



## Open Liberty 2020

# Release Notes for Open Liberty 20.0.0.12 on Red Hat OpenShift Container Platform

Release Notes for Open Liberty 2020 on Red Hat OpenShift Container Platform



# Open Liberty 2020 Release Notes for Open Liberty 20.0.0.12 on Red Hat OpenShift Container Platform

---

Release Notes for Open Liberty 2020 on Red Hat OpenShift Container Platform

## **Legal Notice**

Copyright © 2020 IBM Corp

Code and build scripts are licensed under the Eclipse Public License v1 Documentation files are licensed under Creative Commons Attribution-NoDerivatives 4.0 International (CC BY-ND 4.0)

## **Abstract**

These release notes contain the latest information about new features, enhancements, fixes, and issues contained in Open Liberty 2020 on Red Hat OpenShift Container Platform release.

---

## Table of Contents

<b>CHAPTER 1. FEATURES .....</b>	<b>3</b>
1.1. RUN YOUR APPS USING 20.0.0.12	3
1.2. PROVIDE AND CONSUME GRPC SERVICES FROM YOUR WEB APPLICATIONS (GRPC SERVICE 1.0 AND GRPC CLIENT 1.0)	3
1.2.1. gRPC Service	4
1.2.2. gRPC Client	4
1.2.3. Try out gRPC	4
1.3. SUPPORT FOR CUSTOM JNDI NAMES FOR ENTERPRISE BEANS (JAKARTA ENTERPRISE BEANS 1.X, 2.X, 3.X)	5
1.3.1. Easier Migration for Enterprise Bean Applications	6
1.3.2. Application Configuration details	6
1.3.2.1. Specify Custom bindings in ibm-ejb-jar-bnd.xml for EJB 3.x	6
1.3.2.2. Specify Custom bindings in server.xml	7
1.3.2.3. Specify Custom bindings in ibm-ejb-jar-bnd.xmi for EJB 1.x/2.x	7
1.3.3. Feature configuration details	7
1.4. SUPPORT FOR JAVA SE 15	8
1.5. SIGNIFICANT BUGS FIXED IN THIS RELEASE	8
<b>CHAPTER 2. RESOLVED ISSUES .....</b>	<b>10</b>
<b>CHAPTER 3. FIXED CVES .....</b>	<b>11</b>
<b>CHAPTER 4. KNOWN ISSUES .....</b>	<b>12</b>



# CHAPTER 1. FEATURES

In Open Liberty 20.0.0.12 we now support gRPC 1.0 and gRPC Client 1.0 features. gRPC is a "high-performance, open source universal RPC framework." We've also added support for custom JNDI (Java Naming and Directory Interface) for Enterprise Beans (EJBs). This allows you to use custom JNDI names for looking up and/or injecting enterprise beans. Another major addition for this release is support for running Open Liberty on Java SE 15, which is currently the latest version of Java SE.

In [Open Liberty 20.0.0.12](#):

- [Provide and consume gRPC services from your web applications \(gRPC Service 1.0 and gRPC Client 1.0\)](#)
- [Support for Custom JNDI Names for Enterprise Beans \(Jakarta Enterprise Beans 1.x, 2.x, 3.x\)](#)
- [Support for Java SE 15](#)
- [Significant bugs fixed in this release](#)

View the list of fixed bugs in [20.0.0.12](#).

## 1.1. RUN YOUR APPS USING 20.0.0.12

If you're using [Maven](#), here are the coordinates:

```
<dependency>
  <groupId>io.openliberty</groupId>
  <artifactId>openliberty-runtime</artifactId>
  <version>20.0.0.12</version>
  <type>zip</type>
</dependency>
```

Or for [Gradle](#):

```
dependencies {
  libertyRuntime group: 'io.openliberty', name: 'openliberty-runtime', version: '[20.0.0.12,)'
}
```

Or if you're using Docker:

```
FROM open-liberty
```

## 1.2. PROVIDE AND CONSUME GRPC SERVICES FROM YOUR WEB APPLICATIONS (GRPC SERVICE 1.0 AND GRPC CLIENT 1.0)

[gRPC](#) is a high-performance, open source universal RPC framework. gRPC support on Liberty allows developers to both provide and consume gRPC services from your web applications. The introduction of gRPC support in Open Liberty means you can now take advantage of the benefits of gRPC more easily than before. Those benefits include great performance, simple service definitions via Protocol Buffers, cross-platform and language support, and wide industry adoption.

