Open Liberty 2021

Open Liberty Runtime Guide

Build and deploy cloud native applications with Open Liberty

Last Updated: 2021-01-26

Open Liberty 2021 Open Liberty Runtime Guide

Build and deploy cloud native applications with Open Liberty

Legal Notice

Abstract

These topics provide a brief introduction to Open Liberty and links to in-depth documentation resources.

Table of Contents

CHAPTER 1. WHAT IS OPEN LIBERTY

Open Liberty is a lightweight Java runtime for building cloud-native applications and microservices.

With Open Liberty, it's easy to add and remove modular features from the latest versions of Jakarta EE and Eclipse MicroProfile. This modular structure simplifies microservice development, enabling you to run Just enough Application Server to support the features that your application needs. Furthermore, with Open Liberty zero migration architecture, you can upgrade to the latest version with minimal impact to your current applications and configurations. Open Liberty is compatible with the Jakarta EE 8 Full Platform and Web Profile specifications and with MicroProfile 3.0. For more information, see the Open Liberty website. For the latest updates about Open Liberty features and capabilities, see the Open Liberty blog and Open Liberty docs.

Open Liberty is one of the Java runtimes available on OpenShift, and support is provided as part of a Red Hat subscription. Run Open Liberty on OpenShift to build and deploy cloud-native applications with the benefits of the OpenShift platform. For more information about using Open Liberty with OpenShift, see Deploying microservices to OpenShift.

1.1. OPEN LIBERTY VERSIONING

Traditional versioning follows some variation of the major.minor.micro scheme, where significant new function is only delivered in a major release. These major releases contain key new capabilities, but they also make behavior changes that require application migration and significant regression testing to adopt. As a result, multiple major versions are supported at any one time. The modular feature architecture of Open Liberty, in combination with zero migration, allows the delivery of new function incrementally, without following a major.minor.micro versioning scheme.

Instead of major releases, each Open Liberty release is considered a micro, or patch release. These patch releases follow a yy.0.0.nn version scheme. The first two digits indicate the year of release and the last two digits indicate the number of the release within that year. Even though the first set of digits changes each year, the releases are of equal standing. For example, the difference between 20.0.0.1 (the first release of 2020) and 19.0.0.12 (the last release of 2019) is identical to the difference between 19.0.0.10 and 19.0.0.11.

The lack of major release streams is unusual for server runtimes, but is common for desktop and mobile applications. Some publication systems expect software to have a major version. As a result, in cases where a major version is needed, the year of publication is used as a stand-in. For example, Open Liberty documentation that is published for this guide in 2019 uses the year 2019 as the version number. However, this documentation is as applicable to releases in 2020 as it is to releases in 2019.

CHAPTER 2. ZERO-MIGRATION ARCHITECTURE

Zero-migration architecture is a core design principle of Open Liberty, which supports full compatibility between runtime versions. With zero-migration architecture, you can move to the latest version of Open Liberty with minimal impact to your current applications and configurations.

One of the major challenges for teams that work with runtime servers is the need to continually update to the latest release of the runtime. These updates are often required to resolve security vulnerabilities or bugs that can cause outages. Updating the runtime version can be difficult because new releases also introduce API or behavior changes, sometimes for functions that are not critical for your particular applications. Both the Open Liberty runtime and Open Liberty features are released in numbered versions. Changes in behavior and API support are delivered in new feature versions, which you can decide whether to adopt according to your needs. With zero migration, your existing APIs and behaviors are supported in new runtime versions, and new APIs and behaviors are added in new features and feature versions. Your existing, unmodified configuration and application files work with an updated version of Open Liberty, without unexpected changes in application behavior.

2.1. USER CONFIGURATION FILES

If all the configured features are installed, a single set of configuration files works across multiple versions of Open Liberty without modifications. User configuration files do not require modification when the runtime is updated to a new version. Open Liberty ignores any configuration settings that do not apply to the active version of the runtime.

2.2. OPEN LIBERTY FEATURES

Open Liberty features are encoded into the server configuration, which does not change during an upgrade. When behavior changes are introduced, such as when a Java specification mandates a behavior change, the change is introduced in a new version of the feature. Existing applications and their configuration can continue to use the old behavior with the older feature version, while new applications can use newer feature versions with the new behavior.

For example, Open Liberty supports both the Servlet 3.1 and Servlet 4.0 specifications. The Servlet 4.0 specification includes clarifications to previous servlet versions that can result in changes in application behavior. Open Liberty restricts any such behavior changes to the **servlet-4.0** feature and retains the existing behavior in the **servlet-3.1** feature. If your application is configured for the Servlet 3.1 specification, you can continue to use the **servlet-3.1** feature across runtime updates, regardless of how many other servlet specification levels are supported. You don't need to change your application unless you decide to configure the **servlet-4.0** feature instead. For more information, see Feature overview.

2.2.1. Considerations for using new features

Upgrading to a new version of a feature that you are already using might impact your existing applications. For example, if you currently use the **servlet-3.1** feature and you want to use the **servlet-4.0** feature, your existing servlet applications might require modification to work correctly with the **servlet 4.0** feature. Alternatively, you can keep the application in a server that is configured with the original version and create a different server configuration with the new feature version for new applications.

Some features interact closely with other features and require that you enable a particular feature version when they are both configured in the same server. Some features require specific versions of prerequisite software, for example, the Java SDK.

2.3. EXCEPTIONS TO ZERO MIGRATION

Although Open Liberty is designed to ensure that no breaking changes occur between runtime versions, in some rare cases they are unavoidable. Such exceptions to zero migration fall into one of the following categories:

- Security fixes
  If your application requires a security-related fix that cannot be safely implemented in the context of existing behavior, then you might need to modify your application or configuration.

- Third-party API requirements
  Open Liberty does not control APIs from the third-party class loader configuration. As a result, updates to third-party components are not guaranteed to be compatible with an earlier version of the runtime.

- Undocumented configuration properties
  Some configuration options for Open Liberty are not part of the runtime documentation. These configuration options can be determined by looking at the source code, or through trial and error, based on external sources of information. If options are not documented as a part of Open Liberty, then they are not considered part of Open Liberty. Such options might not be fully implemented and might cause issues if they are used. Since they are not documented, they can be removed or changed at any time and are not covered by zero migration.

- Incompatible Java changes
  Historically, new Java SE versions make few incompatible changes to the Java language. In the rare case that a breaking change is included in a new Java version, Open Liberty attempts to minimize the impact of these changes on applications. However, these attempts might not always be successful and breaking Java changes can sometimes adversely affect your application.

Zero-migration architecture saves developers and enterprises time and money by avoiding the need to migrate existing configuration and application files. Developers can focus on their applications, rather than managing runtime updates, while they continue to benefit from improved performance and administration for their existing server configurations.

CHAPTER 3. DEPLOYMENT ON OPENSHIFT

OpenShift is a Kubernetes-based container application platform that you can use to build, containerize, and deploy your applications so that they're highly available and scalable. You can also use the Open Liberty Operator to simplify deploying and managing your applications in the cloud.

After you develop and containerize your applications, your containers need to communicate with other containers that are running databases, security systems, or other microservices. Containers also need to scale as services are needed. OpenShift provides features that automate and manage containers to meet the needs of developers and operations teams.

