**Microservices using Spring**

* **Monolithic Architecture**

***Context***

You are developing a server-side enterprise application. It must support a variety of different clients including desktop browsers, mobile browsers and native mobile applications. The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either web services or a message broker. The application handles requests (HTTP requests and messages) by executing business logic; accessing a database; exchanging messages with other systems; and returning a HTML/JSON/XML response. There are logical components corresponding to different functional areas of the application.

***Problem***

What’s the application’s deployment architecture?

***Forces***

1. There is a team of developers working on the application
2. New team members must quickly become productive
3. The application must be easy to understand and modify
4. You want to practice continuous deployment of the application
5. You must run multiple instances of the application on multiple machines in order to satisfy scalability and availability requirements
6. You want to take advantage of emerging technologies (frameworks, programming languages, etc.)

***Solution***

Build an application with a monolithic architecture. This solution has a number of benefits:

1. Simple to develop - the goal of current development tools and IDEs is to support the development of monolithic applications
2. Simple to deploy - you simply need to deploy the WAR file (or directory hierarchy) on the appropriate runtime
3. Simple to scale - you can scale the application by running multiple copies of the application behind a load balancer

However, once the application becomes large and the team grows in size, this approach has a number of drawbacks that become increasingly significant. The problems occur when a single monolithic application tries to serve purposes toward conglomeration of businesses. The following list explain some of the common problems in detail.

* The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. Also, because there are not hard module boundaries, modularity breaks down over time. Moreover, because it can be difficult to understand how to correctly implement a change the quality of the code declines over time. It’s a downwards spiral.
* **Overloaded IDE** - the larger the code base the slower the IDE and the less productive developers are.
* **Overloaded web container** - the larger the application the longer it takes to start up. This had have a huge impact on developer productivity because of time wasted waiting for the container to start. It also impacts deployment too.
* **Continuous deployment is difficult** - a large monolithic application is also an obstacle to frequent deployments. In order to update one component you have to redeploy the entire application. This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems. There is also the chance that components that haven’t been updated will fail to start correctly. As a result, the risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers, since they usually need to iterative rapidly and redeploy frequently.
* **Scaling the application can be difficult** - a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application. Some clouds can even adjust the number of instances dynamically based on load. But on the other hand, this architecture can’t scale with an increasing data volume. Each copy of application instance will access all of the data, which makes caching less effective and increases memory consumption and I/O traffic. Also, different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently
* **Obstacle to scaling development** - A monolithic application is also an obstacle to scaling development. Once the application gets to a certain size its useful to divide up the engineering organization into teams that focus on specific functional areas. For example, we might want to have the UI team, accounting team, inventory team, etc. The trouble with a monolithic application is that it prevents the teams from working independently. The teams must coordinate their development efforts and redeployments. It is much more difficult for a team to make a change and update production.
* **Requires a long-term commitment to a technology stack** - a monolithic architecture forces you to be married to the technology stack (and in some cases, to a particular version of that technology) you chose at the start of development . With a monolithic application, can be difficult to incrementally adopt a newer technology. For example, let’s imagine that you chose the JVM. You have some language choices since as well as Java you can use other JVM languages that inter-operate nicely with Java such as Groovy and Scala. But components written in non-JVM languages do not have a place within your monolithic architecture. Also, if your application uses a platform framework that subsequently becomes obsolete then it can be challenging to incrementally migrate the application to a newer and better framework. It’s possible that in order to adopt a newer platform framework you have to rewrite the entire application, which is a risky undertaking.
* **Microservice Architecture**

***Context***

You are developing a server-side enterprise application. It must support a variety of different clients including desktop browsers, mobile browsers and native mobile applications. The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either web services or a message broker. The application handles requests (HTTP requests and messages) by executing business logic; accessing a database; exchanging messages with other systems; and returning a HTML/JSON/XML response. There are logical components corresponding to different functional areas of the application.

***Problem***

What’s the application’s deployment architecture?

***Forces***

* There is a team of developers working on the application
* New team members must quickly become productive
* The application must be easy to understand and modify
* You want to practice continuous deployment of the application
* You must run multiple instances of the application on multiple machines in order to satisfy scalability and availability requirements
* You want to take advantage of emerging technologies (frameworks, programming languages, etc)

***Solution***

Define an architecture that structures the application as a set of loosely coupled, collaborating services. Think of each service as a separate / partially separate application. Each service is:

* Highly maintainable and testable - enables rapid and frequent development and deployment
* Loosely coupled with other services - enables a team to work independently the majority of time on their service(s) without being impacted by changes to other services and without affecting other services
* Independently deployable - enables a team to deploy their service without having to coordinate with other teams
* Capable of being developed by a small team - essential for high productivity by avoiding the high communication head of large teams

Services communicate using either synchronous protocols such as HTTP/REST or asynchronous protocols such as AMQP. Services can be developed and deployed independently of one another. Each service has its own database in order to be decoupled from other services.



