# Event-driven Microservices

Jeremy Deane – jeremy.deane@gmail.com

*Using the Microservices Architectural Style to incrementally adopt an Event-driven Architecture (EDA) lowers up-front costs while decreasing time-to-market. EDA extracts value from existing occurrences, limiting invasive refactoring or disrupting existing application development efforts. Implementing Event-driven Microservices yields intelligence, scalable, extensible, reactive endpoints.*

> *"All those… moments… will be lost in time, like tears… in… rain"*
>
> *— Roy Batty*
> *Blade Runner*

Integration of heterogeneous distributed systems is not only ubiquitous but also required in today's fast economy. It is simply not possible to build a system with all the required features nor is there a one stop-shop commercial solution for all ones needs, although the folks at Salesforce may beg to differ. Consequently, more and more time is spent integrating a myriad of bespoke, open source, and commercial services in order to build a complete solution. And the most widely accepted integration strategy is based on RESTful Web Services.

Producing and consumer REST Web Services is both inexpensive and expedient. However, this synchronous integration approach tightly couples both parties to a contract that is hard to break or work around, even if the Producer supports versioning (e.g. via X-API-Version header). In addition, something critical is lost during this stateless transaction, information about the occurrence itself. Of course both parties may log data, from their perspective, but other concerned parties remain oblivious. Rather than let this valuable data exhaust evaporate off into the ether, capture it as an event.

Events represent a snapshot in time of an occurrence within a system. The data within an event is informational to some parties and often actionable to others. That said, it is not uncommon for event data to simply be noise. For example, a heart rate event of 75 bpm is

normal (informational) but above 200 bpm or below 60 bpm may be actionable. The context of the event, such as age, weight, and health of the person, factors into whether action is required.

> *A new life awaits you in the Off-world colonies! A chance to begin again in a golden land of opportunity and adventure!*
>
> — *Advertisement*
> *Blade Runner*

Events are far more than just the data thrown on the shop floor. An event can capture a command, such as invalidate cache or suspend processing, and then be broadcasted to the entire enterprise. The Event Sourcing Pattern (http://martinfowler.com/eaaDev/EventSourcing.html) uses a command event to capture a request to alter or create data in distributed persistance solutions and in the case of failure allows the events to replayed. Furthermore, events can capture information about integration exchanges or transactions in external systems. An example, in healthcare, is an HL7 FHIR Encounter (http://www.hl7.org/fhir/encounter.html) event emitted by a hospital system that is consumed by both healthcare payer and primary care physician systems.

*HL7 FHIR Encounter*

```javascript
{
  "resourceType": "Encounter",
  "id": "EMR56788",
  "text": {
    "status": "generated",
    "div": "Patient admitted with chest pains</div>"
  },
  "status": "in-progress",
  "class": "inpatient",
  "patient": {
    "reference": "Patient/P12345",
    "display": "Roy Batty"
  }
}
```

In the previous example, integrating several distributed healthcare systems using web services is quite possible but those point-to-point (P2P) synchronous integrations become very hard to maintain overtime. In fact, P2P integration within an enterprise often causes a type of system paralysis because of cascading integration dependencies. The remedy is to

apply an asynchronous integration strategy. Caution, at this point one could easily be baited into adopting a costly “Mega-vendor Thneed”. As Neal Ford is apt to say “These solutions attract Enterprise Architects like rotten meat attracts flies”

Fortunately, a plethora of free open source Message-oriented Middleware (MOM) projects exist allowing for rapid incremental adoption. Broker based solutions such as Apache ActiveMQ and RabittMQ, support widely accepted messaging standards, including JMS and AQMP, and scale both horizontally and vertically. In addition, these solutions support highly available federated topologies. An alternate asynchronous integration strategy is to go with a *brokerless socket-based* MOM solution such as ZeroMQ.

