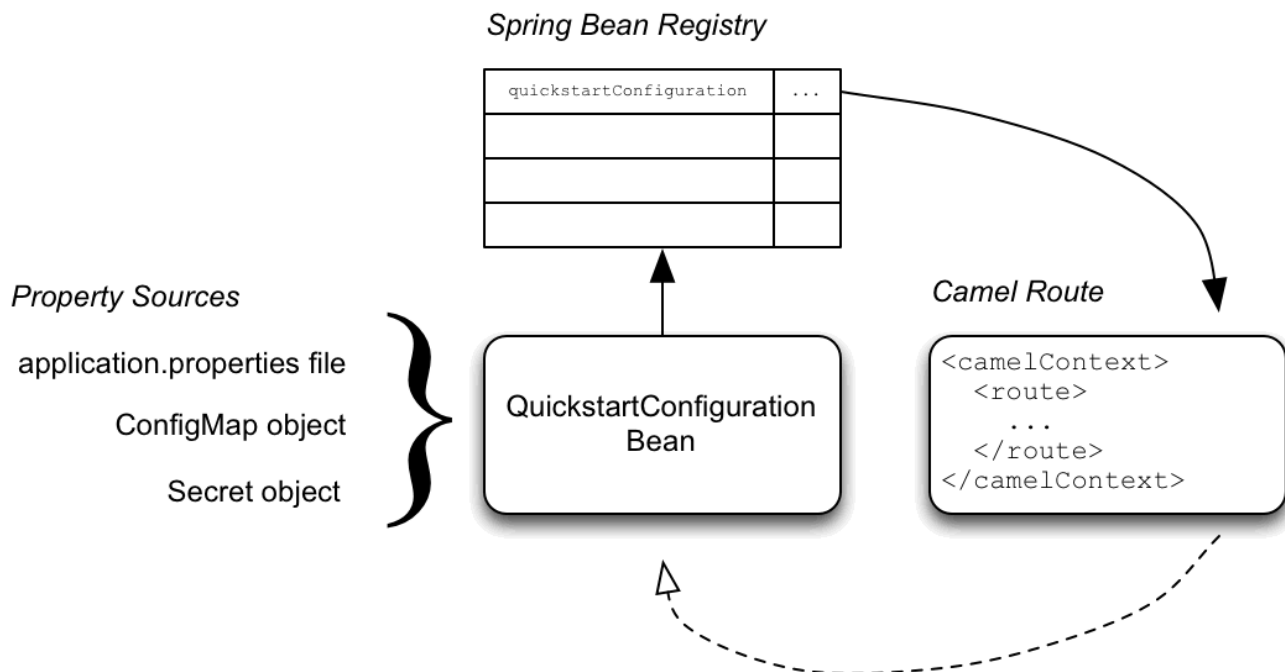
```
16:25:24.121 [Camel (camel) thread #0 - timer://order] INFO generate-order-route -
Generating message message-9, sending to the recipient list
16:25:24.124 [Camel (camel) thread #0 - timer://order] INFO target-route-queue - ---->
message-9 pushed to an async queue (simulation)
16:25:24.125 [Camel (camel) thread #0 - timer://order] INFO target-route-queue - ----> Using
username 'myuser' for the async queue
16:25:24.125 [Camel (camel) thread #0 - timer://order] INFO target-route--file - ---->
message-9 written to a file (simulation)
16:25:24.126 [Camel (camel) thread #0 - timer://order] INFO target-route--mail - ---->
message-9 sent via mail
```

11.2.2. 設定プロパティ Bean

設定プロパティ Bean は、インジェクションによって設定を受け取ることができる標準の Java Bean です。Java コードと外部設定メカニズムの間の基本的なインターフェイスを提供します。

外部化設定および Bean レジストリー

以下のイメージは、**spring-boot-camel-config** クイックスタートで Spring Boot の外部化設定がどのように動作するかを示しています。



設定メカニズムには以下の 2 つの主要部分があります。

プロパティソース

設定へのインジェクションのプロパティ設定を提供します。デフォルトのプロパティソースはアプリケーションの **application.properties** ファイルで、任意で ConfigMap オブジェクトまたは Secret オブジェクトによるオーバーライドが可能です。

設定プロパティ Bean

プロパティソースから設定の更新を受け取ります。設定プロパティ Bean は、**@Configuration** または **@ConfigurationProperties** アノテーションが付けられた Java Bean です。

Spring Bean レジストリー

必要なアノテーションが付けられた 設定プロパティ Bean は、Spring Bean レジストリーに登録されます。

Camel Bean レジストリー

Camel Bean レジストリーは、自動的に Spring Bean レジストリーと統合されるため、登録された Spring Bean を Camel ルートで参照できます。

QuickstartConfiguration クラス

fuse79-configmap プロジェクトの設定プロパティ Bean は、以下のように **QuickstartConfiguration** Java クラス (**src/main/java/org/example/fis/** ディレクトリー下) として定義されます。

```
package org.example.fis;
```

```

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;

@Configuration ❶
@ConfigurationProperties(prefix = "quickstart") ❷
public class QuickstartConfiguration {

    /**
     * A comma-separated list of routes to use as recipients for messages.
     */
    private String recipients; ❸

    /**
     * The username to use when connecting to the async queue (simulation)
     */
    private String queueUsername; ❹

    /**
     * The password to use when connecting to the async queue (simulation)
     */
    private String queuePassword; ❺

    // Setters and Getters for Bean properties
    // NOT SHOWN
    ...
}

```

- ❶ **@Configuration** アノテーションによって、**QuickstartConfiguration** クラスはインスタンス化され、**quickstartConfiguration** を ID として持つ Bean として Spring に登録されます。これにより、Bean は自動的に Camel からアクセスできるようになります。たとえば、**Camel 構文** `${bean:quickstartConfiguration?method=getQueueUsername}` を使用すると、**target-route-queue** ルートは **queueUserName** プロパティにアクセスできるようになります。
- ❷ **@ConfigurationProperties** アノテーションは、プロパティソースのプロパティ値を定義するときに使用する必要がある接頭辞 **quickstart** を定義します。たとえば、プロパティファイルは **recipients** プロパティを **quickstart.recipients** として参照します。
- ❸ **recipient** プロパティはプロパティソースからインジェクトできます。
- ❹ **queueUsername** プロパティはプロパティソースからインジェクトできます。
- ❺ **queuePassword** プロパティはプロパティソースからインジェクトできます。

11.2.3. Secret の設定

このクイックスタートの Kubernetes Secret は、必要な1つの追加ステップ以外は標準の方法でセットアップされています。追加ステップとして、Spring Cloud Kubernetes プラグインを Secret のマウントパスで設定し、起動時に Secret が読み取れるようにする必要があります。Secret を設定するには、以下を行います。

1. サンプル Secret オブジェクトの作成
2. Secret のボリュームマウントの設定
3. Secret プロパティを読み取るため `spring-cloud-kubernetes` を設定

サンプル Secret オブジェクト

クイックスタートプロジェクトによって、以下のようなサンプル Secret **sample-secret.yml** が提供されます。Secret オブジェクトのプロパティ値は常に base64 でエンコードされます (**base64** コマンドラインユーティリティを使用)。Secret が Pod のファイルシステムにマウントされると、値は自動的に元のプレーンテキストにデコードされます。

sample-secret.yml ファイル

```
apiVersion: v1
kind: Secret
metadata: ❶
  name: camel-config
type: Opaque
data:
  # The username is 'myuser'
  quickstart.queue-username: bXl1c2VyCg== ❷
  quickstart.queue-password: MWYyZDFiMmU2N2Rm ❸
```

- ❶ metadata.name: Secret を識別します。OpenShift システムの他の部分はこの識別子を使用して Secret を参照します。
- ❷ quickstart.queue-username: **quickstartConfiguration Bean** の **queueUsername** プロパティにインジェクトすることを目的とします。値は base64 でエンコードする **必要** があります。
- ❸ quickstart.queue-password: **quickstartConfiguration Bean** の **queuePassword** プロパティにインジェクトすることを目的とします。値は base64 でエンコードする **必要** があります。



注記

Kubernetes では、プロパティ名を CamelCase で定義できません (プロパティ名をすべて小文字にする必要があります)。この制限を回避するには、ハイフンを使用した形式 **queue-username** を使用します。Spring はこれを **queueUsername** と一致します。これは、外部化設定に対して Spring Boot の [リラックスバインディングルール](#) を利用します。

Secret のボリュームマウントの設定

Secret をボリュームマウントとして設定し、起動時に Secret がロードされるようにアプリケーションを設定する必要があります。アプリケーションの起動後、Secret プロパティはファイルシステムの指定の場所で利用可能になります。アプリケーションの **deployment.yml** ファイルは、Secret のボリュームマウントを定義する **src/main/jkube/** ディレクトリ下にあります。

deployment.yml ファイル

```
spec:
  template:
    spec:
      serviceAccountName: "qs-camel-config"
      volumes: ❶
      - name: "camel-config"
        secret:
          # The secret must be created before deploying this application
          secretName: "camel-config"
```

```

containers:
-
  volumeMounts: ❷
  - name: "camel-config"
    readOnly: true
    # Mount the secret where spring-cloud-kubernetes is configured to read it
    # see src/main/resources/bootstrap.yml
    mountPath: "/etc/secrets/camel-config"
  resources:
  #   requests:
  #   cpu: "0.2"
  #   memory: 256Mi
  #   limits:
  #   cpu: "1.0"
  #   memory: 256Mi
  env:
  - name: SPRING_APPLICATION_JSON
    value: '{"server":{"undertow":{"io-threads":1, "worker-threads":2 }}}'

```

- ❶ デプロイメントは **volumes** セクションで、**camel-config** という名前の Secret を参照する **camel-config** という名前の新しいボリュームを宣言します。
- ❷ デプロイメントは **volumeMounts** セクションで、新しいボリュームマウントを宣言します。これは、**camel-config** ボリュームを参照し、Secret ボリュームが Pod のファイルシステムの **/etc/secrets/camel-config** パスにマウントする必要があることを指定します。

Secret プロパティーを読み取るため spring-cloud-kubernetes を設定

Secret を Spring Boot の外部化設定と統合するには、Spring Cloud Kubernetes プラグインを Secret のマウントパスで設定する必要があります。Spring Cloud Kubernetes は指定の場所から Secret を読み取り、Spring Boot はそれをプロパティーソースとして利用できるようになります。Spring Cloud Kubernetes プラグインを設定するには、クイックスタートプロジェクトの **src/main/resources** 下にある **bootstrap.yml** ファイルを設定します。

bootstrap.yml ファイル

```

# Startup configuration of Spring-cloud-kubernetes
spring:
  application:
    name: camel-config
  cloud:
    kubernetes:
      reload:
        # Enable live reload on ConfigMap change (disabled for Secrets by default)
        enabled: true
      secrets:
        paths: /etc/secrets/camel-config

```

spring.cloud.kubernetes.secrets.paths プロパティーは、Pod の Secret ボリュームマウントのパスリストを指定します。



注記

bootstrap.properties ファイル (または **bootstrap.yml** ファイル) は、**application.properties** ファイルと同様に動作しますが、アプリケーションの起動時により前の段階でロードされます。**bootstrap.properties** ファイルの Spring Cloud Kubernetes プラグインに関連するプロパティを設定した方が信頼性が高くなります。

11.2.4. ConfigMap の設定

Spring Cloud Kubernetes との統合には、ConfigMap オブジェクトの作成と適切な view パーMISSION の設定の他に、ConfigMap の **metadata.name** がプロジェクトの **bootstrap.yml** ファイルに設定された **spring.application.name** プロパティの値と一致するようにする必要があります。ConfigMap を設定するには、以下を行います。

- サンプル ConfigMap オブジェクトの作成
- view パーMISSION の設定
- Spring Cloud Kubernetes プラグインの設定

サンプル ConfigMap オブジェクト

クイックスタートオブジェクトは、サンプル ConfigMap **sample-configmap.yml** を提供します。

```
kind: ConfigMap
apiVersion: v1
metadata: ❶
  # Must match the 'spring.application.name' property of the application
  name: camel-config
data:
  application.properties: | ❷
    # Override the configuration properties here
    quickstart.recipients=direct:async-queue,direct:file,direct:mail ❸
```

- ❶ `metadata.name: ConfigMap` を識別します。OpenShift システムの他の部分はこの識別子を使用して ConfigMap を参照します。
- ❷ `data.application.properties:` このセクションは、アプリケーションとデプロイされた **application.properties** ファイルの設定をオーバーライドできるプロパティ設定をリストします。
- ❸ `quickstart.recipients:` **quickstartConfiguration** Bean の **recipients** プロパティにインジェクトすることを目的とします。

view パーMISSION の設定

Secret の `deployment.yml` ファイルで説明したとおり、**serviceAccountName** はプロジェクトの **deployment.yml** ファイルで **qs-camel-config** に設定されます。そのため、以下のコマンドを実行して、クイックスタートアプリケーションで **view** パーMISSION を有効にします (**test** プロジェクト namespace にデプロイされることを仮定します)。

```
oc policy add-role-to-user view system:serviceaccount:test:qs-camel-config
```

Spring Cloud Kubernetes プラグインの設定

Spring Cloud Kubernetes プラグインは、**bootstrap.yml** ファイルの以下の設定によって指定されます。

spring.application.name

この値は、ConfigMap オブジェクトの **metadata.name** と一致する必要があります (たとえば、クイックスタートの **sample-configmap.yml** で定義された値)。デフォルトは **application** です。

spring.cloud.kubernetes.reload.enabled

これを **true** に設定すると、ConfigMap オブジェクトの動的リロードが有効になります。

サポートされるプロパティに関する詳細は [PropertySource リロード設定プロパティ](#) を参照してください。

11.3. CONFIGMAP PROPERTYSOURCE の使用

Kubernetes には、設定をアプリケーションに渡すための [ConfigMap](#) という概念があります。Spring Cloud Kubernetes プラグインは、**ConfigMap** とのインテグレーションを提供し、Spring Boot が ConfigMap にアクセスできるようにします。

有効な場合、**ConfigMap PropertySource** はアプリケーションの名前が付いた **ConfigMap** を Kubernetes で検索します (**spring.application.name** を参照)。その ConfigMap が見つかった場合は、そのデータを読み取り、以下を行います。

- [個々のプロパティの適用](#)
- [application.yaml という名前のプロパティの適用](#)
- [application.properties という名前のプロパティの適用](#)

11.3.1. 個々のプロパティの適用

プロパティを使用してそのスレッドプール設定を読み取る **demo** という名前の Spring Boot アプリケーションがあるとします。

- **pool.size.core**
- **pool.size.max**

これを YAML 形式で ConfigMap に外部化することができます。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  pool.size.core: 1
  pool.size.max: 16
```

11.3.2. application.yaml ConfigMap プロパティの適用

ほとんどの場合で個々のプロパティは適切に動作しますが、YAML を使用した方が便利であることがあります。ここでは、**application.yaml** という名前の単一のプロパティを使用し、YAML を内部に組み込みします。


```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.yaml: |-
    pool:
      size:
        core: 1
        max: 16
```

11.3.3. application.properties ConfigMap プロパティの適用

Spring Boot の **application.properties** ファイルのスタイルで ConfigMap プロパティを定義することもできます。ここでは、**application.properties** という名前の単一のプロパティを使用し、内部にプロパティ設定をリストします。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  application.properties: |-
    pool.size.core: 1
    pool.size.max: 16
```

11.3.4. ConfigMap のデプロイ

ConfigMap をデプロイし、Spring Boot アプリケーションにアクセスできるようにするには、以下の手順を実行します。

手順

1. Spring Boot アプリケーションで [外部化設定](#) メカニズムを使用し、ConfigMap プロパティソースへアクセスします。たとえば、Java Bean に **@Configuration** アノテーションを付けると、ConfigMap による Bean のプロパティ値のインジェクションが可能になります。
2. プロジェクトの **@Configuration** ファイル (または **bootstrap.yaml** ファイル) で、**spring.application.name** プロパティが ConfigMap の名前と一致するように設定します。
3. アプリケーションに関連するサービスアカウントの **view** パーミッションを有効にします (デフォルトでは、**default** というサービスアカウントになります)。たとえば、**view** パーミッションを **default** サービスアカウントに追加するには、以下を実行します。

```
oc policy add-role-to-user view system:serviceaccount:$(oc project -q):default -n $(oc project -q)
```

