Spring Cloud Sleuth OTel Reference Documentation

Table of Contents

1. Legal

1.0.0-SNAPSHOT

2. Getting Started

If you are getting started with Spring Cloud Sleuth OTel or Spring in general, start by reading this section. It answers the basic “what?”, “how?” and “why?” questions. It includes an introduction to Spring Cloud Sleuth OTel, along with installation instructions. We then walk you through building your first Spring Cloud Sleuth OTel application, discussing some core principles as we go.

2.1. Introducing Spring Cloud Sleuth OTel

Spring Cloud Sleuth OTel provides integration with OpenTelemetry SDK.

2.2. Developing Your First Spring Cloud sleuth-based Application

This section describes how to develop a small “Hello World!” web application that highlights some of Spring Cloud Sleuth’s key features. We use Maven to build this project, since most IDEs support it. As the tracer implementation we’ll use OpenZipkin Brave.

You can shortcut the steps below by going to start.spring.io and choosing the "Web" and "Spring Cloud Sleuth" starters from the dependencies searcher. Doing so generates a new project structure so that you can start coding right away.

2.2.1. Creating the POM

We need to start by creating a Maven `pom.xml` file. The `pom.xml` is the recipe that is used to build your project. Open your favorite text editor and add the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <!-- Use the latest compatible Spring Boot version. You can check
https://spring.io/projects/spring-cloud for more information -->
        <version>${spring-boot-version}</version>
    </parent>

    <!-- Spring Cloud Sleuth requires a Spring Cloud BOM -->
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <!-- Provide the latest stable Spring Cloud release train version
(e.g. 2020.0.0) -->
                <version>${release.train.version}</version>
```

```
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <!-- (you don't need this if you are using a GA version) -->
    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <url>https://repo.spring.io/snapshot</url>
            <snapshots><enabled>true</enabled></snapshots>
        </repository>
        <repository>
            <id>spring-milestones</id>
            <url>https://repo.spring.io/milestone</url>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>spring-snapshots</id>
            <url>https://repo.spring.io/snapshot</url>
        </pluginRepository>
        <pluginRepository>
            <id>spring-milestones</id>
            <url>https://repo.spring.io/milestone</url>
        </pluginRepository>
    </pluginRepositories>
</project>
```

The preceding listing should give you a working build. You can test it by running `mvn package` (for now, you can ignore the “jar will be empty - no content was marked for inclusion!” warning).

At this point, you could import the project into an IDE (most modern Java IDEs include built-in support for Maven). For simplicity, we continue to use a plain text editor for this example.

2.2.2. Adding Classpath Dependencies

To add the necessary dependencies, edit your `pom.xml` and add the `spring-boot-starter-web` dependency immediately below the `parent` section. To use Sleuth with OpenTelemetry do the following.

```
<dependencies>
    <!-- Boot's Web support -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- Sleuth with Brave tracer implementation -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-sleuth</artifactId>
        <exclusions>
            <!-- Exclude Brave -->
            <exclusion>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-sleuth-brave</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <!-- Add OpenTelemetry tracer -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-sleuth-otel-autoconfigure</artifactId>
    </dependency>
</dependencies>
```

2.2.3. Writing the Code

To finish our application, we need to create a single Java file. By default, Maven compiles sources from `src/main/java`, so you need to create that directory structure and then add a file named `src/main/java/Example.java` to contain the following code:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Example {

    private static final Logger log = LoggerFactory.getLogger(Backend.class);

    @RequestMapping("/")
    String home() {
        log.info("Hello world!");
        return "Hello World!";
    }

    public static void main(String[] args) {
        SpringApplication.run(Example.class, args);
    }

}
```

Although there is not much code here, quite a lot is going on. We step through the important parts in the next few sections.

The @RestController and @RequestMapping Annotations

Spring Boot sets up the Rest Controller and makes our application bind to a Tomcat port. Spring Cloud Sleuth with OTel tracer will provide instrumentation of the incoming request.

