Spring Cloud Sleuth Reference Documentation

Table of Contents

!!!! IMPORTANT !!!!

Spring Cloud Sleuth's last minor version is 3.1. You can check the 3.1.x branch for the latest commits.

WARNING	Spring Cloud Sleuth will not work with Spring Boot 3.x onward. The last major version of Spring Boot that Sleuth will support is 2.x.

The core of this project got moved to Micrometer Tracing project and the instrumentations will be

moved to Micrometer and all respective projects (no longer all instrumentations will be done in a single repository.

You can check the Micrometer Tracing migration guide to learn how to migrate from Spring Cloud Sleuth to Micrometer Tracing.

1. Legal

3.1.11-SNAPSHOT

2. Getting Started

If you are getting started with Spring Cloud Sleuth 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, along with installation instructions. We then walk you through building your first Spring Cloud Sleuth application, discussing some core principles as we go.

2.1. Introducing Spring Cloud Sleuth

Spring Cloud Sleuth provides API for distributed tracing solution for Spring Cloud. It integrates with OpenZipkin Brave

Spring Cloud Sleuth is able to trace your requests and messages so that you can correlate that communication to corresponding log entries. You can also export the tracing information to an external system to visualize latency. Spring Cloud Sleuth supports OpenZipkin compatible systems directly.

2.1.1. Terminology

Spring Cloud Sleuth borrows Dapper’s terminology.

Span: The basic unit of work. For example, sending an RPC is a new span, as is sending a response to an RPC. Spans also have other data, such as descriptions, timestamped events, key-value annotations (tags), the ID of the span that caused them, and process IDs (normally IP addresses).

Spans can be started and stopped, and they keep track of their timing information. Once you create a span, you must stop it at some point in the future.

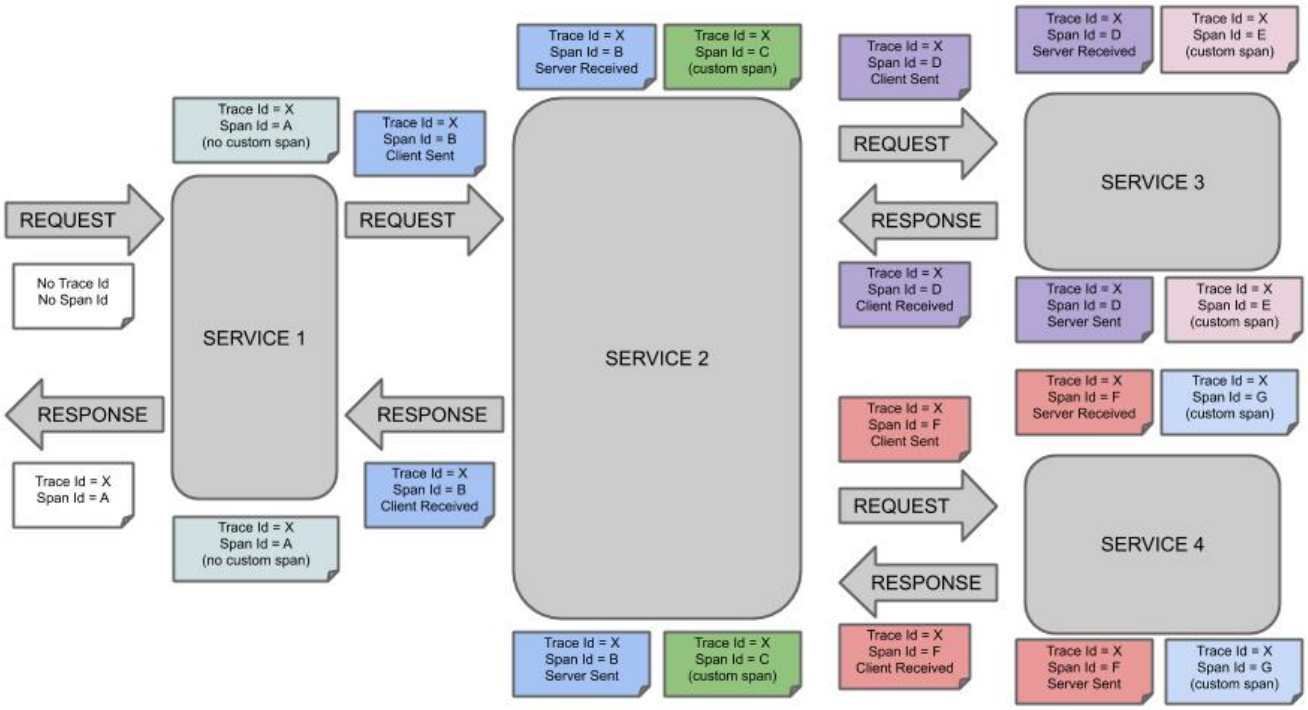
Trace: A set of spans forming a tree-like structure. For example, if you run a distributed big-data store, a trace might be formed by a `PUT` request.

Annotation/Event: Used to record the existence of an event in time.

Conceptually in a typical RPC scenario we mark these events to highlight what kind of an action took place (it doesn't mean that physically such an event will be set on a span).

- **cs**: Client Sent. The client has made a request. This annotation indicates the start of the span.
- **sr**: Server Received: The server side got the request and started processing it. Subtracting the `cs` timestamp from this timestamp reveals the network latency.
- **ss**: Server Sent. Annotated upon completion of request processing (when the response got sent back to the client). Subtracting the `sr` timestamp from this timestamp reveals the time needed by the server side to process the request.
- **cr**: Client Received. Signifies the end of the span. The client has successfully received the response from the server side. Subtracting the `cs` timestamp from this timestamp reveals the whole time needed by the client to receive the response from the server.

The following image shows how **Span** and **Trace** look in a system.



Each color of a note signifies a span (there are seven spans - from **A** to **G**). Consider the following note:

```
Trace Id = X
Span Id = D
Client Sent
```

This note indicates that the current span has **Trace Id** set to **X** and **Span Id** set to **D**. Also, from the RPC perspective, the `Client Sent` event took place.

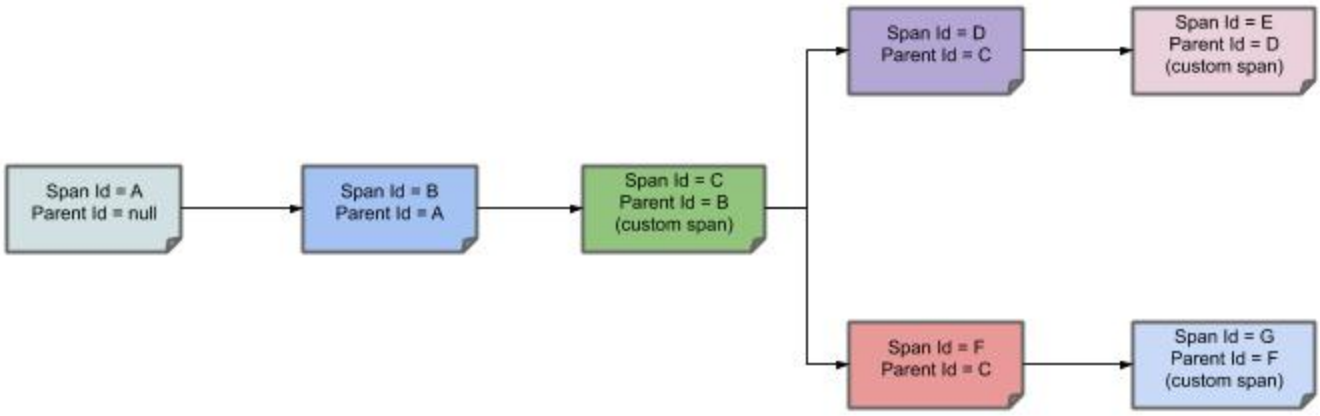
Let's consider more notes:

```
Trace Id = X
Span Id = A
(no custom span)

Trace Id = X
Span Id = C
(custom span)
```

You can continue with a created span (example with `no custom span` indication) or you can create child spans manually (example with `custom span` indication).

The following image shows how parent-child relationships of spans look:



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>$2.6.15</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:

```
<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>
    </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(Example.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 Brave 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 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-specific “how-to” reference documentation.

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

You can find the default project samples at samples.

3. Using Spring Cloud Sleuth

This section goes into more detail about how you should use Spring Cloud Sleuth. It covers topics such as controlling the span lifecycle with Spring Cloud Sleuth API or via annotations. We also cover some Spring Cloud Sleuth best practices.

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

3.1. Span Lifecycle with Spring Cloud Sleuth's API

Spring Cloud Sleuth Core in its `api` module contains all necessary interfaces to be implemented by a tracer. The project comes with OpenZipkin Brave implementation. You can check how the tracers are bridged to the Sleuth's API by looking at the `org.springframework.cloud.sleuth.brave.bridge`.

The most commonly used interfaces are:

- `org.springframework.cloud.sleuth.Tracer` - Using a tracer, you can create a root span capturing the critical path of a request.
- `org.springframework.cloud.sleuth.Span` - Span is a single unit of work that needs to be started and stopped. Contains timing information and events and tags.

You can also use your tracer implementation's API directly.

Let's look at the following Span lifecycle actions.

- start: When you start a span, its name is assigned and the start timestamp is recorded.
- end: The span gets finished (the end time of the span is recorded) and, if the span is sampled, it is eligible for collection (e.g. to Zipkin).
- continue: The span gets continued e.g. in another thread.
- create with explicit parent: You can create a new span and set an explicit parent for it.

Spring Cloud Sleuth creates an instance of `Tracer` for you. In order to use it, you can autowire it.

3.1.1. Creating and Ending Spans

You can manually create spans by using the `Tracer`, as shown in the following example:

```
// Start a span. If there was a span present in this thread it will become
// the `newSpan`'s parent.
Span newSpan = this.tracer.nextSpan().name("calculateTax");
try (Tracer.SpanInScope ws = this.tracer.withSpan(newSpan.start())) {
    // ...
    // You can tag a span
    newSpan.tag("taxValue", taxValue);
    // ...
    // You can log an event on a span
    newSpan.event("taxCalculated");
}
finally {
    // Once done remember to end the span. This will allow collecting
    // the span to send it to a distributed tracing system e.g. Zipkin
    newSpan.end();
}
```

In the preceding example, we could see how to create a new instance of the span. If there is already a span in this thread, it becomes the parent of the new span.

Always clean after you create a span.

If your span contains a name greater than 50 chars, that name is truncated to 50 chars. Your names have to be explicit and concrete. Big names lead to latency issues and sometimes even exceptions.

3.1.2. Continuing Spans

Sometimes, you do not want to create a new span but you want to continue one. An example of such a situation might be as follows:

- **AOP**: If there was already a span created before an aspect was reached, you might not want to create a new span.

To continue a span, you can store the span in one thread and pass it on to another one as shown in the example below.

```
Span spanFromThreadX = this.tracer.nextSpan().name("calculateTax");
try (Tracer.SpanInScope ws = this.tracer.withSpan(spanFromThreadX.start())) {
    executorService.submit(() -> {
        // Pass the span from thread X
        Span continuedSpan = spanFromThreadX;
        // ...
        // You can tag a span
        continuedSpan.tag("taxValue", taxValue);
        // ...
        // You can log an event on a span
        continuedSpan.event("taxCalculated");
    }).get();
}
finally {
    spanFromThreadX.end();
}
```

3.1.3. Creating a Span with an explicit Parent

You might want to start a new span and provide an explicit parent of that span. Assume that the parent of a span is in one thread and you want to start a new span in another thread. Whenever you call `Tracer.nextSpan()`, it creates a span in reference to the span that is currently in scope. You can put the span in scope and then call `Tracer.nextSpan()`, as shown in the following example:

```
// let's assume that we're in a thread Y and we've received
// the `initialSpan` from thread X. `initialSpan` will be the parent
// of the `newSpan`
Span newSpan = null;
try (Tracer.SpanInScope ws = this.tracer.withSpan(initialSpan)) {
    newSpan = this.tracer.nextSpan().name("calculateCommission");
    // ...
    // You can tag a span
    newSpan.tag("commissionValue", commissionValue);
    // ...
    // You can log an event on a span
    newSpan.event("commissionCalculated");
}
finally {
    // Once done remember to end the span. This will allow collecting
    // the span to send it to e.g. Zipkin. The tags and events set on the
    // newSpan will not be present on the parent
    if (newSpan != null) {
        newSpan.end();
    }
}
```

After creating such a span, you must finish it. Otherwise it is not reported (e.g. to Zipkin).

You can also use the `Tracer.nextSpan(Span parentSpan)` version to provide the parent span explicitly.

3.2. Naming Spans

Picking a span name is not a trivial task. A span name should depict an operation name. The name should be low cardinality, so it should not include identifiers.

Since there is a lot of instrumentation going on, some span names are artificial:

- `controller-method-name` when received by a Controller with a method name of `controllerMethodName`
- `async` for asynchronous operations done with wrapped `Callable` and `Runnable` interfaces.
- Methods annotated with `@Scheduled` return the simple name of the class.

Fortunately, for asynchronous processing, you can provide explicit naming.

3.2.1. `@SpanName` Annotation

You can name the span explicitly by using the `@SpanName` annotation, as shown in the following example:

```
@SpanName("calculateTax")
class TaxCountingRunnable implements Runnable {

    @Override
    public void run() {
        // perform logic
    }

}
```

In this case, when processed in the following manner, the span is named `calculateTax`:

```
Runnable runnable = new TraceRunnable(this.tracer, spanNamer, new
TaxCountingRunnable());
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

3.2.2. `toString()` Method

It is pretty rare to create separate classes for `Runnable` or `Callable`. Typically, one creates an anonymous instance of those classes. You cannot annotate such classes. To overcome that

limitation, if there is no `@SpanName` annotation present, we check whether the class has a custom implementation of the `toString()` method.

Running such code leads to creating a span named `calculateTax`, as shown in the following example:

```
Runnable runnable = new TraceRunnable(this.tracer, spanNamer, new Runnable() {
    @Override
    public void run() {
        // perform logic
    }

    @Override
    public String toString() {
        return "calculateTax";
    }
});
Future<?> future = executorService.submit(runnable);
// ... some additional logic ...
future.get();
```

3.3. Managing Spans with Annotations

There are a number of good reasons to manage spans with annotations, including:

- API-agnostic means to collaborate with a span. Use of annotations lets users add to a span with no library dependency on a span api. Doing so lets Sleuth change its core API to create less impact to user code.
- Reduced surface area for basic span operations. Without this feature, you must use the span api, which has lifecycle commands that could be used incorrectly. By only exposing scope, tag, and log functionality, you can collaborate without accidentally breaking span lifecycle.
- Collaboration with runtime generated code. With libraries such as Spring Data and Feign, the implementations of interfaces are generated at runtime. Consequently, span wrapping of objects was tedious. Now you can provide annotations over interfaces and the arguments of those interfaces.

