Problems and Solutions

Logging In MicroServices:

How to achieve centralized logging system?

Ans: There is a very famous approach called ELK:

- **Elasticsearch**: Search and analyse data in real time
- **Logstash**: Collect, parse and enrich data from each machine
- **Kibana**: Explore and visualise your data graphically
- **AWS Cloud Watch**

So all the information is collected by Logstash, stored in Elasticsearch and visualised using Kibana UI.

With this stack no matters who is generating the information (or what technology is using).

USECASE:

Let's imagine you are building an online store that uses the Microservice architecture pattern and that you are implementing the product details page. You need to develop multiple versions of the product details user interface:

- HTML5/JavaScript-based UI for desktop and mobile browsers - HTML is generated by a server-side web application
- Native Android and iPhone clients - these clients interact with the server via REST APIs

In addition, the online store must expose product details via a REST API for use by 3rd party applications.

A product details UI can display a lot of information about a product. For example, the Amazon.com details page for POJOs in Action displays:

- Basic information about the book such as title, author, price, etc.
- Your purchase history for the book
- Availability
- Buying options
- Other items that are frequently bought with this book
- Other items bought by customers who bought this book
- Customer reviews
- Sellers ranking
- …

Since the online store uses the Microservice architecture pattern the product details data is spread over multiple services. For example,

- Product Info Service - basic information about the product such as title, author
- Pricing Service - product price
- Order service - purchase history for product
- Inventory service - product availability
- Review service - customer reviews …

Consequently, the code that displays the product details needs to fetch information from all of these services.

Problem

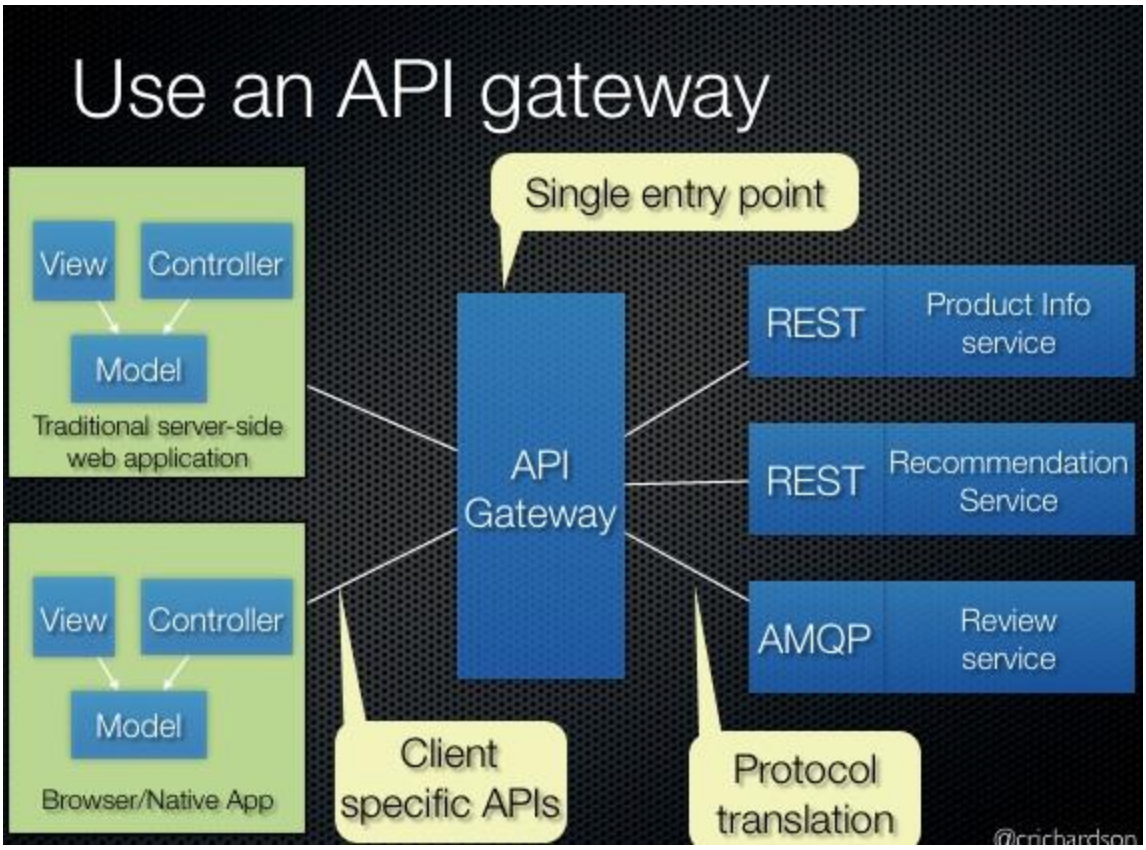
How do the clients of a Microservices-based application access the individual services?

