# Open Liberty 2020

## 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: 2020-09-28

# Open Liberty 2020 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

## 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およびマイクロサービスを構築するための軽量 Java ランタイムです。

Open Liberty を使用すると、最新バージョンの 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。このモジュール構造はマイクロサービスの開発を簡素化するため、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 Application Server を実行できます。Open Liberty ゼロマイグレーション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や設定へ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 の Full Platform および Web Profile 仕様、および 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 機能に関する最新更新は、 Open Liberty のブログ および Open Liberty 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 つであり、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。OpenShift で Open Liberty を実行し、OpenShift プラットフォームの利点により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 Open Liberty を使用する方法は、「Deploying microservices to OpenShift」 を参照してください。

## 1.1. OPEN LIBERTY のバージョン設定

従来のバージョン管理では、major.minor.micro スキームの形に従っており、重要な新しい機能はメジャーリリースでのみ提供されます。これらのメジャーリリースには主要な新機能が含まれていますが、アプリケーションの移行と重要なリグレッションテストの採用を必要とする動作の変更も行われます。その結果、常に複数のメジャーバージョンがサポートされます。Open Liberty のモジュラー機能アーキテクチャーは、ゼロマイグレーションと組み合わせて、major.minor.micro バージョン管理スキームに従うことなく新機能を段階的に配信できるようにします。

メジャーリリースではなく、各 Open Liberty 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っています。最初の 2 桁の数字はリリース年を示し、最後の 2 桁の数字は、その年内のリリース番号を示します。最初の桁のセットは毎年変更されますが、リリースは同じ状態です。たとえば、20.0.0.1 (2020 の最初のリリース) と 19.0.0.12 (2019 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ことはサーバーランタイムでは珍しいことですが、デスクトップおよびモバイルアプリケーションでは一般的です。一部の公開システムでは、ソフトウェアにメジャーバージョンが必要です。その結果、メジャーバージョンが必要な場合は、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 の本ガイドに公開されている Open Liberty ドキュメントでは、2019 年がバージョン番号として使用されます。ただし、このドキュメントは、2019 年のリリースとおよび 2020 年のリリースに適用されます。

# 第2章 OPENSHIFT へのデプロイメント

OpenShift は、高可用性かつスケーラブルなアプリケーションのビルド、コンテナー化、デプロイに使用できる 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、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な場合にスケーリングする必要もあります。OpenShift は、開発者やオペレーションチームのニーズに対応するためにコンテナーを自動化し、管理する機能を提供します。

## 2.1. 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、それらを Service (IaaS) ソリューションとしてクラウドホストインフラストラクチャーまたは現在のオンプレミス構造に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、Open Liberty コンテナーのアプリケーションをコンテナー化し、コンテナー化したアプリケーションを OpenShift クラスターにデプロイします。

マイクロサービスを OpenShift にデプロイするステップバイステップのチュートリアルは、「Deploying microservices to OpenShift guide」 を参照してください。Open Liberty のアプリケーションがどのように各種の OpenShift デプロイメントオプションで使用されるかは、OpenShift ドキュメント を参照してください。

## 2.2. OPEN LIBERTY OPERATOR

オペレーターは、Kubernetes または OpenShift が提供する最初の自動化を超えたタスクを自動化するようにカスタマイズされている Kubernetes の拡張機能 です。Open Liberty Operator の機能レベルは 5 です。つまり、自動スケーリング、サービスバインディング、OpenShift 証明書管理の統合、および Kubernetes Application Navigator (kAppNav) の統合など、最高レベルのエンタープライズ機能を備えています。

Open Liberty Operator を使用すると、水平の自動スケーリングを設定することにより、リソースの可用性と消費に基づいてアプリケーションインスタンスを作成または削除することで、アプリケーションを高可用性にすることができます。Operator はアプリケーションデプロイメントの管理にも役立ちます。たとえば、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、Operator デプロイメントファイルの **applicationImage** フィールドを新規コンテナータグで更新します。次に、Operator はアプリケーションをローリングベースで更新します。

このオペレーターは、永続ストレージまたは高度なストレージのシンプルな構成や、シングルサインオン (SSO) を外部プロバイダーに委任する機能 など、他の本番環境レベルの機能も提供します。オペレーターは、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、アプリケーションを接続し、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関する情報を維持します。

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 認定 Operator としても利用できます。

## 2.3. 参照

ガイド: Deploying microservices to Kubernetes

# 第3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する機能の個別単位です。サーバー設定に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することができます。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とサービスを提供します。サーバー設定ファイルで機能を指定できます。一部の機能には、その機能内に他の機能が含まれ、同じ機能が 1 つ以上の他の機能に含まれる場合があります。