11.4. SECRETS PROPERTYSOURCE の使用

Kubernetes には、パスワードや OAuth トークンなどの機密データを格納するための [シークレット](#) という概念があります。Spring Cloud Kubernetes プラグインは **Secrets** とのインテグレーションを提供し、Spring Boot が Secret へアクセスできるようにします。

有効になっている **Secrets** プロパティソースは、以下のソースから Kubernetes の **Secrets** を検索します。Secret が見つかった場合、アプリケーションはそのデータを利用できます。

1. Secret マウントからの再帰的な読み取り
2. アプリケーションにちなんだ名前の付与 (**spring.application.name** を参照)
3. 一部のラベルとの一致

デフォルトでは、API 経由の Secret の消費 (上記の 2 および 3) は **有効になっていない** ことに注意してください。

11.4.1. Secret の設定例

プロパティを使用して ActiveMQ および PostgreSQL 設定を読み取る **demo** という名前の Spring Boot アプリケーションがあるとします。

```
amq.username
amq.password
pg.username
pg.password
```

これらの Secret は YAML 形式で **Secrets** に対して外部化できます。

ActiveMQ の Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: activemq-secrets
labels:
  broker: activemq
type: Opaque
data:
  amq.username: bXl1c2VyCg==
  amq.password: MWYyZDFIMmU2N2Rm
```

PostgreSQL の Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secrets
labels:
  db: postgres
type: Opaque
data:
  pg.username: dXNlcgo=
  pg.password: cGdhZG1pbgo=
```

11.4.2. Secret の消費

消費する Secret を選択する方法は複数あります。

- Secret がマップされたディレクトリーをリストする方法

```
-Dspring.cloud.kubernetes.secrets.paths=/etc/secrets/activemq,etc/secrets/postgres
```

すべての Secret が共通のルートにマップされている場合は、以下のように設定できます。

```
-Dspring.cloud.kubernetes.secrets.paths=/etc/secrets
```

- 名前付きの Secret を設定する方法

```
-Dspring.cloud.kubernetes.secrets.name=postgres-secrets
```

- ラベルのリストを定義する方法

```
-Dspring.cloud.kubernetes.secrets.labels.broker=activemq
-Dspring.cloud.kubernetes.secrets.labels.db=postgres
```

11.4.3. Secrets PropertySource の設定プロパティ

以下のプロパティを使用して、Secrets プロパティソースを設定できます。

spring.cloud.kubernetes.secrets.enabled

Secrets プロパティソースを有効にします。型は **Boolean** で、デフォルトは **true** です。

spring.cloud.kubernetes.secrets.name

検索する Secret の名前を設定します。型は **String** で、デフォルトは **\${spring.application.name}** です。

spring.cloud.kubernetes.secrets.labels

Secret の検索に使用されるラベルを設定します。このプロパティは [マップベースのバインディング](#) による定義どおりに動作します。型は **java.util.Map** で、デフォルトは **null** です。

spring.cloud.kubernetes.secrets.paths

Secret がマウントされるパスを設定します。このプロパティは [コレクションベースのバインディング](#) による定義どおりに動作します。型は **java.util.List** で、デフォルトは **null** です。

spring.cloud.kubernetes.secrets.enableApi

API 経由で Secret の消費を有効または無効にします。型は **Boolean** で、デフォルトは **false** です。



注記

API 経由でシークレットにアクセスすることは、セキュリティ上の理由で制限されることがあります。シークレットを POD にマウントする方法が推奨されます。

11.5. PROPERTYSOURCE RELOAD の使用

アプリケーションによっては、外部プロパティソースで変更を検出し、内部の状況を更新して、新しい設定を反映する必要があります。Spring Cloud Kubernetes のリロード機能は、関連する ConfigMap または Secret の変更時にアプリケーションのリロードをトリガーできます。

11.5.1. PropertySource reload の有効化

Spring Cloud Kubernetes の **PropertySource reload** 機能は、デフォルトで無効になっています。

手順

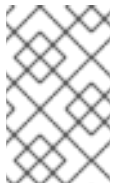
1. クイックスタートの **src/main/resources** ディレクトリーに移動し、**bootstrap.yml** ファイルを開きます。
2. **spring.cloud.kubernetes.reload.enabled=true** 設定プロパティーを変更します。

11.5.2. PropertySource reload のレベル

spring.cloud.kubernetes.reload.strategy プロパティーでは以下のリロードレベルがサポートされます。

refresh

(デフォルト): **@ConfigurationProperties** または **@RefreshScope** アノテーションが付けられた設定 Bean のみがリロードされます。このリロードレベルは、Spring Cloud コンテキストの更新機能を利用します。



注記

PropertySource reload 機能は、リロードストラテジーが **refresh** に設定されている場合に **シンプル**な プロパティー (コレクションではない) のみに使用できます。コレクションによって対応されるプロパティーは起動時に変更しないでください。

restart_context

Spring の **ApplicationContext** 全体が正常に再起動されます。Bean は新しい設定で再作成されます。

shutdown

Spring の **ApplicationContext** がシャットダウンされ、コンテナの再起動がアクティベートされます。このレベルを使用する場合は、デーモンでないすべてのスレッドが **ApplicationContext** にバインドされ、レプリケーションコントローラーまたはレプリカのセットが Pod を再起動するように設定されているようにしてください。

11.5.3. PropertySource reload の例

以下の例では、リロード機能が有効になっている場合の動作について説明します。

手順

1. リロード機能がデフォルト設定 (**refresh** モード) で有効になっていることを仮定します。ConfigMap の変更時に以下の Bean が更新されます。

```
@Configuration
@ConfigurationProperties(prefix = "bean")
public class MyConfig {

    private String message = "a message that can be changed live";

    // getter and setters

}
```

2. 変更の詳細を確認するには、以下のようにメッセージを定期的に出力する別の Bean を作成します。

```

@Component
public class MyBean {

    @Autowired
    private MyConfig config;

    @Scheduled(fixedDelay = 5000)
    public void hello() {
        System.out.println("The message is: " + config.getMessage());
    }
}

```

3. 以下のように ConfigMap を使用すると、アプリケーションによって出力されるメッセージを変更できます。

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: reload-example
data:
  application.properties: |-
    bean.message=Hello World!

```

Pod に関連する ConfigMap の **bean.message** という名前のプロパティに変更を加えると、プログラムの出力に反映されます。

11.5.4. PropertySource reload の操作モード

リロード機能は 2 つの操作モードをサポートします。

event

(デフォルト): Kubernetes API (Web ソケット) を使用して ConfigMap または Secret の変更を監視します。イベントによって設定の再チェックが実行され、変更があった場合はリロードが実行されます。ConfigMap の変更をリッスンするには、サービスアカウントに **view** ロールが必要です。より高いレベルのロール (例: **edit**) が Secret には必要になります (Secret はデフォルトでは監視されません)。

ポーリング

ConfigMap と Secret から定期的に設定を再作成し、設定の変更を確認します。ポーリングの期間は **spring.cloud.kubernetes.reload.period** プロパティを使用して設定でき、デフォルトは **15 秒** です。監視対象のプロパティソースと同じロールが必要です。たとえば、ファイルにマウントされた Secret ソースにポーリングを使用する場合は特定の権限は必要ありません。

11.5.5. PropertySource reload 設定プロパティ

リロード機能の設定には、以下のプロパティを使用できます。

spring.cloud.kubernetes.reload.enabled

プロパティソースおよび設定リロードの監視を有効にします。型は **Boolean** で、デフォルトは **false** です。

spring.cloud.kubernetes.reload.monitoring-config-maps

ConfigMap の変更の監視を許可します。型は **Boolean** で、デフォルトは **true** です。

spring.cloud.kubernetes.reload.monitoring-secrets

Secret の変更の監視を許可します。型は **Boolean** で、デフォルトは **false** です。

spring.cloud.kubernetes.reload.strategy

リロードの実行時に使用するストラテジー (**refresh**、**restart_context**、または **shutdown**)。型は **Enum** で、デフォルトは **refresh** です。

spring.cloud.kubernetes.reload.mode

プロパティソースの変更をリッスンする方法を指定します (**event** または **polling**)。型は **Enum** で、デフォルトは **event** です。

spring.cloud.kubernetes.reload.period

polling ストラテジーの使用時に変更の検証を行う期間をミリ秒単位で指定します。型は **Long** で、デフォルトは **15000** です。

以下の点に注意してください。

- **spring.cloud.kubernetes.reload.*** プロパティは ConfigMap または Secret で使用しないでください。起動時にこのようなプロパティを変更すると、予期せぬ結果が発生する可能性があります。
- **refresh** レベルの使用時に、プロパティまたは ConfigMap 全体を削除しても、Bean は元の状態に復元されません。

第12章 KARAF イメージのアプリケーションの開発

このチュートリアルでは、Karaf イメージのアプリケーションを作成およびデプロイする方法を説明します。

12.1. MAVEN ARCHETYPE を使用した KARAF プロジェクトの作成

Maven archetype を使用して Karaf プロジェクトを作成するには、以下の手順にしたがいます。

手順

1. システムの適切なディレクトリーに移動します。
2. Maven コマンドを起動して、Karaf プロジェクトを作成します。

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
-
DarchetypeCatalog=https://maven.repository.redhat.com/ga/io/fabric8/archetypes/archetypes-
catalog/2.2.0.fuse-sb2-790047-redhat-00004/archetypes-catalog-2.2.0.fuse-sb2-790047-
redhat-00004-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=karaf-camel-log-archetype \
-DarchetypeVersion=2.2.0.fuse-sb2-790047-redhat-00004
```

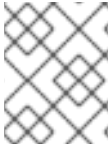
3. archetype プラグインは対話モードに切り替わり、残りのフィールドを追加するよう要求します。

```
Define value for property 'groupId': : org.example.fis
Define value for property 'artifactId': : fuse79-karaf-camel-log
Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': org.example.fis: :
Confirm properties configuration:
groupId: org.example.fis
artifactId: fuse79-karaf-camel-log
version: 1.0-SNAPSHOT
package: org.example.fis
Y: : Y
```

プロンプトが表示されたら、**org.example.fis** を **groupId** の値として入力し、**fuse79-karaf-camel-log** を **artifactId** の値として入力します。残りのフィールドにはデフォルト値を使用します。

4. 上記のコマンドが BUILD SUCCESS 状態で終了した場合、**fuse79-karaf-camel-log** サブディレクトリー内に新しい Fuse on OpenShift プロジェクトが作成されているはずです。
5. これで、**fuse79-karaf-camel-log** プロジェクトをビルドおよびデプロイできるようになりました。OpenShift にログインしている状態で、**fuse79-karaf-camel-log** プロジェクトのディレクトリーに移動し、以下のようにプロジェクトをビルドおよびデプロイします。

```
cd fuse79-karaf-camel-log
mvn oc:deploy -Popenshift
```

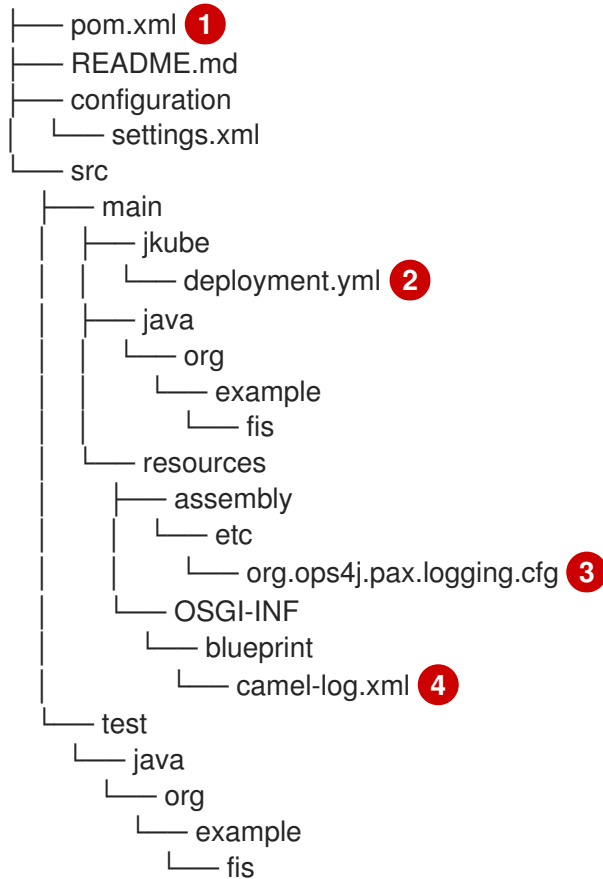


注記

使用できる Karaf archetype の完全リストは [Karaf Archetype カタログ](#) を参照してください。

12.2. CAMEL KARAF アプリケーションの構造

Camel Karaf アプリケーションのディレクトリ構造は以下のようになります。



このうち、Karaf アプリケーションの開発に重要なファイルは次のとおりです。

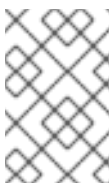
- 1 pom.xml: 追加の依存関係が含まれます。依存関係を **pom.xml** ファイルに追加できます。たとえば、ロギングの場合は SLF4J を使用できます。

```

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
</dependency>

```

- 2 src/main/jkube/deployment.yml: openshift-maven-plugin によって生成されるデフォルトの OpenShift 設定ファイルにマージされる追加設定を提供します。



注記

このファイルは Karaf アプリケーションの一部として使用されませんが、CPU やメモリー使用量などのリソースを制限するためにすべてのクイックスタートで使用されます。

- 3 `org.ops4j.pax.logging.cfg`: ログレベルのカスタマイズ方法を示し、デフォルトのログレベルである INFO ではなく、DEBUG を設定します。
- 4 `camel-log.xml`: アプリケーションのソースコードが含まれます。

12.3. KARAF ARCHETYPE カタログ

Karaf archetype カタログには以下の例が含まれます。

表12.1 Karaf Maven archetype

名前	説明
karaf-camel-amq-archetype	Camel amq コンポーネントを使用して、Apache ActiveMQ メッセージブローカーとメッセージを送受信する方法を実証します。
karaf-camel-log-archetype	メッセージを 5 秒おきにサーバーに記録する、簡単な Apache Camel アプリケーションを実証します。
karaf-camel-rest-sql-archetype	Camel の REST DSL とともに JDBC を介して SQL を使用し、RESTful API を公開する方法を実証します。
karaf-cxf-rest-archetype	CXF を使用して RESTful(JAX-RS) Web サービスを作成し、OSGi HTTP サービス経由で公開する方法を実証します。

12.4. FABRIC8 KARAF 機能の使用

Fabric8 は Apache Karaf のサポートを提供し、Kubernetes の OSGi アプリケーションの開発を容易にします。

Fabric8 の重要な機能を以下に示します。

- Blueprint XML ファイルでプレースホルダーを解決するさまざまなストラテジー。
- 環境変数
- システムプロパティー
- サービス
- Kubernetes の ConfigMap
- Kubernetes の Secret
- Kubernetes の設定マップを使用して、OSGi 設定管理を動的に更新します。
- OSGi サービスの Kubernetes ヘルスチェックを提供します。

12.4.1. Fabric8 Karaf 機能の追加

この機能を追加するには、**fabric8-karaf-features** 依存関係をプロジェクトの POM ファイルに追加します。

手順

1. プロジェクトの **pom.xml** ファイルを開き、**fabric8-karaf-features** 依存関係を追加します。

```
<dependency>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-karaf-features</artifactId>
  <version>${fabric8.version}</version>
  <classifier>features</classifier>
  <type>xml</type>
</dependency>
```

Fabric8 Karaf 機能が Karaf サーバーにインストールされます。

12.4.2. Fabric8 Karaf Core バンドル機能の追加

fabric8-karaf-core バンドルは、Blueprint および ConfigAdmin エクステンションによって使用される機能を提供します。

手順

1. プロジェクトの **pom.xml** ファイルを開き、**fabric8-karaf-core** を **startupFeatures** セクションに追加します。

```
<startupFeatures>
  ...
  <feature>fabric8-karaf-core</feature>
  ...
</startupFeatures>
```

