Open Liberty 2020

Release Notes for Open Liberty 20.0.0.4 on Red Hat OpenShift Container Platform

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

Last Updated: 2020-04-13

Open Liberty 2020 Release Notes for Open Liberty 20.0.0.4 on Red Hat OpenShift Container Platform

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

Legal Notice

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

CHAPTER 1. FEATURES

Open Liberty 20.0.0.4 provides support for MicroProfile 3.3 which includes updates to MicroProfile Rest Client, Fault Tolerance, Metrics, Health, and Config. Improved developer experience is also achieved with support for yum/apt-get installs and the ability to track usage patterns with JAX-RS 2.1.

In Open Liberty 20.0.0.4:

- MicroProfile 3.3
  - Easily determine HTTP headers on outgoing requests (MicroProfile Rest Client 1.4)
  - Monitor microservice applications (MicroProfile Metrics 2.3)
  - Provide health check procedures (MicroProfile Health 2.2)
  - Monitor faults in your microservices (MicroProfile Fault Tolerance 2.1)
  - External configuration of your microservices (MicroProfile Config 1.4)
- Track usage patterns and performance of services (JAX-RS 2.1)
- Support for yum/apt-get installations
- Automatically compress HTTP responses
- Open Liberty Grafana dashboard now available at grafana.com
- Support OpenShift service account credentials for authentication

View the list of fixed bugs in 20.0.0.4.

1.1. RUN YOUR APPS USING 20.0.0.4

If you're using Maven, here are the coordinates:

```
<dependency>
   <groupId>io.openliberty</groupId>
   <artifactId>openliberty-runtime</artifactId>
   <version>20.0.0.4</version>
   <type>zip</type>
</dependency>
```

Or for Gradle:

```
dependencies {
   libertyRuntime group: 'io.openliberty', name: 'openliberty-runtime', version: '[20.0.0.4,)'
}
```

Or if you're using Docker:

```
FROM open-liberty
```

1.2. MICROPROFILE 3.3 SUPPORT

MicroProfile 3.3 improves developer experience with updates to the Rest Client, Fault Tolerance, Metrics, Health, and Config features.

1.2.1. Easily determine HTTP headers on outgoing requests (MicroProfile Rest Client 1.4)

MicroProfile Rest Client 1.4 adds injection into **ClientHeadersFactory** instances. When executing a Rest Client inside a JAX-RS request, it can be useful to pull data from the JAX-RS request's context or from CDI to use to determine which HTTP headers to send on the outgoing request. With MP Rest Client 1.4, this is now possible.

To enable MP Rest Client 1.4, add this feature to your server.xml: **<feature>mpRestClient-1.4</feature>**

CDI and/or JAX-RS injection into your **ClientHeadersFactory** will enable you to do things like:

```
@ApplicationScoped
public class MyCustomClientHeadersFactory implements ClientHeadersFactory {

    @Context
    private UriInfo uriInfo;

    @Inject
    private Foo foo;

    @Override
    public MultivaluedMap <String, String> update(MultivaluedMap<String, String> incomingHeaders
MultivaluedMap<String, String> clientOutgoingHeaders) {
        MultivaluedMap<String, String> myHeaders = new MultivaluedHashMap<>();
        myHeaders.putSingle("X-HEADER_FROM_CUSTOM_CLIENTHEADERSFACTORY", "456");

        URI uri = uriInfo.getAbsolutePath();
        myHeaders.putSingle("X-INJECTED_URI_INFO", uri == null ? "null" : uri.toString());

        myHeaders.putSingle("X-INJECTED_FOO", foo.getWord());

        return myHeaders;
    }
}
```

Learn more about MP Rest Client

1.2.2. Monitor microservice applications easily wth metrics (MicroProfile Metrics 2.3)

MicroProfile Metrics 2.3 introduces a new metric type called a Simple Timer (annotated with **@SimplyTimed**) and runtime provided metrics that track REST resource method usage and is backed by the new Simple Timer metric.

The new Simple Timer metric is a light-weight alternative to the existing Timer metric. It only tracks the total timing duration and counts the amount of times it was invoked. The Timer metric on the other hand is a performance heavy metric that continually calculates duration statistics and throughput statistics resulting in 14 values.