3.3.1. Creating New Spans

If you do not want to create local spans manually, you can use the `@NewSpan` annotation. Also, we provide the `@SpanTag` annotation to add tags in an automated fashion.

Now we can consider some examples of usage.

```
@NewSpan
void testMethod();
```

Annotating the method without any parameter leads to creating a new span whose name equals the annotated method name.

```
@NewSpan("customNameOnTestMethod4")
void testMethod4();
```

If you provide the value in the annotation (either directly or by setting the `name` parameter), the created span has the provided value as the name.

```
// method declaration
@NewSpan(name = "customNameOnTestMethod5")
void testMethod5(@SpanTag("testTag") String param);

// and method execution
this.testBean.testMethod5("test");
```

You can combine both the name and a tag. Let's focus on the latter. In this case, the value of the annotated method's parameter runtime value becomes the value of the tag. In our sample, the tag key is `testTag`, and the tag value is `test`.

```
@NewSpan(name = "customNameOnTestMethod3")
@Override
public void testMethod3() {
}
```

You can place the `@NewSpan` annotation on both the class and an interface. If you override the interface's method and provide a different value for the `@NewSpan` annotation, the most concrete one wins (in this case `customNameOnTestMethod3` is set).

3.3.2. Continuing Spans

If you want to add tags and annotations to an existing span, you can use the `@ContinueSpan` annotation, as shown in the following example:

```
// method declaration
@ContinueSpan(log = "testMethod11")
void testMethod11(@SpanTag("testTag11") String param);

// method execution
this.testBean.testMethod11("test");
this.testBean.testMethod13();
```

(Note that, in contrast with the `@NewSpan` annotation ,you can also add logs with the `log` parameter.)

That way, the span gets continued and:

- Log entries named `testMethod11.before` and `testMethod11.after` are created.
- If an exception is thrown, a log entry named `testMethod11.afterFailure` is also created.
- A tag with a key of `testTag11` and a value of `test` is created.

3.3.3. Advanced Tag Setting

There are 3 different ways to add tags to a span. All of them are controlled by the `SpanTag` annotation. The precedence is as follows:

1. Try with a bean of `TagValueResolver` type and a provided name.
2. If the bean name has not been provided, try to evaluate an expression. We search for a `TagValueExpressionResolver` bean. The default implementation uses SPEL expression resolution. **IMPORTANT** You can only reference properties from the SPEL expression. Method execution is not allowed due to security constraints.
3. If we do not find any expression to evaluate, return the `toString()` value of the parameter.

Custom Extractor

The value of the tag for the following method is computed by an implementation of `TagValueResolver` interface. Its class name has to be passed as the value of the `resolver` attribute.

Consider the following annotated method:

```
@NewSpan
public void getAnnotationForTagValueResolver(
        @SpanTag(key = "test", resolver = TagValueResolver.class) String test) {
}
```

Now further consider the following `TagValueResolver` bean implementation:

```
@Bean(name = "myCustomTagValueResolver")
public TagValueResolver tagValueResolver() {
    return parameter -> "Value from myCustomTagValueResolver";
}
```

The two preceding examples lead to setting a tag value equal to `Value from myCustomTagValueResolver`.

Resolving Expressions for a Value

Consider the following annotated method:

```
@NewSpan
public void getAnnotationForTagValueExpression(
        @SpanTag(key = "test", expression = "'hello' + ' characters'") String test) {
}
```

No custom implementation of a `TagValueExpressionResolver` leads to evaluation of the SPEL expression, and a tag with a value of `hello characters` is set on the span. If you want to use some other expression resolution mechanism, you can create your own implementation of the bean.

Using The `toString()` Method

Consider the following annotated method:

```
@NewSpan
public void getAnnotationForArgumentToString(@SpanTag("test") Long param) {
}
```

Running the preceding method with a value of `15` leads to setting a tag with a String value of `"15"`.

3.4. What to Read Next

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

4. Spring Cloud Sleuth Features

This section dives into the details of Spring Cloud Sleuth. 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" 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 Brave we support `AWS`, `B3`, `W3C` propagation types.

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

4.2. Sampling

Spring Cloud Sleuth pushes the sampling decision down to the tracer implementation. However, there are cases where you can change the sampling decision at runtime.

One of such cases is skip reporting of certain client spans. To achieve that you can set the `spring.sleuth.web.client.skip-pattern` with the path patterns to be skipped. Another option is to provide your own custom `org.springframework.cloud.sleuth.SamplerFunction<`org.springframework.cloud.sleuth.http.HttpRequest>` implementation and define when a given `HttpRequest` should not be sampled.

4.3. Baggage

Distributed tracing works by propagating fields inside and across services that connect the trace together: traceId and spanId notably. The context that holds these fields can optionally push other fields that need to be consistent regardless of many services are touched. The simple name for these extra fields is "Baggage".

Sleuth allows you to define which baggage are permitted to exist in the trace context, including what header names are used.

The following example shows setting baggage values using Spring Cloud Sleuth's API:

```
try (Tracer.SpanInScope ws = this.tracer.withSpan(initialSpan)) {
    BaggageInScope businessProcess =
this.tracer.createBaggage(BUSINESS_PROCESS).set("ALM");
    BaggageInScope countryCode = this.tracer.createBaggage(COUNTRY_CODE).set("FO");
    try {
```

There is currently no limitation of the count or size of baggage items. Keep in mind that too many can decrease system throughput or increase RPC latency. In extreme cases, too much baggage can crash the application, due to exceeding transport-level message or header capacity.

You can use properties to define fields that have no special configuration such as name mapping:

- `spring.sleuth.baggage.remote-fields` is a list of header names to accept and propagate to remote services.
- `spring.sleuth.baggage.local-fields` is a list of names to propagate locally

No prefixing applies with these keys. What you set is literally what is used.

A name set in either of these properties will result in a `Baggage` of the same name.

In order to automatically set the baggage values to Slf4j's MDC, you have to set the `spring.sleuth.baggage.correlation-fields` property with a list of allowed local or remote keys. E.g. `spring.sleuth.baggage.correlation-fields=country-code` will set the value of the `country-code` baggage into MDC.

Note that the extra field is propagated and added to MDC starting with the next downstream trace context. To immediately add the extra field to MDC in the current trace context, configure the field to flush on update:

```
// configuration
@Bean
BaggageField countryCodeField() {
    return BaggageField.create("country-code");
}

@Bean
ScopeDecorator mdcScopeDecorator() {
    return MDCScopeDecorator.newBuilder()
            .clear()
            .add(SingleCorrelationField.newBuilder(countryCodeField())
                    .flushOnUpdate()
                    .build())
            .build();
}

// service
@Autowired
BaggageField countryCodeField;

countryCodeField.updateValue("new-value");
```

Remember that adding entries to MDC can drastically decrease the performance of your application!

If you want to add the baggage entries as tags, to make it possible to search for spans via the baggage entries, you can set the value of `spring.sleuth.baggage.tag-fields` with a list of allowed baggage keys. To disable the feature you have to pass the `spring.sleuth.propagation.tag.enabled=false` property.

4.3.1. Baggage versus Tags

Like trace IDs, Baggage is attached to messages or requests, usually as headers. Tags are key value pairs sent in a Span to Zipkin. Baggage values are not added spans by default, which means you can’t search based on Baggage unless you opt-in.

To make baggage also tags, use the property `spring.sleuth.baggage.tag-fields` like so:

```
spring:
  sleuth:
    baggage:
      foo: bar
      remoteFields:
        - country-code
        - x-vcap-request-id
      tagFields:
        - country-code
```

4.4. OpenZipkin Brave Tracer Integration

Spring Cloud Sleuth integrates with the OpenZipkin Brave tracer via the bridge that is available in the `spring-cloud-sleuth-brave` module. In this section you can read about specific Brave integrations.

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

4.4.1. Brave Basics

Here are the most core types you might use:

- `brave.SpanCustomizer` - to change the span currently in progress
- `brave.Tracer` - to get a start new spans ad-hoc

Here are the most relevant links from the OpenZipkin Brave project:

- Brave's core library
- Baggage (propagated fields)
- HTTP tracing

4.4.2. Brave Sampling

Sampling only applies to tracing backends, such as Zipkin. Trace IDs appear in logs regardless of sample rate. Sampling is a way to prevent overloading the system, by consistently tracing some, but not all requests.

The default rate of 10 traces per second is controlled by the `spring.sleuth.sampler.rate` property and applies when we know Sleuth is used for reasons besides logging. Use a rate above 100 traces per second with extreme caution as it can overload your tracing system.

The sampler can be set by Java Config also, as shown in the following example:

```
@Bean
public Sampler defaultSampler() {
    return Sampler.ALWAYS_SAMPLE;
}
```

You can set the HTTP header `b3` to `1`, or, when doing messaging, you can set the `spanFlags` header to `1`. Doing so forces the current request to be sampled regardless of configuration.

By default samplers will work with the refresh scope mechanism. That means that you can change the sampling properties at runtime, refresh the application and the changes will be reflected. However, sometimes the fact of creating a proxy around samplers and calling it from too early (from `@PostConstruct` annotated method) may lead to dead locks. In such a case either create a sampler bean explicitly, or set the property `spring.sleuth.sampler.refresh.enabled` to `false` to disable the refresh scope support.

4.4.3. Brave Baggage Java configuration

If you need to do anything more advanced than above, do not define properties and instead use a `@Bean` config for the baggage fields you use.

- `BaggagePropagationCustomizer` sets up baggage fields
- Add a `SingleBaggageField` to control header names for a `Baggage`.
- `CorrelationScopeCustomizer` sets up MDC fields
- Add a `SingleCorrelationField` to change the MDC name of a `Baggage` or if updates flush.

4.4.4. Brave Customizations

The `brave.Tracer` object is fully managed by sleuth, so you rarely need to affect it. That said, Sleuth supports a number of `Customizer` types, that allow you to configure anything not already done by Sleuth with auto-configuration or properties.

If you define one of the following as a `Bean`, Sleuth will invoke it to customize behaviour:

- `RpcTracingCustomizer` - for RPC tagging and sampling policy
- `HttpTracingCustomizer` - for HTTP tagging and sampling policy
- `MessagingTracingCustomizer` - for messaging tagging and sampling policy
- `CurrentTraceContextCustomizer` - to integrate decorators such as correlation.
- `BaggagePropagationCustomizer` - for propagating baggage fields in process and over headers
- `CorrelationScopeDecoratorCustomizer` - for scope decorations such as MDC (logging) field correlation

Brave Sampling Customizations

If client /server sampling is required, just register a bean of type `brave.sampler.SamplerFunction<HttpRequest>` and name the bean `sleuthHttpClientSampler` for client sampler and `sleuthHttpServerSampler` for server sampler.

For your convenience the `@HttpClientSampler` and `@HttpServerSampler` annotations can be used to inject the proper beans or to reference the bean names via their static String `NAME` fields.

Check out Brave's code to see an example of how to make a path-based sampler github.com/openzipkin/brave/tree/master/instrumentation/http#sampling-policy

If you want to completely rewrite the `HttpTracing` bean you can use the `SkipPatternProvider` interface to retrieve the URL `Pattern` for spans that should be not sampled. Below you can see an example of usage of `SkipPatternProvider` inside a server side, `Sampler<HttpRequest>`.

```
@Configuration(proxyBeanMethods = false)
    class Config {
  @Bean(name = HttpServerSampler.NAME)
  SamplerFunction<HttpRequest> myHttpSampler(SkipPatternProvider provider) {
      Pattern pattern = provider.skipPattern();
      return request -> {
          String url = request.path();
          boolean shouldSkip = pattern.matcher(url).matches();
          if (shouldSkip) {
              return false;
          }
          return null;
      };
  }
}
```

4.4.5. Brave Messaging

Sleuth automatically configures the `MessagingTracing` bean which serves as a foundation for Messaging instrumentation such as Kafka or JMS.

If a customization of producer / consumer sampling of messaging traces is required, just register a bean of type `brave.sampler.SamplerFunction<MessagingRequest>` and name the bean `sleuthProducerSampler` for producer sampler and `sleuthConsumerSampler` for consumer sampler.

For your convenience the `@ProducerSampler` and `@ConsumerSampler` annotations can be used to inject the proper beans or to reference the bean names via their static String `NAME` fields.

Ex. Here's a sampler that traces 100 consumer requests per second, except for the "alerts" channel. Other requests will use a global rate provided by the `Tracing` component.

```
@Configuration(proxyBeanMethods = false)
    class Config {
  @Bean(name = ConsumerSampler.NAME)
  SamplerFunction<MessagingRequest> myMessagingSampler() {
      return MessagingRuleSampler.newBuilder().putRule(channelNameEquals("alerts"),
Sampler.NEVER_SAMPLE)
              .putRule(Matchers.alwaysMatch(),
RateLimitingSampler.create(100)).build();
  }
}
```

For more, see github.com/openzipkin/brave/tree/master/instrumentation/messaging#sampling-policy

4.4.6. Brave Opentracing

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

4.5. 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".

When the span is closed, it is sent to Zipkin over HTTP. The communication is asynchronous. You can configure the URL by setting the `spring.zipkin.baseUrl` property, as follows:

```
spring.zipkin.baseUrl: https://192.168.99.100:9411/
```

If you want to find Zipkin through service discovery, you can pass the Zipkin's service ID inside the URL, as shown in the following example for `zipkinserver` service ID:

```
spring.zipkin.baseUrl: https://zipkinserver/
```

To disable this feature just set `spring.zipkin.discovery-client-enabled` to `false`.

When the Discovery Client feature is enabled, Sleuth uses `LoadBalancerClient` to find the URL of the Zipkin Server. It means that you can set up the load balancing configuration.

If you have `web`, `rabbit`, `activemq` or `kafka` together on the classpath, you might need to pick the means by which you would like to send spans to zipkin. To do so, set `web`, `rabbit`, `activemq` or `kafka` to the `spring.zipkin.sender.type` property. The following example shows setting the sender type for `web`:

```
spring.zipkin.sender.type: web
```

If you're running a non-reactive application we will use a `RestTemplate` based span sender. Otherwise a `WebClient` based span sender will be chosen.

