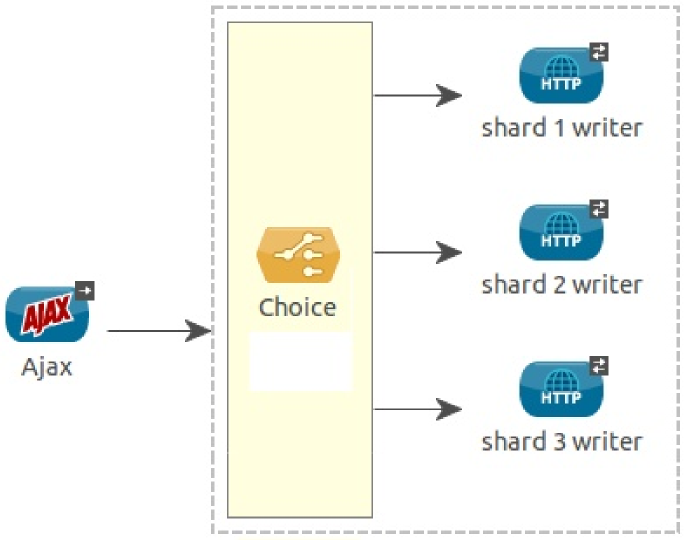
Taming the Single Writer Principle

Mule versus Apache Camel

published September 2013
By Glenn Engstrand

What is the single writer principle? When trying to build a highly scalable system the single biggest limitation on scalability is having multiple writers contend for any item of data or resource. If you can design a system that guarantees that any single item can only be updated from a single writer, then you eliminate a lot of complexity trying to enforce mutual exclusion or implementing optimistic concurrency.

We will be comparing Mule and Camel by implementing a shard aware content based router to enforce the single writer principle.

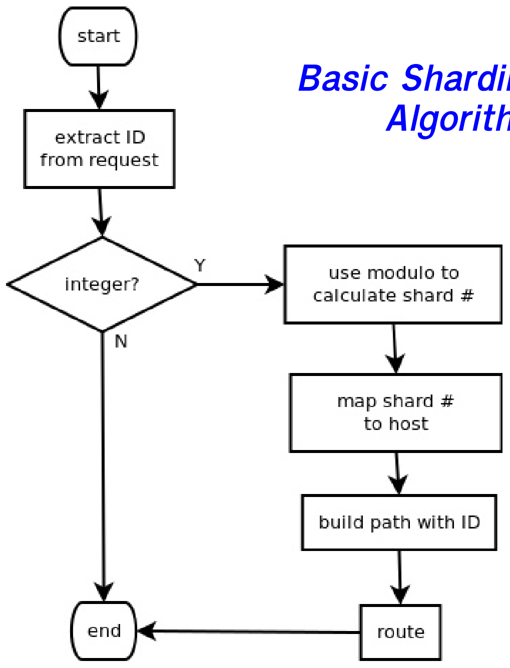


Content Based Router

The single writer principle is effective only in the case of non-idempotent transactions. If your transactions are idempotent, then it is not necessary to enforce the single writer principle. For any transaction to be idempotent, it could be repeated or replayed in any order and not invalidate the outcome. Adding columns to a wide row in cassandra is usually idempotent. Incrementing a column in a relational database is usually not.

What is a content based router? It is a particular Enterprise Integration Pattern that examines the message content and routes the message onto a different channel based on data contained in the message. The routing can be based on a number of criteria such as existence of fields, specific field values etc.

