Open Liberty 2021

Open Libertyランタイムガイド

Open Liberty を使用したクラウドネイティブアプリケーションのビルドおよびデプロイ

Last Updated: 2021-04-02

Open Liberty 2021 Open Libertyランタイムガイド

Open Liberty を使用したクラウドネイティブアプリケーションのビルドおよびデプロイ

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

概要

これらのトピックでは、Open Liberty の概要と、詳細なドキュメントリソースへのリンクを紹介します。

目次

第1章 OPEN LIBERTY とは

Open Liberty は、クラウドネイティブアプリケーションとマイクロサービスを構築する軽量の Java ランタイムです。

Open Liberty を使用すると、Jakarta EE および Eclipse MicroProfile からモジュール機能を簡単に追加および削除できます。このモジュラー構造により、マイクロサービスの開発が簡素化され、アプリケーションが必要とする機能をサポートするのに十分なアプリケーションサーバーを実行できます。また、Open Liberty zero 移行アーキテクチャーを使用すると、現在のアプリケーションと設定に最低限の影響を最小限に抑えて最新バージョンにアップグレードできます。Open Liberty は Jakarta EE 8 Full Platform および Web Profile 仕様と MicroProfile 3.0 と互換性があります。詳細は、Open Liberty の Web サイト を参照してください。Open Liberty 機能と機能に関する最新情報は、Open Liberty ブログ と Open Liberty の ドキュメント を参照してください。

Open Liberty は、OpenShift で利用可能な Java ランタイムの 1 つで、サポートは Red Hat サブスクリプションの一部として提供されます。Open Liberty を OpenShift で実行し、OpenShift プラットフォームの利点でクラウドネイティブアプリケーションをビルドおよびデプロイします。OpenShift での Open Liberty の使用に関する詳細は、「 OpenShift への マイクロサービスのデプロイ」を参照してください。

1.1. OPEN LIBERTY VERSIONING

従来のバージョン管理は、major.minor.micro スキームのさまざまな形に従います。この場合、重要な新機能はメジャーリリースでのみ配信されます。これらのメジャーリリースには、主要な新機能が含まれますが、アプリケーションの移行と、採用に重要なリグレッションテストを必要とする動作の変更も行います。したがって、一度に複数のメジャーバージョンがサポートされます。Open Liberty のモジュラー機能のアーキテクチャーは、ゼロ移行と組み合わせることで、major.minor.micro バージョン管理スキームに従わずに、段階的に新機能の配信が可能になります。

メジャーリリースではなく、Open Liberty の各リリースはマイクロまたはパッチリリースとみなされます。これらのパッチリリースは、yy.0.0.nn バージョンスキームに従います。最初の 2 桁はリリースの年数を示し、最後の 2 桁の数字はこの年内のリリース数を示します。最初の数値セットは毎年変わりますが、リリースは同等です。たとえば、20.0.0.1（20.0.0.1 の最初のリリース）と 19.0.0.12（20.0.0.12 の最後のリリース）の相違点は 19.0.0.10 と 19.0.0.11 の相違点と同じです。

メジャーリリースストリームがないサーバーランタイムにはほとんどありませんが、デスクトップおよびモバイルアプリケーションには一般的です。一部の公開システムでは、ソフトウェアのメジャーバージョンがあることが想定されます。その結果、メジャーバージョンが必要な場合には、公開年がスタンドアロンとして使用されます。たとえば、2019 で本ガイドに公開されている Open Liberty ドキュメントは、2019 年目でバージョン番号として使用されています。ただし、このドキュメントは 2019 年にリリースされるため、2020 のリリースの対象となります。

第2章 移行中のアーキテクチャー

ゼロ移行アーキテクチャーは、Open Liberty の中核となる設計原則で、ランタイムバージョン間の完全な互換性をサポートします。移行中のアーキテクチャーでは、現在のアプリケーションや設定への影響を最小限に抑えるために、最新バージョンの Open Liberty に移行できます。

ランタイムサーバーと連携するチームの主要な課題の 1 つは、ランタイムの最新リリースに継続的に更新する必要があります。これらの更新は、障害を引き起こす可能性のあるセキュリティー脆弱性を解決するために必要になることが多くあります。新しいリリースによって API や動作の変更も導入されるため、ランタイムバージョンを更新することは困難になる可能性があります。Open Liberty ランタイムおよび Open Liberty 機能の両方が、番号付きのバージョンでリリースされます。動作および API サポートの変更は新機能で提供されます。これを使用して、ニーズに合わせて採用するかどうかを決定できます。移行がゼロの場合、既存の API および動作は新規のランタイムバージョンでサポートされ、新しい API および動作が新機能および機能バージョンに追加されます。既存の変更されていない設定およびアプリケーションファイルは、アプリケーションの動作に予期しない変更なしに、更新されたバージョンの Open Liberty で動作します。

2.1. ユーザー設定ファイル

設定された機能がすべてインストールされていると、1 つの設定ファイルのセットが、変更なしに複数のバージョンの Open Liberty で動作します。ユーザー設定ファイルには、ランタイムが新規バージョンに更新されると変更する必要はありません。Open Liberty は、ランタイムのアクティブなバージョンに適用しない設定を無視します。

2.2. OPEN LIBERTY 機能

Open Liberty 機能はサーバー設定にエンコードされ、アップグレード中は変更されません。Java 仕様の動作変更を規定するなど、動作の変更が発生すると、その変更は新しいバージョンの機能に導入されます。既存のアプリケーションやそれらの設定は、古い機能バージョンで引き続き使用できますが、新しいアプリケーションは新しい動作で新しい機能バージョンを使用できます。

たとえば、Open Liberty は Servlet 3.1 および Servlet 4.0 の両方の仕様をサポートします。Servlet 4.0 仕様には、アプリケーションの動作の変更を可能にする以前のサーブレットバージョンへの明確化が含まれます。Open Liberty は、このような動作の変更を **servlet-4.0** 機能に制限し、**servlet-3.1** 機能の既存の動作を維持します。アプリケーションがサーブレット 3.1 仕様に設定されている場合は、他の **サーブレット仕様レベルをサポートするものに関係なく、ランタイム更新全体で servlet-3.1** 機能を使用できます。代わりに **servlet-4.0** 機能を設定しない限り、アプリケーションを変更する必要はありません。詳細は、「機能の 概要 」を参照してください。

2.2.1. 新機能の使用に関する考慮事項

すでに使用中の新しいバージョンの機能にアップグレードすると、既存のアプリケーションに影響する可能性があります。たとえば、現在 **servlet-3.1 機能を使用し、servlet-** 4.0 機能を使用する場合は、既存のサーブレットアプリケーションに servlet **4.0** 機能で正しく動作させる必要がある場合があり **ます。**つまり、元のバージョンで設定されたサーバーにアプリケーションを維持し、新しいバージョンで異なるサーバー設定を作成することができます。

一部の機能は他の機能と密接に対話し、両方が同じサーバーに設定されている場合に特定の機能バージョンを有効にする必要があります。一部の機能には、Java SDK などの前提条件ソフトウェアの特定バージョンが必要になります。

2.3. ゼロへの移行の例外

Open Liberty はランタイムバージョン間で変更されないように設計されていますが、まれなケースで使用できなくなることがあります。このような例外は、以下のカテゴリーのいずれかに分類されます。

- セキュリティー修正
  アプリケーションに、既存の動作のコンテキストで安全に実装できないセキュリティー関連の修正が必要な場合は、アプリケーションまたは設定を変更する必要がある場合があります。

- サードパーティー API の要件
  Open Liberty はサードパーティークラスローダー設定から API を制御しません。そのため、サードパーティーコンポーネントへの更新は、以前のバージョンのランタイムとの互換性が保証されていません。

- ドキュメント化されていない設定プロパティー
  Open Liberty の一部の設定オプションはランタイムドキュメントの一部ではありません。これらの設定オプションは、ソースコード、または外部の情報ソースをもとに、試行およびエラーを使用して判断できます。オプションが Open Liberty の一部として文書化されていない場合は、Open Liberty の一部とは見なされません。このようなオプションは完全に実装されていない可能性があり、使用されていると問題が発生する可能性があります。ドキュメントは記載されていないため、いつでも削除または変更でき、ゼロ移行では対応していません。

- Java の変更
  互換性のない新しい Java SE バージョンでは、Java 言語に互換性のない変更がいくつか加えられました。変更が新たな Java バージョンに含まれているまれなケースでは、Open Liberty はこれらの変更の影響を最小限に抑えることを試みます。しかし、これらの試行が常に成功しない可能性があり、Java の変更に悪影響を与える可能性があります。

移行中のアーキテクチャーは、既存の設定およびアプリケーションファイルの移行を必要とせずに開発者や企業時間および教育を保存します。開発者はランタイム更新を管理するのではなく、アプリケーションに集中できますが、既存のサーバー設定のパフォーマンスおよび管理性を向上できます。

第3章 OPENSHIFT でのデプロイメント

OpenShift は、Kubernetes ベースのコンテナーアプリケーションプラットフォームで、アプリケーションをビルド、コンテナー化、デプロイして、高可用性でスケーラブルな状態にすることができます。Open Liberty Operator を使用して、クラウドでのアプリケーションのデプロイおよび管理を単純化することもできます。

アプリケーションを開発およびコンテナー化した後に、コンテナーはデータベース、セキュリティーシステム、またはその他のマイクロサービスを実行している他のコンテナーと通信する必要があります。また、コンテナーはサービスが必要となり、スケーリングも必要になります。OpenShift は、開発者およびオペレーションチームのニーズに合わせてコンテナーを自動化および管理する機能を提供します。

3.1. OPENSHIFT でのアプリケーションの実行

OpenShift でアプリケーションを実行する利点の 1 つは、クラウドホスト型インフラストラクチャーにサービス(IaaS)ソリューションとして、または現在のオンプレミス構造としてデプロイできることです。OpenShift CLI または OpenShift Do CLI を使用して、アプリケーションを開発できます。次に、アプリケーションを Open Liberty コンテナーにコンテナー化し、それらを OpenShift クラスターにデプロイします。

マイクロサービスの OpenShift へのデプロイに関する段階的なチュートリアルについては、『OpenShift へのマイクロサービスのデプロイ』を参照してください。Open Liberty 上のアプリケーションを異なる OpenShift デプロイメントオプションと共にどのように使用できるかについての詳細は、OpenShift の ドキュメントを参照してください。