Forces

- The granularity of APIs provided by microservices is often different than what a client needs. Microservices typically provide fine-grained APIs, which means that clients need to interact with multiple services. For example, as described above, a client needing the details for a product needs to fetch data from numerous services.
- Different clients need different data. For example, the desktop browser version of a product details page desktop is typically more elaborate then the mobile version.
- Network performance is different for different types of clients. For example, a mobile network is typically much slower and has much higher latency than a non-mobile network. And, of course, any WAN is much slower than a LAN. This means that a native mobile client uses a network that has very difference performance characteristics than a LAN used by a server-side web application. The server-side web application can make multiple requests to backend services without impacting the user experience where as a mobile client can only make a few.
- The number of service instances and their locations (host+port) changes dynamically
- Partitioning into services can change over time and should be hidden from clients
- Services might use a diverse set of protocols, some of which might not be web friendly

Solution

Implement an API gateway that is the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services.



Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client. For example, the Netflix API gateway runs client-specific adapter code that provides each client with an API that's best suited to its requirements.

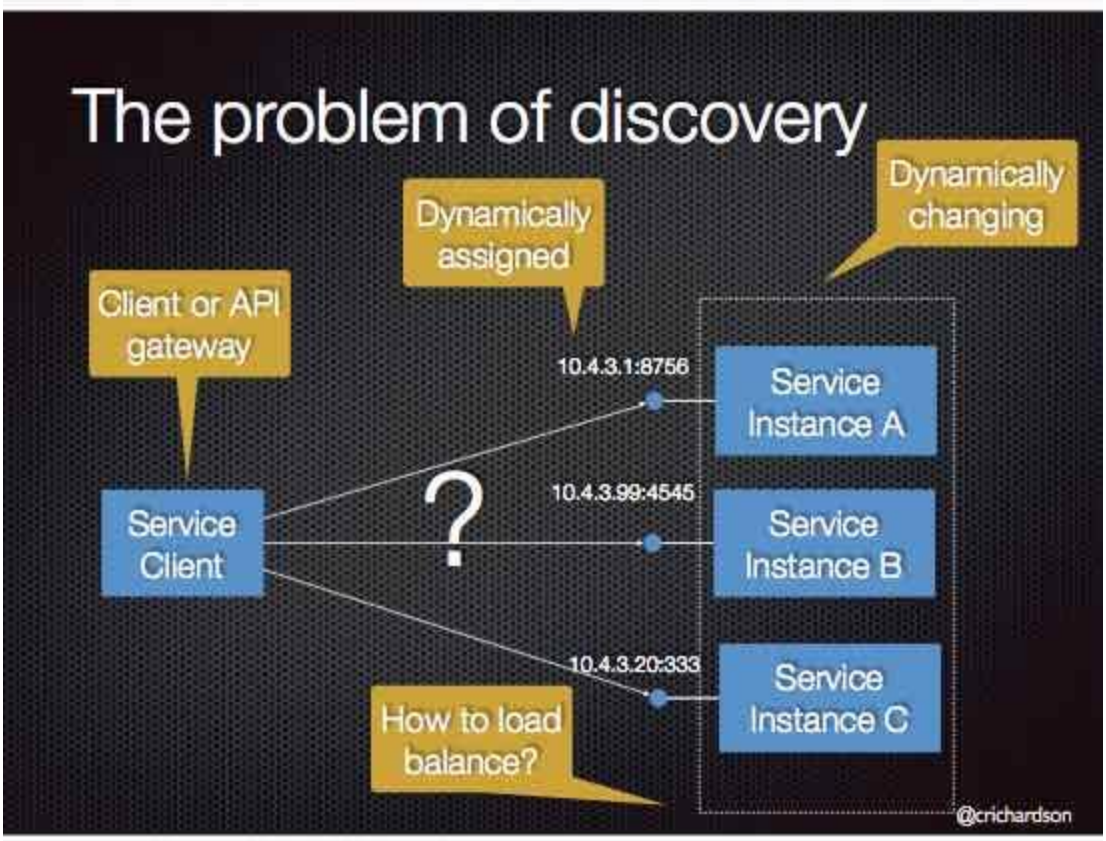
The API gateway might also implement security, e.g. verify that the client is authorized to perform the request