To customize the `RestTemplate` that sends spans to Zipkin via HTTP, you can register the `ZipkinRestTemplateCustomizer` bean in your `@Configuration` annotated Spring configuration class.

```
@Bean
ZipkinRestTemplateCustomizer myZipkinRestTemplateCustomizer() {
    return new ZipkinRestTemplateCustomizer() {
        @Override
        public RestTemplate customizeTemplate(RestTemplate restTemplate) {
            // customize the RestTemplate
            return restTemplate;
        }
    };
}
```

If, however, you would like to control the full process of creating the `RestTemplate` object, you will have to create a bean of `ZipkinRestTemplateProvider` type.

```
@Bean
ZipkinRestTemplateProvider myZipkinRestTemplateProvider() {
    return MyRestTemplate::new;
}
```

By default, api path will be set to `api/v2/spans` or `api/v1/spans` depending on the encoder version. If you want to use a custom api path, you can configure it using the following property (empty case, set ""):

```
spring.zipkin.api-path: v2/path2
```

In case of a reactive application, we're creating a simple `WebClient.Builder` instance. If you want to provide your own or reuse an existing one you need to create an instance of a `ZipkinWebClientBuilderProvider` bean.

```
@Bean
ZipkinWebClientBuilderProvider myZipkinWebClientBuilderProvider() {
    // create your own instance or inject one from the Spring Context
    return () -> WebClient.builder();
}
```

4.5.1. Custom service name

By default, Sleuth assumes that, when you send a span to Zipkin, you want the span's service name to be equal to the value of the `spring.application.name` property. That is not always the case, though. There are situations in which you want to explicitly provide a different service name for all spans coming from your application. To achieve that, you can pass the following property to your application to override that value (the example is for a service named `myService`):

```
spring.zipkin.service.name: myService
```

4.5.2. Host Locator

This section is about defining **host** from service discovery. It is **NOT** about finding Zipkin through service discovery.

To define the host that corresponds to a particular span, we need to resolve the host name and port. The default approach is to take these values from server properties. If those are not set, we try to retrieve the host name from the network interfaces.

If you have the discovery client enabled and prefer to retrieve the host address from the registered instance in a service registry, you have to set the `spring.zipkin.locator.discovery.enabled` property (it is applicable for both HTTP-based and Stream-based span reporting), as follows:

```
spring.zipkin.locator.discovery.enabled: true
```

4.5.3. Customization of Reported Spans

In Sleuth, we generate spans with a fixed name. Some users want to modify the name depending on values of tags.

Sleuth registers a `SpanFilter` bean that can automatically skip reporting spans of given name patterns. The property `spring.sleuth.span-filter.span-name-patterns-to-skip` contains the default skip patterns for span names. The property `spring.sleuth.span-filter.additional-span-name-patterns-to-skip` will append the provided span name patterns to the existing ones. In order to disable this functionality just set `spring.sleuth.span-filter.enabled` to `false`.

Brave Customization of Reported Spans

This section is applicable for Brave tracer only.

Before reporting spans (for example, to Zipkin) you may want to modify that span in some way. You can do so by implementing a `SpanHandler`.

The following example shows how to register two beans that implement `SpanHandler`:

```
@Bean
SpanHandler handlerOne() {
    return new SpanHandler() {
        @Override
        public boolean end(TraceContext traceContext, MutableSpan span, Cause cause) {
            span.name("foo");
            return true; // keep this span
        }
    };
}

@Bean
SpanHandler handlerTwo() {
    return new SpanHandler() {
        @Override
        public boolean end(TraceContext traceContext, MutableSpan span, Cause cause) {
            span.name(span.name() + " bar");
            return true; // keep this span
        }
    };
}
```

The preceding example results in changing the name of the reported span to `foo bar`, just before it gets reported (for example, to Zipkin).

4.5.4. Overriding the auto-configuration of Zipkin

Spring Cloud Sleuth supports sending traces to multiple tracing systems as of version 2.1.0. In order to get this to work, every tracing system needs to have a `Reporter<Span>` and `Sender`. If you want to override the provided beans you need to give them a specific name. To do this you can use respectively `ZipkinAutoConfiguration.REPORTER_BEAN_NAME` and `ZipkinAutoConfiguration.SENDER_BEAN_NAME`.

```
@Configuration(proxyBeanMethods = false)
protected static class MyConfig {

    @Bean(ZipkinAutoConfiguration.REPORTER_BEAN_NAME)
    Reporter<zipkin2.Span>
myReporter(@Qualifier(ZipkinAutoConfiguration.SENDER_BEAN_NAME) MySender mySender) {
        return AsyncReporter.create(mySender);
    }

    @Bean(ZipkinAutoConfiguration.SENDER_BEAN_NAME)
    MySender mySender() {
        return new MySender();
    }

    static class MySender extends Sender {

        private boolean spanSent = false;

        boolean isSpanSent() {
            return this.spanSent;
        }

        @Override
        public Encoding encoding() {
            return Encoding.JSON;
        }

        @Override
        public int messageMaxBytes() {
            return Integer.MAX_VALUE;
        }

        @Override
        public int messageSizeInBytes(List<byte[]> encodedSpans) {
            return encoding().listSizeInBytes(encodedSpans);
        }

        @Override
        public Call<Void> sendSpans(List<byte[]> encodedSpans) {
            this.spanSent = true;
            return Call.create(null);
        }

    }

}
```

4.6. Log integration

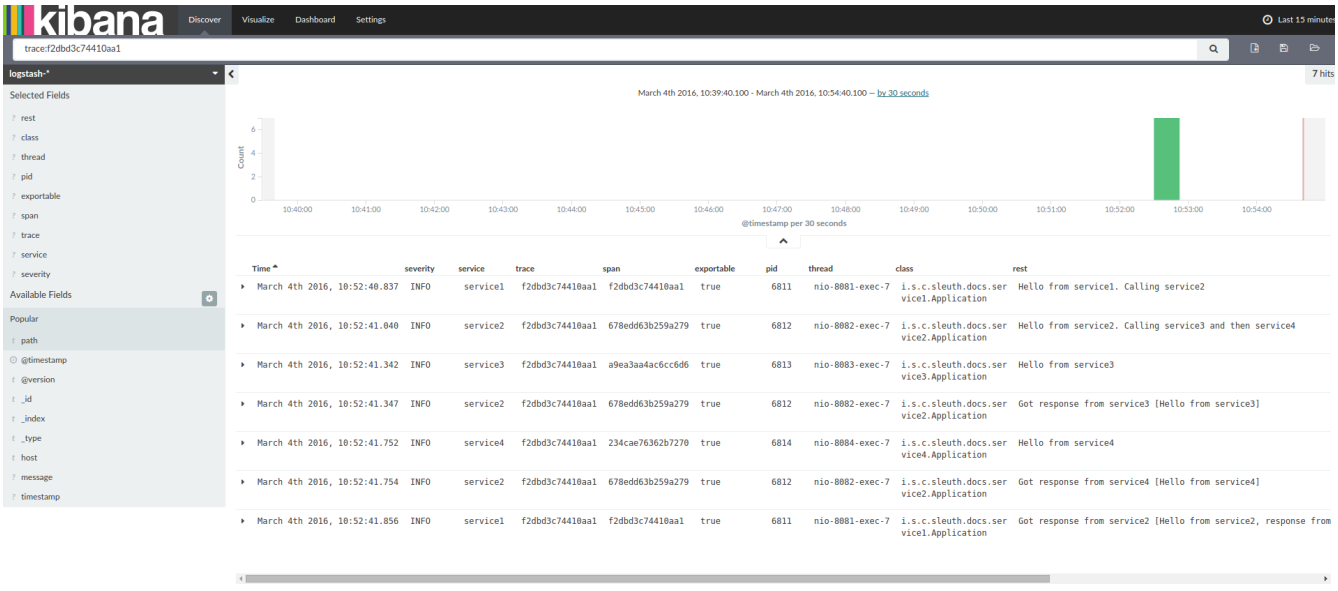
Sleuth configures the logging context with variables including the service name (`%{spring.zipkin.service.name}` or `%{spring.application.name}` if the previous one was not set), span ID (`%{spanId}`) and the trace ID (`%{traceId}`). These help you connect logs with distributed traces and allow you choice in what tools you use to troubleshoot your services.

Once you find any log with an error, you can look for the trace ID in the message. Paste that into your distributed tracing system to visualize the entire trace, regardless of how many services the first request ended up hitting.

```
backend.log:  2020-04-09 17:45:40.516 ERROR
[backend,5e8eeec48b08e26882aba313eb08f0a4,dcc1df555b5777b3] 97203 --- [nio-9000-exec-
1] o.s.c.s.i.web.ExceptionLoggingFilter     : Uncaught exception thrown
frontend.log:2020-04-09 17:45:40.574 ERROR
[frontend,5e8eeec48b08e26882aba313eb08f0a4,82aba313eb08f0a4] 97192 --- [nio-8081-exec-
2] o.s.c.s.i.web.ExceptionLoggingFilter     : Uncaught exception thrown
```

Above, you'll notice the trace ID is `5e8eeec48b08e26882aba313eb08f0a4`, for example. This log configuration was automatically setup by Sleuth. You can disable it by disabling Sleuth via `spring.sleuth.enabled=false` property or putting your own `logging.pattern.level` property.

If you use a log aggregating tool (such as Kibana, Splunk, and others), you can order the events that took place. An example from Kibana would resemble the following image:



If you want to use Logstash, the following listing shows the Grok pattern for Logstash:

```
filter {
  # pattern matching logback pattern
  grok {
    match => { "message" =>
"%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%{DATA:trac
e},%{DATA:span}\]\s+%{DATA:pid}\s+---
\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }
  }
  date {
    match => ["timestamp", "ISO8601"]
  }
  mutate {
    remove_field => ["timestamp"]
  }
}
```

If you want to use Grok together with the logs from Cloud Foundry, you have to use the following pattern:

```
filter {
  # pattern matching logback pattern
  grok {
    match => { "message" =>
"(?m)OUT\s+%{TIMESTAMP_ISO8601:timestamp}\s+%{LOGLEVEL:severity}\s+\[%{DATA:service},%
{DATA:trace},%{DATA:span}\]\s+%{DATA:pid}\s+---
\s+\[%{DATA:thread}\]\s+%{DATA:class}\s+:\s+%{GREEDYDATA:rest}" }
  }
  date {
    match => ["timestamp", "ISO8601"]
  }
  mutate {
    remove_field => ["timestamp"]
  }
}
```

4.6.1. JSON Logback with Logstash

Often, you do not want to store your logs in a text file but in a JSON file that Logstash can immediately pick. To do so, you have to do the following (for readability, we pass the dependencies in the `groupId:artifactId:version` notation).

Dependencies Setup

1. Ensure that Logback is on the classpath (`ch.qos.logback:logback-core`).
2. Add Logstash Logback encode. For example, to use version `4.6`, add `net.logstash.logback:logstash-logback-encoder:4.6`.

Logback Setup

Consider the following example of a Logback configuration file (logback-spring.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml"/>
    <springProperty scope="context" name="springAppName"
source="spring.application.name"/>
    <!-- Example for logging into the build folder of your project -->
    <property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}"/>

    <!-- You can override this to have a custom pattern -->
    <property name="CONSOLE_LOG_PATTERN"
              value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint}
%clr(${LOG_LEVEL_PATTERN:-%5p}) %clr(${PID:- }){magenta} %clr(---){faint}
%clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint}
%m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}"/>

    <!-- Appender to log to console -->
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <!-- Minimum logging level to be presented in the console logs-->
            <level>DEBUG</level>
        </filter>
        <encoder>
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
            <charset>utf8</charset>
        </encoder>
    </appender>

    <!-- Appender to log to file -->
    <appender name="flatfile" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${LOG_FILE}</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${LOG_FILE}.%d{yyyy-MM-dd}.gz</fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder>
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
            <charset>utf8</charset>
        </encoder>
    </appender>
    <!-- Appender to log to file in a JSON format -->
    <appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${LOG_FILE}.json</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder
```

```
class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
            <providers>
                <timestamp>
                    <timeZone>UTC</timeZone>
                </timestamp>
                <pattern>
                    <pattern>
                        {
                        "timestamp": "@timestamp",
                        "severity": "%level",
                        "service": "${springAppName:-}",
                        "trace": "%X{traceId:-}",
                        "span": "%X{spanId:-}",
                        "pid": "${PID:-}",
                        "thread": "%thread",
                        "class": "%logger{40}",
                        "rest": "%message"
                        }
                    </pattern>
                </pattern>
            </providers>
        </encoder>
    </appender>
    <root level="INFO">
        <appender-ref ref="console"/>
        <!-- uncomment this to have also JSON logs -->
        <!--<appender-ref ref="logstash"/>-->
        <!--<appender-ref ref="flatfile"/>-->
    </root>
</configuration>
```

That Logback configuration file:

- Logs information from the application in a JSON format to a `build/${spring.application.name}.json` file.
- Has commented out two additional appenders: console and standard log file.
- Has the same logging pattern as the one presented in the previous section.

If you use a custom `logback-spring.xml`, you must pass the `spring.application.name` in the `bootstrap` rather than the `application` property file. Otherwise, your custom logback file does not properly read the property.

4.7. Self Documenting Spans

A declarative format of representing span configuration was introduced via the `DocumentedSpan` abstraction. By analyzing Sleuth's source code an appendix with all span characteristics is created (including allowed tag keys and event names). You can check the Sleuth Spans appendix for more information.

4.8. Traces Actuator Endpoint

Spring Cloud Sleuth comes with a `traces` Actuator endpoint that can store finished spans. The endpoint can be queried either via an HTTP Get method to simply retrieve the list of stored spans or via HTTP Post method to retrieve the list and clear it.

The size of the queue where the spans are stored can be configured via the `management.endpoint.traces.queue-size` property.

Please read the Spring Boot Actuator: Production-ready Features section of the documentation to read more about the Actuator endpoints configuration options.

4.9. 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.

If you are comfortable with Spring Cloud Sleuth's core features, you can continue on and read about Spring Cloud Sleuth's integrations.

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. 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` 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 Brave?

Add the Sleuth starter to the classpath.

Maven

```
<dependencyManagement>
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${release.train-version}</version>
              <type>pom</type>
              <scope>import</scope>
          </dependency>
      </dependencies>
</dependencyManagement>

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

Gradle

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

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

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

Add the Sleuth starter and Zipkin to the classpath.

Maven

```
<dependencyManagement>
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${release.train-version}</version>
              <type>pom</type>
              <scope>import</scope>
          </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>
```

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"
}
```

5.3. How to Set Up Sleuth with Brave & 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>
              <version>${release.train-version}</version>
              <type>pom</type>
              <scope>import</scope>
          </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.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"
    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-starter-sleuth`, `spring-cloud-sleuth-`

`zipkin` and `spring-rabbit` dependencies.

Maven

```
<dependencyManagement>
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${release.train-version}</version>
              <type>pom</type>
              <scope>import</scope>
          </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 ActiveMQ, add the `spring-cloud-starter-sleuth`, `spring-cloud-sleuth-zipkin` and `activemq-client` dependencies.

