Advanced Spring

Distributed Tracing with Spring

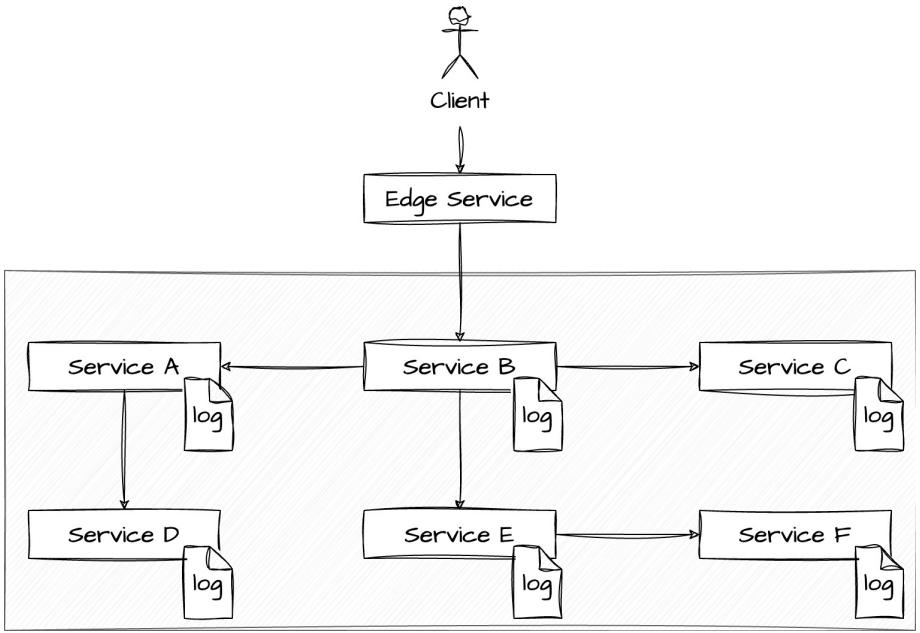
Boris Fresow, Markus Günther

Adesso eduCamp 2023

Mastichari, Kos

Some difficult questions

- Which services are involved in a use-case?
- Where are our bottlenecks?
- Why do we see so many 4xx / 5xx in our logs?
- Are we hitting our SLOs/NFRs?

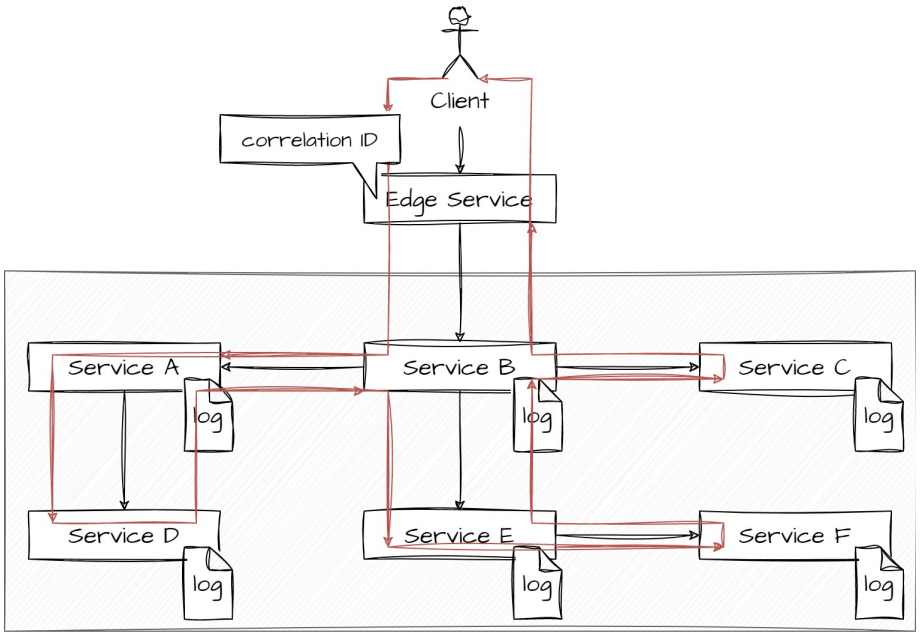


Distributed Tracing

Distributed tracing provides end-to-end visibility into the performance and behavior of requests across interconnected services in a distributed system, enabling faster troubleshooting and optimization of the application.