> *“Nothing is worse than having an itch you can never scratch!*
>
> *— Leon Kowalski*
> *Blade Runner*

Regardless of the underlying delivery mechanism, events flow asynchronously across the enterprise. Of course, the real value is manipulating and reacting to these events streams. Caution, another trap is to centrally control and orchestrate event processing. It will not be long before the Enterprise Architects will be setting up a “Center of Excellence” resulting in another type of organizational paralysis.

A better approach is Implementing intelligent endpoints that simply emit and react to events. Intelligent reactive endpoints have a single purpose, such as applying a set of rules to event data or enriching the event based on context. These endpoints can maintain state but do not share it with other endpoints. Implementing *single purpose* endpoints is the ethos of the Microservice Architectural Style.

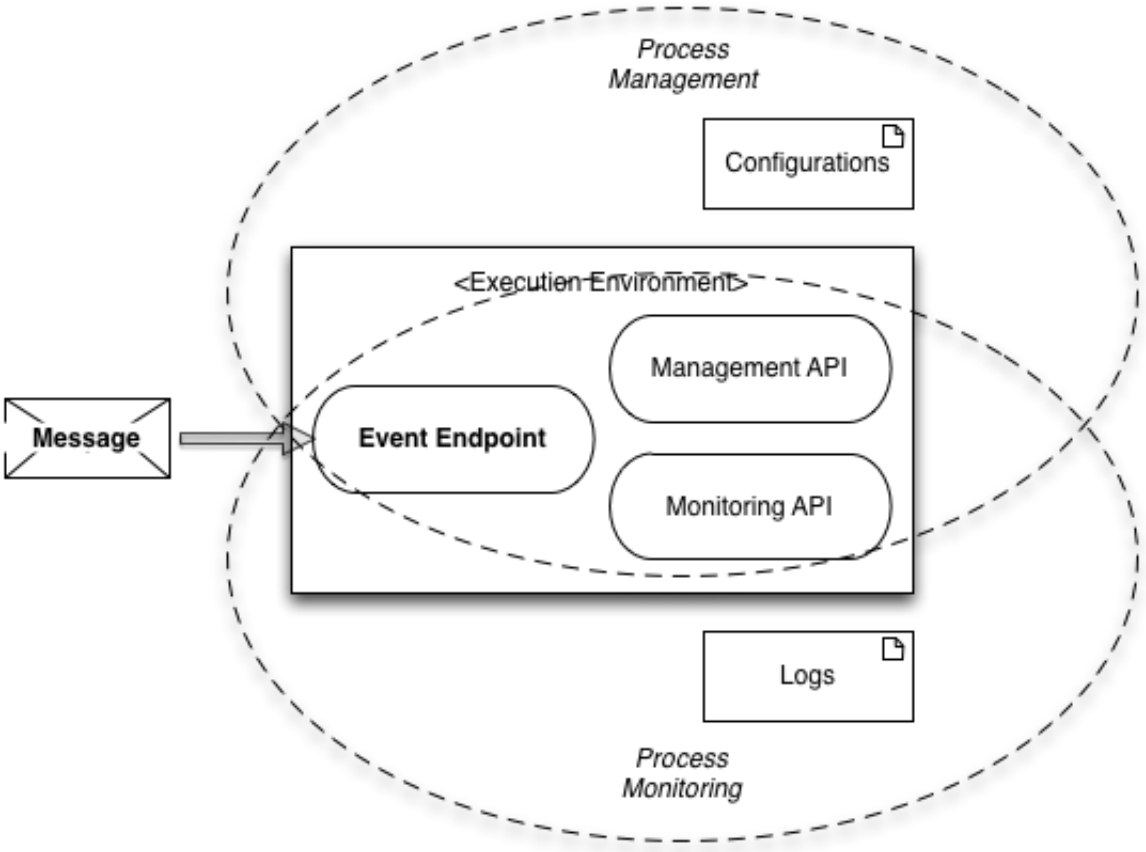


*Figure 1. Event-driven Microservices*

Intelligent reactive endpoints should implement the Tolerant Reader Pattern (http://martinfowler.com/bliki/TolerantReader.html) by refraining from object serialization processing and automatic object marshalling. Rather these endpoints should parse the event text directly, using technologies such as XPath or JSONPath, or implementing the Builder Pattern that only uses what is necessary to create an a valid object. Endpoints should delegate failure processing to another endpoint, itself dedicated to implementing failure policies such as logging, retrying, and alerting.