*This solution has a number of benefits:*

* Enables the continuous delivery and deployment of large, complex applications.
  + Improved maintainability - each service is relatively small and so is easier to understand and change
  + Better testability - services are smaller and faster to test
  + Better deployability - services can be deployed independently
  + It enables you to organize the development effort around multiple, autonomous teams. Each (so called two pizza) team owns and is responsible for one or more services. Each team can develop, test, deploy and scale their services independently of all of the other teams.
* Each microservice is relatively small:
  + Easier for a developer to understand
  + The IDE is faster making developers more productive
  + The application starts faster, which makes developers more productive, and speeds up deployments
* Improved fault isolation. For example, if there is a memory leak in one service then only that service will be affected. The other services will continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.
* Eliminates any long-term commitment to a technology stack. When developing a new service you can pick a new technology stack. Similarly, when making major changes to an existing service you can rewrite it using a new technology stack.

***Drawbacks***

This solution has a number of drawbacks:

* Developers must deal with the additional complexity of creating a distributed system
  + Developers must implement the inter-service communication mechanism and deal with partial failure
  + Implementing requests that span multiple services is more difficult
  + Testing the interactions between services is more difficult
  + Implementing requests that span multiple services requires careful coordination between the teams
  + Developer tools/IDEs are oriented on building monolithic applications and don’t provide explicit support for developing distributed applications.
* Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different services.
* Increased memory consumption. The microservice architecture replaces N monolithic application instances with NxM services instances. If each service runs in its own JVM (or equivalent), which is usually necessary to isolate the instances, then there is the overhead of M times as many JVM runtimes. Moreover, if each service runs on its own VM (e.g. EC2 instance), as is the case at Netflix, the overhead is even higher.

***Issues***

**When to use the microservice architecture?**

One challenge with using this approach is deciding when it makes sense to use it. When developing the first version of an application, you often do not have the problems that this approach solves. Moreover, using an elaborate, distributed architecture will slow down development. This can be a major problem for start-ups whose biggest challenge is often how to rapidly evolve the business model and accompanying application. Using Y-axis splits might make it much more difficult to iterate rapidly. Later on, however, when the challenge is how to scale and you need to use functional decomposition, the tangled dependencies might make it difficult to decompose your monolithic application into a set of services.

How to decompose the application into services?

Another challenge is deciding how to partition the system into microservices. This is very much an art, but there are a number of strategies that can help:

* Decompose by business capability and define services corresponding to business capabilities.
* Decompose by domain-driven design subdomain.
* Decompose by verb or use case and define services that are responsible for particular actions. e.g. a Shipping Service that’s responsible for shipping complete orders.
* Decompose by nouns or resources by defining a service that is responsible for all operations on entities/resources of a given type. e.g. an Account Service that is responsible for managing user accounts.

Ideally, each service should have only a small set of responsibilities. (Uncle) Bob Martin talks about designing classes using the Single Responsibility Principle (SRP). The SRP defines a responsibility of a class as a reason to change, and states that a class should only have one reason to change. It make sense to apply the SRP to service design as well.

Another analogy that helps with service design is the design of Unix utilities. Unix provides a large number of utilities such as grep, cat and find. Each utility does exactly one thing, often exceptionally well, and can be combined with other utilities using a shell script to perform complex tasks.

**How to maintain data consistency?**

In order to ensure loose coupling, each service has its own database. Maintaining data consistency between services is a challenge because 2 phase-commit/distributed transactions is not an option for many applications. An application must instead use the Saga pattern. A service publishes an event when its data changes. Other services consume that event and update their data. There are several ways of reliably updating data and publishing events including Event Sourcing and Transaction Log Tailing.

**How to implement queries?**

Another challenge is implementing queries that need to retrieve data owned by multiple services. There are several other patterns that need to be adopted and implemented in order to implement service specific querying.

* **Known uses**
* Most large scale web sites including Netflix, Amazon and eBay have evolved from a monolithic architecture to a microservice architecture.
* Netflix, which is a very popular video streaming service that’s responsible for up to 30% of Internet traffic, has a large scale, service-oriented architecture. They handle over a billion calls per day to their video streaming API from over 800 different kinds of devices. Each API call fans out to an average of six calls to backend services.
* Amazon.com originally had a two-tier architecture. In order to scale they migrated to a service-oriented architecture consisting of hundreds of backend services. Several applications call these services including the applications that implement the Amazon.com website and the web service API. The Amazon.com website application calls 100-150 services to get the data that used to build a web page.
* The auction site ebay.com also evolved from a monolithic architecture to a service-oriented architecture. The application tier consists of multiple independent applications. Each application implements the business logic for a specific function area such as buying or selling. Each application uses X-axis splits and some applications such as search use Z-axis splits. Ebay.com also applies a combination of X-, Y- and Z-style scaling to the database tier.
* There are numerous other examples of companies using the microservice architecture.

