Pattern: Microservice chassis

Context

When you start the development of an application you often spend a significant amount of time putting in place the mechanisms to handle cross-cutting concerns. Examples of cross-cutting concern include:

- Externalized configuration - includes credentials, and network locations of external services such as databases and message brokers
- Logging - configuring of a logging framework such as log4j or logback
- Health checks - a url that a monitoring service can “ping” to determine the health of the application
- Metrics - measurements that provide insight into what the application is doing and how it is performing
- Distributed tracing (observability/distributed-tracing.html) - instrument services with code that assigns each external request an unique identifier that is passed between services.

As well as these generic cross-cutting concerns, there are also cross-cutting concerns that are specific to the technologies that an application uses. Applications that use infrastructure services such as databases or a message brokers require boilerplate configuration in order to do that. For example, applications that use a relational database must be configured with a connection pool. Web applications that process HTTP requests also need boilerplate configuration.

It is common to spend one or two days, sometimes even longer, setting up these mechanisms. If you going to spend months or years developing a monolithic application then the upfront investment in handling cross-cutting concerns is insignificant. The situation is very different, however, if you are developing an application that has the microservice architecture (microservices.html). There are tens or hundreds of services. You will frequently create new services, each of which will only take days or weeks to develop. You cannot afford to spend a few days configuring the mechanisms to handle cross-cutting concerns. What is even worse is that in a microservice architecture there are additional cross-cutting concerns that you have to deal with including service registration and discovery, and circuit breakers for reliably handling partial failure.

Forces

- Creating a new microservice should be fast and easy
- When creating a microservice you must handle cross-cutting concerns such as externalized configuration, logging, health checks, metrics, service registration and discovery, circuit breakers. There are also cross-cutting concerns that are specific to the technologies that the microservices uses.

Solution

Build your microservices using a microservice chassis framework, which handles cross-cutting concerns

Example

Examples of microservice chassis frameworks:

- Java
  - Spring Boot (http://projects.spring.io/spring-boot/) and Spring Cloud (http://cloud.spring.io/)
  - Dropwizard (https://dropwizard.github.io/)
- Go
  - Gizmo (http://open.blogs.nytimes.com/2015/12/17/introducing-gizmo/?_r=2)
  - Micro (https://github.com/micro)

    - Go kit (https://github.com/go-kit/kit)

Resulting context

The major benefit of a microservice chassis is that you can quickly and easy get started with developing a microservice.

You need a microservice chassis for each programming language/framework that you want to use. This can be an obstacle to adopting a new programming language or framework.

Related patterns

There are the following related patterns:

- Microservices (microservices.html) - this pattern motivates the need for the Microservice Chassis pattern
- Self Registration (self-registration.html) - the microservice chassis is often responsible for registering the service with the service registry
- Client-side discovery (client-side-discovery.html) - the microservice chassis is often responsible for client-side service discovery
- Circuit Breaker (reliability/circuit-breaker.html) - the microservice chassis framework might implement this pattern
- Distributed tracing (observability/distributed-tracing.html) - the microservice chassis framework might instrument the code

Tweet

Follow @MicroSvcArch