The new REST stat metrics are gathered from REST resource method usage (i.e **GET**, **POST**, **PUT**, **DELETE**, **OPTIONS**, **PATCH**, **HEAD**). Total time duration and total count of invocation is tracked (by use of the Simple Timer metric). This functionality is properly enabled when used in combination with the **jaxrs-2.1** feature. All REST stat metrics will use the **REST.request** metric name and will be tagged/labeled with their fully qualified class name and method signature.

To enable the feature, include the following in the **server.xml**.

```
<feature>mpMetrics-2.3</feature>
```

The **monitor-1.0** feature will be activated with the **mpMetrics-2.3** feature. As a result, vendor metrics will be enabled due to the **monitor-1.0** activation. To filter out this data you can configure the following into the server.xml to choose the stats you want to see:

```
<monitor filter="JVM,ThreadPool,WebContainer,Session,ConnectionPool,REST"/>
```

Alternatively, disable all **monitor-1.0** stats:

```
<monitor filter=“ ”/> <!-- space required -->
```

To use the new **SimpleTimer** metric programmatically:

```
@Inject
MetricRegistry metricRegistry;

//create metric
Metadata metadata= Metadata.builder().withName("sampleSimpleTimer").build();
SimpleTimer simpleTimer = metricRegistry.simpleTimer(metadata);

//retrieve simple timer context (will start timing)
SimpleTimer.Context simpleTimerContext = simpleTimer.time()

doLogic();
//stops the simple timer from timing
simpleTimerContext.close();
```

To use the **SimpleTimer** metric with annotations:

```
@SimplyTimed(name=“sampleSimpleTimer”)
public void doSomething() {
   doLogic();
}
```

Resulting `OpenMetrics output:

```
# TYPE application_sampleSimpleTimer_total counter
application_sampleSimpleTimer_total 12
# TYPE application_sampleSimpleTimer_elapsedTime_seconds gauge
application_sampleSimpleTimer_elapsedTime_seconds 12.3200000
```

REST stat metrics will be enabled with the **mpMetrics-2.3** feature given the following REST resource:

```
package org.eclipse.microprofile.metrics.demo;

@ApplicationScoped
```

```
public class RestDemo {

  @POST
  public void postMethod(String... s, Object o){
      ...
  }
}
```

Regarding REST stat metrics, the **OpenMetrics** formatted REST metrics would be:

```
# TYPE base_REST_request_total counter
base_REST_request_total{class="org.eclipse.microprofile.metrics.demo.RestDemo",method="postMeth
od_java.lang.String[]_java.lang.Object"} 1
# TYPE base_REST_request_elapsedTime_seconds gauge
base_REST_request_elapsedTime_seconds{class="org.eclipse.microprofile.metrics.demo.RestDemo",
method="postMethod_java.lang.String[]_java.lang.Object"} 1.000
```

1.2.3. Provide your own health check procedures (MicroProfile Health 2.2)

MicroProfile Health Check 2.2 enables you to provide your own health check procedures to be invoked by Open Liberty to verify the health of your microservice.

In the **mpHealth-2.2** feature, all of the supported Qualifiers (Liveness and Readiness) now have annotation literals added in the specification. These ease programmatic lookup and support for inline instantiation of the qualifiers, which was not supported in the previous versions.

Also, for better integration with third party frameworks, like MicroProfile Rest Client, the **HealthCheckResponse** class declaration was changed from an abstract class to a concrete class with constructors allowing for direct instantiation on the consuming end.

To enable the feature, include the following in the **server.xml**:

```
feature>mpHealth-2.2</feature>
```

Applications are expected to provide health check procedures by implementing the **HealthCheck** interface with the **@Liveness** or **@Readiness** annotations. These are used by Open Liberty to verify the Liveness or Readiness of the application, respectively. Add the logic of your health check in the **call()** method, and return the **HealthCheckResponse** object, by using the simple **up()**/**down()** methods from the API:

```
//Liveness Check
@Liveness
@ApplicationScoped
public class AppLiveCheck implements HealthCheck {
...
    @Override
     public HealthCheckResponse call() {
       ...
       HealthCheckResponse.up("myCheck");
       ...
     }
}
```

To view the status of each health check, access the either the **http://<hostname>:<port>/health/live** or **http://<hostname>:<port>/health/ready** endpoints.

1.2.4. Monitor faults in your microservices (MicroProfile Fault Tolerance 2.1)

MicroProfile Fault Tolerance allows developers to easily apply strategies for mitigating failure to their code. It provides annotations which developers can add to methods to use bulkhead, circuit breaker, retry, timeout and fallback strategies. In addition, it provides an annotation which causes a method to be run asynchronously.