サーバーが起動すると JVM が起動し、制御が Open Liberty カーネルに渡されます。「Server configuration overview」で説明されているように、設定が読み込まれます。設定が解析されると、機能マネージャーは制御を取得し、**featureManager** 設定を処理して、要求された機能をサーバーに読み込み、必要なコンポーネントを起動します。最後に、アプリケーションを起動します。設定を変更すると、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開始および停止することにより、新しく要求された機能に必要なコードを再評価します。アプリケーションの変更も同様に処理されます。

## 3.1. 機能の使用

機能は、**server.xml** ファイル、およびその他の include ファイルであるシステム設定ファイルで指定されます。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。以下の例は、**servlet-4.0** および **jdbc-4.3** の機能を設定します。

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

ランタイムには、指定する必要のある設定が最小限に保持されるようにデフォルト設定が含まれます。**server.xml** ファイルで、デフォルト設定の追加またはオーバーライドに必要な機能を指定します。サーバー設定の詳細は「Server configuration overview」を参照してください。

## 3.2. ゼロマイグレーション

Open Liberty のゼロマイグレーション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て、最新バージョンの Open Liberty に移行できます。ゼロマイグレーションアーキテクチャーでは、動作が必要ない、または予期しない変更は行わずに既存の設定ファイルや変更されていない設定ファイルを Open Liberty ランタイム環境の更新バージョンで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と、既存の API および動作が新規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 仕様の両方がサポートされます。API の動作の変更は新機能のバージョンでのみ行われるため、アプリケーションに適切な機能バージョンを選択することができます。これらのバージョン付けされた機能は、Open Liberty の更新後もサポートされます。

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 サーバーには **servlet-3.1** 機能が必要です。Open Liberty を更新し、他の Servlet 仕様レベルがどの程度サポートされるかに関わらず、**servlet-3.1** 機能を無限に使用できます。代わりに **servlet-4.0** 機能を使用する場合のみアプリケーションを移行する必要があります。

## 3.3. 機能の統合

サーバーを異なるバージョンの機能を持つように設定しようとすると、Open Liberty は同じ機能のバージョンを組み合わせていないため、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は、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。サーバー構成にシングルトン機能の複数のバージョンが含まれている場合は、**server.xml** ファイルの直接構成または機能の依存関係により、その構成にエラーがあり、その機能のどちらのバージョンも読み込まれません。この問題を解決するには、設定された機能がすべて、そのシングルトン機能の同じバージョンを指定または許容していることを確認してください。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動する必要があります。

Liberty は、Java EE 7と Java EE 8 仕様がコンポーネント仕様バージョンを共有する場合を除いて、Java EE 7 と Java EE 8 の両方の機能の組み合わせをサポートしていません。サーバー設定で 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 および 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。

## 3.4. 代替機能

機能が置き換えられた場合は、新しい機能、または機能の組み合わせが、代替機能よりも優れている場合があります。新しい機能は、代替機能の機能を完全には置き換えない可能性があるため、構成を変更するかどうかを決定する前に、シナリオを検討する必要があります。代替機能はサポートされ、設定で

使用するために有効ですが、新機能を使用することで設定を改善することができる可能性があります。

他の機能を含む機能は、これらすべての機能が含まれていない新しいバージョンの機能によって置き換えられることがあります。新しいバージョンに含まれない機能は、分離されていると見なされます。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、分離された機能を設定に明示的に追加する必要があります。

以下の表は、置き換えられた Open Liberty 機能を示しています。

| 代替機能 | 代替機能 | 削除される依存機能 |
| --- | --- | --- |
| **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** 機能および **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** 機能および **transportSecurity-1.0** 機能は機能的に同等です。ただし、**ssl-1.0** はセキュアでないネットワークプロトコルが使用されるため、それよりも **transportSecurity-1.0** が優先されます。 |

