WHITEPAPER

5 INSTRUMENTATION STRATEGIES FOR ARCHITECTING CONTAINERIZED APPLICATIONS

David Gordon

EXECUTIVE SUMMARY

As organizations worldwide adopt container-based application architectures, developers must adjust existing software implementations—originally deployed to traditional infrastructure—for a container platform environment. This whitepaper introduces five fundamental concerns for application containerization and corresponding application instrumentation strategies that take advantage of trending open source and container-native technologies.

EXTERNALIZED CONFIGURATION

In general, continuous delivery in container environments is based on the principle that a container image is an immutable and certifiable artifact. This implies that the same image tested in a (quality assurance) QA environment, for example, is deployed without changes to the next environment in the continuous integration/continuous delivery (CI/CD) pipeline sequence. However, some application configuration is typically intended to vary between environments and for good reason. Common examples of environment-specific application configuration include external database connection strings, public application programming interface (API) endpoints, and feature toggle flags. In order to preserve container immutability, environment-specific configuration should be loaded from an external source.

Traditional software architectures often employ a technique where applications consume environment-specific configuration data located in a file system attached to an application's host machine. This approach makes a key assumption that a given application is deployed to a predictable host or set of hosts.

Platforms such as Kubernetes and OpenShift use a scheduling algorithm to evaluate available infrastructure and then dynamically assign containers to hosts in the cluster. A container can be assigned to any host node that meets the scheduling algorithm's criteria. Therefore application configuration information must be available everywhere in the cluster.

Modern approaches to providing application configuration in a container environment share a common basic principle: provide configuration data via a service endpoint.

Spring Cloud Config Server is a popular open source application configuration provider service. By default, the service exposes a RESTful endpoint that returns configuration property files maintained in a Git repository. The Spring community also provides an application-side client for retrieving configuration information. Using these components, a Spring application can be configured to periodically poll the configuration provider service and reload its context if changes are detected. This strategy requires a running instance of Spring Cloud Config Server and a configuration data store such as a hosted Git repository.

For an alternative approach, the OpenShift API exposes a ConfigMap object, which can represent a file, such as an application property file, accessible by any authorized pod in the cluster. The Spring Cloud Config Kubernetes client allows Spring applications to use configuration data directly from ConfigMaps at run-time. So even without a running Spring Cloud Config Server, a Spring application deployed to OpenShift can use configuration dynamically. The client is supported in Red Hat® JBoss® Fuse and was developed upstream in the Fabric8 community.

LOG MANAGEMENT

Traditionally, application logs are written to files on a disk, and again, the assumption likely exists that the application host is predetermined. Knowing where an application process will run means operations teams know where logs will be written. Because containers are dynamically assigned to nodes when deployed to a container platform like Kubernetes, an application designed to write logs to its local file system is not efficient. Logs with various formatting, naming, and path conventions end up scattered throughout the cluster infrastructure. Furthermore, a container's local file system typically follows the life cycle of the container itself. If a container process is destroyed, anything written to the local file system is destroyed as well.

A cluster-wide convention for log destination and formatting is required for a container-native log management solution, and this can be achieved by delegating the concern to the container engine. For example, by default, Docker captures standard output and standard error information for a container's main process, and writes it to files using a standardized JSON-based format. These files are located under /etc/docker and uniquely named using the container ID that produced the log. In this case, the impact on the application implementation is subtle: direct all application logging to standard output. Log frameworks such as Logback, the self-proclaimed successor to log4j, can direct logging to a standard output with a simple configuration. Design container-based applications to use loggers from configurable frameworks like Logback so that formatting and output destination can be managed efficiently. Standard output (stdout) and standard error (stderr) should be the only logging destinations.

In addition to container log organization, operations teams need a strategy for collecting and searching logs that are scattered throughout container platform infrastructure. A popular technology stack for providing log aggregation, storage, and presentation is known as EFK (ElasticSearch, Fluent.d, Kibana). Container logs, as well as platform component logs, are collected and streamed to a distributed datastore. Data is presented using Kibana's highly configurable user interface with capabilities that include designing dashboards and investigating individual pod behavior.

DISTRIBUTED TRACING

A trace tells the story of a transaction as it propagates through a distributed system. So, a tracing implementation must piece together information about a transaction using data gathered from several components of a system.

Container-based applications are often deployed as several components that work together as a system. Applications architected as a suite of independently deployable and modular services follows a trending style called microservices. While it is completely optional to follow strict microservice patterns when developing applications for deployment to a container platform, it is important to remember that applications deployed to container platforms tend to be composed with multiple containers. Transaction tracing through a highly distributed and/or decomposed architecture is challenging because the sources of trace data are scattered throughout a pool of infrastructure. Several open source solutions for distributed systems tracing are gaining industry momentum.