Maven

```
<dependencyManagement>
      <dependencies>
          <dependency>
              <groupId>org.springframework.cloud</groupId>
              <artifactId>spring-cloud-dependencies</artifactId>
              <version>${release.train-version}</version>
              <type>pom</type>
              <scope>import</scope>
          </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.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"
    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 See Spans in an External System?

If you can't see spans get reported to an external system (e.g. Zipkin), then it's most likely due to the following causes:

- Your span is not being sampled
- You have forgotten to add the dependency to report to an external system (e.g. `spring-cloud-sleuth-zipkin`)
- You have misconfigured the connection to the external system

5.4.1. Your Span Is Not Being Sampled

In order to check if the span is not being sampled it's enough to see if the exportable flag is being set. Let's look at the following example:

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

If the boolean value in the section `[backend,0b6aaf642574edd3,0b6aaf642574edd3,true]` is `true` means that the span is being sampled and should be reported.

5.4.2. Missing Dependency

Up till Sleuth 3.0.0 the dependency `spring-cloud-starter-zipkin` included the `spring-cloud-starter-sleuth` dependency and the `spring-cloud-sleuth-zipkin` dependency. With 3.0.0 `spring-cloud-starter-zipkin` was removed, so you need to change it to `spring-cloud-sleuth-zipkin`.

5.4.3. Connection Misconfiguration

Double check if the remote system address is correct (e.g. `spring.zipkin.baseUrl`) and that if trying to communicate over the broker, your broker connection is set up properly.

5.5. How to Make RestTemplate, WebClient, etc. Work?

If you're observing that the tracing context is not being propagated then cause is one of the following:

- We are not instrumenting the given library
- We are instrumenting the library, however you misconfigured the setup

In case of lack of instrumentation capabilities please file an issue with a request to add such instrumentation.

In case of the misconfiguration please ensure that the client you're using to communicate is a Spring bean. If you create the client manually via the `new` operator the instrumentation will not work.

Example where instrumentation will work:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration(proxyBeanMethods = false)
class MyConfiguration {
    @Bean RestTemplate myRestTemplate() {
        return new RestTemplate();
    }
}

@Service
class MyService {
    private final RestTemplate restTemplate;

    MyService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    String makeACall() {
        return this.restTemplate.getForObject("http://example.com", String.class);
    }

}
```

Example where instrumentation will **NOT** work:

```
@Service
class MyService {

    String makeACall() {
        // This will not work because RestTemplate is not a bean
        return new RestTemplate().getForObject("http://example.com",
String.class);
    }

}
```

5.6. How to Add Headers to the HTTP Server Response?

Register a filter that will set the server response.

```
import org.springframework.cloud.sleuth.Span;
import org.springframework.cloud.sleuth.Tracer;

import javax.servlet.Filter;
import org.springframework.web.server.WebFilter;

@Configuration(proxyBeanMethods = false)
class MyConfig {

        // Example of a servlet Filter for non-reactive applications
        @Bean
        Filter traceIdInResponseFilter(Tracer tracer) {
            return (request, response, chain) -> {
                Span currentSpan = tracer.currentSpan();
                if (currentSpan != null) {
                    HttpServletResponse resp = (HttpServletResponse) response;
                    // putting trace id value in [mytraceid] response header
                    resp.addHeader("mytraceid", currentSpan.context().traceId());
                }
                chain.doFilter(request, response);
            };
        }

        // Example of a reactive WebFilter for reactive applications
        @Bean
        WebFilter traceIdInResponseFilter(Tracer tracer) {
            return (exchange, chain) -> {
                Span currentSpan = tracer.currentSpan();
                if (currentSpan != null) {
                    // putting trace id value in [mytraceid] response header
                    exchange.getResponse().getHeaders().add("mytraceid",
currentSpan.context().traceId());
                }
                return chain.filter(exchange);
            };
        }
}
```

Your spans need to be sampled for the parser to work. That means that you need to be able to export spans to e.g. Zipkin.

5.7. How to Customize HTTP Client Spans?

Register a bean of `HttpRequestParser` type whose name is `HttpClientRequestParser.NAME` to add customization for the request side. Register a bean of `HttpResponseParser` type whose name is `HttpClientRequestParser.NAME` to add customization for the response side.

```
@Configuration(proxyBeanMethods = false)
public static class ClientParserConfiguration {

    // example for Feign
    @Bean(name = HttpClientRequestParser.NAME)
    HttpRequestParser myHttpClientRequestParser() {
        return (request, context, span) -> {
            // Span customization
            span.name(request.method());
            span.tag("ClientRequest", "Tag");
            Object unwrap = request.unwrap();
            if (unwrap instanceof feign.Request) {
                feign.Request req = (feign.Request) unwrap;
                // Span customization
                span.tag("ClientRequestFeign", req.httpMethod().name());
            }
        };
    }

    // example for Feign
    @Bean(name = HttpClientResponseParser.NAME)
    HttpResponseParser myHttpClientResponseParser() {
        return (response, context, span) -> {
            // Span customization
            span.tag("ClientResponse", "Tag");
            Object unwrap = response.unwrap();
            if (unwrap instanceof feign.Response) {
                feign.Response resp = (feign.Response) unwrap;
                // Span customization
                span.tag("ClientResponseFeign", String.valueOf(resp.status()));
            }
        };
    }

}
```

5.8. How to Customize HTTP Server Spans?

Register a bean of `HttpRequestParser` type whose name is `HttpServerRequestParser.NAME` to add customization for the request side. Register a bean of `HttpResponseParser` type whose name is `HttpServerResponseParser.NAME` to add customization for the response side.

```
@Configuration(proxyBeanMethods = false)
public static class ServerParserConfiguration {

    @Bean(name = HttpServerRequestParser.NAME)
    HttpRequestParser myHttpRequestParser() {
        return (request, context, span) -> {
            // Span customization
            span.tag("ServerRequest", "Tag");
            Object unwrap = request.unwrap();
            if (unwrap instanceof HttpServletRequest) {
                HttpServletRequest req = (HttpServletRequest) unwrap;
                // Span customization
                span.tag("ServerRequestServlet", req.getMethod());
            }
        };
    }

    @Bean(name = HttpServerResponseParser.NAME)
    HttpResponseParser myHttpResponseParser() {
        return (response, context, span) -> {
            // Span customization
            span.tag("ServerResponse", "Tag");
            Object unwrap = response.unwrap();
            if (unwrap instanceof HttpServletResponse) {
                HttpServletResponse resp = (HttpServletResponse) unwrap;
                // Span customization
                span.tag("ServerResponseServlet",
String.valueOf(resp.getStatus()));
            }
        };
    }

    @Bean
    Filter traceIdInResponseFilter(Tracer tracer) {
        return (request, response, chain) -> {
            Span currentSpan = tracer.currentSpan();
            if (currentSpan != null) {
                HttpServletResponse resp = (HttpServletResponse) response;
                resp.addHeader("mytraceid", currentSpan.context().traceId());
            }
            chain.doFilter(request, response);
        };
    }

}
```

Your spans need to be sampled for the parser to work. That means that you need to be able to export spans to e.g. Zipkin.

5.9. How to See the Application Name in Logs?

Assuming that you haven't changed the default logging format set the `spring.application.name` property in `bootstrap.yml`, not in `application.yml`.

With the new Spring Cloud configuration bootstrap this should no longer be required since there will be no Bootstrap Context anymore.

5.10. 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 Brave we support `AWS`, `B3`, `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). Below you can find the examples:

```
@Component
class CustomPropagator extends Propagation.Factory implements Propagation<String>
{

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

    @Override
    public <R> TraceContext.Injector<R> injector(Setter<R, String> setter) {
        return (traceContext, request) -> {
            setter.put(request, "myCustomTraceId", traceContext.traceIdString());
            setter.put(request, "myCustomSpanId", traceContext.spanIdString());
        };
    }

    @Override
    public <R> TraceContext.Extractor<R> extractor(Getter<R, String> getter) {
        return request ->
TraceContextOrSamplingFlags.create(TraceContext.newBuilder()
                .traceId(HexCodec.lowerHexToUnsignedLong(getter.get(request,
"myCustomTraceId")))
                .spanId(HexCodec.lowerHexToUnsignedLong(getter.get(request,
"myCustomSpanId"))).build());
    }

    @Override
    public Propagation<String> get() {
        return this;
    }

}
```

5.11. How to Implement My Own Tracer?

Spring Cloud Sleuth API contains all necessary interfaces to be implemented by a tracer. The project comes with OpenZipkin Brave implementation. You can check how both tracers are bridged to the Sleuth's API by looking at the `org.springframework.cloud.sleuth.brave.bridge` module.

6. Spring Cloud Sleuth customization

In this section, we describe how to customize various parts of Spring Cloud Sleuth. Please check the appendix for the list of spans, tags and events.

6.1. Apache Kafka

This feature is available for all tracer implementations.

We decorate the Kafka clients (`KafkaProducer` and `KafkaConsumer`) to create a span for each event that is produced or consumed. You can disable this feature by setting the value of `spring.sleuth.kafka.enabled` to `false`.

You have to register the `Producer` or `Consumer` as beans in order for Sleuth's auto-configuration to decorate them. When you then inject the beans, the expected type must be `Producer` or `Consumer` (and NOT e.g. `KafkaProducer`).

For use with project reactor we decorate `KafkaReceiver<K,V>` with `TracingKafkaReceiver<K,V>` for every bean of that type declared. This will create separate publisher for each element received with its own tracing context propagated. When used with reactor instrumentation you will have access to the context of spans.

If there's no parent context, it will create just child span with new trace-id.

```
@Bean
KafkaReceiver<K, V> reactiveKafkaReceiver(ReceiverOptions<K,V> options) {
    return KafkaReceiver.create(options);
}
```

Later you can simply start receiving your elements to process the context.

```
@Bean
DisposableBean exampleRunningConsumer(KafkaReceiver<String, String> receiver){
    reactor.core.Disposable disposable = receiver.receive()
        //If you need to read context you can for example use deferContextual
        .flatMap(record -> Mono.deferContextual(context -> Mono.just(record)))
        .doOnNext(record -> log.info("I will be coorelated to the child span created
with parent context from kafka record"))
        .subscribe(record -> record.receiverOffset().acknowledge());

    return disposable::dispose;
}
```

6.2. Asynchronous Communication

In this section, we describe how to customize asynchronous communication with Spring Cloud Sleuth.

6.2.1. `@Async` Annotated methods

This feature is available for all tracer implementations.

In Spring Cloud Sleuth, we instrument async-related components so that the tracing information is passed between threads. You can disable this behavior by setting the value of `spring.sleuth.async.enabled` to `false`.

If you annotate your method with `@Async`, we automatically modify the existing Span as follows:

- If the method is annotated with `@SpanName`, the value of the annotation is the Span's name.
- If the method is not annotated with `@SpanName`, the Span name is the annotated method name.
- The span is tagged with the method's class name and method name.

Since we're modifying the existing span, if you want to maintain its original name (e.g. a span created by receiving an HTTP request) you should wrap your `@Async` annotated method with a `@NewSpan` annotation or create a new span manually.

6.2.2. `@Scheduled` Annotated Methods

This feature is available for all tracer implementations.

In Spring Cloud Sleuth, we instrument scheduled method execution so that the tracing information is passed between threads. You can disable this behavior by setting the value of `spring.sleuth.scheduled.enabled` to `false`.

If you annotate your method with `@Scheduled`, we automatically create a new span with the following characteristics:

- The span name is the annotated method name.
- The span is tagged with the method's class name and method name.

If you want to skip span creation for some `@Scheduled` annotated classes, you can set the `spring.sleuth.scheduled.skipPattern` with a regular expression that matches the fully qualified name of the `@Scheduled` annotated class.

6.2.3. Executor, ExecutorService, and ScheduledExecutorService

This feature is available for all tracer implementations.

We provide `LazyTraceExecutor`, `TraceableExecutorService`, and `TraceableScheduledExecutorService`. Those implementations create spans each time a new task is submitted, invoked, or scheduled.

The following example shows how to pass tracing information with `TraceableExecutorService` when working with `CompletableFuture`:

```
CompletableFuture<Long> completableFuture = CompletableFuture.supplyAsync(() -> {
    // perform some logic
    return 1_000_000L;
}, new TraceableExecutorService(beanFactory, executorService,
        // 'calculateTax' explicitly names the span - this param is optional
        "calculateTax"));
```

Sleuth does not work with `parallelStream()` out of the box. If you want to have the tracing information propagated through the stream, you have to use the approach with `supplyAsync(…)`, as shown earlier.

If there are beans that implement the `Executor` interface that you would like to exclude from span creation, you can use the `spring.sleuth.async.ignored-beans` property where you can provide a list of bean names.

You can disable this behavior by setting the value of `spring.sleuth.async.enabled` to `false`.

Customization of Executors

Sometimes, you need to set up a custom instance of the `AsyncExecutor`. The following example shows how to set up such a custom `Executor`:

```
@Configuration(proxyBeanMethods = false)
@EnableAutoConfiguration
@EnableAsync
// add the infrastructure role to ensure that the bean gets auto-proxied
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
public static class CustomExecutorConfig extends AsyncConfigurerSupport {

    @Autowired
    BeanFactory beanFactory;

    @Override
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        // CUSTOMIZE HERE
        executor.setCorePoolSize(7);
        executor.setMaxPoolSize(42);
        executor.setQueueCapacity(11);
        executor.setThreadNamePrefix("MyExecutor-");
        // DON'T FORGET TO INITIALIZE
        executor.initialize();
        return new LazyTraceExecutor(this.beanFactory, executor);
    }

}
```

To ensure that your configuration gets post processed, remember to add the `@Role(BeanDefinition.ROLE_INFRASTRUCTURE)` on your `@Configuration` class

6.3. HTTP Client Integration

Features from this section can be disabled by setting the `spring.sleuth.web.client.enabled` property with value equal to `false`.

6.3.1. Synchronous Rest Template

This feature is available for all tracer implementations.

We inject a `RestTemplate` interceptor to ensure that all the tracing information is passed to the requests. Each time a call is made, a new Span is created. It gets closed upon receiving the response. To block the synchronous `RestTemplate` features, set `spring.sleuth.web.client.enabled` to `false`.

You have to register `RestTemplate` as a bean so that the interceptors get injected. If you create a `RestTemplate` instance with a `new` keyword, the instrumentation does NOT work.

6.3.2. Asynchronous Rest Template

This feature is available for all tracer implementations.

Starting with Sleuth `2.0.0`, we no longer register a bean of `AsyncRestTemplate` type. It is up to you to create such a bean. Then we instrument it.

To block the `AsyncRestTemplate` features, set `spring.sleuth.web.async.client.enabled` to `false`. To disable creation of the default `TraceAsyncClientHttpRequestFactoryWrapper`, set `spring.sleuth.web.async.client.factory.enabled` to `false`. If you do not want to create `AsyncRestClient` at all, set `spring.sleuth.web.async.client.template.enabled` to `false`.