MicroProfile Fault Tolerance 2.1 includes the following changes:

- Adds new parameters **applyOn** and **skipOn** to **@Fallback** and adds **skipOn** to **@CircuitBreaker** to give the user more control over which exceptions should trigger these strategies, for example:

```
@Fallback(applyOn=IOException.class, skipOn=FileNotFoundException.class,
fallbackMethod="fallbackForService")
public String readTheFile() {
   ...
}
```

- Ensures that the CDI request context is active during the execution of methods annotated with **@Asynchronous**.
- This Fault Tolerance release also adds more detail into the Javadoc and makes some minor clarifications to the specification.

For more information:

- Get an introduction to MicroProfile Fault Tolerance:
  - Failing fast and recovering from errors
  - Preventing repeated failed calls to microservices
- Reference the Javadoc
- Reference the full specification including the 2.1 release notes
- Report any issues on Github

1.2.5. External configuration of your microservices (MicroProfile Config 1.4)

The MicroProfile Config 1.4 feature provides an implementation of the Eclipse MicroProfile Config 1.4 API which has mainly had changes to the built-in and implicit converters.

The Open Liberty implementation already supported **byte**/**Byte** and **short**/**Short** but **char**/**Character** has now been added.

If we have the following properties available in a **ConfigSource**:

```
byte1=128
short1=5
char1=\u00F6
```

You can inject those properties into your application, either as primitives or as their boxed equivalents:

```
@Dependent
public class MyBean {

    @Inject
    @ConfigProperty(name = "byte1")
    private Byte property1;

    @Inject
    @ConfigProperty(name = "byte1")
    private byte property2;

    @Inject
    @ConfigProperty(name = "short1")
    private Short property3;

    @Inject
    @ConfigProperty(name = "short1")
    private short property4;

    @Inject
    @ConfigProperty(name = "char1")
    private Character property5;

    @Inject
    @ConfigProperty(name = "char1")
    private char property6;
}
```

The implicit converter order has been slightly changed. Previously the order was:

- **of(String)**
- **valueOf(String)**
- **constructor(String)**
- **parse(CharSequence)**

In version 1.4, the last two have been swapped:

- **of(String)**
- **valueOf(String)**
- **parse(CharSequence)**
- **constructor(String)**

The reason for this change is that static **parse(CharSequence)** methods typically have some built-in caching of their results and are therefore faster in some cases. Also, in many cases throughout the JDK, the String constructors have been deprecated.

In the following example, the **MyType** class has two possible implicit converter methods available; a String constructor and a **static parse(CharSequence)** method:

–

```
public class MyType {

    private static final ConcurrentMap<CharSequence, MyType> cache = new ConcurrentHashMap<>
();
    private String value;

    private MyType(CharSequence raw, boolean cached) {
        if (cached) {
            this.value = "Cached: " + raw;
        } else {
            this.value = "Constructor: " + raw;
        }
    }

    public MyType(String raw) {
        this(raw, false);
    }

    public static MyType parse(CharSequence raw) {
        MyType cached = cache.get(raw);
        if (cached == null) {
            cached = new MyType(raw, true);
            MyType previous = cache.putIfAbsent(raw, cached);
            if (previous != null) {
                cached = previous;
            }
        }
        return cached;
    }

    @Override
    public String toString() {
        return value;
    }
}
```

To enable the feature, include the following in the **server.xml**:

```
<feature>mpConfig-1.4</feature>
```

In MicroProfile Config 1.3, the **String** constructor would have been used to do the implicit conversion. In version 1.4, the **parse(CharSequence)** method will be used instead. Notice that the parse method uses a simple cache. If the same raw **String** (**String** extends **CharSequence**) is converted twice then the same instance of **MyType** will be returned. This would not be possible with a **String** constructor.

We have also made a notable internal change to the Open Liberty implementation. In versions prior to 1.4, our implementation included a background update thread which frequently scanned through the available **ConfigSources** and cached the results. This made calls to the **Config API** very fast. However, since the size and complexity of user provided **ConfigSources** is unknown, this was a potentially expensive thing to be doing in the background.

The background update thread has been replaced with an expiry process. What this means is that the first request for a property may be a little slower as it may need to go through all the available **ConfigSources** to find a value. Once found, this value is then cached and a timer started to expire the cache. If a second request is made for that property before the cache expires then the cached value is used and will return quickly. In order to maintain the same dynamic characteristics of the previous versions, the expiry time is set to only 500ms. This value may be increased by setting the **microprofile.config.refresh.rate** system property. 500ms is the minimum expiry time allowed but if the property is set to 0 or less then caching is disabled.