これにより、**fabric8-karaf-core** 機能がカスタム Karaf ディストリビューションに追加されます。

12.4.3. プロパティープレースホルダーサービスのオプション設定

fabric8-karaf-core バンドルは、以下のインターフェイスで **PlaceholderResolver** サービスをエクスポートします。

```
public interface PlaceholderResolver {
  /**
   * Resolve a placeholder using the strategy indicated by the prefix
   *
   * @param value the placeholder to resolve
   * @return the resolved value or null if not resolved
   */
  String resolve(String value);

  /**
   * Replaces all the occurrences of variables with their matching values from the resolver using the
   * given source string as a template.
   *
   */
}
```

```

    * @param source the string to replace in
    * @return the result of the replace operation
    */
    String replace(String value);

    /**
     * Replaces all the occurrences of variables within the given source builder with their matching
     values from the resolver.
     *
     * @param value the builder to replace in
     * @return true if altered
     */
    boolean replaceIn(StringBuilder value);

    /**
     * Replaces all the occurrences of variables within the given dictionary
     *
     * @param dictionary the dictionary to replace in
     * @return true if altered
     */
    boolean replaceAll(Dictionary<String, Object> dictionary);

    /**
     * Replaces all the occurrences of variables within the given dictionary
     *
     * @param dictionary the dictionary to replace in
     * @return true if altered
     */
    boolean replaceAll(Map<String, Object> dictionary);
}

```

PlaceholderResolver サービスは、異なるプロパティープレースホルダー解決ストラテジーのコレクターとして動作します。デフォルトで提供される解決ストラテジーの一覧は [解決ストラテジー](#) を参照してください。プロパティープレースホルダーサービスのオプションを設定するには、システムプロパティと環境変数のどちらかか両方を使用します。

手順

1. OpenShift で ConfigMap にアクセスするには、サービスアカウントに view パーミッションが必要になります。view パーミッションをサービスアカウントに追加します。

```
oc policy add-role-to-user view system:serviceaccount:$(oc project -q):default -n $(oc project -q)
```

2. API 経由での Secret へのアクセスは制限されている可能性があるため、Secret を Pod にマウントします。
3. Secret は Pod ではボリュームマウントとして使用でき、以下のように secret という名前のディレクトリーにマップされます。

```
containers:
-
  env:
  - name: FABRIC8_K8S_SECRETS_PATH
    value: /etc/secrets
```

```

    volumeMounts:
    - name: activemq-secret-volume
      mountPath: /etc/secrets/activemq
      readOnly: true
    - name: postgres-secret-volume
      mountPath: /etc/secrets/postgres
      readOnly: true

    volumes:
    - name: activemq-secret-volume
      secret:
        secretName: activemq
    - name: postgres-secret-volume
      secret:
        secretName: postgres

```

12.4.4. カスタムのプロパティープレースホルダーリゾルバーの追加

カスタムのプレースホルダーリゾルバーを追加して、カスタムの暗号化などの特定の必要項目をサポートできます。また、**PlaceholderResolver** サービスを使用して、Blueprint および ConfigAdmin がリゾルバーを使用できるようにすることもできます。

手順

1. 以下の mvn 依存関係をプロジェクトの **pom.xml** に追加します。

pom.xml

```

---
<dependency>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-karaf-core</artifactId>
</dependency>
---

```

2. **PropertiesFunction** インターフェイスを実装し、CSR を使用して OSGi サービスとして登録します。

```

import io.fabric8.karaf.core.properties.function.PropertiesFunction;
import org.apache.felix.scr.annotations.Component;
import org.apache.felix.scr.annotations.ConfigurationPolicy;
import org.apache.felix.scr.annotations.Service;

@Component(
    immediate = true,
    policy = ConfigurationPolicy.IGNORE,
    createPid = false
)
@Service(PropertiesFunction.class)
public class MyPropertiesFunction implements PropertiesFunction {
    @Override
    public String getName() {
        return "myResolver";
    }
}

```

```

@Override
public String apply(String remainder) {
    // Parse and resolve remainder
    return remainder;
}
}

```

3. 以下のように、設定管理でリゾルバーを参照することができます。

properties

```
my.property = ${myResolver:value-to-resolve}
```

12.4.5. 解決ストラテジーの一覧

PlaceholderResolver サービスは、異なるプロパティプレースホルダー解決ストラテジーのコレクターとして動作します。デフォルトで提供される解決ストラテジーを以下の表に示します。

1. 解決ストラテジーの一覧

接頭辞	例	説明
env	env:JAVA_HOME	OS 環境変数からプロパティを検索します。
sys	sys:java.version	Java JVM システムプロパティからプロパティを検索します。
service	service:amq	サービス命名規則を使用して、OS 環境変数からプロパティを検索します。
service.host	service.host:amq	ホスト名の部分のみを返すサービス命名規則を使用して、OS 環境変数からプロパティを検索します。
service.port	service.port:amq	ポートの部分のみを返すサービス命名規則を使用して、OS 環境変数からプロパティを検索します。
k8s:map	k8s:map:myMap/myKey	Kubernetes ConfigMap (API 経由) からプロパティを検索します。
k8s:secret	k8s:secret:amq/password	Kubernetes の Secret (API またはボリュームマウント経由) からプロパティを検索します。

12.4.6. プロパティプレースホルダーサービスのオプション一覧

プロパティースペースホルダーサービスは以下のオプションをサポートします。

1. プロパティースペースホルダーサービスのオプション一覧

名前	デフォルト	説明
fabric8.placeholder.prefix	<code>\$[</code>	スペースホルダーの接頭辞。
fabric8.placeholder.suffix	<code>]</code>	スペースホルダーの接尾辞。
fabric8.k8s.secrets.path	<code>null</code>	Secret がマップされたパスのコンマ区切りリスト。
fabric8.k8s.secrets.api.enabled	<code>false</code>	API 経由で Secret の消費を有効または無効にする。

12.5. FABRIC8 KARAF CONFIG 管理サポートの追加

12.5.1. Fabric8 Karaf Config 管理サポートの追加

カスタムの Karaf ディストリビューションに Fabric8 Karaf Config 管理サポートを追加できます。

手順

- プロジェクトの **pom.xml** ファイルを開き、**fabric8-karaf-cm** を **startupFeatures** セクションに追加します。

pom.xml

```
<startupFeatures>
...
<feature>fabric8-karaf-cm</feature>
...
</startupFeatures>
```

12.5.2. ConfigMap インジェクションの追加

fabric8-karaf-cm は、Karaf の **ConfigAdmin** で **ConfigMap** の値をインジェクトする **ConfigAdmin** ブリッジを提供します。

手順

- ConfigMap が ConfigAdmin ブリッジによって追加されるようにするには、ConfigMap に **karaf.pid** をラベル付けする必要があります。**karaf.pid** の値はコンポーネントの pid に対応します。以下に例を示します。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: myconfig
  labels:
```

```
karaf.pid: com.mycompany.bundle
data:
  example.property.1: my property one
  example.property.2: my property two
```

2. 設定を定義するには、単一のプロパティ名を使用できます。ほとんどの場合で個別のプロパティは動作します。これは **karaf/etc** の pid ファイルと同じです。以下に例を示します。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: myconfig
  labels:
    karaf.pid: com.mycompany.bundle
data:
  com.mycompany.bundle.cfg: |
    example.property.1: my property one
    example.property.2: my property two
```

12.5.3. 設定プラグイン

fabric8-karaf-cm は、設定プロパティプレースホルダーを解決する **ConfigurationPlugin** を提供します。

fabric8-karaf-cm プラグインでプロパティの置き換えを有効にするには、**fabric8.config.plugin.enabled** Java プロパティを **true** に設定する必要があります。たとえば、以下のように Karaf イメージで **JAVA_OPTIONS** 環境変数を使用して、このプロパティを設定することができます。

```
JAVA_OPTIONS=-Dfabric8.config.plugin.enabled=true
```

12.5.4. 設定プロパティプレースホルダー

以下に設定プロパティプレースホルダーの例を示します。

my.service.cfg

```
amq.usr = ${k8s:secret:[env:ACTIVEMQ_SERVICE_NAME]/username}
amq.pwd = ${k8s:secret:[env:ACTIVEMQ_SERVICE_NAME]/password}
amq.url = tcp://${env+service:ACTIVEMQ_SERVICE_NAME}
```

my-service.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
  xsi:schemaLocation="
    http://www.osgi.org/xmlns/blueprint/v1.0.0
    https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
    http://camel.apache.org/schema/blueprint
    http://camel.apache.org/schema/blueprint/camel-blueprint.xsd">
```

```

<cm:property-placeholder persistent-id="my.service" id="my.service" update-strategy="reload"/>

<bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="userName" value="${amq.usr}"/>
  <property name="password" value="${amq.pwd}"/>
  <property name="brokerURL" value="${amq.url}"/>
</bean>
</blueprint>

```

12.5.5. Fabric8 Karaf 設定管理オプション

Fabric8 Karaf 設定管理は以下のオプションをサポートします。

名前	デフォルト	説明
fabric8.config.plugin.enabled	false	ConfigurationPlugin を有効にします。
fabric8.cm.bridge.enabled	true	ConfigAdmin ブリッジを有効にします。
fabric8.config.watch	true	ConfigMap の変更の監視を有効にします。
fabric8.config.merge	false	ConfigAdmin での ConfigMap 値のマージを有効にします。
fabric8.config.meta	true	ConfigAdmin ブリッジでの ConfigMap のインジェクトを有効にします。
fabric8.pid.label	karaf.pid	ConfigAdmin ブリッジが検索するラベルを定義します (よって、選択する必要がある ConfigMap にそのラベルが存在する必要がある、その値で PID の関連付けが判断されます)。

名前	デフォルト	説明
fabric8.pid.filters	empty	<p>ConfigAdmin ブリッジの追加の条件を定義し、ConfigMap を選択します。サポートされる構文は以下のとおりです。</p> <ul style="list-style-type: none"> 異なるラベルの条件はコンマ (,) で区切れ、ラベルの間の AND として解釈されます。 ラベル内のセミコロン (;) は OR として解釈され、ラベル値で条件の区切り文字として使用できます。 <p>たとえば、 - Dfabric8.pid.filters=appName=A;B,database.name=my.oracle.datasource のようなフィルターの場合、A または B の値を持つラベル <code>appName</code> と、<code>my.oracle.datasource</code> と同等のラベル <code>database.name</code> の両方を持つすべての ConfigMap を提供します。</p>



重要

ConfigurationPlugin には **Aries Blueprint CM 1.0.9** 以上が必要です。

12.6. FABRIC8 KARAF BLUEPRINT サポートの追加

fabric8-karaf-blueprint は [Aries PropertyEvaluator](#) と、**fabric8-karaf-core** からのプロパティプレースホルダーリゾルバーを使用して、Blueprint XML ファイルのプレースホルダーを解決します。

手順

- カスタム Karaf ディストリビューションに Blueprint サポートの機能を含めるには、プロジェクトの **pom.xml** ファイルの **startupFeatures** セクションに **fabric8-karaf-blueprint** を追加します。

```
<startupFeatures>
...
<feature>fabric8-karaf-blueprint</feature>
...
</startupFeatures>
```

例

Fabric8 エバリュエーターは、**\${env+service:MY_ENV_VAR}** のようなチェーンされたエバリュエーターをサポートします。**MY_ENV_VAR** 変数を環境変数に対して解決する必要があります。結果はサービス関数を使用して解決されます。以下に例を示します。


```
<?xml version="1.0" encoding="UTF-8"?>

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.2.0"
  xsi:schemaLocation="
    http://www.osgi.org/xmlns/blueprint/v1.0.0
    https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
    http://camel.apache.org/schema/blueprint
    http://camel.apache.org/schema/blueprint/camel-blueprint.xsd
    http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.3.0
    http://aries.apache.org/schemas/blueprint-ext/blueprint-ext-1.3.xsd">

  <ext:property-placeholder evaluator="fabric8" placeholder-prefix="$[" placeholder-suffix=""]"/>

  <bean id="activemq" class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="userName"
value="$[k8s:secret:$[env:ACTIVEMQ_SERVICE_NAME]/username]"/>
    <property name="password"
value="$[k8s:secret:$[env:ACTIVEMQ_SERVICE_NAME]/password]"/>
    <property name="brokerURL" value="tcp://$[env+service:ACTIVEMQ_SERVICE_NAME]"/>
  </bean>
</blueprint>
```

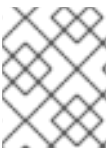


重要

入れ子のプロパティープレースホルダーの置換には **Aries Blueprint Core 1.7.0** 以上が必要です。

12.7. FABRIC8 KARAF ヘルスチェックの有効化

fabric8-karaf-checks をスタートアップ機能としてインストールすることが推奨されます。有効にすると、Karaf サーバーは <http://0.0.0.0:8181/readiness-check> および <http://0.0.0.0:8181/health-check> URL を公開できます。これらの URL は、Kubernetes が readiness probe および liveness probe のために使用できます。



注記

これらの URL は、以下が true の場合に HTTP 200 ステータスコードのみで応答します。

- OSGi Framework が開始している。
- すべての OSGi バンドルが開始している。
- すべてのブート機能がインストールされている。
- デプロイされた Blueprint バンドルがすべて作成済み状態である。
- デプロイされた SCR バンドルはすべてアクティブ、登録済み、またはファクトリー状態である。
- すべての Web バンドルが Web サーバーにデプロイされている。

- 作成された Camel コンテキストがすべて開始済み状態である。

手順

1. プロジェクトの **pom.xml** を開き、**fabric8-karaf-checks** 機能を **startupFeatures** セクションに追加します。

pom.xml

```
<startupFeatures>
...
<feature>fabric8-karaf-checks</feature>
...
</startupFeatures>
```

oc:resources ゴールは **fabric8-karaf-checks** 機能が使用されているかを検出し、readiness および liveness probe について Kubernetes をコンテナの設定に追加します。

12.7.1. ヘルスチェックの設定

デフォルトでは、**fabric8-karaf-checks** エンドポイントは、ポート **8181** で実行しているビルトイン HTTP サーバーエンジン (Undertow) に登録されます。コンテナ内で長期実行されている他の HTTP プロセスによってヘルスおよび readiness チェックリクエストがブロックされないようにするために、エンドポイントを別の Undertow コンテナに登録することができます。

これらのチェックは、以下のプロパティーを設定し、**etc/io.fabric8.checks.cfg** ファイルで設定できます。

- **httpPort**: このプロパティーが指定され、有効なポート番号である場合、**readiness-check** および **health-check** エンドポイントは Undertow サーバーの別のインスタンスに登録されます。
- **readinessCheckPath** および **healthCheckPath** プロパティーを使用すると、readiness およびヘルスチェックに使用できる実際の URI を設定することができます。デフォルトでは、これらは以前の値と同じです。



注記

これらのプロパティーは、Fuse-Karaf の起動後に変更できますが、カスタム Karaf distro の一部である **etc/io.fabric8.checks.cfg** ファイルで指定することもできます。このファイルは、追加設定なしで **fabric8-karaf-checks** 機能を実行する場合に使用されます。

以下の例は、**etc/io.fabric8.checks.cfg** ファイルの health および readiness プロパティーの設定を示しています。

例

```
httpPort = 8182
readinessCheckPath = /readiness-check
healthCheckPath = /health-check
```

12.8. カスタムヘルスチェックの追加

追加のカスタムヘルスチェックを提供して、リクエストを処理する準備が整う前に Karaf サーバーが

ユーザートラフィックを受信しないようにすることができます。カスタムヘルスチェックを有効にするには、**io.fabric8.karaf.checks.HealthChecker** または **io.fabric8.karaf.checks.ReadinessChecker** インターフェイスを実装し、これらのオブジェクトを OSGi レジストリーに登録する必要があります。

手順

- 以下の mvn 依存関係をプロジェクトの **pom.xml** ファイルに追加します。

pom.xml

