This presentation is available for download from:
http://ciurana.eu

Beyond Java: Enterprise Apps, Python Programming and the JVM

Eugene Ciurana
Open-Source Evangelist
CIME Software Labs

http://ciurana.eu/contact

About Eugene...

- 15+ years building mission-critical, high-availability systems
- 14+ years Java work
- Open source evangelist
- Official adoption of open source/ Linux at Walmart worldwide
- State of the art tech for main line of business roll-outs
  - Largest companies in the world
  - Retail
  - Finance
  - Oil
  - Background: robotics to on-line retail

epicenter

This presentation is about...

- JSR223 Scripting support and what it means to you
- Python, Ruby, Groovy, mongoDB, Mule ESB, Spring
- Areas of application for 3rd-party languages
- Implementing code faster through scripting and continuous prototyping
- Leveraging other skills available in your organization
- Code deployments without OSGi or bundles
  - OSGi may not be available
  - Reduce or eliminate build/compile/package/test cycles
- Modifying your app server's code without stopping the container

What You'll Learn

- Identify the applications best suited for scripting development
- How to use non-Java languages for developing enterprise applications
- How to minimize cross-language impedance mismatch
- The advantages of mixing languages other than Java in your JVM
- How do implement an agile build/deploy cycle around scripting
- How to break away from the shackles of type checking everywhere
- Writing cross-platform business objects in Python, Ruby, or other languages is easy - and fun

Scripting and Java

- Scripting is built into Java 6
  - Spring has limited dynamic language support
  - Mule ESB, ServiceMix, other spring containers offer better or worse support on top of Spring; read the documentation
- Mix and match scripting language features and standard Java!!!
- If your container or app don't support your chosen language it's easy to extend it
  - Take a look at the javax.script package
- Is this popular? You bet!
  - At least 40 languages supported
  - Most run on the JVM itself
    - Python, Ruby, awk, JavaScript, Groovy, Scheme, Scala run as bytecodes
  - Some use a JNI bridge between Java and the native scripting engine

Reasons for Using Scripting

- Rapid prototyping
- Better tools in some problem domain
  - Python: outstanding system management tools
  - awk: runs circles around Java for massive text processing
  - Groovy: fast Java prototyping
  - You get the idea
- Missing features in the Java language
  - Generators and comprehensions
  - Continuations
  - Everything is an object and introspection
  - Dynamic event handling
- Leverage domain expertise
  - Long learning curve for Java coders in new problem domain
  - Domain experts may have robust, mature code written in other languages
- It's FUN

Reasons for Avoiding Scripting

- Performance
  - The JVM, the JIT, and Java are optimized to work together
  - Scripting languages, even when compiled to .class, will run slower
- Resource consumption
  - Java library + scripting language's library?
- Threading? Java's threads are superb
- Type safety may be critical
- Completeness
  - Java's language shortcomings are often overcome by its superb class libraries

Is Scripting In Your App's Future?

- Does your app require expertise that's already coded in something other than Java that you can use?
- Will coding in a scripting language pose a significant advantage in...
  - time to market?
  - language features not available in Java?
  - libraries or APIs not available or awkward in Java?
- Are you or your team proficient programmers in some scripting language?
  - Scripting for Java is not the place to learn a new language
- Do your SLAs allow for slower computational performance?
  - Java can be from 2 to 50 times faster than scripting languages

epicenter

So… Which Language?

- Lots of options, with new languages being added all the time
- Selection based on functional requirements:
  - Processing a lot of text? Use Jawk
  - XML manipulation? XSLT
- Selection based on platform:
  - Need scripting but have little time to learn? Groovy has the shortest learning curve; it's very "Java-like"
  - Cross-platform lower-level abstractions? Python
- Selection based on chosen standard framework
  - Spring? Groovy, Ruby, BeanShell
  - Mule? Anything that supports JSR-223 ScriptEngineFactory
- Language vs. implementation
  - Python vs. Jython; Ruby vs. JRuby; JavaScript vs. Rhino

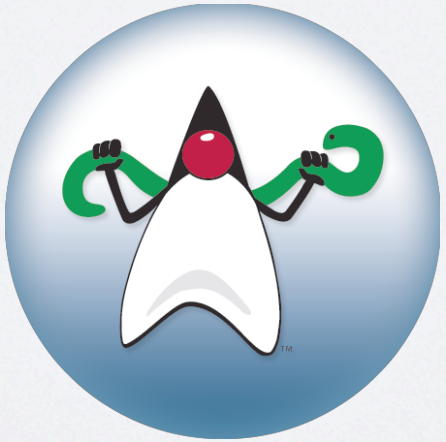
Now What?