For more information:

- Changes to the API since 1.3

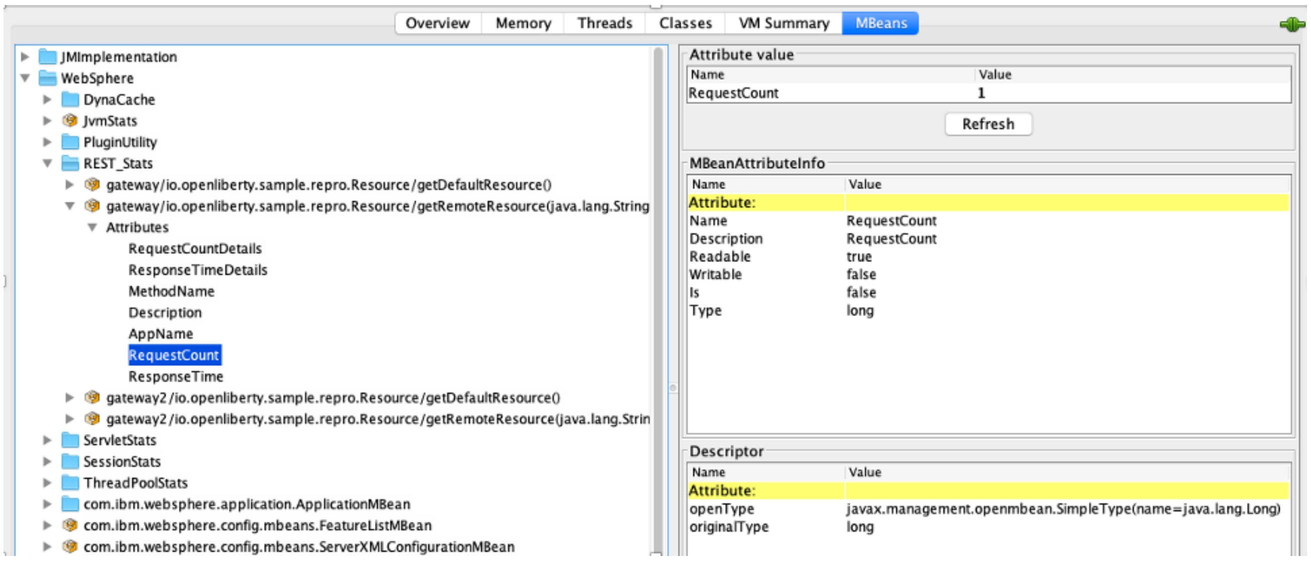
1.3. TRACK USAGE PATTERNS AND PERFORMANCE OF SERVICES (JAX-RS 2.1)

The JAX-RS 2.1 auto-feature is enabled whenever the **jaxrs-2.0** (or **jaxrs-2.1**) features are specified within the **server.xml** along with the **monitor-1.0** feature. This auto-feature introduces the capability to collect statistics related to the execution of an application's RESTful resource methods (specifically the number of invocations and the cumulative execution time. This data is useful for design, debug, and monitoring purposes. RESTful metrics can be accessed via the monitor-1.0 feature in combination with the **mpMetrics-2.3** feature. This information is also accessible via JMX (JConsole, etc...).

Include the following in the server.xml for JMX/PMI access:

```
<feature>jaxrs-2.0</feature> (or jaxrs-2.1)
<feature>monitor-1.0</feature>
```

The JMX/PMI data collected is per-method and is aggregated to the class and web module level. For example, suppose a server has two web modules, each with identically named classes containing two resource methods, the results for the REST_Stats in jconsole will look like the following:



```
▶ JMImplementation
▼ WebSphere
  ▶ DynaCache
  ▶ JvmStats
  ▶ PluginUtility
  ▼ REST_Stats
    ▶ gateway/io.openliberty.sample.repro.Resource/getDefaultResource()
    ▼ gateway/io.openliberty.sample.repro.Resource/getRemoteResource(java.lang.String
      ▼ Attributes
          RequestCountDetails
          ResponseTimeDetails
          MethodName
          Description
          AppName
          RequestCount
          ResponseTime
    ▶ gateway2/io.openliberty.sample.repro.Resource/getDefaultResource()
    ▶ gateway2/io.openliberty.sample.repro.Resource/getRemoteResource(java.lang.Strin
  ▶ ServletStats
  ▶ SessionStats
  ▶ ThreadPoolStats
  ▶ com.ibm.websphere.application.ApplicationMBean
  ▶ com.ibm.websphere.config.mbeans.FeatureListMBean
  ▶ com.ibm.websphere.config.mbeans.ServerXMLConfigurationMBean
```

