Microservices in Java

Survival is not mandatory

It’s not controversial to suggest that startups iterate and innovate faster than larger organizations, but what about the larger organizations — the Netflixes, the Alibabas, the Amazons, etc.? How do they innovate so quickly? The secret to their agility lays in the size of their teams. These organizations deploy small teams that in turn build small, singly focused, independently deployable, microservices. Microservices are easier to build because the implementation choices don’t bleed over to other functionality, they minimize the scope and impact of a given change, they are easier to test; and they are easier to scale. Microservices give us a lot of benefits, but they also introduce complexities related to incepting new services and addressing the dynamics of distribution.

Moving beyond the wiki page: “500 easy steps to production”

Microservices are APIs. How quickly can you stand up a new service? Microframeworks like Spring Boot, Grails (which builds on Spring Boot), JHipster (which builds on Spring Boot), DropWizard, Lagom, and WildFly Swarm are, at a minimum, optimized for quickly standing up REST services with a minimum of fuss. Some of these options go further and address all production requirements including security, observability and monitoring. Cloud computing technologies like Cloud Foundry, OpenShift, Heroku, and Google App Engine provide higherlevel abstractions for managing the lifecycle of software. At both layers, stronger opinions result in consistency which results in velocity. These technologies let organizations move beyond the burdensome internal Wiki page, “500 Easy Steps to Production.”

Survival is not mandatory

One thing is common across all organizations, no matter what their technology stack: when the pager goes off, somebody is roused and sat in front of a computer in order to stop the system’s bleeding. Root cause analysis will come later; the priority is to restore service. The more we can do upfront to support remediation in those waning and critical seconds, minutes, and hours after a production outage, the better. Services should expose their own health endpoints, metadata about the service itself, logs, threadumps, configuration and environment information, and whatever else could be operationally useful. Services should also track the progression of both business and operational metrics. Metrics can be overwhelming, and so it makes sense to publish collected metrics to time-series databases that support analytics and visualization. There are many time-series databases like Graphite, Atlas, InfluxDB, and OpenTSDB. Metrics are keys and values over time. Spring Boot suppots the collection of metrics, and can delegate to projects like Dropwizard Metrics and Micrometer.io to publish them.

```
@SpringBootApplication
public class DemoApplication {

  public static void main(String args[]) {
    SpringApplication.run(DemoApplication.class, args);
  }

  @Bean
  GraphiteReporter graphite(@Value(“${graphite.prefix}”) String
    GraphiteReporter reporter = GraphiteReporter.forRegistry(reg
    reporter.start(1, TimeUnit.SECONDS);
    return reporter;
  }

}

@RestController
class FulfillmentRestController {

  @Autowired
  private CounterService counterService;
  @RequestMapping(“/customers/{customerId}/fulfillment”)
  Fulfillment fulfill(@PathVariable long customerId) {
    // ..
    counterService.increment(“meter.customers-fulfilled”);
    // ..
  }

}
```

Log multiplexers like Logstash or Cloud Foundry’s Loggregator funnel the logs from application instances and ship them to downstream log analysis tools like ElasticSearch, Splunk, or PaperTrail.

Centralized configuration

The Twelve-Factor App methodology provides a set of guidelines for building applications with good, clean cloud hygiene. One tenet is that environment-specific configuration should live external to the application itself. It might live in environment variables, -D arguments, externalized .properties, .yml files, or any other place, so long as the application code itself need not be recompiled. DropWizard, Spring Boot, Apache Commons Configuration, and others support this foundational requirement. However, this approach fails a few 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 provides the Spring Cloud Config Server (and a client) which stands up a REST API in front of a version-controlled repository of configuration files, and Spring Cloud provides support for using Apache Zookeeper and HashiCorp Consul as configuration sources. Spring Cloud provides 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 that 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, too

The config server

```
# application.properties
spring.cloud.config.server.git.uri=https://github.com/joshlong/m
server.port=8888
```

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServiceApplication {

  public static void main(String[] args) {
    SpringApplication.run(ConfigServiceApplication.class, args);
  }

}
```

The config client

```
# application.properties
spring.cloud.config.uri=http://localhost:8888
spring.application.name=message-client
```