The solution

- A unique ID that is passed from system to system
- All systems log into a central component
- Traces with the same ID are combined into a single trail



Key Points

- (Enhanced) Observability
  - Comprehensive view of interaction between services
- Faster root cause analysis
  - Especially for Bottlenecks / Latency issues
- Service Dependency Visibility
  - Get a per use-case view of all involved services
- Anomaly Detection
  - Basis for monitoring, especially for non-functional requirements

Components

Conceptual Components

- **Instrumentation**
  - Capture & record information about performance characteristics
- **Propagation**
  - Provide the necessary information to correlate further operations
  - Pass HTTP-Headers for HTTP Requests
  - Log the span- / trace-id with the service logs for later analysis
- **Collection**
  - Aggregate the collected data for a context

- Send it to a central component

Conceptual Components (cont.)

- **Storage & Processing**
  - Persist data received
  - Correlate the related datasets
- **Visualization & Analysis**
  - User Interface for the data
  - Capability to search & filter
- **Alerting & Integration**
  - Automated analysis based on the collected data

Libraries and Systems

- **Brave**: Tracer Library for Instrumentation (developed and maintained by Zipkin)
- **Spring Cloud Sleuth**: Layer on top of Brave for smooth Spring integration
  - *No longer available for Spring Cloud 2022.x*
- **Micrometer Observation**: Vendor/technology agnostic metrics collector for Java
  - Can be used with Brave & Zipkin or other technologies
  - *The new default for Spring Cloud 2022.x*
- **Zipkin**: Distributed Tracing System for collection & visualization

Data Model

- **Trace**: Unique object that contains 1..n spans - builds a latency tree
- **Span**: Single host-view of an operation. Most important fields are:
  - `traceId`: the root trace ID
  - `parentId`: the ID of the parent span (null if root)
  - `name`: the name of the span
- **Tag**: A marker for a span - can be used in a query
- **Annotation**: A specific point in time in a span with an attached string value

Spring Cloud Sleuth

Spring Cloud Sleuth

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.

We'll only take a quick look at Sleuth for the sake of legacy knowledge

Integration

- Easy integration via Spring Cloud dependencies

```
<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>
```

- This includes the auto configuration part that ...
  - exposes Java beans for customizing data collection & presentation
  - decorates existing beans (e.g. `RestTemplate` for propagation)

Configuration

- The default configuration should be changed - for our example we used this config

```
spring.application.name: Client Application

spring:
    zipkin:
        base-url: ${ZIPKIN_API_ENDPOINT:http://127.0.0.1:9411/}
    sleuth:
        sampler:
            probability: 1.0
```

- Application name will be reported with the spans
- The Base-URL is where the traces will be sent to
- The probability default is `0.1` - so only 10% of traces would be gathered
    - Crucial overload protection for high load systems

Creating Traces

- Spans will be created and reported implictly for supported operations
  - `@Scheduled`, `@RestController`, …
  - The current span can also be modified
- Can also be done explicitly
  - Via annotations
  - Programatically

Creating Traces via Annotations

- `@NewSpan("name")` tells Sleuth to open a new span with the given name
  - Will be the child of the already existing span - if there is one
- `@ContinueSpan` can be used to annotate the current span
- `@SpanTag` can be used on parameters to add them as tags

```
@NewSpan("user-lookup")
public User lookupUserByName(@SpanTag(key = "username") final String username) {
  (...)
}
```