Name	Value
RequestCount	1

Name	Value
Attribute:	
Name	RequestCount
Description	RequestCount
Readable	true
Writable	false
Is	false
Type	long

Name	Value
Attribute:	
openType	javax.management.openmbean.SimpleType(name=java.lang.Long)
originalType	long

1.4. SUPPORT FOR YUM/APT-GET INSTALLATIONS

Open Liberty is now available as a native linux **.deb** or **.rpm** package so can now use native OS tools (**yum**/**apt**) to manage your Open Liberty installations. To access Open Liberty **rpms**/**debs**, you'll have to configure your machine to use the Open Liberty repository.

On Ubuntu systems:

Append the following line to **file /etc/apt/sources.list**:

```
deb https://public.dhe.ibm.com/ibmdl/export/pub/software/openliberty/runtime/os-native-
packages/deb/ /
```

Add the repositories' public key with command:

```
sudo wget -O http://public.dhe.ibm.com/ibmdl/export/pub/software/openliberty/runtime/os-native-
packages/public.key | sudo apt-key add -
```

Run command:

```
sudo apt-get update
```

The latest version of Open Liberty can then be installed from the repository by running:

```
sudo apt-get install openliberty
```

On Red Hat Systems:

Create the following file named **/etc/yum.repos.d/openliberty.repo**:

```
[olrepo]
name=olrepo
baseurl=http://public.dhe.ibm.com/ibmdl/export/pub/software/openliberty/runtime/os-native-
packages/rpm/
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://public.dhe.ibm.com/ibmdl/export/pub/software/openliberty/runtime/os-native-
packages/public.key
```

The latest Open Liberty can then be installed by:

```
sudo yum update
sudo yum install openliberty
```

After the **openliberty.rpm** or **openliberty.deb** are installed, the empty **defaultServer** is created and configured to run as a service.

- Open Liberty services will run as user **openliberty**
- The server is located in **/var/lib/openliberty/usr/servers/defaultServer**
- Logs will be stored in **/var/log/openliberty/defaultServer**
- PID for server is in **/var/run/openliberty/defaultServer.pid**

You can use the following standard linux service commands:

- **systemctl status openliberty@defaultServer.service**
- **systemctl start openliberty@defaultServer.service**
- **systemctl restart openliberty@defaultServer.service**

- **systemctl stop openliberty@defaultServer.service**

1.5. AUTOMATICALLY COMPRESS HTTP RESPONSES

You can now try out HTTP response compression.

Previous to this feature, Open Liberty only considered compression through the use of the **$WSZIP** private header. There was no way for a customer to configure the compression of response messages. Support now mainly consists of using the **Accept-Encoding** header in conjunction with the **Content-Type header**, of determining if compression of the response message is possible and supported. It allows the Liberty server to compress response messages when possible. It is beneficial because customers will want to use the compression feature to help reduce network traffic, therefore reducing bandwidth and decreasing the exchange times between clients and Liberty servers.

A new element, **<compression>**, has been made available within the **<httpEndpoint>** for a user to be able to opt-in to using the compression support.

The optional **types** attribute will allow the user to configure a comma-delimited list of content types that should or should not be considered for compression. This list supports the use of the plus “+” and minus “-“ characters, to add or remove content types to and from the default list. Content types contain a type and a subtype separated by a slash “/” character. A wild card "*" character can be used as the subtype to indicate all subtypes for a specific type.