```
<dependency>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-karaf-checks</artifactId>
</dependency>
```



注記

SCR を使用すると、最も簡単に OSGi レジストリーでオブジェクトを作成および登録できます。

例

以下は、ヘルスチェックを実行して空きのディスク領域が確保されるようにする例になります。

```
import io.fabric8.karaf.checks.*;
import org.apache.felix.scr.annotations.*;
import org.apache.commons.io.FileSystemUtils;
import java.util.Collections;
import java.util.List;

@Component(
  name = "example.DiskChecker",
  immediate = true,
  enabled = true,
  policy = ConfigurationPolicy.IGNORE,
  createPid = false
)
@Service({HealthChecker.class, ReadinessChecker.class})
public class DiskChecker implements HealthChecker, ReadinessChecker {

  public List<Check> getFailingReadinessChecks() {
    // lets just use the same checks for both readiness and health
    return getFailingHealthChecks();
  }

  public List<Check> getFailingHealthChecks() {
    long free = FileSystemUtils.freeSpaceKb("/");
    if (free < 1024 * 500) {
      return Collections.singletonList(new Check("disk-space-low", "Only " + free + "kb of disk space left."));
    }
    return Collections.emptyList();
  }
}
```

第13章 JBOSS EAP イメージのアプリケーションの開発

JBoss EAP で Fuse アプリケーションを開発するために、S2I ソースワークフローを使用して、EAP で Red Hat Camel CDI の OpenShift プロジェクトを作成することもできます。

前提条件

- OpenShift が適切に稼働し、Fuse イメージストリームがすでに OpenShift にインストールされているようにしてください。[管理者向けの基本情報](#) を参照してください。
- Maven リポジトリが Fuse に対して設定されているようにしてください。詳細は [Maven リポジトリの設定](#) を参照してください。

13.1. S2I ソースワークフローを使用した JBOSS EAP プロジェクトの作成

JBoss EAP で Fuse アプリケーションを開発するために、S2I ソースワークフローを使用して、EAP で Red Hat Camel CDI の OpenShift プロジェクトを作成することもできます。

手順

1. **view** ロールをデフォルトのサービスアカウントに追加して、クラスターリングを有効にします。これにより、**default** サービスアカウントへの **view** アクセス権限がユーザーに付与されます。サービスアカウントは、ビルド、デプロイメント、およびその他の Pod を実行するために各プロジェクトで必要になります。シェルプロンプトに以下の **oc** クライアントコマンドを入力します。

```
oc login -u developer -p developer
oc policy add-role-to-user view -z default
```

2. インストールされた Fuse on OpenShift テンプレートを表示します。

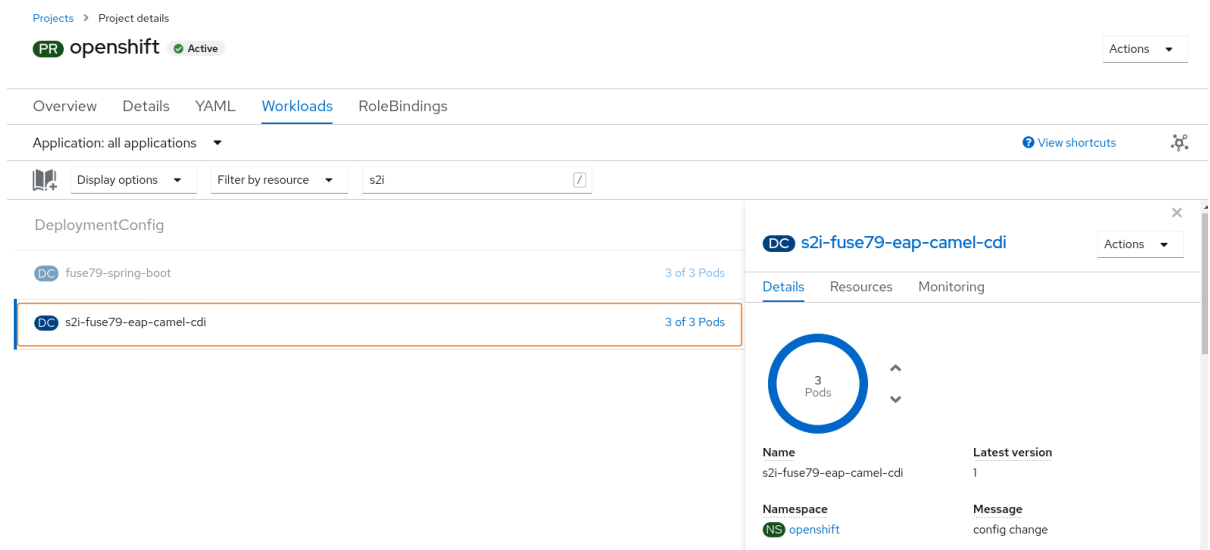
```
oc get template -n openshift
```

3. 以下のコマンドを入力し、**Red Hat Fuse 7.12 Camel CDI with EAP** クイックスタートの実行に必要なリソースを作成します。これにより、クイックスタートのデプロイメント設定およびビルド設定が作成されます。ターミナルにクイックスタートや作成されたリソースに関する情報が表示されます。

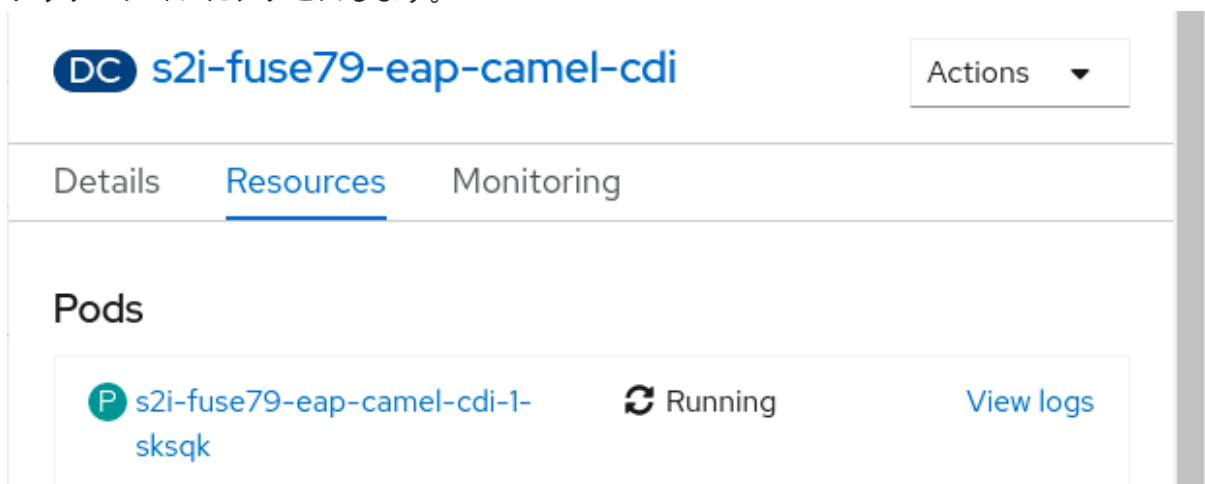
```
oc new-app s2i-fuse7-eap-camel-cdi

--> Creating resources ...
service "s2i-fuse7-eap-camel-cdi" created
service "s2i-fuse7-eap-camel-cdi-ping" created
route.route.openshift.io "s2i-fuse7-eap-camel-cdi" created
imagestream.image.openshift.io "s2i-fuse7-eap-camel-cdi" created
buildconfig.build.openshift.io "s2i-fuse7-eap-camel-cdi" created
deploymentconfig.apps.openshift.io "s2i-fuse7-eap-camel-cdi" created
--> Success
Access your application via route 's2i-fuse7-eap-camel-cdi-OPENSHIFT_IP_ADDR'
Build scheduled, use 'oc logs -f bc/s2i-fuse7-eap-camel-cdi' to track its progress.
Run 'oc status' to view your app.
```

4. ブラウザーで https://OPENSIFT_IP_ADDR の OpenShift Web コンソールに移動します (**OPENSIFT_IP_ADDR** はクラスターの IP アドレスに置き換えます)。クレデンシャル (例: ユーザー名 **developer**、パスワード **developer**) を使用して、コンソールにログインします。
5. 左側のパネルで **Home** を展開します。**Status** をクリックして **Project Status** ページを表示します。選択された namespace (例: openshift) の既存のアプリケーションがすべて表示されます。
6. **s2i-fuse7-eap-camel-cdi** をクリックして、クイックスタートの **Overview** 情報ページを表示します。



7. **Resources** タブをクリックした後に Routes セクションに表示されたリンクをクリックし、アプリケーションにアクセスします。



リンクの形式は http://s2i-fuse7-eap-camel-cdi-OPENSIFT_IP_ADDR になります。これにより、ブラウザーで以下のようなメッセージが表示されます。

```
Hello world from 172.17.0.3
```


8. URL の name パラメーターを使用して名前を指定することもできます。たとえば、ブラウザーに URL <http://s2i-fuse7-eap-camel-cdi-openshift.apps.cluster-name.openshift.com/?name=jdoo> を入力すると、以下のような応答が表示されます。

```
Hello jdoo from 172.17.0.3
```

9. **View Logs** をクリックし、アプリケーションのログを表示します。

10. 稼働中の Pod を終了するには以下を行います。

a. **Overview** タブをクリックし、アプリケーションの Overview 情報ページに戻ります。

b. Desired Count の横にある  アイコンをクリックします。**Edit Count** ウィンドウが表示されます。

c. 下矢印を使用して値をゼロにし、Pod を停止します。

13.2. JBOSS EAP アプリケーションの構造

Red Hat Fuse 7.12 Camel CDI with EAP サンプルのソースコードは、以下の場所にあります。

<https://github.com/wildfly-extras/wildfly-camel-examples/tree/wildfly-camel-examples-5.2.0.fuse-720021/camel-cdi>

Camel on EAP アプリケーションのディレクトリー構造は以下のようになります。

```

├── pom.xml
├── README.md
├── configuration
│   └── settings.xml
├── src
│   └── main
│       ├── java
│       │   ├── org
│       │   │   ├── wildfly
│       │   │   │   ├── camel
│       │   │   │   │   ├── examples
│       │   │   │   │   │   ├── cdi
│       │   │   │   │   │   │   ├── camel
│       │   │   │   │   │   │   │   ├── MyRouteBuilder.java
│       │   │   │   │   │   │   │   ├── SimpleServlet.java
│       │   │   │   │   │   │   │   └── SomeBean.java
│       │   └── webapp
│       │       ├── WEB-INF
│       │       │   └── beans.xml

```

このうち、JBoss EAP アプリケーションの開発に重要なファイルは次のとおりです。

pom.xml

追加の依存関係が含まれます。

13.3. JBOSS EAP クイックスタートテンプレート

Fuse on JBoss EAP には以下の S2I テンプレートが提供されます。

表13.1 JBoss EAP S2I テンプレート

名前	説明
JBoss Fuse 7.12 Camel A-MQ with EAP (eap-camel-amq-template)	camel-activemq コンポーネントを使用して、OpenShift 上で実行している AMQ メッセージブローカーに接続します。ブローカーがすでにデプロイされていることを前提とします。
Red Hat Fuse 7.12 Camel CDI with EAP (eap-camel-cdi-template)	camel-cdi コンポーネントを使用して、CDI Bean を Camel ルートに統合します。
Red Hat Fuse 7.12 CXF JAX-RS with EAP (eap-camel-cxf-jaxrs-template)	camel-cxf コンポーネントを使用して JAX-RS REST サービスを作成および消費します。
Red Hat Fuse 7.12 CXF JAX-WS with EAP (eap-camel-cxf-jaxws-template)	camel-cxf コンポーネントを使用して JAX-WS Web サービスを作成および消費します。

第14章 FUSE ON OPENSIFT での永続ストレージの使用

Fuse on OpenShift アプリケーションは、永続ファイルシステムがない OpenShift コンテナをベースとしています。アプリケーションを起動するたびに、イミュータブルな Docker 形式のイメージで新しいコンテナに起動されます。そのため、コンテナが停止するとファイルシステムの永続データが失われます。しかし、アプリケーションは一部の状態を永続ストアのデータとして保存する必要があり、アプリケーションは共通のデータストアへのアクセスを共有することがあります。OpenShift プラットフォームは、外部のストアを永続ストレージとして提供することをサポートします。

14.1. ボリュームおよびボリュームタイプ

OpenShift では、Pod およびコンテナは [ボリューム](#) を、複数のローカルまたはネットワーク接続ストレージエンドポイントが対応するファイルシステムとしてマウントすることができます。

ボリュームタイプには以下が含まれます。

- `emptydir` (空のディレクトリー): デフォルトのボリュームタイプです。これは、Pod がローカルホストに作成されると割り当てられるディレクトリーです。サーバー全体でコピーされず、Pod を削除するとディレクトリーも削除されます。
- `configmap`: 名前付きの `configmap` からキーと値のペアで内容が追加されるディレクトリーです。
- `hostPath` (ホストディレクトリー): 任意のホスト上の特定のパスを持つディレクトリーで、昇格された権限が必要です。
- `secret` (マウントされたシークレット): シークレットボリュームは名前付きのシークレットを提供されるディレクトリーにマウントします。
- `persistentvolumeclaim` または `pvc` (永続ボリュームクレーム): コンテナのボリュームディレクトリーを、名前でも割り当てた永続ボリュームクレームにリンクします。永続ボリュームクレームは、ストレージ割り当てのリクエストです。クレームがバインドされていないと、Pod が開始しないことに注意してください。

ボリュームは Pod レベルで設定され、`hostPath` を使用して外部ストレージへの直接アクセスのみが可能です。そのため、複数の Pod の共有リソースへのアクセスを `hostPath` ボリュームとして管理することは困難です。

14.2. PERSISTENTVOLUMES

PersistentVolumes は、NFS、Ceph RBD、AWS Elastic Block Store (EBS) などのさまざまな種類のネットワークストレージが対応するクラスター全体のストレージをクラスター管理者がプロビジョニングできるようにします。また、**PersistentVolumes** は容量、アクセスモード、およびリサイクルポリシーも指定します。これにより、基盤のリソースに関係なく、複数のプロジェクトの Pod が永続ストレージにアクセスできます。

さまざまな種類の **PersistentVolumes** の作成に関しては、[永続ストレージの設定](#) を参照してください。

14.3. 永続ボリュームの設定

永続ボリュームをプロビジョニングするには、設定ファイルを作成します。その後、**PersistentVolume** クレームを作成すると、このストレージにアクセスできます。

手順

1. 以下の設定例を使用して **pv.yaml** という名前の設定ファイルを作成します。これは、ホストマシン上のパスを pv001 という名前の PersistentVolume としてプロビジョニングします。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 2Mi
  hostPath:
    path: /data/pv0001/
```

この例では、ホストパスは **/data/pv0001** で、ストレージの容量が 2MB に制限されます。たとえば、OpenShift CDK を使用する場合は、OpenShift クラスターをホストする仮想マシンから **/data/pv0001** ディレクトリーをプロビジョニングします。

2. **PersistentVolume** を作成します。

```
oc create -f pv.yaml
```

3. **PersistentVolume** の作成を検証します。これは、OpenShift クラスターに設定されたすべての **PersistentVolumes** をリストします。

```
oc get pv
```

14.4. PERSISTENTVOLUMECLAIMS の作成

PersistentVolume はストレージエンドポイントを OpenShift クラスターの名前付きのエンティティーとして公開します。プロジェクトからこのストレージにアクセスするには、**PersistentVolume** にアクセスできる **PersistentVolumeClaims** を作成する必要があります。**PersistentVolumeClaims** は、特定のストレージ容量およびアクセスモードのカスタマイズされたクレームで各プロジェクトに作成されます。

手順

- 以下の設定例は、pv0001 という名前の **PersistentVolume** に対して1度だけ読み書きアクセスする 1MB のストレージの pvc0001 という名前のクレームを作成します。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc0001
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Mi
```

14.5. POD での永続ボリュームの使用

Pod はボリュームマウントを使用してファイルマウントの場所を定義し、ボリュームを使用して **PersistentVolumeClaims** 参照を定義します。

手順

1. ファイルシステムの **/usr/share/data** に **PersistentVolumeClaim** pvc0001 をマウントする、コンテナ設定の例を以下のように作成します。

