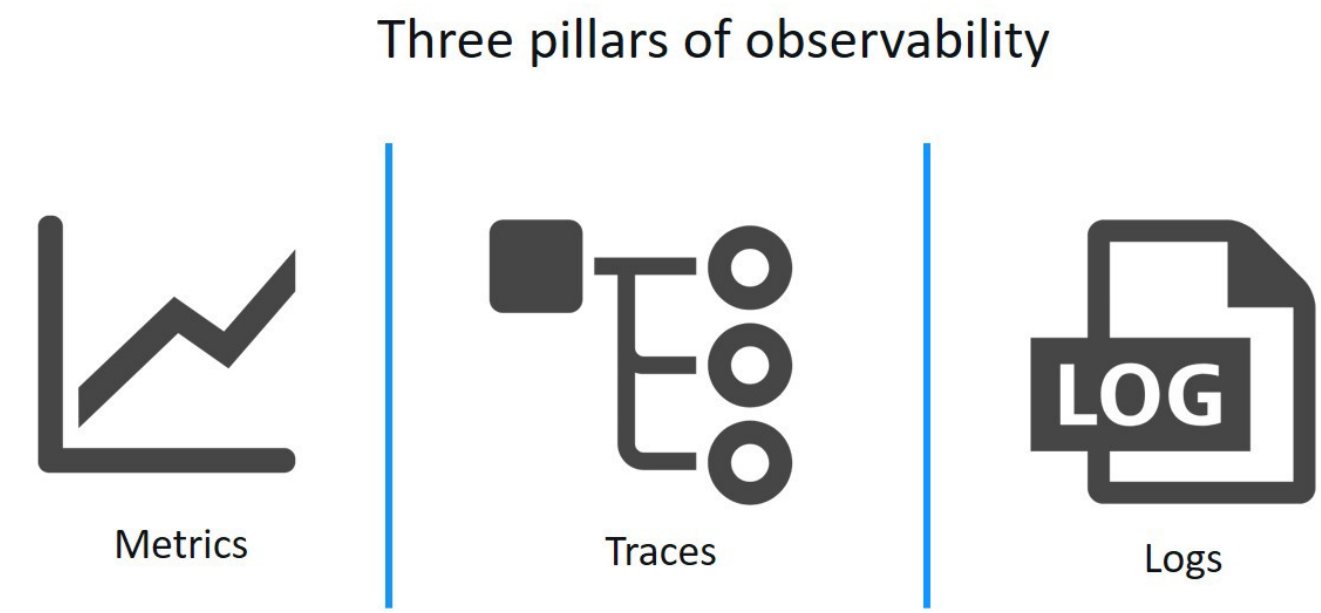
Ajay Yadav

Follow 404 Followers About

Observabilidad en microservicios

Ajay Yadav 31 de ene · 5 min de lectura



En los servicios listos para la producción de artículos , cubrimos brevemente la observabilidad. Ahora profundizaremos en los detalles de la observabilidad.

¿Por qué se requiere la observabilidad en un servicio?

Una vez que implementa un servicio en el entorno de producción, para brindar una calidad de servicio a su cliente, observa algunos parámetros críticos del servicio:

- Supervise su eficiencia en términos de latencia o rendimiento.

- Avise al desarrollador o al equipo de operaciones en caso de cualquier problema en el sistema. Por ejemplo, el disco se llena, el servicio se bloquea, etc.
- Solucione e identifique la causa raíz en caso de que surja un problema.

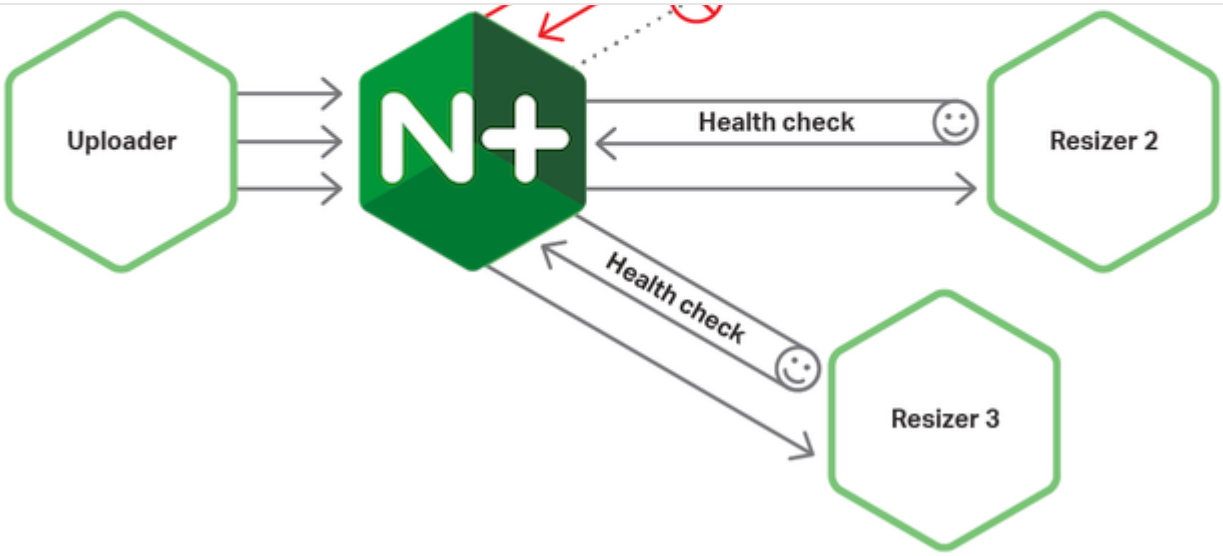
Los siguientes son los patrones de la observabilidad:

- *API de comprobación de estado* : exponga el punto final para comprobar el estado del servicio
- *Agregación de registros* : registra la actividad del servicio y escribe el registro en un lugar centralizado.
- *Rastreo distribuido* : agregue un ID de solicitud a cada solicitud externa y rastree solicitudes a medida que fluyen en el sistema.
- *Seguimiento de excepciones* : envíe excepciones al servicio de seguimiento de excepciones, que deduplica la excepción, alerta a los desarrolladores y realiza un seguimiento de la resolución de cada excepción.
- *Métricas de la aplicación* : el servicio mantiene las métricas y las expone al servidor de métricas.
- Registro de *auditoría* : registra las acciones del usuario.

Patrón de la API de verificación de estado

Cuando se implementa un servicio en el entorno de producción, no desea enrutar las solicitudes hasta que esté listo para aceptar la solicitud. (Sano)

P.ej. Servicios millas g ht toman unos segundos para que éste se inicia en el establecimiento de conexiones de bases de datos, poblando la caché, etc. Es inútil a petición de ruta enviado una solicitud, ya que fallarán en el futuro hasta que el servicio está listo.



Hay dos componentes del patrón de la API de verificación de estado

- Implementación del punto final de verificación de estado
- Infraestructura de implementación para invocar periódicamente el punto final de verificación de estado

API de comprobación de estado

El servicio expone un punto final de verificación de estado, como GET / health, que devuelve el estado del servicio.

La biblioteca Spring Boot actuator implementa un punto final GET / actuator / health que devuelve 200 solo si el servicio está en buen estado y 503 de lo contrario.

Invocación de la API de verificación de estado

The deployment infrastructure periodically invokes the endpoint to identify if the service is healthy and perform the appropriate operation if the service is unhealthy. K8s is a good example of it.

Also, you can implement service registries using Netflix Eureka to invoke health check API periodically.

application, a good starting point is logs.

But in microservice, there are multiple services and for each service, there will be multiple instances. Hence, using logs in a microservice architecture is challenging.

The solution is log aggregation.

Log aggregation

Aggregate the logs of all the services in a centralized database that supports searching and alerting.

ELK is a popular system based on the concept of log aggregation.

ELK consist of three open sources:

- *Elasticsearch*: A text-search oriented NoSQL database that's used as a logging server.
- *Logstash*: A log pipeline that aggregates the service logs and writes them to the elastic search.
- *Kibana*: A visualization tool for Elastic search.

Another tool used across the industry for log aggregation Coralogix.

Distributed Tracing pattern

In a microservice architecture, when a request lands on the first service (or API gateway) it nested invokes many other services and then returns the result to the client.

Distributed tracing is the solution to the problem.

Distributed Tracing

Assign each external request a unique Id and record how it flows through the system from one service to the next service in a centralized server that provides visualization and analysis.

Zipkin is a popular distributed tracing server.

Distributed tracing has two components

- *Trace*: Each external request and combination of one or more spans.
- *Span*: Represents a request to internal services with properties: operation, its attributes, start timestamp, and end timestamp.

Metrics Pattern

The metrics system collects metrics that represent the critical information about every component of the system.

Primarily there are two categories of the metrics

- *Infrastructural level metrics*: CPU, memory, and disk utilization.
- *Application-level metrics:* Number of requests, latency, etc.

Service reports metrics to a central server that provides aggregation, visualization, and alerting.

A service exposes an endpoint for metrics, for eg. in the spring boot framework, you can include a micrometer metrics library in your application that exposes metrics endpoint.

Deliver metrics to the metrics service

A service delivers metrics to the metrics service in two ways: Push or Pull.

Push model

Service instance sends the metrics to the metrics service. Eg. AWS cloud watch is an example of a push model.

Pull model

Metrics service (or an agent running locally) invokes a service API to retrieve the metrics from the service instance. Eg. Prometheus, a popular framework for monitoring and alerting uses a pull model.

Metric Sample

A metric sample has the following properties

- Name
- Value
- Timestamp
- Dimension

Exception Tracking pattern

Whenever a service reports an exception, you should identify the root cause.

The traditional way of looking for the exception is to look into logs.

There are few problems with the traditional approach:

- Logs are intended for single line entries but the exception consists of multiple lines.

- No mechanism to avoid duplicate errors which are confusing most of the time.

A better approach is to use an exception tracking service

Exception tracking

Services report exceptions to the central service that de-duplicates exceptions, generate alerts, and manage the resolution of the exceptions.

Audit logging pattern

Audit logging is important to track the user's activity.

Each audit log identify

- Un usuario que realizó la operación.
- Una operación que se está realizando.
- Una entidad comercial en la que se realiza la operación.

Las auditorías son importantes para garantizar una mejor atención al cliente, cumplimiento en el sistema y detectar comportamientos sospechosos en el sistema.

A continuación se muestran diferentes formas de implementar el registro de auditoría

- Utilice la programación orientada a aspectos
- Abastecimiento de eventos

Microservicios Sistemas distribuidos Desarrollo de software Ingeniería de software

Arquitectura de software