Spring Sleuth and ZipKin

- Advances in technology and cloud computing have made it easier to stand up and deploy services with ease. Cloud computing enables us to automate away the pain (from days or weeks (gasp!) to minutes!) associated with standing up new services. This increase in velocity in turn enables us to be more agile, to think about smaller batches of independently deployable services. The proliferation of new services complicates reasoning about system-wide and request-specific performance characteristics.

- When all of an application's functionality lives in a monolith - what we call applications written as one, large, unbroken deployable like a .war or .ear - it's much easier to reason about where things have gone wrong. Is there a memory leak? It's in the monolith. Is a component not handling requests correctly? It's in the monolith. Messages getting dropped? Also, probably in the monolith. Distribution changes everything.

- Systems behave differently under load and at scale. The specification of a system's behavior often diverges from the actual behavior of the system, and the actual behavior may itself vary in different contexts. It is important to contextualize requests as they transit through a system. It's also important to be able to talk about the nature of a specific request and to be able to understand that specific request's behavior relative to the general behavior of similar requests in the past minute, hour, day (or whatever!) other useful interval provides a statistically significant sampling. Context helps us establish whether a request was abnormal and whether it merits attention. You can't trace bugs in a system until you've established a baseline for what normal is. How long is is long? For some systems it might be microseconds, for others it might be seconds or minutes!

- Tracing is simple, in theory. As a request flows from one component to another in a system, through ingress and egress points, tracers add logic where possible to perpetuate a unique trace ID that's generated when the first request is made. As a request arrives at a component along its journey, a new span ID is assigned for that component and added to the trace. A trace represents the whole journey of a request, and a span is each individual hop along the way, each request. Spans may contain tags, or metadata, that can be used to later contextualize the request. Spans typically contain common tags like start timestamps and stop timestamp, though it's easy to associate semantically relevant tags like an a business entity ID with a span.

- Spring Cloud Sleuth (org.springframework.cloud:spring-cloud-starter-sleuth), once added to the CLASSPATH, automatically instruments common communication channels:

- requests over messaging technologies like Apache Kafka or RabbitMQ (or any other Spring Cloud Stream binder
- HTTP headers received at Spring MVC controllers
- requests that pass through a Netflix Zuul microproxy
- requests made with the RestTemplate, etc.

- Spring Cloud Sleuth also makes this information available to any Spring Cloud Sleuth-aware Spring application by simply injecting the SpanAccessor. You can also use this to instrument your own components that aren't already instrumented by Spring Cloud so that they can perpetuate trace information. Naturally, each tracer is going to be different, but Spring Cloud Sleuth's code itself (e.g.: TraceFeignClientAutoConfiguration) is suggestive of how a typical tracer works:

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.
- If you have a trace ID in a log file, you can jump directly to it. Otherwise, you can query based on attributes such as service, operation name, tags and duration. Some interesting data will be summarized for you, such as the percentage of time spent in a service, and whether or not operations failed.