```
spec:
  template:
    spec:
      containers:
        - volumeMounts:
            - name: vol0001
              mountPath: /usr/share/data
          volumes:
            - name: vol0001
              persistentVolumeClaim:
                claimName: pvc0001
```

アプリケーションによって **/usr/share/data** ディレクトリーに書き込まれたすべてのデータがコンテナの再起動後も保持されるようになりました。

2. この設定を Fuse on OpenShift アプリケーションの **src/main/jkube/deployment.yml** ファイルに追加し、以下のコマンドを使用して OpenShift リソースを作成します。

```
mvn oc:resource-apply
```

3. 作成した **DeploymentConfiguration** にボリュームマウントとボリュームがあることを確認します。

```
oc describe deploymentconfig <application-dc-name>
```

Fuse on OpenShift クイックスタートでは、**<application-dc-name>** を Maven プロジェクト名 (例: **spring-boot-camel**) に置き換えます。

第15章 FUSE ON OPENSIFT のパッチ適用

以下のタスクを1つ以上実行して、Fuse on OpenShift 製品を最新のパッチレベルにする必要がある場合があります。

Fuse on OpenShift イメージのパッチ適用

OpenShift サーバーの Fuse on OpenShift イメージを更新し、新しいアプリケーションのビルドが Fuse ベースイメージのパッチが適用されたバージョンをベースにするようにします。

BOM を使用したアプリケーション依存関係のパッチ適用

プロジェクト POM ファイルの依存関係を更新し、アプリケーションがパッチが適用されたバージョンの Maven アーティファクトを使用するようにします。

Fuse on OpenShift テンプレートのパッチ適用

OpenShift サーバーの Fuse on OpenShift テンプレートを更新し、Fuse on OpenShift テンプレートで作成された新しいプロジェクトがパッチが適用されたバージョンの Maven アーティファクトを使用するようにします。

15.1. BOM および MAVEN 依存関係に関する重要事項

Fuse on OpenShift のコンテキストでは、アプリケーションはすべて Red Hat Maven リポジトリからダウンロードした Maven アーティファクトを使用してビルドされます。そのため、アプリケーションコードにパッチを適用するには、プロジェクトの POM ファイルを編集し、Maven 依存関係を変更して適切な Fuse on OpenShift パッチバージョンを使用することのみが必要となります。

Fuse on OpenShift の Maven 依存関係をすべて一緒にアップグレードし、すべて同じパッチバージョンの依存関係をプロジェクトが使用することが重要になります。Fuse on OpenShift は、一緒にビルドおよびテストされた、Maven アーティファクトのセットで設定されます。Fuse on OpenShift の異なるパッチレベルの Maven アーティファクトを一緒に使用すると、Red Hat がテストおよびサポートしていない信頼できない設定になる可能性があります。このような状況を回避する最も簡単な方法は、Fuse on OpenShift でサポートされる Maven アーティファクトのバージョンをすべて定義する BOM (Bill of Materials) ファイルを Maven で使用することです。BOM ファイルのバージョンを更新すると、プロジェクトの POM にあるすべての Fuse on OpenShift Maven アーティファクトのバージョンが自動的に更新されます。

Fuse on OpenShift Maven archetype または Fuse on OpenShift テンプレートによって生成された POM ファイルには、BOM ファイルを使用する標準レイアウトがあり、必要な特定のプラグインのバージョンを定義します。標準レイアウトはアプリケーションの依存関係のパッチ適用やアップグレードを大変容易にするため、独自のアプリケーションでこの標準レイアウトを使用することが推奨されます。

15.2. FUSE ON OPENSIFT イメージのパッチ適用

Fuse on OpenShift イメージは、メインの Fuse 製品から独立して更新されます。Fuse on OpenShift イメージにパッチが必要な場合、更新されたイメージは標準の Fuse on OpenShift イメージストリームで利用可能になり、更新されたイメージを **registry.redhat.io** からダウンロードできます。Fuse on OpenShift は以下のイメージストリームを提供します (OpenShift イメージストリーム名によって識別されます)。

- **fuse-java-openshift-rhel8**
- **fuse-java-openshift-jdk11-rhel8**
- **fuse-karaf-openshift-rhel8**
- **fuse-eap-openshift-jdk8-rhel7**

- **fuse-eap-openshift-jdk11-rhel8**
- **fuse-console-rhel8**
- **fuse-apicurito-generator-rhel8**
- **fuse-apicurito-rhel8**

手順

1. 通常、Fuse on OpenShift イメージストリームは、OpenShift サーバーの **openshift** プロジェクトにインストールされます。OpenShift の Fuse on OpenShift イメージの状態をチェックするには、管理者として OpenShift にログインし、以下のコマンドを入力します。

```
$ oc get is -n openshift
NAME                                DOCKER REPO                                TAGS
UPDATED
fuse-console-rhel8                  172.30.1.1:5000/openshift/fuse7/fuse-console-rhel8
1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9  About an hour ago
fuse7-eap-openshift-jdk8-rhel7      172.30.1.1:5000/openshift/fuse7/fuse-eap-openshift-
jdk8-rhel7 1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9  About an hour ago
fuse7-eap-openshift-jdk11-rhel8     172.30.1.1:5000/openshift/fuse7/fuse-eap-openshift-
jdk11-rhel8 1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9  About an hour ago
fuse7-java-openshift-rhel8          172.30.1.1:5000/openshift/fuse7/fuse-java-openshift-rhel8
1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9  About an hour ago
fuse7-java-openshift-jdk11-rhel8    172.30.1.1:5000/openshift/fuse7/fuse-java-openshift-
jdk11-rhel8 1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9  About an hour ago
fuse7-karaf-openshift-rhel8         172.30.1.1:5000/openshift/fuse7/fuse-karaf-openshift-rhel8
1.0,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9  About an hour ago
fuse-apicurito-generator-rhel8      172.30.1.1:5000/openshift/fuse7/fuse-apicurito-generator-
rhel8 1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9  About an hour ago
apicurito-ui-rhel8                  172.30.1.1:5000/openshift/fuse7/apicurito-ui-rhel8
1.2,1.3,1.4,1.5,1.6,1.7,1.8, 1.9  About an hour ago
```

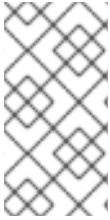
2. 各イメージストリームを1つずつ更新できるようになりました。

```
oc import-image -n openshift fuse7/fuse7-java-openshift-rhel8:1.9
oc import-image -n openshift fuse7/fuse7-java-openshift-jdk11-rhel8:1.9
oc import-image -n openshift fuse7/fuse7-karaf-openshift-rhel8:1.9
oc import-image -n openshift fuse7/fuse7-eap-openshift-jdk8-rhel7:1.9
oc import-image -n openshift fuse7/fuse7-eap-openshift-jdk11-rhel8:1.9
oc import-image -n openshift fuse7/fuse7-console-rhel8:1.9
oc import-image -n openshift fuse7/apicurito-ui-rhel8:1.9
oc import-image -n openshift fuse7/fuse-apicurito-generator-rhel8:1.9
```



注記

イメージストリームのバージョンタグの形式は **1.9-<BUILDNUMBER>** です。タグを **1.9** として指定すると、**1.9** ストリームの最新ビルドが取得されます。



注記

新しい Fuse on OpenShift イメージが利用できるようになったときに毎回再ビルドを自動的にトリガーするよう、Fuse アプリケーションを設定することもできます。詳細は、OpenShift Container Platform ドキュメントビルドの [ビルドのトリガーおよび変更](#) を参照してください。

15.3. FUSE ON OPENSIFT テンプレートのパッチ適用

適切なパッチが適応された依存関係を使用して新しいテンプレートベースのプロジェクトがビルドされるようにするため、Fuse on OpenShift テンプレートを最新のパッチレベルに更新する必要があります。

手順

1. Fuse on OpenShift テンプレートの更新には管理者権限が必要です。以下のように、OpenShift サーバーに管理者としてログインします。

```
oc login URL -u ADMIN_USER -p ADMIN_PASS
```

ここで、**URL** は OpenShift サーバーの URL に置き換え、**ADMIN_USER** および **ADMIN_PASS** には OpenShift サーバーの管理者アカウントのクレデンシャルを使用します。

2. パッチが適用された Fuse on OpenShift テンプレートをインストールします。コマンドプロンプトに以下のコマンドを入力します。

```
BASEURL=https://raw.githubusercontent.com/jboss-fuse/application-templates/application-templates-2.1.0.fuse-sb2-790047-redhat-00005
oc replace --force -n openshift -f ${BASEURL}/quickstarts/eap-camel-amq-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/eap-camel-cdi-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/eap-camel-cxf-jaxrs-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/eap-camel-cxf-jaxws-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf-camel-amq-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf-camel-log-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf-camel-rest-sql-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/karaf-cxf-rest-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-amq-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-config-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-drools-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-infinispan-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-camel-xml-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-cxf-jaxrs-template.json
oc replace --force -n openshift -f ${BASEURL}/quickstarts/spring-boot-cxf-jaxws-template.json
```



注記

BASEURL は、クイックスタートテンプレートを格納する Git リポジトリの GA ブランチを示し、常に最新のテンプレートが **HEAD** にあります。そのため、これらのコマンドを実行するたびに最新バージョンのテンプレートが取得されます。

15.4. BOM を使用したアプリケーション依存関係のパッチ適用

新しいスタイルの BOM を使用するよう、アプリケーションの **pom.xml** ファイルが設定されている場合は、本セクションの手順にしたがって Maven 依存関係をアップグレードします。

15.4.1. Spring Boot アプリケーションでの依存関係の更新

以下のコードは、Fuse on OpenShift における Spring Boot アプリケーションの POM ファイルの標準的なレイアウトを表しています。重要なプロパティ設定がいくつか含まれています。

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project ...>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

    <fuse.version>7.9.0.fuse-sb2-790065-redhat-00001</fuse.version>
    ...
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>fuse-springboot-bom</artifactId>
        <version>${fuse.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
  <build>
    ...
    <plugins>
      <!-- Core plugins -->
      ...
      <plugin>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        ...
        <version>${fuse.version}</version>
      </plugin>
    </plugins>
  </build>

  <profiles>
```

```

<profile>
  <id>openshift</id>
  <build>
    <plugins>
      <plugin>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>openshift-maven-plugin</artifactId>
        ...
        <version>${fuse.version}</version>
      </plugin>
    </plugins>
  </build>
</profile>
</profiles>
</project>

```

アプリケーションのパッチ適用やアップグレードでは、以下のバージョン設定が重要になります。

fuse.version

新しいスタイルの **fuse-springboot-bom** BOM のバージョンや、 **openshift-maven-plugin** プラグインおよび **spring-boot-maven-plugin** プラグインのバージョンを定義します。

15.4.2. Karaf アプリケーションの依存関係の更新

以下のコードは、Fuse on OpenShift における Karaf アプリケーションの POM ファイルの標準的なレイアウトを表しています。重要なプロパティ設定がいくつか含まれています。

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project ...>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <fuse.version>7.9.0.fuse-sb2-790065-redhat-00001</fuse.version>
    ...
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>fuse-karaf-bom</artifactId>
        <version>${fuse.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <groupId>org.jboss.redhat-fuse</groupId>

```

```

    <artifactId>karaf-maven-plugin</artifactId>
    <version>${fuse.version}</version>
    ...
  </plugin>
  ...
  <plugin>
    <groupId>org.jboss.redhat-fuse</groupId>
    <artifactId>openshift-maven-plugin</artifactId>
    <version>${fuse.version}</version>
    ...
  </plugin>
</plugins>
</build>

</project>

```

アプリケーションのパッチ適用やアップグレードでは、以下のバージョン設定が重要になります。

fuse.version

新しいスタイルの **fuse-karaf-bom** BOM のバージョンや、**openshift-maven-plugin** プラグインおよび **karaf-maven-plugin** プラグインのバージョンを定義します。

15.4.3. JBoss EAP アプリケーションでの依存関係の更新

以下のコードは、Fuse on OpenShift における JBoss EAP アプリケーションの POM ファイルの標準的なレイアウトを表しています。重要なプロパティ設定がいくつか含まれています。

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project ...>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <fuse.version>7.9.0.fuse-sb2-790065-redhat-00001</fuse.version>
    ...
  </properties>

  <!-- Dependency Management -->
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>fuse-eap-bom</artifactId>
        <version>${fuse.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>

```

アプリケーションのパッチ適用やアップグレードでは、以下のバージョン設定が重要になります。

fuse.version

fuse-eap-bom BOM ファイル (旧式の **wildfly-camel-bom** BOM ファイルの代わりとなる) のバージョンを定義します。BOM のバージョンを特定のパッチバージョンに更新すると、Fuse on JBoss EAP の Maven 依存関係がすべて更新されます。

15.5. 利用可能な BOM バージョン

以下の表は、異なるパッチリリースの Red Hat Fuse に対応する新しいスタイルの BOM バージョンを表しています。

表15.1 Red Hat Fuse リリースおよび対応する新しいスタイルの BOM バージョン

Red Hat Fuse リリース	org.jboss.redhat-fuse BOM バージョン
Red Hat Fuse 7.12 GA	7.9.0.fuse-sb2-790065-redhat-00001
Red Hat Fuse 7.0.1 patch	7.0.1.fuse-000008-redhat-4

アプリケーション POM を特定の Red Hat Fuse パッチリリースにアップグレードするには、**fuse.version** プロパティを対応する BOM バージョンに設定します。

付録A SPRING BOOT MAVEN プラグイン

Spring Boot Maven プラグインによって Maven で Spring Boot のサポートが提供され、実行可能な **jar** または **war** アーカイブをパッケージ化や、アプリケーション **in-place** の実行を可能にします。

A.1. SPRING BOOT MAVEN プラグインのゴール

Spring Boot Maven プラグインには以下のゴールが含まれます。

- **spring-boot:run** は Spring Boot アプリケーションを実行します。
- **spring-boot:repackage** は、**.jar** および **.war** ファイルを再パッケージして実行可能にします。
- **spring-boot:start** および **spring-boot:stop** の両方は、Spring Boot アプリケーションのライフサイクルを管理するために使用されます。
- **spring-boot:build-info** は、Actuator が使用できるビルド情報を生成します。

A.2. SPRING BOOT MAVEN プラグインの使用

Spring Boot プラグインの使用方法に関する一般的な手順は、<https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/#using> を参照してください。以下の例は Spring Boot の **spring-boot-maven-plugin** の使用方法を示しています。

- [Spring Boot 2 の例](#)

Spring Boot Maven プラグインの詳細は、<https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/> を参照してください。

A.2.1. Spring Boot 2 の Spring Boot Maven プラグインの使用

以下の例は Spring Boot 2 の **spring-boot-maven-plugin** の使用方法を示しています。

例

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.redhat.fuse</groupId>
  <artifactId>spring-boot-camel</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

    <!-- configure the Fuse version you want to use here -->
    <fuse.bom.version>7.9.0.fuse-sb2-790065-redhat-00001</fuse.bom.version>

    <!-- maven plugin versions -->
    <maven-compiler-plugin.version>3.7.0</maven-compiler-plugin.version>
    <maven-surefire-plugin.version>2.19.1</maven-surefire-plugin.version>
  </properties>
```

```
<build>
  <defaultGoal>spring-boot:run</defaultGoal>

  <plugins>
    <plugin>
      <groupId>org.jboss.redhat-fuse</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>${fuse.bom.version}</version>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

<repositories>
  <repository>
    <id>redhat-ga-repository</id>
    <url>https://maven.repository.redhat.com/ga</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>redhat-ea-repository</id>
    <url>https://maven.repository.redhat.com/earlyaccess/all</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>

<pluginRepositories>
  <pluginRepository>
    <id>redhat-ga-repository</id>
    <url>https://maven.repository.redhat.com/ga</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>redhat-ea-repository</id>
```