USECASE 2:

Client-side service discovery

Context

Services typically need to call one another. In a monolithic application, services invoke one another through language-level method or procedure calls. In a traditional distributed system deployment, services run at fixed, well known locations (hosts and ports) and so can easily call one another using HTTP/REST or some RPC mechanism. However, a modern microservice-based application typically runs in a virtualized or containerized environments where the number of instances of a service and their locations changes dynamically.



Consequently, you must implement a mechanism for that enables the clients of service to make requests to a dynamically changing set of ephemeral service instances.

Problem

How does the client of a service - the API gateway or another service - discover the location of a service instance?

Forces

Each instance of a service exposes a remote API such as HTTP/REST, or Thrift etc. at a particular location (host and port)

The number of services instances and their locations changes dynamically.

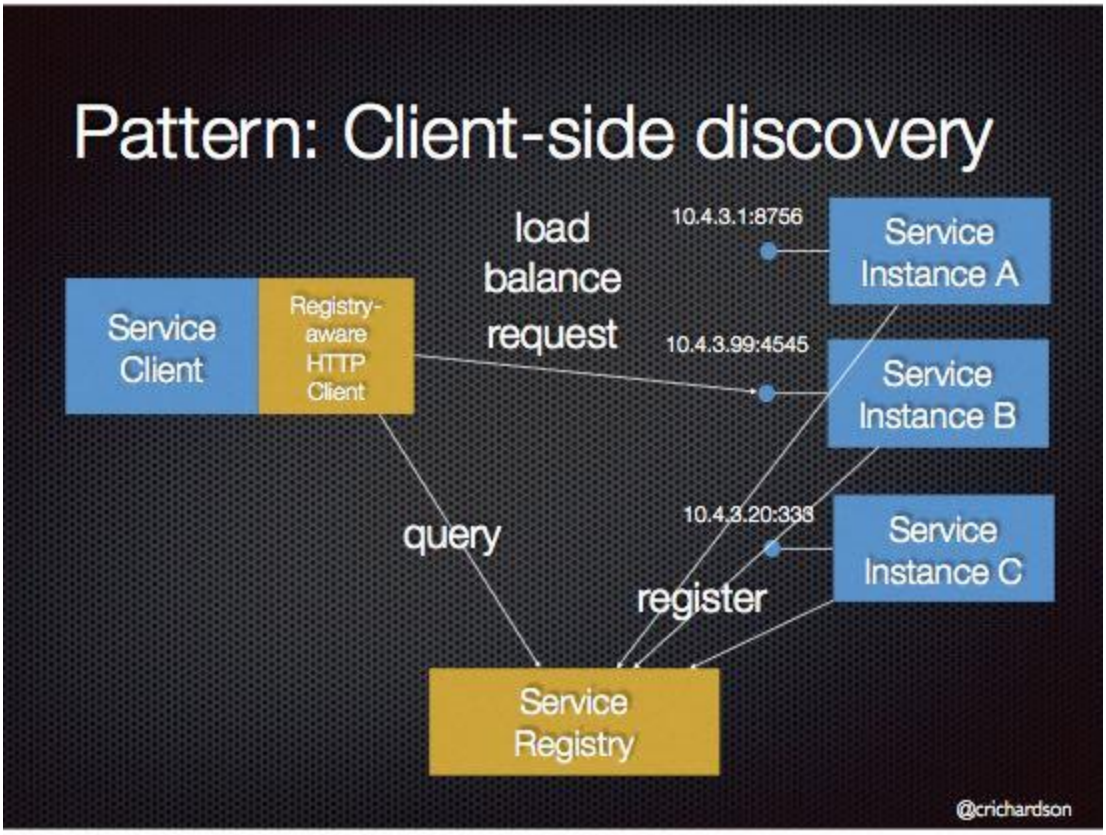
Virtual machines and containers are usually assigned dynamic IP addresses.

