# **The Power, Patterns, and Pains of Microservices**

### **Learn to use known distributed systems patterns to make your microservices-based applications more resilient and more robust.**

 by

[Joshua Long](https://dzone.com/users/206091/starbuxman.html)

·

Oct. 24, 15 · [Java Zone](https://dzone.com/java-jdk-development-tutorials-tools-news)

Liked [(6)](https://dzone.com/articles/the-power-patterns-and-pains-of-microservices#)

Comment (0)

Saved

[Tweet](https://dzone.com/articles/the-power-patterns-and-pains-of-microservices#)

10.43k Views

Discover how you can skip the build and redeploy process by using [JRebel by ZeroTurnaround](https://dzone.com/go?i=45221&u=http%3A%2F%2Fzeroturnaround.com%2Fsoftware%2Fjrebel%2F%3Futm_source%3Ddzone%26utm_medium%3Dbumper%26utm_campaign%3Djrebel).

*For more insights on microservices, JVM languages, and more trends in Java, get your* [*free copy*](https://dzone.com/guides/the-java-ecosystem-2015-edition?utm_source=DZone&utm_medium=Articles&utm_content=JLong&utm_campaign=Guide+to+Java) *of the* [*DZone Guide to the Java Ecosystem*](https://dzone.com/guides/the-java-ecosystem-2015-edition?utm_source=DZone&utm_medium=Articles&utm_content=JLong&utm_campaign=Guide+to+Java)*!*

It is not necessary to change. Survival is not mandatory.

—W. Edwards Deming

The principles [of the agile manifesto](http://agilemanifesto.org/principles.html) , now non-controversial and well accepted, speak to how to both write and deploy software more quickly and more safely—to production. Indeed, the very measure of success is defined by how quickly software is delivered to customers working— and working reliably. It's easy to forget this, but unless your customers can use it, it's not shipped . Software in production provides a vital feedback loop that helps businesses better react to market forces. Software in production is the only differentiator for any software business and—as my friend [Andrew Clay Shafer](http://twitter.com/littleidea) reminds us—"you are either building a software business, or you will be losing to someone who is."

Is this news? No, of course not. A full-throated advocate of winning knows that the one constant in business is change. The winners in today's ecosystem learned this early and quickly.

One such example is Amazon. They realized early on that they were spending entirely too much time specifying and clarifying servers and infrastructure with operations instead of deploying working software. They collapsed the divide and created what we now know as Amazon Web Services (AWS). AWS provides a set of well-known primitives, a cloud , that any developer can use to deploy software faster. Indeed, the crux of the DevOps movement is about breaking down the invisible wall between what we knew as developers and operations to remove the cost of this back-and-forth.

Another company that realized this is Netflix. They realized that while their developers were using TDD and agile methodologies, work spent far too long in queue, flowing from isolated workstations—product management, UX, developers, QA, various admins, etc.—until finally it was deployed into production. While each workstation may have processed its work efficiently, the clock time associated with all the queueing meant that it could sometimes be weeks (or, gulp , more!) to get software into production.

In 2009, Netflix moved to what they described as the cloud-native architecture . They decomposed their applications and teams in terms of features; small (small enough to be fed with two pizza-boxes !) collocated teams of product managers, UX, developers, administrators, etc., tasked with delivering one feature or one independently useful product. Because each team delivered a set of free-standing services and applications, individual teams could iterate and deliver as their use cases and business drivers required, independently of each other. What were in-process method invocations became independently deployed network services.

Microservices, done correctly, hack [Conway's law](https://en.wikipedia.org/wiki/Conway%27s_law) and refactor organizations to optimize for the continuous and safe delivery of small, independently useful software to customers. Independently deployed software can be more readily scaled at runtime. Independently deployed software formalizes service boundaries and domain models; domain models are forced to be internally consistent, something Dr. Eric Evans refers to as a bounded context in his epic tome, [Domain Driven Design](https://en.wikipedia.org/wiki/Domain-driven_design) .

Independent deployability implies agility but also implies complexity; as soon as network hops are involved you have a distributed systems problem!

## **Ride the Ride**

Thankfully, we don't have to solve the common distributed systems problems ourselves! The giants of the web who've come before and won have shared a lot of what they've done, and the rest of us—forever on the cusp of the next big viral mobile app or social-network runner—can learn from and lean on what they've provided. Let's look at some of the common patterns and various approaches to using them.