- Start coding!
  - Adhere to best practices for the chosen language
- Don't waste time arguing with $LANGUAGE bigots
  - There is always some clown trying to convince you to switch $LANGUAGE from the JVM to native
- Don't waste time arguing with the Java bigots
  - There is always some clown trying to convince you that Java is the one and only true way
- Evaluate your progress
  - Was this the right decision?
  - Adapt

epicenter

Weighing the Alternatives

- Is dynamic code deployment the main reason for using scripting in your app?
  - Consider OSGi instead
- Is the reduction of compilation/build cycles the main reason for using scripting in your app?
  - Consider JRebel instead
- There are no absolutes
  - Weigh your functional requirements and SLAs
  - There is no universally good answer
  - There is only a good answer for your situation



How Does Your Language Rate?

- TIOBE Programming Language Index (March 2010)

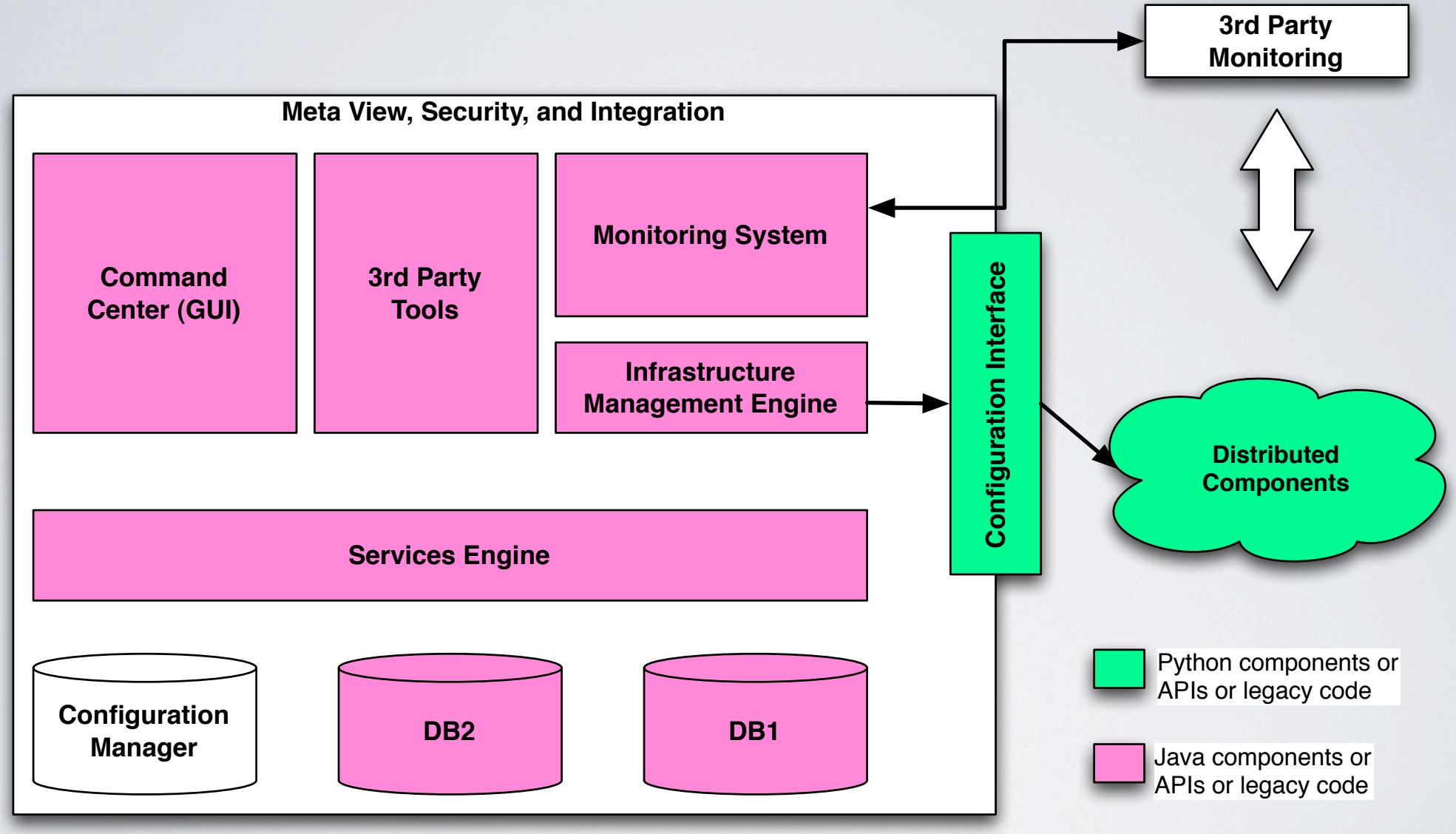
1	Java
2	C
3	PHP
4	C++
5	Visual BASIC
6	C#
7	Python
8	Perl
9	Delphi
10	JavaScript
11	Ruby