# 第4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) および任意のファイルセットで構成されます。**server.xml** ファイルは適切に作成された XML であり、ルート要素は **server** 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** ファイルの例では、次のことを実行するようにサーバーを構成します。

```
<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** への着信トラフィックをリッスンします。

3 デプロイ時に自動的に WAR ファイルを展開します。

**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、または特に XML ファイルにある設定を参照するのに使用できます。コンテキストが明確ではない場合は、**サーバーの XML 設定** という用語を使用して XML ファイルで設定を参照できます。

## 4.1. 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** - このファイルに環境変数が指定されます。
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および機能を指定します。

### 4.1.1. 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込まれ、主に **bin/server** スクリプトの動作に影響を与えるために使用され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込まれます。

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は、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。これを設定しないと、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。その他の場所は **usr** ディレクトリーの相対パスであるため、**etc/server.env** ファイルに設定されます。
- **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、サーバーは設定を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。

次の例に示すように、**server.env** ファイルは **KEY=value** 形式です。

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

キー値に空白を含めることはできません。値は文字どおり解釈されるため、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。

### 4.1.2. jvm.options

**jvm.options** ファイルは任意です。このファイルは、**bin/server** シェルスクリプトによって読み込まれ、JVM が Open Liberty に対して起動されるときに使用するオプションを判別します。**jvm.options**ファイルは、次の場所から順番に読み込まれます。

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

このような場所に **jvm.options** ファイルが存在しないと、(そのようなディレクトリーが存在する場合) サーバースクリプトは **${wlp.install.dir}/etc** 内のファイルを探します。

**jvm.options** ファイルの一般的な使用には、以下が含まれます。

- JVM メモリー制限の設定
- モニタリング製品が提供する 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 行使用します。

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

空白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 に順番に読み込まれ、提供されます。複数のオプションを指定すると、すべて JVM によって確認されます。これらのファイルは、変数の置換をサポートしません。

### 4.1.3. 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 ブートストラップ時に読み込まれ、サーバー起動の初期段階の設定を提供します。これは **server.xml** ファイルよりに前のサーバーから読み込まれます。したがって、Open Liberty カーネルの起動および動作に影響を及ぼす可能性があります。**bootstrap.properties** ファイルは簡単な 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、ロギングを設定することです。

**bootstrap.properties** ファイルは、ブートストラップのステージにも読み込まれる別のプロパティーファイルを指定する **bootstrap.include** という特別なオプションのプロパティーをサポートします。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セットを含めることができます。**bootstrap.include** ファイルを絶対または相対ファイルパスに設定します。

### 4.1.4. server.xml

最も重要で唯一必要な設定ファイルは **server.xml** ファイルです。**server.xml** ファイルは適切に作成された XML であり、ルート要素は **server** である必要があります。サーバーでサポートされる正確な要素は、設定されている機能によって異なり、不明な設定は無視されます。

Open Liberty は、例外として構成の原則を使用します。これにより、簡潔な構成ファイルが可能になります。ランタイム環境は、一連の組み込み構成のデフォルト設定で動作します。この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、次の場所から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、その他のファイルは任意です。

サーバー形式の XML ファイルをディレクトリーに置くことで、設定を柔軟に作成できます。ファイルは、2 つの **configDropins** ディレクトリーにそれぞれアルファベット順に読み込まれます。

## 4.2. 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することができます。値への変数参照を解決するために、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たデフォルト値は、他の値が指定されていない場合にのみ使用されます。

```
<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。これを行うと、コマンドラインで指定された変数は、他のすべての変数ソースをオーバーライドし、サーバーの起動時に変更できません。

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 の時点で、環境変数名を直接参照することができます。変数を指定したとおりに解決できない場合、**server.xml** ファイルは環境変数名で以下の差異を探します。

- 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}** と入力すると、環境変数が以下の名前で検索されます。

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 **env.** を環境変数名の開始に追加することで、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、常に単純なタイプ変換を持つ文字列として解釈されます。そのため、ポート一覧 (**80,443** など) は、2 つのポート番号ではなく単一の文字列として解釈されることがあります。以下の例のように **list** 関数を使用すると、**,** 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)}" />
```

整数値を持つ変数では、単純な算術がサポートされています。演算子の左側および右端には、変数または数字のいずれかを指定できます。以下の例のように、演算子は **+**、**-**、*****、または / 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は複数あります。

- **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** - サーバーの名前。
- **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** - サーバーがワークエリア、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。

## 4.3. 設定のマージ

設定は複数のファイルで構成できるため、2 つのファイルが同じ構成を提供する可能性があります。このような状況では、サーバー構成は一連の単純なルールに従ってマージされます。Open Liberty では、設定はシングルトンとファクトリー設定に分離され、マージに関する独自のルールがあります。シングルトン設定は単一の要素 (例: ロギング) を設定するのに使用されます。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、複数のエンティティーを設定するために使用されます。

### 4.3.1. シングルトン設定のマージ

シングルトン設定要素の複数回指定すると、設定はマージされます。2 つの要素が異なる属性に存在する場合は、両方の属性が使用されます。例を以下に示します。

```
<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>
```

同じ属性が 2 回指定されていると、最後のインスタンスが優先されます。例を以下に示します。

```
<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することで提供されることがあります。

このような場合は、指定したすべての値を使用して設定をマージします。最も一般的なシナリオは、機能の設定です。例を以下に示します。

```
<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>
```

### 4.3.2. ファクトリー設定のマージ

ファクトリー設定のマージは、要素名が一致するという理由だけで要素が自動的にマージされないことを除いて、シングルトン構成と同じルールを使用します。ファクトリー設定では、同じ要素の設定が有効で、2 つの異なる論理オブジェクトを意味します。そのため、各要素は個別のオブジェクトを設定することを想定します。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は、**id** 属性が同じであることを示すように各要素に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、このアプリケーションには **myawesomeapp** のコンテキストルートがあります。他のアプリケーションは **myapp2.war** で、コンテキストルートとして **myapp2** があります。