3.2. OPEN LIBERTY OPERATOR

Operator は、Kubernetes または OpenShift が提供する初期自動化以外のタスクを自動化できるようにカスタマイズされる Kubernetes の拡張 機能です。Open Liberty Operator には機能レベルが 5 つあります。これは、自動スケーリング、サービスバインディング、OpenShift 証明書管理統合、および Kubernetes Application Navigator(kAppNav)統合など、エンタープライズレベルの機能レベルが高いことを意味 します。

Open Liberty Operator を使用すると、水平の自動スケーリングを設定してアプリケーションを高可用性にすることができます。これにより、リソースの可用性や消費に基づいてアプリケーションインスタンスの作成や削除が可能になります。Operator はアプリケーションのデプロイメントを管理するのに役立ちます。たとえば、新しいバージョンのアプリケーションの新規コンテナータグをアップロードした後に、新規コンテナータグで Operator デプロイメントファイルの **applicationImage** フィールドを更新します。次に、Operator はローリングベースでアプリケーションを更新します。

また、この Operator は、永続ストレージの単純な設定や、単一署名(SSO)を外部プロバイダーに委譲する機能など、他の実稼働環境グレードの機能を 提供します。Operator はアプリケーション間のバインディング情報の更新を自動化するため、アプリケーションは接続し、特定のアプリケーションがサービスを生成または消費するかどうかについての情報を維持します。

Kubernetes または OpenShift で使用するために 、OperatorHub から Open Liberty Operator をインストール できます。Operator は、OpenShift Container Platform(OCP)から Red Hat で認定された Operator としても利用 できます。

3.3. も併せて参照してください。

Guide: マイクロサービスの Kubernetes へのデプロイ

第4章 機能の概要

機能は、特定のサーバーに読み込まれるランタイム環境の一部を制御できる機能単位です。サーバー設定から機能を追加または削除することで、サーバーが実行できる機能を制御できます。機能は、アプリケーションが必要とするプログラミングモデルおよびサービスを提供します。サーバー設定ファイルで任意の機能を指定できます。一部の機能には、それらに他の機能が含まれ、同じ機能が 1 つ以上の機能に含まれることがあります。

サーバーが起動すると、JVM は起動し、コントロールが Open Liberty カーネルに渡されます。「サーバー設定の 概要」の説明どおりに、設定 が読み込まれます。設定が解析されると、機能マネージャーが制御を取得し、featureManager 設定を処理 し、サーバーに要求された機能をロードして必要なコンポーネントを起動します。最後に、アプリケーションが起動します。設定が変更されると、機能マネージャーはサーバーを再起動せずに必要に応じてランタイムの一部を起動および停止することで、新たに要求された機能に必要なコードを再評価します。アプリケーションへの変更は同様の方法で処理されます。

4.1. 機能の使用

機能は、**server.xml** ファイルおよびその他の含まれるファイルであるシステム設定ファイルに指定されます。機能マネージャーは、**server.xml** ファイルの featureManager **要素** を使用して設定されます。必要な各機能は、**feature** 要素を使用して設定されます。以下の例では、**servlet-4.0** および **jdbc-4.3** 機能を設定します。

```
<server>
  <featureManager>
    <feature>servlet-4.0</feature>
    <feature>jdbc-4.3</feature>
  </featureManager>
</server>
```

ランタイムにはデフォルトの設定が含まれるため、指定が必要な設定が最小限に保たれます。**server.xml ファイルのデフォルト設定に追加または上書きする機能と共に、必要な機能を指定します。**サーバー設定の詳細は、「サーバー設定の 概要」を参照 してください。

4.2. ZERO-MIGRATION

Open Liberty zero-migration アーキテクチャーでは、現在のアプリケーションと設定に最低限の影響を最小限に抑えて、最新バージョンの Open Liberty に移行できます。移行可能なアーキテクチャーとは、動作で予期せずに、変更できない既存の Open Liberty ランタイム環境で、更新されたバージョンの Open Liberty ランタイム環境を使用する、既存の変更されていない設定およびアプリケーションファイルを使用できることを意味します。

Open Liberty ランタイム環境でプラグ可能な機能を使用することで、既存の API および動作が新バージョンの製品バージョンでサポートされ、新しい API および動作が新機能に追加されます。たとえば、Servlet 3.1 および 4.0 の両方の仕様がサポートされます。API の動作の変更は新機能バージョンでのみ実行されるため、アプリケーションに適した機能バージョンを選択できます。これらのバージョン管理機能は、引き続き Open Liberty の更新で引き続きサポートされます。

同じ機能バージョンを引き続き使用する場合は、アプリケーションを移行する必要はありません。たとえば、アプリケーションが Servlet 3.1 を使用する場合は、アプリケーションを実行する Open Liberty サーバーに **servlet-3.1** 機能が必要になります。サポートされるサーブレット仕様のレベルに関係なく、Open Liberty を更新し、**servlet-3.1** 機能の使用を継続できます。代わりに **servlet-4.0** 機能の使用を選択している場合に限り、アプリケーションを移行する必要があります。

4.3. 機能の統合

異なるバージョンの機能を持つようにサーバーを設定しようとすると、Open Liberty が同じ機能の組み合わせをサポートしないため、エラーが報告されます。つまり、ほとんどの Open Liberty 機能はシングルトン機能です。シングルトン機能は、サーバーで使用できるバージョンを 1 つだけ設定できる機能です。

異なるバージョンのシングルトン機能を必要とするアプリケーションがある場合、それらを異なるサーバーにデプロイする必要があります。サーバー設定で、**server.xml** ファイルの直接設定、または機能依存関係を介して複数のバージョンのシングルトン機能が含まれる場合、その設定はエラーとなり、その機能のバージョンは読み込まれません。この問題を解決するには、設定された機能をすべて指定または許容する（シングルトン機能の同じバージョン）ことを確認します。両方の機能バージョンにハード要件がある場合は、一部のアプリケーションを別のサーバーに移行する必要があります。

Java EE 7 および Java EE 8 仕様がコンポーネント仕様バージョンを共有している場合を除き、liberty は Java EE 7 と Java EE 8 の両方の機能を組み合わせることをサポートしません。サーバー設定に Java EE 7 および Java EE 8 機能を組み合わせると、サーバーは起動時にエラーを報告します。

以下の機能は、Java EE 7 と Java EE 8 の両方に含まれています。

- appClientSupport-1.0
- batch-1.0
- concurrent-1.0
- ejb-3.2
- j2eeManagement-1.1
- jacc-1.5
- jaxws-2.2
- jca-1.7
- jcaInboundSecurity-1.0
- jdbc-4.2
- jdbc-4.3
- jms-2.0
- wasJmsClient-2.0
- wasJmsSecurity-1.0
- wasJmsServer-1.0

Java EE 7 をサポートする機能の完全リストは、javaee-7.0 機能を参照してください。Java EE 8 をサポートする機能の完全リストは、javaee-8.0 機能を参照してください。

4.4. 代替機能

機能が置き換えられると、新機能または機能の組み合わせが、置き換えられた機能に重点が置かれる場合があります。新機能または機能は、置き換えられた機能の機能を完全に置き換えることができない可

能性があるため、設定を変更する前にシナリオを検討する必要があります。置き換えられた機能は、設定での使用はサポートされ、有効のままになりますが、より新しい機能を使用して設定を改善することができる可能性があります。

他の機能が含まれる機能は、これらすべてを含まない新しいバージョンの機能に置き換えられる可能性があります。新しいバージョンに含まれていない機能は、分離されているとみなされます。アプリケーションが、分離された機能の機能に依存する場合は、その設定に個別の機能を明示的に追加する必要があります。

以下の表には、代わりに使用される Open Liberty 機能が記載されています。

代替機能	代替機能	依存機能の削除
appSecurity-1.0	**appSecurity-2.0**	ldapRegistry および **servlet-3.0** 機能は appSecurity- **2.0** 機能の定義から削除されました。
jmsMdb-3.2	**jms-2.0** および **mdb-3.2**	**jms-2.0 および mdb- 3.2 機能は、jmsMdb-3.2** 機能と同じ **機能** を提供します。
ssl-1.0	**transportSecurity-1.0**	**ssl-1.0 および transportSecurity -1.0** 機能は機能的に同等です。ただし、**ssl-1.0** は、非セキュアなネットワークプロトコルが使用されているため、transportSecurity **-1.0 を** 置き換えることを意味します。

第5章 サーバー設定の概要

Open Liberty サーバー設定は、1つの必須ファイル、**server.xml** ファイル、およびオプションのファイルセットで構成されます。**server.xml** ファイルは適切に形式の XML であり、ルート要素が **サーバー** である必要があります。**server.xml** ファイルが処理されると、認識されない要素または属性は無視されます。

この **server.xml** ファイル例では、以下の操作を行うようにサーバーを設定します。

```
<server description="new server">
   <featureManager>
      <feature>jsp-2.3</feature> (1)
   </featureManager>
   <httpEndpoint id="defaultHttpEndpoint"
                 httpPort="9080" (2)
                 httpsPort="9443" />
   <applicationManager autoExpand="true" /> (3)
</server>
```

(1) JavaServer Pages 2.3 機能のサポート

(2) ポート **9080**の **localhost** への受信トラフィックをリッスンする

(3) デプロイ時に WAR ファイルを自動的に拡張

サーバー設定 という用語は、サーバー設定を構成するすべてのファイル、または XML ファイルの設定を参照するために使用できます。コンテキストが明確でない場合、**サーバーの XML 設定** という用語を使用して XML ファイルの設定を参照します。

5.1. 設定ファイル

サーバー設定ファイルは、以下の順序で処理されます。

1. **server.env**: 環境変数がこのファイルに指定されます。
2. **jvm.options**: JVM オプションはこのファイルに設定されます。
3. **bootstrap.properties**: このファイルは Open Liberty サーバーの起動に影響します。
4. **server.xml**: この必須ファイルは、サーバー設定と機能を指定します。

5.1.1. server.env

server.env ファイルはオプションです。このファイルは **bin/server** シェルスクリプトにより読み取り、主に **bin/server スクリプト** の動作に影響を与える環境変数を指定します。**server.env** ファイルは、以下の場所から読み取ります。

1. **${wlp.install.dir}/etc/**
2. **${wlp.user.dir}/shared/**
3. **${server.config.dir}/**

