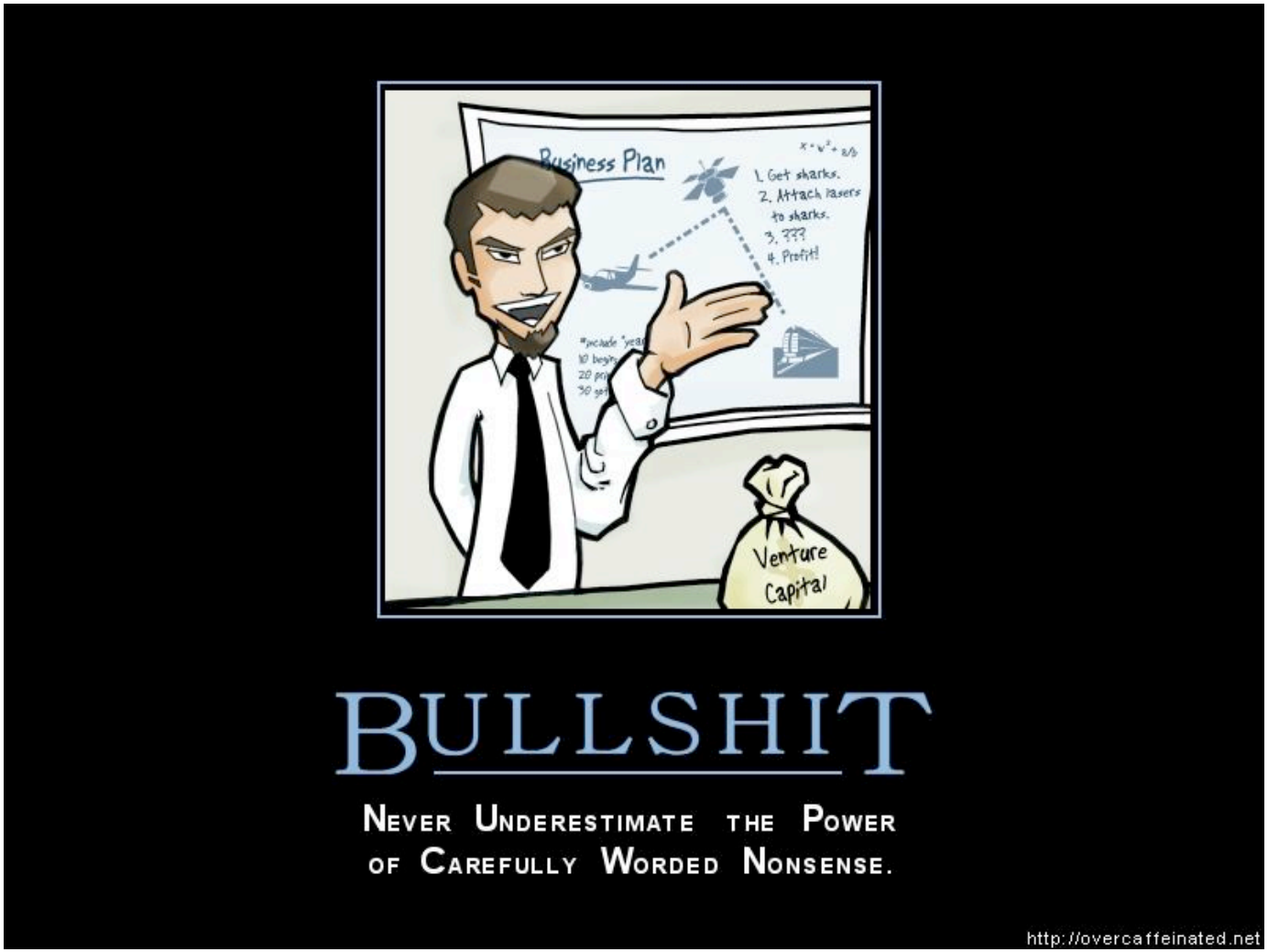
MuleSource

the open source choice for SOA infrastructure

Lightweight RESTful Integration with Mule

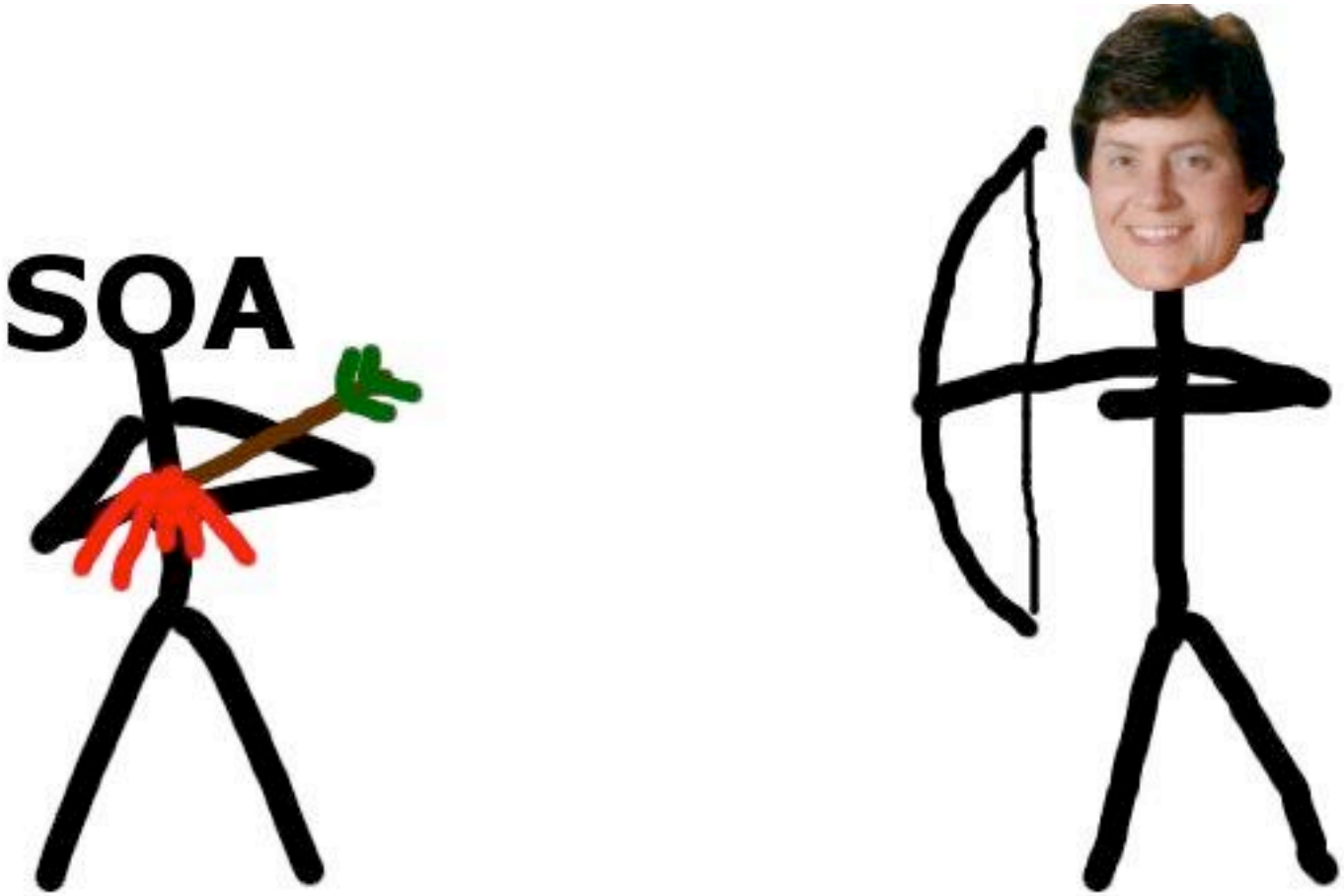
Dan Diephouse

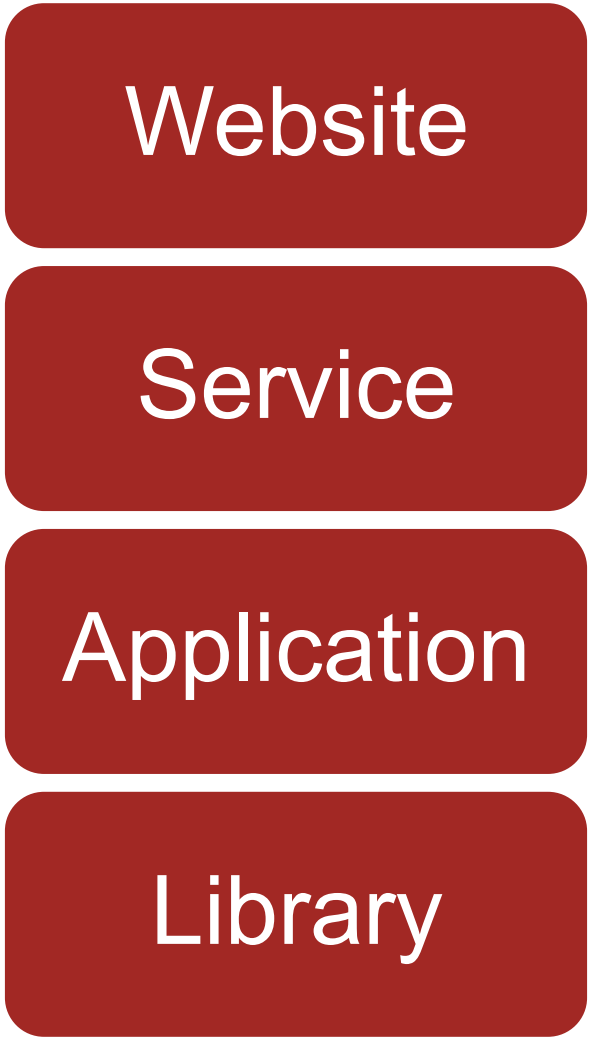
SOA defined



SOA is dead?

MuleSource





Crap to deal with when building services