```
# Will read https://github.com/joshlong/my-config/messageclient.
@SpringBootApplication
public class ConfigClientApplication {

  public static void main(String[] args) {
    SpringApplication.run(ConfigClientApplication.class, args);
  }

}
// supports dynamic re-configuration:
// curl -d{} http://localhost:8000/refresh
@RestController
@RefreshScope
class MessageRestController {

  @Value(“${message}”)
  private String message;

  @RequestMapping(“/message”)
  String read() {
    return this.message;
  }

}
```

Service registration and discovery

DNS is sometimes a poor fit for intra-service communication. DNS benefits from layers of caching and time-to-liveness that work against
services in a dynamic cloud environment. In most cloud environments, DNS resolution requires a trip out of the platform to the router and then
back again, introducing latency. DNS doesn’t provide a way to answer the question: is the service I am trying to call still alive and responding?
It can only tell us where something is supposed to be. A request to such a fallen service will block until the service responds, unless the client specifies a timeout (which it should!). DNS is often paired with load balancers, but third-party load balancers are not sophisticated things:
they may support round-robin load balancing, or even availability zoneaware load balancing, but may not be able to accomodate businesslogic-specific
routing, like routing a request with an OAuth token to a specific node, or routing requests to nodes collocated with data, etc. It’s
important to decouple the client from the location of the service, but DNS might be a poor fit. A little bit of indirection is required. A service registry provides 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.
Netflix’s Eureka, Apache Zookeeper, and HashiCorp Consul are three good examples. Spring Cloud’s DiscoveryClient abstraction provides a convenient client-side API for working with service registries. Here, we inject the DiscoveryClient to interrogate the registered services:

```
@Autowired
public void enumerateServiceInstances(DiscoveryClient client) {
  client.getInstances(“reservation-service”).forEach( si -> Syst
}
```

Client-side load balancing

A big benefit of using a service registry is client-side load balancing. Client-side load balancing lets the client pick from among the 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, least-recently used, cloud-provider availability-zone awareness, multitenancy, etc.—to which node a request should be sent. Netflix provides a great client-side load balancer called Ribbon that Spring Cloud integrates with. Ribbon is automatically in play at all layers of the framework, whether you’re using the RestTemplate, the reactive WebFlux WebClient, declarative REST clients powered by Netflix’s Feign, the Zuul microproxy or Spring Cloud Gateway.

```
@EnableDiscoveryClient
@SpringBootApplication
public class ReservationClientApplication {

  @Bean
  @LoadBalanced // lets us use service registry service IDs as h
  RestTemplate restTemplate() {
    return new RestTemplate();
  }

  public static void main(String[] args) {
    SpringApplication.run(ReservationClientApplication.class, ar
  }

}

@RestController
class ApiClientRestController {

  @Autowired
  private RestTemplate restTemplate;

  @RequestMapping(method = RequestMethod.GET, value = “/reservat
  public Collection<String> names() {
    ResponseEntity<JsonNode> responseEntity = restTemplate.excha
    HttpMethod.GET, null, JsonNode.class);
    // ...
  }

}
```

Edge services: api gateways and api adapters

Client-side load-balancing works for intra-service communication, usually behind a firewall. External clients—iPhones, HTML5 clients, Android clients, etc.—will need DNS and will have client-specific security, payload, and protocol requirements. An edge service is the first port of call for requests coming from these external clients.
You address client-specific concerns at the edge service and then forward the requests to downstream services. An API gateway (sometimes called a backend for a frontend) supports declarative, cross-cutting concerns like rate limiting, authentication, compression, and routing.
API adapters have more insight into the semantics of the downstream services; they might expose a synthetic view of the responses of downstream services, combining, filtering, or enriching them.
There are many API gateways, some hosted and some not, like Apigee, WS02, Nginx, Kong, Netflix Zuul, and Spring Cloud Gateway. I like to think of Netflix Zuul and Spring Cloud Gateway as microproxies. They're small and embeddable. API Gateways need to be as fast as possible and able to absorb as much incoming traffic as possible. Here, non-blocking, reactive APIs (using technologies like Netflix's RXJava 2, Spring WebFlux, RedHat's Vert.x and Lightbend's Akka Streams) are a very good choice.