> *"Memories! You're talking about memories!*
>
> *— Rick Deckard*
> *Blade Runner*

Complicated *non-linear* enterprise workflows can be implemented by choreographing interactions of intelligent reactive endpoints. However, monitoring and management at the individual endpoint and enterprise level is required. The entire endpoint process stack must be monitored including the physical machine, virtualization software, O/S virtual machine, and Java Virtual Machine (JVM). In addition, robust log solutions such as Splunk or Logstash are required to monitor and manage the distributed endpoint logs.

A Tracking Identifier (a.k.a. Correlation ID) should be used to support rapid endpoint problem identification and performance monitoring. The Tracking ID is a simple contract that requires an endpoint, upon reacting to event, to create and log a Globally Unique Identifier (GUID), unless the incoming message already has a GUID, in which case the endpoint simply logs the received Tracking ID. In turn, the endpoint must pass along the Tracking ID with any external call or event emission. A simple query for an individual Tracking ID, via a logging solution, will return a correlated list of distributed event processing activities.

### *Camel Processor - Tracking Identifier*

JAVA

```
/**
 * Create Tracking ID if it does not already exist
 * TrackingIdProcessor.java
 */
public void process(Exchange exchange) throws Exception {

        if (exchange.getIn().getHeader("TrackingID") != null) {

                logger.info("Existing Tracking ID: " +
                                exchange.getIn().getHeader("TrackingID"));

        } else {

                String uuid = UUID.randomUUID().toString();

                exchange.getIn().setHeader("TrackingID", uuid);

                logger.info("Created Tracking ID: " + uuid);
        }
}
```

There are cases where just one faulty or compromised endpoint can have a disastrous impact on the entire enterprise. Proper endpoint monitoring provides awareness but endpoint management is required to react to those situations. Endpoints that emit messages or make external calls should implement the Circuit Breaker Pattern (http://martinfowler.com/bliki/CircuitBreaker.html) allowing those endpoints to be configured to throttle or suspend processing in the case of repeated failure. Endpoints should also expose a management interface, for example via the Java Management Extension (JMX), or be able to process special *command* events such as suspend, resume, or log health status.

All the monitoring and management data is then aggregated to command console to support technical operations. This console view provides a holistic view of the overall health and visually alerts operators to any endpoint issues. Furthermore, operators can drill down to an individual endpoint and use the Tracking ID to aggregate any upstream or downstream related activities. Finally, the operator can remediate any specific endpoint issue by using the management interfaces or issues a *command* event.

> *I need ya, Deck. This is a bad one, the worst yet. I need the old blade runner, I need your magic.*
>
> — *Harry Bryant*
> *Blade Runner*

Apache Camel is a lightweight Enterprise Integration Framework, and when combined with Spring Boot, is ideal solution for rapidly building, testing, and deploying Event-driven Microservices. Camel provides several Domain Specific Languages (DSL), Spring XML, Java, and Scala, for implementing endpoint processing (a.k.a. Camel Routes). A plethora of Camel Components provide out of the box integration capabilities for processing incoming messages and then sending outgoing messages. In addition, Camel Processors, implementing Enterprise Integration Patterns, allow configurable pipeline processing of those messages.