Multiple Asynchronous Rest Templates

Sometimes you need to use multiple implementations of the Asynchronous Rest Template. In the following snippet, you can see an example of how to set up such a custom `AsyncRestTemplate`:

```
@Configuration(proxyBeanMethods = false)
public static class TestConfig {

    @Bean(name = "customAsyncRestTemplate")
    public AsyncRestTemplate traceAsyncRestTemplate() {
        return new AsyncRestTemplate(asyncClientFactory(),
clientHttpRequestFactory());
    }

    private ClientHttpRequestFactory clientHttpRequestFactory() {
        ClientHttpRequestFactory clientHttpRequestFactory = new
CustomClientHttpRequestFactory();
        // CUSTOMIZE HERE
        return clientHttpRequestFactory;
    }

    private AsyncClientHttpRequestFactory asyncClientFactory() {
        AsyncClientHttpRequestFactory factory = new
CustomAsyncClientHttpRequestFactory();
        // CUSTOMIZE HERE
        return factory;
    }

}
```

Troubleshooting Async Configuration Issues

Sine Sleuth uses a Bean Post Processor (BPP) to modify executors the `Executor` type that you provided will be modified and the return type will be the trace version of that executor. That can lead to exceptions simillar to this

```
12:12:49.606 [main] DEBUG
org.springframework.context.annotation.AnnotationConfigApplicationContext - Closing
org.springframework.context.annotation.AnnotationConfigApplicationContext@2401f4c3,
started on Wed Jan 19 12:12:49 GMT 2022
Exception in thread "main"
org.springframework.beans.factory.BeanNotOfRequiredTypeException: Bean named
'exampleConfigurer' is expected to be of type
'com.example.demo.Gh29151Application$Example' but was actually of type
'org.springframework.cloud.sleuth.instrument.async.LazyTraceAsyncCustomizer'
```

If you see such exceptions you must

- disable sleuth async
- manually instrument the executors using their trace representations

Example

```
spring.sleuth.async.enabled=false
```

and configuration example

```
@Configuration
@EnableAsync
public class AsyncConfiguration implements AsyncConfigurer {

    private final BeanFactory beanFactory;

    public AsyncConfiguration(BeanFactory beanFactory) {
        this.beanFactory = beanFactory;
    }

    @Override
    @Bean("AsyncTaskExecutor")
    public Executor getAsyncExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.initialize();
        return new LazyTraceThreadPoolTaskExecutor(beanFactory, executor);
    }
}
```

You can read more about this in this issue.

`WebClient`

This feature is available for all tracer implementations.

We inject a `ExchangeFilterFunction` implementation that creates a span and, through on-success and on-error callbacks, takes care of closing client-side spans.

To block this feature, set `spring.sleuth.web.client.enabled` to `false`.

You have to register `WebClient` as a bean so that the tracing instrumentation gets applied. If you create a `WebClient` instance with a `new` keyword, the instrumentation does NOT work.

Logbook with WebClient

In order to add support for Logbook with WebClient `org.zalando:logbook-spring-boot-webflux-autoconfigure` you need to add the following configuration. You can read more about this integration in this issue.

```
@Configuration
@Import(LogbookWebFluxAutoConfiguration.class)
public class LogbookConfiguration {

    @Bean
    public LogstashLogbackSink logbackSink(final HttpLogFormatter formatter) {
        return new LogstashLogbackSink(formatter);
    }

    @Bean
    public CorrelationId correlationId(final Tracer tracer) {
        return request ->
requireNonNull(requireNonNull(tracer.currentSpan())).context().traceId();
    }

    @Bean
    ReactorNettyHttpTracing reactorNettyHttpTracing(final HttpTracing httpTracing) {
        return ReactorNettyHttpTracing.create(httpTracing);
    }

    @Bean
    NettyServerCustomizer nettyServerCustomizer(final Logbook logbook,
            final ReactorNettyHttpTracing reactorNettyHttpTracing) {
        return server -> reactorNettyHttpTracing.decorateHttpServer(
                server.doOnConnection(conn -> conn.addHandlerFirst(new
LogbookServerHandler(logbook))));
    }

    @Bean
    WebClient webClient(final Logbook logbook, final ReactorNettyHttpTracing
reactorNettyHttpTracing) {
        return WebClient.builder()
                .clientConnector(new
ReactorClientHttpConnector(reactorNettyHttpTracing.decorateHttpClient(HttpClient
                        .create().doOnConnected(conn -> conn.addHandlerLast(new
LogbookClientHandler(logbook))))))
                .build();
    }

}
```

Traverson

This feature is available for all tracer implementations.

If you use the Traverson library, you can inject a `RestTemplate` as a bean into your Traverson object. Since `RestTemplate` is already intercepted, you get full support for tracing in your client. The following pseudo code shows how to do that:

```
@Autowired RestTemplate restTemplate;

Traverson traverson = new Traverson(URI.create("https://some/address"),
    MediaType.APPLICATION_JSON,
MediaType.APPLICATION_JSON_UTF8).setRestOperations(restTemplate);
// use Traverson
```

Apache `HttpClientBuilder` and `HttpAsyncClientBuilder`

This feature is available for Brave tracer implementation.

We instrument the `HttpClientBuilder` and `HttpAsyncClientBuilder` so that tracing context gets injected to the sent requests.

To block these features, set `spring.sleuth.web.client.enabled` to `false`.

Netty `HttpClient`

This feature is available for all tracer implementations.

We instrument the Netty's `HttpClient`.

To block this feature, set `spring.sleuth.web.client.enabled` to `false`.

You have to register `HttpClient` as a bean so that the instrumentation happens. If you create a `HttpClient` instance with a `new` keyword, the instrumentation does NOT work.

`UserInfoRestTemplateCustomizer`

This feature is available for all tracer implementations.

We instrument the Spring Security's `UserInfoRestTemplateCustomizer`.

To block this feature, set `spring.sleuth.web.client.enabled` to `false`.

6.4. HTTP Server Integration

Features from this section can be disabled by setting the `spring.sleuth.web.enabled` property with value equal to `false`.

6.4.1. HTTP Filter

This feature is available for all tracer implementations.

Through the `TracingFilter`, all sampled incoming requests result in creation of a Span. You can configure which URIs you would like to skip by setting the `spring.sleuth.web.skipPattern` property. If you have `ManagementServerProperties` on classpath, its value of `contextPath` gets appended to the provided skip pattern. If you want to reuse the Sleuth's default skip patterns and just append your

own, pass those patterns by using the `spring.sleuth.web.additionalSkipPattern`.

By default, all the spring boot actuator endpoints are automatically added to the skip pattern. If you want to disable this behaviour set `spring.sleuth.web.ignore-auto-configured-skip-patterns` to `true`.

To change the order of tracing filter registration, please set the `spring.sleuth.web.filter-order` property.

To disable the filter that logs uncaught exceptions you can disable the `spring.sleuth.web.exception-throwing-filter-enabled` property.

6.4.2. Async Servlet support

This feature is available for all tracer implementations.

If your controller returns a `Callable` or a `WebAsyncTask`, Spring Cloud Sleuth continues the existing span instead of creating a new one.

6.4.3. WebFlux support

This feature is available for all tracer implementations.

Through `TraceWebFilter`, all sampled incoming requests result in creation of a Span. That Span's name is `http:` + the path to which the request was sent. For example, if the request was sent to `/this/that`, the name is `http:/this/that`. You can configure which URIs you would like to skip by using the `spring.sleuth.web.skipPattern` property. If you have `ManagementServerProperties` on the classpath, its value of `contextPath` gets appended to the provided skip pattern. If you want to reuse Sleuth's default skip patterns and append your own, pass those patterns by using the `spring.sleuth.web.additionalSkipPattern`.

In order to achieve best results in terms of performance and context propagation we suggest that you switch the `spring.sleuth.reactor.instrumentation-type` to `MANUAL`. In order to execute code with the span in scope you can call `WebFluxSleuthOperators.withSpanInScope`. Example:

```
@GetMapping("/simpleManual")
public Mono<String> simpleManual() {
    return Mono.just("hello").map(String::toUpperCase).doOnEach(WebFluxSleuthOperators
            .withSpanInScope(SignalType.ON_NEXT, signal -> log.info("Hello from simple
[{}]", signal.get())));
}
```

To change the order of tracing filter registration, please set the `spring.sleuth.web.filter-order` property.

6.4.4. Reactor Netty HttpServer

If you're using Reactor Netty and would like to have your access logs instrumented you need to add the `io.projectreactor.netty:reactor-netty-http-brave` (this will work only for the Brave Tracer). Also add the following configuration to your project.

```
import brave.http.HttpTracing;
import reactor.netty.http.brave.ReactorNettyHttpTracing;

@Configuration(proxyBeanMethods = false)
class TraceNettyConfig {

    @Bean
    NettyServerCustomizer traceNettyServerCustomizer(ObjectProvider<HttpTracing>
tracing) {
        return server ->
ReactorNettyHttpTracing.create(tracing.getObject()).decorateHttpServer(server);
    }
}
```

6.5. Messaging

Features from this section can be disabled by setting the `spring.sleuth.messaging.enabled` property with value equal to `false`.

6.5.1. Spring Integration

This feature is available for all tracer implementations.

Spring Cloud Sleuth integrates with Spring Integration. It creates spans for publish and subscribe events. To disable Spring Integration instrumentation, set `spring.sleuth.integration.enabled` to `false`.

You can provide the `spring.sleuth.integration.patterns` pattern to explicitly provide the names of channels that you want to include for tracing. By default, all channels but `hystrixStreamOutput` channel are included.

When using the `Executor` to build a Spring Integration `IntegrationFlow`, you must use the untraced version of the `Executor`. Decorating the Spring Integration Executor Channel with `TraceableExecutorService` causes the spans to be improperly closed.

If you want to customize the way tracing context is read from and written to message headers, it's enough for you to register beans of types:

- `Propagator.Setter<MessageHeaderAccessor>` - for writing headers to the message
- `Propagator.Getter<MessageHeaderAccessor>` - for reading headers from the message

Spring Integration Customization

Customizing messaging spans

In order to change the default span names and tags, just register a bean of type `MessageSpanCustomizer`. You can also override the existing `DefaultMessageSpanCustomizer` to extend

the existing behaviour.

```
@Component
  class MyMessageSpanCustomizer extends DefaultMessageSpanCustomizer {
      @Override
      public Span customizeHandle(Span spanCustomizer,
              Message<?> message, MessageChannel messageChannel) {
          return super.customizeHandle(spanCustomizer, message, messageChannel)
                  .name("changedHandle")
                  .tag("handleKey", "handleValue")
                  .tag("channelName", channelName(messageChannel));
      }

      @Override
      public Span.Builder customizeSend(Span.Builder builder,
              Message<?> message, MessageChannel messageChannel) {
          return super.customizeSend(builder, message, messageChannel)
                  .name("changedSend")
                  .tag("sendKey", "sendValue")
                  .tag("channelName", channelName(messageChannel));
      }
  }
```

6.5.2. Spring Cloud Function and Spring Cloud Stream

This feature is available for all tracer implementations.

Spring Cloud Sleuth can instrument Spring Cloud Function. Since Spring Cloud Stream uses Spring Cloud Function you will get the messaging instrumentation out of the box.

The way to achieve it is to provide a `Function` or `Consumer` or `Supplier` that takes in a `Message` as a parameter e.g. `Function<Message<String>, Message<Integer>>`. If the type **is not** `Message` then instrumentation **will not** take place.

For a reactive `Consumer<Flux<Message<?>>>` remember to manually close the span and clear the context before you call `.subscribe()`. Example:

```
@Bean
    Consumer<Flux<Message<String>>> channel(Tracer tracer) {
        // For the reactive consumer remember to call "subscribe()" at the end,
otherwise
        // you'll get the "Dispatcher has no subscribers" error
        return i -> i
                    .doOnNext(s -> log.info("HELLO"))
                    // You must finish the span yourself and clear the tracing context
like presented below.
                    // Otherwise you will be missing out the span that wraps the
function execution.
                    .doOnNext(s -> {
                        tracer.currentSpan().end();
                        tracer.withSpan(null);
                    })
                    .subscribe();
    }
}
```

NOTE: For Sleuth to work with any `Supplier` (e.g. `Supplier<Flux<Message<String>>>`) you must fall back to Spring Integration based instrumentation by setting `spring.sleuth.integration.enabled` to `true`.

You can disable Spring Cloud Stream integration by setting the value of `spring.sleuth.function.enabled` to `false`.

If you want to fully control the life cycle of spans within the reactive messaging context of Spring Cloud Stream remember to disable the Spring Cloud Stream integration and leverage the `MessagingSleuthOperators` utility class that allows you to manipulate the input and output messages in order to continue the tracing context and to execute custom code within the tracing context.

```
class SimpleReactiveManualFunction implements Function<Flux<Message<String>>,
Flux<Message<String>>> {

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

    private final BeanFactory beanFactory;

    SimpleReactiveManualFunction(BeanFactory beanFactory) {
        this.beanFactory = beanFactory;
    }

    @Override
    public Flux<Message<String>> apply(Flux<Message<String>> input) {
        return input.map(message ->
(MessagingSleuthOperators.asFunction(this.beanFactory, message))
                .andThen(msg ->
MessagingSleuthOperators.withSpanInScope(this.beanFactory, msg, stringMessage -> {
                    log.info("Hello from simple manual [{}]",
stringMessage.getPayload());
                    return stringMessage;
                })).andThen(msg ->
MessagingSleuthOperators.afterMessageHandled(this.beanFactory, msg, null))
                .andThen(msg ->
MessageBuilder.createMessage(msg.getPayload().toUpperCase(), msg.getHeaders()))
                .andThen(msg ->
MessagingSleuthOperators.handleOutputMessage(this.beanFactory, msg)).apply(message));
    }

}
```

6.5.3. Spring RabbitMq

This feature is available for Brave tracer implementation.

We instrument the `RabbitTemplate` so that tracing headers get injected into the message.

To block this feature, set `spring.sleuth.messaging.rabbit.enabled` to `false`.

6.5.4. Spring Kafka

This feature is available for Brave tracer implementation.

We instrument the Spring Kafka's `ProducerFactory` and `ConsumerFactory` so that tracing headers get injected into the created Spring Kafka's `Producer` and `Consumer`.

To block this feature, set `spring.sleuth.messaging.kafka.enabled` to `false`.

6.5.5. Spring Kafka Streams

This feature is available for Brave tracer implementation.

We instrument the `KafkaStreams KafkaClientSupplier` so that tracing headers get injected into the `Producer` and `Consumer`s. A `KafkaStreamsTracing` bean allows for further instrumentation through additional `TransformerSupplier` and `ProcessorSupplier` methods.

To block this feature, set `spring.sleuth.messaging.kafka.streams.enabled` to `false`.

6.5.6. Spring JMS

This feature is available for Brave tracer implementation.

We instrument the `JmsTemplate` so that tracing headers get injected into the message. We also support `@JmsListener` annotated methods on the consumer side.

To block this feature, set `spring.sleuth.messaging.jms.enabled` to `false`.

We don't support baggage propagation for JMS

6.6. OpenFeign

This feature is available for all tracer implementations.

By default, Spring Cloud Sleuth provides integration with Feign through `TraceFeignClientAutoConfiguration`. You can disable it entirely by setting `spring.sleuth.feign.enabled` to `false`. If you do so, no Feign-related instrumentation take place.