同じプロパティーが複数の場所に設定されている場合、最後に見つかった値が使用されます。

これらのファイルの最も一般的な使用方法は、以下の環境変数を設定することです。

- **JAVA_HOME**: 使用する JVM を示します。これが設定されていない場合は、システムのデフォルトが使用されます。
- **WLP_USER_DIR**: サーバー設定が含まれる **usr** ディレクトリーの場所を示します。これは、他の場所は **usr** ディレクトリーと相対的であるため、**etc/server.env** ファイルでのみ設定できます。
- **WLP_OUTPUT_DIR**: サーバーがファイルを書き込む場所を示します。デフォルトでは、サーバーは、設定が読み取るディレクトリー構造に書き込みます。ただし、一部の安全なプロファイルでは、サーバー設定を読み取り専用にする必要があるため、サーバーが別の場所にファイルを書き込む必要があります。

以下の例のように、**server.env** ファイルは **KEY=value** 形式です。

```
JAVA_HOME=/opt/ibm/java
WLP_USER_DIR=/opt/wlp-usr
```

キーの値にはスペースを含めることはできません。値は文字通り解釈されるため、スペースなどの特殊文字をエスケープする必要はありません。このファイルは、変数の置換をサポートしません。

5.1.2. jvm.options

jvm.options ファイルはオプションです。このファイルは、**bin/server** シェルスクリプトにより読み取り、JVM が Open Liberty で JVM の起動時に使用するオプションを決定します。**jvm.options** ファイルは以下の場所から読み取るのではなく、以下の場所から読み込まれます。

1. **${wlp.user.dir}/shared/jvm.options**
2. **${server.config.dir}/configDropins/defaults/**
3. **${server.config.dir}/**
4. **${server.config.dir}/configDropins/overrides/**

jvm.options ファイルがこれらの場所に存在しない場合は、サーバースクリプトが **${wlp.install.dir}/etc** にあるファイルを探します（そのようなディレクトリーが存在する場合）。

jvm.options ファイルの一般的な用途は次のとおりです。

- JVM メモリー制限の設定
- 製品の監視により提供される Java エージェントの有効化
- Java システムプロパティーの設定

jvm.options ファイル形式は、以下の例のように JVM オプションごとに 1 行を使用します。

```
-Xmx512m
-Dmy.system.prop=This is the value.
```

空白文字などの特殊文字をエスケープする必要はありません。オプションは JVM に読み取り、提供されます。複数のオプションを指定すると、それらはすべて JVM によって表示されます。このファイルは、変数の置換をサポートしません。

5.1.3. bootstrap.properties

bootstrap.properties ファイルはオプションです。

このファイルは Open Liberty ブートストラップ中に読み取り、サーバー起動の初期段階の設定を提供します。server **.xml ファイルよりも前のサーバー** が読み取り、起動から Open Liberty カーネルの起動および動作に影響を与える可能性があります。**bootstrap.properties** ファイルは単純な Java プロパティーファイルで、**${server.config.dir}** にあります。**bootstrap.properties ファイルの一般的な使用方法は、server. xml** ファイルを読み取る前にロギング動作に影響を与える可能性があるためです。

bootstrap.properties ファイルは特殊なオプションプロパティー **bootstrap.include** をサポートします。これは、ブートストラップステージでも読み取る別のプロパティーファイルを指定します。たとえば、この boostrap **.include** ファイルには、複数のサーバーが使用するブートストラッププロパティーの共通のセットを含めることができます。**bootstrap.include** ファイルを絶対パスまたは相対パスに設定します。

5.1.4. server.xml

最も重要な設定ファイルは **server.xml** ファイルです。**server.xml** ファイルは適切に形式の XML であり、ルート要素が **サーバー** である必要があります。サーバーがサポートしている正確な要素は、設定される機能に応じて異なり、未知の設定は無視されます。

Open Liberty は例外によって設定の原則を使用します。これにより、succinct 設定ファイルが可能になります。ランタイム環境は、ビルトイン設定のセットから動作します。デフォルト設定を上書きする設定のみを指定します。

サーバー設定ファイルは、以下の場所から読み取り、順番に読み込まれます。

1. **${server.config.dir}/configDropins/defaults/**
2. **${server.config.dir}/server.xml**
3. **${server.config.dir}/configDropins/overrides/**

$ **{server.config.dir}/server.xml ファイルが存在する必要があり** ますが、他のファイルはオプションです。

サーバー形式の XML ファイルをディレクトリーにドロップすると、設定を柔軟に作成できます。ファイルは、2 つの configDropins ディレクトリーごとにアルファベット順で読み取ら **れ** ます。

5.2. 変数の置換

変数を使用してサーバー設定をパラメーター化できます。値への変数参照を解決するには、以下のソースを順番に参照します。

1. **server.xml** ファイルのデフォルト変数値
2. 環境変数
3. **bootstrap.properties**
4. Java システムプロパティー
5. **server.xml** ファイルの設定
6. コマンドラインで宣言された変数

変数は **${variableName}** 構文を使用して参照されます。

以下の例のようにサーバー設定に変数を指定します。

```
<variable name="variableName" value="some.value" />
```

server.xml ファイルに指定されるデフォルト値は、他の値が指定されていない場合のみ使用されます。

```
<variable name="variableName" defaultValue="some.default.value" />
```

コマンドラインから、起動時に変数を指定することもできます。これを実行する場合、コマンドラインで指定された変数は、変数の他のソースをすべて上書きし、サーバーの起動後に変更することはできません。

環境変数には変数としてアクセスできます。バージョン 19.0.0.3 では、環境変数名を直接参照できます。変数が指定どおりに解決できない場合、**server.xml** ファイルは環境変数名に以下のバリエーションを検索します。

- 英数字以外の文字をすべてアンダースコア(_)に置き換えます。
- すべての文字を大文字に変更します。

たとえば、**server.xml** ファイルに **${my.env.var}** を入力すると、以下のような名前で環境変数が検索されます。

1. my.env.var
2. my_env_var
3. MY_ENV_VAR

バージョン 19.0.0.3 以前では、以下の例のように、環境変数名の先頭に **env.** を追加して環境変数にアクセスできます。

```
<httpEndpoint id="defaultHttpEndpoint"
              host="${env.HOST}"
              httpPort="9080" />
```

変数の値は、常に単純な型変換を持つ文字列として解釈されます。したがって、ポートのリスト（80、443 など）は、2 つのポート番号ではなく単一文字列として解釈される可能性があります。以下の例のように **list** 関数を使用して 、 変数の置換を強制的に分割できます。

```
<mongo ports="${list(mongoPorts)}" hosts="${list(mongoHosts)}" />
```

Simple 演算は、整数値を持つ変数でサポートされます。演算子の左側と右辺は、変数または数字のいずれかになります。以下の例のように、演算子は **+**、**-**、*、または / のいずれかになります。

```
<variable name="one" value="1" />
<variable name="two" value="${one+1}" />
<variable name="three" value="${one+two}" />
<variable name="six" value="${two*three}" />
<variable name="five" value="${six-one}" />
<variable name="threeagain" value="${six/two}" />
```

事前定義された変数には多数あります。

- **wlp.install.dir**: Open Liberty ランタイムがインストールされているディレクトリー。
- **wlp.server.name**: サーバー名。
- **wlp.user.dir**: **usr** フォルダーのディレクトリー。デフォルトは **${wlp.install.dir}/usr** です。
- **shared.app.dir**: 共有アプリケーションのディレクトリー。デフォルトは **${wlp.user.dir}/shared/apps** です。
- **shared.config.dir**: 共有設定ファイルのディレクトリー。デフォルトは **${wlp.user.dir}/shared/config** です。
- **shared.resource.dir**: 共有リソースファイルのディレクトリー。デフォルトは **${wlp.user.dir}/shared/resources** です。
- **server.config.dir**: サーバー設定が保存されるディレクトリー。デフォルトは **${wlp.user.dir}/servers/${wlp.server.name}** です。
- **server.output.dir**: サーバーが workarea、ログ、およびその他のランタイム生成ファイルを書き込むディレクトリー。デフォルトは **${server.config.dir}** です。

5.3. 設定のマージ

この設定では複数のファイルで構成されるため、2 つのファイルが同じ設定を提供している可能性があります。このため、単純なルールセットに従ってサーバー設定がマージされます。Open Liberty では、設定はシングルトンとファクトリー設定に分けられ、それぞれがマージするための独自のルールを持ちます。単一要素の設定にはシングルトン設定（ロギングなど）が使用されます。ファクトリー設定は、アプリケーション全体やデータソースなどの複数のエンティティーを設定するために使用されます。

5.3.1. シングルトン設定のマージ

複数回指定されるシングルトン設定要素の場合、設定はマージされます。異なる属性を持つ 2 つの要素が存在する場合は、両方の属性が使用されます。以下に例を示します。

```
<server>
   <logging a="true" />
   <logging b="false" />
</server>
```

は以下のように処理されます。

```
<server>
   <logging a="true" b="false" />
</server>
```

同じ属性を 2 回指定すると、最後のインスタンスが優先されます。以下に例を示します。

```
<server>
   <logging a="true" b="true" />
   <logging b="false" />
</server>
```

は以下のように処理されます。

```
<server>
   <logging a="true" b="false" />
</server>
```

設定は、テキストを取得する子要素を使用して提供することが可能です。

このような場合、設定は指定されたすべての値を使用してマージされます。最も一般的なシナリオは、機能を設定することです。以下に例を示します。

```
<server>
   <featureManager>
      <feature>servlet-4.0</feature>
   </featureManager>
   <featureManager>
      <feature>restConnector-2.0</feature>
   </featureManager>
</server>
```

は以下のように処理されます。

```
<server>
   <featureManager>
      <feature>servlet-4.0</feature>
      <feature>restConnector-2.0</feature>
   </featureManager>
</server>
```

5.3.2. ファクトリー設定をマージします。

ファクトリー設定のマージは、要素名が一致するため、自動的にマージされないことを除き、シングルトン設定と同じルールを使用します。ファクトリー設定では、同じ要素の設定に有効です。つまり、2つの異なる論理オブジェクトになります。そのため、各要素は個別のオブジェクトを設定することが想定されます。1つの論理オブジェクトが2つの要素で設定されている場合、そのオブジェクトが同じであることを示すために各要素に **id** 属性を設定する必要があります。**id** 属性の変数置換はサポートされていません。

以下の例では、2つのアプリケーションを設定します。最初のアプリケーションは **myapp.war** で、myawesomeapp のコンテキストルートを持ち **ます**。他のアプリケーションは **myapp2.war** で、コンテキストルートは **myapp2** になります。

```
<server>
   <webApplication id="app1" location="myapp.war" />
   <webApplication location="myapp2.war" />
   <webApplication id="app1" contextRoot="/myawesomeapp" />
</server>
```