***Here is a list of companies using microservices:***

1. Comcast Cable
2. Uber
3. Netflix
4. Amazon
5. Ebay
6. Sound Cloud
7. Karma
8. Groupon
9. Hailo
10. Gilt
11. Zalando
12. Capital One
13. Lending Club
14. AutoScout24

**(fig. Multiple Patterns used in Micro Services)**

* **Netflix Zuul**

Zuul is the front door for all requests from devices and web sites to the backend of the Netflix streaming application. As an edge service application, Zuul is built to enable dynamic routing, monitoring, resiliency and security. It also has the ability to route requests to multiple Amazon Auto Scaling Groups as appropriate.

**Why Zuul?**

The volume and diversity of API traffic sometimes results in production issues arising quickly and without warning. We need a system that allows us to rapidly change behaviour in order to react to these situations.

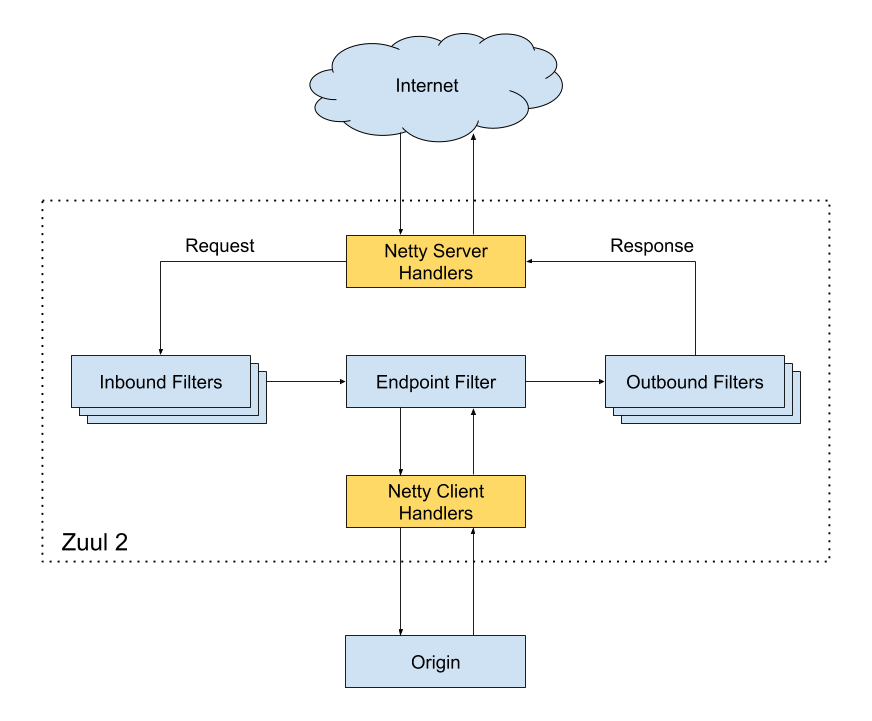
Zuul uses a range of different types of filters that enables us to quickly and nimbly apply functionality to our edge service. These filters help us perform the following functions:

* Authentication and Security - identifying authentication requirements for each resource and rejecting requests that do not satisfy them.
* Insights and Monitoring - tracking meaningful data and statistics at the edge in order to give us an accurate view of production.
* Dynamic Routing - dynamically routing requests to different backend clusters as needed.
* Stress Testing - gradually increasing the traffic to a cluster in order to gauge performance.
* Load Shedding - allocating capacity for each type of request and dropping requests that go over the limit.
* Static Response handling - building some responses directly at the edge instead of forwarding them to an internal cluster
* Multiregional Resiliency - routing requests across AWS regions in order to diversify our ELB usage and move our edge closer to our members

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 microservices 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 created and open-sourced its Zuul proxy server and later Spring under Pivotal has adapted this in its spring cloud stack.

Zuul is an edge service that proxies requests to multiple backing services. It provides a unified “front door” to your ecosystem, which allows any browser, mobile app or other user interface to consume services from multiple hosts. You can integrate Zuul with other Netflix stack components like Hystrix for fault tolerance and Eureka for service discovery or use it to manage routing rules, filters and load balancing across your system. Most importantly all of those components are well adapted by spring framework through spring boot/cloud approach.



**Dependencies**

<dependency>

<groupId>com.netflix.zuul</groupId>

<artifactId>zuul-core</artifactId>

<version>2.1.2</version>

</dependency>

Git: <https://github.com/jaydeepresources/Tibco/tree/microservices-zuul>

* **Netflix Eureka**

Eureka is a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers.