Part of Feign instrumentation is done through a `FeignBeanPostProcessor`. You can disable it by setting `spring.sleuth.feign.processor.enabled` to `false`. If you set it to `false`, Spring Cloud Sleuth does not instrument any of your custom Feign components. However, all the default instrumentation is still there.

6.7. OpenTracing

This feature is available for all tracer implementations.

Spring Cloud Sleuth is compatible with OpenTracing. If you have OpenTracing on the classpath, we automatically register the OpenTracing `Tracer` bean. If you wish to disable this, set `spring.sleuth.opentracing.enabled` to `false`

6.8. Quartz

This feature is available for all tracer implementations.

We instrument quartz jobs by adding Job/Trigger listeners to the Quartz Scheduler.

To turn off this feature, set the `spring.sleuth.quartz.enabled` property to `false`.

6.9. Reactor

This feature is available for all tracer implementations.

We have the following modes of instrumenting reactor based applications that can be set via `spring.sleuth.reactor.instrumentation-type` property:

- `DECORATE_QUEUES` - With the new Reactor queue wrapping mechanism (Reactor 3.4.3) we're instrumenting the way threads are switched by Reactor. This should lead to feature parity with `ON_EACH` with low performance impact.
- `DECORATE_ON_EACH` - wraps every Reactor operator in a trace representation. Passes the tracing context in most cases. This mode might lead to drastic performance degradation.
- `DECORATE_ON_LAST` - wraps last Reactor operator in a trace representation. Passes the tracing context in some cases thus accessing MDC context might not work. This mode might lead to medium performance degradation.
- `MANUAL` - wraps every Reactor in the least invasive way without passing of tracing context. It's up to the user to do it.

Current default is `ON_EACH` for backward compatibility reasons, however we encourage the users to migrate to the `MANUAL` instrumentation and profit from `WebFluxSleuthOperators` and `MessagingSleuthOperators`. The performance improvement can be substantial. Example:

```
@GetMapping("/simpleManual")
public Mono<String> simpleManual() {
    return Mono.just("hello").map(String::toUpperCase).doOnEach(WebFluxSleuthOperators
            .withSpanInScope(SignalType.ON_NEXT, signal -> log.info("Hello from simple
[{}]", signal.get())));
}
```

To disable Reactor support, set the `spring.sleuth.reactor.enabled` property to `false`.

6.10. Redis

This feature is available for all tracer implementations.

We're using the `Tracing` abstraction from Lettuce. If Brave is on the classpath we configure `Tracing` to be `BraveTracing`.

To disable Redis support, set the `spring.sleuth.redis.enabled` property to `false`.

6.10.1. Redis With Legacy Brave Only Support

To use the Brave only supported feature you need to set the value of `spring.sleuth.redis.legacy.enabled` to `true`. This is the default mechanism available up till version 3.1.0 of Spring Cloud Sleuth.

We set `tracing` property to Lettuce `ClientResources` instance to enable Brave tracing built in Lettuce.

Spring Cloud Sleuth will provide a traced version of the `ClientResources` bean. If you have your own implementation of that bean, remember to customize the `ClientResources.Builder` with a stream of `ClientResourcesBuilderCustomizer`s like presented below:

```
@Bean(destroyMethod = "shutdown")
DefaultClientResources
myLettuceClientResources(ObjectProvider<ClientResourcesBuilderCustomizer> customizer)
{
    DefaultClientResources.Builder builder = DefaultClientResources.builder();
    // setting up the builder manually
    customizer.stream().forEach(c -> c.customize(builder));
    return builder.build();
}
```

6.11. Runnable and Callable

This feature is available for all tracer implementations.

If you wrap your logic in `Runnable` or `Callable`, you can wrap those classes in their Sleuth representative, as shown in the following example for `Runnable`:

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        // do some work
    }

    @Override
    public String toString() {
        return "spanNameFromToStringMethod";
    }
};
// Manual `TraceRunnable` creation with explicit "calculateTax" Span name
Runnable traceRunnable = new TraceRunnable(this.tracer, spanNamer, runnable,
"calculateTax");
```

The following example shows how to do so for `Callable`:

```
Callable<String> callable = new Callable<String>() {
    @Override
    public String call() throws Exception {
        return someLogic();
    }

    @Override
    public String toString() {
        return "spanNameFromToStringMethod";
    }
};
// Manual `TraceCallable` creation with explicit "calculateTax" Span name
Callable<String> traceCallable = new TraceCallable<>(tracer, spanNamer, callable,
"calculateTax");
```

That way, you ensure that a new span is created and closed for each execution.

6.12. RPC

This feature is available for Brave tracer implementation.

Sleuth automatically configures the `RpcTracing` bean which serves as a foundation for RPC instrumentation such as gRPC or Dubbo.

If a customization of client / server sampling of the RPC traces is required, just register a bean of type `brave.sampler.SamplerFunction<RpcRequest>` and name the bean `sleuthRpcClientSampler` for client sampler and `sleuthRpcServerSampler` for server sampler.

For your convenience the `@RpcClientSampler` and `@RpcServerSampler` annotations can be used to inject the proper beans or to reference the bean names via their static String `NAME` fields.

Ex. Here's a sampler that traces 100 "GetUserToken" server requests per second. This doesn't start new traces for requests to the health check service. Other requests will use the global sampling configuration.

```
@Configuration(proxyBeanMethods = false)
    class Config {
  @Bean(name = RpcServerSampler.NAME)
  SamplerFunction<RpcRequest> myRpcSampler() {
      Matcher<RpcRequest> userAuth = and(serviceEquals("users.UserService"),
methodEquals("GetUserToken"));
      return
RpcRuleSampler.newBuilder().putRule(serviceEquals("grpc.health.v1.Health"),
Sampler.NEVER_SAMPLE)
              .putRule(userAuth, RateLimitingSampler.create(100)).build();
  }
}
```

For more, see github.com/openzipkin/brave/tree/master/instrumentation/rpc#sampling-policy

6.12.1. Dubbo RPC support

Via the integration with Brave, Spring Cloud Sleuth supports Dubbo. It's enough to add the `brave-instrumentation-dubbo` dependency:

```
<dependency>
    <groupId>io.zipkin.brave</groupId>
    <artifactId>brave-instrumentation-dubbo</artifactId>
</dependency>
```

You need to also set a `dubbo.properties` file with the following contents:

```
dubbo.provider.filter=tracing
dubbo.consumer.filter=tracing
```

You can read more about Brave - Dubbo integration here. An example of Spring Cloud Sleuth and Dubbo can be found here.

6.12.2. gRPC

Spring Cloud Sleuth provides instrumentation for gRPC via the Brave tracer. You can disable it entirely by setting `spring.sleuth.grpc.enabled` to `false`.

Variant 1

Dependencies

The gRPC integration relies on two external libraries to instrument clients and servers and both of those libraries must be on the class path to enable the instrumentation.

Maven:

```
        <dependency>
            <groupId>io.github.lognet</groupId>
            <artifactId>grpc-spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>io.zipkin.brave</groupId>
            <artifactId>brave-instrumentation-grpc</artifactId>
        </dependency>
```

Gradle:

```
    compile("io.github.lognet:grpc-spring-boot-starter")
    compile("io.zipkin.brave:brave-instrumentation-grpc")
```

Server Instrumentation

Spring Cloud Sleuth leverages grpc-spring-boot-starter to register Brave's gRPC server interceptor with all services annotated with `@GRpcService`.

Client Instrumentation

gRPC clients leverage a `ManagedChannelBuilder` to construct a `ManagedChannel` used to communicate to the gRPC server. The native `ManagedChannelBuilder` provides static methods as entry points for construction of `ManagedChannel` instances, however, this mechanism is outside the influence of the Spring application context.

Spring Cloud Sleuth provides a `SpringAwareManagedChannelBuilder` that can be customized through the Spring application context and injected by gRPC clients. **This builder must be used when creating `ManagedChannel` instances.**

Sleuth creates a `TracingManagedChannelBuilderCustomizer` which inject Brave's client interceptor into the `SpringAwareManagedChannelBuilder`.

Variant 2

Grpc Spring Boot Starter automatically detects the presence of Spring Cloud Sleuth and Brave's instrumentation for gRPC and registers the necessary client and/or server tooling.

6.13. RxJava

This feature is available for all tracer implementations.

We registering a custom `RxJavaSchedulersHook` that wraps all `Action0` instances in their Sleuth representative, which is called `TraceAction`. The hook either starts or continues a span, depending on whether tracing was already going on before the Action was scheduled. To disable the custom `RxJavaSchedulersHook`, set the `spring.sleuth.rxjava.schedulers.hook.enabled` to `false`.

You can define a list of regular expressions for thread names for which you do not want spans to be created. To do so, provide a comma-separated list of regular expressions in the `spring.sleuth.rxjava.schedulers.ignoredthreads` property.

The suggested approach to reactive programming and Sleuth is to use the Reactor support.

6.14. Spring Cloud CircuitBreaker

This feature is available for all tracer implementations.

If you have Spring Cloud CircuitBreaker on the classpath, we will wrap the passed command `Supplier` and the fallback `Function` in its trace representations. We will also instrument the reactive implementation of the CircuitBreaker. In order to disable this instrumentation set `spring.sleuth.circuitbreaker.enabled` to `false`.

6.15. Spring Cloud Config Server

This feature is available for all tracer implementations.

If you have Spring Cloud Config Server running on the classpath, we will wrap the `EnvironmentRepository` in a span. In order to disable this instrumentation set `spring.sleuth.config.server.enabled` to `false`.

6.16. Spring Cloud Deployer

This feature is available for all tracer implementations.

If you have Spring Cloud Deployer running on the classpath, we wrap the `AppDeployer` in a trace representation. We are polling the application for its status at a default interval. You can change that default by setting the `spring.sleuth.deployer.status-poll-delay` property. In order to disable this instrumentation set `spring.sleuth.deployer.enabled` to `false`.

6.17. Spring RSocket

This feature is available for all tracer implementations.

If you have Spring RSocket running on the classpath, we wrap the inbound and outbound communication to propagate the tracing context via the metadata. In order to disable this instrumentation set `spring.sleuth.rsocket.enabled` to `false`.

6.18. Spring Batch

This feature is available for all tracer implementations.

If you have Spring Batch running on the classpath, we wrap the `StepBuilderFactory` and the `JobBuilderFactory` to propagate the tracing context. In order to disable this instrumentation set `spring.sleuth.batch.enabled` to `false`.

6.19. Spring Cloud Task

This feature is available for all tracer implementations.

If you have Spring Cloud Task running on the classpath, we're instrumenting `TaskExecutionListener` and `CommandLineRunner` and `ApplicationRunner`. In order to disable this instrumentation set `spring.sleuth.task.enabled` to `false`.

6.20. Spring Tx

This feature is available for all tracer implementations.

If you have Spring Tx on the classpath we will instrument the `PlatformTransactionManager` and the `ReactiveTransactionManager` to create a span whenever a new transaction is created. Due to technical constraints we will not instrument classes that extend Spring's `AbstractPlatformTransactionManager`. In order to disable this instrumentation set `spring.sleuth.tx.enabled` to `false`.

6.21. Spring Security

This feature is available for all tracer implementations.

If you have Spring Security on the classpath, we create an implementation of `SecurityContextChangedListener` that annotates a current span with an event when context has changed. In order to disable this instrumentation set `spring.sleuth.security.enabled` to `false`.

6.22. R2DBC

This feature is available for all tracer implementations.

If you have R2DBC Proxy on the classpath we will instrument the `ConnectionFactory`so that it contains a custom `ProxyExecutionListener`. In order to disable this instrumentation set `spring.sleuth.r2dbc.enabled` to `false`.

6.23. Spring Vault

This feature is available for all tracer implementations.

We're instrumenting the `RestTemplate` or `WebClient` instances used by Spring Vault to communicate with Vault. In order to disable this instrumentation set `spring.sleuth.vault.enabled` to `false`.

6.24. Spring Tomcat

This feature is available for all tracer implementations.

We're adding an instrumented Tomcat's `Valve` that originates the span. In order to disable this instrumentation set `spring.sleuth.web.tomcat.enabled` to `false`.

6.25. Spring Data Cassandra

This feature is available for all tracer implementations.

We're instrumenting Casandra's `CqlSession` and `ReactiveSession` interfaces and we're providing our own implementation of the `RequestTracker`. In order to disable this instrumentation set `spring.sleuth.cassandra.enabled` to `false`.

6.26. Spring JDBC

This feature is available for all tracer implementations. It has been ported from the spring-boot-datasource-decorator project.

We're decorating `DataSource`s in a trace representation. We delegate actual proxying to either p6spy or datasource-proxy. In order to use this feature you need to have them on the classpath.

P6Spy Maven

```
<dependency>
    <groupId>p6spy</groupId>
    <artifactId>p6spy</artifactId>
    <version>${p6spy.version}</version>
    <scope>runtime</scope>
</dependency>
```

P6Spy Gradle

```
runtimeOnly "p6spy:p6spy:${p6spyVersion}"
```

Datasource Proxy Maven

```
<dependency>
    <groupId>net.ttddyy</groupId>
    <artifactId>datasource-proxy</artifactId>
    <version>${datasource-proxy.version}</version>
    <scope>runtime</scope>
</dependency>
```

Datasource Proxy Gradle