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

## 4.4. INCLUDE の処理

デフォルトの場所に加えて、**include** 要素を使用すると追加の設定ファイルを組み込むことができます。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と、サーバーは参照されたファイルの内容を **include** 要素の代わりにインラインとして処理します。

以下の例では、**other2.xml** ファイルの内容を処理する前に、サーバーは **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、include 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は、以下の例のように **optional** 属性を **true** に設定します。

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

ファイルを指定する場合は、**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** に設定できます。

```
<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ファイルパス、もしくは HTTP URL に設定できます。

## 4.5. 設定参照

Open Liberty のほとんどの構成は自己完結型ですが、設定を共有すると便利なことがよくあります。たとえば、JDBC ドライバー構成は複数のデータソースで共有される場合があります。**server** 要素の直接の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、常に参照される要素の **id** 属性を使用します。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>
```

## 4.6. 動的更新

サーバーは、サーバーの XML 設定で更新の有無を監視し、変更が検出されると動的に再読み込みします。非 XML ファイル (**server.env**、**bootstrap.properties**、および **jvm.options**) への変更は、起動時にのみ読み込ま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、すべて 500ms の更新で監視されます。XML 設定ファイルモニタリングの頻度を設定できます。たとえば、10 分ごとにサーバーを監視するように設定するには、以下を実行します。

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

MBean が通知された場合に限りファイルシステムのポーリングおよび再読み込みを無効にするには、以下を指定します。

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

## 4.7. ログメッセージ

サーバーを実行すると、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。ログの参照では、XPath に似た構造を使用して構成要素を指定します。要素名は、角括弧内の **id** 属性の値で指定します。サーバー設定に **id** が指定されていないと、**id** はが自動的に生成されます。以下のサーバーの XML 設定例に基づいて、**dataStore** 要素と子 **dataSource** は、ログで **dataStore[myDS]** and **dataStore[myDS]/dataSource[default-0]** として識別されます。

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

# 第5章 開発モード

開発モードで Open Liberty を実行すると、統合開発環境 (IDE) またはテキストエディターからアプリケーションを迅速にコーディング、デプロイ、テスト、およびデバッグできます。開発モードと呼ばれる開発モードを有効にして、ビルド自動化ツール Maven または Gradle のいずれかと連携させることができます。

開発モードを使用すると、コードの変更を迅速に繰り返し、オンデマンドまたは自動の単体テストと統合テストから即座にフィードバックを得ることができます。デバッガーを接続して、いつでもコードをステップ実行できるようにすることもできます。開発モードは、Open Liberty Mavenプラグイン の目標として、または the Liberty Gradle プラグイン のタスクとして使用できます。これは Open Liberty の性能のセットを統合し、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ます。開発モードは、変更のデプロイ、テストの実行、およびデバッグの 3 つの主要な焦点分野に対応します。

## 5.1. DEV モードで OPEN LIBERTY を実行します。

開発モードで 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、以下のいずれかのコマンドを実行します。

Maven: **mvn liberty:dev**

Gradle: **gradle libertyDev**

### 5.1.1. コード変更の検出、再コンパイル、デプロイ

開発モードでは、IDE またはテキストエディターで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、およびデプロイできます。dev モードは、以下の変更をアプリケーションソースに自動的に検出します。

- Java ソースファイルおよびテストファイルの変更
- リソースファイルの変更
- 設定ディレクトリーおよび設定ファイルの変更
- Maven ユーザーの **pom.xml** ファイルまたは Gradle ユーザー用の **build.gradle** ファイルへの新しい依存関係の追加
- Open Liberty サーバー設定の新機能の追加

リソースファイル、設定ファイル、および設定ディレクトリーの変更が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、起動しています。

特定の設定ディレクトリーまたはファイルの追加などの一部の変更は、開発モードを再起動するまで有効になりません。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、**CTRL+C** を押して最初に dev モードを終了するか、**q** を押して **Enter** を押します。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、変更が検出され、再コンパイルされ、実行中のサーバーによって取得されます。

開発モードの開始時にパラメーターを指定することにより、開発モードがコードへの変更を処理する方法を構成できます。設定パラメーターの詳細は、「the dev goal of the Open Liberty Maven plug-in」または「the libertyDev task of the Open Liberty Gradle plug-in」 を参照してください。

### 5.1.2. オンデマンドでユニットテストおよび統合テストの実行

開発モードが実行しているコマンドウィンドウで **Enter** を押すと、単体テストと統合テストをオンデマンドで実行できます。開発モードは、プロジェクト用に構成された単体テストと統合テストを実行します。プロジェクトにテストを追加すると、開発モードはコンパイルされ、次にテストを実行するときにそれを含めます。

