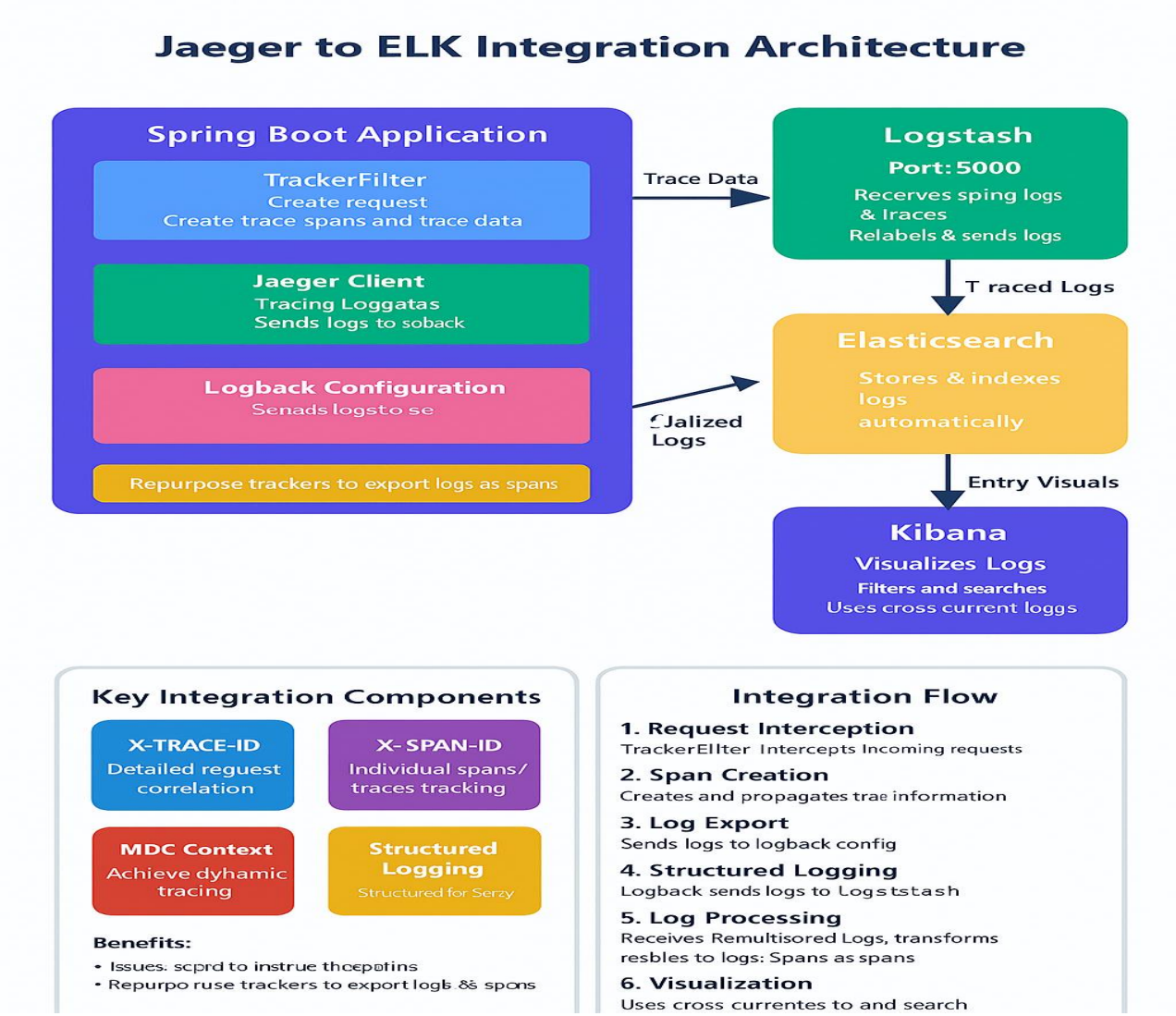
# Jaeger Tracing Integration with ELK Stack for Spring Boot

This documentation outlines the process of integrating **Jaeger** with **Spring Boot** for distributed tracing and configuring **ELK (Elasticsearch, Logstash, Kibana)** to visualize trace logs.