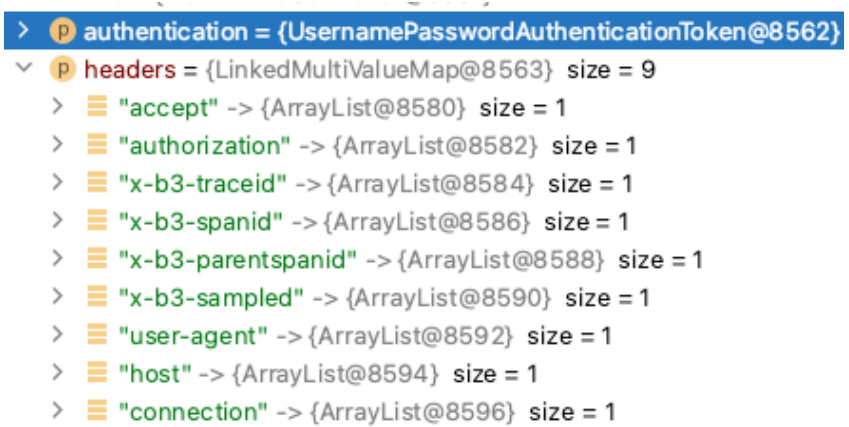
Creating Traces programmatically

- `@NewSpan("name")` tells Sleuth to open a new span with the given name
  - Will be the child of the already existing span - if there is one
- `@ContinueSpan` can be used to annotate the current span
- `@SpanTag` can be used on parameters to add them as tags

```
Span newSpan = null;
try (Tracer.SpanInScope ws = this.tracer.withSpanInScope(initialSpan)) {
  newSpan = this.tracer.nextSpan().name("calculateCommission");
  newSpan.tag("username", username)
} finally {
  if (newSpan != null) {
    newSpan.finish();
  }
}
```

Propagation

- Sleuth will add the tracing context to outgoing requests
  - will work out of the box for directly supported mechanisms (e.g. `RestTemplate`)
- For HTTP requests HTTP-Headers (`x-b3-`) will be added
- Instrumentation library on the receiving side will continue the context



Micrometer Observation

Micrometer Observation

- Micrometer is a very powerful library for collecting metrics
  - You will hear more about this tomorrow!
- The Micrometer Observation API is vendor neutral
  - Modular approach allows a *do once, use many times* approach.
  - Has a bridge to Brave and therefore Zipkin

Integration

- Dependencies are not from Spring
- Requires out-of-band dependency management
- Some integration with `spring-boot-starter-actuator` for autowiring beans

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-observation</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-brave</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.reporter2</groupId>
    <artifactId>zipkin-reporter-brave</artifactId>
</dependency>
```

Integration (cont.)

- Has a dedicated dependency for test support
  - Very useful - we will take a look at this tomorrow

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-observation-test</artifactId>
    <scope>test</scope>
</dependency>
```

Architecture

- `Observation` is a wrapper class for an observable operation
- `Observations` follow a life cycle
  - `start`: The operation has begun execution
  - `scope started`: A scope within the operation has been opened
  - `scope ended`: A scope was closed
  - `event`: An event has happened while observing (e.g. annotations in Sleuth)
  - `error`: The operation threw an error
  - `stop`: The operation has ended

Architecture

- `Observations` are handled by `ObservationHandlers`
  - The signature is `ObservationHandler<T extends Observation.Context>`
  - The interface provides handling methods for all lifecycle events
- Micrometer provides an `ObservationRegistry` to register `ObservationHandler`
  - `ObservationHandler` are called when they support a specific `Observation.Context`
- The `Observation.Context` is a mutable data holder to share data between states
  - The context will be propagated between threads!

Configuration

```
spring.application.name: Micrometer Tracing
logging.pattern.level: "%5p [`${spring.zipkin.service.name:$`{spring.application.name:}},%X{traceId:-},%X{spanId:-}

management:
    tracing:
        propagation:
            type: b3
        sampling:
            probability: 1.0
    zipkin:
        tracing:
            endpoint: ${ZIPKIN_API_ENDPOINT:http://127.0.0.1:9411/api/v2/spans}
```

- Probability and endpoint are similar (plus path in URL) to Sleuth
- We need to reconfigure the logging pattern manually to log trace- and span-IDs
- Propagation type b3 for interop with Sleuth (default is w3c)

Configuration (cont.)