2.2.4. Running the Example

At this point, your application should work. Since you used the `spring-boot-starter-parent` POM, you have a useful `run` goal that you can use to start the application. Type `SPRING_APPLICATION_NAME=backend mvn spring-boot:run` from the root project directory to start the application. You should see output similar to the following:

```
$ mvn spring-boot:run


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 ...
....... . . .
....... . . . (log output here)
....... . . .
........ Started Example in 2.222 seconds (JVM running for 6.514)
```

If you open a web browser to `localhost:8080`, you should see the following output:

```
Hello World!
```

If you check the logs you should see a similar output

```
2020-10-21 12:01:16.285  INFO [backend,0b6aaf642574edd3,0b6aaf642574edd3] 289589 ---
[nio-9000-exec-1] Example              : Hello world!
```

You can notice that the logging format has been updated with the following information `[backend,0b6aaf642574edd3,0b6aaf642574edd3`. This entry corresponds to `[application name,trace id, span id]`. The application name got read from the `SPRING_APPLICATION_NAME` environment variable.

Instead of logging the request in the handler explicitly, you could set `logging.level.org.springframework.web.servlet.DispatcherServlet=DEBUG`.

To gracefully exit the application, press `ctrl-c`.

2.3. Next Steps

Hopefully, this section provided some of the Spring Cloud Sleuth OTel basics and got you on your way to writing your own applications. If you are a task-oriented type of developer, you might want to jump over to spring.io and check out some of the getting started guides that solve specific “How do I do that with Spring?” problems. We also have Spring Cloud Sleuth OTel-specific “how-to” reference documentation.

Otherwise, the next logical step is to read Using Spring Cloud Sleuth OTel. If you are really impatient, you could also jump ahead and read about Spring Cloud Sleuth OTel features.

3. Using Spring Cloud Sleuth OTel

You should just use Spring Cloud Sleuth's API. Please read the documentation of Spring Cloud Sleuth in order to better understand how the API looks like.

If you are starting out with Spring Cloud Sleuth OTel, you should probably read the Getting Started guide before diving into this section.

3.1. What to Read Next

You should now understand how you can use Spring Cloud Sleuth OTel and some best practices that you should follow. You can now go on to learn about specific Spring Cloud Sleuth OTel features, or you could skip ahead and read about the integrations available in Spring Cloud Sleuth OTel.

4. Spring Cloud Sleuth OTel Features

This section dives into the details of Spring Cloud Sleuth OTel. Here you can learn about the key features that you may want to use and customize. If you have not already done so, you might want to read the "Getting Started" and "Using Spring Cloud Sleuth OTel" sections, so that you have a good grounding in the basics.

4.1. Context Propagation

Traces connect from service to service using header propagation. The default format is B3. Similar to data formats, you can configure alternate header formats also, provided trace and span IDs are compatible with B3. Most notably, this means the trace ID and span IDs are lower-case hex, not UUIDs. Besides trace identifiers, other properties (Baggage) can also be passed along with the request. Remote Baggage must be predefined, but is flexible otherwise.

To use the provided defaults you can set the `spring.sleuth.propagation.type` property. The value can be a list in which case you will propagate more tracing headers.

For OpenTelemetry we support `AWS`, `B3`, `JAEGER`, `OT_TRACER` and `W3C` via the `io.opentelemetry:opentelemetry-extension-trace-propagators` dependency that you have to manually add to your classpath.

You can read more about how to provide custom context propagation in this "how to section".

4.2. OpenTelemetry Tracer Integration

Spring Cloud Sleuth integrates with the OpenTelemetry (OTel in short) SDK tracer via the bridge that is available in the `spring-cloud-sleuth-otel` module. In this section you can read about specific OTel integrations.

You can choose to use either Sleuth's API or the OpenTelemetry API directly in your code (e.g. either Sleuth's `Tracer` or OpenTelemetry's `Tracer`). If you want to use this tracer implementation's API directly please read their documentation to learn more about it.

4.2.1. OpenTelemetry Logging Integration

We're providing an Slf4j integration via a `SpanProcessor` that injects to and removes entries (trace / span ids, baggage, tags etc.) from MDC. You can disable that via the `spring.sleuth.otel.log.slf4j.enabled=false` property.

If it's there on the classpath, we integrate with the `LoggingSpanExporter`. You can disable that integration via the `spring.sleuth.otel.log.exporter.enabled=false` property.

4.2.2. OpenTelemetry Opentracing

You can integrate with OpenTelemetry and OpenTracing via the `io.opentelemetry:opentelemetry-opentracing-shim` bridge. Just add it to the classpath and the OpenTracing `Tracer` will be set up automatically.

4.3. Sending Spans to Zipkin

Spring Cloud Sleuth provides various integrations with the OpenZipkin distributed tracing system. Regardless of the chosen tracer implementation it's enough to add `spring-cloud-sleuth-zipkin` to the classpath to start sending spans to Zipkin. You can choose whether to do that via HTTP or messaging. You can read more about how to do that in "how to section".