3.1. RUNNING APPLICATIONS ON OPENSHIFT

One of the benefits of running your applications on OpenShift is that you can deploy them to a cloud-hosted Infrastructure as a Service (IaaS) solution, or to your current on-premises structure. You can use the OpenShift CLI or OpenShift Do CLI to develop your applications. Then, containerize applications in Open Liberty containers and deploy them to an OpenShift cluster.

For a step-by-step tutorial on deploying microservices to OpenShift, see the Deploying microservices to OpenShift guide. To learn more about how your applications on Open Liberty can be used with the different OpenShift deployment options, see the OpenShift documentation.

3.2. OPEN LIBERTY OPERATOR

Operators are extensions to Kubernetes that are customized to automate tasks beyond the initial automation that Kubernetes or OpenShift provide. The Open Liberty Operator has a capability level of five, which means that it has the highest level of enterprise capabilities, including auto-scaling, service binding, OpenShift certificate management integration, and Kubernetes Application Navigator (kAppNav) integration.

With the Open Liberty Operator, you can make applications highly available by configuring horizontal auto-scaling, which creates or deletes application instances based on resource availability and consumption. The Operator also helps manage application deployments. For example, after you upload a new container tag for a new version of an application, update the **applicationImage** field in the Operator deployment file with the new container tag. Then, the Operator updates the application on a rolling basis.

This Operator also offers other production-grade capabilities, including simple configuration of persistent or advanced storage and the ability to delegate single-sign on (SSO) to external providers . The Operator automates updates of binding information among applications, meaning that it connects applications and maintains information about whether a particular application produces or consumes a service.

You can install the Open Liberty Operator from OperatorHub for use on Kubernetes or OpenShift. The Operator is also available as a Red Hat-certified Operator from OpenShift Container Platform (OCP).

3.3. SEE ALSO

Guide: Deploying microservices to Kubernetes

CHAPTER 4. FEATURE OVERVIEW

Features are the discrete units of functionality by which you control the pieces of the runtime environment that are loaded into a particular server. By adding or removing features from your server configuration, you can control what functions the server can perform. Features provide the programming models and services that applications require. You can specify any feature in the server configuration files. Some features include other features within them, and the same feature might be included in one or more other features.

When the server is started, the JVM is launched and control is passed to the Open Liberty kernel. The configuration is loaded as described in the Server configuration overview. When the configuration is parsed, the feature manager gains control and processes the **featureManager** configuration to load the requested features into the server and start the required components. Finally, the applications are started. When the configuration is changed, the feature manager reevaluates the code that is required for the newly requested features by starting and stopping parts of the runtime as necessary without restarting the server. Changes to applications are processed in a similar way.

4.1. USING FEATURES

Features are specified in the system configuration files that are the **server.xml** file and any other included files. The feature manager is configured by using the **featureManager** element in the **server.xml** file. Each feature that is required is configured by using the **feature** element. The following example configures the **servlet-4.0** and **jdbc-4.3** features:

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

The runtime contains default configuration settings so that the configuration you need to specify is kept to a minimum. You specify the features that you need, along with any additions or overrides to the default settings, in the **server.xml** file. For details about the server configuration, see the Server configuration overview.

4.2. ZERO-MIGRATION

With the Open Liberty zero-migration architecture, you can move to the latest version of Open Liberty with minimal impact to your current applications and configurations. Zero-migration architecture means that you can use existing, unmodified configuration and application files with an updated version of the Open Liberty runtime environment without unwanted or unexpected change in behavior.

With the use of pluggable features in the Open Liberty runtime environment, your existing APIs and behaviors are supported in new product versions, and new APIs and behaviors are added in new features. For example, both the Servlet 3.1 and 4.0 specifications are supported. Changes in API behavior only happen in new feature versions, so you can choose the appropriate feature version for your application. These versioned features continue to be supported across Open Liberty updates.

If you continue to use the same feature version, you never need to migrate your application. For example, if your application uses Servlet 3.1, the Open Liberty server that runs the application must have the **servlet-3.1** feature. You can update Open Liberty and continue to use the **servlet-3.1** feature indefinitely, regardless of how many other Servlet specification levels are supported. You need to migrate your applications only if you choose to use the **servlet-4.0** feature instead.

4.3. COMBINING FEATURES

If you try to configure a server to have different versions of a feature, an error is reported because Open Liberty doesn't support combining different versions of the same feature. This means that most Open Liberty features are singleton features. A singleton feature is a feature for which you can configure only one version for use in a server.

If you have applications that need different versions of the singleton feature, you must deploy them in different servers. If your server configuration includes multiple versions of a singleton feature, either through direct configuration in the **server.xml** file, or through feature dependencies, that configuration is in error and neither version of that feature is loaded. To resolve this problem, ensure that the configured features all specify, or tolerate, the same version of that singleton feature. If you have hard requirements on both feature versions, you must move some of your applications to a different server.

Liberty doesn't support combining features from both Java EE 7 and Java EE 8, except when the Java EE 7 and Java EE 8 specifications share a component specification version. If you combine Java EE 7 and Java EE 8 features in a server configuration, the server reports errors at startup.

The following features are included in both Java EE 7 and Java EE 8:

- 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

For a complete list of features that support Java EE 7, see the javaee-7.0 feature. For a complete list of features that support Java EE 8, see the javaee-8.0 feature.

4.4. SUPERSEDED FEATURES

If a feature is superseded, a new feature or a combination of features might provide an advantage over the superseded feature. The new feature or features might not completely replace the function of the

superseded feature, so you must consider your scenario before you decide whether to change your configuration. Superseded features remain supported and valid for use in your configuration, but you might be able to improve your configuration by using the newer features.

Occasionally, a feature that includes other features is superseded by a new version of the feature that does not include all those features. The features that are not included in the new version are considered to be separated. If your application depends on the functions of a separated feature, you must explicitly add the separated feature to your configuration.

The following table lists the Open Liberty features that are superseded:

Superseded feature	Superseding feature	Dependent feature removed
appSecurity-1.0	**appSecurity-2.0**	The ldapRegistry and the **servlet-3.0** feature were removed from the definition of the **appSecurity-2.0** feature.
jmsMdb-3.2	**jms-2.0** and **mdb-3.2**	Together, the **jms-2.0** and **mdb-3.2** features provide the same function as the **jmsMdb-3.2** feature.
ssl-1.0	**transportSecurity-1.0**	The **ssl-1.0** and **transportSecurity-1.0** features are functionally equivalent. However, **ssl-1.0** implies that an insecure network protocol is used, so **transportSecurity-1.0** supersedes it.

CHAPTER 5. SERVER CONFIGURATION OVERVIEW

The Open Liberty server configuration is made up of one mandatory file, the **server.xml** file, and a set of optional files. The **server.xml** file must be well-formed XML and the root element must be **server**. When the **server.xml** file is processed, any elements or attributes that are not understood are ignored.