- To use the `@Observed` annotation we need to register the Bean for AOP
- The `ObservationRegistry` is the registry responsible for observation state management

```
@Bean
public ObservedAspect observedAspect(ObservationRegistry observationRegistry) {
  return new ObservedAspect(observationRegistry);
}
```

Creating Observations via Annotations

- `@Observed` tells Micrometer that the method should be observed

```
@Observed(
  name = "time.reporting",
  contextualName = "time-report",
  lowCardinalityKeyValues = {"class.name", "TimeReportingTask"}
)
public void reportTimeTask() {
  (...)
}
```

Creating Observations programatically

- `Observation` offers several static factory methods to create a new `Observation`
- `observe` will
  - start/stop the `Observation` (if not already done)
  - Open/close a new scope
  - Catch all errors

```
void observeWork(@Autowired ObservationRegistry registry) {
  final var context = new Observation.Context().put(String.class, "information");
  // The context is optional
  Observation
    .createNotStarted("operation.name", () -> context, registry)
    .observe(this::doWork);
}
```

Manipulate Observations

- The `ObservationRegistry` is the global holder for all `Observations` (comparable to the `Tracer` in Sleuth)
- Fetch current observation via `registry.getCurrentObservation()`
  - Will be `null` if we're not being observed

```
void doWork(@Autowired ObservationRegistry registry) {
  final var observation = registry.getCurrentObservation();
  (...)
}
```

Observations Bridge

- An `Observation` conceptually has nothing to do with a trace
- `micrometer-tracing` provides a set of `ObservationHandler` to handle `Observations`
  - Transitive dependency of `micrometer-tracing-bridge-brave`
- `micrometer-tracing-bridge-brave` is responsible for
  - converting the traces to Brave
  - Reporting to Zipkin via Brave

Zipkin

Zipkin

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.

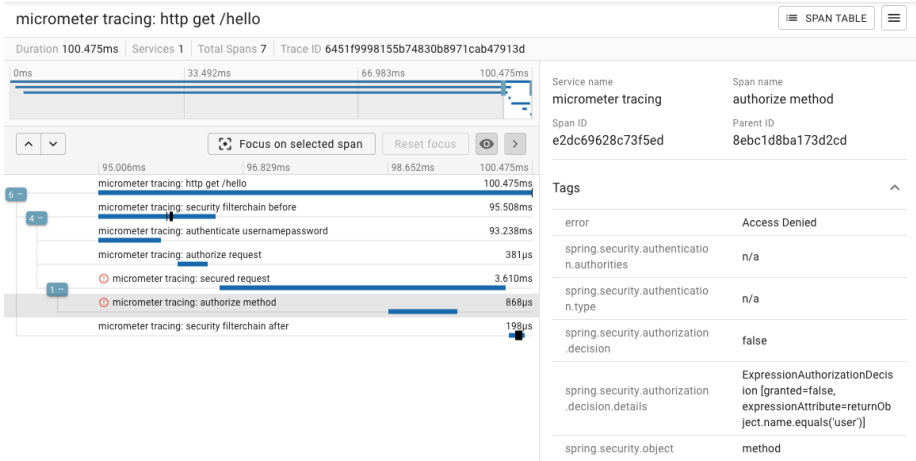
Zipkin (cont.)

- Zipkin provides
  - a HTTP Facade to receive (and query!) spans
  - an UI for visualization & querying
- Is usually run as a standalone application
  - but can be integrated in a service as well
- Spans can be collected via HTTP, Kafka or RabbitMQ
- Storage options are in-memory, MySQL, Cassandra or Elasticsearch
  - There are a lot of community supported plugins e.g. for SQS, Kinesis

Zipkin (cont.)

Lab assignments will focus on

- visualization & querying of data
- tracing HTTP Requests



Questions?

Lab

Lab

It's time to get started!

Let's take a look at the repository and README.md