5.4. 処理を含める

デフォルトの場所に加え、**include** 要素を使用して追加の設定ファイルを取得できます。サーバー設定ファイルに別のファイルへの包含参照が含まれる場合、サーバーは、include **要素** の代わりにインラインに含まれているかのように、参照されるファイルの内容を処理します。

以下の例では、サーバーは **other 2.xml ファイルの内容を処理する前に other.xml** ファイルの内容を処理します。

```
<server>
   <include location="other.xml" />
   <include location="other2.xml" />
</server>
```

デフォルトでは、包含ファイルが存在する必要があります。include ファイルが存在しない場合には、以下の例のように **オプション** の属性を **true** に設定します。

```
<server>
   <include location="other.xml" optional="true" />
</server>
```

ファイルを含めると、onConflict **属性** を指定して通常のマージルールを変更できます。onConflict 属性の値を **IGNORE** または **REPLACE の** いずれかに設定することができます。

```
<server>
   <include location="other.xml" onConflict="IGNORE" />
   <include location="other2.xml" onConflict="REPLACE" />
</server>
```

location 属性は、相対パスまたは絶対パスまたは HTTP URL に設定できます。

5.5. 設定リファレンス

Open Liberty での設定のほとんどは自己完結していますが、JDBC ドライバー設定が複数のデータソースで共有される場合など、設定を共有すると便利です。**server** 要素の直接子として定義されるファクトリー設定要素を参照できます。

設定への参照は、参照される要素の **id** 属性を常に使用します。参照を行う設定要素は、以下の例のように、常に **Ref** で終わる属性を使用します。

```
<server>
  <dataSource jndiName="jdbc/fred" jdbcDriverRef="myDriver" />
  <jdbcDriver id="myDriver" />
</server>
```

5.6. 動的更新

サーバーは、サーバー XML 設定を更新して更新を監視し、変更の検出時に動的にリロードします。XML 以外のファイル（**server.env、bootstrap. properties、および jvm.options** ）への変更は起動時にのみ読み取られるため、動的ではありません。ローカルディスクのサーバー XML 設定ファイルは、500ms ごとに更新について監視されます。XML 設定ファイルの監視頻度を設定できます。たとえば、10 分ごとに監視するようにサーバーを設定するには、以下を指定します。

```
<config monitorInterval="10m" />
```

ファイルシステムのポーリングを無効にし、MBean が通知される場合にのみ再読み込みするには、以下を指定します。

```
<config updateTrigger="mbean" />
```

5.7. ログメッセージ

サーバーが実行されると、設定を参照するログメッセージが出力される可能性があります。ログの参照は、XPath のような構造を使用して設定要素を指定します。要素名は、角括弧内の **id** 属性の値で指定されます。サーバー設定に **ID** が指定されていない場合、**ID** は自動的に生成されます。以下のサーバー XML 設定の例に基づいて、**dataStore** 要素と子 **dataSource** はログで dataStore[myDS **] および dataStore[** myDS **]/dataSource[default-0]** としてログで識別されます。

```
<server>
   <dataStore id="myDS">
      <dataSource />
   </dataStore>
</server>
```

第6章 OPEN LIBERTY OPERATOR

Operator は、Kubernetes または OpenShift が提供する初期自動化以外のタスクを自動化できるようにカスタマイズされる Kubernetes の拡張機能です。Open Liberty Operator は、Kubernetes ベースのクラスターでアプリケーションをデプロイし、管理できるようにします。

6.1. OPEN LIBERTY OPERATOR とは

Open Liberty Operator でアプリケーションをデプロイする場合、Operator は Open Liberty リソースを監視し、リソースの現在の状態を、設定したリソースの状態と比較します。リソースの現在の状態と、設定した状態の間に不一致が存在する場合は、Operator は Kubernetes リソースの作成、更新、または削除を行い、設定した状態に戻ります。これらの Kubernetes リソースには、デプロイメント、サービス、または ルート が含まれる可能性があります。Operator なしでは、デプロイメント、サービス、ルートその他の Kubernetes リソースを手動で作成する必要があります。これには、時間のかかる学習曲線が関係する可能性があります。Operator を使用して、アプリケーションイメージ、サービスポート、およびクラスター外部でアプリケーションを公開するかどうかなど、アプリケーションの詳細を指定できます。次に、Operator はすべての Kubernetes リソースを作成し、管理します。今回のリリースより、多くのリソースではなく、単一の **OpenLibertyApplication** リソースのみを管理するようになりました。さらに、Operator はクラスター内のアプリケーションに関連するイベントを継続的に監視し、データおよびリソースを同期するために必要なアクションを実行します。Operator は Kubernetes リソースの管理を容易にするため、Operator はクラウドデプロイメントの詳細を多く処理しながらアプリケーションに集中することができます。

Open Liberty Operator は Runtime Component Operator をベースとしています。これはランタイム固有の Operator にインポートして標準化されたエンタープライズ機能を提供するための汎用演算子です。そのため、イメージストリームの使用や高可用性のためにアプリケーションの複数のインスタンスを実行する機能など、これらの 2 つの Operator 間に共通機能が存在します。Open Liberty Operator を使用すると、Operator コンテナーおよびコントローラーは Kubernetes Pod にデプロイされ、Operator は **kind: OpenLibertyApplication ステートメントで受信リソースをリッスンし** ます。OpenLibertyApplication カスタムリソース(CR **)** を作成する場合、Operator はアプリケーションがクラスターで実行されるために必要な Kubernetes リソースを管理します。

6.2. OPERATOR の機能

Open Liberty Operator にはケイパビリティーレベル 5 があり、以下の機能を含む エンタープライズレベルの機能を持つことになり ます。

- **水平の自動スケーリングによって提供される高可用性**
  水平の自動スケーリングを設定して、リソースの消費に基づいてアプリケーションのインスタンスの作成や削除を行うことができます。これは、アプリケーションの複数のインスタンスを実行して自動スケーリングする機能により、アプリケーションが高可用性になります。

- **デプロイメント管理の強化**
  Open Liberty Operator を使用すると、Kubernetes にデプロイされたアプリケーションを簡単に管理できます。たとえば、Operator デプロイメントファイルで、**applicationImage** フィールドに イメージストリーム を指定できます。次に、新しいバージョンのアプリケーションの新規コンテナータグをアップロードした後、Operator はローリングでアプリケーションを更新します。

- **自動サービスバインディング**
  Operator はアプリケーション間のバインディング情報の更新を自動化します。つまり、アプリケーションが接続し、特定のアプリケーションがサービスを生成または消費するかどうかについての情報を維持します。この情報により、Operator は Kubernetes Secret の作成および挿入を含む Kubernetes レベルの詳細を自動的に処理し、アプリケーションが中断せずに必要なサービスに接続できるようにします。

- **シングルサインオン(SSO)委譲**
  Open Liberty を使用すると、SSO 認証を外部プロバイダーに委譲 できます。Open Liberty Operator を使用すると、アプリケーションの SSO 情報の設定および管理が容易になります。

- **OpenShift Serverless(Knative)の統合**
  Operator を Knative と統合できます。Operator が Knative と統合されると、単一のトグルを使用してサーバーレスランタイムコンポーネントをデプロイします。Operator は生成されたすべてのリソースを Knative リソースに変換します。これにより、Pod をアイドル状態の場合に自動的にゼロにスケーリングできます。

- **Kubernetes Application Navigator(kAppNav)の統合**
  Operator は kAppNav とのインテグレーションを自動的に設定でき ます。このインテグレーションを使用すると、アプリケーションのリソースを監視し、コンポーネントのヘルスステータスが変更されるとアラートを受信できます。統合ペインから、コンポーネントのトレースの有効化や Kibana または Grafana ダッシュボードの表示など、操作中心の機能にアクセスすることもできます。

- **OpenShift 証明書管理の統合**
  Operator は、Kubernetes クラスター にインストールされている場合に cert-manager ツールを利用 できます。cert-manager ツールを使用すると、オペレーターは Pod およびルートの Transport Layer Security(TLS)証明書を自動的にプロビジョニングできます。証明書は Kubernetes Secret からコンテナーにマウントされ、証明書が更新されると自動的に更新されます。

6.3. OPEN LIBERTY OPERATOR での保守性および可観測性

ストレージのサイズのみを指定 し、ストレージをマウントする場所を指定して、アプリケーションの永続性を有効 にできます。次に、Operator はストレージ要求を作成し、管理します。永続ボリューム要求(PVC)の追加詳細の設定を可能にする高度なモードを利用できます。また、サーバーメモリーダンプやサーバートレースの収集など、サービス関連の操作に単一のストレージを設定し、使用することもできます。サーバーメモリーダンプおよびサーバートレースの収集に関する詳細は、「 Day-2 operations 」を参照してください。

Operator の設定後に、Open Liberty と Kubernetes クラスターで可観測性のロギングおよび監視ツールを統合 することができます。監視するさまざまな種類の Open Liberty データを選択できます。ロギングイベントを可視化および追跡するには、提供される Open Liberty Kibana ダッシュボードのいずれかをデプロイします。MicroProfile Metrics、Prometheus、および Grafana を使用してメトリクスデータを収集し、収集し、視覚化することで Open Liberty メトリクスをモニターできます。また、MicroProfile Health を有効にして liveness チェックおよび readiness チェックを実行して、Kubernetes がコンテナーの正常性を確認できるようにすることもできます。

6.4. OPERATOR のインストールと設定

Kubernetes または OpenShift で使用するために 、OperatorHub から Open Liberty Operator をインストール できます。この オペレーターは、OpenShift Container Platform(OCP)から Red Hat 認定演算子としても利用 できます。Open Liberty Operator ドキュメントでは、基本設定、カスタムリソース定義(CRD)パラメーター、Open Liberty コンソールのロギング環境変数、永続ストレージ仕様など、Operator の設定に関する詳細が記載されています。

6.5. も併せて参照してください。

- Guide: Kubernetes Operator を使用したアプリケーションの OpenShift へのデプロイ
- Open Liberty Operator のトラブルシューティング

第7章 開発モード

開発モードで Open Liberty を実行する場合、統合開発環境(IDE)またはテキストエディターから直接コード、デプロイ、テスト、およびデバッグを行うことができます。dev モードとして知られる開発モードを有効にして、Maven または Gradle ビルド自動化ツールのいずれかと連携できます。

dev モードでは、コードへの変更を迅速に繰り返し実行し、オンデマンドまたは自動ユニットおよび統合テストから即座にフィードバックを受け取ることができます。また、デバッガーを割り当てて、いつでもコードを実施することもできます。dev モードは、Open Liberty Maven プラグインの目標として、または Liberty Gradle プラグイン のタスクとしても利用でき ます。実行中のサーバーを再起動せずにアプリケーションをリアルタイムで編集および監視できるように、Open Liberty の機能セットを統合します。dev モードは、変更のデプロイ、実行テスト、およびデバッグの 3 つの主要なフォーカスエリアに対応します。