MuleSource

- Versioning
- Coupling
- Interoperability
- Scalability
- Infrastructure

REST: Loosely coupled

MuleSource

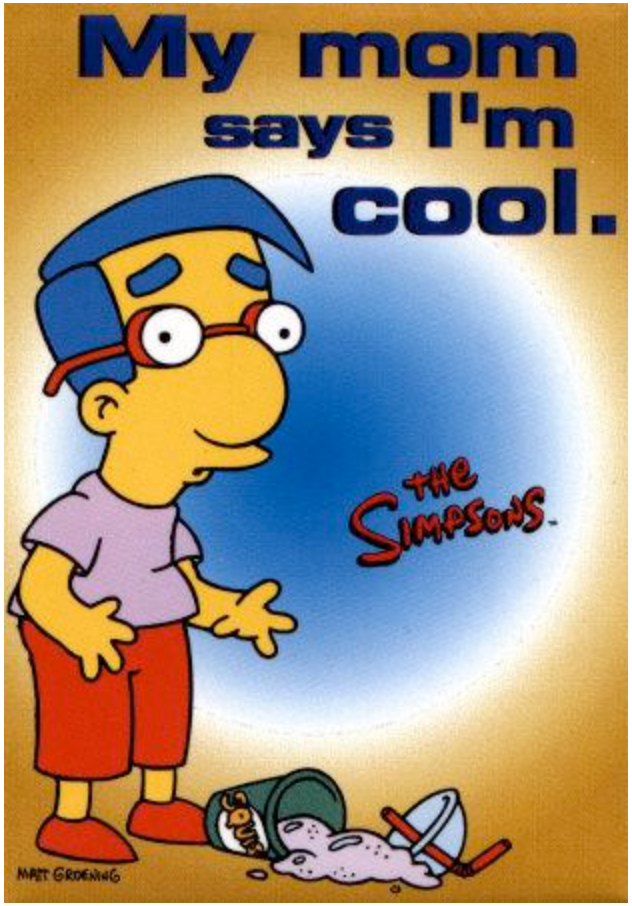
REST: Interoperability

MuleSource

REST: Scalability

MuleSource

So RESTful services are cool….



The magical mystery bus!

MuleSource

Why do developers choose Mule?