This example **server.xml** file configures the server to do the following things:

```
<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. Support the JavaServer Pages 2.3 feature
2. Listen to incoming traffic to **localhost** on port **9080**
3. Automatically expand WAR files when they are deployed

The term *server configuration* can be used to refer to all of the files that make up the server configuration or specifically to the configuration that's in the XML files. If it's not clear in context, the term *server XML configuration* might be used to refer to the configuration in the XML files.

5.1. CONFIGURATION FILES

The server configuration files are processed in the following order:

1. **server.env** - Environment variables are specified in this file.
2. **jvm.options** - JVM options are set in this file.
3. **bootstrap.properties** - This file influences the startup of the Open Liberty server.
4. **server.xml** - This mandatory file specifies the server configuration and features.

5.1.1. server.env

The **server.env** files are optional. These files are read by the **bin/server** shell script and specify environment variables that are primarily used to influence the behavior of the **bin/server** script. **server.env** files are read from the following locations in order:

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

If the same property is set in multiple locations, then the last value found is used.

The most common use of these files is to set the following environment variables:

- **JAVA_HOME** - Indicates which JVM to use. If this is not set, the system default is used.
- **WLP_USER_DIR** - Indicates the location of the **usr** directory that contains the server configuration This can only be set in the **etc/server.env** file because the other locations are relative to the **usr** directory.
- **WLP_OUTPUT_DIR** - Indicates where the server writes files to. By default, the server writes to the directory structure that the configuration is read from. However, in some secure profiles the server configuration needs to be read-only so the server must write files to another location.

The **server.env** file is in **KEY=value** format, as shown in the following example:

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

Key values must not contain spaces. The values are interpreted literally so you don't need to escape special characters, such as spaces. These files don't support variable substitution.

5.1.2. jvm.options

The **jvm.options** files are optional. These files are read by the **bin/server** shell script to determine what options to use when the JVM is launched for Open Liberty. **jvm.options** files are read from the following locations in order:

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

If no **jvm.options** files exist in these locations, then the server script looks for the file in **${wlp.install.dir}/etc**, if such a directory exists.

Common uses of **jvm.options** files include:

- Setting JVM memory limits
- Enabling Java Agents that are provided by monitoring products
- Setting Java System Properties

The **jvm.options** file format uses one line per JVM option, as shown in the following example:

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

You don't need to escape special characters, such as spaces. Options are read and provided to the JVM in order. If you provide multiple options, then they are all seen by the JVM. These files do not support variable substitution.

5.1.3. bootstrap.properties

The **bootstrap.properties** file is optional.

This file is read during Open Liberty bootstrap to provide configuration for the earliest stages of the server startup. It is read by the server earlier than the **server.xml** file so it can affect the startup and behavior of the Open Liberty kernel from the start. The **bootstrap.properties** file is a simple Java properties file and is located in **${server.config.dir}**. A common use of the **bootstrap.properties** file is to configure logging because it can affect logging behavior before the **server.xml** file is read.

The **bootstrap.properties** file supports a special optional property, **bootstrap.include**, which specifies another properties file to also be read during the bootstrap stage. For example, this **boostrap.include** file can contain a common set of bootstrap properties for multiple servers to use. Set the **bootstrap.include** file to an absolute or relative file path.

5.1.4. server.xml

The most important and only required configuration file is the **server.xml** file. The **server.xml** file must be well-formed XML and the root element must be **server**. The exact elements that are supported by a server depend on which features are configured, and any unknown configuration is ignored.

Open Liberty uses a principle of configuration by exception, which allows for succinct configuration files. The runtime environment operates from a set of built-in configuration default settings. You only specify configuration that overrides those default settings.

Server configuration files are read from the following locations in order:

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

The **${server.config.dir}/server.xml** file must be present, but the other files are optional.

You can flexibly compose configuration by dropping server-formatted XML files into directories. Files are read in alphabetical order in each of the two **configDropins** directories.

5.2. VARIABLE SUBSTITUTION

You can use variables to parameterize the server configuration. To resolve variable references to their values, the following sources are consulted in order:

1. **server.xml** file default variable values
2. environment variables
3. **bootstrap.properties**
4. Java system properties
5. **server.xml** file configuration
6. variables declared on the command line

Variables are referenced by using the **${variableName}** syntax.

Specify variables in the server configuration as shown in the following example:

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

Default values, which are specified in the **server.xml** file, are used only if no other value is specified:

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

You can also specify variables at startup from the command line. If you do, the variables that are specified on the command line override all other sources of variables and can't be changed after the server starts.

Environment variables can be accessed as variables. As of version 19.0.0.3, you can reference the environment variable name directly. If the variable cannot be resolved as specified, the **server.xml** file looks for the following variations on the environment variable name:

- Replace all non-alphanumeric characters with the underscore character (_)
- Change all characters to uppercase

For example, if you enter **${my.env.var}** in the **server.xml** file, it looks for environment variables with the following names:

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

For versions 19.0.0.3 and earlier, you can access environment variables by adding **env.** to the start of the environment variable name, as shown in the following example:

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

Variable values are always interpreted as a string with simple type conversion. Therefore, a list of ports (such as **80,443**) might be interpreted as a single string rather than as two port numbers. You can force the variable substitution to split on the **,** by using a **list** function, as shown in the following example:

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

Simple arithmetic is supported for variables with integer values. The left and right sides of the operator can be either a variable or a number. The operator can be **+**, **-**, *****, or /, as shown in the following example:

```
<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}" />
```

There are a number of predefined variables:

- **wlp.install.dir** - the directory where the Open Liberty runtime is installed.
- **wlp.server.name** - the name of the server.

- **wlp.user.dir** - the directory of the **usr** folder. The default is **${wlp.install.dir}/usr**.
- **shared.app.dir** - the directory of shared applications. The default is **${wlp.user.dir}/shared/apps**.
- **shared.config.dir** - the directory of shared configuration files. The default is **${wlp.user.dir}/shared/config**.
- **shared.resource.dir** - the directory of shared resource files. The default is **${wlp.user.dir}/shared/resources**.
- **server.config.dir** - the directory where the server configuration is stored. The default is **${wlp.user.dir}/servers/${wlp.server.name}**.
- **server.output.dir** - the directory where the server writes the workarea, logs, and other runtime-generated files. The default is **${server.config.dir}**.

5.3. CONFIGURATION MERGING

Since the configuration can consist of multiple files, it is possible that two files provide the same configuration In these situations, the server configuration is merged according to a set of simple rules. In Open Liberty, configuration is separated into singleton and factory configuration each of which has its own rules for merging. Singleton configuration is used to configure a single element (for example, logging). Factory configuration is used to configure multiple entities, such as an entire application or data source.

5.3.1. Merging singleton configuration

For singleton configuration elements that are specified more than once, the configuration is merged. If two elements exist with different attributes, both attributes are used. For example:

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

is treated as:

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

If the same attribute is specified twice, then the last instance takes precedence. For example:

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

is treated as:

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

Configuration is sometimes provided by using child elements that take text.

In these cases, the configuration is merged by using all of the values specified. The most common scenario is configuring features. For example:

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

is treated as:

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

5.3.2. Merging factory configuration

Factory configuration merges use the same rules as singleton configuration except elements are not automatically merged just because the element names match. With factory configuration it is valid to configure the same element and mean two different logical objects. Therefore, each element is assumed to configure a distinct object. If a single logical object is configured by two elements, the **id** attribute must be set on each element to indicate they are the same thing. Variable substitution on an **id** attribute is not supported.

The following example configures two applications. The first application is **myapp.war**, which has a context root of **myawesomeapp**. The other application is **myapp2.war**, which has **myapp2** as the context root:

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

5.4. INCLUDE PROCESSING

In addition to the default locations, additional configuration files can be brought in by using the **include** element. When a server configuration file contains an include reference to another file, the server processes the contents of the referenced file as if they were included inline in place of the **include** element.

In the following example, the server processes the contents of the **other.xml** file before it processes the contents of the **other2.xml** file:

```
<server>
```

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

By default, the include file must exist. If the include file might not be present, set the **optional** attribute to **true**, as shown in the following example:

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

When you include a file, you can specify the **onConflict** attribute to change the normal merge rules. You can set the value of the **onConflict** attribute to **IGNORE** or **REPLACE** any conflicting config:

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

You can set the **location** attribute to a relative or absolute file path, or to an HTTP URL.

5.5. CONFIGURATION REFERENCES

Most configuration in Open Liberty is self-contained but it is often useful to share configuration For example, the JDBC driver configuration might be shared by multiple data sources. You can refer to any factory configuration element that is defined as a direct child of the **server** element.

A reference to configuration always uses the **id** attribute of the element that is being referenced. The configuration element that makes the reference uses an attribute that always ends with **Ref**, as shown in the following example:

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

5.6. DYNAMIC UPDATES

The server monitors the server XML configuration for updates and dynamically reloads when changes are detected. Changes to non-XML files (**server.env**, **bootstrap.properties**, and **jvm.options**) are not dynamic because they are only read at startup. Any server XML configuration file on the local disk is monitored for updates every 500ms. You can configure the frequency of XML configuration file monitoring. For example, to configure the server to monitor every 10 minutes, specify:

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

To disable file system polling and reload only when an MBean is notified, specify:

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

5.7. LOG MESSAGES

When the server runs, it might output log messages that reference configuration. The references in the log use an XPath-like structure to specify configuration elements. The element name is given with the value of the **id** attribute inside square brackets. If no **id** is specified in the server configuration an **id** is automatically generated. Based on the following server XML configuration example, the **dataStore** element and the child **dataSource** are identified in the logs as **dataStore[myDS]** and **dataStore[myDS]/dataSource[default-0]**.

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

CHAPTER 6. OPEN LIBERTY OPERATOR

Operators are extensions to Kubernetes that are customized to automate tasks beyond the initial automation that Kubernetes or OpenShift provides. The Open Liberty Operator helps you deploy and manage applications on Kubernetes-based clusters.

6.1. WHAT IS THE OPEN LIBERTY OPERATOR?

When you deploy an application with the Open Liberty Operator, the operator watches Open Liberty resources and compares the current state of resources to the state of resources that you configured. When a discrepancy exists between the current state of resources and the state that you configured, the operator creates, updates, or deletes Kubernetes resources to return to the state that you configured. These Kubernetes resources might include deployments, services, or routes. Without the operator, you must manually create deployments, services, routes, and other Kubernetes resources, which can involve a time-consuming learning curve. With the operator, you can specify details for your application, including your application image, service port, and whether to expose the application outside the cluster. Then, the operator creates and manages all Kubernetes resources. Now, you manage only a single **OpenLibertyApplication** resource instead of many resources. In addition, the operator continuously monitors the events that are related to the application in the cluster and takes necessary actions to synchronize data and resources. Because the operator helps you manage Kubernetes resources, you can focus on your application while the operator handles many of the cloud deployment details.

The Open Liberty Operator is based on the Runtime Component Operator, which is a generic operator that can be imported into runtime-specific operators to provide standardized enterprise capabilities. As such, common functionality exists between these two operators, such as the use of image streams and the ability to run multiple instances of your application for high availability. When you use the Open Liberty Operator, the operator container and controller are deployed to a Kubernetes pod, and the operator listens for incoming resources with the **kind: OpenLibertyApplication** statement. When you create an **OpenLibertyApplication** custom resource (CR), the operator manages the Kubernetes resources that the application needs to run on your cluster.

6.2. OPERATOR CAPABILITIES

The Open Liberty Operator has a capability level of five, which means that it has the highest level of enterprise capabilities, including the following capabilities:

- **High availability that's provided by horizontal auto-scaling**
  You can configure horizontal auto-scaling to create and delete instances of your application based on resource consumption. This ability to run multiple instances of your application and auto-scale them means that your application is made highly available.

- **Enhanced deployment management**
  With the Open Liberty Operator, you can more easily manage applications that are deployed to Kubernetes. For example, in the operator deployment file, you can specify an image stream in the **applicationImage** field. Then, after you upload a new container tag for a new version of an application, the operator updates the application on a rolling basis.

- **Automated service binding**
  The operator automates updates of binding information among applications, meaning that it connects applications and maintains information about whether a particular application produces or consumes a service. With this information, the operator automatically handles Kubernetes-level details, including creating and injecting Kubernetes Secrets, so that your applications can connect to required services without interruption.

- **Single sign-on (SSO) delegation**
  With Open Liberty, you can delegate SSO authentication to external providers . The Open Liberty Operator enables easier configuration and management of SSO information for your applications.

- **OpenShift Serverless (Knative) integration**
  You can integrate the operator with Knative. When the operator is integrated with Knative, you deploy your serverless runtime component by using a single toggle. The operator converts all of its generated resources into Knative resources, which allows your pod to automatically scale to zero when it's idle.

- **Kubernetes Application Navigator (kAppNav) integration**
  The operator can automatically configure integration with kAppNav. With this integration, you can monitor resources of your application and receive alerts when the health status of a component changes. From an integrated pane, you can also access operations-focused capabilities, such as enabling trace for a component and viewing Kibana or Grafana dashboards.

- **OpenShift certificate management integration**
  The operator takes advantage of the cert-manager tool, if it's installed on the Kubernetes cluster. The cert-manager tool allows the operator to automatically provision Transport Layer Security (TLS) certificates for pods and routes. Certificates are mounted into containers from a Kubernetes Secret so that the certificates are automatically refreshed when they're updated.

6.3. SERVICEABILITY AND OBSERVABILITY WITH THE OPEN LIBERTY OPERATOR