4.4. What to Read Next

If you want to learn more about any of the classes discussed in this section, you can browse the source code directly. If you have specific questions, see the how-to section.

5. "How-to" Guides

This section provides answers to some common "how do I do that…?" questions that often arise when using Spring Cloud Sleuth OTel. Its coverage is not exhaustive, but it does cover quite a lot.

If you have a specific problem that we do not cover here, you might want to check out stackoverflow.com to see if someone has already provided an answer. Stack Overflow is also a great place to ask new questions (please use the `spring-cloud-sleuth-otel` tag).

We are also more than happy to extend this section. If you want to add a "how-to", send us a pull request.

5.1. How to Set Up Sleuth with OpenTelemetry?

Add the Sleuth starter, exclude Brave and add OpenTelemetry dependency to the classpath.

Maven

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <!-- Provide the latest stable Spring Cloud release train version
(e.g. 2020.0.0) -->
            <version>${release.train.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-sleuth-otel-dependencies</artifactId>
            <!-- Provide the version of the Spring Cloud Sleuth OpenTelemetry
project -->
            <version>${spring-cloud-sleuth-otel.version}</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-sleuth-brave</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-otel-autoconfigure</artifactId>
</dependency>
```

Gradle

```
dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${releaseTrainVersion}"
    }
}

dependencies {
    implementation("org.springframework.cloud:spring-cloud-starter-sleuth") {
        exclude group: 'org.springframework.cloud', module: 'spring-cloud-sleuth-
brave'
    }
}
```

5.2. How to Set Up Sleuth with OpenTelemetry & Zipkin via HTTP?

Add the Sleuth starter, exclude Brave, add OTel and Zipkin to the classpath.

Maven

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <!-- Provide the latest stable Spring Cloud release train version
(e.g. 2020.0.0) -->
            <version>${release.train.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-sleuth-otel-dependencies</artifactId>
            <!-- Provide the version of the Spring Cloud Sleuth OpenTelemetry
project -->
            <version>${spring-cloud-sleuth-otel.version}</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-sleuth-brave</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-otel-autoconfigure</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

Gradle

```
dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${releaseTrainVersion}"
    }
}

dependencies {
    implementation("org.springframework.cloud:spring-cloud-starter-sleuth") {
        exclude group: 'org.springframework.cloud', module: 'spring-cloud-sleuth-
brave'
    }
    implementation "org.springframework.cloud:spring-cloud-sleuth-zipkin"
}
```

5.3. How to Set Up Sleuth with OpenTelemetry & Zipkin via Messaging?

If you want to use RabbitMQ, Kafka or ActiveMQ instead of HTTP, add the `spring-rabbit`, `spring-kafka` or `org.apache.activemq:activemq-client` dependency. The default destination name is `Zipkin`.

If using Kafka, you must add the Kafka dependency.

Maven

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <!-- Provide the latest stable Spring Cloud release train version
(e.g. 2020.0.0) -->
            <version>${release.train.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-sleuth-otel-dependencies</artifactId>
            <!-- Provide the version of the Spring Cloud Sleuth OpenTelemetry
project -->
            <version>${spring-cloud-sleuth-otel.version}</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-sleuth-brave</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-otel-autoconfigure</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

Gradle

```
dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${releaseTrainVersion}"
    }
}

dependencies {
    implementation("org.springframework.cloud:spring-cloud-starter-sleuth") {
        exclude group: 'org.springframework.cloud', module: 'spring-cloud-sleuth-
brave'
    }
    implementation "org.springframework.cloud:spring-cloud-sleuth-zipkin"
    implementation "org.springframework.kafka:spring-kafka"
}
```

Also, you need to set the property `spring.zipkin.sender.type` property accordingly:

```
spring.zipkin.sender.type: kafka
```

If you want Sleuth over RabbitMQ, add the `spring-cloud-sleuth-otel` (exclude `spring-cloud-sleuth-brave`), `spring-cloud-sleuth-zipkin` and `spring-rabbit` dependencies.

Maven

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <!-- Provide the latest stable Spring Cloud release train version
(e.g. 2020.0.0) -->
            <version>${release.train.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-sleuth-otel-dependencies</artifactId>
            <!-- Provide the version of the Spring Cloud Sleuth OpenTelemetry
project -->
            <version>${spring-cloud-sleuth-otel.version}</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
</dependency>
```

Gradle

```
dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${releaseTrainVersion}"
    }
}

dependencies {
    implementation "org.springframework.cloud:spring-cloud-starter-sleuth"
    implementation "org.springframework.cloud:spring-cloud-sleuth-zipkin"
    implementation "org.springframework.amqp:spring-rabbit"
}
```

If you want Sleuth over RabbitMQ, add the `spring-cloud-sleuth-otel` (exclude `spring-cloud-sleuth-brave`), `spring-cloud-sleuth-zipkin` and `activemq-client` dependencies.