```
runtimeOnly "net.ttddyy:datasource-proxy:${datasourceProxyVersion}"
```

Please check the appendix page under `spring.sleuth.jdbc.p6spy` for all p6spy configuration options and `spring.sleuth.jdbc.datasource-proxy` for all datasource proxy configuration options.

For P6Spy by default logging parameter values will be disabled, set `spring.sleuth.jdbc.p6spy.tracing.include-parameter-values` to `true` to enable it.

You can configure P6Spy manually using one of available configuration methods. For more information please refer to the P6Spy Configuration Guide.

For Datasource Proxy by default logging queries will be disabled, set `spring.sleuth.jdbc.datasource-proxy.slow-query.enable-logging` to `true` to enable logging slow queries and set `spring.sleuth.jdbc.datasource-proxy.query.enable-logging` to `true` to enable logging

all queries.

In order to disable this instrumentation set `spring.sleuth.jdbc.enabled` to `false`.

6.27. MongoDB

This feature is available for all tracer implementations.

We're adding command listeners that wrap all commands in a span. If you want to have additional socket address related tags on the span set the `spring.sleuth.mongodb.socket-address-span-customizer.enabled` to `true`.

In order to disable this instrumentation set `spring.sleuth.mongodb.enabled` to `false`.

6.28. Spring Session

This feature is available for all tracer implementations.

We're instrumenting the `Session` repositories that wraps all operations in a span. In order to disable this instrumentation set `spring.sleuth.session.enabled` to `false`.

6.29. Kotlin Coroutines

This feature is available for all tracer implementations.

We're adding Kotlin Coroutines that allow you to retrieve the current span via the `Tracer` bean. You can either pass the bean to the Kotlin Coroutine context via `Tracer.asContextElement()` method execution or if you have Reactor Kotlin Coroutine integration on the classpath, we will retrieve it from Reactor's context. To retrieve the current span you can call the `currentSpan()` method within the Kotlin Coroutine.

6.30. Prometheus Exemplars

This feature is available for all tracer implementations.

Prometheus Exemplars are supported through `SpanContextSupplier`. If you use Micrometer, this will be auto-configured for you, but you can register `SpanContextSupplier` directly to Prometheus if you want.
Please check the Prometheus Docs, since this feature needs to be explicitly enabled on Prometheus' side, and it is only supported using the OpenMetrics format.

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 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.sleuth.async.configurer.enabled	true	Enable default AsyncConfigurer.
spring.sleuth.async.enabled	true	Enable instrumenting async related components so that the tracing information is passed between threads.
spring.sleuth.async.ignored-beans		List of {@link java.util.concurrent.Executor} bean names that should be ignored and not wrapped in a trace representation.
spring.sleuth.baggage.correlation-enabled	true	Enables correlating the baggage context with logging contexts.
spring.sleuth.baggage.correlation-fields		List of fields that should be propagated over the wire.
spring.sleuth.baggage.local-fields		List of fields that should be accessible within the JVM process but not propagated over the wire.
spring.sleuth.baggage.remote-fields		List of fields that are referenced the same in-process as it is on the wire. For example, the field "x-vcap-request-id" would be set as-is including the prefix.
spring.sleuth.baggage.tag-fields		List of fields that should automatically become tags.
spring.sleuth.batch.enabled	true	Enable Spring Batch instrumentation.
spring.sleuth.cassandra.enabled	true	Enable Cassandra instrumentation.
spring.sleuth.circuitbreaker.enabled	true	Enable Spring Cloud CircuitBreaker instrumentation.
spring.sleuth.config.server.enabled	true	Enable Spring Cloud Config Server instrumentation.
spring.sleuth.deployer.enabled	true	Enable Spring Cloud Deployer instrumentation.

Name	Default	Description
spring.sleuth.deployer.status-poll-delay	`500`	Default poll delay to retrieve the deployed application status.
spring.sleuth.enabled	`true`	
spring.sleuth.feign.enabled	`true`	Enable span information propagation when using Feign.
spring.sleuth.feign.processor.enabled	`true`	Enable post processor that wraps Feign Context in its tracing representations.
spring.sleuth.function.enabled	`true`	Enable instrumenting of Spring Cloud Function and Spring Cloud Function based projects (e.g. Spring Cloud Stream).
spring.sleuth.grpc.enabled	`true`	Enable span information propagation when using GRPC.
spring.sleuth.http.enabled	`true`	Enables HTTP support.
spring.sleuth.integration.enabled	`true`	Enable Spring Integration instrumentation.
spring.sleuth.integration.patterns	`[!hystrixStreamOutput*, *, !channel*]`	An array of patterns against which channel names will be matched. @see org.springframework.integration.config.GlobalChannelInterceptor#patterns() Defaults to any channel name not matching the Hystrix Stream and functional Stream channel names.
spring.sleuth.integration.websockets.enabled	`true`	Enable tracing for WebSockets.
spring.sleuth.jdbc.datasource-proxy.enabled	`true`	Should the datasource-proxy tracing be enabled?
spring.sleuth.jdbc.datasource-proxy.json-format	`false`	Use json output for logging query. @see ProxyDataSourceBuilder#asJson()
spring.sleuth.jdbc.datasource-proxy.logging		Logging to use for logging queries.
spring.sleuth.jdbc.datasource-proxy.multiline	`true`	Use multiline output for logging query. @see ProxyDataSourceBuilder#multiline()

Name	Default	Description
spring.sleuth.jdbc.datasource-proxy.query.enable-logging	`false`	Enable logging all queries to the log.
spring.sleuth.jdbc.datasource-proxy.query.log-level	`DEBUG`	Severity of query logger.
spring.sleuth.jdbc.datasource-proxy.query.logger-name		Name of query logger.
spring.sleuth.jdbc.datasource-proxy.slow-query.enable-logging	`false`	Enable logging slow queries to the log.
spring.sleuth.jdbc.datasource-proxy.slow-query.log-level	`WARN`	Severity of slow query logger.
spring.sleuth.jdbc.datasource-proxy.slow-query.logger-name		Name of slow query logger.
spring.sleuth.jdbc.datasource-proxy.slow-query.threshold	`300`	Number of seconds to consider query as slow.
spring.sleuth.jdbc.enabled	`true`	Enables JDBC instrumentation.
spring.sleuth.jdbc.excluded-data-source-bean-names		List of DataSource bean names that will not be decorated.
spring.sleuth.jdbc.includes		Which types of tracing we would like to include.
spring.sleuth.jdbc.p6spy.custom-appender-class		Class file to use (only with logging=custom). The class must implement {@link com.p6spy.engine.spy.appender.FormattedLogger}.
spring.sleuth.jdbc.p6spy.enable-logging	`false`	Enables logging JDBC events.
spring.sleuth.jdbc.p6spy.enabled	`true`	Should the p6spy tracing be enabled?
spring.sleuth.jdbc.p6spy.log-file	`spy.log`	Name of log file to use (only with logging=file).
spring.sleuth.jdbc.p6spy.log-filter.pattern		Use regex pattern to filter log messages. Only matched messages will be logged.
spring.sleuth.jdbc.p6spy.log-format		Custom log format.
spring.sleuth.jdbc.p6spy.logging		Logging to use for logging queries.

Name	Default	Description
spring.sleuth.jdbc.p6spy.multiline	true	Enables multiline output.
spring.sleuth.jdbc.p6spy.tracing.include-parameter-values	false	Report the effective sql string (with '?' replaced with real values) to tracing systems. <p> NOTE this setting does not affect the logging message.
spring.sleuth.kafka.enabled	true	Enable instrumenting of Apache Kafka clients.
spring.sleuth.messaging.aspect.enabled	false	Should {@link MessageMapping} wrapping be enabled.
spring.sleuth.messaging.enabled	false	Should messaging be turned on.
spring.sleuth.messaging.jms.enabled	true	Enable tracing of JMS.
spring.sleuth.messaging.jms.remote-service-name	jms	JMS remote service name.
spring.sleuth.messaging.kafka.enabled	true	Enable tracing of Kafka.
spring.sleuth.messaging.kafka.mapper.enabled	true	Enable DefaultKafkaHeaderMapper tracing for Kafka.
spring.sleuth.messaging.kafka.remote-service-name	kafka	Kafka remote service name.
spring.sleuth.messaging.kafka.streams.enabled	false	Should Kafka Streams be turned on.
spring.sleuth.messaging.rabbit.enabled	true	Enable tracing of RabbitMQ.
spring.sleuth.messaging.rabbit.remote-service-name	rabbitmq	Rabbit remote service name.
spring.sleuth.mongodb.enabled	true	Enable tracing for MongoDb.
spring.sleuth.mongodb.socket-address-span-customizer.enabled	false	Enable setting of SocketAddress information on the Mongo span.
spring.sleuth.opentracing.enabled	true	Enables OpenTracing support.
spring.sleuth.propagation.type		Tracing context propagation types.

Name	Default	Description
spring.sleuth.quartz.enabled	`true`	Enable tracing for Quartz.
spring.sleuth.r2dbc.enabled	`true`	Enable R2dbc instrumentation.
spring.sleuth.reactor.decorate-on-each	`true`	When true decorates on each operator, will be less performing, but logging will always contain the tracing entries in each operator. When false decorates on last operator, will be more performing, but logging might not always contain the tracing entries. @deprecated use explicit value via {@link SleuthReactorProperties#instrumentationType}
spring.sleuth.reactor.enabled	`true`	When true enables instrumentation for reactor.
spring.sleuth.reactor.instrumentation-type		
spring.sleuth.reactor.netty.debug.enabled	`false`	WARNING: Use with caution, can lead to serious performance issues. Enable additional instrumentation for Reactor Netty.
spring.sleuth.redis.enabled	`true`	Enable span information propagation when using Redis.
spring.sleuth.redis.legacy.enabled	`false`	Enable legacy tracing of Redis that works only via Brave.
spring.sleuth.redis.remote-service-name	`redis`	Service name for the remote Redis endpoint.
spring.sleuth.rpc.enabled	`true`	Enable tracing of RPC.
spring.sleuth.rsocket.enabled	`true`	When true enables instrumentation for rsocket.
spring.sleuth.rxjava.schedulers.hook.enabled	`true`	Enable support for RxJava via RxJavaSchedulersHook.
spring.sleuth.rxjava.schedulers.ignoredthreads	`[HystrixMetricPoller, ^RxComputation.*$]`	Thread names for which spans will not be sampled.

Name	Default	Description
spring.sleuth.sampler.probability		Probability of requests that should be sampled. E.g. 1.0 - 100% requests should be sampled. The precision is whole-numbers only (i.e. there's no support for 0.1% of the traces).
spring.sleuth.sampler.rate	`10`	A rate per second can be a nice choice for low-traffic endpoints as it allows you surge protection. For example, you may never expect the endpoint to get more than 50 requests per second. If there was a sudden surge of traffic, to 5000 requests per second, you would still end up with 50 traces per second. Conversely, if you had a percentage, like 10%, the same surge would end up with 500 traces per second, possibly overloading your storage. Amazon X-Ray includes a rate-limited sampler (named Reservoir) for this purpose. Brave has taken the same approach via the {@link brave.sampler.RateLimitingSampler}.
spring.sleuth.sampler.refresh.enabled	`true`	Enable refresh scope for sampler.
spring.sleuth.scheduled.enabled	`true`	Enable tracing for {@link org.springframework.scheduling.annotation.Scheduled}.
spring.sleuth.scheduled.skip-pattern		Pattern for the fully qualified name of a class that should be skipped.
spring.sleuth.session.enabled	`true`	Enable Spring Session instrumentation.
spring.sleuth.span-filter.additional-span-name-patterns-to-ignore		Additional list of span names to ignore. Will be appended to {@link #spanNamePatternsToSkip}.

Name	Default	Description
spring.sleuth.span-filter.enabled	`false`	Will turn on the default Sleuth handler mechanism. Might ignore exporting of certain spans;
spring.sleuth.span-filter.span-name-patterns-to-skip	`^catalogWatchTaskScheduler$`	List of span names to ignore. They will not be sent to external systems.
spring.sleuth.supports-join	`true`	True means the tracing system supports sharing a span ID between a client and server.
spring.sleuth.task.enabled	`true`	Enable Spring Cloud Task instrumentation.
spring.sleuth.trace-id128	`false`	When true, generate 128-bit trace IDs instead of 64-bit ones.
spring.sleuth.tracer.mode		Set which tracer implementation should be picked.
spring.sleuth.tx.enabled	`true`	Enable Spring TX instrumentation.
spring.sleuth.vault.enabled	`true`	Enable Spring Vault instrumentation.
spring.sleuth.web.additional-skip-pattern		Additional pattern for URLs that should be skipped in tracing. This will be appended to the {@link SleuthWebProperties#skipPattern}.
spring.sleuth.web.client.enabled	`true`	Enable interceptor injecting into {@link org.springframework.web.client.RestTemplate}.
spring.sleuth.web.client.skip-pattern		Pattern for URLs that should be skipped in client side tracing.
spring.sleuth.web.enabled	`true`	When true enables instrumentation for web applications.
spring.sleuth.web.filter-order	`0`	Order in which the tracing filters should be registered.
spring.sleuth.web.ignore-auto-configured-skip-patterns	`false`	If set to true, auto-configured skip patterns will be ignored.

Name	Default	Description
spring.sleuth.web.servlet.enabled	`true`	Enable servlet instrumentation.
spring.sleuth.web.skip-pattern	`/api-docs.*|/swagger.*|.*\.png|.*\.css|.*\.js|.*\.html|/favicon.ico|/hystrix.stream`	Pattern for URLs that should be skipped in tracing.
spring.sleuth.web.tomcat.enabled	`true`	Enable tracing instrumentation for Tomcat.
spring.sleuth.web.webclient.enabled	`true`	Enable tracing instrumentation for WebClient.
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.api-path		The API path to append to baseUrl (above) as suffix. This applies if you use other monitoring tools, such as New Relic. The trace API doesn't need the API path, so you can set it to blank ("") in the configuration.
spring.zipkin.base-url	`http://localhost:9411/`	URL of the zipkin query server instance. You can also provide the service id of the Zipkin server if Zipkin's registered in service discovery (e.g. zipkinserver/).
spring.zipkin.check-timeout	`1000`	Timeout in millis for the check for Zipkin availability.
spring.zipkin.compression.enabled	`false`	
spring.zipkin.discovery-client-enabled		If set to {@code false}, will treat the {@link ZipkinProperties#baseUrl} as a URL always.
spring.zipkin.enabled	`true`	Enables sending spans to Zipkin.

Name	Default	Description
spring.zipkin.encoder		Encoding type of spans sent to Zipkin. Set to {@link SpanBytesEncoder#JSON_V1} if your server is not recent.
spring.zipkin.kafka.topic	`zipkin`	Name of the Kafka topic where spans should be sent to Zipkin.
spring.zipkin.locator.discovery.enabled	`false`	Enabling of locating the host name via service discovery.
spring.zipkin.message-timeout	`1`	Timeout in seconds before pending spans will be sent in batches to Zipkin.
spring.zipkin.queued-max-spans	`1000`	Maximum backlog of spans reported vs sent.
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.
spring.zipkin.service.name		The name of the service, from which the Span was sent via HTTP, that should appear in Zipkin.

6.31. Spring Cloud Sleuth Spans

Below you can find a list of all the spans that are created by Spring Cloud Sleuth.

6.31.1. Annotation New Or Continue Span

> Span that wraps a @NewSpan or @ContinueSpan annotations.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.annotation.SleuthAnnotationSpan`

Table 1. Tag Keys

Name	Description

class	Class name where a method got annotated with a Sleuth annotation.
method	Method name that got annotated with Sleuth annotation.

Table 2. Event Values

Name	Description
%s.after	Annotated after executing a method annotated with @ContinueSpan or @NewSpan. (since the name contains `%s` the final value will be resolved at runtime)
%s.afterFailure	Annotated after throwing an exception from a method annotated with @ContinueSpan or @NewSpan. (since the name contains `%s` the final value will be resolved at runtime)
%s.before	Annotated before executing a method annotated with @ContinueSpan or @NewSpan. (since the name contains `%s` the final value will be resolved at runtime)

6.31.2. Async Annotation Span

> Span that wraps a @Async annotation. Either continues an existing one or creates a new one if there was no present one.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.async.SleuthAsyncSpan`

Table 3. Tag Keys

Name	Description
class	Class name where a method got annotated with @Async.
method	Method name that got annotated with @Async.