These two features were previously only available in beta are now generally available:

- **grpc-1.0**, which enables gRPC services, and
- **grpcClient-1.0**, which enables the use of a gRPC client for outbound calls.

### 1.2.1. gRPC Service

The **grpc-1.0** feature works by scanning web apps for gRPC service implementations, through implementors of **io.grpc.BindableService**. The web app must include the protocol buffer compiler-generated code for the services it intends to provide, and additionally the service class must provide a no-argument constructor. The web app does not need to include any core gRPC libraries; those are provided by the Liberty runtime. Once a gRPC service is scanned and started, it becomes accessible to remote gRPC clients on the configured HTTP ports.

### 1.2.2. gRPC Client

The **grpcClient-1.0** feature provides applications with access to a [Netty](#) gRPC client, as well as the related libraries. A web app must provide a client implementation and stubs, and can make outbound calls with a **io.grpc.ManagedChannel** without needing to provide the supporting client libraries.

### 1.2.3. Try out gRPC

You can now try gRPC by either following the instructions in this [GitHub repository's](#) README or following this basic Hello World service (add the **grpc-1.0** feature to your **server.xml**):

```
package com.ibm.ws.grpc;

import com.ibm.ws.grpc.beans.GreetingBean;

import io.grpc.examples.helloworld.GreeterGrpc;
import io.grpc.examples.helloworld.HelloReply;
import io.grpc.examples.helloworld.HelloRequest;
import io.grpc.stub.StreamObserver;

public class HelloWorldService extends GreeterGrpc.GreeterImplBase {

    public HelloWorldService(){}

    @Override
    public void sayHello(HelloRequest req, StreamObserver<HelloReply> responseObserver) {
        HelloReply reply = HelloReply.newBuilder().setMessage("Hello " + req.getName()).build();
        responseObserver.onNext(reply);
        responseObserver.onCompleted();
    }
}
```

For this example, the application must provide the [helloworld protobuf definition](#) along with the protobuf compiler output. No additional libraries need to be provided with the application, and once it's started the helloworld greeter service will be accessible on the server's HTTP endpoints.

For a client example, a basic Servlet using gRPC can be defined using the **grpcClient-1.0** feature with:

```
package com.ibm.ws.grpc;

import io.grpc.examples.helloworld.GreeterGrpc;
```



```

import io.grpc.examples.helloworld.HelloReply;
import io.grpc.examples.helloworld.HelloRequest;

import io.grpc.ManagedChannel;
import io.grpc.ManagedChannelBuilder;
...
@WebServlet(name = "grpcClient", urlPatterns = { "/grpcClient" }, loadOnStartup = 1)
public class GrpcClientServlet extends HttpServlet {

    ManagedChannel channel;
    private GreeterGrpc.GreeterBlockingStub greetingService;

    private void startService(String address, int port)
    {
        channel = ManagedChannelBuilder.forAddress(address , port).usePlaintext().build();
        greetingService = GreeterGrpc.newBlockingStub(channel);
    }

    private void stopService()
    {
        channel.shutdownNow();
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // set user, address, port params
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // grab user, address, port params
        startService(address, port);
        HelloRequest person = HelloRequest.newBuilder().setName(user).build();
        HelloReply greeting = greetingService.sayHello(person);

        // send the greeting in a response
        stopService();
    }
}

```

As with the service example, the application must provide the [helloworld protobuf definition](#) along with the protobuf compiler output. All required gRPC client libraries are provided by the **grpcClient-1.0** feature.

### 1.3. SUPPORT FOR CUSTOM JNDI NAMES FOR ENTERPRISE BEANS (JAKARTA ENTERPRISE BEANS 1.X, 2.X, 3.X)

Support for Custom JNDI Names for Enterprise Beans (EJBs) is an enhancement to all existing

enterprise beans features that allows an application to configure and use custom JNDI names for looking up and/or injecting enterprise beans, or use legacy default JNDI names instead of the specification defined JNDI names.

Prior to this enhancement, Liberty only supported looking up enterprise beans using the specification defined JNDI names : **java:global/<app>/<module>/<bean>!<interface>** & variations for **java:app** and **java:module**. With this enhancement, and without any additional configuration, legacy default JNDI names are now available for use by applications to lookup and/or inject enterprise beans. Also, rather than using the defaults, a custom name for each EJB may be specified in the **ibm-ejb-jar-bnd.xml** file (or **ibm-ejb-jar-bnd.xmi** file for EJB 2.x and 1.x modules). These new JNDI name options are available in addition to the existing specification required names.