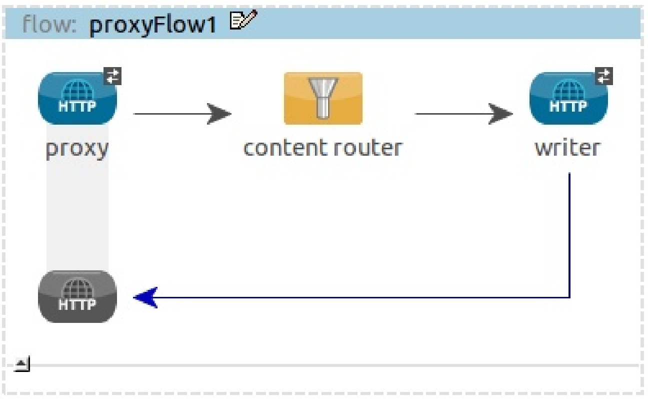
Basic Sharding Algorithm



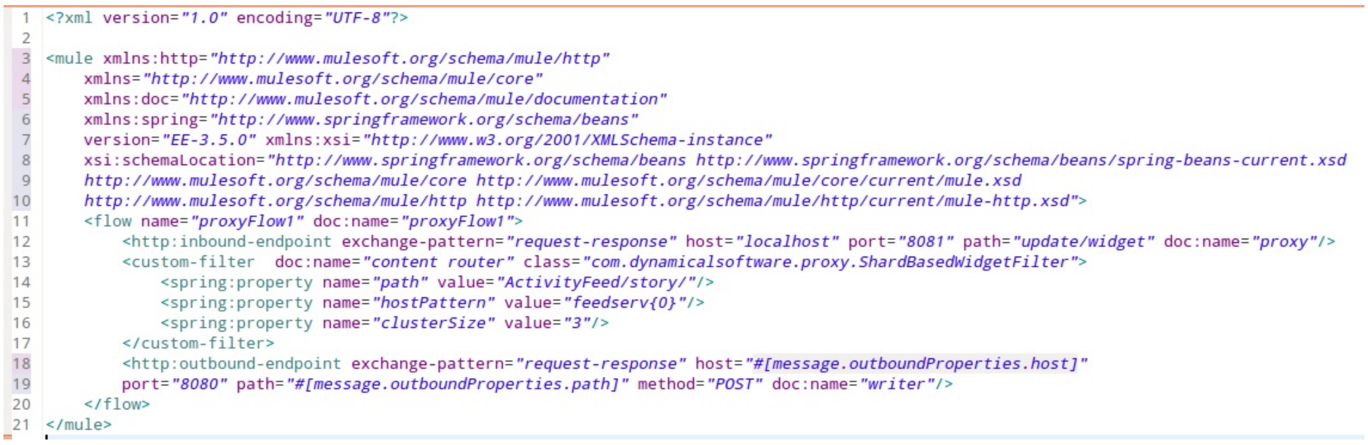
What is sharding? It is a particular partitioning strategy that supports horizontal scaling. It is usually talked about in the context of splitting up a lot of data into multiple relational databases, all with the same schema. We will use a sharding algorithm in the design of our content based router because it guarantees that any single primary key value will always map to the same host.

Here you will learn about two similar open source projects, Mule and Camel, in their different approaches to implementing a proxy to enforce the single writer principle.

In Mule, you define a series of flows which describe routes between endpoints. Inbound endpoints articulate how messages flow into your proxy. These messages route around until they are forwarded on to outbound endpoints. These endpoints could be HTTP based (as in our example) but they could also be ASMQ, JMS, SMTP, etc. These flows are captured as XML and there is an Eclipse plugin that permits you to visually design them. HTTP endpoints depend on the specification of host, port, path, and method.

