Open Liberty 2020

Open Liberty Runtime Guide

Build and deploy cloud native applications with Open Liberty
Build and deploy cloud native applications with Open Liberty
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 ................................................................. 3
   1.1. OPEN LIBERTY VERSIONING ..................................................... 3

CHAPTER 2. PACKAGING AND DEPLOYING APPLICATIONS ......................... 4

CHAPTER 3. FEATURE OVERVIEW ............................................................... 5
   3.1. USING FEATURES ................................................................. 5
   3.2. ZERO-MIGRATION ................................................................. 5
   3.3. COMBINING FEATURES ....................................................... 6
   3.4. SUPERSEDED FEATURES ..................................................... 6

CHAPTER 4. SERVER CONFIGURATION OVERVIEW ...................................... 8
   4.1. CONFIG FILES ........................................................................ 8
      4.1.1. server.env ................................................................. 8
      4.1.2. jvm.options .............................................................. 9
      4.1.3. bootstrap.properties ................................................. 9
      4.1.4. server.xml .............................................................. 10
   4.2. VARIABLE SUBSTITUTION .................................................... 10
   4.3. CONFIG MERGING ............................................................... 12
      4.3.1. Merging singleton config ........................................... 12
      4.3.2. Merging factory config ............................................ 13
   4.4. INCLUDE PROCESSING ....................................................... 13
   4.5. CONFIG REFERENCES .......................................................... 14
   4.6. DYNAMIC UPDATES ............................................................. 14
   4.7. LOG MESSAGES ................................................................. 14

CHAPTER 5. LOGGING AND TRACING ......................................................... 16
   5.1. LOGGING CONFIGURATION .................................................... 16
   5.2. EXAMPLE LOGGING CONFIGURATION .................................... 16
      5.2.1. Managing log file storage ........................................... 16
      5.2.2. JSON logging ............................................................ 16
      5.2.3. Configuring logging for a Docker image ....................... 17
      5.2.4. Binary logging .......................................................... 17
   5.3. CONFIGURATION SETTINGS BY SOURCE ................................... 18

CHAPTER 6. TESTING WITH A CONTAINER .................................................. 25

CHAPTER 7. DEBUGGING .......................................................................... 26
   7.1. DEV MODE .......................................................................... 26
   7.2. MANAGING BUILD PROCESSES WITH THE OPEN LIBERTY MAVEN PLUG-IN .................................................. 26
   7.3. TRACING REQUESTS .............................................................. 27

CHAPTER 8. MONITORING OPEN LIBERTY ................................................ 28
   8.1. ENABLING MONITORING IN MICROSERVICES ....................... 28
   8.2. ENABLING HEALTH CHECKS IN MICROSERVICES .................. 28

APPENDIX A. ADDITIONAL OPEN LIBERTY RESOURCES ............................ 30
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. PACKAGING AND DEPLOYING APPLICATIONS

You can package everything you need to run your application in a container image and use containers to deploy it in different environments. When you run Open Liberty applications on OpenShift, you combine the most flexible server runtime available to Java developers with a streamlined DevOps process and an intuitive development pipeline.

For more information, see Deploying microservices to OpenShift.

To learn how to containerize and run your microservices with Open Liberty, see Containerizing microservices.

Open Liberty supports all MicroProfile and Java EE APIs and deploys to every major cloud platform. To get started with Open Liberty, learn how to run and update a simple REST microservice with Packaging and deploying applications.

Additional resources

- Deploying microservices to Kubernetes
CHAPTER 3. 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.

3.1. 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.

3.2. 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.
3.3. 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.

3.4. 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:

<table>
<thead>
<tr>
<th>Superseded feature</th>
<th>Superseding feature</th>
<th>Dependent feature removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>appSecurity-1.0</td>
<td>appSecurity-2.0</td>
<td>The ldapRegistry and the servlet-3.0 feature were removed from the definition of the appSecurity-2.0 feature.</td>
</tr>
<tr>
<td>jmsMdb-3.2</td>
<td>jms-2.0 and mdb-3.2</td>
<td>Together, the jms-2.0 and mdb-3.2 features provide the same function as the jmsMdb-3.2 feature.</td>
</tr>
<tr>
<td>ssl-1.0</td>
<td>transportSecurity-1.0</td>
<td>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.</td>
</tr>
</tbody>
</table>
CHAPTER 4. SERVER CONFIGURATION OVERVIEW

The Open Liberty server config 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>  
  </featureManager>
  <httpEndpoint id="defaultHttpEndpoint"
               httpPort="9080"  
               httpsPort="9443" />
  <applicationManager autoExpand="true" />
</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 config can be used to refer to all of the files that make up the server config, or specifically to the config that’s in the XML files. If it’s not clear in context, the term server XML config might be used to refer to the config in the XML files.