```
@EnableDiscoveryClient
@SpringBootApplication
public class EdgeServiceApplication {

  // configure reactive, Ribbon aware `WebClient`
  @Bean
  WebClient client(LoadBalancerExchangeFilterFunction lb) {
    return WebClient.builder().filter(lb).build();
  }

  // API Adapter using reactive Spring WebFlux
  WebClient
  @Bean
  RouterFunction<?> endpoints(WebClient client) {
    List<String> badCars = Arrays.asList(“Pinto”, “Gremlin”);
    return route(GET(“/good-cars”), req -> {
      Publisher<Car> beers = client.get().uri(“http://car-catalo
      Publisher<Car> circuit = HystrixCommands.from(beers).comma
      return ServerResponse.ok().body(circuit, Car.class);
    });
  }

  // API gateway with Spring Cloud Gateway
  @Bean
  RouteLocator gateway() {
    return Routes.locator().service.route(“path_route”).predicat
  }

  public static void main(String[] args) {
    SpringApplication.run(EdgeServiceApplication.class, args);
  }

}
```

Clustering primitives

In a complex distributed system, 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, and Hazelcast. Spring Integration supports a clean integration with these kinds of technologies. In the following example, we’ve configured a component to change its state whenever OnGrantedEvent or an OnRevokedEvent is emitted, which it will do when the underlying coordination technology promotes and demotes a leader node.

```
@Component
class LeadershipApplicationListener {

  @EventListener(OnGrantedEvent.class)
  public void leadershipGranted(OnGrantedEvent evt){
    // ..
  }

  @EventListener(OnRevokedEvent.class)
  public void leadershipRevoked(OnRevokedEvent evt){
    // ..
    }

  }
```

becomes more difficult. The reflex of the experienced architect might be to reach for distributed transactions, a la JTA. Ignore this impulse at all costs. Distributed transactions are a stop-the-world approach to state synchronization; that is the worst possible outcome in a distributed system. And, indeed, their use is moot since your microservices will not speak the X-Open protocol—for which JTA is middleware—as your RDBMS or JMS message queue might. One of the main reasons to move to microservices is to retain autonomy over your services' implementation, so that you don't need to constantly synchronize with other parts of the organization when making changes.
So allowing access to your JTA-capable resources isn't an option, anyway. Instead, services today use eventual consistency through messaging to ensure that state eventually reflects the correct system world-view. 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 Lightbend Akka and message brokers like Apache ActiveMQ, Apache Kafka, RabbitMQ, or even Redis 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 Spring Integration, Apache Camel and —at a higher abstraction level— Spring Cloud Stream. Using messaging for writes and REST for reads optimizes reads separately from writes. The Command Query Responsibility Segregation—or CQRS—design pattern specifically espouses this approach (though it does so separately from any discussion of a particular protocol or technology).

In the example below, Spring Cloud Stream connects a client to three services described in terms of MessageChannel definitions in the CrmChannels interface. Messages sent into the channels are communicated to other nodes through a Spring Cloud Stream binder implementation that in turn talks to a messaging technology like RabbitMQ or Apache Kafka. The configuration that binds a MessageChannel to a destination in a messaging technology is external to the code.

```
// producer side
@EnableBinding(CrmChannels.class)
```

```
@SpringBootApplication
public class ProductsEdgeService {

  public static void main(String[] args) {
    SpringApplication.run(ReservationClientApplication.class, ar
  }

}

interface CrmChannels {

  @Output
  MessageChannel orders();

  @Output
  MessageChannel customers();

  @Output
  MessageChannel products();

}

@RestController
@RequestMapping(“/products”)
class ProductsApiGatewayRestController {

  @Autowired
  private MessageChannel products;

  @RequestMapping(method = RequestMethod.POST)
  public void write(@RequestBody Product p) {
    Message<Product> msg = MessageBuilder.withPayload(p).build()
    products.send(msg);
  }

}
```

On the consumer side you might consume incoming messages using a `@StreamListener`:

```
// on the consumer side
@EnableBinding(Sink.class) // contains a definition for an `inpu
@SpringBootApplication
public class MessageConsumerService {

  public static void main(String[] args) {
    SpringApplication.run(MessageConsumerService.class, args);
  }

}

@EnableBinding(Sink.class)
public class ProductHandler {

  @Autowired
  private ProductRepository products;

  @StreamListener(Sink.INPUT)
  public void handle(Product p) {
    products.addProduct(vote);
  }