At Netflix, Eureka is used for the following purposes apart from playing a critical part in mid-tier load balancing.

* For aiding Netflix Asgard - an open source service which makes cloud deployments easier, in
  + Fast rollback of versions in case of problems avoiding the re-launch of 100's of instances which could take a long time.
  + In rolling pushes, for avoiding propagation of a new version to all instances in case of problems.
* For our cassandra deployments to take instances out of traffic for maintenance.
* For our memcached caching services to identify the list of nodes in the ring.
* For carrying other additional application specific metadata about services for various other reasons.

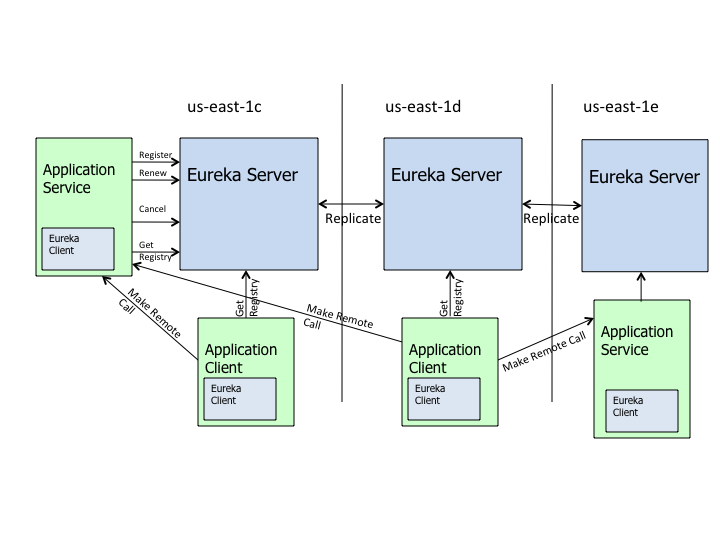
**What is the need for Eureka?**

Eureka is a REST (Representational State Transfer) based service that is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers. We call this service, the **Eureka Server**. Eureka also comes with a Java-based client component, the **Eureka Client**, which makes interactions with the service much easier. The client also has a built-in load balancer that does basic round-robin load balancing. At Netflix, a much more sophisticated load balancer wraps Eureka to provide weighted load balancing based on several factors like traffic, resource usage, error conditions etc to provide superior resiliency.

n AWS cloud, because of its inherent nature, servers come and go. Unlike the traditional load balancers which work with servers with well known IP addresses and host names, in AWS, load balancing requires much more sophistication in registering and de-registering servers with load balancer on the fly. Since AWS does not yet provide a middle tier load balancer, Eureka fills a big gap in the area of mid-tier load balancing.

You typically run in the AWS cloud and you have a host of middle tier services which you do not want to register with AWS ELB or expose traffic from outside world. You are either looking for a simple round-robin load balancing solution or are willing to write your own wrapper around Eureka based on your load balancing need. You do not have the need for sticky sessions and load session data in an external cache such as memcached. More importantly, if your architecture fits the model where a client based load balancer is favored, Eureka is well positioned to fit that usage.

The easiest way to configure Eureka client is by using property files. By default, Eureka client searches for the property file *eureka-client.properties* in the *classpath*. It further searches for environment specific overrides in the environment specific properties files. The environment is typically *test* or *prod* and is supplied by a *-Deureka.environment* java commandline switch to the eureka client (without the *.properties* suffix). Accordingly, the client also searches for *eureka-client-{test,prod}.properties.*



**Dependencies**

<dependency>

<groupId>com.netflix.eureka</groupId>

<artifactId>eureka-client</artifactId>

</dependency>

Git: <https://github.com/jaydeepresources/Tibco/tree/microservices-eureka>

* **Ribbon: Client side load balancing**

Ribbon is a client-side load balancer that gives you a lot of control over the behavior of HTTP and TCP clients. A central concept in Ribbon is that of the named client. Each load balancer is part of an ensemble of components that work together to contact a remote server on demand, and the ensemble has a name that you give it as an application developer. On demand, Spring Cloud creates a new ensemble as an ApplicationContext for each named client by using RibbonClientConfiguration. This contains (amongst other things) an ILoadBalancer, a RestClient, and a ServerListFilter. You can configure some bits of a Ribbon client by using external properties in <client>.ribbon.\*

This approach does mean that our Ribbon configuration will be part of the main application context and therefore shared by *all* Ribbon clients in the User application. In a normal application, you can avoid this by keeping Ribbon beans out of the main application context.