4.1. CONFIG FILES

The server config 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 config and features.

4.1.1. 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 config. 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 config is read from. However, in some secure profiles the server config 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.

### 4.1.2. 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.

### 4.1.3. bootstrap.properties
The **bootstrap.properties** file is optional. This file is read during Open Liberty bootstrap to provide config 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 **bootstrap.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.

### 4.1.4. **server.xml**

The most important and only required config 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 config is ignored.

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

Server config 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 config by dropping server-formatted XML files into directories. Files are read in alphabetical order in each of the two configDropins directories.

### 4.2. **VARIABLE SUBSTITUTION**

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

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

Variables are referenced by using the ${{variableName}} syntax. Specify variables in the server config, as shown in the following example:

```xml
<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:

```xml
<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:

```xml
<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:

```xml
<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:

```xml
<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 config 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 config 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}`.

---

### 4.3. CONFIG MERGING

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

#### 4.3.1. Merging singleton config

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

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

is treated as:

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

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

```xml
<server>
  <logging b="false" />
</server>
```

is treated as:

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

Config is sometimes provided by using child elements that take text. In these cases, the config is merged by using all of the values specified. The most common scenario is configuring features. For example:
4.3.2. Merging factory config

Factory config merges use the same rules as singleton config except elements are not automatically merged just because the element names match. With factory config, 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 \texttt{id} attribute must be set on each element to indicate they are the same thing. Variable substitution on an \texttt{id} attribute is not supported.

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

4.4. INCLUDE PROCESSING

In addition to the default locations, additional config files can be brought in by using the \texttt{include} element. When a server config 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 \texttt{include} element. In the following example, the server processes the contents of the \texttt{other.xml} file before it processes the contents of the \texttt{other2.xml} file:

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

4.5. CONFIG REFERENCES

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

A reference to config always uses the `id` attribute of the element that is being referenced. The config 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>
```

4.6. DYNAMIC UPDATES

The server monitors the server XML config 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 config file on the local disk is monitored for updates every 500ms. You can configure the frequency of XML config 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" />
```

4.7. LOG MESSAGES

When the server runs, it might output log messages that reference config. The references in the log use an XPath-like structure to specify config elements. The element name is given with the value of the `id` attribute inside square brackets. If no `id` is specified in the server config, an `id` is automatically generated. Based on the following server XML config 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 5. 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.

5.1. 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 bootstrap.properties and in some cases using environment variables.

5.2. EXAMPLE LOGGING CONFIGURATION

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

5.2.1. 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:

```properties
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
```

5.2.2. 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.

### 5.2.3. 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:

- **Using environment variables:**