### **Consistency Improves Velocity**

My friend and former colleague [Dave McCrory](https://twitter.com/mccrory) coined the idea of data gravity —the inclination for pools of data to attract more and more data. The same thing— monolith gravity —exists with existing monolithic applications; any existing application will have inertia. A development team will face less friction in adding new endpoints and tables in a large SQL database to that application if the cost associated with standing up new services is significant. For many organizations, standing up new services can be a daunting task indeed! Many organizations have wiki pages with dozens of steps that must be carried out before a service can be deployed, most of which have little or nothing to do with the services' business value and drivers!

Microservices are APIs, typically REST APIs. How quickly can you stand up a new REST service? Microframeworks like [Spring Boot](http://start.spring.io) , [Grails](http://grails.org) , [Dropwizard](http://dropwizard.io) , [Play framework](https://www.playframework.com/) and [Wildfly Swarm](http://wildfly.org/swarm/) are optimized for quickly standing up REST services with minimum fuss. Extra points go to technologies that make it easy to build smart, self-describing hypermedia APIs as Spring Boot does with Spring HATEOAS.

Services are tiny, ephemeral, and numerous. The economics that made deploying lots of applications into the same JVM and application server interesting 15 years ago are long since gone for most of us, and most environments these days embrace process concurrency and isolation. What this means, in practice, is self-contained fat jars , which all of the aforementioned web frameworks will easily create for you.

You can't fix what you can't measure: how quickly can a service expose application state—metrics (gauges, meters, histograms, and counters), health checks, etc.—and how easy is it to report microservice state in a joined-up view or analysis tool like StatsD, Graphite, Splunk, the ELK (Elastic Search/Logstash/Kibana) stack, or OpenTSDB? One framework that brought metrics and log reporting to the forefront is [the Dropwizard microframework](http://www.dropwizard.io/) . [Spring Boot's Actuator](http://start.spring.io) module provides many of the same capabilities (and in some cases more) and transparently integrates with the Dropwizard Metrics library if it's on the CLASSPATH . A good platform like [Cloud Foundry](http://cloudfoundry.org) will also make centralized log collection and analysis dead simple.

Getting all of this out of the box is a good start, but it’s not enough. There is often much more to be done before a service can get to production. Spring Boot uses a mechanism called auto-configuration that lets developers codify things—identity provider integrations, connection pools, frameworks, auditing infrastructure, literally anything —and have it stood up as part of the Spring Boot application ( if all the conditions stipulated by the auto-configuration are met) just by being on the CLASSPATH! These conditions can be anything, and Spring Boot ships with many common and reusable conditions: is a library on the CLASSPATH ? Is a bean of a certain type defined (or not defined)? Is an environment property specified?

Starting a new service need not be more complex than a public static void main entry-point and a library on the CLASSPATH if you use the right technology.

### **Centralized Configuration**

The 12 Factor manifesto provides a set of guidelines for building applications with good cloud hygiene. One of the guidelines is to externalize configuration from the build so that one build of the final application can be promoted from development, QA, integration testing, and finally to a production environment. Environment variables and -D arguments, externalized .properties , and .yml files—which Dropwizard, Spring Boot, [Apache Commons Configuration](http://commons.apache.org/proper/commons-configuration/) and others readily support—are a good start, but even this can become tedious as you need to manage more than a few instances of a few types of services. This approach also fails several key use cases. How do you change configuration centrally and propagate those changes? How do you support symmetric encryption and decryption of things like connection credentials? How do you support feature flags, which toggle configuration values at runtime, without restarting the process?

[Spring Cloud](http://start.spring.io) provides the Spring Cloud Config Server, which stands up a REST API in front of a version controlled repository of configuration files. Spring Cloud also provides support for using Apache Zookeeper and Hashicorp Consul as configuration sources, as well as various clients for all of these so that all properties—whether they come from the Config Server, Consul, a -D argument, or an environment variable—work the same way for a Spring client. Netflix provides a solution called [Archaius](https://github.com/Netflix/archaius) , which acts as a client to a pollable configuration source. This is a bit too low-level for many organizations and lacks a supported, open-source configuration source counterpart, but Spring Cloud bridges the Archaius properties with Spring's, as well.