**Load Balancing**

Eventually you'll reach a point where you need to run multiple instances of an application or a service for high availability or to manage increased load. That's what load balancers are for. There's generally two different types:

1. Server-side load balancers
2. Client-side load balancers

**Server-side Load Balancing**

What many people would call a "load balancer" is actually a server-side load balancer. It can be implemented in hardware or software. The traffic is sent to a dedicated service that decides where to send the traffic, using an algorithm like round-robin, to one of the many instances. Examples of server-side load balancers are hardware-based ones like the devices from F5 Networks or software-based ones like AWS's ELBs.

**Client-side Load Balancing**

Many are familiar with what server-side load balancing is but the lesser known, client-side load balancing, has begun to climb in popularity due to SOA and microservices. Instead of relying on another service to distribute the load, the client itself, is responsible for deciding where to send the traffic also using an algorithm like round-robin. It can either discover the instances, via service discovery, or can be configured with a predefined list.

**Dependencies**

<dependency>

<groupId>com.netflix.ribbon</groupId>

<artifactId>ribbon</artifactId>

</dependency>

Git: <https://github.com/jaydeepresources/Tibco/tree/microservices-ribbon>

* **Hystrix: Fault Tolerance and Circuit Breaker**

Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable. In a distributed environment, inevitably some of the many service dependencies will fail. Hystrix is a library that helps you control the interactions between these distributed services by adding latency tolerance and fault tolerance logic. Hystrix does this by isolating points of access between the services, stopping cascading failures across them, and providing fallback options, all of which improve your system’s overall resiliency.

***Hystrix is designed to do the following:***

* Give protection from and control over latency and failure from dependencies accessed (typically over the network) via third-party client libraries.
* Stop cascading failures in a complex distributed system.
* Fail fast and rapidly recover.
* Fallback and gracefully degrade when possible.
* Enable near real-time monitoring, alerting, and operational control.

Applications in complex distributed architectures have dozens of dependencies, each of which will inevitably fail at some point. If the host application is not isolated from these external failures, it risks being taken down with them.

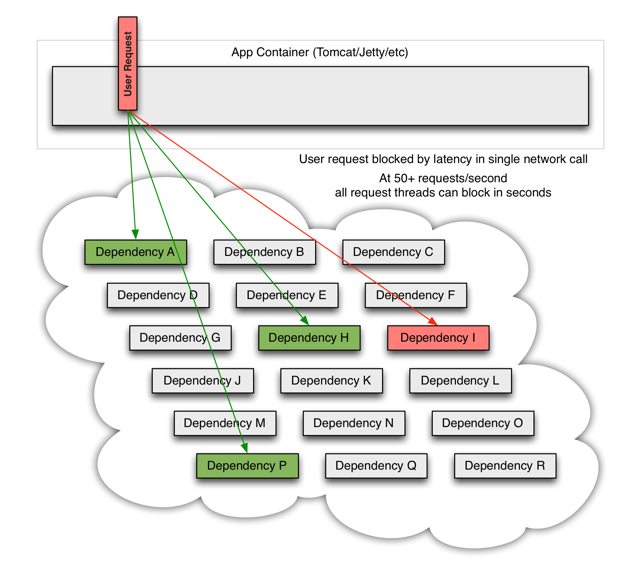
For example, for an application that depends on 30 services where each service has 99.99% uptime, here is what you can expect:

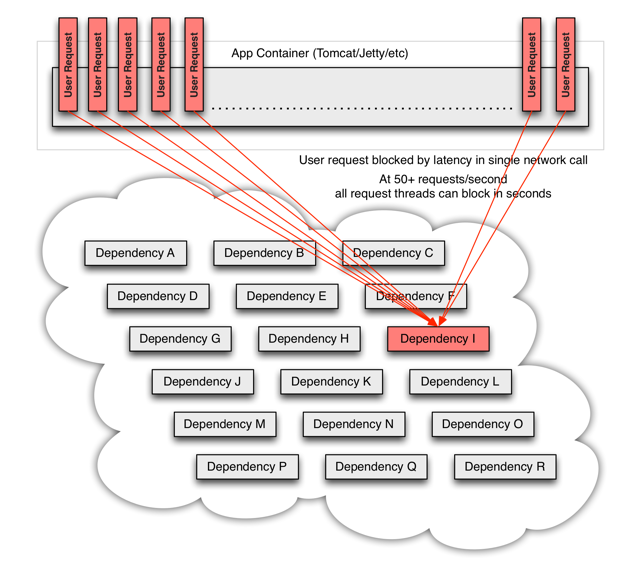
99.9930 = 99.7% uptime  
0.3% of 1 billion requests = 3,000,000 failures  
2+ hours downtime/month even if all dependencies have excellent uptime.

Reality is generally worse.

Even when all dependencies perform well the aggregate impact of even 0.01% downtime on each of dozens of services equates to potentially hours a month of downtime if you do not engineer the whole system for resilience.

