Important changes introduced with ticket 1269 - Integrate exporter registration with OpenTelemetry and opentelemetry-opencensus-shim, ticket 1279 - Upgrade OpenTelemetry to version 1.10.0 and ticket 1297 - Add exporter services for Jaeger proto and OTLP (OpenTelemetry Protocol) gRPC and http (tracing and metrics)

Date: 2022-03-16 Author: Heiko Holz

With ticket 1269 , ticket 1279, and [ticket 1297](Add exporter services for Jaeger proto and OTLP (OpenTelemetry Protocol) gRPC and http (tracing and metrics)), we have integrated previously supported trace and metrics exporter services with the OpenTelemetry framework.

Updated trace and metric exporters

The state of supported trace and metrics exporters have changed as follows:

Trace exporters

Exporter	Working
Logging Exporter	✓
Jaeger (Thrift)	✓
Zipkin	✓
Jaeger (GRPC)	✓
OTLP (gRPC)	✓
OTLP (HTTP)	✓

Metric exporters

Exporter	Working
Logging Exporter	✓
Prometheus Exporter	✓
InfluxDB Exporter	✓
OTLP (gRPC)	✓
OTLP (HTTP)	✓

Note: The `user` and `password` properties of the InfluxDB Exporter are now mandatory, as inspectIT Ocelot uses the `BasicAuthInterceptor.java` and `Credentials.kt` from `com.squareup.okhttp3:http:4.9.0`, which does not allow `null` for username or password.

Tracing

For tracing, the following properties have been added to `inspectit.tracing`

Property	Default Value	Description
`max-export-batch-size`	512	The max export batch size used for every export, see `io.opentelemetry.sdk.trace.export.BatchSpanProcessor#maxExportBatchSize`

Property	Default Value	Description
`schedule-delay-millis`	5000	The delay interval between two consecutive exports in milliseconds, see `io.opentelemetry.sdk.trace.export.BatchSpanProcessor#maxExportBatchSize#scheduleDelayNanos`.

(Un-)registering trace and metrics exporter services

Previously implemented trace and metrics exporter services are now re-implemented and tested (with the exception of the `OcAgentTraceExporterService` and `OcAgentMetricsExporterService`, which have been completely removed). For this, the `rocks.inspectit.ocelot.bootstrap.opentelemetry.IOpenTelemetryController` interface and the respective `rocks.inspectit.ocelot.core.opentelemetry.OpenTelemetryControllerImpl` have been implemented. The `OpenTelmetryControllerImpl` is responsible for setting up tracing and metrics with OpenTelemetry and updating custom implementations of OpenTelemetry interfaces to register and unregister exporter services.

The individual services, i.e., `DynamicallyActivatableTraceExporterService` and `DynamicallyActivatableMetricsExporterService`, register in their `doEnable` method to the `openTelemetryController` and unregister in their `doDisable` method from the controller.

Custom implementation of `OpenTelemetry` and `SpanExporters`

OpenTelemetry does not come with a `SpanExporter` or `SpanProcessor` implementation that supports adding new span exporters after the `SdkTracerProvider` has been built and registered. To be able to add (register) and remove (unregister) `SpanExporter` dynamically, we have implemented the `DynamicMultiSpanExporter`.

Additionally, to be able to update (set) `GlobalOpenTelemetry#globalOpenTelemetry` without using `GlobalOpenTelemetry#resetForTesting()`, we have implemented our own `OpenTelemetryImpl`. The `OpenTelemetryImpl` class is a wrapper class for the `OpenTelemetrySdk` that is used to be able to change the underlying `OpenTelemetrySdk` object in `OpenTelemetryImpl#openTelemetry`. The `OpenTelemetryImpl` can be registered as the `GlobalOpenTelemetry#globalOpenTelemetry` via `OpenTelemetryImpl#registerGlobal()` and the underlying `OpenTelemetrySdk` can be changed via `OpenTelemetryImpl#set(OpenTelemetrySdk otel)`.

