Observability Patterns

MOSTAFA RASTGAR

AUGUST 2019

Agenda

- Observability Patterns
  - Log Aggregation
  - Performance Metrics
  - Distributed Tracing
  - Health Check
- Demo

Log Aggregation

- Problem
  - Consider a use case where an application consists of multiple service instances that are running on multiple machines. Requests often span multiple service instances. Each service instance generates a log file in a standardized format. How can we understand the application behavior through logs for a particular request?
- Solution
  - We need a centralized logging service that aggregates logs from each service instance. Users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs. For example, PCF does have Loggeregator, which collects logs from each component (router, controller, diego, etc...) of the PCF platform along with applications. AWS Cloud Watch also does the same.
  - Kafa + ELK could be a general best-parctice solution

Performance Metrics

- Problem
  - When the service portfolio increases due to microservice architecture, it becomes critical to keep a watch on the transactions so that patterns can be monitored and alerts sent when an issue happens. How should we collect metrics to monitor application perfomance?
- Solution
  - A metrics service is required to gather statistics about individual operations. It should aggregate the metrics of an application service, which provides reporting and alerting. There are two models for aggregating metrics:
    - Push — the service pushes metrics to the metrics service e.g. NewRelic, AppDynamics
    - Pull — the metrics services pulls metrics from the service e.g. Prometheus + Spring Boot Actuator

Distributed Tracing

- Problem
  - In microservice architecture, requests often span multiple services. Each service handles a request by performing one or more operations across multiple services. Then, how do we trace a request end-to-end to troubleshoot the problem?
- Solution
  - We need a service which
    - Assigns each external request a unique external request id.
    - Passes the external request id to all services.
    - Includes the external request id in all log messages.
    - Records information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service.
  - Spring Cloud Slueth, along with Zipkin server, is a common implementation.

Health Check

- Problem
  - When microservice architecture has been implemented, there is a chance that a service might be up but not able to handle transactions. In that case, how do you ensure a request doesn't go to those failed instances? With a load balancing pattern implementation.
- Solution
  - Each service needs to have an endpoint which can be used to check the health of the application, such as /health. This API should check the status of the host, the connection to other services/infrastructure, and any specific logic.
  - Spring Boot Actuator does implement a /health endpoint and the implementation can be customized, as well.

Review Circuit breaker in Cross-Cutting Concern Patterns category

- Problem:
  - What will happen if downstream service is down?
    - First, the request will keep going to the down service, exhausting network resources and slowing performance
    - Second, How do we avoid cascading service failures and handle failures gracefully
- Solution:
  - The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker
  - When the number of consecutive failures crosses a threshold, the circuit breaker trips
  - After the timeout expires the circuit breaker allows a limited number of test requests to pass through
  - If those requests succeed, the circuit breaker resumes normal operation
- Netflix Hystrix is a good implementation of the circuit breaker pattern. It also helps you to define a fallback mechanism which can be used when the circuit breaker trips

Demo-Actuator

- Actuator
  - In essence, Actuator brings production-ready features to our application
  - Monitoring our app, gathering metrics, understanding traffic or the state of our database becomes trivial with this dependency.
  - Endpoints
    - **/health:** Shows application health information
    - **/info:** Displays arbitrary application info; not sensitive by default
      - info.app.name=Application Name
      - info.app.description=Application Description
      - info.app.version=Application Version
    - **/metrics:** Shows 'metrics' information for the current application; it's also sensitive by default
    - **/trace:** Displays trace information (by default the last few HTTP requests)

Demo-Circuit Breaker

- Hystrix
  - Fallback
  - Properties
    - **metrics.rollingStats.timeInMilliseconds:** This property sets the duration of the statistical rolling window, in milliseconds
    - **circuitBreaker.requestVolumeThreshold:** This property sets the minimum number of requests in a rolling window that will trip the circuit
    - **circuitBreaker.errorThresholdPercentage:** This property sets the error percentage at or above which the circuit should trip open and start short-circuiting requests to fallback logic
    - **The conclusion of two above items:** within a timespan of duration metrics.rollingStats.timeInMilliseconds, the percentage of actions resulting in a handled exception exceeds errorThresholdPercentage, provided also that the number of actions through the circuit in the timespan is at least requestVolumeThreshold
    - **circuitBreaker.sleepWindowInMilliseconds:** This property sets the amount of time, after tripping the circuit, to reject requests before allowing attempts again to determine if the circuit should again be closed
    - For more detail refer https://github.com/Netflix/Hystrix/wiki/Configuration#circuitBreaker.sleepWindowInMilliseconds
- Hystrix Dashboard
  - First, enable actuator
  - Set http://[your host]:[your port]/actuator/hystrix.stream for the dashboard and monitor your application circuit

Demo- Zipkin

- Zipkin
  - You can trace your request among the microservices
  - Zipkin owes credit to its original founders, Twitter, and the Google Dapper paper on which it was based. In the last year Zipkin left the nest (pun intended) and is now an organisation called OpenZipkin. OpenZipkin has had a lot of recent work from Uber, MdSol, SoundCloud, Yelp, Prezi, Firebase, …
- Sleuth
  - Spring Cloud Sleuth implements a distributed tracing solution for Spring Cloud, borrowing heavily from Dapper, Zipkin and Htrace
  - For most users Sleuth should be invisible, and all your interactions with external systems should be instrumented automatically
  - For more detail refer https://spring.io/projects/spring-cloud-sleuth