### **Service Registration and Discovery**

Applications spin up and down, and their locations may change. For this reason DNS—with its time-to-live expiration values—may be a poor fit for service discovery and location. It's important to decouple the client from the location of the service; a little bit of indirection is required. A service registry adds that indirection. A service registry is a phonebook, letting clients look up services by their logical names. There are many such service registries out there: some common examples include [Netflix's Eureka](https://github.com/Netflix/eureka) , [Apache Zookeeper](https://zookeeper.apache.org/) , and [Hashicorp Consul](https://www.consul.io/) . Modern platforms like Cloud Foundry don't necessarily need a separate service registry because of course it already knows where services live and how to find them given a logical name. At the very least, all applications will read from a service registry to inquire where other services live. Spring Cloud's DiscoveryClient abstraction provides convenient client-side API implementations for working with all manner of service registries, be they Apache Zookeeper, Netfix Eureka, Hashicorp Consul, Etcd, [Cloud Foundry](http://cloudfoundry.org) , [Lattice](http://lattice.cf) , etc. It's easy enough to plug in other implementations since Spring is a framework and a framework is (to borrow the Eiffel definition) “ open for extension .”

### **Client-Side Load Balancing**

A big benefit of using a service registry is client-side load balancing. Client-side load balancing lets the client find all the relevant registered instances of a given service—if there are 10 or a thousand, they're all discovered through the registry—and then choose from among the candidate instances which one to route requests to. The client can programmatically decide, based on whatever criteria it likes—capacity, round-robin, cloud-provider availability-zone awareness, multi-tenancy, etc.—to which node a request should be sent. Netflix provides a great client-side load balancer called [Ribbon](https://github.com/Netflix/ribbon) . Spring Cloud readily integrates Ribbon at all layers of the framework, so that whether you're using the RestTemplate, declarative REST clients powered by Netflix Feign, the Zuul microproxy, or anything else, the provided Ribbon load balancer strategy is in play automatically.

### **Edge Services: Microproxies and API Gateways**

Client-side load balancing is only used within the data center, or within the cloud, when making requests from one service to another. All of these services live behind the firewall. Services that live at the edge of the datacenter, exposed to public traffic, are exposed using DNS. An HTML5, Android, Playstation, or iPhone application will not use Ribbon. Services exposed at the edge have to be more defensive; they cannot propagate exceptions to the client. Edge services are intermediaries, and an ideal place to insert API translation or protocol translation. Take for example an HTML5 application. An HTML5 application can't run afoul of CORS restrictions: it must issue requests to the same host and port. A possible route might be to add a policy to every backend microservice that lets the client make requests. This of course is untenable and unscalable as you add more and more microservices. Instead, organizations like Netflix use a microproxy like [Netflix's Zuul](https://github.com/Netflix/zuul) . A microproxy like Zuul simply forwards all requests at the edge service to the backend microservices as enumerated in a registry. If your application is an HTML5 application, it might be enough to stand up a microproxy, insert HTTP BASIC or OAuth security, use HTTPS, and be done with it.

Sometimes the client needs a coarser-grained view of the data coming from the services. This implies API translation. An edge service, stood up using something like Spring Boot, might use Reactive programming technologies like [Netflix's RxJava](https://github.com/ReactiveX/RxJava) , [Typesafe's Akka](http://akka.io) , [RedHat's Vert.x](http://vertx.io/) , or [Pivotal's Reactor](http://projectreactor.io/) to compose requests and transformations across multiple services into a single response. Indeed, all of these implement a common API called the [reactive streams API](http://www.reactive-streams.org/) because this subset of problems is so common.

### **Clustering Primitives**

In complex distributed systems, there are many actors with many roles to play. Cluster coordination and cluster consensus is one of the most difficult problems to solve. How do you handle leadership election, active/passive handoff, or global locks? Thankfully, many technologies provide the primitives required to support this sort of coordination, including Apache Zookeeper, [Redis](http://redis.io) ,and [Hazelcast](https://hazelcast.com/) . [Spring Cloud's Cluster](http://start.spring.io) support provides a clean integration with all of these technologies.

### **Messaging, CQRS, and Stream Processing**

When you move into the world of microservices, state synchronization becomes more difficult. The reflex of the experienced architect might be to reach for distributed transactions, a la JTA. Ignore this urge at all costs. Transactions are a stop-the-world approach to state synchronization and slow the system as a whole—the worst possible outcome in a distributed system. Instead, services today use eventual consistency through messaging to ensure that state eventually reflects the correct system worldview. REST is a fine technology for reading data but it doesn't provide any guarantees about the propagation and eventual processing of a transaction. Actor systems like [Typesafe Akka](http://akka.io) and message brokers like [Apache ActiveMQ](http://activemq.apache.org/) , [Apache Kafka](http://kafka.apache.org/) , [RabbitMQ](http://rabbitmq.com) , or [even Redis](http://redis.io/) have become the norm. Akka provides a supervisory system that guarantees a message will be processed at least once. If you're using messaging, there are many APIs that can simplify the chore, including [Apache Camel](http://camel.apache.org/) , [Spring Integration](http://projects.spring.io/spring-integration/) , and—at a higher abstraction level and focusing specifically on the aforementioned Kafka, RabbitMQ, and Redis—Spring Cloud Stream. Using messaging for writes and using REST for reads optimizes reads separately from writes. The [Command Query Responsibility Segregation](http://martinfowler.com/bliki/CQRS.html) , or CQRS, design pattern specifically describes this approach.