How OpenTelemetry is configured and updated

When configurations for tracing or metrics exporter services change, they usually unregister and then re-register to the `OpenTelemetryControllerImpl` during an `InspectitConfigChangedEvent`. The `OpenTelemetryControllerImpl` then re-configures OpenTelemetry after all exporter services have (un-/re-)registered. For this, the respective event listeners have been annotated with the `@Order` annotation, i.e., for `DynamicallyActivatableService#checkForUpdates(InspectitConfigChangedEvent ev)`:

```
    @EventListener(InspectitConfigChangedEvent.class)
    @Order(Ordered.HIGHEST_PRECEDENCE) // make sure this is called before
  OpenTelemetryController#configureOpenTelemetry
        synchronized void checkForUpdates(InspectitConfigChangedEvent ev) {
        ...
        }
```

and for `OpentTelemetryImpl#configureOpenTelemetry(InspectitConfigChangedEvent ev)`:

```
    @EventListener(InspectitConfigChangedEvent.class)
    @Order(Ordered.LOWEST_PRECEDENCE)
    // make sure this is called after the individual services have (un)-registered
    synchronized boolean configureOpenTelemetry() {
        ...
    }
```

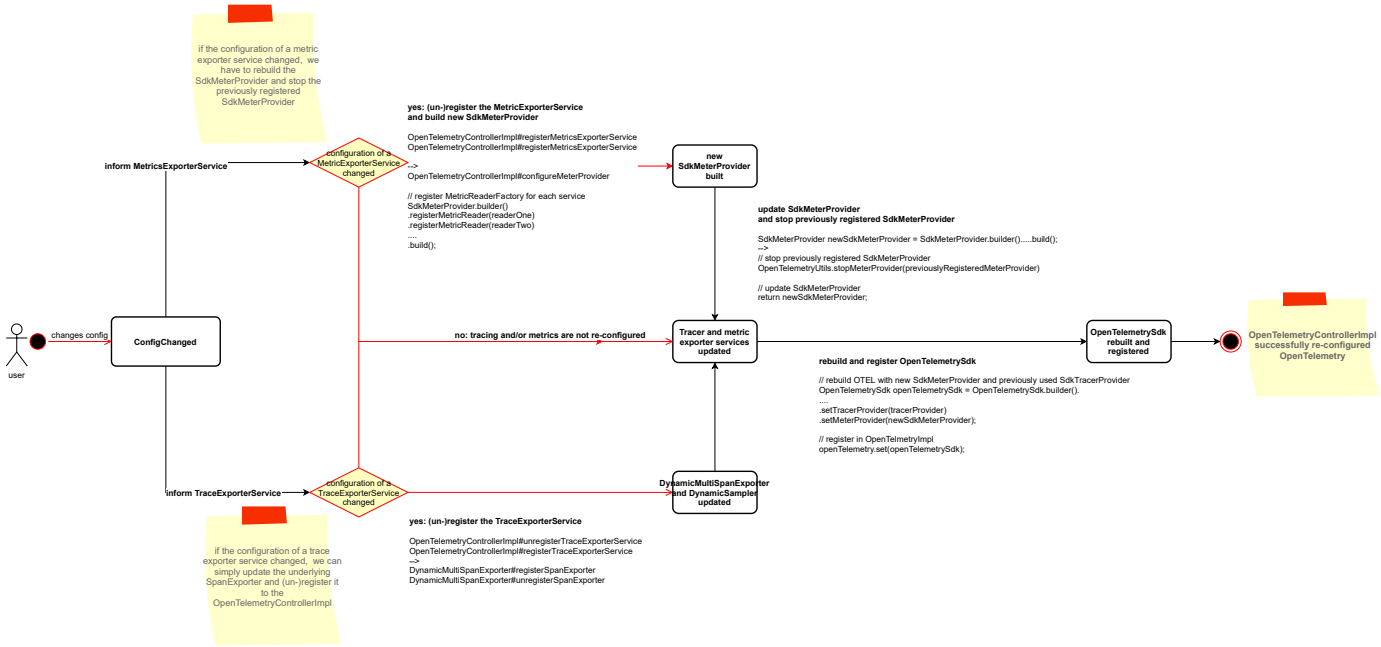
(Re-)configuring trace exporter services

Tracing is re-configured in the `OpenTelemetryControllerImpl#configureTracing(InspectitConfig configuration)` method. Due to our custom `DynamicMultiSpanExporter` and `DynamicSampler` implementation, only the sample probability is updated and no `SdkTracerProvider`/`SpanProcessor`/`SpanExporter` is (re-)built. Thus, when building a new `OpenTelemetrySdk` in `OpenTelemetryControllerImpl#configureOpenTelemetry()`, we keep using the previously built `SdkTracerProvider`.