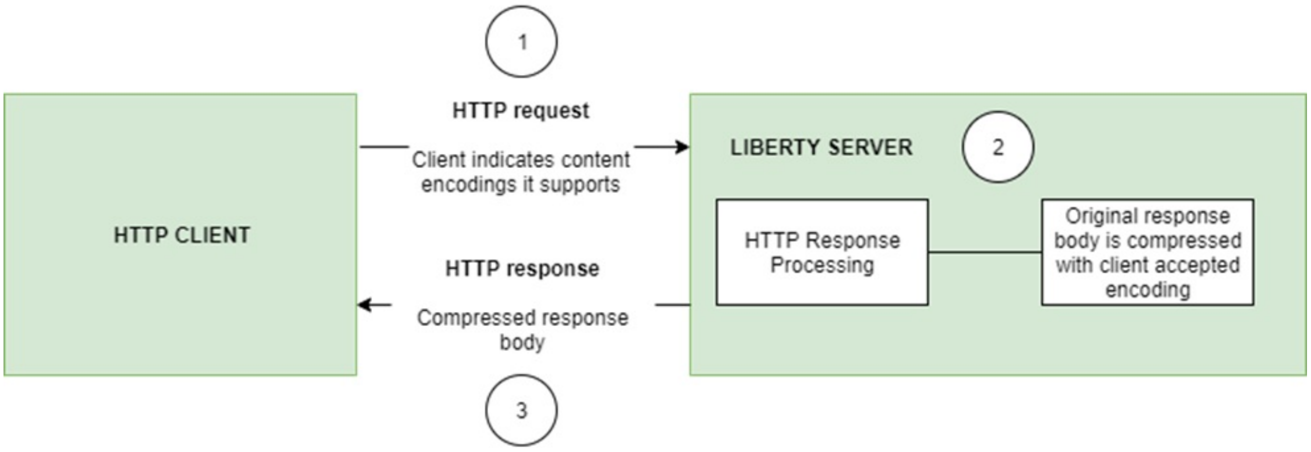
The default value of the types optional attribute is: **text/*, application/javascript**.

Configuring the optional **serverPreferredAlgorithm** attribute, the configured value is verified against the “Accept-Encoding” header values. If the client accepts the configured value, this is set as the compression algorithm to use. If the client does not accept the configured value, or if the configured value is set to ‘none’, the client preferred compression algorithm is chosen by default.

```
<httpEndpoint  id="defaultHttpEndpoint"
       httpPort="9080"
       httpsPort="9443">
    <compression types=“+application/pdf, -text/html”
serverPreferredAlgorithm=“gzip”/></httpEndpoint>
```

Open Liberty supports the following compression algorithms: **gzip**, **x-gzip**, **deflate**, **zlib**, and **identity (no compression)**

The **Http Response Compression** functionality has been designed from the following Open Liberty Epic: #7502. The design is outlined within the Epic for more detailed reading. The basic flow of the design is shown in the below diagrams:


1
HTTP request
Client indicates content encodings it supports
HTTP CLIENT
LIBERTY SERVER
2
HTTP Response Processing
Original response body is compressed with client accepted encoding
HTTP response
Compressed response body
3

1.5.1. Open Liberty Grafana dashboard now available at grafana.com

The Grafana dashboard provides a wide range of time-series visualizations of MicroProfile Metrics data such as CPU, Servlet, Connection Pool, and Garbage Collection metrics. It is powered by a Prometheus datasource which is configured to ingest data from one or more Liberty servers' **/metrics** endpoint, enabling us to view on Grafana in near real-time.

This new dashboard works with Liberty instances outside of OpenShift Container Platform. For Liberty servers running on OCP use the Grafana dashboards published here. The new Grafana dashboard is intended for Open Liberty servers, with **mpMetrics-2.x**, that are not running on OCP.

You can use this dashboard to help spot performance issues when running your applications in Open Liberty. For instance, metrics such as servlet response times, CPU or heap usage when seen as a time-series on Grafana, could be indicative of an underlying performance issue or memory leak.

To configure the dashboard, first add the **mpMetrics-2.3** feature. This will automatically enable the **monitor-1.0** feature:

```
<featureManager>
   <feature>mpMetrics-2.3</feature>
</featureManager>

<mpMetrics authentication="false" />`
```

For metrics on a secure endpoint:

```
<featureManager>
   <feature>mpMetrics-2.3</feature>
</featureManager>

<quickStartSecurity userName="<your-username>" userPassword="<your-password>" />
```

Run the server using the following command:

```
./server run DashboardTest
```

Then, download Prometheus. Once unpackaged, it should contain a startup script called prometheus alongside a configuration file, **prometheus.yml**. Within **prometheus.yml**, append to **scrape_configs** one of the following jobs:

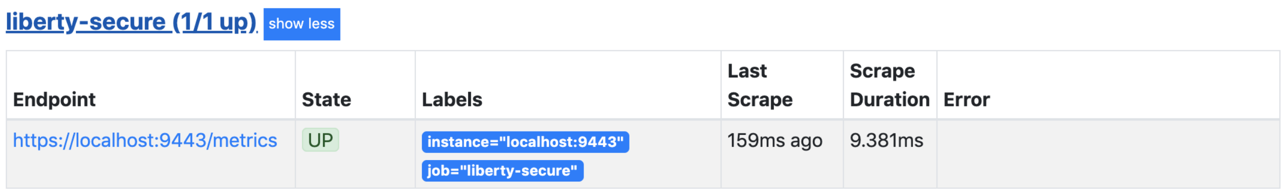
For **mpMetrics** on an insecure endpoint:

```
- job_name: 'liberty'
  scrape_interval: 5s
  static_configs:
    - targets: ['localhost:9080']
```

For **mpMetrics** on a secure endpoint:

```
- job_name: 'liberty-secure'
  scrape_interval: 5s
  static_configs:
    - targets: ['localhost:9443']
  basic_auth:
    username: "<your-username>"
    password: "<your-password>"
  tls_config:
     insecure_skip_verify: true
  scheme: "https"
```

Start the prometheus script and visit **http://localhost:9090/targets**, where you should see your Open Liberty server listed as one of the targets.

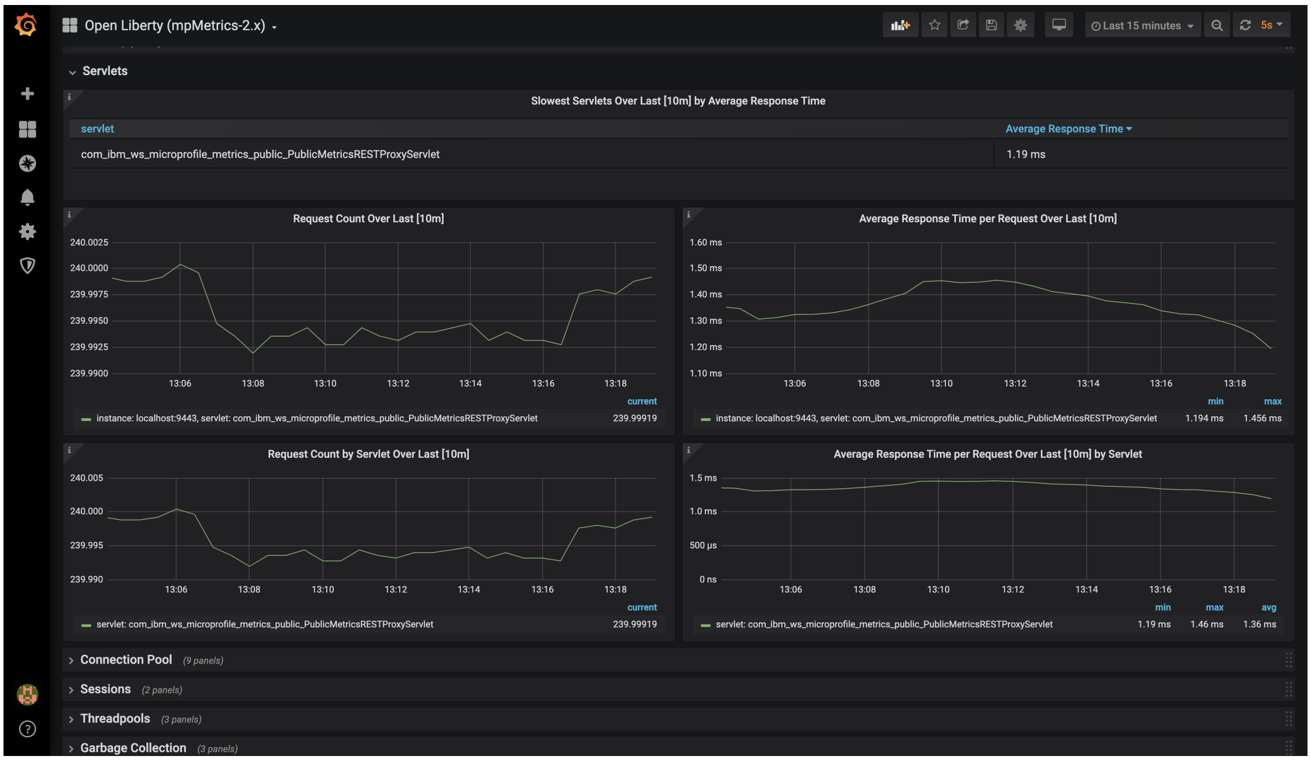

liberty-secure (1/1 up) show less

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
https://localhost:9443/metrics	UP	instance="localhost:9443" job="liberty-secure"	159ms ago	9.381ms	

Download Grafana and once finished, visit https://localhost:3000.