MuleSource

No prescribed message format

- XML, CSV, Binary, Streams, Record, Java Objects
- Mix and match

Zero code intrusion

- Mule does not impose an API on service objects
- Objects are fully portable

Existing objects can be managed

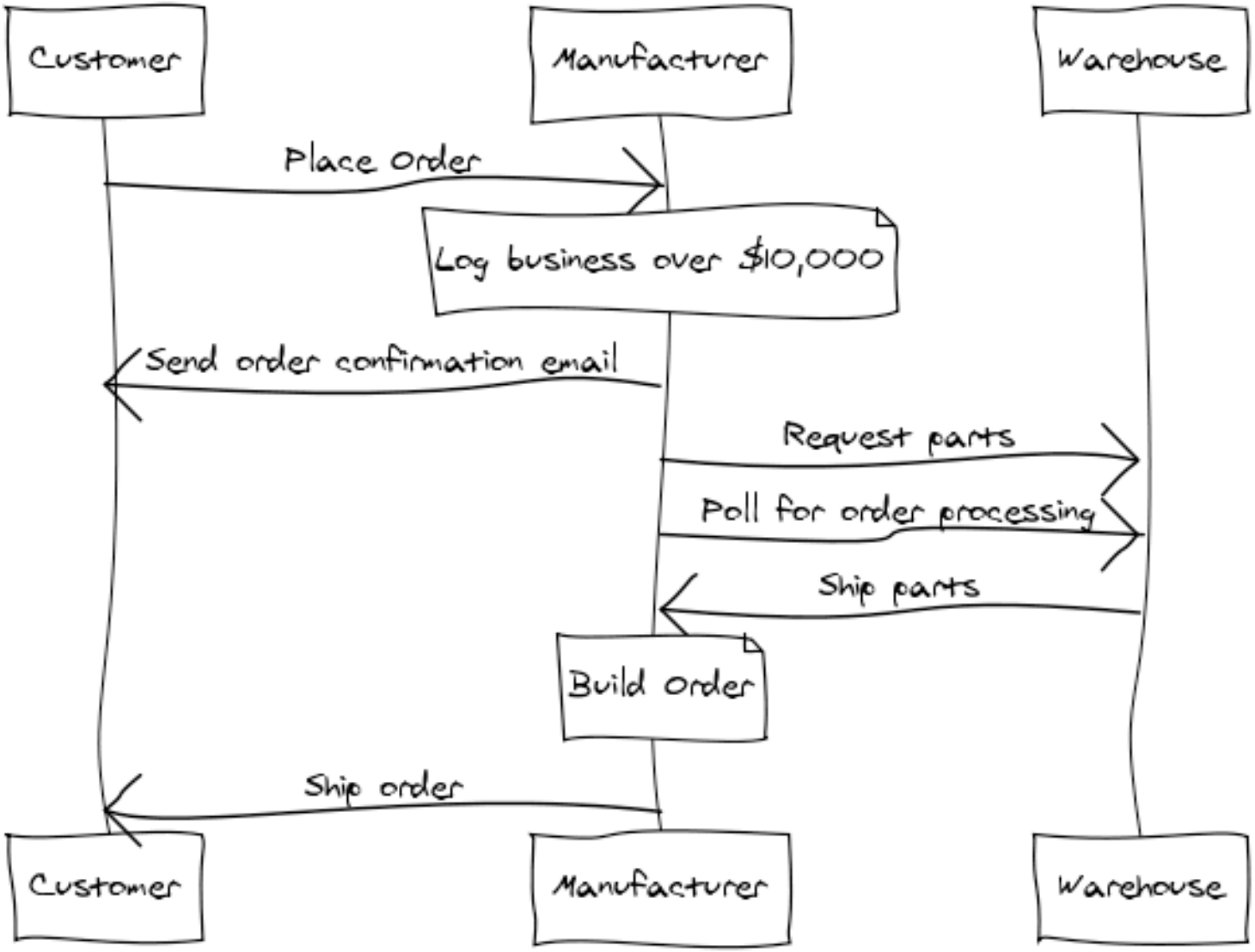
- POJOs, IoC Objects, EJB Session Beans, Remote Objects
- REST & Web Services

Easy to test

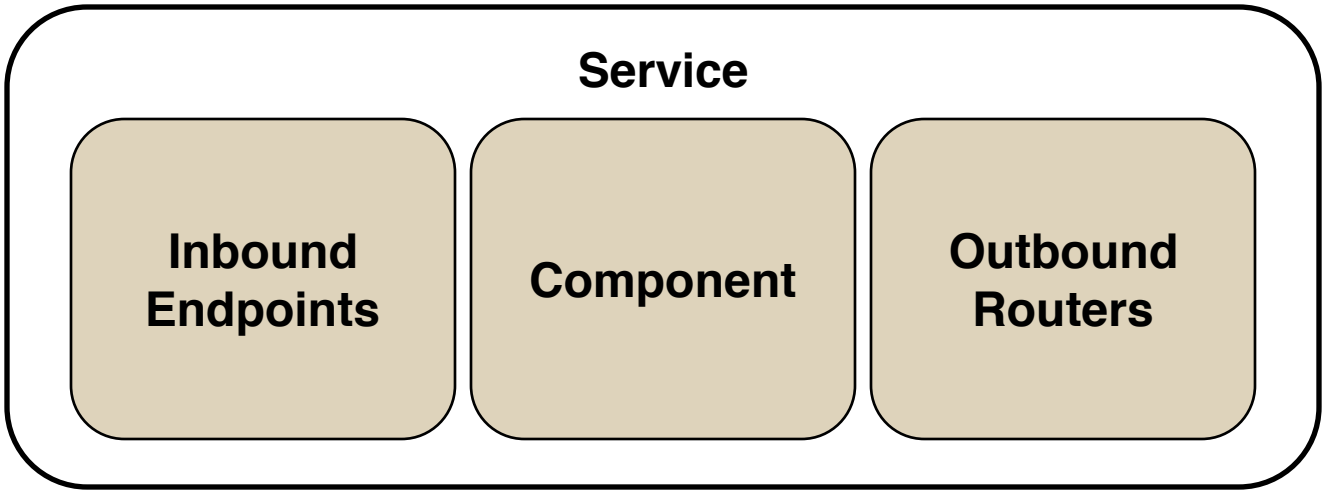
- Mule can be run easily from a JUnit test case
- Framework provides a Test compatibility kit
- Scales down as well as up

Mule Manufacturing Co.