You can enable persistence for your application by specifying only the size of storage and where you want the storage to be mounted. Then, the operator creates and manages the storage claim for you. An advanced mode is available that allows the configuration of extra details of the persistent volume claim. You can also configure and use a single storage for serviceability-related operations, such as gathering server memory dumps and server traces. For more information about gathering server memory dumps and server traces, see Day-2 operations.

After you configure the operator, you can integrate Open Liberty with logging and monitoring tools for observability in the Kubernetes cluster. You can select different types of Open Liberty data that you want to monitor. To visualize and track logging events, deploy one of the provided Open Liberty Kibana dashboards. You can monitor Open Liberty metrics by using MicroProfile Metrics, Prometheus, and Grafana to gather, scrape, and visualize metric data. You can also enable MicroProfile Health to perform liveness checks and readiness checks so that Kubernetes can check the health of your containers.

6.4. OPERATOR INSTALLATION AND CONFIGURATION

You can install the Open Liberty Operator from OperatorHub for use on Kubernetes or OpenShift. The operator is also available as a Red Hat-certified operator from OpenShift Container Platform (OCP). The Open Liberty Operator documentation provides details about configuring the operator, including basic configuration, Custom Resource Definition (CRD) parameters, Open Liberty console logging environment variables, and persistent storage specifications.

6.5. SEE ALSO

- Guide: Deploying an application to OpenShift by using Kubernetes Operators
- Troubleshooting the Open Liberty Operator

CHAPTER 7. DEVELOPMENT MODE

When you run Open Liberty in development mode, you can rapidly code, deploy, test, and debug applications directly from your integrated development environment (IDE) or text editor. You can enable development mode, known as dev mode, to work with either Maven or Gradle build automation tools.

With dev mode, you can quickly iterate on changes to your code and get immediate feedback from on-demand or automatic unit and integration tests. You can also attach a debugger to step through your code at any time. Dev mode is available as a goal of the Open Liberty Maven plug-in or as a task of the Liberty Gradle plug-in. It integrates a set of capabilities for Open Liberty so that you can edit and monitor your application in real time, without restarting your running server. Dev mode addresses three primary focus areas: deploying changes, running tests, and debugging.

7.1. RUN OPEN LIBERTY IN DEV MODE

To run Open Liberty in dev mode, enable the Open Liberty Maven plug-in or the Open Liberty Gradle plug-in and run one of the following commands:

Maven: **mvn liberty:dev**

Gradle: **gradle libertyDev**

7.1.1. Detect, recompile, and deploy code changes

Dev mode can automatically detect, recompile, and deploy code changes whenever you save a new change in your IDE or text editor. Dev mode automatically detects the following changes to your application source:

- Java source file and test file changes
- Resource file changes
- Configuration directory and configuration file changes
- New dependency additions to your **pom.xml** file for Maven users or **build.gradle** file for Gradle users
- New feature additions in the Open Liberty server configuration

Resource file, configuration file, and configuration directory changes are copied into your target directory. New dependencies in your **pom.xml** file or **build.gradle** file are added to your class path. New features are installed and started.

Some changes, such as adding certain configuration directories or files, do not take effect until you restart dev mode. To enable these changes, restart dev mode when prompted. To restart, first exit dev mode by pressing **CTRL+C**, or by typing **q** and pressing **Enter**. Then, run the **mvn liberty:dev** command or the **gradle libertyDev** command to restart. After the server restarts, the changes are detected, recompiled, and picked up by the running server.

You can configure how dev mode handles changes to your code by specifying parameters when you start dev mode. For more information about configuration parameters, see the dev goal of the Open Liberty Maven plug-in or the libertyDev task of the Open Liberty Gradle plug-in .

7.1.2. Run unit and integration tests on demand

You can run unit and integration tests on demand by pressing **Enter** in the command window where dev mode is running. Dev mode runs the unit tests and integration tests that are configured for your project. If you add a test to your project, dev mode compiles and includes it the next time that you run tests.

You can get immediate feedback on your changes by configuring dev mode to run hot tests. Hot tests are unit or integration tests that run automatically whenever you start dev mode or make a code change. To configure hot testing, specify the hot test parameter when you start dev mode, as shown in the following examples:

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

Gradle: **gradle libertyDev --hotTests**

You can also add parameters to specify whether to skip tests. For Maven, you can add parameters to skip unit tests, skip integration tests, or skip all tests. For Gradle, you can add a parameter to skip all tests. For more information about configuration parameters, see the dev goal of the Open Liberty Maven plug-in or the libertyDev task of the Open Liberty Gradle plug-in .

7.1.3. Attach a debugger to the running server

You can attach a debugger to the running server to step through your code at any time. You can specify breakpoints in your source code to locally debug different parts of your application. The default port for debugging is **7777**. If the default port is not available, dev mode selects a random port to use as the port for debugging.

7.1.4. VS Code extension for dev mode

With the Open Liberty Tools VS Code extension , you can start dev mode, make dynamic code changes, run tests, and debug your application, all without leaving the VS Code editor. After you install the extension and enable either the Maven or Gradle plug-in, you can select your project under the Liberty Dev Dashboard in the VS Code side bar. You can access dev mode functions by right-clicking your project name and selecting a command from the menu.

7.2. SEE ALSO

- The demo-devmode sample project (Maven and Gradle users)
- Guide: Getting started with Open Liberty (Maven users)

CHAPTER 8. THREAD POOL TUNING

Open Liberty provides a self-tuning algorithm that controls the size of its thread pool. Although you do not need to manually tune the thread pool for most applications, in some cases you might need to adjust the **coreThreads** and **maxThreads** attributes.

All the application code in Open Liberty runs in a single thread pool that is called the default executor. The size of this pool is set by a self-tuning controller, which can manage a wide range of workloads. The default executor pool is the set of threads where your application code runs. Open Liberty uses other threads for tasks like serving the OSGi framework, collecting JVM garbage, and providing Java NIO transport functions. However, these threads are not directly relevant to application performance and most of them are not configurable.

8.1. THREAD POOL TUNING BEHAVIOR

The Open Liberty thread pool self-tuning process runs every 1.5 seconds. The thread pool controller maintains a set of data about the thread pool performance from the time the server started. The throughput is determined by the number of tasks that are completed by the controller each cycle. The controller records the throughput for the various pool sizes that were previously tried. This historical throughput data is then compared to throughput for the current cycle to decide the optimal pool size. At each cycle, the pool size can be incrementally increased or decreased, or left unchanged.