With high volume traffic a single backend dependency becoming latent can cause all resources to become saturated in seconds on all servers. Every point in an application that reaches out over the network or into a client library that might result in network requests is a source of potential failure. Worse than failures, these applications can also result in increased latencies between services, which backs up queues, threads, and other system resources causing even more cascading failures across the system. These issues are exacerbated when network access is performed through a third-party client — a “black box” where implementation details are hidden and can change at any time, and network or resource configurations are different for each client library and often difficult to monitor and change.





Even worse are transitive dependencies that perform potentially expensive or fault-prone network calls without being explicitly invoked by the application. Network connections fail or degrade. Services and servers fail or become slow. New libraries or service deployments change behavior or performance characteristics. Client libraries have bugs. All of these represent failure and latency that needs to be isolated and managed so that a single failing dependency can’t take down an entire application or system.

*Hystrix works by:*

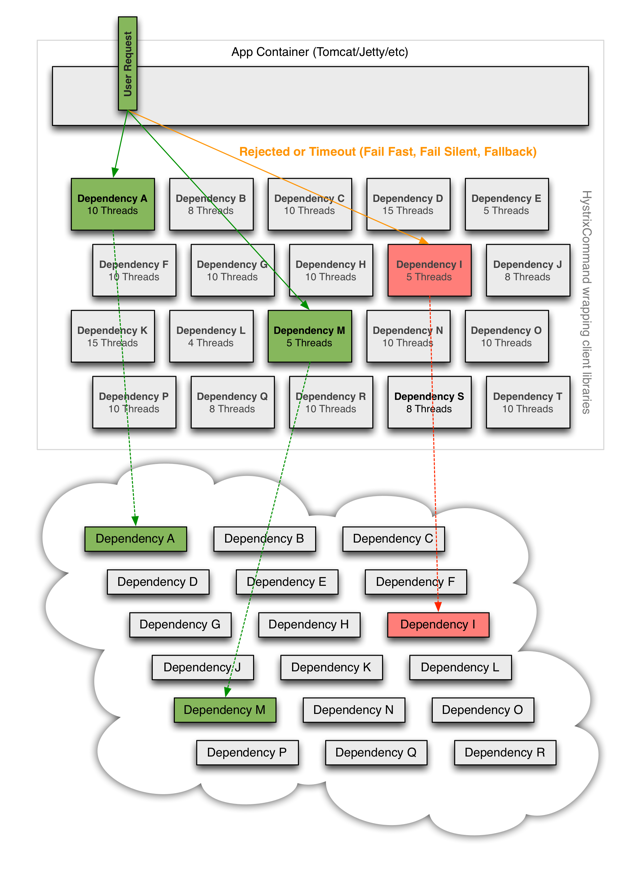
* Preventing any single dependency from using up all container (such as Tomcat) user threads.
* Shedding load and failing fast instead of queueing.
* Providing fallbacks wherever feasible to protect users from failure.
* Using isolation techniques (such as bulkhead, swimlane, and circuit breaker patterns) to limit the impact of any one dependency.
* Optimizing for time-to-discovery through near real-time metrics, monitoring, and alerting
* Optimizing for time-to-recovery by means of low latency propagation of configuration changes and support for dynamic property changes in most aspects of Hystrix, which allows you to make real-time operational modifications with low latency feedback loops.
* Protecting against failures in the entire dependency client execution, not just in the network traffic.

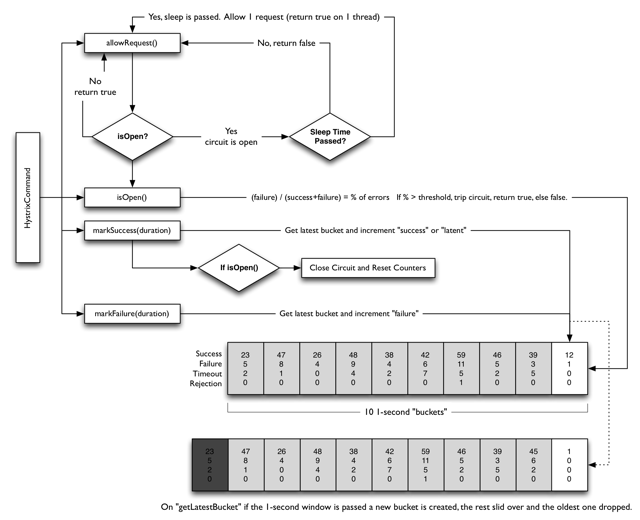
*Hystrix does this by:*

* Wrapping all calls to external systems (or “dependencies”) in a HystrixCommand or HystrixObservableCommand object which typically executes within a separate thread (this is an example of the command pattern).
* Timing-out calls that take longer than thresholds you define. There is a default, but for most dependencies you custom-set these timeouts by means of “properties” so that they are slightly higher than the measured 99.5th percentile performance for each dependency.
* Maintaining a small thread-pool (or semaphore) for each dependency; if it becomes full, requests destined for that dependency will be immediately rejected instead of queued up.
* Measuring successes, failures (exceptions thrown by client), timeouts, and thread rejections.
* Tripping a circuit-breaker to stop all requests to a particular service for a period of time, either manually or automatically if the error percentage for the service passes a threshold.
* Performing fallback logic when a request fails, is rejected, times-out, or short-circuits.
* Monitoring metrics and configuration changes in near real-time.