JVM

Case Study: System Management Tool

- Enterprise system for managing private clouds lifecycle
- Interfaces with monitoring tools, reporting systems, applications, and system management tools
- Public interfaces via web services
  - HTTP
  - JMS
- Common data format
  - JSON
  - BSON
- Requires maximum data storage non-transactional flexibility
  - mongoDB

Case Study: System Management Tool



epicenter

Selection Criteria

- Java was cumbersome for some low-level requirements
- The language and class library are rich
  - The abstractions weren't appropriate for problem domain
- Rich class library and ecosystem
- Portability across many heterogeneous platforms
- Language stability and robustness
- Stand-alone and JVM implementations
- Existing know-how

The Java language is Awesome as long as you don't need to break its abstractions!

Selection Criteria

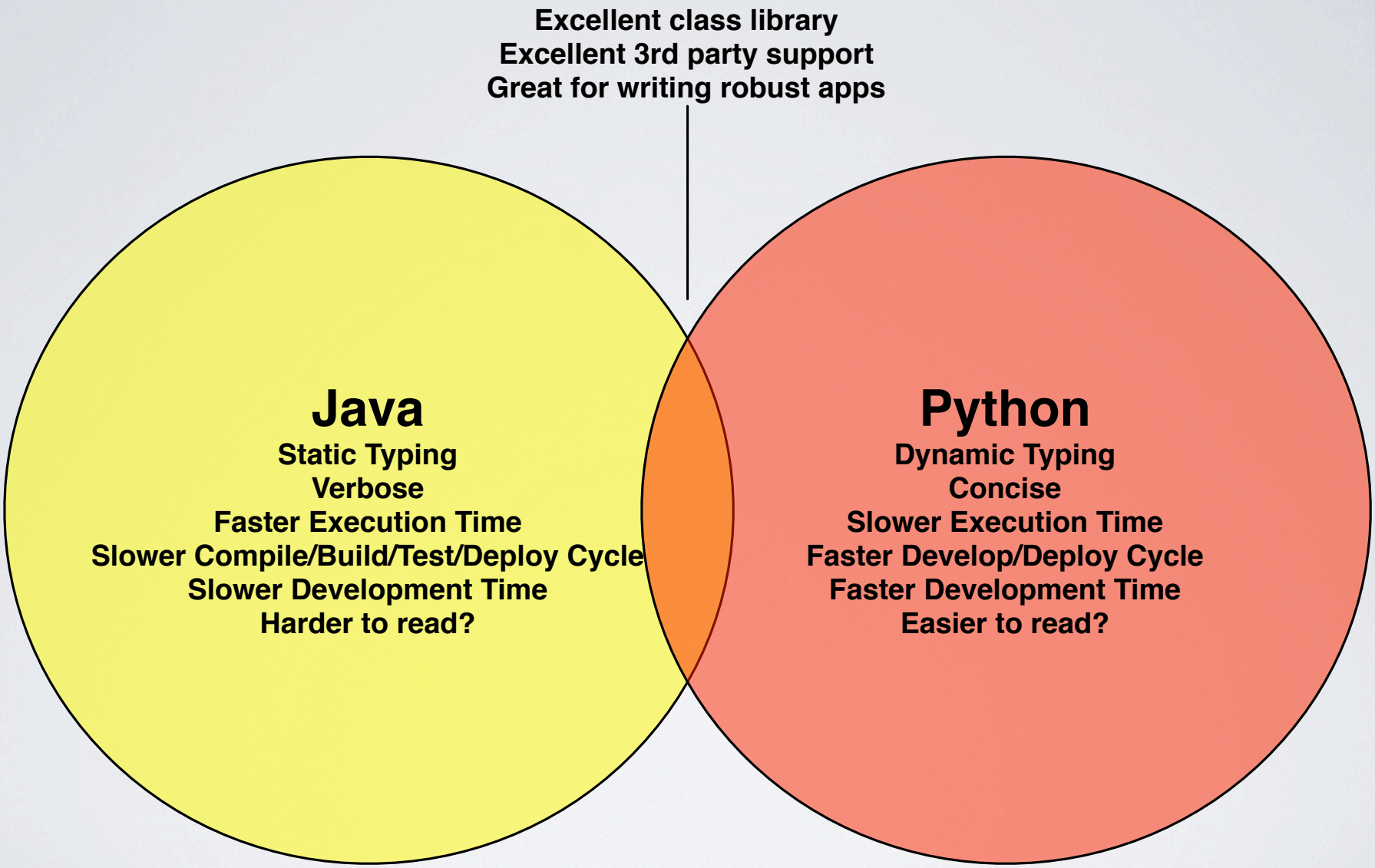
- Outstanding support for system-level operations
- Mid-level language preferred
- Rich class library and ecosystem
- Portability across many heterogeneous platforms
- Language stability and robustness
- Stand-alone and JVM implementations
- Existing know-how