7.1. DEV モードでの OPEN LIBERTY の実行

dev モードで Open Liberty を実行するには、Open Liberty Maven プラグインまたは Open Liberty Gradle プラグインを有効 にして、以下の コマンドのいずれかを実行します。

maven: **mvn liberty:dev**

gradle: **gradle libertyDev**

7.1.1. コードの変更の検出、再コンパイル、デプロイ

Dev モードは、IDE またはテキストエディターに新しい変更を保存するたびに、コードの変更を自動的に検出、再コンパイル、デプロイできます。dev モードは、アプリケーションソースに以下の変更を自動的に検出します。

- Java ソースファイルとファイル変更のテスト
- リソースファイルの変更
- 設定ファイルと設定ファイルの変更
- Gradle ユーザーの Maven ユーザーまたは **build .gradle ファイルの pom. xml** ファイルに新しい依存関係を追加
- Open Liberty サーバー設定に新機能を追加

リソースファイル、設定ファイル、設定ディレクトリーの変更は、ターゲットディレクトリーにコピーされます。**pom.xml ファイルまたは build. gradle** ファイルの新しい依存関係がクラスパスに追加されます。新機能がインストールされ、起動している。

特定の設定ディレクトリーやファイルの追加など、一部の変更は、dev モードを再起動するまで反映されません。これらの変更を有効にするには、プロンプトが表示されたら dev モードを再起動します。再起動するには、最初に **CTRL+C** を押して、**q** を入力して **Enter** を押して、最初に dev モードを終了します。次に、**mvn liberty:dev** コマンドまたは **gradle libertyDev コマンドを実行して** 再起動します。サーバーの再起動後、変更は稼働中のサーバーによって検出、再コンパイル、および選択されます。

dev モードの開始時にパラメーターを指定すると、dev モードがコードへの変更を処理する方法を設定できます。設定パラメーターの詳細は、Open Liberty Maven プラグインの dev ゴールまたは Open Liberty Gradle プラグイン の libertyDev タスクを参照して ください。

7.1.2. オンデマンドでユニットと統合テストの実行

dev モードが実行しているコマンドウィンドウで **Enter** を押して、ユニットおよび統合テストをオンデマンドで実行できます。dev モードは、プロジェクトに設定されたユニットテストおよび統合テストを実行します。テストをプロジェクトに追加する場合、dev モードがコンパイルされ、次にテストを実行するときにこれが含まれます。

dev モードがホットテストを実行するように設定することで、変更に関するフィードバックを即座に受け取ることができます。ホットテストとは、dev モードを起動するか、コード変更を行うたびに自動実行されるユニットまたは統合テストです。ホットテストを設定するには、以下の例のように、dev モードの開始時にホットテストパラメーターを指定します。

Maven: **mvn liberty:dev -DhotTests**

gradle: **gradle libertyDev --hotTests**

また、パラメーターを追加して、テストをスキップするかどうかを指定することも可能です。Maven では、パラメーターを追加してユニットテストをスキップしたり、統合テストをスキップしたり、すべてのテストをスキップしたりできます。Gradle の場合は、パラメーターを追加してすべてのテストをスキップできます。設定パラメーターの詳細は、Open Liberty Maven プラグインの dev ゴールまたは Open Liberty Gradle プラグイン の libertyDev タスクを参照して ください。

7.1.3. 実行中のサーバーにデバッガーを割り当てます。

実行中のサーバーにデバッガーを割り当て、いつでもコードを追跡できます。ソースコードにブレークポイントを指定して、アプリケーションの異なる部分をローカルでデバッグできます。デバッグのデフォルトポートは *7777* **です**。デフォルトのポートが利用できない場合、dev モードはデバッグ用のポートとして使用するランダムポートを選択します。

7.1.4. dev モードの VS Code 拡張

Open Liberty Tools VS Code 拡張 を使用すると、dev モードを起動し、動的コードの変更、テストの実行、およびデバッグを行うことができます。すべての VS Code エディターを終了します。エクステンションをインストールし、Maven または Gradle プラグインのいずれかをインストールしたら、VS Code サイドバーの Liberty Dev Dashboard でプロジェクトを選択できます。プロジェクト名を右クリックし、メニューからコマンドを選択して、dev モード機能にアクセスできます。

7.2. も併せて参照してください。

- demo-devmode サンプルプロジェクト （Maven および Gradle ユーザー）
- Guide: Open Liberty （Maven ユーザー）の使用

第8章 スレッドプールの調整

Open Liberty は、スレッドプールのサイズを制御する自己調整アルゴリズムを提供します。ほとんどのアプリケーションのスレッドプールを手動で調整する必要はありませんが、場合によっては、コアスレッドおよび **maxThreads** 属性の調整が必要に **なる** 場合があります。

Open Liberty のすべてのアプリケーションコードは、デフォルトのエグゼキューターと呼ばれる単一のスレッドプールで実行されます。このプールのサイズは自己チューニングコントローラーによって設定され、幅広いワークロードを管理できます。デフォルトのエグゼキュータープールは、アプリケーションコードが実行されるスレッドのセットです。Open Liberty は、OSGi フレームワークを提供し、JVM ガベージを収集し、Java NIO トランスポート機能を提供するタスクに他のスレッドを使用します。ただし、これらのスレッドはアプリケーションのパフォーマンスに直接関連せず、そのほとんどは設定できません。

8.1. スレッドプールのチューニング動作

Open Liberty スレッドプール自己チューニングプロセスは 1.5 秒ごとに実行されます。スレッドプールコントローラーは、サーバーの起動時にスレッドプールのパフォーマンスに関するデータセットを維持します。スループットは、コントローラーが各サイクルで完了するタスク数により決定されます。コントローラーは、以前に試行されたさまざまなプールサイズのスループットを記録します。その後、この履歴スループットデータは、現在のサイクルのスループットと比較して、最適なプールサイズを決定します。各サイクルのプールサイズは、段階的に増減したり、変更したりしないでください。

場合によっては、決定をガイドする履歴データが利用できない場合があります。たとえば、プールが各サイクルで拡張され、現在のサイクルが最大サイズに達すると、大きなプールサイズのスループットに関するデータは存在しません。このような場合には、コントローラーはプールのサイズを増やすかどうかを無作為に決定します。次に、その決定の結果に基づいて次のサイクルの正式の読み取りを行います。このプロセスは、さまざまなスレッドプールサイズを試して設定およびワークロードに最適な値を判断するために、ヒューマンスレッドプールサイズに類似しています。

1.5 秒ごとのサイクルごとに、スレッドプールコントローラーは以下の自己チューニング操作で実行されます。

1. プールのスレッドがハングしているかどうかを確認します。タスクがキューにあり、以前のサイクルでタスクが完了しなかった場合、コントローラーはスレッドがハング状態にあると見なします。この場合、コントローラーは設定で指定したスレッドプールサイズを増やし、ステップ 5 に進みます。

2. 直近のコントローラーサイクルで完了したタスクの数で、過去のデータセットを更新します。パフォーマンスは、各プールサイズの重み平均として記録されます。このパフォーマンスは履歴の結果を反映しますが、ワークロードの特性を変更するためにすぐに調整されます。

3. 履歴データを使用して、パフォーマンスの小さいか、またはそれ以上のプールサイズでパフォーマンスが向上するかどうかを予測します。小さいプールサイズまたはそれ以上のプールサイズの履歴データが存在しない場合、スレッドプールコントローラーはプールのサイズを拡大または縮小するかを決定します。

4. 予測パフォーマンスに基づいて、設定で指定した境界内にプールサイズを増減したり、変更せずにそのまま残します。

5. スリープ状態に戻ります。

スレッドプールサイズ以外のさまざまな要因は、Open Liberty サーバーのスループットに影響を与える可能性があります。プールサイズと監視スループットの関係は、スムーズまたは継続的ではありません。したがって、過去のスループットデータから派生する予測を改善するために、コントローラーは、最も最も近いプールサイズだけでなく、各方向の増分数も考慮します。

8.1.1. ハング解決

アプリケーションシナリオによっては、プール内のすべてのスレッドは、他の作業が終了するまで待機する必要のあるタスクによってブロックされる可能性があります。このような場合、特定のプールサイズでサーバーがハングする可能性があります。この状況を解決するために、スレッドプールコントローラーがハング解決モードに入ります。

ハング解決によりスレッドがプールに追加され、サーバーが通常の操作を再開できるようになります。また、ハング解決により、コントローラーサイクルの期間が短縮され、デッドロックがすぐに破損します。

コントローラーがタスクを再度完了していることを確認すると、通常の操作が再開します。コントローラーサイクルは通常の期間に戻り、プールサイズは通常のスループット基準に基づいて調整されます。

8.2. 手動スレッドプールの調整

多くの環境、設定、およびワークロードでは、Open Liberty スレッドプールに手動の設定またはチューニングは必要ありません。スレッドプールの自己チューニングは、最適なサーバーのスループットを提供するために必要なスレッド数を決定します。スレッドプールコントローラーは、コアスレッド **および** maxThreads 属性に定義されたバインド内のプール内のスレッド数を継続的に調整 **し** ます。ただし、コアスレッドまたは **maxThreads** 属性の設定が **必要** になることがあります。以下のセクションでは、これらの属性を説明し、手動で調整する必要がある可能性のある条件の例を示します。

- **coreThreads**
  この属性は、プール内のスレッドの最小数を指定します。この属性の最小値は 4 です。Open Liberty は、スレッドの数がこの属性の値と同じになるまで、提供された作業ごとに新しいスレッドを作成します。coreThreads **属性** が設定されていない場合、デフォルトは Open Liberty プロセスで利用可能なハードウェアスレッドの数の倍数に設定されます。
  Open Liberty を共有環境で実行されている場合は、スレッドプールコントローラーが利用可能な CPU を共有する他のプロセスに対応できません。このような場合、coreThreads 属性のデフォルト値により 、 Open Liberty が CPU リソースの競合している他のプロセスを考慮して、最適なスレッドを作成する可能性があります。このような場合には、Open Liberty の **実行** に必要な CPU リソースの割合のみを反映する coreThreads 属性の値を値に制限することができます。