(Re-)configuring metric exporter services

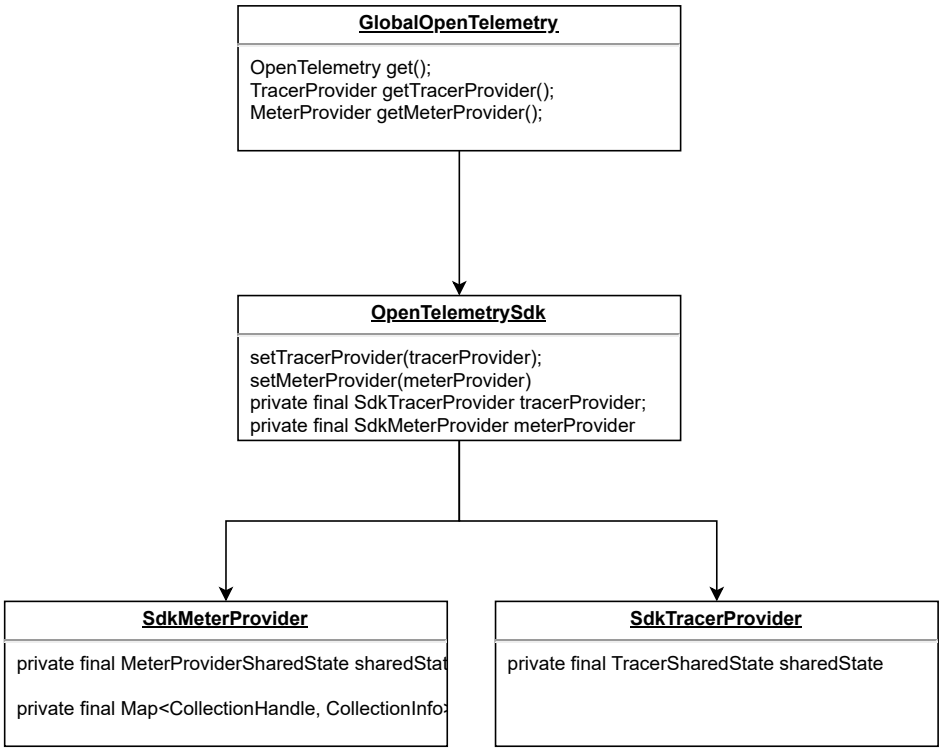
Metrics are re-configured in the `OpenTelemetryControllerImpl#configureMeterProvider(InspectitConfiguration configuration)` method. As OpenTelemetry does not give us access to underlying `MetricExporter` or `MetricReader`, we have to rebuild the `SdkMeterProvider` and register a new `MetricReaderFactory` for each registered metrics exporter service. After being rebuilt, the new `SdkMeterProvider` is then registered to our custom `MeterProviderImpl`. For this, the previously registered `SdkMeterProvider` has to be closed before, which can take up to 10 seconds (e.g., the `io.opentelemetry.exporter.prometheus.PrometheusHttpServer` takes 10 seconds to close, see `PrometheusHttpServer#close`).

An overview of the re-configuration process is given below:

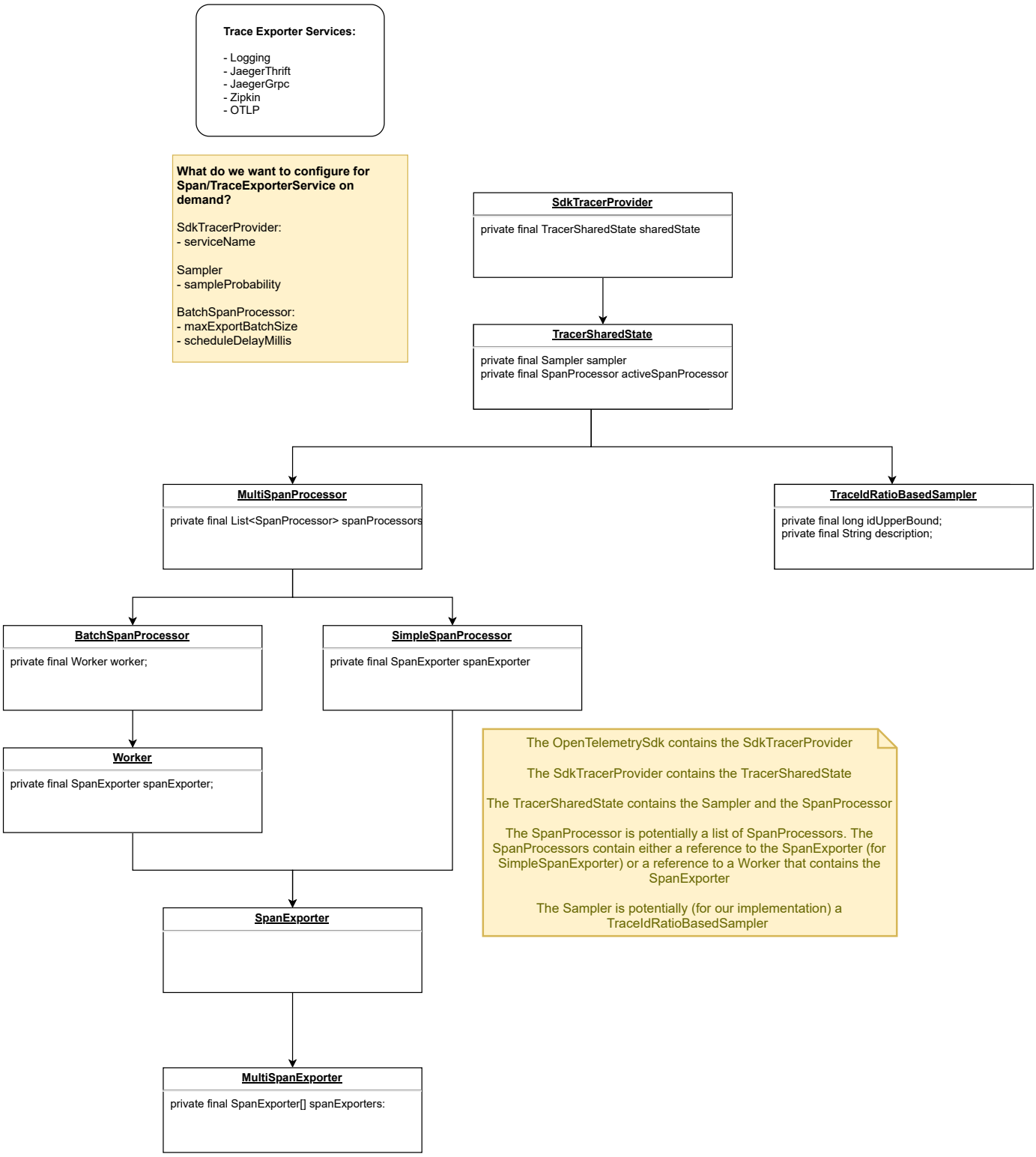


Flowchart on how OpenTelemetry is re-configured in inspectIT Ocelot

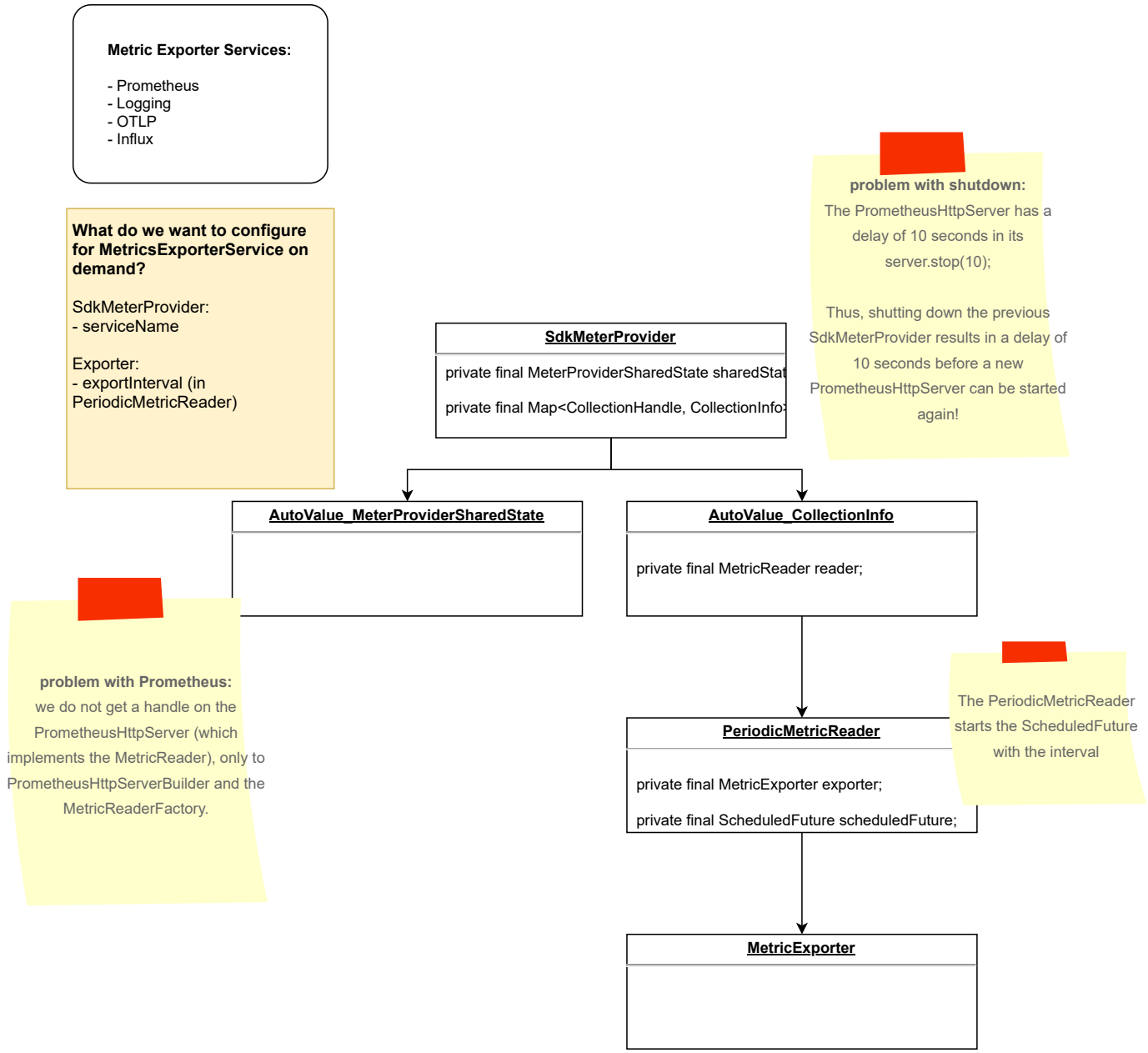
The hierarchy of `SdkMeterProvider` and `OpenTelemetrySdk` that led us to the above architecture decisions are given below:



Overview of OpenTelemetrySdk architecture (OpenTelemetrySdk)



Overview of OpenTelemetry tracing architecture (SdkTracerProvider)



Overview of OpenTelemetry metrics architeture (SdkMetricsProvider)

Things that have been removed or do not work anymore

- The `service-name` property of the trace and metrics exporter services (`JaegerExporterServiceSettings`, `ZipkinExporterSettings`, `LoggingTraceExporterSettings`, and `LoggingMetricsExporterSettings`) has been removed, as it is not supported by OpenTelemetry
  - OpenTelemetry supports a global `serviceNameResource` (`Resource.create(Attributes.of(ResourceAttributes.SERVICE_NAME, "your-service-name"))`)
  - OpenTelemetry does **not** support individual service names for different exporters
  - → Please use the global `inspectit.service-name` property instead