```
<url>https://maven.repository.redhat.com/earlyaccess/all</url>
<releases>
  <enabled>true</enabled>
</releases>
<snapshots>
  <enabled>false</enabled>
</snapshots>
</pluginRepository>
</pluginRepositories>
</project>
```

付録B KARAF MAVEN プラグインの使用

karaf-maven-plugin は、Karaf コンテナのマイクロサービススタイルパッケージである Karaf サーバーアセンブリの作成を可能にします。終了したアセンブリには、Karaf インストールに必要なすべてのコンポーネント (etc/、data/、lib、およびシステムディレクトリーの内容を含む) が含まれますが、アプリケーションの実行に最低限必要なコンポーネントのみに限定されます。

B.1. MAVEN 依存関係

karaf-assembly プロジェクトの Maven 依存関係は feature リポジトリ (**features** 分類子) または kar アーカイブになります。

- feature リポジトリは Maven 構造の system/internal リポジトリにインストールされます。
- kar アーカイブのコンテンツはサーバーの上で展開され、含まれた feature リポジトリはインストールされています。

Maven 依存関係の範囲

依存関係の Maven スコープは、features サービス設定ファイル **etc/org.apache.karaf.features.cfg** (featuresRepositories プロパティ以下) にその feature リポジトリがリストされているかどうかを判断します。これらのスコープは次のとおりです。

- compile (デフォルト): リポジトリ (または kar アーカイブ) のすべての機能が **startup.properties** にインストールされます。feature リポジトリは features サービス設定ファイルにリストされません。
- runtime: **karaf-maven-plugin** のブートステージ。
- Provided: **karaf-maven-plugin** のインストールステージ。

B.2. KARAF MAVEN プラグインの設定

karaf-maven-plugin は、Maven スコープに関連する 3 つのステージを定義します。プラグイン設定は、インストールされた feature リポジトリから機能を参照して、これらの要素を使用して機能をインストールする方法を制御します。

- スタートアップステージ: **etc/startup.properties**
このステージでは、**etc/startup.properties** に含まれるバンドルのリストを準備するためにスタートアップ機能、スタートアッププロファイル、およびスタートアップバンドルが使用されます。この結果、機能バンドルは適切な開始レベルで **etc/startup.properties** にリストされ、バンドルは **system** 内部リポジトリにコピーされます。**feature_name** または **feature_name/feature_version** 形式を使用できます (例: `<startupFeature>foo</startupFeature>`)。
- ブートステージ: **etc/org.apache.karaf.features.cfg**
このステージは、**featuresBoot** プロパティで利用できる機能と、**featuresRepositories** プロパティのリポジトリを管理します。この結果、機能名が features サービス設定の **boot-features** に追加され、機能のすべてのバンドルが **system** 内部リポジトリにコピーされます。**feature_name** または **feature_name/feature_version** 形式を使用できます (例: `<bootFeature>bar</bootFeature>`)。
- インストールステージ:
このステージは、アーティファクトを `${karaf.home}/${karaf.default.repository}` にインストールします。この結果、機能のバンドルすべてが **system** 内部リポジトリにインストール

されます。そのため、機能は起動時に外部リポジトリにアクセスしなくてもインストールされることがあります。**feature_name** または **feature_name/feature_version** 形式を使用できます (例: `<installedFeature>baz</installedFeature>`)。

- ライブラリー
プラグインは、ライブラリー URL を指定する 1 つ以上の `library` 子要素を持つ、`libraries` 要素を受け入れます。

例

```
<libraries>
  <library>mvn:org.postgresql/postgresql/9.3-1102-jdbc41;type:=endorsed</library>
</libraries>
```

B.3. カスタマイズされた KARAF アセンブリー

karaf-maven-plugin によって提供された **karaf:assembly** ゴールを使用するのが、推奨される Karaf サーバーアセンブリーの作成方法です。この方法では、プロジェクトの **pom.xml** ファイルの Maven 依存関係からサーバーをアセンブルします。**karaf-maven-plugin** 設定に指定されたバンドル (または機能) と、**pom.xml** の **<dependencies>** セクションに指定された依存関係の両方をカスタマイズされた Karaf アセンブリーに追加できます。

- **kar**
kar タイプの依存関係は、スタートアップ (`scope=compile`)、ブート (`scope=runtime`)、またはインストール (`scope=provided`) **kars** として **karaf-maven-plugin** に追加されます。**kar** は作業ディレクトリー (`target/assembly`) に展開され、機能 XML が検索され、追加の **feature** リポジトリとして使用されます (ステージは指定の **kar** のステージと同じです)。
- **features.xml**
features 分類子の依存関係は、**karaf-maven-plugin** でスタートアップ (`scope=compile`)、ブート (`scope=runtime`) またはインストール (`scope=provided`) リポジトリとして使用されます。**kar** で見つかった **feature** リポジトリを明示的に追加する必要はありません。
- **jar** および **bundle**
bundle または **jar** タイプの依存関係は、**karaf-maven-plugin** でスタートアップ (`scope=compile`)、ブート (`scope=runtime`)、またはインストール (`scope=provided`) バンドルとして使用されます。

B.3.1. karaf:assembly goal

karaf-maven-plugin によって提供される **karaf:assembly** ゴールを使用して Karaf サーバーアセンブリーを作成できます。このゴールは、プロジェクト POM の Maven 依存関係からマイクロサービススタイルサーバーアセンブリーをアセンブルします。Fuse on OpenShift プロジェクトでは、**karaf:assembly** ゴールを Maven インストールフェーズにバインドすることが推奨されます。プロジェクトはバンドルパッケージを使用し、**bootBundles** 要素内部にリストすることでプロジェクト自体が Karaf コンテナにインストールされます。



注記

Karaf フレームワーク機能などの必要な要素のみをスタートアップステージに含めるようにしてください。これは **etc/startup.properties** に追加され、このステージでは Karaf **features** サービスは完全に開始していないためです。他の要素はブートステージに移します。

例

以下の例は、クイックスタートの典型的な Maven 設定を表しています。

```
<plugin>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>karaf-maven-plugin</artifactId>
  <version>${fuse.version}</version>
  <extensions>true</extensions>
  <executions>
    <execution>
      <id>karaf-assembly</id>
      <goals>
        <goal>assembly</goal>
      </goals>
      <phase>install</phase>
    </execution>
  </executions>
  <configuration>

    <karafVersion>{karafMavenPluginVersion}</karafVersion>
    <useReferenceUrls>true</useReferenceUrls>
    <archiveTarGz>false</archiveTarGz>
    <includeBuildOutputDirectory>false</includeBuildOutputDirectory>
    <startupFeatures>
      <feature>karaf-framework</feature>
    </startupFeatures>
    <bootFeatures>
      <feature>shell</feature>
      <feature>jaas</feature>
      <feature>aries-blueprint</feature>
      <feature>camel-blueprint</feature>
      <feature>fabric8-karaf-blueprint</feature>
      <feature>fabric8-karaf-checks</feature>
    </bootFeatures>
    <bootBundles>
      <bundle>mvn:${project.groupId}/${project.artifactId}/${project.version}</bundle>
    </bootBundles>
  </configuration>
</plugin>
```

付録C OPENSIFT MAVEN プラグイン

OpenShift Maven プラグインは、OpenShift に Java アプリケーションをビルドおよびデプロイするために使用されます。Java アプリケーションを OpenShift に提供します。これは、maven への密なインテグレーション、およびすでに提供されているビルド設定からの利点を提供します。3つのタスクに焦点を当てています。

- S2I イメージのビルド
- OpenShift リソースの作成
- OpenShift へのアプリケーションのデプロイ

C.1. OPENSIFT MAVEN プラグインについて

OpenShift Maven プラグインには以下の機能があります。

- S2I イメージを扱うため、柔軟で強力な設定を継承します。
- 両方の OpenShift 記述子をサポートします。
- バイナリソースを使用した OpenShift Docker ビルド (Docker デーモンに対する直接イメージビルドの代替)
- 複数の設定スタイル:
 - 固有のデフォルトが事前設定される迅速な増加のためのゼロ設定。
 - XML 構文のプラグイン設定内のインライン設定。
 - プラグインによって強化される実際のデプロイメント記述子の外部設定テンプレート。
- 柔軟なカスタマイズ:
 - ジェネレーターは、Maven ビルドおよび特定システムに対して生成された自動 Docker イメージ設定 (spring-boot、プレーンな java、karaf) を分析します。
 - エンリッチャーは、SCM ラベルなどの追加情報で OpenShift リソース記述子を拡張し、サービスなどのデフォルトオブジェクトを追加できます。
 - ジェネレーターとエンリッチャーは個別に設定され、プロファイルに統合できます。

C.2. イメージのビルド

oc:build ゴールは、アプリケーションが含まれる Docker 形式イメージの作成に使用されます。これらは、Kubernetes または OpenShift に後でデプロイできます。このプラグインは **maven-assembly-plugin** のアセンブリー記述子を使用し、イメージに追加されるコンテンツを指定します。これらのイメージは、**oc:push** でパブリックまたはプライベート Docker レジストリーにプッシュできます。**oc:watch** ゴールにより、コードの変更に反応して、自動的にイメージを再作成したり、新しいアーティファクトを実行中のコンテナにコピーしたりできます。

C.3. KUBERNETES および OPENSIFT リソース

Kubernetes および OpenShift リソース記述子は、**oc:resource** で作成できます。これらのファイルは Maven アーティファクト内にパッケージされ、**oc:apply** を使用して稼働中のオーケストレーションプラットフォームにデプロイできます。

設定

4つのレベルの設定があります。

- Zero-Config モードは、**pom.xml** ファイルにあるデータを基にして、使用するベースイメージや公開するポートなど、非常に的確な決定を行えるようにします。これは開始点として使用され、クイックスタートアプリケーションを小さくきれいに維持するために使用されます。
- XML プラグイン設定モードは docker-maven-plugin が提供するモードと似ています。IDE サポートでタイプセーフな設定が可能ですが、可能なリソース記述子機能のサブセットのみが提供されます。
- Kubernetes and OpenShift リソースフラグメントは、プラグインで強化できるユーザー提供の YAML ファイルです。これにより、上級ユーザーはすべての機能が含まれるプレーンな設定ファイルを使用できますが、プロジェクト固有のビルド情報を追加したり、定型コードの使用を回避することができます。
- OpenShift クラスターで docker compose デプロイメントを立ち上げるために Docker Compose が使用されます。これには、OpenShift のデプロイメントプロセスに関する知識はほとんど必要ありません。

C.4. OPENSIFT MAVEN プラグインのインストール

このプラグインは Maven の中央リポジトリで使用でき、以下のようにインテグレーション前後のフェーズに接続することができます。デフォルトでは、Maven は org.apache.maven.plugins パッケージおよび org.codehaus.mojo パッケージ内のプラグインのみを検索します。JKube プラグインゴールのプロバイダーを解決するには、`~/.m2/settings.xml` ファイルを編集し、**org.eclipse.jkube** namespace を `<pluginGroups>` 設定に追加します。

手順

- OpenShift Maven プラグインをインテグレーション前およびインテグレーション後のフェーズに接続するには、以下の `~/.m2/settings.xml` ファイルを追加します。

```
<settings>
...

  <pluginGroups>
    <pluginGroup>org.jboss.redhat-fuse</pluginGroup>
  </pluginGroups>

  ...
</settings>

<plugin>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>openshift-maven-plugin</artifactId>
  <version>${fuse.version}</version>

  <configuration>
    ....
    <images>
```

```

<!-- A single's image configuration -->
<image>
  ...
  <build>
    ....
    </build>
  </image>
  ....
</images>
</configuration>

<!-- Connect oc:resource, oc:build and oc:helm to lifecycle phases -->
<executions>
  <execution>
    <id>jkube</id>
    <goals>
      <goal>resource</goal>
      <goal>build</goal>
      <goal>helm</goal>
    </goals>
  </execution>
</executions>
</plugin>

```

C.5. OPENSIFT MAVEN プラグインのビルドゴールについて

ビルドゴールは、Docker 形式イメージや S2I ビルドなどの Kubernetes および OpenShift ビルドアーティファクトを作成および管理するために使用されます。

表C.1 OpenShift Maven プラグインのビルドゴール

目的	説明
oc:resource	Kubernetes または OpenShift リソース記述子を作成します。生成されるリソースは target/classes/META-INF/jkube/openshift ディレクトリーにあります。
oc:build	ビルドイメージ。
oc:push	イメージをレジストリーにプッシュします。プッシュするレジストリーはデフォルトでは docker.io ですが、イメージ名の一部として指定できます。
oc:apply	リソースを実行中のクラスターに適用します。このゴールは oc:deploy と似ていますが、完全なデプロイメントサイクルを実行しません。

C.6. OPENSIFT MAVEN プラグインの開発ゴールについて

開発ゴールは、リソース記述子を開発クラスターにデプロイするために使用されます。また、開発クラスターのライフサイクルを管理するのにも便利です。

表C.2 OpenShift Maven プラグインの開発ゴール

目的	説明
oc:deploy	リソース記述子の作成後にクラスターへデプロイし、アプリケーションをビルドします。バックグラウンドで実行すること以外は oc:apply と同じです。
oc:undeploy	クラスターからリソース記述子をアンデプロイおよび削除します。
oc:log	実行中のアプリケーションのログを表示します。
oc:debug	リモートのデバッグを有効にします。
oc:watch	ファイルの変更の有無を監視し、再ビルドおよび再デプロイを実行します。

付録D FABRIC8 MAVEN プラグイン



注記

Fabric8 Maven プラグインは**非推奨**になりました。OpenShift Maven プラグインを使用してアプリケーションをビルドおよびデプロイします。

fabric8-maven-plugin を利用すると、Java アプリケーションを OpenShift にデプロイできます。これは Maven との緊密なインテグレーションを提供し、すでに提供されているビルド設定の利点を活用できます。このプラグインは以下のタスクを中心に行います。

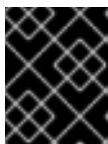
- Docker 形式イメージのビルド
- OpenShift リソース記述子の作成

大変柔軟に設定でき、以下を作成するために複数の設定モデルをサポートします。

- **Zero-Config** の設定: opinionated default (推奨されるデフォルト) で迅速な立ち上げが可能になります。または、より高度な要件を可能にします。
- **XML 設定**: **pom.xml** ファイルに追加できるその他の設定オプションを提供します。

D.1. イメージのビルド

fabric8:build ゴールは、アプリケーションが含まれる Docker 形式イメージの作成に使用されます。ビルドアーティファクトやイメージの依存関係を簡単に含めることができます。このプラグインは **maven-assembly-plugin** のアセンブリー記述子を使用し、イメージに追加されるコンテンツを指定します。



重要

Fuse on OpenShift では、OpenShift **s2i** ビルドストラテジーのみがサポートされます。**docker** ビルドストラテジーはサポートされません。

D.2. KUBERNETES および OPENSIFT リソース

Kubernetes および OpenShift リソース記述子は、**fabric8:resource** で作成できます。これらのファイルは Maven アーティファクト内にパッケージされ、**fabric8:apply** を使用して稼働中のオーケストレーションプラットフォームにデプロイできます。

設定

4つのレベルの設定があります。

- Zero-Config モードは、**pom.xml** ファイルにあるデータを基にして、使用するベースイメージや公開するポートなど、非常に的確な決定を行えるようにします。これは開始点として使用され、クイックスタートアプリケーションを小さくきれいに維持するために使用されます。
- XML プラグイン設定モードは docker-maven-plugin が提供するモードと似ています。IDE サポートでタイプセーフな設定が可能ですが、可能なリソース記述子機能のサブセットのみが提供されます。
- Kubernetes and OpenShift リソースフラグメントは、プラグインで強化できるユーザー提供の YAML ファイルです。これにより、上級ユーザーはすべての機能が含まれるプレーンな設定

ファイルを使用できますが、プロジェクト固有のビルド情報を追加したり、定型コードの使用を回避することができます。OpenShift クラスターで docker compose デプロイメントを立ち上げるために Docker Compose が使用されます。これには、OpenShift のデプロイメントプロセスに関する知識はほとんど必要ありません。設定に関する詳細は <https://maven.fabric8.io/#configuration> を参照してください。

D.3. プラグインのインストール

Fabric8 Maven プラグインは Maven の中央リポジトリで使用でき、以下のようにインテグレーション前後のフェーズに接続することができます。

手順

- Fabric8 Maven プラグインをインテグレーション前およびインテグレーション後のフェーズに接続するには、以下の **settings.xml** ファイルを追加します。