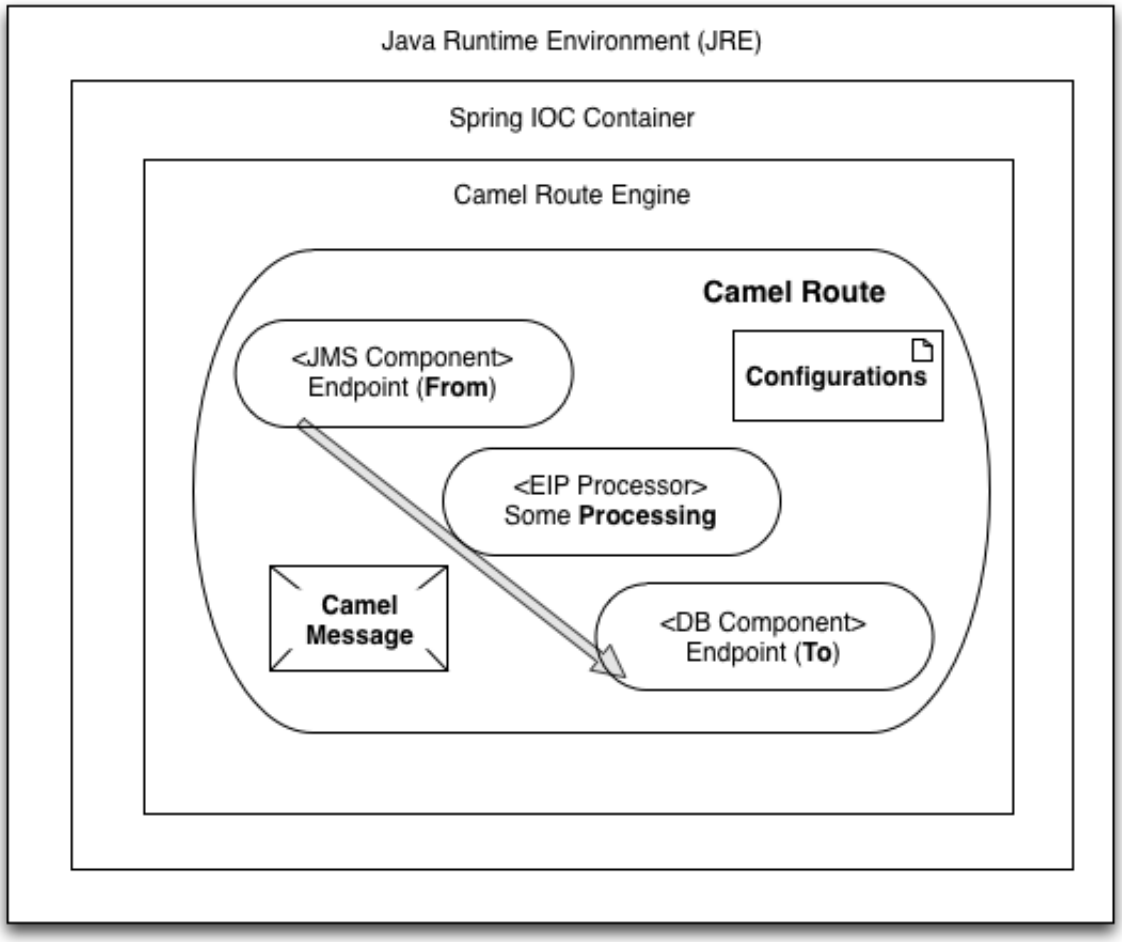


*Figure 2. Apache Camel Route Processing*

Apache Camel exposes monitoring and management interfaces via JMX and out of the box support for the Circuit Breaker Pattern. Apache Camel also provides a number of error handling mechanisms such as Dead Letter Queue (DLQ), Redelivery Policy, or even bespoke error handling. Camel Routes can use external configuration properties injected via Camel or Spring properties. Finally, a robust execution engine is used to execute Camel Route Processing.