Maven

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <!-- Provide the latest stable Spring Cloud release train version
(e.g. 2020.0.0) -->
            <version>${release.train.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-sleuth-otel-dependencies</artifactId>
            <!-- Provide the version of the Spring Cloud Sleuth OpenTelemetry
project -->
            <version>${spring-cloud-sleuth-otel.version}</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-sleuth-brave</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-otel-autoconfigure</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-client</artifactId>
</dependency>
```

Gradle

```
dependencyManagement {
    imports {
        mavenBom "org.springframework.cloud:spring-cloud-
dependencies:${releaseTrainVersion}"
    }
}

dependencies {
    implementation("org.springframework.cloud:spring-cloud-starter-sleuth") {
        exclude group: 'org.springframework.cloud', module: 'spring-cloud-sleuth-
brave'
    }
    implementation "org.springframework.cloud:spring-cloud-sleuth-zipkin"
    implementation "org.apache.activemq:activemq-client"
}
```

Also, you need to set the property `spring.zipkin.sender.type` property accordingly:

```
spring.zipkin.sender.type: activemq
```

5.4. How to Change The Context Propagation Mechanism?

To use the provided defaults you can set the `spring.sleuth.propagation.type` property. The value can be a list in which case you will propagate more tracing headers.

For OpenTelemetry we support `AWS`, `B3`, `JAEGER`, `W3C`, `OT_TRACER` and `W3C` propagation types.

If you want to provide a custom propagation mechanism set the `spring.sleuth.propagation.type` property to `CUSTOM` and implement your own bean (`Propagation.Factory` for Brave and `TextMapPropagator` for OpenTelemetry). Below you can find the examples:

```
@Component
class CustomPropagator implements TextMapPropagator {

    @Override
    public List<String> fields() {
        return Arrays.asList("myCustomTraceId", "myCustomSpanId");
    }

    @Override
    public <C> void inject(Context context, C carrier, Setter<C> setter) {
        SpanContext spanContext = Span.fromContext(context).getSpanContext();
        if (!spanContext.isValid()) {
            return;
        }
        setter.set(carrier, "myCustomTraceId",
spanContext.getTraceIdAsHexString());
        setter.set(carrier, "myCustomSpanId", spanContext.getSpanIdAsHexString());
    }

    @Override
    public <C> Context extract(Context context, C carrier, Getter<C> getter) {
        String traceParent = getter.get(carrier, "myCustomTraceId");
        if (traceParent == null) {
            return Span.getInvalid().storeInContext(context);
        }
        String spanId = getter.get(carrier, "myCustomSpanId");
        return Span.wrap(SpanContext.createFromRemoteParent(traceParent, spanId,
TraceFlags.getSampled(),
                TraceState.builder().build())).storeInContext(context);
    }

}
```

Common application properties

Various properties can be specified inside your `application.properties` file, inside your `application.yml` file, or as command line switches. This appendix provides a list of common Spring Cloud Sleuth OTel properties and references to the underlying classes that consume them.

Property contributions can come from additional jar files on your classpath, so you should not consider this an exhaustive list. Also, you can define your own properties.

Name	Default	Description
spring.zipkin.activemq.message-max-bytes	`100000`	Maximum number of bytes for a given message with spans sent to Zipkin over ActiveMQ.
spring.zipkin.activemq.queue	`zipkin`	Name of the ActiveMQ queue where spans should be sent to Zipkin.
spring.zipkin.kafka.topic	`zipkin`	Name of the Kafka topic where spans should be sent to Zipkin.
spring.zipkin.rabbitmq.addresses		Addresses of the RabbitMQ brokers used to send spans to Zipkin
spring.zipkin.rabbitmq.queue	`zipkin`	Name of the RabbitMQ queue where spans should be sent to Zipkin.
spring.zipkin.sender.type		Means of sending spans to Zipkin.