### **Circuit Breakers**

In a microservice system it's critical that services be designed to be fault-tolerant: if something happens, then the services should gracefully degrade. Systems are complex, living things. Failure in one system can trigger a domino effect across other systems if care isn't taken to isolate them. One way to prevent failure cascades is to use a circuit-breaker . A circuit-breaker is a stateful component around potentially shaky service-to-service calls that—when something goes wrong—prevents further traffic across the downed path. The circuit will slowly attempt to let traffic through until the pathway is closed again. [Netflix's Hystrix circuit-breaker](https://github.com/Netflix/Hystrix) is a very popular option, complete with a usual dashboard which can aggregate and visualize potentially open circuits in a system. Wildfly Swarm, as of this writing in Q3 2015, has support for using Hystrix in master, and the [Play Framework](https://www.playframework.com/) provides support for circuit breakers. Naturally, Spring Cloud also has deep support for Hystrix, and we're investigating a possible integration with [JRugged](https://github.com/Comcast/jrugged) .

### **Distributed Tracing**

A microservice system with REST, messaging, and proxy egress and ingress points can be very hard to reason about in the aggregate: how do you trace—correlate requests across a series of services and understand where something may have failed? This is very difficult without a sufficient upfront investment in a tracing strategy. [Google's Dapper](http://research.google.com/pubs/pub36356.html) first described such a distributed tracing tool. Google's Dapper paper paved the way for many other such systems, including one at Netflix, which they have not open-sourced. [Apache HTRace is also Dapper-inspired](http://incubator.apache.org/projects/htrace.html) . [Twitter's Zipkin](https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin) is open-source and actively maintained. It provides the infrastructure and a visually appealing dashboard on which you can view waterfall graphs of calls across services. Spring Cloud has a module called Spring Cloud Sleuth that provides correlation IDs and instrumentation across various components. Spring Cloud Zipkin integrates Twitter Zipkin in terms of the Spring Cloud Sleuth API. Once added to a Spring Cloud module, requests across messaging endpoints using Spring Cloud Stream, REST calls using the RestTemplate, and HTTP requests powered by Spring MVC are all transparently and automatically traced.

### **Single Sign-On**

Security is hard. In a distributed system, it is critical to ascertain the provenance and authenticity of a request in a consistent way across all services, quickly. On the open web, OAuth and OpenID Connect are very popular. In the enterprise, technologies like SAML are very popular. OAuth 2 provides explicit integration with SAML. API gateway tools like [Apigee](http://apigee.com/) and SaaS identity providers like [Stormpath](https://stormpath.com/) can act as a security hub, exposing OAuth (for example) and connecting the backend to more traditional identity providers like ActiveDirectory, SiteMinder, or LDAP. Finally, [Spring Security OAuth](http://projects.spring.io/spring-security-oauth/) provides an identity server, which can then talk to any identity provider in the backend. Whatever your choice of identity provider, it should be trivial to protect services based on some sort of token. Spring Cloud Security makes short work of protecting any REST API with tokens from any OAuth 2 provider—Google, Facebook, the Spring Security OAuth server, Stormpath, etc. [Apache Shiro](http://shiro.apache.org/) can also act as an OAuth client using the Scribe OAuth client.

## **Don't Reinvent the Ride's Wheels**

We've looked at a fairly extensive list of concerns that are unique to building cloud-native applications. Trust me: you do not want to reinvent this stuff yourself. There's a lot of stuff to care for, and unless you've got Netflix's R&D budget (and smarts!), you're not going to get there anytime soon. Building the pieces is one thing, but pulling them together into one coherent framework? That's a whole other kettle of fish and few pull it off well. Indeed, even the likes of Netflix, Alibaba, and TicketMaster are using Spring Cloud (which builds on Spring Boot) because it removes so much complexity and lets developers focus on the essence of the business problem.

In a sufficiently distributed system, it is increasingly futile to optimize for high availability and paramount to optimize for reduced time-to-remediation. Services will fall down; the question is: how quickly can you stand them up again? This is why microservices and cloud-computing platforms go hand-in-hand: as the platform picks up the pieces, the software needs to be smart enough to adapt to the changing landscape, and to degrade gracefully when something unexpected happens.