### *Camel Route - Event Ingestion*

JAVA

```
/**
 * Content Based Routing - Inpatient-Outpatient Events
 * EventIngestionRouteBuilder.java
 */
from("activemq:event.ingestion").
process(new TrackingIdProcessor()).
choice().
        when().simple("${in.body} contains 'inpatient'").
                to("activemq:event.inpatient").
        otherwise().
                to("activemq:event.outpatient").
        end().
to("activemq:event.audit");
```

Revisiting the healthcare admission example, an HL7 FHIR Encounter (JSON) is received and processed as an event. The Ingestion endpoint uses the Content Based Routing Pattern (http://www.enterpriseintegrationpatterns.com/ContentBasedRouter.html) to determine if the event should be published to the Inpatient queue. All ingested events are also routed to an Event Repository to support audit reporting and possible replay of events. A Dead Letter Queue (DLQ) policy, a.k.a. Dead Letter Channel (http://www.enterpriseintegrationpatterns.com/DeadLetterChannel.html), is put in place to handle any processing failures.

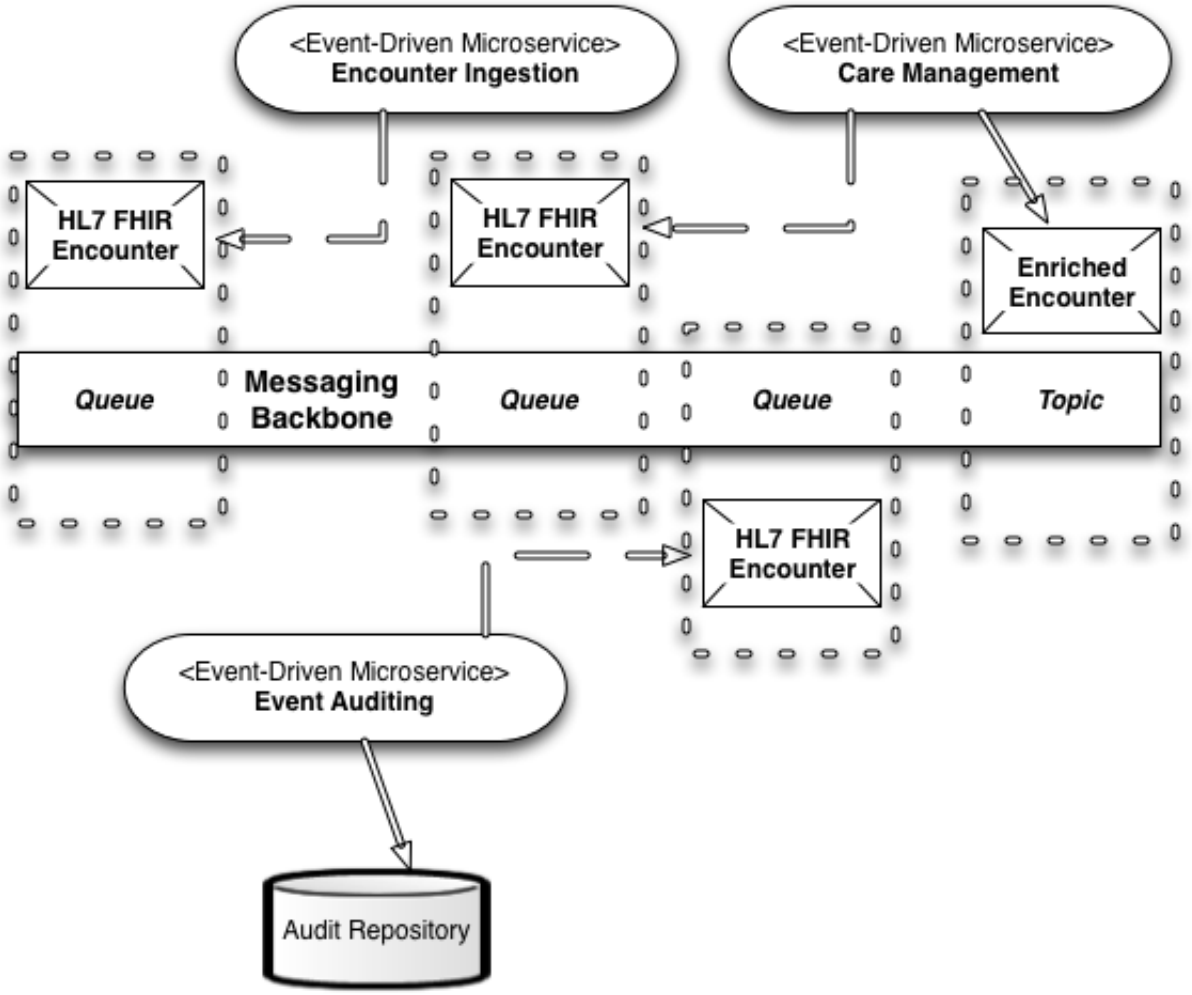


*Figure 3. HL7 FHIR Encounter Choreography*

A Care Management endpoint evaluates events from the Inpatient queue to see if there is a match against their patient roster and if so assigns a Case Number and queues up the event for further processing (Content Enrichment (http://www.enterpriseintegrationpatterns.com/DataEnricher.html)). Otherwise the endpoint logs the message as a non-match. Because the match logic is temperamental the endpoint implements a retry policy before sending a message to DLQ.

The Wire Tap Pattern (http://www.enterpriseintegrationpatterns.com/WireTap.html) is used to extend this business processes without interrupting existing workflows. Complex Event Processing (CEP) can then be applied to determine if the Encounter is fraudulent. If positive, the CEP endpoint publishes a new Finance Alert event allowing subscribing endpoints to react. Otherwise the event is ignored. In general, not all events require a reaction from an endpoint.

## *Camel Routes - Event Wire-Tap and Complex Event Processing*

JAVA

```
/**
 * Audit Storage (Mock) - Normally would use something like Elasticsearch
 * EventAuditingRouteBuilder.java
 */
from("activemq:event.audit").
        process(new TrackingIdProcessor()).
        wireTap("activemq:event.cep").