ホットテストを実行するように開発モードを構成することにより、変更に関する即時フィードバックを得ることができます。ホットテストは、開発モードを開始するかコードを変更するたびに自動的に実行される単体テストまたは統合テストです。ホットテストを構成するには、次の例に示すように、開発モードを開始するとき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**

パラメーターを追加して、テストを省略するかどうかを指定することもできます。Maven では、パラメーターを追加して、ユニットテストの省略、統合テストの省略、またはすべてのテストを省略できます。Gradle の場合は、パラメーターを追加すると、すべてのテストを省略できます。設定パラメーターの詳細は、「the dev goal of the Open Liberty Maven plug-in」 または「the libertyDev task of the Open Liberty Gradle plug-in」を参照してください。

### 5.1.3. 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接続して、いつでもコードをステップ実行できるようにすることができます。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パーツをローカルにデバッグできます。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使用できない場合、開発モードは、デバッグ用のポートとして使用するランダムなポートを選択します。

### 5.1.4. dev モードの VS コード拡張

Open Liberty Tools VS Code 拡張機能 を使用すると、VS Code エディターから離れることなく、開発モードの開始、動的なコード変更、テストの実行、アプリケーションのデバッグを行うことができます。拡張機能をインストールし、Maven または Gradl eプラグインを有効にした後、VSCode 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てメニューからコマンドを選択すると、開発モードの機能にアクセスできます。

## 5.2. 参照

- demo-devmode サンプルプロジェクト (Maven ユーザーおよび Gradle ユーザー)
- ガイド: Getting started with Open Liberty (Maven ユーザー)

# 第6章 スレッドプールのチューニング

Open Liberty は、スレッドプールのサイズを制御するセルフチューニング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、**coreThreads** 属性と **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セルフチューニングコントローラーで設定され、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するスレッドのセットです。Open Liberty は、OSGi フレームワークの提供、JVM ガベージの収集、Java NIO トランスポート機能の提供などのタスクに他のスレッドを使用します。しかし、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、ほとんどのスレッドは設定できません。

## 6.1. スレッドプールのチューニング動作

Open Liberty スレッドプールのセルフ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一連のデータを維持します。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の数によって決まります。コントローラーは、以前に試行し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。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され、プールの最適なサイズを決定します。サイクルごとに、プールサイズは徐々に増減したりせず、そのまま残ります。

場合によっては、決定を導くための履歴データがありません。たとえば、プールが各サイクルで拡大し、現在のサイクルがこれまでの最大サイズであれば、大きなプールサイズのスループットに関するデータが存在しません。このような場合、コントローラーはプールのサイズを大きくする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に再調整します。このプロセスは、さまざまなサイズのスレッドプールを試してパフォーマンスを確認し、設定とワークロードの最適値を決定するヒューマンスレッドプールチューナーに似ています。

各 1.5 秒のサイクル中に、スレッドプールコントローラーは次の自動調整操作を実行します。

1. アクティブになり、プール内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していると見なします。この場合、コントローラーは設定で指定され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. 最新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、各プールサイズに対する重み付けされた移動平均として記録されます。このパフォーマンスは過去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迅速に調整されます。

3. 履歴データを使用して、プールサイズが小さいか、大きいかを予測します。より小さいプールサイズまたは大きなプールサイズの履歴データがない場合、スレッドプールコントローラーはプールのサイズを増減するかどうかを決定します。

4. 予測されるパフォーマンスに基づいて、設定で指定された範囲内のプールサイズを増減するか、そのままにします。

5. 再び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。プールサイズと確認されるスループットの関係は、完全にスムーズまたは連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導出される予測を改善するために、コントローラーは、最も近い大きいプールサイズと小さいプールサイズだけでなく、両方向の増分もいくつか考慮します。

### 6.1.1. 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が、実行する前に他の作業が完了するのを待つ必要が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、サーバーは特定のプールサイズでハングできます。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。

ハングの解決により、プールにスレッドが追加され、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が再度完了したことを確認すると、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。

## 6.2. スレッドプールの手動チューニング

ほとんどの環境、設定、およびワークロードでは、Open Liberty スレッドプールを手動で設定またはチューニングする必要がありません。スレッドプールはセルフチューニングされ、最適なサーバースループットを提供するために必要なスレッドの数を決定します。スレッドプールコントローラーは、**coreThreads** 属性および **maxThreads** 属性の定義範囲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、**coreThreads** 属性または **maxThreads** 属性の設定が必要になる場合があります。以下のセクションでは、これらの属性を説明し、手動での調整が必要になる可能性がある条件の例を提供します。