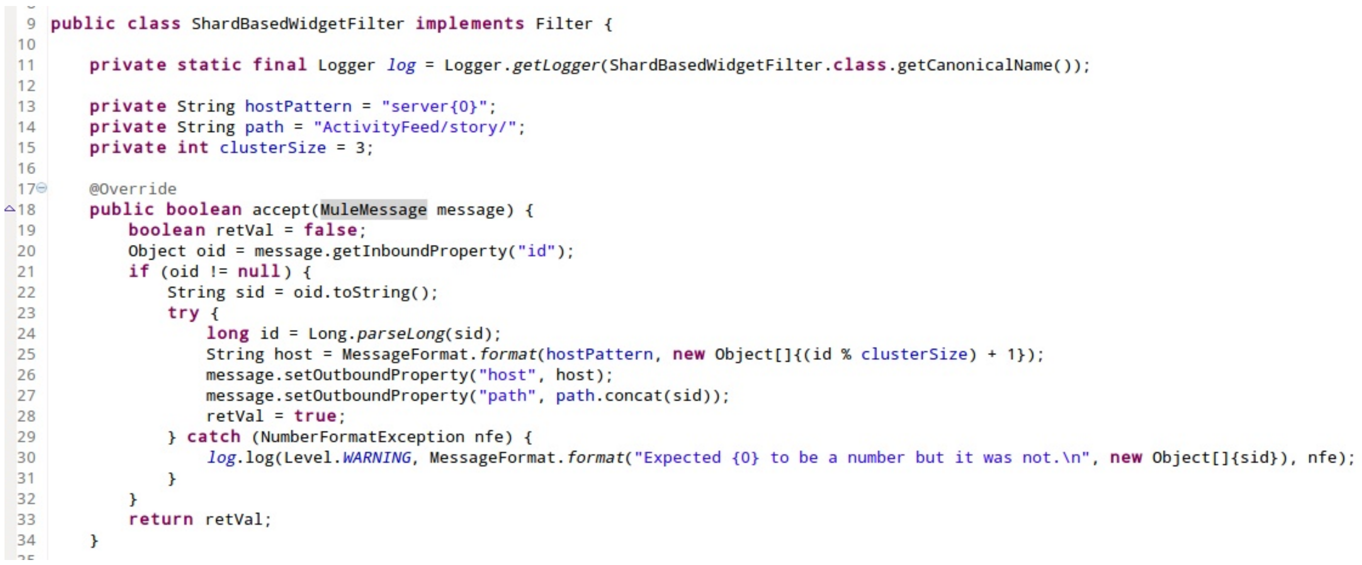


Content Based Routing the Mule Way



```
<?xml version="1.0" encoding="UTF-8"?>

<mule xmlns:http="http://www.mulesoft.org/schema/mule/http"
    xmlns="http://www.mulesoft.org/schema/mule/core"
    xmlns:doc="http://www.mulesoft.org/schema/mule/documentation"
    xmlns:spring="http://www.springframework.org/schema/beans"
    version="EE-3.5.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-current.xsd
    http://www.mulesoft.org/schema/mule/core http://www.mulesoft.org/schema/mule/core/current/mule.xsd
    http://www.mulesoft.org/schema/mule/http http://www.mulesoft.org/schema/mule/http/current/mule-http.xsd">
    <flow name="proxyFlow1" doc:name="proxyFlow1">
        <http:inbound-endpoint exchange-pattern="request-response" host="localhost" port="8081" path="update/widget" doc:name="proxy"/>
        <custom-filter  doc:name="content router" class="com.dynamicalsoftware.proxy.ShardBasedWidgetFilter">
            <spring:property name="path" value="ActivityFeed/story/"/>
            <spring:property name="hostPattern" value="feedserv{0}"/>
            <spring:property name="clusterSize" value="3"/>
        </custom-filter>
        <http:outbound-endpoint exchange-pattern="request-response" host="#[message.outboundProperties.host]"
        port="8080" path="#[message.outboundProperties.path]" method="POST" doc:name="writer"/>
    </flow>
</mule>
```

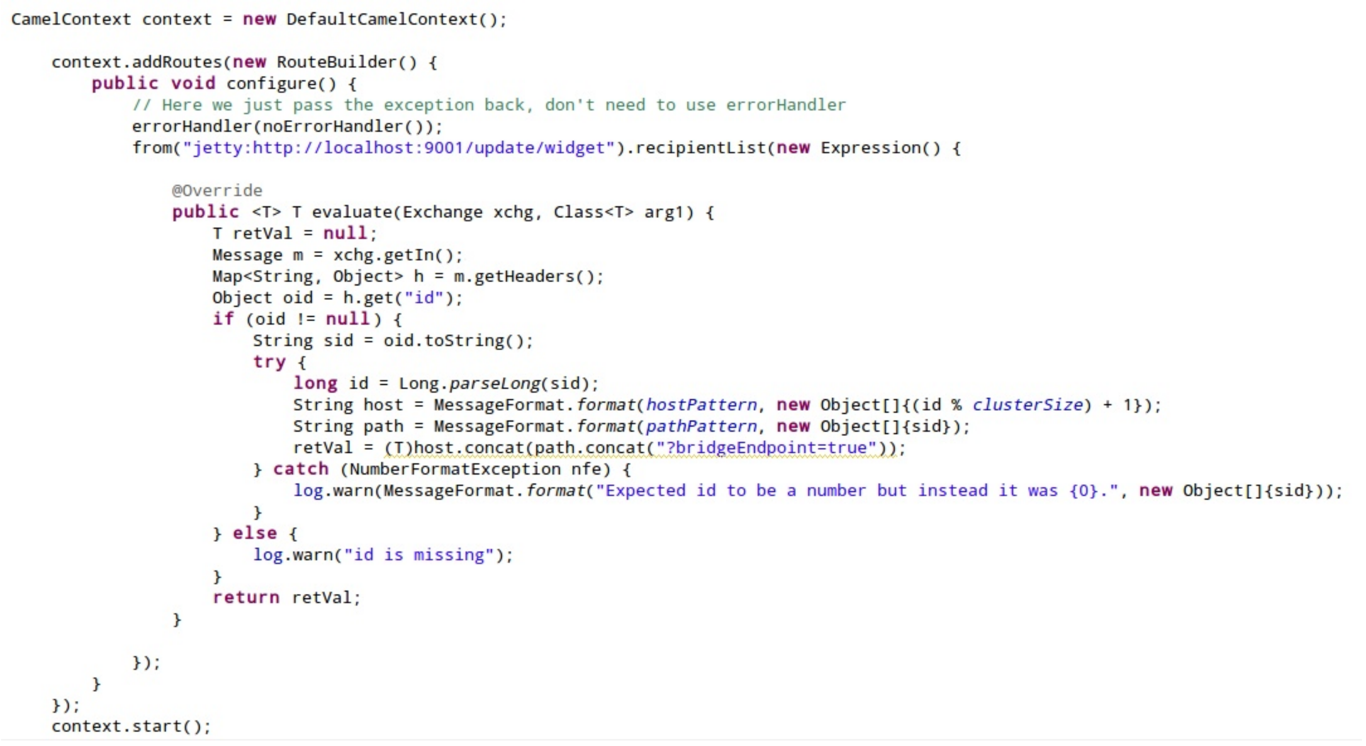
Our Mule implementation of the shard aware content based router is a custom filter where the inbound properties of each mule message are queried and used to set the outbound properties. The identifier for the entity to be edited is extracted and passed on as a part of the path. That identifier is converted to a number and modulo with the cluster size to map to which server will serve as the single writer. Cluster size and host pattern are spring injected properties of the custom filter.



```
public class ShardBasedWidgetFilter implements Filter {

    private static final Logger log = Logger.getLogger(ShardBasedWidgetFilter.class.getCanonicalName());

    private String hostPattern = "server{0}";
    private String path = "ActivityFeed/story/";
    private int clusterSize = 3;

    @Override
    public boolean accept(MuleMessage message) {
        boolean retVal = false;
        Object oid = message.getInboundProperty("id");
        if (oid != null) {
            String sid = oid.toString();
            try {
                long id = Long.parseLong(sid);
                String host = MessageFormat.format(hostPattern, new Object[]{(id % clusterSize) + 1});
                message.setOutboundProperty("host", host);
                message.setOutboundProperty("path", path.concat(sid));
                retVal = true;
            } catch (NumberFormatException nfe) {
                log.log(Level.WARNING, MessageFormat.format("Expected {0} to be a number but it was not.\n", new Object[]{sid}), nfe);
            }
        }
        return retVal;
    }
```