Our choice:

Python

epicenter

Selection Criteria



Prime Directive

- **Whenever possible, Python code must run on CPython and Jython**
- Jython version is one revision behind current GA CPython
- Define best practices for mix-n-match Java code and Python
- Use your judgment: implement the best language option when both Java and Python provide equivalent functionality in the class library or syntactical feature
- Don’t mix-n-match languages within packages or modules

We Start Coding

- Python was used for the distributed system, lower level, activities
  - Portable way of replacing things like bash
  - Requirement to run the same code across Windows, UNIX
  - Portable support for OS-level operations
- Java was used for the business logic
  - Traditional stack
  - Mule as an app container hosts the web services
  - Jersey, Restlet API, all working well
  - Traditional database / JPA
- Longer development cycle
  - Proficiency in Java let us crank code out quickly
  - Edit/build compile cycle is annoying
  - Database changes require rebuilds/refactoring even with annotations

We Start Coding

- Replaced the RDBMs with mongoDB
  - NoSQL, document-oriented database
  - Faster turnaround: need a new “column”? Just add it!
  - mongoDB stores records in BSON (a JSON cousin)
- External web service APIs were all JSON
  - Mapping from JSON to BSON is almost 1:1!
- Java API for JSON is nice, but verbose
- Some Python components also require mongoDB access
- Eureka! The functionally equivalent Python code to Java’s is, at least half as long!
- Tests show that though Python is computationally slower than Java, normal network and database latency make it a non-issue
  - Milliseconds vs. nanoseconds

epicenter

mongoDB Java vs. Python

- Original payload (BSON or JSON):
  - `{ "name" : "Eugene", "nCount" : 42 }`
- Dealing with the Java code:
  - Deserialize the JSON code to some Java object
  - `MyObject o = (new Gson()).fromJson(payload, MyObject.class);`
  - Forces to define a new type or go through some tedious specificaton
  - Requires annotations if using Jersey
  - Code, code, code, code and more code!
- Dealing with Python code:
  - Deserialize to some Python object:
  - `o = json.loads(payload)`
  - Use the built-in dictionary as the payload (akin to a Map)
  - No need to define a new Java object or add all the extra code/ annotations for type checking!

epicenter

mongoDB Java vs. Python

- Simple operation: insert new payload

```
public ObjectId add(MyObject payload) {
  BasicDBObject o   = new BasicDBObject();
  ObjectId      oID = null;

  o.put("name", payload.getName());
  o.put("nCount", payload.getCount());

  docs.insert(o); // database collection "insert" persistent

  oID = docs.findOne(o).get("_id");

  return oID;
}
```

```
{ "name"   : "Eugene",
  "nCount" : 42,
  "_id"    : { "$_oid" : "123456789abcdef426798efcafebabe" }
}
```

epicenter

mongoDB Java vs. Python

- Simple operation: insert new payload

```
def add(payload):
  objectID = None
  objectID = docs.insert(payload)

  return objectID
```

```
{ "name"    : "Eugene",
  "nCount" : 42,
  "_id"     : { "$_oid" : "123456789abcdef426798efcafebabe" }
}
```