- **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、スレッド数がこの属性の値に等しくなるまで、提供される各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と、デフォルトでは、Open Liberty プロセスで利用できるハードウェアスレッド数の倍数に設定されます。
  Open Liberty が共有環境で稼働している場合、スレッドプールコントローラーは利用可能な CPU を共有する他のプロセスに対応できません。この場合、**coreThreads** 属性のデフォルト値により、Open Liberty は CPU リソースと競合している他のプロセスを考慮して、最適なスレッド数よりも多くの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、**coreThreads** 属性を、Open Liberty が実行する必要のある CPU リソースの割合のみを反映する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 で、**MAX_INT** に相当するか、実質的に無制限になります。
  一部の環境では、プロセスを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 には、そのような上限が適用されるかどうか、またはその値が何かを知る方法がありません。Open Liberty がスレッド制限の環境で実行している場合、Operator は **maxThreads** 属性を許容値として設定できます。

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と設定を処理するように設計されています。場合によっては、**coreThreads** 属性および **maxThreads** 属性の調整が必要になる場合があります。ただし、最初にデフォルトの動作を試して、調整を行う必要があることを確認します。

## 6.3. 参照

エグゼキューター管理

# 第7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によって書き込まれるメッセージを処理し、FFDC (First Failure Data Capture) 機能を提供する統合ロギングコンポーネントがあります。**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、3 つの主なログファイルがあります。

- **console.log** - このファイルは、**server start** コマンドで作成されます。これには、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のストリームが含まれます。このコンソール出力は人が直接使用するためのものであるため、自動ログ分析に役立つ情報が不足しています。

- **messages.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まれたり、取得された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たすべてのメッセージには、メッセージのタイムスタンプや、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が含まれません。

- **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まれるか、または取得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する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が含まれません。

## 7.1. 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で制御できます。ロギングコンポーネントは、**logging** 要素を使用すると **server.xml** に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** 経由でロギングを設定すると、ログエントリーは後で別のログ設定を使用することがあります。このため、**boostrap.properties** を使用し、場合によっては環境変数を使用してロギング設定を行うこともできます。

## 7.2. 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紹介します。

### 7.2.1. ログファイルストレージの管理

**console.log** ファイルは、プロセスの **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** に提供するログロールオーバーと同様、同じレベルの管理を提供できません。**console.log** ファイルのサイズの増加を懸念する場合は、**console.log** ファイルを無効にし、代わりに **messages.log** ファイルを使用することができます。**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、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** が 100Mb で 3 回ロールオーバー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
```

### 7.2.2. 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は、JSON 形式を使用してログファイルを保存すると便利です。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 json 形式を設定する場合、一部のログ行は **server.xml** の起動前にデフォルトの JSON 以外の形式で書き込まれるため、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しません。

### 7.2.3. Docker イメージのログの設定

Docker 環境では、**messages.log** を無効にし、コンソールの出力を JSON としてフォーマットするのが一般的です。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のみ設定できます。

```
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
```

### 7.2.4. バイナリーロギング

Liberty には、トレースファイルを書き込むオーバーヘッドを大幅に削減する高性能のバイナリーログ形式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、**console.log** はパフォーマンスを最適化するために無効にされます。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を使用すると、バイナリーログをテキストファイルに変換できます。