Content Based Routing the Camel Way

Flows in Camel can be described in many different ways. In this example, we will be using what is known as the Java DSL approach to describing our flow. This means creating a Camel context and associating with that context a route builder that dynamically maps the inbound endpoint, handled by the camel jetty plugin, to a recipient list (for us, always 1) of outbound endpoints. The route builder gets a reference to the exchange itself from which it can access the message. In Camel, you access the identifier to be updated through the message headers. You have to include this bridge endpoint attribute in the query string of the outbound endpoint URL.



```java
CamelContext context = new DefaultCamelContext();

    context.addRoutes(new RouteBuilder() {
        public void configure() {
            // Here we just pass the exception back, don't need to use errorHandler
            errorHandler(noErrorHandler());
            from("jetty:http://localhost:9001/update/widget").recipientList(new Expression() {

                @Override
                public <T> T evaluate(Exchange xchg, Class<T> arg1) {
                    T retVal = null;
                    Message m = xchg.getIn();
                    Map<String, Object> h = m.getHeaders();
                    Object oid = h.get("id");
                    if (oid != null) {
                        String sid = oid.toString();
                        try {
                            long id = Long.parseLong(sid);
                            String host = MessageFormat.format(hostPattern, new Object[]{(id % clusterSize) + 1});
                            String path = MessageFormat.format(pathPattern, new Object[]{sid});
                            retVal = (T)host.concat(path.concat("?bridgeEndpoint=true"));
                        } catch (NumberFormatException nfe) {
                            log.warn(MessageFormat.format("Expected id to be a number but instead it was {0}.", new Object[]{sid}));
                        }
                    } else {
                        log.warn("id is missing");
                    }
                    return retVal;
                }

            });
        }
    });
    context.start();
```

The similarities I found in both Mule and Camel during this project is that both approaches are flow oriented and very compatible with EIP. They are both pretty heavy weight with lots of software dependencies and cognitive overheard. They both use Jetty.

The most obvious difference is that Mule has this cool visual designer and Camel does not. I have heard that Camel has a visual designer somewhere in Jboss Fuse but I was never able to find it. What I found favorable for Mule was that its abstractions felt a little more natural to me. No need for this weird bridge endpoint attribute. What I liked about Camel was that inbound message attributes would be automatically transferred to the outbound message. You had to code only what was different. With Camel, a POST gets transferred as a POST and a GET gets transferred as a GET. With Mule, you have to specify the method (POST or GET) at design time.

At one time, Jetty was considered a developer only web server and not ready for production load. Within the past year, I have found some companies using Jetty in production. I guess that shouldn't be a big surprise. Back in the 90's, web sphere, jrun, and resin were considered production grade and tomcat was not. Now, many companies use tomcat in large load production environments.