For EJB 3.x modules, the following defaults will be available if a custom name is not provided:

Short form local interfaces and homes **ejblocal:<package.qualified.interface>**

Short form remote interfaces and homes **<package.qualified.interface>**

Long form local interfaces and homes **ejblocal:<component-id>#<package.qualified.interface>**

Long form remote interfaces and homes **ejb/<component-id>#<package.qualified.interface>**

The component-id defaults to **<application-name>/<module-jar-name>/<ejb-name>**

### 1.3.1. Easier Migration for Enterprise Bean Applications

Custom JNDI name support for enterprise beans provides an easier migration path for applications from other platforms (including WebSphere traditional).

Prior to Java EE 6, the Enterprise Beans specification did not prescribe the JNDI names required for enterprise beans, so every platform provided platform specific default names and custom binding file formats. Since Liberty only supported the specification defined JNDI names, migrating applications from other platforms often requires code changes to modify the platform specific JNDI names to the newer specification defined names. Now, migration from other platforms is simplified because applications may be migrated without changing code, but instead migrating the other platform specific binding files to the new Liberty platform specific binding file format. In some cases, use of the new legacy default names may also allow applications to migrate to liberty without specifying custom JNDI names in a binding file.

### 1.3.2. Application Configuration details

Full details about the legacy default bindings provided, as well as the syntax for declaring custom JNDI names in the **ibm-ejb-jar-bnd.xml** file, including examples, may be found in [this IBM Knowledge Center article](#).

Custom bindings may be configured for an application in the following three locations.

#### 1.3.2.1. Specify Custom bindings in ibm-ejb-jar-bnd.xml for EJB 3.x

Following are examples of how to configure custom bindings for EJB 3.x beans in an EJB JAR module or WAR module in **ibm-ejb-jar-bnd.xml**

Specify a binding per interface:

```
<session name="NoInterceptorBasicStateless">
  <interface class="com.ejbs.InventoryService" binding-name="ejb/Inventory"/>
</session>
```

Specify a component id (a prefix for default long form bindings)

```
<session name="AccountServiceBean" component-id="Dept549/AccountProcessor"/>
```

Simple binding name (one name used for both local and remote)

```
<session name="AccountServiceBean" simple-binding-name="ejb/AccountService"/>
```

Local and Remote home specific binding names

```
<session name="AccountServiceBean" local-home-binding-name="ejblocal:AccountService"/>
<session name="AccountServiceBean" remote-home-binding-
name="ejb/services/AccountService"/>
```

### 1.3.2.2. Specify Custom bindings in server.xml

Following is an example of how to configure custom bindings for EJB 3.x beans in an EJB JAR module or WAR module in **server.xml** in the **<application>** or **<ejbApplication>** elements:

```
<ejbApplication location="EJBTest.jar">
  <ejb-jar-bnd>
    <session name="InventoryServiceBean">
      <interface class="com.ejbs.InventoryService" binding-name="ejb/Inventory"/>
    </session>
  </ejb-jar-bnd>
</ejbApplication>
```

### 1.3.2.3. Specify Custom bindings in ibm-ejb-jar-bnd.xmi for EJB 1.x/2.x

Following is an example of how to configure custom bindings for EJB 1.x or 2.x beans in an EJB JAR module in **ibm-ejb-jar-bnd.xmi**

EJB 1.x and 2.x provide a single JNDI name that applies to both the remote and local home:

```
<ejbBindings xmi:id="BeanBinding_8"
jndiName="suite/r6x/base/misc/poollimits/SLCMTTxTimeoutHome">
  <enterpriseBean xmi:type="ejb:Session" href="META-INF/ejb-jar.xml#SLCMTTxTimeout"/>
</ejbBindings>
```

For a bean with both a remote and local home, the above will provide the following custom bindings:

```
Remote Home : suite/r6x/base/misc/poollimits/SLCMTTxTimeoutHome
Local Home  : local:suite/r6x/base/misc/poollimits/SLCMTTxTimeoutHome
```

### 1.3.3. Feature configuration details

Support for custom and legacy default JNDI names is enabled by default for all Enterprise Bean (EJB) features. This support will not interfere with the existing specification defined **java:** support. However, it is possible to completely disable the new support with the following setting in server.xml:

```
<ejbContainer bindToServerRoot="false"/>
```

It is also possible to disable just the legacy short form default JNDI name support (i.e. the bean is bound using the interface name) with the following setting in server.xml:

```
<ejbContainer disableShortDefaultBindings="true"/>
```