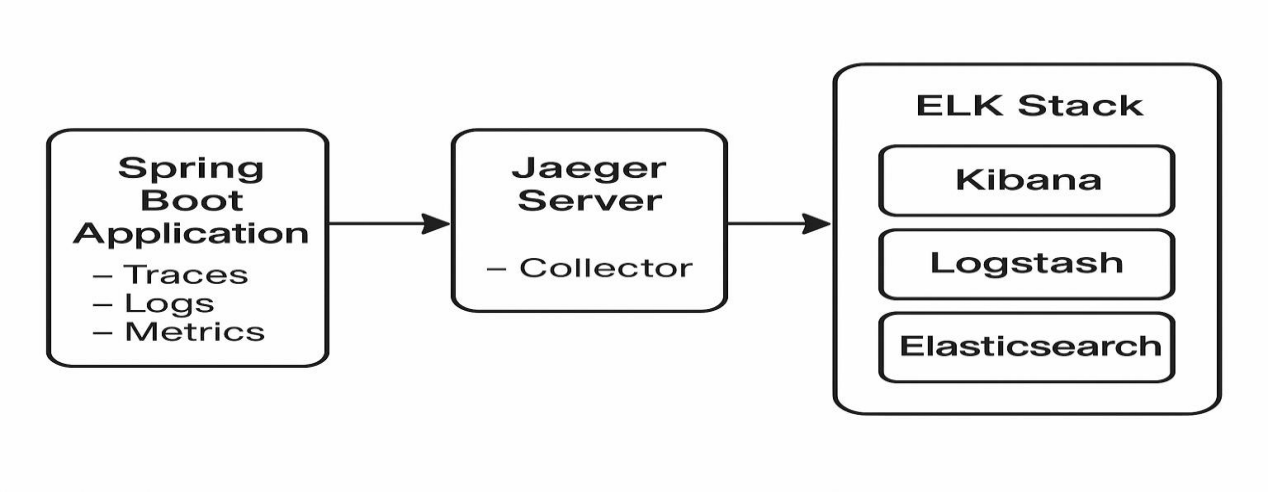
## 🐳 Step 1: Run Jaeger Using Docker Compose

Start by configuring **Jaeger** using Docker Compose.

`docker-compose.yml`

```
version: '3.7'

services:
  jaeger:
    image: jaegertracing/all-in-one:1.50
    container_name: jaeger
    environment:
      - COLLECTOR_OTLP_ENABLED=true
    ports:
      - "6831:6831/udp"    # Jaeger Agent (Thrift over UDP)
      - "16686:16686"      # Jaeger UI
```




---

## ⚙️ Step 2: Configure Jaeger in Spring Boot

Add the following configuration in your `application.yml`:

```
jaeger:
  service-name: jaegar-service
  sampler:
    type: const
    param: 1
  reporter:
    log-spans: false
    sender:
      agent-host: jaeger
      agent-port: 6831
```

# Jaeger Tracing Configuration Guide

## ❓ Step 3: Add Dependencies in pom.xml

```
<dependency>
    <groupId>io.jaegertracing</groupId>
    <artifactId>jaeger-client</artifactId>
    <version>1.8.1</version>
</dependency>
<dependency>
    <groupId>io.opentracing</groupId>
    <artifactId>opentracing-api</artifactId>
    <version>0.33.0</version>
</dependency>
```

These dependencies enable OpenTracing and Jaeger tracing in Spring Boot.