MuleSource



Mule Services



Core Concepts: Connectors

MuleSource

- HTTP
- JMS
- FTP
- TCP
- UDP
- LDAP
- BPM
- CXF
- File
- POP
- SMTP
- Multicast
- VM
- XMPP
- WSDL
- STDIO

Core Concepts: Endpoints

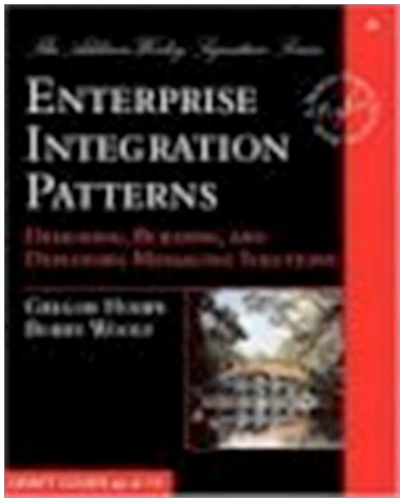
MuleSource

- Used to connect components and external systems together
- Endpoints use a URI for Addressing
- Can have transformer, transaction, filter, security and meta-information associated
- Two types of URI
  - **scheme://[username][:password]@[host][:port]?[params]**
    - **smtp://ross:pass@localhost:25**
  - **scheme://[address]?[params]**
    - **jms://my.queue?persistent=true**

Core Concepts: Routers

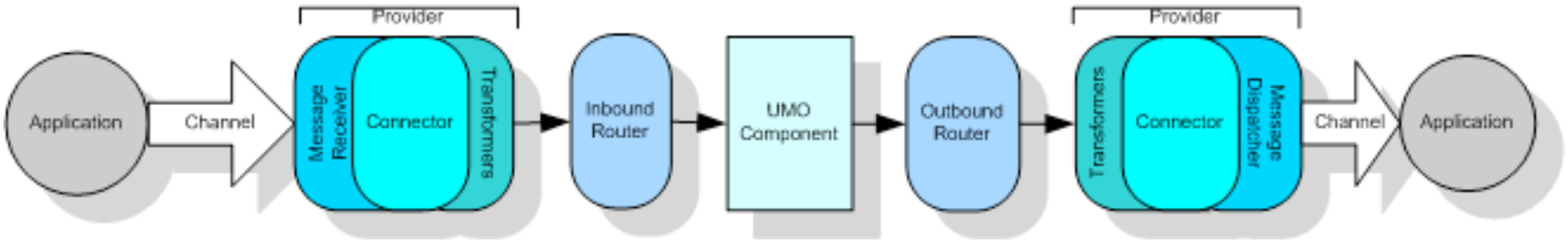
MuleSource

- Control how events are sent and received
- Can model all routing patterns defined in the EIP Book
- **Inbound Routers**
  - Idempotency
  - Selective Consumers
  - Re-sequencing
  - Message aggregation
- **Outbound Routers**
  - Message splitting / Chunking
  - Content-based Routing
  - Broadcasting
  - Rules-based routing
  - Load Balancing



Core Concepts: Transformers

MuleSource



Transformers

- Converts data from one format to another
- Can be chained together to form transformation pipelines

```
<jms:object-to-jms name="XmlToJms"/>

<custom-transformer name="CobolXmlToBusXml"
  class="com.myco.trans.CobolXmlToBusXml"/>

<endpoint address="jms://trades"
        transformers="CobolXmlToBusXml, XmlToJms"/>
```

Simple example

MuleSource

```
<service name="webServices">
  <inbound>
    <inbound-endpoint address="http://localhost:63080"
                      synchronous="true" />
  </inbound>
  <outbound>
    <pass-through-router>
      <outbound-endpoint address="vm://orderQueue"/>
    </pass-through-router>
  </outbound>
</service>
```

Simple example

MuleSource

```
<service name="webServices">
  <inbound>
    <inbound-endpoint address="vm://orderQueue"
                      synchronous="true"
                      transformer-refs="CsvToOrderPojo"/>
  </inbound>
  <component class="…MyOrderService"/>
</service>
```

BUILDING SERVICES

Jersey

MuleSource

- Annotations to expose your classes as a RESTful service
- Implements the JAX-RS (JSR311) specification
- Mule connector makes it possible to embed JAX-RS services in Mule

```
@Path("/helloworld")
public class HelloWorldResource {
    @GET
    @ProduceMime("text/plain")
    public String sayHelloWorld() {
        return "Hello World";
    }
}
```

```
@POST
@Produces("application/xml")
@Consumes("application/xml")
public Response placeOrder(Order order) {
    int number = getNextOrderNumber();
…
    URI location = uriInfo.getAbsolutePath()
        .resolve("/orders/" + number);
    return Response.created(location)
                   .entity(order)
                   .build();
}
```

```
@GET
@Produces("application/xml")
@Path("/{id}")
public Order getOrder(@PathParam("id") int id) {
    return orders.get(id);
}
```

Simple Service Configuration

MuleSource

```
<service name="orderResource">
  <inbound>
    <inbound-endpoint
      address="jersey:http://localhost:63081/orders"/>
  </inbound>
  <component
    class="org.mule.examples.mfg.OrderResource"/>
</service>
```