}
```

Retries and circuit breakers

You can't take for granted that a downstream service will be available. If you attempt to invoke a service and a failure occurs, then you should be ready to retry the call. Services fail for all sorts of often ephemeral reasons. Spring Retry supports automatically retrying a failed action. You can specify what exceptions should trigger a retry, how many retries should be attempted, and when they should happen.

Circuit breakers, like Spring Retry, Netflix's Hystrix or JRugged, go one step beyond basic retry functionality: they are stateful and can determine that a call is not going to succeed and that the fallback behavior should be attempted directly. The effect is that downstream services, which would otherwise be overwhelmed, are given an opportunity to recover while shortening the time to recovery for the client. Some circuit breakers can even execute some behaviors in an isolated thread so that, if something goes wrong, the main processing can continue unimpeded. Spring Cloud supports both Netflix Hystrix and Spring Retry. WildFly Swarm also supports Netflix Hystrix. The Play Framework provides support for circuit breakers.

Circuit breakers represent connections between services in a system; it is important to monitor them. Hystrix provides a dashboard for its circuits. Spring Cloud also has deep support for Hystrix and its dashboard. Each Netflix Hystrix circuit breaker exposes a server-sent event-based stream of status information about the Netflix Hystrix on that node. Spring Cloud

Turbine supports multiplexing those various streams into a single stream to federate the view of all the circuits across all the nodes across the whole system.

```
@RestController
class EdgeService {

  public Collection<String> fallback(){
    // this will be invoked if the `names` method throws an exce
  }

  // the dashboard will show a circuit named ‘reservationservice
  @HystrixCommand(fallbackMethod = “fallback”)
  @RequestMapping(method = RequestMethod.GET, value = “/names”)
  public Collection<String> names() {
    // ..
  }

}
```

Distributed tracing

It is difficult to reason about a microservice system with REST-based, messaging-based, and proxy-based egress and ingress points. How do you trace (correlate) requests across a series of services and understand where something has failed? This is difficult enough a challenge without a sufficient upfront investment in a tracing strategy. Google introduced their distributed tracing strategy in their Dapper paper. Apache HTRace is a Dapper- inspired alternative. Twitter’s Zipkin is another Dapper-inspired tracing system. It provides the trace collection infrastructure and a UI in which you can view waterfall graphs of calls across services, along with their timings and trace-specific information. Spring Cloud Sleuth provides an abstraction around the concepts of distributed tracing. Spring Cloud Sleuth automatically traces common ingress and egress points in the system. Spring Cloud Zipkin integrates Twitter Zipkin in terms of the Spring Cloud Sleuth abstraction.

Single sign-on and security

Security describes authentication, authorization and-often-which client is being used to make a request. OAuth and OpenID Connect are very popular on the open web, and SAML rules the enterprise.

OAuth 2.0 provides explicit integration with SAML. API gateway tools like A pigee and SaaS identity providers like Okta can act as a secure meta-directory, exposing OAuth endpoints (for example) and connecting the backend to more traditional identity providers like Active Directory, Office365, Salesforce, and LDAP. Spring Security OAuth and RedHat's KeyCloak are open-source OAuth and OpenID Connect servers. Whatever your choice of identity provider, it should be trivial to authenticate and authorize clients. Spring Security, Apache Shiro, Nimbus, and Pac4J all provide convenient OAuth clients. Spring Cloud Security can lock down microservces, rejecting un-authenticated requests. It can also propagate authentication contexts from one microservice to another. Frameworks like JHipster integrate Spring's OAuth support.

A cloud native architecture is an agile architecture

Systems must optimize for time-to-remediation; when a service goes down, how quickly can the system replace it? If time-to-remediation is 0 seconds, then the system is (effectively) 100% highly available. The apparent appearance of the system is the same in a single-node service that is 100% highly available, but it has profound impacts on the architecture of the system. The patterns we've looked at in this
Refcard support building systems that are tolerant of failure and service topology changes common in a dynamic cloud environment.
Remember: the goal here is to achieve velocity, and to waste as little time as possible on non-functional requirements. Automation at the
platform and application tiers support this velocity. Embracing one without the other only invites undifferentiating complexity into an
architecture and defeats the purpose of moving to this architecture in the first place.