```
<plugin>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${fuse.version}</version>

  <configuration>
    ....
    <images>
      <!-- A single's image configuration -->
      <image>
        ...
        <build>
          ....
          </build>
        </image>
      ....
    </images>
  </configuration>

  <!-- Connect fabric8:resource and fabric8:build to lifecycle phases -->
  <executions>
    <execution>
      <id>fabric8</id>
      <goals>
        <goal>resource</goal>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

D.4. FABRIC8 MAVEN プラグインのゴールの理解

Fabric8 Maven プラグインは、スムーズな Java 開発環境を提供するゴールのセットをサポートします。ゴールは以下のように分類できます。

- **ビルドゴール** は、Docker 形式イメージや S2I ビルドなどの Kubernetes および OpenShift ビルドアーティファクトを作成および管理するために使用されます。

- **開発ゴール** は、リソース記述子を開発クラスターにデプロイするために使用されます。また、開発クラスターのライフサイクルを管理するのにも便利です。

D.4.1. ビルドおよび開発ゴールの理解

Red Hat Fabric Integration Services 製品の Fabric8 Maven プラグインによってサポートされるゴールは次のとおりです。

表D.1 ビルドゴール

ゴール	説明
<code>fabric8:build</code>	ビルドイメージ。Fuse on OpenShift では、OpenShift s2i ビルドストラテジーのみがサポートされることに注意してください。 docker ビルドストラテジーはサポートされません。
<code>fabric8:resource</code>	Kubernetes または OpenShift リソース記述子を作成します。
<code>fabric8:apply</code>	リソースを実行中のクラスターに適用します。
<code>fabric8:resource-apply</code>	fabric8:resource → fabric8:apply を実行します。

表D.2 開発ゴール

ゴール	説明
<code>fabric8:deploy</code>	リソース記述子の作成後にクラスターへデプロイし、アプリケーションをビルドします。バックグラウンドで実行すること以外は fabric8:run と同じです。
<code>fabric8:undeploy</code>	クラスターからリソース記述子をアンデプロイおよび削除します。
<code>fabric8:start</code>	以前デプロイされたアプリケーションを開始します。
<code>fabric8:stop</code>	以前デプロイされたアプリケーションを停止します。
<code>fabric8:log</code>	実行中のアプリケーションのログを表示します。
<code>fabric8:debug</code>	リモートのデバッグを有効にします。
<code>fabric8:watch</code>	プロジェクトワークスペースで変更を監視し、アプリケーションの再デプロイメントを自動的にトリガーします。

D.4.2. 環境変数の設定

以下のように XML 設定に `env` パラメーターを追加して、1つ以上の環境変数を設定できます。例を以下に示します。

例

```

<configuration>
  <resources>
    <env>
      <JAVA_OPTIONS>-Dmy.custom=option</JAVA_OPTIONS>
      <MY_VAR>value</MY_VAR>
    </env>
  </resources>
</configuration>

```

D.4.3. リソース検証設定

fabric8:resource ゴールは、Kubernetes および OpenShift の API 仕様を使用して、生成されたリソース記述子を検証します。

表D.3 リソース検証設定

設定	説明	デフォルト
fabric8.skipResourceValidation	値が true に設定されている場合、リソース検証はスキップされます。これは、何らかの理由でリソース検証に失敗してもデプロイメントを継続したい場合に便利です。	false
fabric8.failOnValidationError	値が true に設定されている場合、検証エラーによってプラグインの実行がブロックされます。true に設定されていない場合は警告が表示されます。	false
fabric8.build.switchToDeployment	値が true に設定され、OpenShift で ImageStreams を使用していない場合、fabric8-maven-plugin は DeploymentConfig ではなく Deployments に切り替わります。	false
fabric8.openshift.trimImageInContainerSpec	値が true に設定されている場合、コンテナイメージ参照が "" に設定されます。これは、後続のロールアウトによって ImagePullErr が発生する OpenShift 3.7 での不適切な動作を処理するためのものです。	false

Fabric8 Maven プラグインゴールの詳細は <https://maven.fabric8.io/#goals> を参照してください。

D.5. ジェネレーター

Fabric8 Maven プラグインは、特定の種類のアプリケーションに対して自動的にイメージをビルドする機能がある、**generator** コンポーネントを提供します。Fuse on OpenShift では、以下のジェネレータータイプがサポートされます。

- 「Spring Boot」
- 「Karaf」

ジェネレーターフレームワークは、アプリケーションプロジェクトの特性に応じて、必要なビルドタイプを自動検出し、適切なジェネレーターコンポーネントを呼び出します。



注記

Fabric8 Maven プラグインのオープンソースコミュニティバージョンは、追加のジェネレータータイプを提供しますが、これらのタイプは Fuse on OpenShift 製品ではサポートされません。

D.5.1. ゼロ設定

ジェネレーターには設定が**必要ありません**。ジェネレーターはデフォルトで有効になり、Fabric8 Maven プラグインが呼び出されると自動的にデフォルト設定で実行されます。しかし、必要な場合はジェネレーターの設定を簡単にカスタマイズできます。

D.5.2. ベースイメージを指定するモード

Fuse on OpenShift では、アプリケーションビルドのベースイメージは Java イメージ (Spring Boot アプリケーションの場合) または Karaf イメージ (Karaf アプリケーションの場合) のいずれかになります。Fabric8 Maven プラグインは、ベースイメージの指定に以下のモードをサポートします。

istag

(デフォルト): **image stream** は、OpenShift イメージストリームからタグ付けされたイメージを選択して動作します。この場合、ベースイメージは以下の形式で指定されます。

```
<namespace>/<image-stream-name>:<tag>
```

<namespace> は、イメージストリームが定義される OpenShift プロジェクトの名前に置き換ええます (通常は **openshift**)。 **<image-stream-name>** はイメージストリームの名前に置き換ええます。 **<tag>** はストリームの特定のイメージを識別します (またはストリームの**最新**イメージを追跡します)。

docker

docker モードは、イメージレジストリーから特定の Docker 形式イメージを選択して動作します。ベースイメージは直接リモートレジストリーから取得されるため、イメージストリームは必要ありません。この場合、ベースイメージは以下の形式で指定されます。

```
[<registry-location-url>/]<image-namespace>/<image-name>:<tag>
```

イメージ指定子は **任意**で リモートイメージレジストリーの URL である **<registry-location-url>** で始まり、イメージ namespace **<image-namespace>**、イメージ名 **<image-name>**、およびタグ **<tag>** が続きます。



注記

オープンソースコミュニティバージョンの **openshift-maven-plugin** のデフォルト操作は Red Hat の製品化バージョンとは異なります。たとえば、コミュニティバージョンのデフォルトモードは **docker** です。

D.5.2.1. istag モードのデフォルト値

デフォルトである **istag** モードが選択されている場合、Fabric8 Maven プラグインは以下のデフォルトイメージ指定子を使用して Fuse イメージを選択します (形式は **<namespace>/<image-stream-name>:<tag>** です)。


```
fuse7/fuse-eap-openshift:1.9
fuse7/fuse-java-openshift:1.9
fuse7/fuse-karaf-openshift:1.9
```



注記

Fuse イメージストリームでは、個別のイメージに **1.0-1** や **1.0-2** などのビルド番号がタグ付けされます。常に最新のイメージを追跡するよう、**1.0** タグが設定されます。

D.5.2.2. docker モードのデフォルト値

docker モードが選択され、**registry.redhat.io** にアクセスするよう OpenShift 環境が設定されていることを仮定する場合、Fabric8 Maven プラグインは以下のデフォルトイメージ指定子を使用して Fuse イメージを選択します (形式は **<image-namespace>/<image-name>:<tag>** です)。

```
fuse7/fuse-eap-openshift:1.9
fuse7/fuse-java-openshift:1.9
fuse7/fuse-karaf-openshift:1.9
```

D.5.2.3. Spring Boot アプリケーションのモード設定

Spring Boot アプリケーションのビルドに使用されるモード設定とベースイメージの場所をカスタマイズするには、以下の形式で **configuration** 要素をアプリケーションの **pom.xml** ファイルにある **fabric8-maven-plugin** 設定に追加します。

例

```
<configuration>
  <generator>
    <config>
      <spring-boot>
        <fromMode>{istag|docker}</fromMode>
        <from>{image locations}</from>
      </spring-boot>
    </config>
  </generator>
</configuration>
```

D.5.2.4. Karaf アプリケーションのモード設定

Karaf アプリケーションのビルドに使用されるモード設定とベースイメージの場所をカスタマイズするには、以下の形式で **configuration** 要素をアプリケーションの **pom.xml** ファイルにある **fabric8-maven-plugin** 設定に追加します。

例

```
<configuration>
  <generator>
    <config>
      <karaf>
        <fromMode>{istag|docker}</fromMode>
        <from>{image locations}</from>
      </karaf>
    </config>
  </generator>
</configuration>
```

```

</config>
</generator>
</configuration>

```

D.5.2.5. コマンドラインを使用したジェネレーターモードの指定

pom.xml ファイルで直接モードの設定をカスタマイズする代わりに、以下のプロパティ設定をコマンドライン呼び出しに追加し、モードの設定を直接 **mvn** コマンドに渡すことができます。

例

```

//build from Docker-formatted image directly, registry location, image name or tag are subject to
change if desirable
-Dfabric8.generator.fromMode=docker
-Dfabric8.generator.from=<custom-registry-location-url>/<image-namespace>/<image-name>:<tag>

//to use ImageStream from different namespace
-Dfabric8.generator.fromMode=istag //istag is default
-Dfabric8.generator.from=<namespace>/<image-stream-name>:<tag>

```

D.5.3. Spring Boot

Spring Boot ジェネレーターは、**pom.xml** ファイルで **spring-boot-maven-plugin** を見つけるとアクティベートされます。生成されたコンテナポートは **application.properties** ファイルの **server.port** プロパティのから読み取られ、見つからない場合はデフォルトで **8080** になります。

このジェネレーターは一般的なジェネレーターオプションの他に、以下のオプションで設定することができます。

表D.4 Spring Boot 設定オプション

要素	説明	デフォルト
assemblyRef	アセンブリーへの参照が指定されている場合は、含めるアーティファクトの検出を行わずに使用されます。	
targetDir	検出されたアーティファクトが配置される生成されたイメージ内のディレクトリ。ベースイメージも変更された場合のみ変更します。	/deployments
jolokiaPort	ベースイメージによって公開される Jolokia エージェントのポート。Jolokia ポートを公開したくない場合はこれを 0 に設定します。	8778
mainClass	呼び出すメインクラス。指定のない場合は次のようにジェネレーターがメインクラスを検索します。最初に、fat-jar を検出するためにチェックが実行されます。次に、 main メソッドを持つ単一のクラスを探すために target/classes ディレクトリがスキャンされます。何も見つからなかった場合や複数のクラスが見つかった場合は、ジェネレーターは何もしません。	
webPort	サービスとして公開するポート。Web アプリケーションのポートであるはずで。ポートを公開しない場合はこれを 0 に設定します。	8080

要素	説明	デフォルト
color	設定されている場合、Spring Boot のコンソール出力で色を強制的に使用します。	

ジェネレーターは、**application.properties** から読み取られた管理またはサーバーポートのいずれかを示す Kubernetes の liveness および readiness probe を追加します。**server.ssl.key-store** プロパティが **application.properties** に設定されている場合、probe は **https** を使用するように自動的に設定されます。

D.5.4. Karaf

Karaf ジェネレーターは、**pom.xml** ファイルで **karaf-maven-plugin** プラグインを見つけるとアクティベートされます。このジェネレーターは、一般的なジェネレーターオプションに加え、以下のオプションで設定が可能です。

表D.5 Karaf 設定オプション

要素	説明	デフォルト
baseDir	検出されたアーティファクトが配置される生成されたイメージ内のディレクトリ。ベースイメージも変更された場合のみ変更します。	/deployments
jolokiaPort	ベースイメージによって公開される Jolokia エージェントのポート。Jolokia ポートを公開したくない場合はこれを 0 に設定します。	8778
mainClass	呼び出すメインクラス。指定のない場合は次のようにジェネレーターがメインクラスを検索します。最初に、fat-jar を検出するためにチェックが実行されます。次に、 main メソッドを持つ単一のクラスを探すために target/classes ディレクトリーがスキャンされます。何も見つからなかった場合や複数のクラスが見つかった場合は、ジェネレーターは何もしません。	
user	ファイルを追加するユーザーやグループ。ベースイメージにユーザーがすでに存在している必要があります。通常の形式は <user>[:<group>[:<run-user>]] です。ユーザーおよびグループは数値のユーザー id およびグループ id、または名前として指定することができます。グループ id は任意です。	jboss:jboss:jboss
webPort	サービスとして公開するポート。Web アプリケーションのポートであるはずで。ポートを公開しない場合はこれを 0 に設定します。	8080

付録E FABRIC8 CAMEL MAVEN プラグイン

fabric8-camel-maven プラグインを使用して、ソースコードからすべての Camel エンドポイントを検証できます。これにより、Camel アプリケーションまたはユニットテストを実行する前にエンドポイントが有効であることを確認できます。

E.1. FABRIC8 CAMEL MAVEN プラグインのゴール

ソースコードで Camel エンドポイントを検証するには、以下を使用します。

- **fabric8-camel:validate**: このゴールは、Maven プロジェクトのソースコードを検証し、無効な Camel エンドポイント URI を到底します。

E.2. FABRIC8-CAMEL-MAVEN プラグインのプロジェクトへの追加

fabric8-camel-maven プラグインをプロジェクトに追加するには、fabric8-camel-maven プラグインをプロジェクトの **pom.xml** ファイルに追加します。

手順

1. プラグインを有効にするには、以下を **pom.xml** ファイルに追加します。

```
<plugin>
  <groupId>io.fabric8.forge</groupId>
  <artifactId>fabric8-camel-maven-plugin</artifactId>
  <version>2.3.90</version>
</plugin>
```

注記: fabric8-forge リリースの現在のバージョン番号を確認してください。最新のリリースは <https://github.com/fabric8io/fabric8-forge/releases> にあります。

2. その後、ゴールの検証はコマンドラインまたは IDEA や Eclipse などの Java エディターから実行できます。

```
mvn fabric8-camel:validate
```

プラグインの自動実行

プラグインを有効にしてビルドの一部として自動的に実行し、エラーを検出することも可能です。以下の例では、プラグインが実行されるタイミングがフェーズによって決定されます。この例では、メインのソースコードの完了後に実行される **process-classes** がフェーズです。

例

```
<plugin>
  <groupId>io.fabric8.forge</groupId>
  <artifactId>fabric8-camel-maven-plugin</artifactId>
  <version>2.3.80</version>
  <executions>
    <execution>
      <phase>process-classes</phase>
    </execution>
  </executions>
  <goals>
    <goal>validate</goal>
  </goals>
```

```

    </execution>
  </executions>
</plugin>

```

テストソースコードの検証

Maven プラグインを設定して、テストソースコードを検証することもできます。以下のように **process-test-classes** のとおりにフェーズを変更します。

例

```

<plugin>
  <groupId>io.fabric8.forge</groupId>
  <artifactId>fabric8-camel-maven-plugin</artifactId>
  <version>2.3.80</version>
  <executions>
    <execution>
      <configuration>
        <includeTest>true</includeTest>
      </configuration>
      <phase>process-test-classes</phase>
      <goals>
        <goal>validate</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

E.3. 任意の MAVEN プロジェクトでのゴール実行

プラグインを **pom.xml** ファイルに追加せずに Maven プロジェクトでゴールの検証を実行することもできます。完全修飾名を使用してプラグインを指定する必要があります。

手順

- Apache Camel から **camel-example-cdi** プラグインでゴールを実行する場合は、以下のコマンドを実行します。

```

$cd camel-example-cdi
$mvn io.fabric8.forge:fabric8-camel-maven-plugin:2.3.80:validate

```

これにより、以下の出力が表示されます。