---

## ❓ Step 4: Create a Jaeger Tracer Configuration

```
@Configuration
public class JaegerConfig {

    @Bean
    public Tracer jaegerTracer() {
        return new io.jaegertracing.Configuration("jaegar-service")
            .withSampler(io.jaegertracing.Configuration.SamplerConfiguration.
fromEnv().withType("const").withParam(1))
            .withReporter(io.jaegertracing.Configuration.ReporterConfiguratio
n.fromEnv().withLogSpans(true))
            .getTracer();
    }
}
```

### ❓ Explanation

This defines a global tracer bean which Spring will use for injecting tracing logic.

---

## Step 5: Global Tracing Filter (Without Modifying Every API)

To trace every HTTP request automatically, add this custom OncePerRequestFilter.

```
@Component
@Order(1)
public class TracingFilter extends OncePerRequestFilter {

    private final Tracer tracer;
    private static final Logger log = LoggerFactory.getLogger(TracingFilter.c
lass);

    public TracingFilter(Tracer tracer) {
        this.tracer = tracer;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletEx
ception, IOException {

        String operationName = request.getMethod() + " " + request.getRequest
URI();

        Map<String, String> headers = new HashMap<>();
        Collections.list(request.getHeaderNames())
                .forEach(header -> headers.put(header, request.getHeader(head
er)));

        Tracer.SpanBuilder spanBuilder = tracer.buildSpan(operationName).igno
reActiveSpan();

        Span span;
        try {
            var context = tracer.extract(Format.Builtin.HTTP_HEADERS, new Tex
tMapAdapter(headers));
            if (context != null) {
                spanBuilder = spanBuilder.asChildOf(context);
            }
            span = spanBuilder.start();
        } catch (Exception e) {
            log.warn("Could not extract span context from headers", e);
            span = spanBuilder.start();
        }

        try (io.opentracing.Scope scope = tracer.scopeManager().activate(span
)) {
            Tags.HTTP_METHOD.set(span, request.getMethod());
            Tags.HTTP_URL.set(span, request.getRequestURL().toString());
            Tags.COMPONENT.set(span, "http");

            String traceId = span.context().toTraceId();
```

```
            String spanId = span.context().toSpanId();

            response.setHeader("x-trace-id", traceId);
            response.setHeader("x-span-id", spanId);

            MDC.put("traceId", traceId);
            MDC.put("spanId", spanId);

            log.info("Starting request: {} {}", request.getMethod(), request.
getRequestURI());

            filterChain.doFilter(request, response);

            Tags.HTTP_STATUS.set(span, response.getStatus());
            log.info("Completed request: {} {} - {}", request.getMethod(), re
quest.getRequestURI(), response.getStatus());
        } catch (Exception e) {
            Tags.ERROR.set(span, true);
            span.log(Map.of("event", "error", "error.object", e));
            log.error("Request failed", e);
            throw e;
        } finally {
            span.finish();
            MDC.clear();
        }
    }
}
```

---

## ✅ Step 6: Configure logback-spring.xml for ELK Compatibility

```
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level [%X{traceId:-
```

```
},%X{spanId:-}] %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <appender name="LOGSTASH" class="net.logstash.logback.appender.LogstashTc
pSocketAppender">
        <destination>localhost:5000</destination>
        <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJso
nEncoder">
            <providers>
                <timestamp/>
                <logLevel/>
                <loggerName/>
                <message/>
                <mdc/>
                <arguments/>
                <pattern>
                    <pattern>
                        {
                            "service": "jaegar-service"
                        }
                    </pattern>
                </pattern>
            </providers>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="LOGSTASH"/>
        <appender-ref ref="STDOUT"/>
    </root>
</configuration>
```

## ⍰ Why x-trace-id and x-span-id Are Important

- **x-trace-id**: A unique ID that represents a complete request lifecycle across multiple services. Helps correlate logs for the same request across microservices.
- **x-span-id**: Identifies a single operation (like a function or DB call) within a trace. Helpful to drill down and understand specific steps of a request.

✅These IDs are injected in response headers and log context (MDC) so that ELK can group logs by the same traceId, allowing visual end-to-end tracing across distributed systems

.