Zipkin is an open source tracing system that has received high adoption levels in the past few years, popularized as part of the Spring framework ecosystem. Other organizations, such as Uber, have developed new implementations of distributed tracing tools.

To alleviate the concern that a tracing framework implementation could result in system coupling to that particular implementation, the OpenTracing Initiative was founded to create a vendor-neutral tracing standard. The specifications and semantic conventions described by the OpenTracing Initiative are largely inspired by Zipkin. Uber's Jaeger, in which multiple Zipkin components were replaced, adheres to OpenTracing conventions, effectively maintaining the ability to switch between OpenTracing-compliant implementations.

Solutions that comply with OpenTracing standards allow for architectural flexibility in the future. Some of the most effective and well-adopted solutions like Zipkin and Jaeger provide readily available options.

METRICS

Effective application metrics collection in container environments presents many of the same challenges described earlier for logging and tracing. Because of the ephemeral nature of containers, metrics endpoints are not static; container instances are eventually replaced with upgraded instances that could spin up on a different node in the cluster. Platforms like Kubernetes and OpenShift use a networking abstraction called a service, which defines a logical set of pods and a policy by which to access them. However, a Kubernetes service provides an ineffective monitoring endpoint, because more granular statistics are needed about individual containers rather than a group of containers collectively.

Prometheus is an open source monitoring and alerting toolkit that is especially popular in microservices contexts. Prometheus includes components for collecting and displaying metrics and a comprehensive menu of instrumentation libraries including the Java Management Extension (JMX) exporter.

JMX is a standard for monitoring Java™ applications. With the help of metrics agents like Jolokia and Prometheus JMX Exporter, a metrics user can observe an aggregated view of JMX metrics from a set of containers using the Kubernetes API.

Exposing application metrics is recommended for each and every application component in a container environment. Consider using popular instrumentation libraries to publish metrics endpoints that are compliant with Prometheus exposition formats. Much like the distributed tracing domain, application metrics and monitoring seem to be trending toward a commonly accepted specification, and Prometheus has momentum as a leading standard.

HEALTH CHECKS

Instrumenting containerized applications with health check endpoints is critical for self-healing architectures. Applications deployed to traditional infrastructure tend to have static network addresses, and traditional application monitoring often takes advantage of the predictability of a service or host endpoint. Because containers are scheduled to run on nodes dynamically, health monitoring services must keep track of all container instances.

Kubernetes provides a feature that automatically monitors health check endpoints and responds to containers that are in an unhealthy state. All containers in a cluster are monitored, and Kubernetes has the ability to respond to health checks by deploying, deleting, or restarting pods. The application must expose an API for observing its health status, thus requiring some minor instrumentation.

Health check instrumentation provides various levels of effectiveness. For example, an application can expose an isolated controller that returns a 200 HTTP response code when invoked. This health check is useful in many cases, but it only uncovers certain types of problems. If an application does not have a healthy connection to a database, a shallow health check on the endpoint will probably not detect the problem.

The most effective health checks comprehensively inventory the status of all components and connections that are critical to the application. Spring Boot Actuator is a popular health check instrumentation library that scans a Spring application context and interrogates the runtime status of each component it finds. The type of deep health check that Spring Boot Actuator supports is highly recommended for containerized applications.

SUMMARY

As container adoption accelerates, the migration of traditional applications will attempt to keep pace. Having an arsenal of tools to address common containerization concerns will streamline both greenfield development and legacy application migration for a resilient and effective implementation.

In many cases, application implementations can be effective without changes, either deployed on a container platform or to traditional infrastructure. However, efficiently managing applications composed of many containers and deployed to a pool of infrastructure is made possible through technologies that expose information about application runtimes and enable applications to observe and respond to environmental conditions.

These resources provide more information about the instrumentation technologies described in this whitepaper:

- **Spring Cloud Config Kubernetes** for externalized application configuration (https://github.com/spring-cloud-incubator/spring-cloud-kubernetes)
- **Logback** for application log management (https://logback.qos.ch)
- **Zipkin** (https://zipkin.io) and **Jaeger** (http://jaegertracing.io) for distributed tracing using the **Open Tracing** standard (http://opentracing.io)
- **Prometheus** (https://prometheus.io) as a metrics toolkit including **Prometheus JMX Exporter** (https://github.com/prometheus/jmx_exporter) for metrics publishing
- **Spring Boot Actuator** (https://spring.io/guides/gs/actuator-service) for exposing deep health check endpoints

NORTH AMERICA
1 888 REDHAT1

EUROPE, MIDDLE EAST, AND AFRICA
00800 7334 2835
europe@redhat.com

ASIA PACIFIC
+65 6490 4200
apac@redhat.com

LATIN AMERICA
+54 11 4329 7300
info-latam@redhat.com

facebook.com/redhatinc
@redhatnews
linkedin.com/company/red-hat

redhat.com
#f11943_0418