```

[INFO] -----
[INFO] Building Camel :: Example :: CDI 2.16.2
[INFO] -----
[INFO] --- fabric8-camel-maven-plugin:2.3.80:validate (default-cli) @ camel-example-cdi ---
[INFO] Endpoint validation success: (4 = passed, 0 = invalid, 0 = incapable, 0 = unknown
components)
[INFO] Simple validation success: (0 = passed, 0 = invalid)
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----

```

検証に成功した後、4つのエンドポイントを検証できます。次の例は、Camel エンドポイントを検証し、必要な場合は修正する方法を表しています。

例

たとえば、ソースコードに以下のような Camel エンドポイント URI があるとします。

- 1. Camel エンドポイント URI を以下のように修正します。

```
@Uri("timer:foo?period=5000")
```

- 2. **period** オプションに誤りが含まれるように、以下のような変更を加えます。

```
@Uri("timer:foo?perid=5000")
```

- 3. 再度、検証ゴールを実行します。

```
[INFO] -----
[INFO] Building Camel :: Example :: CDI 2.16.2
[INFO] -----
[INFO]
[INFO] --- fabric8-camel-maven-plugin:2.3.80:validate (default-cli) @ camel-example-cdi ---
[WARNING] Endpoint validation error at:
org.apache.camel.example.cdi.MyRoutes(MyRoutes.java:32)

timer:foo?perid=5000

        perid   Unknown option. Did you mean: [period]

[WARNING] Endpoint validation error: (3 = passed, 1 = invalid, 0 = incapable, 0 = unknown
components)
[INFO] Simple validation success: (0 = passed, 0 = invalid)
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
```

上記のように、Camel エンドポイント URI のエラーが表示されます。

E.4. オプション

Maven プラグインは以下のオプションをサポートします。これらのオプションはコマンドラインから設定するか (-D 構文を使用)、<configuration> タグの **pom.xml** ファイルで定義します。

表E.1 Fabric8 Camel Maven プラグインのオプション

パラメーター	デフォルト値	説明
downloadVersion	true	インターネットからの Camel カタログバージョンのダウンロードを許可するかどうか。プロジェクトが使用する Camel バージョンがこのプラグインがデフォルトで使用する Camel バージョンと異なる場合に必要です。

パラメーター	デフォルト値	説明
failOnError	false	無効な Camel バージョンが見つかった場合に失敗するかどうか。デフォルトでは、 WARN レベルでエラーがログに記録されます。
logUnparseable	false	解析不可能なため検証できないエンドポイント URI をログに記録するかどうか。
includeJava	true	無効な Camel エンドポイントについて検証される Java ファイルを含めるかどうか。
includeXML	true	無効な Camel エンドポイントについて検証される XML ファイルを含めるかどうか。
includeTest	false	テストソースコードを含めるかどうか。
includes	-	Java および xml ファイルの名前を絞り込み、指定されたパターンのリスト (ワイルドカードおよび正規表現) と一致するファイルのみが含まれるようにします。複数の値はコンマで区切ることができます。
excludes	-	Java および xml ファイルの名前を絞り込み、指定されたパターンのリスト (ワイルドカードおよび正規表現) と一致するファイルが除外されるようにします。複数の値はコンマで区切ることができます。
ignoreUnknownComponent	true	不明なコンポーネントを無視するかどうか。
ignoreIncapable	true	エンドポイント URI の解析が不可能なことを無視するかどうか。
ignoreLenientProperties	true	lenient プロパティを使用するコンポーネントを無視するかどうか。true の場合、URI の検証はより厳密になりますが、lenient プロパティを使用するため、URI にあってもコンポーネントの一部でないプロパティでは失敗することがあります。HTTP コンポーネントを使用してエンドポイント URI でクエリーパラメーターを提供する場合がこの例になります。
showAll	false	エンドポイントと簡単な式 (無効と有効の両方) をすべて表示するかどうか。

E.5. インクルードテストの検証

Maven プロジェクトがある場合、プラグインを実行して単体テストのソースコードでエンドポイントを検証することもできます。

手順

- 以下のように **-D** スタイルを使用してオプションを渡すことができます。

```
$cd myproject  
$mvn io.fabric8.forge:fabric8-camel-maven-plugin:2.3.80:validate -DincludeTest=true
```


付録F JVM 環境変数のカスタマイズ

JVM 環境変数を使用して、Fuse on OpenShift イメージのオプションをすべて設定できます。

F.1. OPENJDK 8 での S2I JAVA ビルダーイメージの使用

S2I Java ビルダーイメージを使用すると、他のアプリケーションサーバーを使用せずに直接結果を実行できます。この S2I イメージは、フラットなクラスパス (**fat jars** を含む) を持つマイクロサービスに適しています。

Fuse on OpenShift イメージの使用時に Java オプションを設定できます。JVM オプションでは、**JAVA_OPTIONS** 環境変数を使用できます。また、アプリケーションに渡される引数には **JAVA_ARGS** を提供します。

F.2. OPENJDK 8 での S2I KARAF ビルダーイメージの使用

Karaf4 カスタムアセンブリーベースの Maven プロジェクトをビルドするには、S2I Karaf ビルダーイメージを OpenShift の Source To Image ワークフローと使用します。

手順

- 以下のコマンドを使用して、S2I ワークフローを使用します。

```
s2i build <git repo url> registry.redhat.io/fuse7/fuse-karaf-openshift:1.6 <target image name>
docker run <target image name>
```

F.2.1. Karaf4 アセンブリーの設定

Maven プロジェクトによってビルドされた Karaf4 アセンブリーの場所を提供する方法は複数あります。

- 出力ディレクトリーのデフォルトのアセンブリーファイル ***.tar.gz**。
- sti または oc コマンドで **-e flag** を使用。
- プロジェクトソース下の **.sti/environment** にある **FUSE_ASSEMBLY** プロパティを設定。

F.2.2. Maven ビルドのカスタマイズ

Maven のビルドをカスタマイズすることが可能です。**MAVEN_ARGS** 環境変数を設定して、動作を変更できます。デフォルトでは、**MAVEN_ARGS** は以下のように設定されます。

```
`Karaf4: install karaf:assembly karaf:archive -DskipTests -e`
```

F.3. ビルド時の環境変数

ビルド中に、S2I Java および Karaf ビルダーイメージの動作に影響を与える環境変数を以下に示します。

- **MAVEN_ARGS**: Maven の呼び出し時に使用する引数で、デフォルトのパッケージを置き換えます。

- **MAVEN_ARGS_APPEND**: 追加の Maven 引数で、**-X** や **-am -pl** などの一時的な引数の追加に便利です。
- **ARTIFACT_DIR**: マルチモジュールのビルドのために Jar ファイルが作成された **target/** へのパス。これらは **\${MAVEN_ARGS}** に追加されます。
- **ARTIFACT_COPY_ARGS**: アーティファクトを出力ディレクトリーからアプリケーションディレクトリーにコピーするときに使用する引数。イメージの一部になるアーティファクトを指定するのに便利です。
- **MAVEN_CLEAR_REPO**: 設定すると、アーティファクトのビルド後に Maven リポジトリーが削除されます。これはアプリケーションイメージを小さく維持するのに便利ですが、インクリメンタルビルドを使用できません。デフォルト値は **false** です。

F.4. ランタイムの環境変数

以下の環境変数を使用して、実行スクリプトに影響を与えることができます。

- **JAVA_APP_DIR**: アプリケーションがあるディレクトリー。アプリケーションのすべてのパスはこのディレクトリーを基準にした相対パスになります。
- **JAVA_LIB_DIR**: このディレクトリーには、Java jar ファイルと、クラスパスが保持されるオプションのクラスパスファイルが含まれます。単一行のクラスパス (コロン区切り) または行ごとにリストされた jar ファイルのいずれかになります。設定のない場合は **JAVA_LIB_DIR** は **JAVA_APP_DIR** ディレクトリーと同じになります。
- **JAVA_OPTIONS**: Java の呼び出し時に追加するオプション。
- **JAVA_MAX_MEM_RATIO**: **JAVA_OPTIONS** に **-Xmx** オプションがない場合に使用されます。これは、コンテナの制限をベースにしてデフォルトの最大ヒープメモリーを算出するために使用されます。このオプションをコンテナのメモリー制限がない Docker コンテナで使用しても、何も影響はありません。
- **JAVA_MAX_CORE**: ガベージコレクタースレッドの数など、特定のデフォルトを算出するために使用される利用可能なコアの数を手動で制限します。0 に設定すると、コアの数を基にしてベース JVM のチューニングを行うことができません。
- **JAVA_DIAGNOSTICS**: これを設定して、一部の診断情報を取得し、標準出力に送ります。
- **JAVA_MAIN_CLASS**: Java の引数として使用するメインクラス。この環境変数を使用すると **\$JAVA_APP_DIR** ディレクトリーのすべての jar ファイルがクラスパスと **\$JAVA_LIB_DIR** ディレクトリーに追加されます。
- **JAVA_APP_JAR**: **java -jar** で開始できるようにするための適切なマニフェストを持つ jar ファイル。ただし、指定がない場合は **\$JAVA_MAIN_CLASS** が設定されます。すべての場合で jar ファイルはクラスパスに追加されます。
- **JAVA_APP_NAME**: プロセスに使用する名前。
- **JAVA_CLASSPATH**: 使用するクラスパス。指定の無い場合は起動スクリプトが **\${JAVA_APP_DIR}/classpath** ファイルを確認し、その内容をクラスパスとして使用します。このファイルが存在しない場合は、アプリケーションディレクトリーのすべての jar が (**classes:\${JAVA_APP_DIR}/***) 以下に追加されます。
- **JAVA_DEBUG**: 設定するとリモートでのデバッグが有効になります。

- **JAVA_DEBUG_PORT**: リモートでのデバッグに使用されるポート。デフォルト値は 5005 です。

F.5. JOLOKIA の設定

以下の環境を Jolokia で使用できます。

- **AB_JOLOKIA_OFF**: 設定した場合、Jolokia のアクティベートを無効にします (空の値をエコーします)。Jolokia はデフォルトで有効になっています。
- **AB_JOLOKIA_CONFIG**: 設定した場合、ファイル (パスを含む) を Jolokia JVM エージェントプロパティとして使用します。設定のない場合は、設定を使用して `/opt/jolokia/etc/jolokia.properties` が作成されます。
- **AB_JOLOKIA_HOST**: バインドするホストアドレス (デフォルト値は 0.0.0.0)。
- **AB_JOLOKIA_PORT**: 使用するポート (デフォルト値は 8778)。
- **AB_JOLOKIA_USER**: Basic 認証のユーザー。デフォルトでは **jolokia** になります。
- **AB_JOLOKIA_PASSWORD**: Basic 認証のパスワード。デフォルトでは認証は無効になっています。
- **AB_JOLOKIA_PASSWORD_RANDOM**: 値を生成し、`/opt/jolokia/etc/jolokia.pw` ファイルに書き込みます。
- **AB_JOLOKIA_HTTPS**: HTTPS でセキュアな通信を有効にします。デフォルトでは、`serverCert` 設定が **AB_JOLOKIA_OPTS** に指定されていないと、自己署名サーバー証明書が生成されます。
- **AB_JOLOKIA_ID**: 使用するエージェント ID。
- **AB_JOLOKIA_DISCOVERY_ENABLED**: Jolokia の検出を有効にします。デフォルト値は `false` です。
- **AB_JOLOKIA_OPTS**: エージェント設定に追加される追加のオプション。オプションは `key=value` の形式で指定されます。

以下は、さまざまな環境とのインテグレーションにおけるオプションになります。以下は、さまざまな環境とのインテグレーションにおけるオプションになります。

- **AB_JOLOKIA_AUTH_OPENSIFT**: OpenShift TSL 通信のクライアント認証を有効にします。このパラメーターの値がクライアント証明書に存在するようにしてください。このパラメーターを有効にすると、Jolokia が自動的に **HTTPS** 通信モードになります。デフォルトの CA 証明書は `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt` に設定されます。

JAVA_ARGS 変数を対応する値に設定すると、アプリケーションの引数を指定できます。

付録G JVM のチューニングによる LINUX コンテナ内での実行

[JVM エルゴノミクス](#) による、ガベージコレクター、ヒープサイズ、およびランタイムコンパイラーのデフォルト値の設定を許可すると、Linux コンテナ内部で実行している Java プロセスは想定どおりに動作しません。`java -jar mypplication-fat.jar` などのチューニングパラメーターを使用せずに Java アプリケーションを実行すると、JVM はコンテナの制限ではなく、ホストの制限を基にして複数のパラメーターを自動的に設定します。

本セクションでは、コンテナの制限を考慮してデフォルト値が算出されるよう、Linux コンテナ内部で Java アプリケーションをパッケージ化するための情報を提供します。

G.1. JVM のチューニング

現在の Java JVM はコンテナ対応でないため、コンテナのサイズではなく、物理ホストのサイズを基にしてリソースを割り当てます。たとえば、通常 JVM は**最大ヒープサイズ**を、ホスト上の物理メモリーの 1/4 に設定します。大型のホストマシンでは、この値はコンテナに定義されたメモリー制限を簡単に越えてしまいます。実行時にコンテナ制限を越えてしまうと、OpenShift はアプリケーションを強制終了します。

この問題に対応するには、Java JVM が制限されたコンテナ内で実行されることを認識でき、かつ最大ヒープサイズが手動で調整されない場合は自動的に調整されることを認識できる、Fuse on OpenShift ベースのイメージを使用します。これにより、アプリケーションを実行する JVM の最大メモリー制限とコア制限を設定できます。Fuse on OpenShift イメージは以下を行うことができます。

- コンテナのコアを基にした `C1CompilerCount` の設定。
- コンテナメモリー制限が 300MB 未満の場合に C2 JIT コンパイラーを無効にする。
- コンテナメモリー制限が 300MB 未満の場合に、コンテナメモリー制限の 1/4 をデフォルトのヒープサイズに使用する。

G.2. FUSE ON OPENSIFT イメージのデフォルト動作

Fuse on OpenShift では、アプリケーションビルドのベースイメージは Java イメージ (Spring Boot アプリケーションの場合) または Karaf イメージ (Karaf アプリケーションの場合) のいずれかになります。Fuse on OpenShift イメージはコンテナ制限を読み取るスクリプトを実行し、その制限をリソース割り当てのベースとして使用します。デフォルトでは、スクリプトは以下のリソースを JVM に割り当てます。

- コンテナメモリー制限の 50%
- コンテナコア制限の 50%

これには一部の例外があります。Karaf と Java イメージでは、物理メモリーが 300MB のしきい値未満になると、ヒープサイズはデフォルトの半分ではなく、1/4 に回復されます。

G.3. FUSE ON OPENSIFT イメージのカスタムチューニング

スクリプトは、内部リソースをチューニングするためにカスタムアプリケーションによって読み取られる `CONTAINER_MAX_MEMORY` および `CONTAINER_CORE_LIMIT` 環境変数を設定します。さらに、アプリケーションを実行する JVM の設定をカスタマイズ可能にする以下のランタイム環境変数を指定できます。

- `JAVA_OPTIONS`

- **JAVA_MAX_MEM_RATIO**

制限を明示的にカスタマイズするには、Maven プロジェクトで **deployment.yml** ファイルを編集し、**JAVA_MAX_MEM_RATIO** 環境変数を設定します。

例

```
spec:
  template:
    spec:
      containers:
      -
        resources:
          requests:
            cpu: "0.2"
            memory: 256Mi
          limits:
            cpu: "1.0"
            memory: 256Mi
        env:
        - name: JAVA_MAX_MEM_RATIO
          value: 60
```

G.4. サードパーティーライブラリーのチューニング

Red Hat は、Jetty などのサードパーティー Java ライブラリーの制限をカスタマイズすることを推奨します。このようなライブラリーは、制限を手作業でカスタマイズしないと、指定のデフォルト制限を使用します。起動スクリプトは、アプリケーションが使用できるコンテナ制限を記述する一部の環境変数を公開します。

CONTAINER_CORE_LIMIT

算出されたコア制限

CONTAINER_MAX_MEMORY

コンテナに指定されたメモリー制限