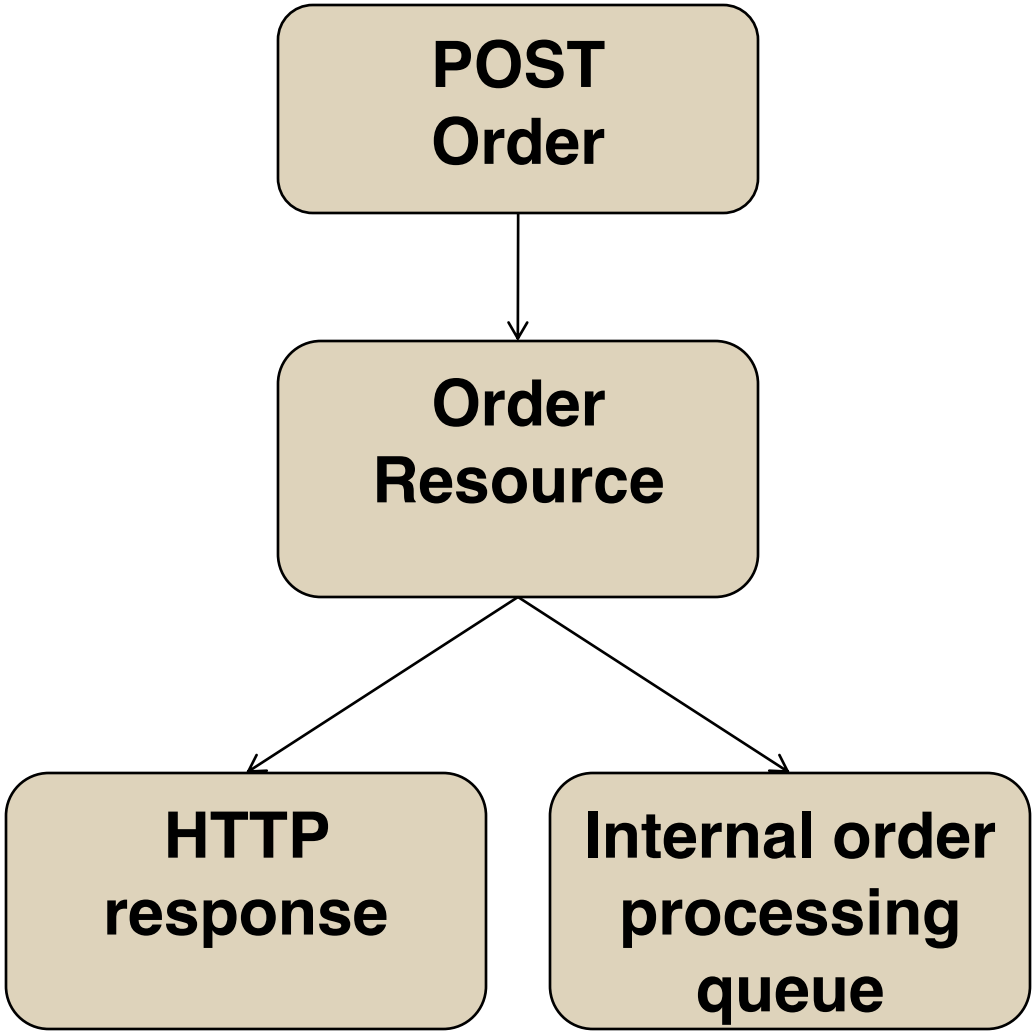
Why Jersey?

MuleSource

- Very easy to build RESTful services inside Mule
- Built in serialization support for
  - XML via JAXB
  - JSON
  - Images
  - Easily write your own serializers

INTEGRATING REST INTO YOUR MESSAGING LAYER

Process flow



```
<service>
…
<!-- Is this a new order? If so, initiate the backend
   processing -->
<filtering-router>
  <outbound-endpoint address="vm://processOrder"/>
  <and-filter>
    <restlet:uri-template-filter
        verbs="POST" pattern="/orders"/>
    <expression-filter
        evaluator="header" expression="http.status=201" />
  </and-filter>
</filtering-router>
</service>
```

Another example: URI template routing

MuleSource

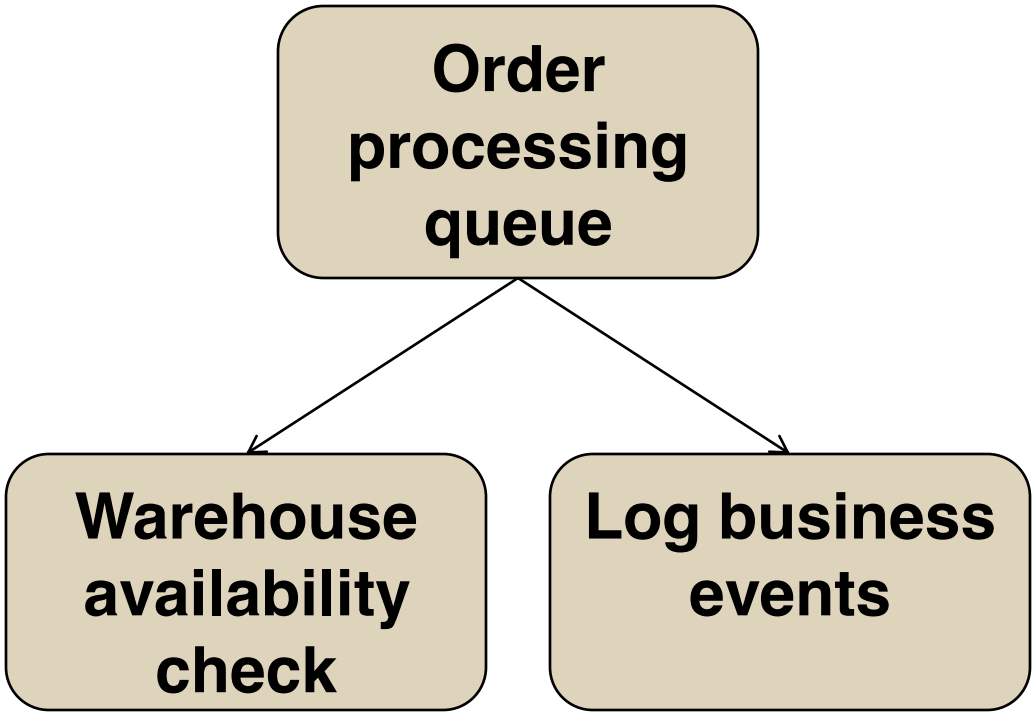
```
<service name="webServices">
  <inbound>
    <inbound-endpoint address="http://localhost:63080"
                      synchronous="true" />
  </inbound>
  <outbound>
    <filtering-router>
      <outbound-endpoint address="vm://orderQueue"/>
      <restlet:uri-template-filter pattern="/orders/{orderId}"/>
    </filtering-router>
    <filtering-router>
      <outbound-endpoint address="vm://userQueue"/>
      <restlet:uri-template-filter pattern="/users" />
    </filtering-router>
  </outbound>
</service>
```

Modify messages while filtering!