epicenter

Java vs. Python

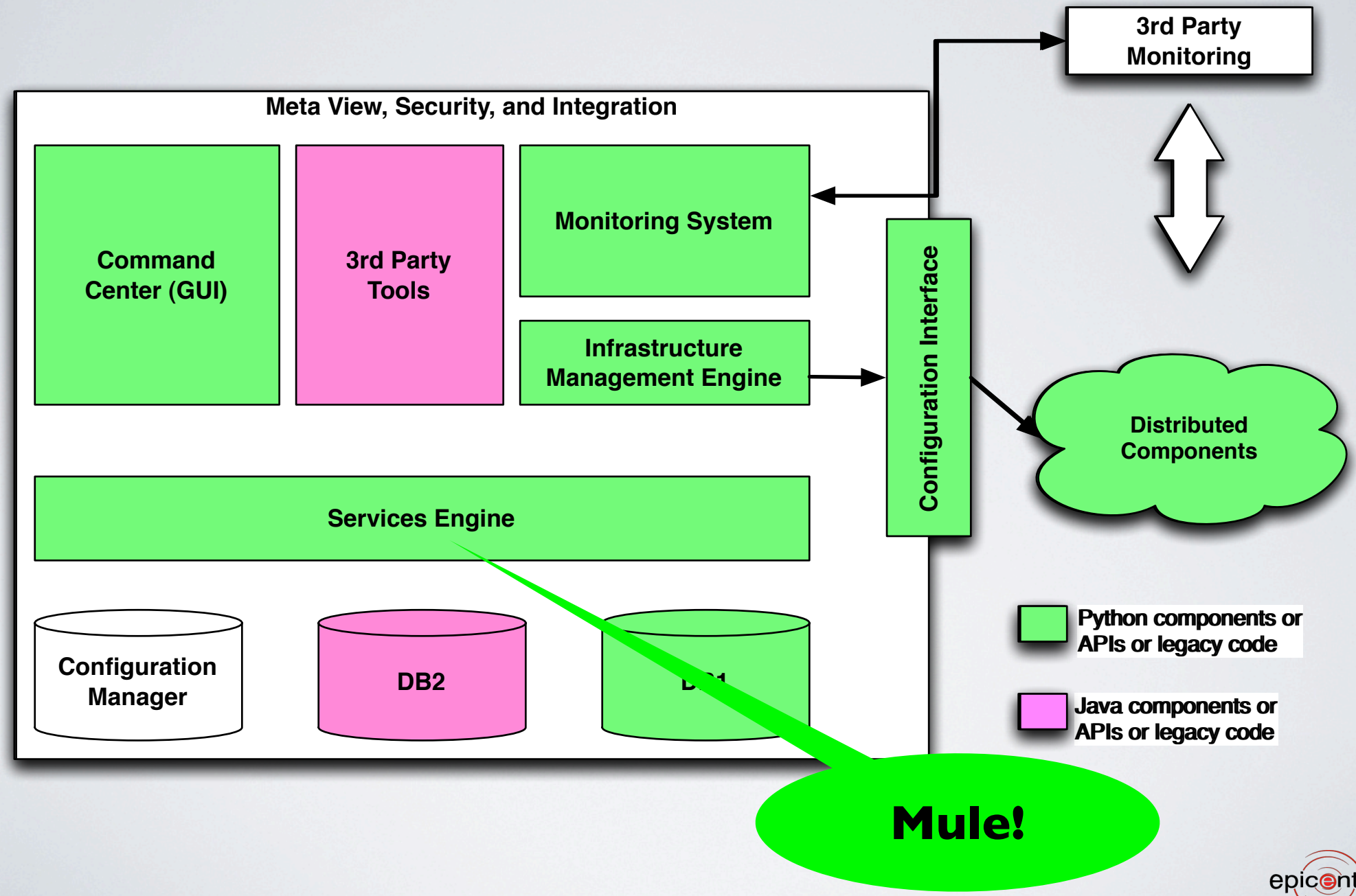
- Simple operation: Create a collection with elements from another.

```
public List<String> getNames() {
  List<String> list = new ArrayList<String>();

  for (MyObject record : someResultSet)
    list.add(record.getName());

  return list;
} // getNames
.
.
myNames = this.getNames();



myNames = [ record.getName() for record in someResultSet ]
```

Java vs. Python

- Class library
- The JSE and JEE class libraries are very complete
- The Python libraries are almost equivalent almost 1:1
  - Less verbose
- Third party libraries are equivalent
- Our team realized that we could start writing the business logic entirely in Python
- Use Java libraries where appropriate
- Use Python libraries where appropriate
- Don’t mix-n-match on the same class/object/module

Case Study: System Management Tool



epicenter

What is Mule?