The number of services instances might vary dynamically. For example, an EC2 Autoscaling Group adjusts the number of instances based on load.

Solution

When making a request to a service, the client obtains the location of a service instance by querying a Service Registry, which knows the locations of all service instances.

The following diagram shows the structure of this pattern.



Context

You have applied the Microservice architecture pattern. The application consists of numerous services. Services often invoke other services. You must write automated tests that verify that a service behaves correctly.

Problem

How do you easily test a service?

Forces

End to end testing (i.e. tests that launch multiple services) is difficult, slow, brittle, and expensive.

Solution

A test suite that tests a service in isolation using test doubles for any services that it invokes.

Examples

Spring Cloud Contract is an open source project that supports this style of testing.

Resulting context

This pattern has the following benefits:

Testing a service in isolation is easier, faster, more reliable and cheap

This pattern has the following drawbacks:

Tests might pass but the application will fail in production

Where Zuul fits in microservices ecosystem?

A common problem, when building microservices, is to provide a unique gateway to the client applications of your system. The fact that your services are split into small microservcies apps that shouldn't be visible to users otherwise it may result in substantial development/maintenance efforts. Also there are scenarios when whole ecosystem network traffic may be passing through a single point which could impact the performance of the cluster.

To solve this problem, Netflix (a major adopter of microservices) created and open-sourced its **Zuul proxy server** and later Spring under Pivotal has adapted this in its **spring cloud stack** and enabled us to use zuul easily and effectively with just few simple steps.

Problem

SpringBoot provides lot of flexibility in externalizing configuration properties via properties or YAML files. We can also configure properties for each environment (dev, qa, prod etc) separately using profile specific configuration files such as **application.properties**, **application-dev.properties**, **application-prod.properties** etc. But once the application is started we can not update the properties at runtime. If we change the properties we need to restart the application to use the updated configuration properties.

Also, in the context of large number of MicroService based applications, we want the ability to configure and manage the configuration properties of all MicroServices from a centralized place.

UseCase:

Context

You have applied the Microservice architecture pattern. Sometimes a service instance can be incapable of handling requests yet still be running. For example, it might have ran out of database connections. When this occurs, the monitoring system should generate a alert. Also, the load balancer or service registry should not route requests to the failed service instance.

Problem

How to detect that a running service instance is unable to handle requests?

Forces

An alert should be generated when a service instance fails

Requests should be routed to working service instances

Solution

A service has an health check API endpoint (e.g. HTTP /health) that returns the health of the service. The API endpoint handler performs various checks, such as

the status of the connections to the infrastructure services used by the service instance

the status of the host, e.g. disk space

application specific logic

A health check client - a monitoring service, service registry or load balancer - periodically invokes the endpoint to check the health of the service instance.

Usecases:

Context

You have applied the Microservice architecture pattern. The application consists of multiple services and service instances that are running on multiple machines. Errors sometimes occur when handling requests. When an error occurs, a service instance throws an exception, which contains an error message and a stack trace.

Problem

How to understand the behavior of an application and troubleshoot problems?

Forces

- Exceptions must be de-duplicated, recorded, investigated by developers and the underlying issue resolved
- Any solution should have minimal runtime overhead

Solution

Report all exceptions to a centralized exception tracking service that aggregates and tracks exceptions and notifies developers.

Resulting Context

This pattern has the following benefits:

- It is easier to view exceptions and track their resolution

This pattern has the following drawbacks:

- The exception tracking service is additional infrastructure

Context

You have applied the Microservice architecture pattern.

Problem

How to understand the behavior of an application and troubleshoot problems?

Forces

Any solution should have minimal runtime overhead

Solution

Instrument a service to gather statistics about individual operations. Aggregate metrics in centralized metrics service, which provides reporting and alerting. There are two models for aggregating metrics:

push - the service pushes metrics to the metrics service

pull - the metrics services pulls metrics from the service

Examples

Instrumentation libraries:

Coda Hale/Yammer Java Metrics Library

Prometheus client libraries

Metrics aggregation services

Prometheus

AWS Cloud Watch

Resulting context

- This pattern has the following benefits:
- It provides deep insight into application behavior
- This pattern has the following drawbacks:
- Metrics code is intertwined with business logic making it more complicated
- This pattern has the following issues:
- Aggregating metrics can require significant infrastructure