When navigated to Grafana, click the gear icon and select **Data Source**. Add a new Prometheus datasource with the URL as http://localhost:9090. Then click **Save & Test**

To import the dashboard, click the plus icon, select **Import**, paste the dashboard ID 11706, and when prompted in the dropdown menu, link it to the new data source you have just created.


Open Liberty (mpMetrics-2.x)
Last 15 minutes
5s
Servlets
Slowest Servlets Over Last [10m] by Average Response Time
Request Count Over Last [10m]
Average Response Time per Request Over Last [10m]
Request Count by Servlet Over Last [10m]
Average Response Time per Request Over Last [10m] by Servlet
Connection Pool (9 panels)
Sessions (2 panels)
Threadpools (3 panels)
Garbage Collection (3 panels)

servlet	Average Response Time
com_ibm_ws_microprofile_metrics_public_PublicMetricsRESTProxyServlet	1.19 ms

	current
instance: localhost:9443, servlet: com_ibm_ws_microprofile_metrics_public_PublicMetricsRESTProxyServlet	239.99919

	min	max
instance: localhost:9443, servlet: com_ibm_ws_microprofile_metrics_public_PublicMetricsRESTProxyServlet	1.194 ms	1.456 ms

	current
servlet: com_ibm_ws_microprofile_metrics_public_PublicMetricsRESTProxyServlet	239.99919

	min	max	avg
servlet: com_ibm_ws_microprofile_metrics_public_PublicMetricsRESTProxyServlet	1.19 ms	1.46 ms	1.36 ms

For more informtation:

- The dashboard found on Grafana's website
- Using Prometheus to create your own custom visualizations

1.5.2. Support OpenShift service account credentials for authentication

The **socialLogin-1.0** feature can now be configured to use OpenShift service accounts to authenticate and authorize protected resource requests. This allows server administrators to secure, for example, monitoring and metrics endpoints that might produce sensitive information but require repeated access by an automated process or non-human entity. The new behavior allows service accounts to authenticate themselves by providing in the request a service account token that was created within the OpenShift cluster.

A new **<okdServiceLogin>** configuration element is now provided to support this behavior. The **socialLogin-1.0** feature must be enabled to gain access to this new element.

The minimum configuration requires only that an <okdServiceLogin> element be specified in the **server xml**:

```
<server>

<!-- Enable features -->
<featureManager>
  <feature>appSecurity-3.0</feature>
  <feature>socialLogin-1.0</feature>
</featureManager>

<okdServiceLogin />

</server>
```

The minimum configuration assumes that the Liberty server is packaged and deployed within an OpenShift cluster. By default, the **<okdServiceLogin>** element will be used to authenticate all protected resource requests that the Liberty server receives.

Incoming requests to protected resources must include a service account token. The token must be specified as a bearer token in the **Authorization** header of the request. The Liberty server will use the service account token to query information about the associated service account from the OpenShift cluster. The OpenShift project that the service account is in will be used as the group for the service account when making authorization decisions. The OpenShift project name is concatenated with the name of the service account to create the user name.

If the Liberty server is not deployed within an OpenShift cluster, the **userValidationApi** attribute should be configured and set to the value for the appropriate User API endpoint in the OpenShift cluster:

```
<okdServiceLogin
userValidationApi="https://cluster.domain.example.com/apis/user.openshift.io/v1/users/~" />
```

Multiple **<okdServiceLogin>** elements can be configured as long as each element has a unique **id** attribute specified. In those cases, authentication filters should also be configured to ensure the appropriate endpoints are protected by a unique **<okdServiceLogin>** instance.

More information about OpenShift service accounts can be found in the OpenShift documentation for Understanding and creating service accounts.

CHAPTER 2. RESOLVED ISSUES

See the Open Liberty 20.0.0.4 issues that were resolved for this release .

CHAPTER 3. FIXED CVES

For a list of CVEs that were fixed in Open Liberty 20.0.0.4, see security vulnerabilities.

CHAPTER 4. KNOWN ISSUES

See the list of issues that were found but not fixed during the development of 20.0.0.4 .