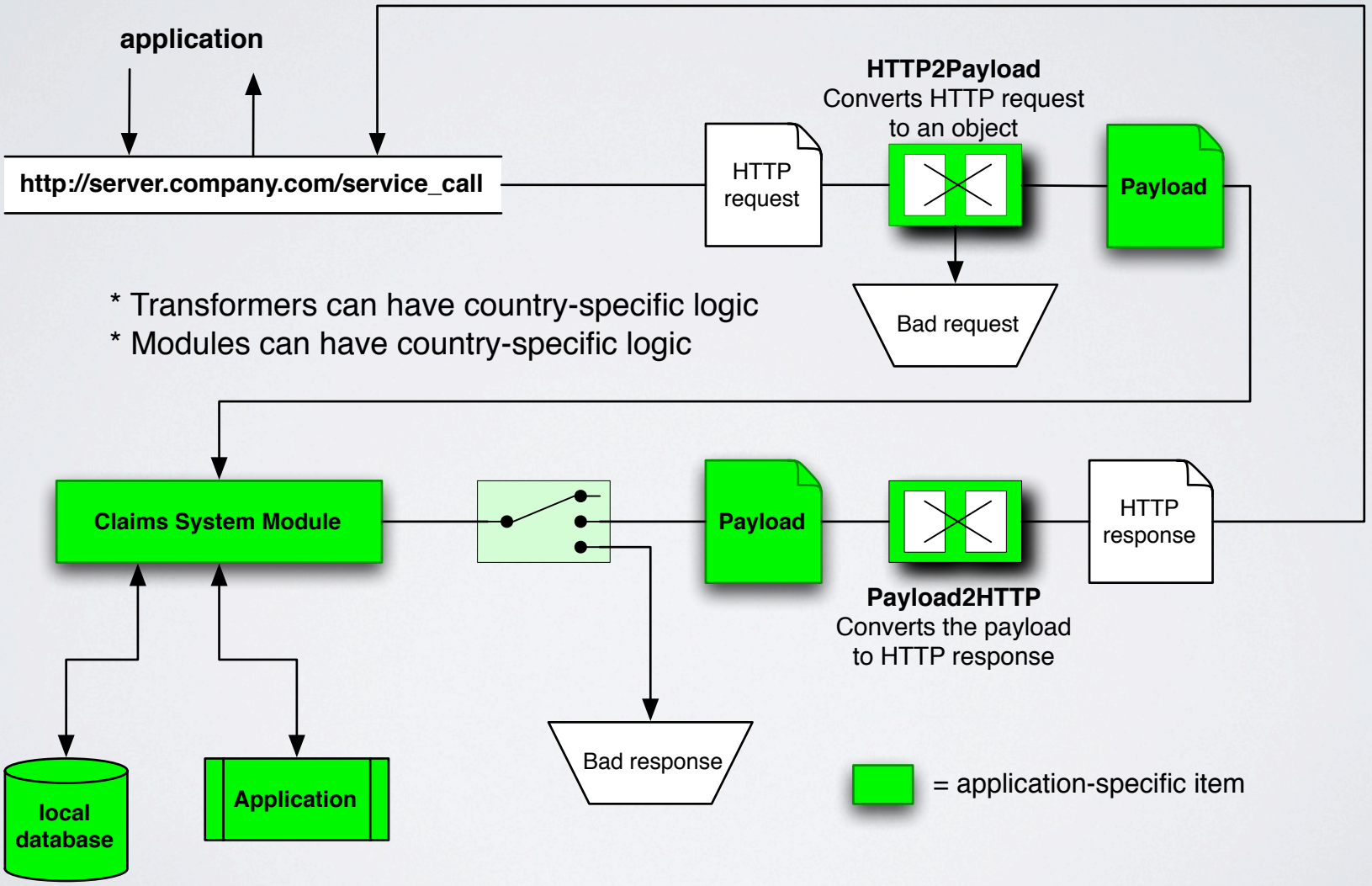
- Mule is an Enterprise Service Bus - a kind of middleware
- In Python terms, Mule is all these things rolled into one:
  - Twisted
  - SOAPpy
  - WSGI
  - PyHJB and JPype
  - ActiveJMS and MQI
  - omniORB
  - TLS Lite
  - PyBPM
  - XMPP
  - Can run in an app server or be a SOA app server, stand alone

- It's used for separating the networking logic (whatever protocol) and the business logic when implementing services

epicenter

What is Mule?

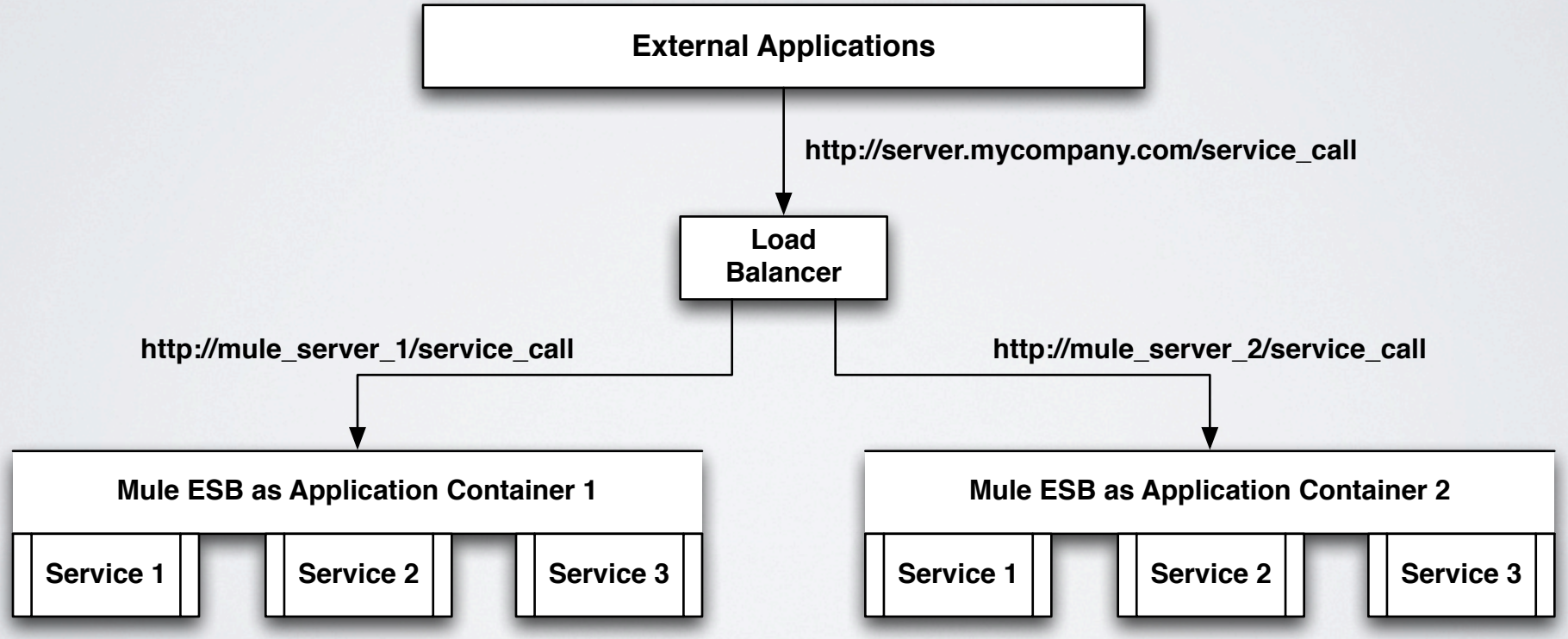
* Green items = code for the application
* Everything else = Mule standard services
* App is written in 100% Pure Java, no Mule-specific code for maximum portability
* Transformers may have Mule API calls