*Important Concepts:*

1. Circuit Breaker
2. Isolation
3. Threads & Thread Pools
4. Request Collapsing
5. Request Caching





Git: <https://github.com/jaydeepresources/Tibco/tree/microservices-hystrix>

* **Feign: Declarative REST Client**

Feign is a declarative web service client. It makes writing web service clients easier. To use Feign create an interface and annotate it. It has pluggable annotation support including Feign annotations and JAX-RS annotations. Feign also supports pluggable encoders and decoders. Spring Cloud adds support for Spring MVC annotations and for using the same HttpMessageConverters used by default in Spring Web. Spring Cloud integrates Ribbon and Eureka to provide a load balanced http client when using Feign.

@FeignClient(name = "foo-service", url = "http://foo.url.com/")

public interface PostsClient {

@GetMapping("/api")

public List<Foo> getFoo();

}

In the example above, we have an interface that declares that it will make GET requests to a foo.url.com and receive a List of Foo objects. In usual scenarios, operations like these are extremely straightforward but still have to be done. Feign provides a simple way of declaring these calls instead of worrying about the underlying implementations.

*Pros*

* Declarative, not concrete
* Feign provides the implementation on the fly
* Same conventions as using a REST Api using REST Template
* Feign internally supports Ribbon
* Feign internally supports Hystrix

*Cons*

* Customizing a REST call completely is very tedious
* It works best in standard API calls
* As of now doesn’t support Asynchronous Web Client