- **maxThreads**
  この属性は、プール内のスレッドの最大数を指定します。デフォルト値は -1（ **MAX_INT** と同等）、または実質的に無制限です。
  一部の環境では、プロセスが作成できるスレッド数にハード制限を設定します。現在、Open Liberty は現在、そのような上限が適用されるかどうか、または値が何であるかを知る方法がありません。Open Liberty がスレッド制限環境で実行されている場合、Operator は **maxThreads** 属性を許可される値に設定できます。

Open Liberty スレッドプールコントローラーは、さまざまなワークロードおよび設定を処理するように設計されています。エッジによっては、コアスレッド **および maxThreads** 属性の調整が必要になる場合があります。ただし、まずデフォルトの動作を試して、調整を行う必要があります。

8.3. も併せて参照してください。

エグゼキューター管理

第9章 ロギングおよびトレース

Open Liberty には、アプリケーションとランタイムが記述されたメッセージを処理し、First Failure Data Capture(FFDC)機能を提供する統一されたロギングコンポーネントが含まれています。**System.out、System.err、または java. util.logging. Logger** を使用してアプリケーションによって書き込まれたデータはサーバーログに統合されます。

サーバーには、主に 3 つのログファイルがあります。

- **console.log** - このファイルは、**サーバー起動** コマンドにより作成されます。JVM プロセスからのリダイレクトされた標準出力と標準エラーストリームが含まれます。このコンソールの出力は、人間による直接消費を目的としているため、ログの自動分析に有用な情報がありません。

- **messages.log**: このファイルには、ロギングコンポーネントが書き込みまたはキャプチャーされるすべてのメッセージが含まれます。このファイルに書き込まれるすべてのメッセージには、メッセージタイムスタンプやメッセージを作成したスレッドの ID などの追加情報が含まれます。このファイルは、自動ログ分析に適しています。このファイルには、JVM プロセスによって直接書き込まれるメッセージは含まれません。

- **trace.log**: このファイルには、ロギングコンポーネントによって書き込みまたはキャプチャーされるすべてのメッセージと追加のトレースが含まれます。このファイルは、追加のトレースを有効にした場合にのみ作成されます。このファイルには、JVM プロセスによって直接書き込まれるメッセージは含まれません。

9.1. ロギングの設定

ロギングコンポーネントはサーバー設定を使用して制御できます。logging コンポーネントは、logging 要素を使用して **server.xml** で完全に設定 **でき** ます。ただし、**server.xml が処理される前にロギングが初期化されるため、server.xml** でロギングを設定すると、後続のログ設定とは異なるログ設定を使用して初期のログエントリーが発生する可能性があります。このため、boostrap **.properties** を使用してロギング設定や、環境変数を使用する状況によっては、多くのロギング設定を提供することもできます。

9.2. ロギング設定の例

以下のセクションでは、一般的なロギング設定の例をいくつか示しています。

9.2.1. ログファイルストレージの管理

console.log ファイルは、プロセス **stdout** および **stderr** をファイルにリダイレクトして作成されます。そのため、Liberty は、**messages.log に提供されるように、ログロールオーバーと同様に、同じレベルの管理を提供することができませ** ん。console.log ファイルのサイズの増加を懸念している場合には、**console.log** ファイルを無効にして 、 代わりに **messages.log** ファイルを使用できます。**console.log に送信されたすべてのログ** メッセージは **messages.log** ファイルに書き込まれます。また、ファイルのロールオーバーを設定できます。

コンソールログを無効にし、**messages.log** を 100Mb に 3 回実行するように設定するには、以下の設定を使用します。

```
com.ibm.ws.logging.max.file.size=100
com.ibm.ws.logging.max.files=3
com.ibm.ws.logging.console.log.level=OFF
com.ibm.ws.logging.copy.system.streams=false
```

9.2.2. JSON ロギング

ログファイルを最新のログ集計および管理ツールに提供する場合、JSON 形式を使用してログファイルを保存すると利点があります。これは、以下の 3 つの方法の 1 つで実行できます。

- **bootstrap.properties** ファイルの使用

```
com.ibm.ws.logging.message.format=json
com.ibm.ws.logging.message.source=message,trace,accessLog,ffdc,audit
```

- 環境変数の使用：

```
WLP_LOGGING_MESSAGE_FORMAT=json
WLP_LOGGING_MESSAGE_SOURCE=message,trace,accessLog,ffdc,audit
```

- **server.xml** ファイルの使用

```
<logging messageFormat="json" messageSource="message,trace,accessLog,ffdc,audit" />
```

server.xml を使用して一部のログ行を設定し、**server.xml** の起動前に一部のログ行がデフォルトの JSON 形式に書き込まれるため、一部のツールで問題が発生する可能性があります。たとえば、**jq** はログファイルを理解できません。

9.2.3. Docker イメージのロギングの設定

messages.log を無効にし、代わりにコンソール出力を JSON としてフォーマットする Docker 環境では一般的です。環境変数を使用してこれを行うことができます。

```
WLP_LOGGING_MESSAGE_FORMAT=json
WLP_LOGGING_MESSAGE_SOURCE=
WLP_LOGGING_CONSOLE_FORMAT=json
WLP_LOGGING_CONSOLE_LOGLEVEL=info
WLP_LOGGING_CONSOLE_SOURCE=message,trace,accessLog,ffdc,audit
```

これは、**-e** を使用して環境変数を設定することで、**docker run** コマンドを実行する際に単に設定できます。

```
docker run -e "WLP_LOGGING_CONSOLE_SOURCE=message,trace,accessLog,ffdc"
      -e "WLP_LOGGING_CONSOLE_FORMAT=json"
      -e "WLP_LOGGING_CONSOLE_LOGLEVEL=info"
      -e "WLP_LOGGING_MESSAGE_FORMAT=json"
      -e "WLP_LOGGING_MESSAGE_SOURCE=" open-liberty
```

9.2.4. バイナリーロギング

liberty には、トレースファイルの書き込みのオーバーヘッドを大幅に削減する高パフォーマンスのバイナリーログフォーマットオプションがあります。通常、バイナリーロギングを設定する場合は **console.log** が無効になり、最適なパフォーマンスが得られます。これは **bootstrap.properties を使用して有効にする必要があり** ます。

```
websphere.log.provider=binaryLogging-1.0
com.ibm.ws.logging.console.log.level=OFF
com.ibm.ws.logging.copy.system.streams=false
```

binaryLog **コマンドラインツール** は、バイナリーログをテキストファイルに変換するために使用できます。

```
binaryLog view defaultServer
```

9.3. ソースによる設定

以下の表は、同等の **server.xml**、**bootstrap.properties**、および環境変数の設定と簡単な説明を示しています。詳細な設定に関するドキュメントは、logging 要素の config reference で利用できます。

サーバー XML 属性	ブートストラッププロパティー	env var	説明
hideMessage	com.ibm.ws.logging.hideMessage		この属性を使用して、Console.log ファイルおよび messages **.log** ファイルから非表示にする必要のある **メッセージキーを設定できます**。メッセージを非表示にするように設定されている場合は、**trace.log** ファイルにリダイレクトされます。
logDirectory	com.ibm.ws.logging.log.directory	LOG_DIR	この属性を使用して、**console.log ファイルを除くすべてのログ** ファイルのディレクトリーを設定できますが、FFDC も含むことができます。デフォルトは WLP **_OUTPUT_DIR/serverName/logs** です。logDirectory を server.xml に設定する **こと** は推奨されません。これは、**server.xml** を読み取る前に一部のログデータがデフォルトの場所に書き込ま **れる** ためです。
コンソールログ設定			

サーバー XML 属性	ブートストラッププロパティー	env var	説明
consoleFormat	com.ibm.ws.logging.console.format	WLP_LOGGING_CONSOLE_FORMAT	コンソールに必要な形式。有効な値は **basic** または **json** 形式です。デフォルトでは、consoleFormat **は basic** に設定されます。
consoleLogLevel	com.ibm.ws.logging.console.log.level	WLP_LOGGING_CONSOLE_LOGLEVEL	このフィルターは、コンソールに送信されるメッセージの粒度を制御します。有効な値は INFO、AUDIT、WARNING、ERROR、OFF です。デフォルトは AUDIT です。Eclipse 開発者ツールを使用する場合は、これをデフォルトに設定する必要があります。
consoleSource	com.ibm.ws.logging.console.source	WLP_LOGGING_CONSOLE_SOURCE	コンソールにルーティングするカンマ区切りのソースの一覧。このプロパティーは、consoleFormat **="json" の場合にのみ適用され**ます。有効な値は **message**、**trace**、**accessLog**、**ffdc**、および **audit** です。デフォルトで、consoleSource **は message** に設定されます。**監査** ソースを使用するには、Liberty audit-1.0 機能を有効にします。accessLog **ソース** を使用するには、httpAccessLogging を設定する必要があります。

サーバー XML 属性	ブートストラッププロパティー	env var	説明
copySystemStreams	com.ibm.ws.logging.copy.system.streams		true の場合、System.out および System.err ストリームに書き込まれるメッセージはプロセス **stdout** および **stderr** にコピーされ、**console.log** に表示されます。false の場合、これらのメッセージは messages.log、**trace.log などの設定されたログに書き込まれますが、 stdout** および **stderr** にコピーされず、**console.log** には表示されません。デフォルト値は true です。
Message Log Config			
	com.ibm.ws.logging.newLogsOnStart		Liberty の開始時に true に設定すると、既存の **messages.log** または **trace.log** ファイルがロールアウトされ、新しい message **.log または trace.log** ファイルへの書き込みがロギングされます。false **messages.log または trace.log** ファイルを設定すると、maxFileSize に到達した場合にのみ更新され **ます**。デフォルトは **true** です。この設定は、サーバーのブートストラップ時にのみ処理されるため、**server.xml** の **logging** 要素を使用して提供できません。

サーバー XML 属性	ブートストラッププロパティー	env var	説明
isoDateFormat	com.ibm.ws.logging.isoDateFormat		ログファイルで ISO-8601 形式の日付を使用するかどうかを指定します。デフォルト値は false です。 true に設定すると、ISO-8601 形式が **messages.log ファイル、trace.log ファイル、および FFDC ログで** 使用されます。形式は **yyyy-MM-dd'T'HH:mm:ss.SSSZ です**。 **false** の値を指定すると、システムに設定したデフォルトのロケールに従って日時がフォーマットされます。デフォルトのロケールが見つからない場合、形式は **dd/MMM/yyyy HH:mm:ss:SSS z** になります。
maxFiles	com.ibm.ws.logging.max.files		各ログファイルが保持される数。この設定は、FFDC の例外要約ログの数にも適用されます。したがって、この番号が **10** の場合、ffdc/ ディレクトリーに 10 個のメッセージログ、10 トレースログ、例外要約が 10 個ある可能 **性があり** ます。デフォルトでは、値は **2** です。**console.log** はロールを持たないため、この設定は適用されません。