MuleSource

```
<restlet:uri-template-filter
   pattern="/orderId/{set-header.orderId}" />
<restlet:uri-template-filter
   pattern="/orderId/{set-payload.orderId}" />
```

Order Processing



Order Processing

MuleSource

```
<service name="orderProcessing">
  <inbound>
    <inbound-endpoint address="vm://processOrder" synchronous="true">
      <expression-transformer>
        <return-argument evaluator="header" expression="jersey.response" />
      </expression-transformer>
      <expression-transformer>
        <return-argument evaluator="groovy" expression="payload.entity" />
      </expression-transformer>
    </inbound-endpoint>
  </inbound>
  <outbound> … </outbound>
</service>
```

Order Processing

MuleSource

```
<service name="orderProcessing">
  <inbound>…</inbound>
  <outbound>
    <chaining-router>
      <outbound-endpoint address="vm://warehouseService"/>

      <!-- Post this to the AtomPub event log -->
      <outbound-endpoint
         address="http://localhost:9002/atompub" ... >

    </chaining-router>
  </outbound>
</service>
```

Warehousing

MuleSource

```
<service name="warehouseService">
  <inbound>
    <inbound-endpoint address="vm://warehouseService"/>
    <inbound-endpoint
              address="jersey:http://localhost:9002/warehouse"
              synchronous="true"/>
  </inbound>
  <component>
    <singleton-object class="…WarehouseService"/>
  </component>
</service>
```

Its just Java

MuleSource

```
public class WarehouseService {

    public void requestParts(Order order) {
…
    }
}
```

ATOMPUB

Consider using Atom Publishing Protocol

MuleSource

- Atom: a format for syndication
  - Describes “lists of related information” – a.k.a. *feeds*
  - Feeds are composed of entries
- *User Extensible*
- More generic than just *blog stuff*

Atom Publishing Protocol

MuleSource

- RESTful protocol for building services
- Create, edit, delete entries in a collection
- Extensible Protocol
  - Paging extensions
  - GData
  - Opensearch
- Properly uses HTTP so can be scalable, reliable and secure

Why you should use APP for your app

MuleSource

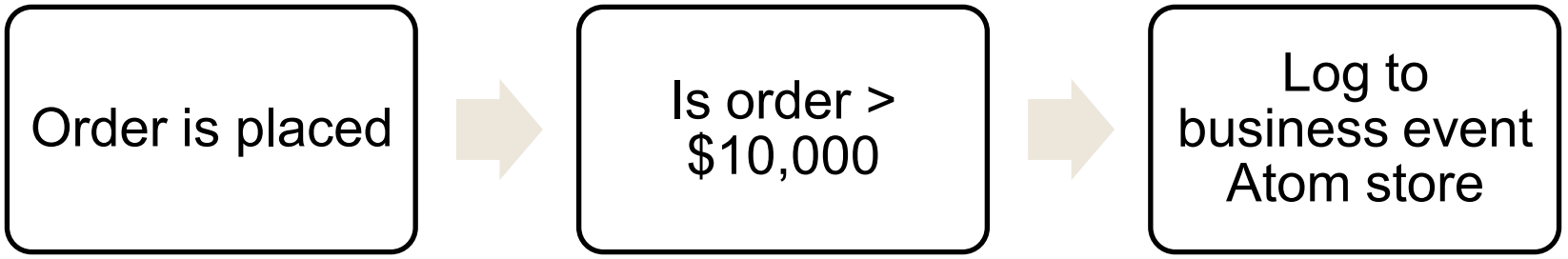
- Provides *ubiquitous elements which have meaning across all contexts*
- You can leverage existing solutions for security
  - HTTP Auth, WSSE, Google Login, XML Sig & Enc
- Eliminates the need for you to write a lot of server/client code
  - ETags, URLs, etc are all handled for you
- Integrates seamlessly with non-XML data
- There are many APP implementations and they are known to work well together