```
binaryLog view defaultServer
```

## 7.3. ソースによる構成設定

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。設定に関するドキュメントは、logging 要素の設定リファレンスに記載されています。

| サーバーの XML 属性 | bootstrap プロパティー | 環境変数 | 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メッセージキーを設定できます。メッセージが非表示にな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 | この属性を使用すると、**console.log** ファイル (FFDC を含む) 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。デフォルトは **WLP_OUTPUT_DIR/serverName/logs** です。**server.xml** が読み込まれる前にログデータがデフォルトの場所に書き込まれてしまう可能性があるため、**server.xml** に **logDirectory** を設定することは推奨されません。 |
| コンソールログの設定 | | | |
| 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。 |

| サーバーの XML 属性 | bootstrap プロパティー | 環境変数 | 説明 |
| --- | --- | --- | --- |
| consoleLogLevel | com.ibm.ws.logging.console.log.level | WLP_LOGGING_CONSOLE_LOGLEVEL | このフィルターは、コンソールに移動するメッセージの粒度を制御します。有効な値は INFO、AUDIT、WARNING、ERROR、および OFF です。デフォルトは AUDIT です。Eclipse 開発者ツールで使用する場合は、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コ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。**audit** 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。 |

| サーバーの XML 属性 | bootstrap プロパティー | 環境変数 | 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、プロセスの **stdout** および **stderr** にコピーされるため、**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** または **trace.log** などの設定済みログに書き込まれますが、**stdout** や **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。 |
| メッセージログの設定 | | | |
| | com.ibm.ws.logging.newLogsOnStart | | Liberty の起動時に true に設定すると、既存の **messages.log** ファイルまたは **trace.log** ファイルはロールオーバーされ、新しい **messages.log** ファイルまたは **trace.log** ファイルに書き込みが記録されます。false に設定すると、**messages.log** ファイルまたは trace.log ファイルは、**maxFileSize** に達したとき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ブートストラップでのみ処理されるため、**server.xml** の **logging** 要素を使用して指定することはできません。 |

| サーバーの XML 属性 | bootstrap プロパティー | 環境変数 | 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。<br><br>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。<br><br>**false** の値を指定すると、日付と時刻はシステムに設定されたデフォルトのロケールに応じて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 | | 保持される各ログファイルの数。この設定は、FFDC の例外サマリーログの数にも適用されます。したがって、この数が **10** の場合は、**ffdc/** ディレクトリーにメッセージログを 10 個、トレースログを 10 個、例外サマリーを 10 個を持つことができます。デフォルトでは、値は **2** です。**console.log** はロールしないため、この設定は適用されません。 |
| maxFileSize | com.ibm.ws.logging.max.file.size | | ログファイルがロールされる前に到達できる最大サイズ (MB)。値を **0** に設定すると、ログの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** はロールしないため、この設定は適用されません。 |

| サーバーの XML 属性 | bootstrap プロパティー | 環境変数 | 説明 |
| --- | --- | --- | --- |
| 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、**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) が含まれています。このログには、タイムスタンプと発行スレッド 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、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。 |
| 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コ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、**messageSource** は **message** に設定されます。**audit** 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 | | | |

| サーバーの XML 属性 | bootstrap プロパティー | 環境変数 | 説明 |
| --- | --- | --- | --- |
| 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際に機密データを公開できます。**true** に設定すると、機密性の高い情報がログやトレースファイルに公開されなくなります。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な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。 |
| 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。 |
| 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のに使用されます。ログの詳細レベルの仕様の形式:<br><br>component = level<br><br>ここでの **component** は **level** を設定するログソースを指定し、**level** は、出力されるトレースのサイズを指定します。指定するレベルは、**off**、**fatal**、**severe**、**warning**、**audit**、**info**、**config**、**detail**、**fine**、**finer**、**finest**、**all** の中から 1 つを選択します。コロンで区切ることで、複数のログ詳細レベル指定を提供できます。 |

| サーバーの XML 属性 | bootstrap プロパティー | 環境変数 | 説明 |
| --- | --- | --- | --- |
| | | | コンポーネントはロガー名、トレースグループ、またはクラス名にすることができます。アスタリスク * はワイルドカードとして動作し、接頭辞に基づいて複数のコンポーネントと照合します。例を以下に示します。<br>● * 製品システムコードや顧客コードなど、アプリケーションサーバーで実行しているすべての追跡可能なコードを指定します。<br>● **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。<br>● **com.ibm.ws.classloading.AppClassLoader** AppClassLoader クラスのみを指定します。 |

# 第8章 実稼働環境向け統合テスト

MicroShed Testing は、実稼働環境に似た環境でアプリケーションに対して実際に実稼働環境向けの統合テストを作成するのに役立つ Java ライブラリーです。開発と実稼働環境間の同等性の問題を最小限に抑えるために、実稼働で使用するのと同じ Docker コンテナーでアプリケーションをテストできます。