**Dependencies**

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

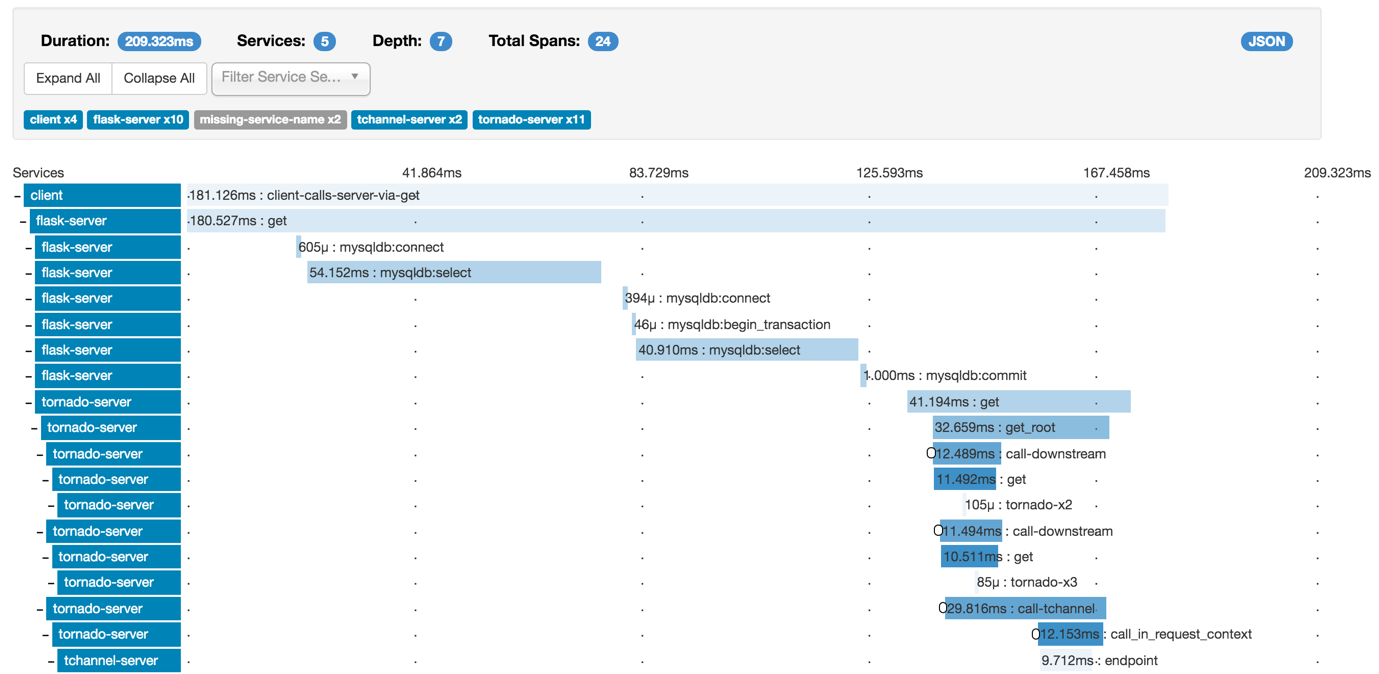
<version>1.4.6.RELEASE</version>

</dependency>

Git: <https://github.com/jaydeepresources/Tibco/tree/microservices-feign>

* **Zipkin and Sleuth: Cloud Tracing**

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in microservice architectures. It manages both the collection and lookup of this data. Zipkin’s design is based on the Google Dapperpaper.



Tracing information is collected on each host using the instrumented libraries and sent to Zipkin. When the host makes a request to another application, it passes a few tracing identifiers along with the request to Zipkin so we can later tie the data together into spans.

Tracing information is collected on each host using the instrumented libraries and sent to Zipkin. When the host makes a request to another application, it passes a few tracing identifiers along with the request to Zipkin so we can later tie the data together into spans.

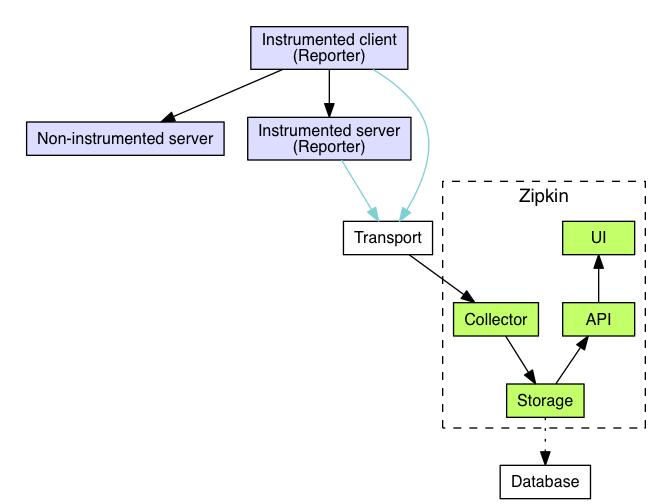
* **Span** – A span is an individual operation.
* **Trace** – A Trace is an end-to-end latency graph, composed of spans.
* **Tracers** – Tracers records spans and passes context required to connect them into a trace.

If you have Java 8 or higher installed, the quickest way to get started is to fetch the latest release as a self-contained executable jar:

curl -sSL https://zipkin.io/quickstart.sh | bash -s

java -jar zipkin.jar

This will star the zipkin server. We have to make some configurations so that our application/s can provide tracing information to the Zipkin Server.



* Tracers live in your applications and record timing and metadata about operations that took place. They often instrument libraries, so that their use is transparent to users. For example, an instrumented web server records when it received a request and when it sent a response. The trace data collected is called a Span.
* Instrumentation is written to be safe in production and have little overhead. For this reason, they only propagate IDs in-band, to tell the receiver there’s a trace in progress. Completed spans are reported to Zipkin out-of-band, similar to how applications report metrics asynchronously.
* For example, when an operation is being traced and it needs to make an outgoing http request, a few headers are added to propagate IDs. Headers are not used to send details such as the operation name.
* The component in an instrumented app that sends data to Zipkin is called a Reporter. Reporters send trace data via one of several transports to Zipkin collectors, which persist trace data to storage. Later, storage is queried by the API to provide data to the UI.

Zipkin UI can be used to see the visualization of traces and spans over time with restpect to each request.

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. You can capture data simply in logs, or by sending it to a remote collector service.

**Spring Cloud Sleuth features:**

1. Adds trace and span ids to the Slf4J MDC, so you can extract all the logs from a given trace or span in a log aggregator.
2. Provides an abstraction over common distributed tracing data models: traces, spans (forming a DAG), annotations, key-value annotations. Loosely based on HTrace, but Zipkin (Dapper) compatible.
3. Instruments common ingress and egress points from Spring applications (servlet filter, rest template, scheduled actions, message channels, zuul filters, feign client).
4. If spring-cloud-sleuth-zipkin is available then the app will generate and collect Zipkin-compatible traces via HTTP. By default it sends them to a Zipkin collector service on localhost (port 9411). Configure the location of the service using spring.zipkin.baseUrl.

**Dependencies**

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-sleuth</artifactId>

<version>${spring-cloud-sleuth.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

</dependencies>

Git: <https://github.com/jaydeepresources/Tibco/tree/microservices-zipkin>