        to("file:target/events?fileName=event-${in.header.TrackingID}.json");


/**
 * Complex Event Processing (CEP) - Check fradulent paitent list
 * EventCEPRouteBuilder.java
 */
from("activemq:event.cep").
process(new TrackingIdProcessor()).
process(new Processor() {
        public void process(Exchange exchange) {

                String json = (String) exchange.getIn().getBody();
                String patient = JsonPath.read(json,"$.patient.display");

                if (patients.contains(patient)) {
                        exchange.getIn().setHeader("Fraud", "true");
                } else {
                        exchange.getIn().setHeader("Fraud", "false");
                }
        }
}).
choice().
        when().simple("${in.header.Fraud} contains 'true'").
                to("activemq:topic:event.fraud.alert").
        otherwise().
                log("...off into the ether");
```

The complete Event-driven Microservice Demo (https://github.com/jtdeane/event-driven-microservices) is available on GitHub. The repo includes the individual microservice projects, a test harness, and execution instructions.

> *He was wrong. Tyrell had told me Rachael was special. No termination date. I didn't know how long we had together… Who does?*
>
> *— Rick Deckard*
> *Blade Runner*

The jury is still out whether Microservices will become the predominant architectural style. Microservices require a higher-level commitment to test and deployment automation, as well as endpoint monitoring and management, than traditional monolithic application development. Choreographing Microservices is also quite challenging. In addition, the process of decomposing monolithic applications into Microservices is not only difficult but can be contentious and even political. For all those reasons starting with Event-driven Microservices may a good first step for most organizations.

Event-driven Architecture (EDA) is extremely powerful with a myriad of mature open source implementation technologies and frameworks. Using the Microservices Architectural Style to implement intelligent reactive endpoints allows for incremental EDA adoption, with the added benefit of failing fast or better yet accruing immediate value, without a costly up-front investment. Furthermore, very little refactoring of existing applications is required to extract value from the events *already* occurring in the enterprise.

Last updated 2015-05-30 12:19:00 EDT