Eventing

MuleSource

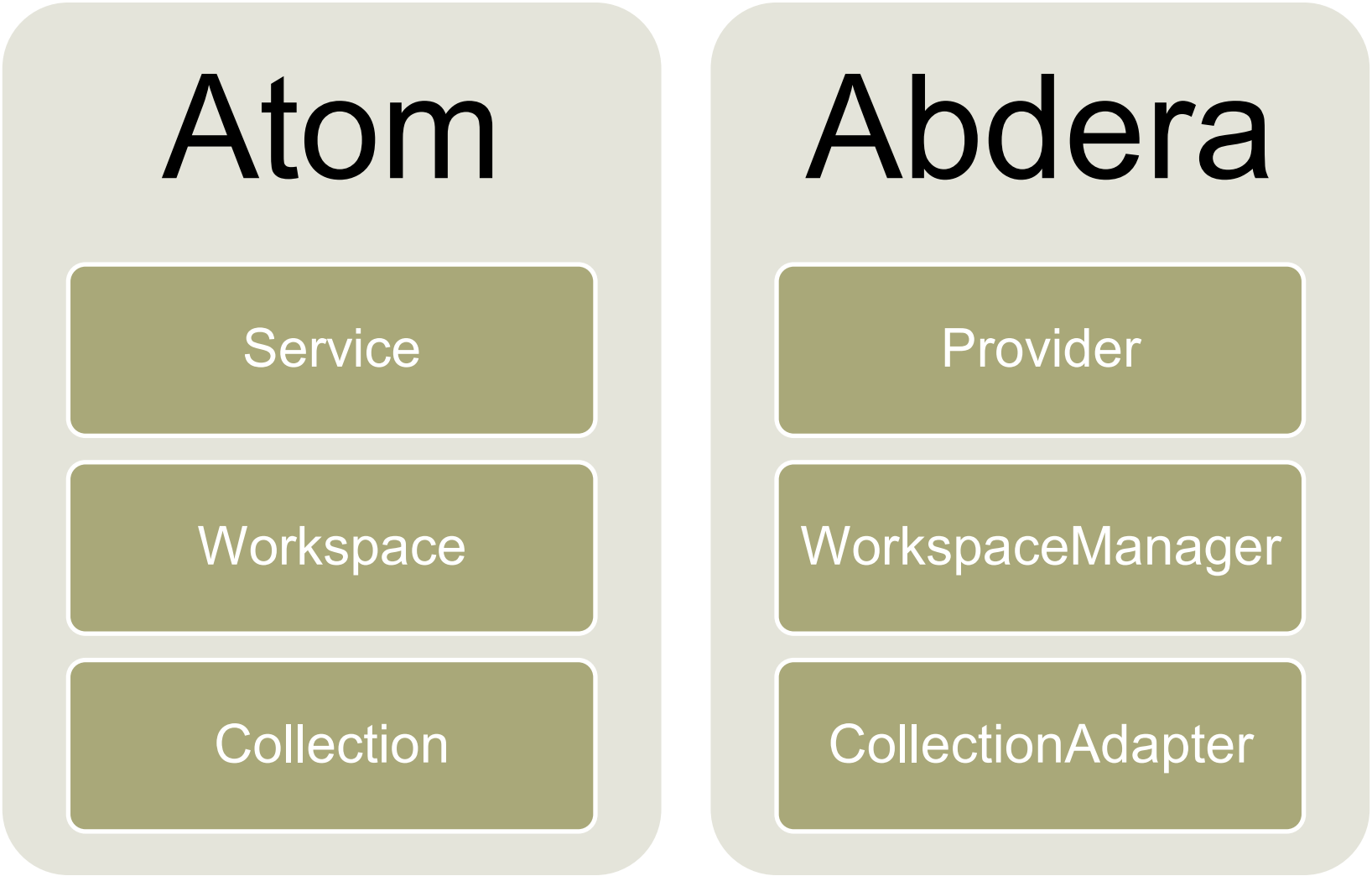
- Publish and consume entries which map to events
- Application level events
  - Exceptions/fault monitoring
- Business level events
  - A expense over $1000 was registered
- Use query parameters to narrow down the criteria
- Works with any client which understands Atom
- Powerful combination with opensearch

Process



Abdera Concepts

MuleSource



CollectionAdapters

MuleSource

- Write your own
- Built in CollectionAdapters
  - JCR
  - JDBC
  - Filesystem

```
<mule:service name="customerService">
  <mule:inbound>
    <mule:inbound-endpoint address="http://localhost:9002"
                           synchronous="true"/>
  </mule:inbound>
  <abdera:component provider-ref="abderaProvider"/>
</mule:service>
```

```
<a:provider id="abderaProvider">

  <a:workspace title="Event Workspace">
    <ref bean="jcrAdapter"/>
  </a:workspace>

</a:provider>
```

JCR Store

MuleSource

```
<bean id="jcrAdapter"
   class="…abdera.protocol.server.adapters.jcr.JcrCollectionAdapter"
   init-method="initialize">
  <property name="author" value="Mule"/>
  <property name="title" value="Event Queue"/>
  <property name="collectionNodePath" value="entries"/>
  <property name="repository" ref="jcrRepository"/>
  <property name="credentials">
    <bean class="javax.jcr.SimpleCredentials">
      <constructor-arg><value>username</value></constructor-arg>
      <constructor-arg><value>password</value></constructor-arg>
    </bean>
  </property>
  <property name="href" value="events"/>
</bean>
```

Creating Atom Entries

MuleSource

```
<!-- Post this to the AtomPub event log -->
<outbound-endpoint address="http://localhost:9002/atompub">
  <expression-filter evaluator="groovy"
                     expression="payload.price > 10000"/>
  <message-properties-transformer >
    <add-message-property key="Content-Type"
        value="application/atom+xml;type=entry"/>
  </message-properties-transformer>
  <abdera:entry-transformer …>
</outbound-endpoint>
```

Transformers

MuleSource

```
public class EntryTransformer extends
   AbstractTransformer {

      @Override
      protected Object doTransform(
            Object src, String encoding)
            throws TransformerException {
  …
      }
}
```

Transformers

MuleSource

```
Order order = (Order) src;

Entry entry = factory.newEntry();
entry.setTitle("Order for " + order.getPrice() +
   " from " + order.getCustomer());
entry.setId(factory.newUuidUri());
entry.setContent("An order was placed for " +
   order.getPrice() + " from " +
   order.getCustomer());
entry.setUpdated(new Date());
entry.addAuthor("Mule Mfg Company");

return entry;
```

XML version

MuleSource

```
<abdera:entry-builder-transformer>
    <abdera:entry-property name="author"
        evaluator="string" expression="Ross Mason"/>
    <abdera:entry-property name="content"
        evaluator="payload" expression=""/>
    <abdera:entry-property name="title"
        evaluator="header" expression="title"/>
    <abdera:entry-property name="updated"
        evaluator="function" expression="now"/>
    <abdera:entry-property name="id"
        evaluator="groovy"
        expression="'urn:uuid:' +
              java.util.UUID.randomUUID().toString()"/>
</abdera:entry-builder-transformer>
```

POLLING VS. MESSAGING

ETag Header

MuleSource

- Resources may return an ETag header when it is accessed
- On subsequent retrieval of the resource, Client sends this ETag header back
- If the resource has not changed (i.e. the ETag is the same), an empty response with a 304 code is returned
- Reduces bandwidth/latency

ETag Example

MuleSource

```
GET /feed.atom
Host: www.acme.com
…
```

Client

```
HTTP/1.1 200 OK
Date: …
ETag: "3e86-410-3596fbbc"
Content-Length: 1040
Content-Type: text/html
…
```

Server

```
GET /feed.atom
If-None-Match:
  "3e86-410-3596fbbc"
Host: www.acme.com
…
```

Client

```
HTTP/1.1 304 Not Modified
Date: …
ETag: "3e86-410-3596fbbc"
Content-Length: 0…
```

Server

LastModified Example

MuleSource

```
GET /feed.atom
Host: www.acme.com
…
```

```
GET /feed.atom
If-Modified-Since:
  Sat, 29 Oct 1994
  19:43:31 GMT
Host: www.acme.com
…
```

Client

```
HTTP/1.1 200 OK
Date: …
Last-Modified: Sat, 29 Oct
   1994 19:43:31 GMT
Content-Length: 1040
Content-Type: text/html
…
```

```
HTTP/1.1 304 Not Modified
Date: …
Last-Modified: Sat, 29 Oct
  1994 19:43:31 GMT
Content-Length: 0
```

Server

Process

Wiring with XML

MuleSource

```
<http:polling-connector
   name="PollingHttpConnector"
   pollingFrequency="30000"
   checkEtag="true"/>

<service name="eventConsumer">
   <inbound>
      <inbound-endpoint
    address="http://localhost:9002/warehouse"
    connector-ref="PollingHttpConnector"/>
   </inbound>
</service>
```

Is polling the answer?