6.31.3. Async Callable Span

> Span created whenever a Callable needs to be instrumented.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.async.SleuthAsyncSpan`

6.31.4. Async Runnable Span

> Span created whenever a Runnable needs to be instrumented.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.async.SleuthAsyncSpan`

6.31.5. Batch Job Span

> Span created around a Job execution.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.batch.SleuthBatchSpan`

Table 4. Tag Keys

Name	Description
batch.job.executionId	ID of the Spring Batch execution.
batch.job.instanceId	ID of the Spring Batch job instance.
batch.job.name	Name of the Spring Batch job.

6.31.6. Batch Step Span

> Span created around a Job execution.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.batch.SleuthBatchSpan`

Table 5. Tag Keys

Name	Description
batch.job.executionId	ID of the Spring Batch execution.
batch.step.executionId	ID of the Spring Batch execution.
batch.step.name	Name of the Spring Batch job.
batch.step.type	Type of the Spring Batch job.

6.31.7. Cassandra Span

> Span created around CqlSession executions.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.cassandra.SleuthCassandraSpan`

All tags and events must be prefixed with `cassandra.` prefix!

Table 6. Tag Keys

Name	Description
cassandra.cql	A tag containing Cassandra CQL.
cassandra.keyspace	Name of the Cassandra keyspace.
cassandra.node[%s].error	A tag containing error that occurred for the given node. (since the name contains `%s` the final value will be resolved at runtime)

Table 7. Event Values

Name	Description
cassandra.node.error	Set whenever an error occurred for the given node.
cassandra.node.success	Set when a success occurred for the session processing.

6.31.8. Circuit Breaker Function Span

> Span created when we wrap a Function passed to the CircuitBreaker. as fallback.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.circuitbreaker.SleuthCircuitBreakerSpan`

6.31.9. Circuit Breaker Supplier Span

> Span created when we wrap a Supplier passed to the CircuitBreaker.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.circuitbreaker.SleuthCircuitBreakerSpan`

6.31.10. Config Span

> Span created around an EnvironmentRepository.

Span name `find`.

Fully qualified name of the enclosing class

`org.springframework.cloud.sleuth.instrument.config.SleuthConfigSpan`

Table 8. Tag Keys

Name	Description
config.environment.class	Implementation of the EnvironmentRepository.
config.environment.method	Method executed on the EnvironmentRepository.

6.31.11. Deployer Deploy Span

> Span created upon deploying of an application.

Span name `deploy`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.deployer.SleuthDeployerSpan`

Table 9. Tag Keys

Name	Description
deployer.app.group	Group of the deployed application.
deployer.app.id	ID of the deployed application.
deployer.app.name	Name of the deployed application.
deployer.platform.cf.org	CloudFoundry org.
deployer.platform.cf.space	CloudFoundry space.
deployer.platform.cf.url	CloudFoundry API URL.
deployer.platform.k8s.namespace	Kubernetes namespace.
deployer.platform.k8s.url	Kubernetes API URL.
deployer.platform.name	Name of the platform to which apps are being deployed.

Table 10. Event Values

Name	Description
%s	When deployer changes the state of the deployed application. (since the name contains `%s` the final value will be resolved at runtime)
deployer.start	When deployer started deploying the application.

6.31.12. Deployer Get Log Span

> Span created upon asking for logs of deployed applications.

Span name `getLog`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.deployer.SleuthDeployerSpan`

Table 11. Tag Keys

Name	Description
deployer.app.group	Group of the deployed application.
deployer.app.id	ID of the deployed application.
deployer.app.name	Name of the deployed application.
deployer.platform.cf.org	CloudFoundry org.
deployer.platform.cf.space	CloudFoundry space.
deployer.platform.cf.url	CloudFoundry API URL.
deployer.platform.k8s.namespace	Kubernetes namespace.
deployer.platform.k8s.url	Kubernetes API URL.
deployer.platform.name	Name of the platform to which apps are being deployed.

Table 12. Event Values

Name	Description
%s	When deployer changes the state of the deployed application. (since the name contains `%s` the final value will be resolved at runtime)
deployer.start	When deployer started deploying the application.

6.31.13. Deployer Scale Span

> Span created upon asking for logs of deployed applications.

Span name `scale`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.deployer.SleuthDeployerSpan`

Table 13. Tag Keys

Name	Description
deployer.app.group	Group of the deployed application.
deployer.app.id	ID of the deployed application.
deployer.app.name	Name of the deployed application.
deployer.platform.cf.org	CloudFoundry org.

deployer.platform.cf.space	CloudFoundry space.
deployer.platform.cf.url	CloudFoundry API URL.
deployer.platform.k8s.namespace	Kubernetes namespace.
deployer.platform.k8s.url	Kubernetes API URL.
deployer.platform.name	Name of the platform to which apps are being deployed.
deployer.scale.count	Scale count.
deployer.scale.deploymentId	Scale command deployment id.

Table 14. Event Values

Name	Description
%s	When deployer changes the state of the deployed application. (since the name contains `%s` the final value will be resolved at runtime)
deployer.start	When deployer started deploying the application.

6.31.14. Deployer Statuses Span

> Span created upon asking for statuses of deployed applications.

Span name `statuses`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.deployer.SleuthDeployerSpan`

Table 15. Tag Keys

Name	Description
deployer.app.group	Group of the deployed application.
deployer.app.id	ID of the deployed application.
deployer.app.name	Name of the deployed application.
deployer.platform.cf.org	CloudFoundry org.
deployer.platform.cf.space	CloudFoundry space.
deployer.platform.cf.url	CloudFoundry API URL.
deployer.platform.k8s.namespace	Kubernetes namespace.
deployer.platform.k8s.url	Kubernetes API URL.
deployer.platform.name	Name of the platform to which apps are being deployed.

Table 16. Event Values

Name	Description
%s	When deployer changes the state of the deployed application. (since the name contains `%s` the final value will be resolved at runtime)
deployer.start	When deployer started deploying the application.

6.31.15. Deployer Status Span

> Span created upon asking for a status of a deployed application.

Span name `status`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.deployer.SleuthDeployerSpan`

Table 17. Tag Keys

Name	Description
deployer.app.group	Group of the deployed application.
deployer.app.id	ID of the deployed application.
deployer.app.name	Name of the deployed application.
deployer.platform.cf.org	CloudFoundry org.
deployer.platform.cf.space	CloudFoundry space.
deployer.platform.cf.url	CloudFoundry API URL.
deployer.platform.k8s.namespace	Kubernetes namespace.
deployer.platform.k8s.url	Kubernetes API URL.
deployer.platform.name	Name of the platform to which apps are being deployed.

Table 18. Event Values

Name	Description
%s	When deployer changes the state of the deployed application. (since the name contains `%s` the final value will be resolved at runtime)
deployer.start	When deployer started deploying the application.

6.31.16. Deployer Undeploy Span

> Span created upon undeploying of an application.

Span name `undeploy`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.deployer.SleuthDeployerSpan`

Table 19. Tag Keys

Name	Description
deployer.app.group	Group of the deployed application.
deployer.app.id	ID of the deployed application.
deployer.app.name	Name of the deployed application.
deployer.platform.cf.org	CloudFoundry org.
deployer.platform.cf.space	CloudFoundry space.
deployer.platform.cf.url	CloudFoundry API URL.
deployer.platform.k8s.namespace	Kubernetes namespace.
deployer.platform.k8s.url	Kubernetes API URL.
deployer.platform.name	Name of the platform to which apps are being deployed.

Table 20. Event Values

Name	Description
%s	When deployer changes the state of the deployed application. (since the name contains `%s` the final value will be resolved at runtime)
deployer.start	When deployer started deploying the application.

6.31.17. Jdbc Connection Span

> Span created when a JDBC connection takes place.

Span name `connection`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.jdbc.SleuthJdbcSpan`

All tags and events must be prefixed with `jdbc.` prefix!

Table 21. Tag Keys

Name	Description
jdbc.datasource.driver	Name of the JDBC datasource driver.
jdbc.datasource.pool	Name of the JDBC datasource pool.

6.31.18. Jdbc Query Span

> Span created when a JDBC query gets executed.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.jdbc.SleuthJdbcSpan`

All tags and events must be prefixed with `jdbc.` prefix!

Table 22. Tag Keys

Name	Description
jdbc.query	The SQL query value.
jdbc.row-count	Number of SQL rows.

Table 23. Event Values

Name	Description
jdbc.commit	When the transaction gets committed.
jdbc.rollback	When the transaction gets rolled back.

6.31.19. Jdbc Result Set Span

> Span created when working with JDBC result set.

Span name `result-set`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.jdbc.SleuthJdbcSpan`

All tags and events must be prefixed with `jdbc.` prefix!

Table 24. Tag Keys

Name	Description
jdbc.query	The SQL query value.
jdbc.row-count	Number of SQL rows.

Table 25. Event Values

Name	Description
jdbc.commit	When the transaction gets committed.
jdbc.rollback	When the transaction gets rolled back.

6.31.20. Kafka Consumer Span

> Span created on the Kafka consumer side.

Span name `kafka.consume`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.kafka.SleuthKafkaSpan`

All tags and events must be prefixed with `kafka.` prefix!

Table 26. Tag Keys

Name	Description
kafka.offset	Kafka offset number.
kafka.partition	Kafka partition number.
kafka.topic	Name of the Kafka topic.

6.31.21. Kafka On Message Span

> Span created on the Kafka consumer side when using a MessageListener.

Span name `kafka.on-message`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.kafka.SleuthKafkaSpan`

All tags and events must be prefixed with `kafka.` prefix!

Table 27. Tag Keys

Name	Description
kafka.offset	Kafka offset number.
kafka.partition	Kafka partition number.
kafka.topic	Name of the Kafka topic.

6.31.22. Kafka Producer Span

> Span created on the Kafka consumer side.

Span name `kafka.produce`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.kafka.SleuthKafkaSpan`

All tags and events must be prefixed with `kafka.` prefix!

Table 28. Tag Keys

Name	Description
kafka.topic	Name of the Kafka topic.

6.31.23. Messaging Span

> Span created when message is sent or received.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.messaging.SleuthMessagingSpan`

Table 29. Tag Keys

Name	Description
%s	User provided keys via customization options. (since the name contains `%s` the final value will be resolved at runtime)
channel	Name of the Spring Integration channel.
function.name	Name of the Spring Cloud Function function name.

6.31.24. Mvc Handler Interceptor Span

> Span around a HandlerInterceptor. Will continue the current span and tag it

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.web.mvc.SleuthMvcSpan`

Table 30. Tag Keys

Name	Description
mvc.controller.class	Class name where a method got annotated with @Scheduled.
mvc.controller.method	Method name that got annotated with @Scheduled.

6.31.25. Quartz Trigger Span

> Span created when trigger is fired and then completed.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class

`org.springframework.cloud.sleuth.instrument.quartz.SleuthQuartzSpan`

Table 31. Tag Keys

Name	Description
quartz.trigger	Name of the trigger.

6.31.26. R2dbc Query Span

> Span created on the Kafka consumer side.

Span name `query`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.r2dbc.SleuthR2dbcSpan`

All tags and events must be prefixed with `r2dbc.` prefix!

Table 32. Tag Keys

Name	Description
r2dbc.connection	Name of the R2DBC connection.
r2dbc.query[%s]	Name of the R2DBC query. (since the name contains `%s` the final value will be resolved at runtime)
r2dbc.thread	Name of the R2DBC thread.

6.31.27. Rsocket Requester Span

> Span created on the RSocket responder side.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.rsocket.SleuthRSocketSpan`

All tags and events must be prefixed with `rsocket.` prefix!

Table 33. Tag Keys

Name	Description
rsocket.request-type	Name of the R2DBC thread.
rsocket.route	Name of the RSocket route.

6.31.28. Rsocket Responder Span

> Span created on the RSocket responder side.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.rsocket.SleuthRSocketSpan`

6.31.29. Rx Java Trace Action Span

> Span that wraps a Rx Java .

Span name `rxjava`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.rxjava.SleuthRxJavaSpan`

Table 34. Tag Keys

Name	Description
thread	Name of the thread.

6.31.30. Scheduled Annotation Span

> Span that wraps a annotated method. Either creates a new span or continues an existing one.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.scheduling.SleuthSchedulingSpan`

Table 35. Tag Keys

Name	Description
class	Class name where a method got annotated with @Scheduled.
method	Method name that got annotated with @Scheduled.

6.31.31. Security Context Change

> Indicates that a SecurityContextChangedEvent happened during the current span.

Span name `Security Context Change`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.security.SleuthSecuritySpan`

Table 36. Event Values

Name	Description

Authentication cleared %s	Event created when an Authentication object is removed from the SecurityContext. (since the name contains `%s` the final value will be resolved at runtime)
Authentication replaced %s	Event created when an Authentication object is replaced with a new one in the SecurityContext. (since the name contains `%s` the final value will be resolved at runtime)
Authentication set %s	Event created when an Authentication object is added to the SecurityContext. (since the name contains `%s` the final value will be resolved at runtime)

6.31.32. Session Create Span

> Span created when a new session has to be created.

Span name `session.create`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.session.SleuthSessionSpan`

6.31.33. Session Delete Span

> Span created when a session is deleted.

Span name `session.delete`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.session.SleuthSessionSpan`

6.31.34. Session Find Span

> Span created when a new session is searched for.

Span name `session.find`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.session.SleuthSessionSpan`

All tags and events must be prefixed with `session.` prefix!

Table 37. Tag Keys

Name	Description
session.index.name	

6.31.35. Session Save Span

> Span created when a new session is saved.

Span name `session.save`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.session.SleuthSessionSpan`

6.31.36. Task Execution Listener Span

> Span created within the lifecycle of a task.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.task.SleuthTaskSpan`

6.31.37. Task Runner Span

> Span created when a task runner is executed.

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.task.SleuthTaskSpan`

6.31.38. Tx Span

> Span created when there was no previous transaction. If there was one, we will continue it unless propagation is required.

Span name `tx`.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.tx.SleuthTxSpan`

All tags and events must be prefixed with `tx.` prefix!

Table 38. Tag Keys

Name	Description
tx.isolation-level	Transaction isolation level.
tx.name	Transaction name.
tx.propagation-level	Transaction propagation level.
tx.read-only	Whether the transaction is read-only.
tx.timeout	Transaction timeout.

tx.transaction-manager	Name of the TransactionManager.

6.31.39. Web Filter Span

> Span around a WebFilter. Will continue the current span or create a new one and tag it

Span name `%s` - since it contains `%s`, the name is dynamic and will be resolved at runtime.

Fully qualified name of the enclosing class `org.springframework.cloud.sleuth.instrument.web.SleuthWebSpan`

Table 39. Tag Keys

Name	Description
http.status_code	Response status code.
mvc.controller.class	Name of the class that is processing the request.
mvc.controller.method	Name of the method that is processing the request.