In some cases, no historical data is available to guide the decision. For example, if the pool is growing with each cycle and the current cycle is at the largest size so far, no data exists about throughput for larger pool size. In such a case, the controller decides at random whether to increase the size of the pool. Then, it readjusts for the next cycle based on the results of that decision. This process is analogous to a human thread pool tuner who tries various thread pool sizes to see how they perform and decides on an optimal value for the configuration and workload.

During each 1.5-second cycle, the thread pool controller runs through the following self-tuning operations:

1. Wakes up and checks whether the threads in the pool are hung. If tasks are in the queue and no tasks were completed in the previous cycle, the controller considers the threads to be hung. In this case, the controller increases the thread pool size as specified by settings and skips to step 5.

2. Updates the historical data set with the number of tasks that completed in the most recent controller cycle. Performance is recorded as a weighted moving average for each pool size. This performance reflects historical results but adjusts quickly to changing workload characteristics.

3. Uses historical data to predict whether performance would be better at smaller or larger pool size. If no historical data exists for the smaller or larger pool size, the thread pool controller decides whether to increase or shrink the size of the pool.

4. Increases or decreases the pool size within the bounds that are specified in settings, or leaves it unchanged, based on predicted performance.

5. Goes back to sleep.

Various factors other than the thread pool size can affect throughput in the Open Liberty server. The relationship between pool size and observed throughput is not perfectly smooth or continuous. Therefore, to improve the predictions that are derived from the historical throughput data, the controller considers not just the closest larger and smaller pool size, but also several increments in each direction.

8.1.1. Hang resolution

In some application scenarios, all the threads in the pool can become blocked by tasks that must wait for other work to finish before they can run. In these cases, the server can become hung at a certain pool size. To resolve this situation, the thread pool controller enters a hang resolution mode.

Hang resolution adds threads to the pool to allow the server to resume normal operation. Hang resolution also shortens the controller cycle duration to break the deadlock quickly.

When the controller observes that tasks are being completed again, normal operation resumes. The controller cycle returns to its normal duration, and pool size is adjusted based on the usual throughput criteria.

8.2. MANUAL THREAD POOL TUNING

In most environments, configurations, and workloads, the Open Liberty thread pool does not require manual configuration or tuning. The thread pool self-tunes to determine how many threads are needed to provide optimal server throughput. The thread pool controller continually adjusts the number of threads in the pool within the defined bounds for the **coreThreads** and **maxThreads** attributes. However, in some situations, setting the **coreThreads** or **maxThreads** attributes might be necessary. The following sections describe these attributes and provide examples of conditions under which they might need to be manually tuned.

- **coreThreads**
  This attribute specifies the minimum number of threads in the pool. The minimum value for this attribute is 4. Open Liberty creates a new thread for each piece of offered work until the number of threads equals the value of this attribute. If the **coreThreads** attribute is not configured, it defaults to a multiple of the number of hardware threads available to the Open Liberty process.
  If Open Liberty is running in a shared environment, the thread pool controller cannot account for other processes with which it is sharing the available CPUs. In these cases, the default value of the **coreThreads** attribute might cause Open Liberty to create more threads than is optimal, considering the other processes that are competing for CPU resources. In this situation, you can limit the **coreThreads** attribute to a value that reflects only the proportion of the CPU resources that Open Liberty needs to run.

- **maxThreads**
  This attribute specifies the maximum number of threads in the pool. The default value is -1, which is equal to **MAX_INT**, or effectively unlimited.
  Some environments set a hard limit on the number of threads that a process can create. Currently, Open Liberty has no way to know whether such a cap applies, or what the value is. If Open Liberty is running in a thread-limited environment, the operator can configure the **maxThreads** attribute to an acceptable value.

The Open Liberty thread pool controller is designed to handle a wide range of workloads and configurations. In some edge cases, you might need to adjust the **coreThreads** and **maxThreads** attributes. However, try the default behavior first to make sure you need to make adjustments.

8.3. SEE ALSO

Executor Management

CHAPTER 9. LOGGING AND TRACING

Open Liberty has a unified logging component that handles messages that are written by applications and the runtime, and provides First Failure Data Capture (FFDC) capability. Logging data written by applications using **System.out**, **System.err**, or **java.util.logging.Logger** are combined into the server logs.

There are three primary log files for a server:

- **console.log** - This file is created by the **server start** command. It contains the redirected standard output and standard error streams from the JVM process. This console output is intended for direct human consumption so lacks some information useful for automated log analysis.
- **messages.log** - This file contains all messages that are written or captured by the logging component. All messages that are written to this file contain additional information such as the message time stamp and the ID of the thread that wrote the message. This file is suitable for automated log analysis. This file does not contain messages that are written directly by the JVM process.
- **trace.log** - This file contains all messages that are written or captured by the logging component and any additional trace. This file is created only if you enable additional trace. This file does not contain messages that are written directly by the JVM process.

9.1. LOGGING CONFIGURATION

The logging component can be controlled through the server configuration. The logging component can be fully configured in **server.xml** using the **logging** element. However, logging is initialized before **server.xml** has been processed so configuring logging through **server.xml** can result in early log entries using a different log configuration from later ones. For this reason it is also possible to provide much of the logging configuration using **boostrap.properties** and in some cases using environment variables.

9.2. EXAMPLE LOGGING CONFIGURATION

Some common logging configuration examples are given in the following sections.

9.2.1. Managing log file storage

The **console.log** file is created by redirecting the process **stdout** and **stderr** to a file. As a result, Liberty is unable to offer the same level of management, like log rollover, as it offers for **messages.log**. If you are concerned about the increasing size of the **console.log** file, you can disable the **console.log** file and use the **messages.log** file instead. All the log messages sent to **console.log** are written to the **messages.log** file, and you can configure file rollover.

To disable the console log, and configure **messages.log** to roll over three times at 100Mb, use the following configuration:

```
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 logging

When feeding log files into modern log aggregation and management tools it can be advantageous to have the log files stored using JSON format. This can be done in one of three ways:

- Using the **bootstrap.properties** file:

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

- Using environment variables:

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

- Using the **server.xml** file:

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

When using **server.xml** to configure json format some log lines are written in the default non-JSON format prior to **server.xml** startup which can cause issues with some tools. For example, **jq** would have trouble understanding the log files.

9.2.3. Configuring logging for a Docker image

It is common in Docker environments to disable **messages.log** and instead format the console output as JSON. This can be done using environment variables:

```
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
```

This can be simply set when running the **docker run** command by using **-e** to set the envrionment variables:

```
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. Binary logging

Liberty has a high performance binary log format option that significantly reduces the overhead of writing trace files. Generally, when configuring binary logging, the **console.log** is disabled for best performance. This must be enabled using **bootstrap.properties**:

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

The **binaryLog** command line tool can be used to convert the binary log to a text file:

binaryLog view defaultServer

9.3. CONFIGURATION SETTINGS BY SOURCE

The table below shows the equivalent **server.xml**, **bootstrap.properties**, and environment variable configurations along with brief descriptions. Full configuration documentation is available in the config reference for the logging element.

Server XML Attribute	bootstrap property	Env var	Description
hideMessage	com.ibm.ws.logging.hideMessage		You can use this attribute to configure the messages keys that you want to hide from the **console.log** and **messages.log** files. If the messages are configured to be hidden, then they are redirected to the **trace.log** file.
logDirectory	com.ibm.ws.logging.log.directory	LOG_DIR	You can use this attribute to set a directory for all log files, excluding the **console.log** file, but including FFDC. The default is **WLP_OUTPUT_DIR/serverName/logs**. It is not recommended to set the **logDirectory** in **server.xml** since it can result in some log data being written to the default location prior to **server.xml** being read.
Console Log Config			
consoleFormat	com.ibm.ws.logging.console.format	WLP_LOGGING_CONSOLE_FORMAT	The required format for the console. Valid values are **basic** or **json** format. By default, **consoleFormat** is set to **basic**.

Server XML Attribute	bootstrap property	Env var	Description
consoleLogLevel	com.ibm.ws.logging.console.log.level	WLP_LOGGING_CONSOLE_LOGLEVEL	This filter controls the granularity of messages that go to the console. The valid values are INFO, AUDIT, WARNING, ERROR, and OFF. The default is AUDIT. If using with the Eclipse developer tools this must be set to the default.
consoleSource	com.ibm.ws.logging.console.source	WLP_LOGGING_CONSOLE_SOURCE	The list of comma-separated sources that route to the console. This property applies only when **consoleFormat="json"**. Valid values are **message**, **trace**, **accessLog**, **ffdc**, and **audit**. By default, **consoleSource** is set to **message**. To use the **audit** source, enable the Liberty audit-1.0 feature. To use the **accessLog** source you need to have configured httpAccessLogging.
copySystemStreams	com.ibm.ws.logging.copy.system.streams		If true, messages that are written to the System.out and System.err streams are copied to process **stdout** and **stderr** and so appear in **console.log**. If false, those messages are written to configured logs such as **messages.log** or **trace.log**, but they are not copied to **stdout** and **stderr** and do not appear in **console.log**. The default value is true.
Message Log Config			

Server XML Attribute	bootstrap property	Env var	Description
	com.ibm.ws.logging.newLogsOnStart		If set to true when Liberty starts, any existing **messages.log** or **trace.log** files are rolled over and logging writes to a new **messages.log** or **trace.log** file. If set to false **messages.log** or trace.log files only refresh when they hit the **maxFileSize**. The default is **true**. This setting cannot be provided using the **logging** element in **server.xml** because it is only processed during server bootstrap.
isoDateFormat	com.ibm.ws.logging.isoDateFormat		Specifies whether to use ISO-8601 formatted dates in log files. The default value is false. If set to true, the ISO-8601 format is used in the **messages.log** file, the **trace.log** file, and the FFDC logs. The format is **yyyy-MM-dd'T'HH:mm:ss.SSSZ**. If you specify a value of **false**, the date and time are formatted according to the default locale set in the system. If the default locale is not found, the format is **dd/MMM/yyyy HH:mm:ss:SSS z**.

Server XML Attribute	bootstrap property	Env var	Description
maxFiles	com.ibm.ws.logging.max.files		How many of each of the logs files are kept. This setting also applies to the number of exception summary logs for FFDC. So if this number is **10**, you might have 10 message logs, 10 trace logs, and 10 exception summaries in the **ffdc/** directory. By default, the value is **2**. The **console.log** does not roll so this setting does not apply.
maxFileSize	com.ibm.ws.logging.max.file.size		The maximum size (in MB) that a log file can reach before it is rolled. Setting the value to **0** disables log rolling. The default value is **20**. The **console.log** does not roll so this setting does not apply.
messageFileName	com.ibm.ws.logging.message.file.name		The message log has a default name of **messages.log**. This file always exists, and contains INFO and other (AUDIT, WARNING, ERROR, FAILURE) messages in addition to **System.out** and **System.err**. This log also contains time stamps and the issuing thread ID. If the log file is rolled over, the names of earlier log files have the format **messages_timestamp.log**

Server XML Attribute	bootstrap property	Env var	Description
messageFormat	com.ibm.ws.logging.message.format	WLP_LOGGING_MESSAGE_FORMAT	The required format for the **messages.log** file. Valid values are **basic** or **json** format. By default, **messageFormat** is set to **basic**.
messageSource	com.ibm.ws.logging.message.source	WLP_LOGGING_MESSAGE_SOURCE	The list of comma-separated sources that route to the **messages.log** file. This property applies only when **messageFormat="json"**. Valid values are **message**, **trace**, **accessLog**, **ffdc**, and **audit**. By default, **messageSource** is set to **message**. To use the **audit** source, enable the Liberty audit-1.0 feature. To use the **accessLog** source you need to have configured httpAccessLogging.
Trace Config			
suppressSensitiveTrace			The server trace can expose sensitive data when it traces untyped data, such as bytes received over a network connection. This attribute, when set to **true**, prevents potentially sensitive information from being exposed in log and trace files. The default value is **false**.

Server XML Attribute	bootstrap property	Env var	Description
traceFileName	com.ibm.ws.logging.trace.file.name		The **trace.log** file is only created if additional or detailed trace is enabled. **stdout** is recognized as a special value, and causes trace to be directed to the original standard out stream.
traceFormat	com.ibm.ws.logging.trace.format		This attribute controls the format of the trace log. The default format for Liberty is **ENHANCED**. You can also use **BASIC** and **ADVANCED** formats.
traceSpecification	com.ibm.ws.logging.trace.specification		The trace string is used to selectively enable trace. The format of the log detail level specification: component = level where **component** specifies what log sources the **level** should be set to, and **level** specifies how much trace should be output using one of: **off**, **fatal**, **severe**, **warning**, **audit**, **info**, **config**, **detail**, **fine**, **finer**, **finest**, **all**. Multiple log detail level specifications can be provided by separating them with colons. A component can be a logger name, trace group or class name. An asterisk * acts as a wildcard to match multiple components based on a prefix. For example: ● * Specifies all traceable code

Server XML Attribute	bootstrap property	Env var	Description
			that is running the application server, including the product system code and customer code. ● **com.ibm.ws.*** Specifies all classes with the package name beginning with com.ibm.ws. ● **com.ibm.ws.classloading.AppClassLoader** Specifies the AppClassLoader class only.

CHAPTER 10. TRUE-TO-PRODUCTION INTEGRATION TESTING

MicroShed Testing is a Java library that helps you write true-to-production integration tests for your application in an environment similar to production. To minimize the parity issues between development and production, you can test your application in the same Docker container that you use in production.

MicroShed Testing uses the Testcontainers framework to analyze your application from outside the Docker container without accessing the application internals. You can use MicroShed Testing to develop integration tests for your Open Liberty application.

You need to make sure that your application works accurately in production, similar to the development environment. Integration tests assess multiple test classes and components, but integration tests take longer to set up and configure than unit tests. Alternatively, unit tests are shorter and involve testing individual modules of an application. With shorter development cycles and limited time available, developers often run unit tests. MicroShed Testing helps you write and run true-to-production integration tests for your applications, and streamlines your integration tests with the Testcontainers framework for an efficient workflow.

10.1. DEVELOPMENT AND PRODUCTION PARITY

Development and production parity is one of the factors in the twelve-factor app, which is a methodology to build modern applications. The idea behind development and production parity is to keep the development, staging, and production environments similar, regarding time, personnel, and tools. To simplify the development process, developers often use tools in development that are different from production. For example, you might use a local Maven build to build a project for your application in development, but the application might be deployed to a Docker container in production. Differences between the environments can cause a test to fail in production, though the test passes in the development environment. MicroShed Testing helps achieve development and production parity by testing the application in an environment similar to production.

10.2. WRITING INTEGRATION TESTS WITH THE MICROSHED TESTING LIBRARY

You can deploy your applications in different environments with containers. With the Testcontainers framework, you can use containers in a test environment. Microshed Testing is a Java library for MicroProfile and Jakarta EE developers to test their applications in an environment similar to production. Microshed Testing implements the Testcontainers framework to support development and production parity in testing.

With MicroShed Testing, you can write an integration test that looks like the following example:

```
@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));
    }
}
```

The **@MicroShedTest** annotation searches for a Dockerfile document in the repository, starts up the application in a Docker container, and waits for the application to be ready before the test starts. The **@Container** annotation injects a REST Client proxy of the PersonService class, which helps you send HTTP requests on the running application container. The **@RESTClient** annotation sends an HTTP POST request to the running container, which triggers the PersonService#createPerson endpoint and returns the generated ID. By using the generated ID, you can send an HTTP GET request to read the record that was created. The JSON response automatically converts to a Person object by using a JSON-B token. The **@Test** annotation sends an HTTP GET request to find a Person object with the **-1** ID, which does not exist. The annotation asserts that the application container returns an HTTP 404 exception.

10.3. SEE ALSO

Guide: Testing a MicroProfile or Jakarta EE application

CHAPTER 11. DEBUGGING

11.1. MANAGING BUILD PROCESSES WITH THE OPEN LIBERTY MAVEN PLUG-IN

You can build and test your applications, whether they are simple applications with a single module or more complex applications that consist of multiple modules.

After you define the details and dependencies of a project, Maven automatically downloads and installs all of the dependencies. It also runs automated tests on an application after it is built. If the tests don't pass after you update an application, the build fails. You must fix your code.

The following coordinates for the Maven plug-in are required:

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

To learn how to configure a simple web servlet application by using Maven and the Liberty Maven plug-in, see Building a web application with Maven .

Jakarta EE applications consist of multiple modules that work together as one entity. To learn how to build an application with multiple modules by using Maven and Open Liberty, see Creating a multi-module application.

11.2. TRACING REQUESTS

Distributed tracing helps you troubleshoot microservices by examining and logging requests as they propagate through a distributed system, allowing developers to tackle the otherwise difficult task of debugging these requests. Without a distributed tracing system in place, it's difficult to analyze workflows and pinpoint when and by whom a request is received or when a response is returned.

To learn how to monitor and trace logging requests across microservices in an application, see Enabling distributed tracing in microservices.

CHAPTER 12. MONITORING OPEN LIBERTY

You can use MicroProfile Metrics and MicroProfile Health to monitor microservices and applications that run on Open Liberty. Enabling and reporting metric and health check data for your microservices helps you pinpoint issues, collect data for capacity planning, and decide when to scale a service up or down.

12.1. MONITORING WITH METRICS

With Open Liberty, two types of metrics are available to monitor your applications, REST endpoint-style metrics that are provided by MicroProfile Metrics, and Java Management Extensions (JMX) metrics.

MicroProfile Metrics can be accessed by monitoring tools, such as Prometheus, and by any client that's capable of making REST requests. JMX metrics are suitable for use by Java-based monitoring tools that can communicate with JMX servers, or by custom JMX clients.

12.1.1. MicroProfile Metrics and the /metrics endpoint

The MicroProfile Metrics feature provides a /**metrics** REST interface that conforms to the MicroProfile Metrics specification. Open Liberty uses MicroProfile Metrics to expose metrics that describe the internal state of many Open Liberty components. Developers can also use the MicroProfile Metrics API to expose metrics from their applications.

Metrics come in various forms that include counters, gauges, timers, histograms, and meters. You can access MicroProfile Metrics by enabling the MicroProfile Metrics feature . Real-time values of all metrics are available by calling the /**metrics** endpoint. For more information about adding these metrics to your applications, see Microservice observability with metrics. For a list of all REST endpoint-style metrics that are available for Open Liberty, see the metrics reference list.

The /**metrics** endpoint provides two output formats. The format that each response uses depends on the HTTP accept header of the corresponding request. Prometheus format is a representation of the metrics that is compatible with the Prometheus monitoring tool, which is an open source metrics scraper, data store, and basic visualization tool. This format is returned for requests with a **text/plain** accept header. JSON format is a JSON representation of the metrics. This format is returned for requests with an **application/json** accept header.

The following table displays the different endpoints that can be accessed to provide metrics in Prometheus or JSON format with a GET request:

Table 12.1. Metrics endpoints that are accessible by GET request

Endpoint	Request type	Supported formats	Description
/metrics	GET	Prometheus, JSON	Returns all registered metrics.
/metrics/<scope>	GET	Prometheus, JSON	Returns metrics that are registered for the specified scope.

Endpoint	Request type	Supported formats	Description
/metrics/<scope>/<metric name>	GET	Prometheus, JSON	Returns metrics that match the metric name for the specified scope.

12.1.2. JMX metrics

You can access JMX metrics by enabling the Performance Monitoring feature. After you add this feature to your server configuration, JMX metrics are automatically monitored. The Performance Monitoring feature provides MXBeans that you can use with monitoring tools that use JMX, such as JConsole.

JConsole is a graphical monitoring tool that you can use to monitor JVM and Java applications. After you enable monitoring for Open Liberty, you can use JConsole to connect to the JVM and view performance data by clicking attributes of the MXBeans. You can also use other products that consume JMX metrics to view your metrics information. For a list of all JMX metrics that are available for Open Liberty, see the JMX metrics reference list.

12.1.3. Both types of metrics combined

The MicroProfile Metrics feature provides basic metrics about the JVM and about metrics that are added to applications by using the MicroProfile Metrics API. However, the MicroProfile Metrics feature doesn't provide metrics about Open Liberty server components unless it's used along with the Performance Monitoring feature.

If you enable both the MicroProfile Metrics and Performance Monitoring features, then the MXBeans for monitoring and the **/metrics** REST endpoint are activated. In this case, metrics for Open Liberty server components are exposed through both interfaces. The MicroProfile Metrics feature version 2.3 and later automatically enables the Performance Monitoring feature.

12.2. HEALTH CHECKS FOR MICROSERVICES

A health check is a special REST API implementation that you can use to validate the status of a microservice and its dependencies. MicroProfile Health enables microservices in an application to self-check their health and then publishes the overall health status to a defined endpoint.

A health check can assess anything that a microservice needs, including dependencies, system properties, database connections, endpoint connections, and resource availability. Services report as their availability by implementing the API provided by MicroProfile Health. When a microservice is available, it reports an **UP** status. If a microservice is unavailable, it reports a **DOWN** status. A service orchestrator can then use these status reports to decide how to manage and scale the microservices within an application. Health checks can also interact with Kubernetes liveness and readiness probes.

For example, in a microservices-based banking application, you might implement health checks on the login microservice, the balance transfer microservice, and the bill pay microservice. If a health check on the balance transfer microservice detects a bug or deadlock, it returns a **DOWN** status. In this case, if the **/health/live** endpoint is configured in a Kubernetes liveness probe, the probe restarts the microservice automatically. Restarting the microservice saves the user from encountering downtime or an error and preserves the functions of the other microservices in the application.

12.2.1. MicroProfile Health endpoints and annotations

MicroProfile Health provides the following three endpoints:

- **/health/ready**: returns the readiness of a microservice, or whether it is ready to process requests. This endpoint corresponds to the Kubernetes readiness probe.
- **/health/live**: returns the liveness of a microservice, or whether it encountered a bug or deadlock. If this check fails, the microservice is not running and can be stopped. This endpoint corresponds to the Kubernetes liveness probe, which automatically restarts the pod if the check fails.
- **/health**: aggregates the responses from the **/health/live** and **/health/ready** endpoints. This endpoint corresponds to the deprecated **@Health** annotation and is only available to provide compatibility with MicroProfile 1.0. If you are using MicroProfile 2.0 or higher, use the **/health/ready** or **/health/live** endpoints instead.

To implement readiness or liveness health checks, add either the **@Liveness** or **@Readiness** annotation to your code. These annotations link the provided endpoints to Kubernetes liveness and readiness probes.

The following example demonstrates the **@Liveness** annotation, which checks the heap memory usage. If memory consumption is less than 90%, it returns an **UP** status. If memory usage exceeds 90%, it returns a **DOWN** status:

```
@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;
   }
}
```

The following example shows the JSON response from the **/health/live** endpoint if heap memory usage is less than 90%. The first status indicates the overall status of all health checks that are returned from the endpoint. The second status indicates the status of the particular check that is specified by the preceding **name** value, which in this example is **heap-memory**. In order for the overall status to be **UP**, all checks that are run on the endpoint must pass:

```
{
  "status": "UP",
  "checks": [
    {
      "name": "heap-memory",
      "status": "UP",
      "data": {
        "used": "1475462",
```

```
      "max": "51681681"
    }
   }
  ]
}
```

The next example shows the **@Readiness** annotation, which checks for an available database connection. If the connection is successful, it returns an **UP** status. If the connection is unavailable, it returns a **DOWN** status:

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

   @Override
   public HealthCheckResponse call() {

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

The following example shows the JSON response from the **/health/ready** endpoint if the database connection is unavailable. The first status indicates the overall status of all health checks returned from the endpoint. The second status indicates the status of the particular check that is specified by the preceding **name** value, which in this example is **databaseReady**. If any check that runs on the endpoint fails, the overall status is **DOWN**.

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

12.2.2. See also:

- Guide: Adding health reports to microservices
- Guide: Checking the health of microservices on Kubernetes .
- MicroProfile Health feature
- MicroProfile Health on GitHub

APPENDIX A. ADDITIONAL OPEN LIBERTY RESOURCES

You can learn more about Open Liberty and the APIs it supports by viewing resources on the Open Liberty website.

- Open Liberty server commands
- Open Liberty guides
- Java EE API
- MicroProfile API