  ```
  WLP_LOGGING_MESSAGE_FORMAT=json
  WLP_LOGGING_MESSAGE_SOURCE=message,trace,accessLog,ffdc,audit
  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 environment 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="
  -e "WLP_LOGGING_CONSOLE_SOURCE="
  open-liberty
  ```

### 5.2.4. 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:
5.3. 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.

<table>
<thead>
<tr>
<th>Server XML Attribute</th>
<th>bootstrap property</th>
<th>Env var</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hideMessage</td>
<td>com.ibm.ws.logging.hideMessage</td>
<td></td>
<td>You can use this attribute to configure the messages keys that you want to hide from the <code>console.log</code> and <code>messages.log</code> files. If the messages are configured to be hidden, then they are redirected to the <code>trace.log</code> file.</td>
</tr>
<tr>
<td>logDirectory</td>
<td>com.ibm.ws.logging.log.directory</td>
<td>LOG_DIR</td>
<td>You can use this attribute to set a directory for all log files, excluding the <code>console.log</code> file, but including FFDC. The default is <code>WLP_OUTPUT_DIR/serverName/logs</code>. It is not recommended to set the <code>logDirectory</code> in <code>server.xml</code> since it can result in some log data being written to the default location prior to <code>server.xml</code> being read.</td>
</tr>
</tbody>
</table>

**Console Log Config**

<p>| consoleFormat        | com.ibm.ws.logging.console.format   | WLP_LOGGING_CONSOLE_FORMAT | The required format for the console. Valid values are <code>basic</code> or <code>json</code> format. By default, <code>consoleFormat</code> is set to <code>basic</code>. |</p>
<table>
<thead>
<tr>
<th>Server XML Attribute</th>
<th>bootstrap property</th>
<th>Env var</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>consoleLogLevel</td>
<td>com.ibm.ws.logging.console.log.level</td>
<td>WLP_LOGGING_CONSOLE_LOGLEVEL</td>
<td>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.</td>
</tr>
<tr>
<td>consoleSource</td>
<td>com.ibm.ws.logging.console.source</td>
<td>WLP_LOGGING_CONSOLE_SOURCE</td>
<td>The list of comma-separated sources that route to the console. This property applies only when <code>consoleFormat=&quot;json&quot;</code>. Valid values are message, trace, accessLog, ffdc, and audit. By default, <code>consoleSource</code> 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.</td>
</tr>
<tr>
<td>copySystemStreams</td>
<td>com.ibm.ws.logging.copy.system.streams</td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

Message Log Config
<table>
<thead>
<tr>
<th>Server XML Attribute</th>
<th>bootstrap property</th>
<th>Env var</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.ws.logging.new LogsOnStart</td>
<td></td>
<td></td>
<td>If set to true when Liberty starts, any existing <code>messages.log</code> or <code>trace.log</code> files are rolled over and logging writes to a new <code>messages.log</code> or <code>trace.log</code> file. If set to false <code>messages.log</code> or <code>trace.log</code> files only refresh when they hit the <code>maxFileSize</code>. The default is <code>true</code>. This setting cannot be provided using the <code>logging</code> element in <code>server.xml</code> because it is only processed during server bootstrap.</td>
</tr>
<tr>
<td>isoDateFormat</td>
<td>com.ibm.ws.logging.isoDateFormat</td>
<td></td>
<td>Specifies whether to use ISO-8601 formatted dates in log files. The default value is false.</td>
</tr>
<tr>
<td>isoDateFormat</td>
<td>com.ibm.ws.logging.isoDateFormat</td>
<td></td>
<td>If set to true, the ISO-8601 format is used in the <code>messages.log</code> file, the <code>trace.log</code> file, and the FFDC logs. The format is <code>yyyy-MM-dd'T'HH:mm:ss.SSS Z</code>.</td>
</tr>
<tr>
<td>isoDateFormat</td>
<td>com.ibm.ws.logging.isoDateFormat</td>
<td></td>
<td>If you specify a value of <code>false</code>, 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 <code>dd/MMM/yyyy HH:mm:ss:SSS z</code>.</td>
</tr>
<tr>
<td>Server XML Attribute</td>
<td>bootstrap property</td>
<td>Env var</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>maxFiles</td>
<td>com.ibm.ws.logging.max. files</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>maxFileSize</td>
<td>com.ibm.ws.logging.max. file.size</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>messageFileName</td>
<td>com.ibm.ws.logging.message.file.name</td>
<td></td>
<td>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</td>
</tr>
<tr>
<td>Server XML Attribute</td>
<td>bootstrap property</td>
<td>Env var</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>messageFormat</td>
<td>com.ibm.ws.logging.messages.format</td>
<td>WLP_LOGGING_MESSAGE_FORMAT</td>
<td>The required format for the messages.log file. Valid values are basic or json format. By default, messageFormat is set to basic.</td>
</tr>
<tr>
<td>messageSource</td>
<td>com.ibm.ws.logging.messages.source</td>
<td>WLP_LOGGING_MESSAGE_SOURCE</td>
<td>The list of comma-separated sources that route to the messages.log file. This property applies only when messageFormat=&quot;json&quot;. 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.</td>
</tr>
</tbody>
</table>

### Trace Config

<table>
<thead>
<tr>
<th>Server XML Attribute</th>
<th>bootstrap property</th>
<th>Env var</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>suppressSensitiveTrace</td>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
The `traceFileName` attribute determines the base name of the trace file. The 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 output stream.

The `traceFormat` attribute controls the format of the trace log. The default format for Liberty is `ENHANCED`. You can also use `BASIC` and `ADVANCED` formats.

The `traceSpecification` attribute allows you 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
that is running in 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.

<table>
<thead>
<tr>
<th>Server XML Attribute</th>
<th>bootstrap property</th>
<th>Env var</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 6. TESTING WITH A CONTAINER

You can use MicroShed Testing to develop integration tests for your Open Liberty application. You test your application from outside its container so that the tests run against the same image that you use in production.

To write a test with MicroShed testing, first add microshed-testing-testcontainers and junit-jupiter artifact IDs as test-scoped dependencies, as shown in the following example:

```xml
<dependency>
    <groupId>org.microshed</groupId>
    <artifactId>microshed-testing-testcontainers</artifactId>
    <version>0.4.1</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.4.2</version>
    <scope>test</scope>
</dependency>

<!-- Any compatible version of JUnit Jupiter 5.X will work -->
```

Next, create a test class with the MicroShedTest annotation. Then, create a public static MicroProfileApplication method and inject one or more public static JAX-RS resources, as shown in the following example:

```java
@MicroShedTest
public class MyTest {

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

    @Inject
    public static MyService mySvc;

    // write @Test methods as normal

    In cases where a Dockerfile or container image is not available, use the microshed-testing-liberty adapter. This adapter produces a container image that is similar to the following Dockerfile:

```
```text
FROM open-liberty:microProfile3
ADD build/libs/$APP_FILE /config/dropins
COPY src/main/liberty/config /config
```

For more information, see MicroShed Testing.

Additional resources

- Arquillian Liberty managed container adapter
CHAPTER 7. DEBUGGING

7.1. DEV MODE

With Liberty dev mode, you can rapidly code, deploy, and debug applications with Liberty. You enable dev mode through the Liberty Maven-plugin.

To use dev mode, add the following code to your Maven pom.xml file and run the `mvn liberty:dev` command.

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

Dev mode provides three key features:

- Your running server can detect, recompile, and pick up code changes.
- You can run unit and integration tests on demand, by pressing `Enter` in the command window where dev mode is running.
- You can attach a debugger to the running server to step through your code at any time.

When you use dev mode, Liberty automatically detects the following changes to your application source:

- Java source file changes and Java test file changes.
- Dependencies that are added to your pom.xml file. Liberty detects the dependencies and adds them to your class path.
- Resource file changes. Liberty detects them and copies them into your target directory.
- Configuration directory and configuration file changes. Liberty detects them and copies them into your target directory.
- Addition of new features in the Liberty server configuration. Liberty detects the new features, installs them, and starts them.

Dev Mode does not detect some changes. These changes include the following ones:

- The addition of a configuration directory or file.
- Changes to the host and port for Liberty.

If either of these changes are made, Liberty can detect them after you restart dev mode. To restart, first exit dev mode by typing `Control-C`, or by typing `q` and pressing `Enter`. Then, run the `mvn liberty:dev` command to restart.

For more information, see Liberty Dev mode.

7.2. 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.

### 7.3. 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 8. 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.

8.1. ENABLING MONITORING IN MICROSERVICES

Building observability into microservices externalizes the internal status of a system to enable operations teams to monitor microservice systems more effectively. It’s important that microservices are written to produce metrics that can be used by operations teams when the microservices are running in production.

Metrics are emitted from a number of different places. You can obtain them from applications, the Open Liberty runtime, and the Java virtual machine (JVM). MicroProfile Metrics provides a /metrics endpoint from which you can access all metrics that are emitted by the Open Liberty server and deployed applications. They can be gathered and stored in database tools, such as Prometheus, and displayed on dashboards, such as Grafana.

Metrics come in various forms, including counters, gauges, timers, histograms, and meters. You can enable metrics in your Open Liberty application with the MicroProfile Metrics feature, which defines annotations that help you quickly build metrics into your code.

For a list of all available Open Liberty metrics, see the metrics reference list.

To learn how to use MicroProfile Metrics to enable and provide metrics from a microservice, see Providing metrics from a microservice.

Additional Resources

- Microservice observability with metrics

8.2. ENABLING HEALTH CHECKS IN MICROSERVICES

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

A self-check can be used to assess anything that the service needs, such as:

- Dependencies
- System properties
- Database connections
- Endpoint connections
- Resource availability

With MicroProfile Health, you can enable the services in your liberty application to self-check for liveness and readiness. A liveness check determines whether a service encountered a bug or deadlock. If this check fails, the service is not running and can be terminated. This check corresponds to the
Kubernetes liveness probe, which automatically restarts the pod if the check fails. A readiness check determines whether a service is ready to process requests. This check corresponds to the readiness probe in Kubernetes.

To learn how to use MicroProfile Health to enable and report microservice health checks, see *Adding health reports to microservices*.

**Additional resources**

- *Enabling health checking of microservices*
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