サーバー XML 属性	ブートストラッププロパティー	env var	説明
maxFileSize	com.ibm.ws.logging.max.file.size		ログファイルが展開される前に到達できる最大サイズ（MB 単位）。値を **0** に設定すると、ログローリングが無効になります。デフォルト値は **20** です。**console.log** はロールを持たないため、この設定は適用されません。
messageFileName	com.ibm.ws.logging.message.file.name		メッセージログのデフォルト名は **messages.log** です。このファイルは常に存在し、System. **out および System. err に加えて INFO およびその他のメッセージ（AUDIT、WARNING、ERROR、FAILURE）メッセージが含まれ**ます。このログには、タイムスタンプと発行先のスレッド ID も含まれます。ログファイルがロールオーバーされると、以前のログファイルの名前の形式は **messages_timestamp.log**になります。
messageFormat	com.ibm.ws.logging.message.format	WLP_LOGGING_MESSAGE_FORMAT	**messages.log** ファイルに必要な形式。有効な値は **basic** または **json** 形式です。デフォルトでは、messageFormat **は basic** に設定されます。

サーバー XML 属性	ブートストラッププロパティー	env var	説明
messageSource	com.ibm.ws.logging.message.source	WLP_LOGGING_MESSAGE_SOURCE	**messages.log** ファイルにルーティングするカンマ区切りのソースの一覧。このプロパティーは、messageFormat **="json" の場合にのみ適用され** ます。有効な値は **message**、**trace**、**accessLog**、**ffdc**、および **audit** です。デフォルトでは、messageSource **は message** に設定されます。**監査** ソースを使用するには、Liberty audit-1.0 機能を有効にします。accessLog **ソース** を使用するには、httpAccessLogging を設定する必要があります。
トレース設定			
suppressSensitiveTrace			サーバートレースは、ネットワーク接続で受信されるバイトなど、タイプのないデータを追跡すると機密データを公開できます。この属性は、**true** に設定すると、潜在的な機密情報がログおよびトレースファイルで公開されないようにします。デフォルト値は **false** です。
traceFileName	com.ibm.ws.logging.trace.file.name		**trace.log** ファイルは、追加のトレースまたは詳細なトレースが有効になっている場合にのみ作成されます。**stdout** は特殊な値として認識され、トレースは元の標準出力ストリームに転送されます。

サーバー XML 属性	ブートストラッププロパティー	env var	説明
traceFormat	com.ibm.ws.logging.trace.format		この属性は、トレースログの形式を制御します。Liberty のデフォルト形式は ENHANCED **です**。**BASIC** 形式および ADVANCED 形式を使用すること **も** できます。

サーバー XML 属性	ブートストラッププロパティー	env var	説明
traceSpecification	com.ibm.ws.logging.trace.specification		trace 文字列は、トレースを選択的に有効化するために使用されます。ログレベル仕様の形式： component = level **component** は、レベルに設定するログソースを指定し、**level** は、以下のいずれかを使用して、出力すべきトレースの量を **指定 し** ます。**fatal**、**severe**、**warning**、**audit**、**info**、**config**、 details、**finer** 、**finer**、**finest**、**all**複数のログレベルを指定することで、コロンで区切ります。 コンポーネントには、ロガー名、トレースグループ、またはクラス名を指定できます。アスタリスク * はワイルドカードとして機能し、プレフィックスに基づいて複数のコンポーネントに一致するワイルドカードとして機能します。以下に例を示します。 ● * 製品システムコードやカスタマーコードなど、アプリケーションサーバーで実行されているすべての追跡可能なコードを指定します。 ● **com.ibm.ws.*** は、com.ibm.ws で始まるパッケージ名を持つすべてのクラスを指定します。 ● **com.ibm.ws.classloading.AppClassLoader は**

サーバー XML 属性	ブートストラッププロパティー	env var	説明
			AppClassLoader クラスのみを指定します。

第10章 TRUE-TO-PRODUCTION インテグレーションテスト

MicroShed Testing は Java ライブラリーで、実稼働環境に似た環境でアプリケーションの実際の統合テストを作成するのに役立ちます。開発と実稼働間のパリティーの問題を最小限に抑えるには、実稼働環境で使用する Docker コンテナーと同じ Docker コンテナーでアプリケーションをテストします。

MicroShed Testing は Testcontainers フレームワーク を 使用して、アプリケーションの内部にアクセスせずに Docker コンテナー外からアプリケーションを分析します。MicroShed Testing を使用して、Open Liberty アプリケーションの統合テストを開発できます。

開発環境と同様に、アプリケーションが正確に機能することを確認する必要があります。統合テストは複数のテストクラスおよびコンポーネントを評価しますが、統合テストではユニットテストよりも設定に長い時間がかかります。つまり、ユニットテストは短く、アプリケーションの個別のモジュールのテストを行います。開発サイクルおよび時間制限が短縮され、開発者はユニットテストを実行することが多くあります。MicroShed Testing は、アプリケーションの実稼働用の統合テストを作成および実行し、統合テストを効率的なワークフロー用に Testcontainers フレームワークで効率化します。

10.1. 開発および実稼働パリティー

開発と実稼働パリティー は、最新のアプリケーション を構築する ための方法論です。開発および実稼働パリティーの背後にある概念は、時間、人員、およびツールと同様の開発、ステージング、および実稼働環境を維持することです。開発プロセスを簡素化するために、開発者は実稼働環境で異なる開発のツールを使用します。たとえば、ローカルの Maven ビルドを使用して、開発時にアプリケーションのプロジェクトをビルドできますが、アプリケーションは実稼働環境で Docker コンテナーにデプロイされる可能性があります。環境間で異なると、テストは開発環境で渡されますが、テストは本番環境で失敗する可能性があります。MicroShed Testing は、実稼働環境でアプリケーションをテストすることで、開発および実稼働のパリティーを実現するために役立ちます。

10.2. MICROSHED TESTING ライブラリーでの統合テストの作成

コンテナーを使用する異なる環境にアプリケーションをデプロイすることができます。Testcontainers フレームワークを使用すると、テスト環境でコンテナーを使用できます。Microshed Testing は、MicroProfile および Jakarta EE 開発者向けの Java ライブラリーで、実稼働環境でアプリケーションをテストします。Microshed Testing は Testcontainers フレームワークを実装し、テストで開発および実稼働パリティーをサポートします。

MicroShed Testing を使用すると、以下の例のようなインテグレーションテストを作成できます。

```
@MicroShedTest
public class BasicJAXRSServiceTest {

    @Container
    public static ApplicationContainer app = new ApplicationContainer()
                .withAppContextRoot("/myservice");

    @RESTClient
    public static PersonService personSvc;

    @Test
    public void testGetPerson() {
        Long bobId = personSvc.createPerson("Bob", 24);
        Person bob = personSvc.getPerson(bobId);

        assertEquals("Bob", bob.name);
        assertEquals(24, bob.age);
```

```
        assertNotNull(bob.id);
    }

    @Test
    public void testGetUnknownPerson() {
        assertThrows(NotFoundException.class, () -> personSvc.getPerson(-1L));
    }
}
```

@MicroShedTest **アノテーション** は、リポジトリーの Dockerfile ドキュメントを検索し、Docker コンテナーでアプリケーションを起動し、テストの開始前にアプリケーションが準備されるのを待機します。@ **Container アノテーション** は PersonService クラスの REST Client プロキシーを注入します。これは、実行中のアプリケーションコンテナーで HTTP 要求を送信するのに役立ちます。@RESTClient **アノテーションは**、HTTP POST リクエストを実行中のコンテナーに送信します。PersonService#createPerson エンドポイントをトリガーし、生成された ID を返します。生成された ID を使用すると、HTTP GET リクエストを送信して、作成されたレコードを読み取ることができます。JSON 応答は、JSON-B トークンを使用して Person オブジェクトに自動的に変換します。@ **Test アノテーション** は HTTP GET リクエストを送信し、 **-1** ID がない Person オブジェクトを検索します。アノテーションは、アプリケーションコンテナーが HTTP 404 例外を返すことをアサートします。

10.3. も併せて参照してください。

Guide: MicroProfile または Jakarta EE アプリケーションのテスト

第11章 デバッグ

11.1. OPEN LIBERTY MAVEN プラグインを使用したビルドプロセスの管理

単一のモジュールを持つシンプルなアプリケーションか、複数のモジュールで構成されるより複雑なアプリケーションかどうか、アプリケーションを構築してテストできます。

プロジェクトの詳細と依存関係を定義した後、Maven は自動的にすべての依存関係をダウンロードしてインストールします。また、ビルド後にアプリケーション上で自動テストを実行します。アプリケーションの更新後にテストを通過しない場合は、ビルドに失敗します。コードを修正する必要があります。

Maven プラグインには、以下のコーディネートが必要です。

```
<groupId>io.openliberty.tools</groupId>
<artifactId>liberty-maven-plugin</artifactId>
<version>3.1.0</version>
```

Maven および Liberty Maven プラグインを使用して簡単な Web サーブレットアプリケーションを設定する方法は、「 Maven を使用した Web アプリケーションのビルド」を参照してください。

Jakarta EE アプリケーションは、1 つのエンティティーとして連携する複数のモジュールで構成されます。Maven および Open Liberty を使用して複数のモジュールでアプリケーションを構築する方法は、「マルチモジュールアプリケーションの 作成」を参照してください。

11.2. リクエストの追跡

分散トレースは、分散システムで伝播する際に要求を検査およびロギングすることでマイクロサービスのトラブルシューティングに役立ちます。これにより、開発者はこれらのリクエストのデバッグの困難な作業に対応できます。分散トレースシステムが配置されない場合、リクエストを受信する時または応答が返されるタイミングと、ワークフローおよびピンポイントの分析が困難です。

アプリケーションのマイクロサービス全体でのロギング要求を監視およびトレースする方法は、「マイクロサービスでの 分散トレースの有効化」を参照してください。

第12章 OPEN LIBERTY の監視

MicroProfile Metrics および MicroProfile Health を使用して、Open Liberty で実行されるマイクロサービスとアプリケーションを監視できます。マイクロサービスのメトリクスおよびヘルスチェックデータを有効化および報告すると、問題の特定、キャパシティープランニング用のデータを収集し、サービスのスケールアップまたはダウンのタイミングを決定するのに役立ちます。