MicroShed テスト 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 テストを使用して、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実稼働環境で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とコンポーネントを評価しますが、統合テストでは単体テストよりもセットアップおよび設定に時間がかかります。または、単体テストの方が短く、アプリケーションの個々のモジュールをテストする必要があります。開発サイクルが短く、利用可能になる時間が短縮され、開発者は通常、ユニットテストを実行します。MicroShed テストは、アプリケーションに対して実稼働環境向け統合テストを作成して実行し、Testcontainers フレームワークを使用して統合テストを合理化して、効率的なワークフローを実現できます。

## 8.1. 開発と実稼働の同等性

開発および実稼働環境の同等性 は、12 要素のアプリケーション の要素の 1 つです。これは、最新のアプリケーションを構築するための方法論です。開発および実稼働環境の同等性に関する概念は、時間、人員、およびツールについて、開発環境、ステージング環境、実稼働環境を同等に保つことです。開発プロセスを簡素化するために、開発者は多くの場合、実稼働環境とは異なる開発で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中のアプリケーションのプロジェクトをビルドすることはできますが、アプリケーションは実稼働環境の Docker コンテナーにデプロイされる可能性があります。開発環境ではテストに合格しますが、環境間の違いにより、本番環境ではテストが失敗する可能性があります。MicroShed テストは、本番環境と同様の環境でアプリケーションをテストすることで、開発および実稼働環境の同等性を実現するのに役立ちます。

## 8.2. MICROSHED TESTING ライブラリーを使用した統合テストの作成

コンテナーを使用して、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 の開発者が実稼働環境と似た環境でアプリケーションをテストするための Java ライブラリーです。Microshed Testing は Testcontainers フレームワークを実装し、テストで開発と実稼働環境の同等性をサポートします。

MicroShed テストでは、以下の例のように統合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、リポジトリーで 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、テストを開始する前にアプリケーションが準備状態になるのを待ちます。**@Container** アノテーションは、実行中のアプリケーションコンテナーに、HTTP 要求を送信しやすくする PersonService クラスの REST クライアントプロキシーを挿入します。**@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。これにより、PersonService#createPerson エンドポイントがトリガーされ、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、作成したレコードを読み込むことができます。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、存在しない **-1** ID の Person オブジェクトを見つけます。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。

## 8.3. 参照

ガイド: Testing a MicroProfile or Jakarta EE application

# 第9章 デバッグ

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

アプリケーションは、単一のモジュールを使用した単純なアプリケーションであるか、または複数のモジュールで構成される複雑なアプリケーションであっても、ビルドおよびテストできます。

プロジェクトの詳細と依存関係を定義したら、Maven はすべての依存関係を自動的に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の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に合格しないと、ビルドは失敗します。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、「Building a web application with Maven」 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ビルドする方法は、「Creating a multi-module application」 を参照してください。

## 9.2. 要求のトレース

分散トレースは、分散システムを介して伝播する要求を調べて記録することにより、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、このような要求のデバッグという難しいタスクに取り組むことができます。分散トレースシステムを導入しないと、ワークフローを分析して、いつ誰がリクエストを受信したか、いつ誰がレスポンスを返したかを特定することは困難です。

アプリケーション内のマイクロサービス間でのロギング要求を監視およびトレースする方法は、「Enabling distributed tracing in microservices」 を参照してください。

# 第10章 OPEN LIBERTY のモニタリング

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および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にして報告すると、問題を特定し、容量計画のデータを収集し、サービスを拡大または縮小するタイミングを決定することができます。

## 10.1. マイクロサービスでのモニタリングの有効化

監視可能性をマイクロサービスに組み込むと、システムの内部ステータスが外部化され、オペレーションチームがマイクロサービスシステムをより効果的に監視できるようになります。実稼働環境でマイクロサービスが実行しているときにオペレーションチームが使用できるメトリクスを生成するようにマイクロサービスを作成することが重要になります。

メトリクスは、さまざまな場所から作成されます。アプリケーション、Open Liberty ランタイム、および Java 仮想マシンから取得できます。MicroProfile Metrics は、Open Liberty サーバーおよびデプロイされたアプリケーションが生成したすべてのメトリクスにアクセスできる **/metrics** エンドポイントを提供します。これらは、Prometheus などのデータベースツールに収集および保存でき、Grafana などのダッシュボードに表示されます。

メトリクスには、カウンター、ゲージ、タイマー、ヒストグラム、メーターなど、さまざまな形式があります。MicroProfile Metrics 機能を使用して Open Liberty アプリケーションでメトリクスを有効にすることができます。

利用可能な Open Liberty メトリクスの一覧は、「metrics reference list」を参照してください。

MicroProfile Metrics を使用してマイクロサービスからメトリクスを有効にして提供する方法は、「Providing metrics from a microservice」を参照してください。

**追加リソース**

- メトリクスによるマイクロサービスの可観測性

## 10.2. マイクロサービスでのヘルスチェックの有効化

ヘルスチェックは、マイクロサービスおよび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 です。MicroProfile Health を使用すると、アプリケーション内のサービスがそれぞれのヘルスをセルフチェックし、全体的なヘルスステータスを定義済みのエンドポイントに公開できます。

セルフチェックは、次のようなサービスが必要とするすべてのものを評価するために使用できます。

- 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

MicroProfile Health を使用すると、liberty アプリケーションのサービスを有効にして、liveness および readiness をセルフチェックできます。liveness チェックは、サービスでバグまたはデッドロックが発生したかどうかを決定します。このチェックに失敗すると、サービスは実行されず、終了されます。こ

のチェックは Kubernetes liveness プローブに対応し、チェックが失敗すると Pod を自動的に再起動します。readiness チェックは、サービスがリクエストを処理する準備ができているかどうかを判断します。このチェックは、Kubernetes の readiness プローブに対応します。

MicroProfile Health を使用してマイクロサービスヘルスチェックを有効にして報告する方法は、「Adding health reports to microservices」 を参照してください。

**参考情報**

- マイクロサービスのヘルスチェック

# 付録A OPEN LIBERTY リソースの追加

Open Liberty の Web サイトでリソースを表示すると、Open Liberty とそれがサポートする API の詳細を確認できます。

- Open Liberty サーバーコマンド
- Open Liberty ガイド
- Java EE API
- MicroProfile API