Since the new support for custom JNDI names and legacy defaults provide alternative JNDI names, it is now possible to disable the EJB specification required JNDI names. This is done in server.xml as follows:

```
<ejbContainer bindToJavaGlobal="false"/>
```

Finally, the following new configuration attribute on the **<ejbContainer>** element in open-liberty enables the failing application start when multiple beans are bound to the same JNDI name:

```
<ejbContainer customBindingsOnError="FAIL"/>
```

## 1.4. SUPPORT FOR JAVA SE 15

Any official Java SE 15 release from [AdoptOpenJDK](#), [Oracle](#), or other OpenJDK vendors will work with Open Liberty. Although Java SE 15 is currently the latest available version, it is not a long-term supported release, with standard support scheduled to end in March 2021.

Keep in mind, Eclipse OpenJ9 [typically offers faster startup times](#) than Hotspot.

The primary features added in this release include:

- [JEP 379](#) Shenandoah: A Low-Pause-Time Garbage Collector
- [JEP 377](#) ZGC: A Scalable Low-Latency Garbage Collector
- [JEP 378](#) Text Blocks
- [JEP 384](#) Records (Second Preview)
- [JEP 360](#) Sealed Classes (Preview)

For more information on downloading a version of Java 15, see [AdoptOpenJDK.net](#), [Eclipse.org](#) or [OpenJDK.java.net](#).

For working with the **server.env** file in Open Liberty, see the **Configuration Files** section of the Open Liberty [Server Configuration Overview documentation](#).

For more information on new features available in Java 15, see [OpenJDK](#).

## 1.5. SIGNIFICANT BUGS FIXED IN THIS RELEASE

We've spent some time fixing bugs. The following sections describe just some of the issues resolved in this release. If you're interested, here's the [full list of bugs fixed in 20.0.0.12](#).

- [IllegalAccessError when using MP Rest Client with Java SE 15](#)  
There was a MP Rest Client failure when running with Java SE 15.. This was fixed by changing the visibility of the **compareCustomStatus** method to public. If you would like to know more about MP Rest Client check out our [Consuming RESTful services with template interfaces](#) and [Consuming RESTful services asynchronously with template interfaces](#) guides.

- [MP GraphQL does not scan JARs in WEB-INF/lib for GraphQL components](#)

Classes annotated with things like **@GraphQLApi**, **@Type**, etc. or types referenced from root level queries that exist in JARs in the WEB-INF/lib directory were not processed. This meant that only classes in the WEB-INF/classes directory were processed by the MP GraphQL runtime.

- [Increased CPU when moving from Liberty 19.0.0.6 to newer releases](#)

When moving from 19.0.0.6 to a newer release, there was an issue with an increase in CPU usage. This was caused by a fix for a previous issue which made the processing of the Audit feature's method arguments to be performed whether the feature was enabled or not. This issue was fixed by undoing the previous change and fixing the original problem differently. For more details on Liberty's performance, see this [post](#).

- [Variables in include files not recognized after config update](#)

We added support for using variables in include elements in 20.0.0.3. The changes required to support temporary resolution of variables during processing of configuration resulted in a bug where changes to variables may not be recognized during a configuration update. For more information take a look at our [Server Configuration Overview](#).

- [Prevent jsonp-1.0 and jsonpContainer-1.1 features from both starting](#)

When **jsonp-1.0** and **jsonpContainer-1.1** were configured at the same time it would cause a cryptic **CWWKE0702E** error of:

CWWKE0702E: Could not resolve module: com.ibm.websphere.javaee.jsonp.1.0 [265]  
Bundle was not resolved because of a uses constraint violation.

The error handling of this invalid configuration has been improved to give an error message of:

CWWKF0033E: The singleton features com.ibm.websphere.appserver.jsonpImpl-1.1.0 and com.ibm.websphere.appserver.jsonpImpl-1.0.0 cannot be loaded at the same time. The configured features jsonpContainer-1.1 and jsonp-1.0 include one or more features that cause the conflict. Your configuration is not supported; update server.xml to remove incompatible features.

## CHAPTER 2. RESOLVED ISSUES

See the [Open Liberty 20.0.0.12 issues that were resolved for this release](#) .

## CHAPTER 3. FIXED CVES

For a list of CVEs that were fixed in Open Liberty 20.0.0.12, see [security vulnerabilities](#).

## CHAPTER 4. KNOWN ISSUES

See the [list of issues that were found but not fixed during the development of 20.0.0.12](#) .