12.1. メトリクスを使用したモニタリング

Open Liberty では、アプリケーション、MicroProfile Metrics によって提供される REST エンドポイントスタイルのメトリクス、および Java Management Extensions(JMX)メトリクスを監視できるメトリクスが 2 つ利用できます。

MicroProfile Metrics には、Prometheus などのモニタリングツールや REST 要求を実行できるすべてのクライアントがアクセスできます。JMX メトリクスは、JMX サーバーと通信できる Java ベースの監視ツールやカスタム JMX クライアントによる使用に適しています。

12.1.1. MicroProfile Metrics および /metrics エンドポイント

MicroProfile Metrics 機能は、MicroProfile Metrics 仕様に準拠する **/metrics** REST インターフェースを提供します。Open Liberty は MicroProfile Metrics を使用して、多くの Open Liberty コンポーネントの内部状態を記述するメトリクスを公開します。開発者は MicroProfile Metrics API を使用して、アプリケーションからメトリクスを公開することもできます。

メトリックは、カウンター、測定、タイマー、ヒストグラム、ヒストグラムなどのさまざまな形式で提供されます。MicroProfile Metrics 機能を 有効にして MicroProfile Metrics にアクセスでき ます。すべてのメトリックのリアルタイム値は、**/metrics** エンドポイントを呼び出すことで利用できます。これらのメトリクスをアプリケーションに追加する方法は、「メトリクスを使用した マイクロサービス可観測性」を参照してください。Open Liberty で利用可能なすべての REST エンドポイントスタイルのメトリクスの一覧は、「 メトリクス参照リスト」を参照 してください。

/metrics エンドポイントは 2 つの出力形式を提供します。各応答が使用する形式は、対応するリクエストの HTTP 許可ヘッダーによって異なります。Prometheus 形式は、オープンソースメトリクス収集、データストア 、 および基本的な視覚化ツールです。この形式は、**テキスト/plain** accept ヘッダーが設定されたリクエストに対して返されます。JSON 形式はメトリクスの JSON 表現です。この形式は、**application/json** accept ヘッダーが設定されたリクエストに対して返されます。

以下の表は、Prometheus または JSON 形式でメトリクスを提供するために GET 要求でアクセスできる各種のエンドポイントを示しています。

表12.1 GET リクエストがアクセスできるメトリクスエンドポイント

エンドポイント	要求タイプ	サポートされる形式	説明
/metrics	GET	Prometheus、JSON	登録したすべてのメトリクスを返します。
/metrics/<scope>	GET	Prometheus、JSON	指定されたスコープに登録されるメトリクスを返します。

エンドポイント	要求タイプ	サポートされる形式	説明
/metrics/<scope>/<metric name>	GET	Prometheus、JSON	指定されたスコープのメトリクス名に一致するメトリクスを返します。

12.1.2. JMX メトリクス

JMX メトリクスにアクセスするには、Performance Monitoring 機能を有効にし ます。この機能をサーバー設定に追加すると、JMX メトリクスは自動的に監視されます。Performance Monitoring 機能は、JConsole などの JMX を使用するモニタリングツールで使用できる MXBean を提供します。

JConsole は、JVM および Java アプリケーションの監視に使用できるグラフィカルな監視ツールです。Open Liberty のモニタリングを有効にしたら、JConsole を使用して JVM に接続し、MXBeans の属性をクリックしてパフォーマンスデータを表示できます。JMX メトリクスを使用する他のプロダクトを使用して、メトリクス情報を表示することもできます。Open Liberty で利用可能なすべての JMX メトリクスの一覧は、「 JMX metrics reference list 」を参照してください。

12.1.3. 両方のタイプのメトリクスの組み合わせ

MicroProfile Metrics 機能は、JVM および MicroProfile Metrics API を使用してアプリケーションに追加されるメトリクスに関する基本的なメトリクスを提供します。ただし、MicroProfile Metrics 機能は Performance Monitoring 機能と併用しない限り、Open Liberty サーバーコンポーネントのメトリクスを提供しません。

MicroProfile Metrics および Performance Monitoring 機能の両方を有効にすると、監視用の MXBeans と **/metrics** REST エンドポイントがアクティベートされます。この場合、Open Liberty サーバーコンポーネントのメトリクスは両方のインターフェースで公開されます。MicroProfile Metrics 機能バージョン 2.3 以降は、Performance Monitoring 機能を自動的に有効にします。

12.2. マイクロサービスのヘルスチェック

ヘルスチェックは、マイクロサービスとその依存関係のステータスを検証するために使用できる特別な REST API 実装です。MicroProfile Health を使用すると、アプリケーションのマイクロサービスでそれらの健全性を自己チェックし、全体的なヘルスステータスが定義されたエンドポイントに公開されます。

ヘルスチェックは、依存関係、システムプロパティー、データベース接続、エンドポイント接続、リソースの可用性など、マイクロサービスが必要とするものをすべて評価できます。サービスは、MicroProfile Health によって提供される API を実装して可用性として報告します。マイクロサービスが利用可能になると、**UP** のステータスを報告します。マイクロサービスが利用できない場合は、**DOWN** のステータスを報告します。サービスオーソライザーはこれらのステータスレポートを使用して、アプリケーション内でマイクロサービスを管理し、スケーリングする方法を決定できます。ヘルスチェックは、Kubernetes の liveness および readiness プローブと対話すること もできます。

たとえば、マイクロサービスベースの銀行アプリケーションでは、ログインマイクロサービス、バランス転送マイクロサービス、および請求用のマイクロサービスにヘルスチェックを実装する場合があります。バランス転送マイクロサービスのヘルスチェックがバグまたはデッドロックを検出すると、**DOWN** ステータスを返します。この場合、**/health/live** エンドポイントが Kubernetes liveness プローブで設定されている場合、プローブはマイクロサービスを自動的に再起動します。マイクロサービスを再起動すると、ユーザーにダウンタイムやエラーが発生し、アプリケーション内の他のマイクロサービスの機能が維持されます。

12.2.1. MicroProfile Health エンドポイントおよびアノテーション

MicroProfile Health は、以下の 3 つのエンドポイントを提供します。

- **/health/ready**: マイクロサービスの準備状態を返します。または、要求を処理する準備ができているかどうかを返します。このエンドポイントは Kubernetes の readiness プローブに対応します。

- **/health/live**: マイクロサービスの liveness、またはバグまたはデッドロックが発生したかどうかを返します。このチェックに失敗すると、マイクロサービスは実行されず、停止できます。このエンドポイントは Kubernetes liveness プローブに対応します。これは、チェックに失敗した場合に Pod を自動的に再起動します。

- **/health**: /health **/live および /health/ ready** エンドポイントからの応答を集約します。このエンドポイントは非推奨の **@Health** アノテーションに対応し、MicroProfile 1.0 との互換性を提供する場合にのみ利用できます。MicroProfile 2.0 以降を使用している場合は、代わりに **/health/ready** または **/health/live** エンドポイントを使用してください。

readiness または liveness ヘルスチェックを実装するには、コードに **@Liveness** または **@Readiness** アノテーションを追加します。これらのアノテーションは、提供されたエンドポイントを Kubernetes の liveness および readiness プローブにリンクします。

以下の例は、ヒープメモリーの使用状況をチェックする **@Liveness** アノテーションを示しています。メモリー消費が 90% 未満の場合は、**UP** ステータスを返します。メモリー使用量が 90% を超えると、**DOWN** ステータスが返されます。

```
@Liveness
@ApplicationScoped
public class MemoryCheck implements HealthCheck {
 @Override
 public HealthCheckResponse call() {
      // status is up if used memory is < 90% of max
      MemoryMXBean memoryBean = ManagementFactory.getMemoryMXBean();
      long memUsed = memoryBean.getHeapMemoryUsage().getUsed();
      long memMax = memoryBean.getHeapMemoryUsage().getMax();

      HealthCheckResponse response = HealthCheckResponse.named("heap-memory")
            .withData("used", memUsed)
            .withData("max", memMax)
            .state(memUsed < memMax * 0.9)
            .build();
      return response;
   }
}
```

以下の例は、ヒープメモリー使用量が 90% 未満の場合に **/health/live** エンドポイントからの JSON 応答を示しています。最初のステータスは、エンドポイントから返されるすべてのヘルスチェックの全体的なステータスを示します。2 つ目のステータスは、前述の **name** の値で指定された特定のチェックのステータスを示しています。この例では **ヒープメモリー** です。全体的なステータスが **UP** になるようにするには、エンドポイントで実行されるすべてのチェックに渡す必要があります。

```
{
  "status": "UP",
  "checks": [
    {
      "name": "heap-memory",
```

```
      "status": "UP",
      "data": {
        "used": "1475462",
        "max": ”51681681"
      }
    }
  ]
}
```

以下の例は、利用可能なデータベース接続をチェックする **@Readiness** アノテーションを示しています。接続に成功すると、**UP** ステータスが返されます。接続が利用できない場合は、**DOWN** のステータスを返します。

```
@Readiness
@ApplicationScoped
public class DatabaseReadyCheck implements HealthCheck {

    @Override
    public HealthCheckResponse call() {

        if (isDBConnected()) {
            return HealthCheckResponse.up(“databaseReady”);
        }
        else {
            return HealthCheckResponse.down(“databaseReady”);
        }
    }
}
```

以下の例は、データベース接続が利用できない場合に、**/health/ready** エンドポイントからの JSON 応答を示しています。最初のステータスは、エンドポイントから返されるすべてのヘルスチェックの全体的なステータスを示します。2 つ目のステータスは、前述の **name** の値で指定された特定のチェックのステータスを示しています。この例では databaseReady **です**。エンドポイントで実行されるチェックに失敗した場合、全体のステータスは **DOWN になり** ます。

```
{
  "status": ”DOWN",
  "checks": [
    {
      "name": ”databaseReady",
      "status": ”DOWN",
    }
  ]
}
```

12.2.2. 以下も参照してください。

- ガイド： マイクロサービスへのヘルスレポートの追加
- Guide: Kubernetes でのマイクロサービスの正常性の確認
- MicroProfile Health 機能
- MicroProfile Health on GitHub

付録A 追加の OPEN LIBERTY リソース

Open Liberty およびこれがサポートする API の詳細は、Open Liberty の Web サイトのリソースを参照してください。

- Open Liberty server コマンド
- Open Liberty Guide
- Java EE API
- MicroProfile API