epicenter

What is Mule?

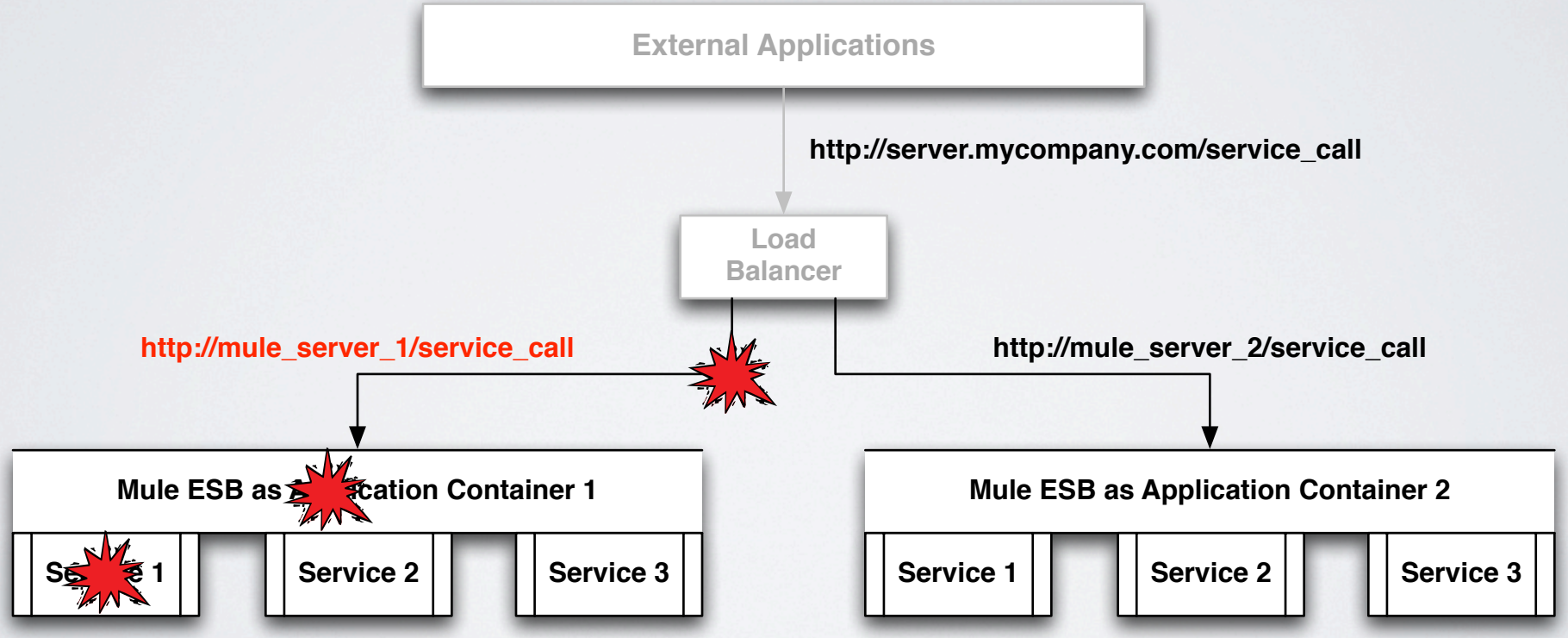
* Two or more Mule instances can provide services, for scalability if there is high demand
* Load balanced configuration has built-in fail-over
* External apps see a single point of entry: the service endpoint name
* Load balancer or proxy sends the request to any available Mule server
* Increased demand - add another Mule server without interrupting the existing ones
* Decreased demand - remove Mule servers without interrupting other servers
* This is an active/active configuration - any server can handle a request at any time
* Assumes that the service application components are **stateless**



epicenter

What is Mule?