MuleSource

Example from OSCON:

> "On July 21st, 2008, friendfeed crawled flickr 2.9 million times to get the latest photos of 45,754 users, of which 6,721 of that 45,754 *potentially* uploaded a photo."

Frequent updates are common

MuleSource

- Flickr
- Blogs
- Twitter
- Business events
- Presence on IM
- Stock data

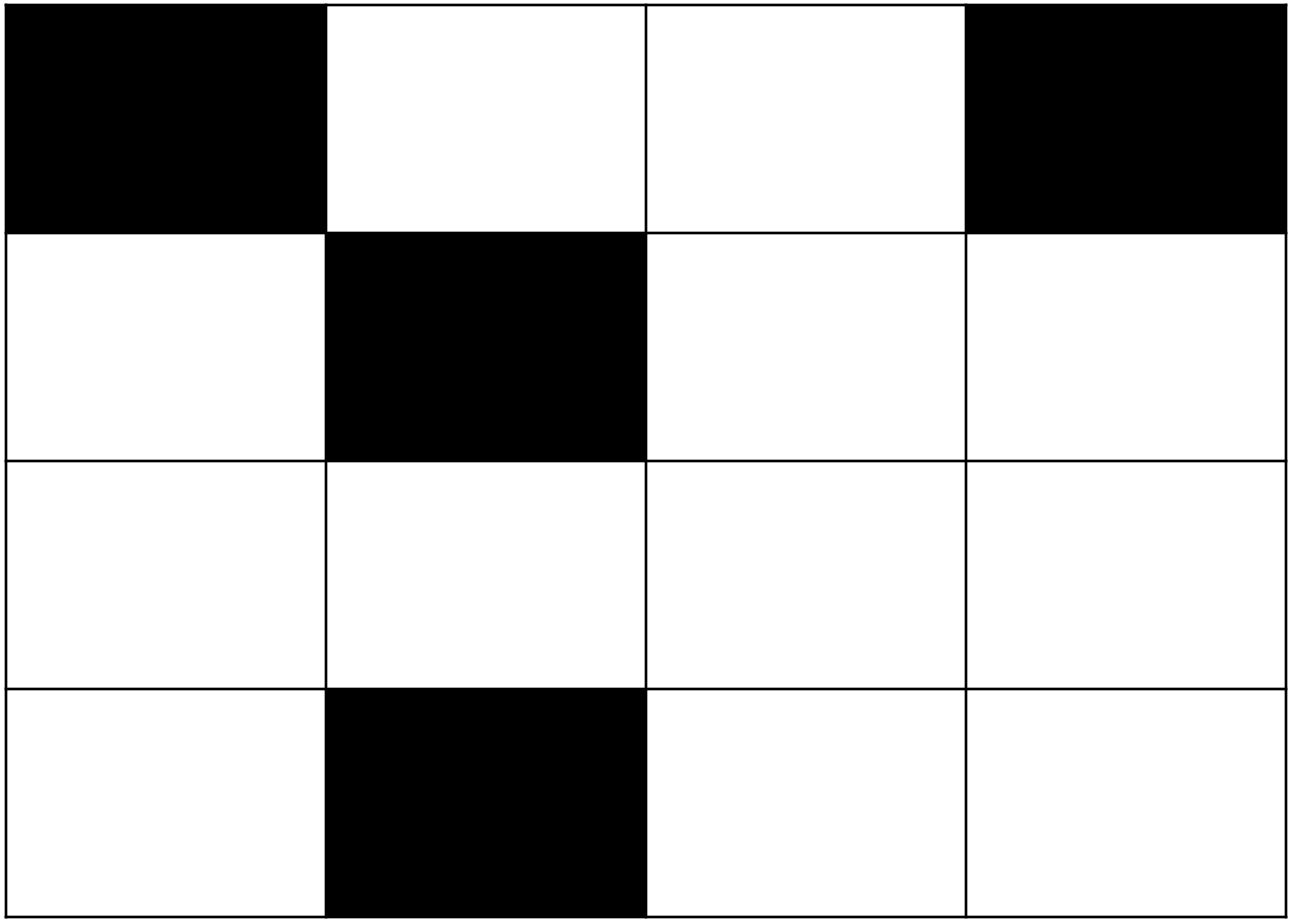
Is messaging the answer?

MuleSource

- Messaging is asynchronous
- Doesn’t put undue load on the server and the client

Roy's Solution for Flickr/Friendfeed

MuleSource



Lookup service

MuleSource

http://example.com/_2008/09/03/users?{userid}

Returns coordinate in sparse array

Leaves us with

MuleSource

- Periodic poll (every minute or so) with a 1KB response (1440 times/day)
- 6700 GETs for new pictures
- A GET for every time a user signs up

In contrast to

MuleSource

- 6700 XMPP messages per day
- 6700 GETs for new pictures
- Maintenance of new infrastructure

Conclusions

MuleSource

- Services are hard!
- In the long run, REST helps you
- Getting away from the edges, Mule can help
- Abdera, Jersey and Restlet are cool
- Mule has a bunch of other tools to help you too

Questions?

MuleSource

- http://mulesource.com
- http://mule.mulesource.org/display/MULE/MULE+RESTpack
- My Blog: http://netzooid.com/blog
- Twitter: dandiep
- dan@netzooid.com