* A/A configuration uses the load balancer to dispatch service calls
* The load balancer takes a failing service out of rotation automatically
* Failure reason no. 1: network connectivity
* Failure reason no. 2: Mule container
* Failure reason no. 3: Service application bug



epicenter

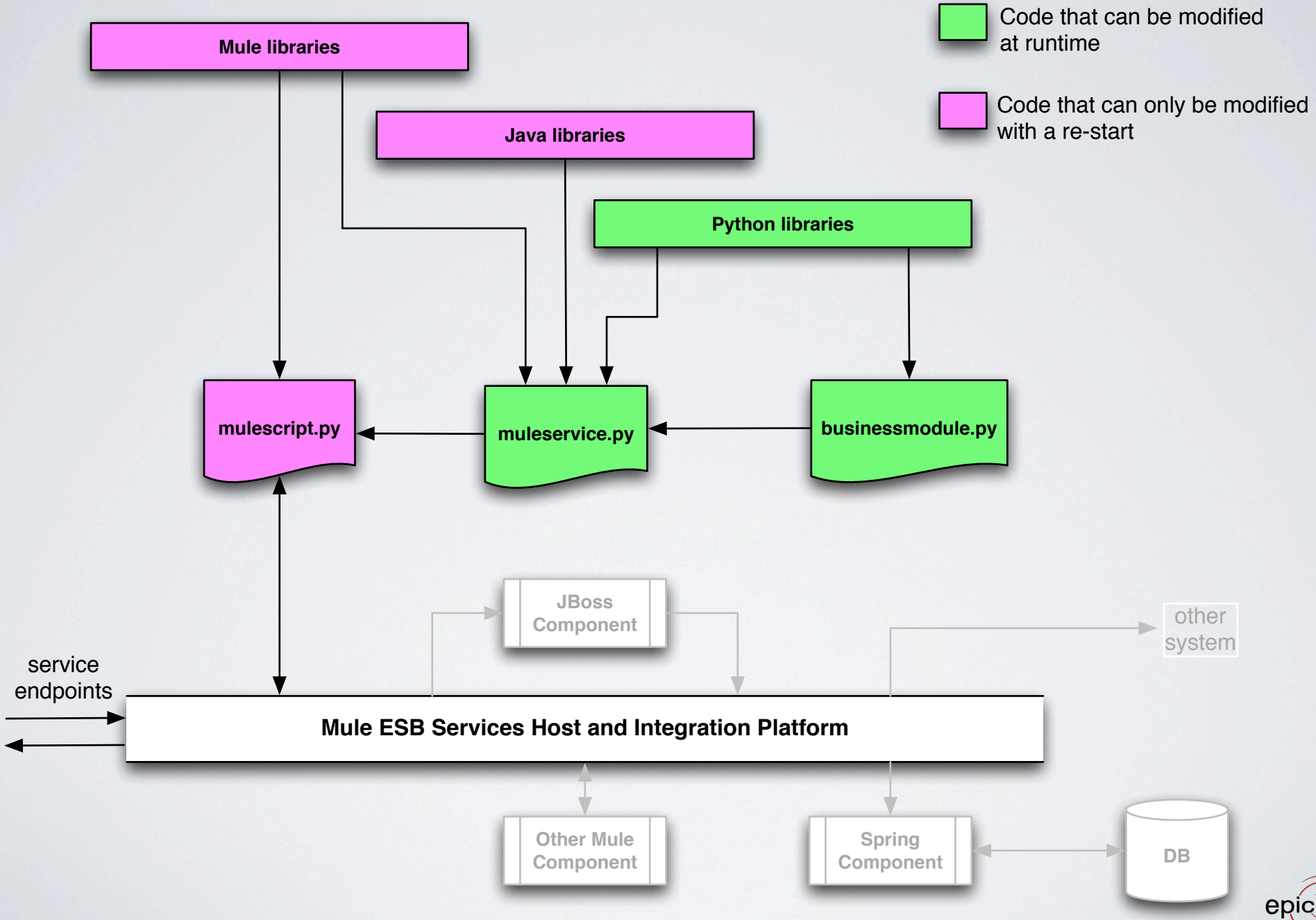
Mule Services in Python

- Mule is great for putting services together across multiple protocols
  - No OSGi support in 2.x
  - Tedious compile/build/package/deploy/test/run cycle
- Mule is based on Spring
  - It has Dynamic Language Support
  - It's more general and supports any scripting language conforming with JSR 223
- Business objects and transformers can be implemented in any scripting language
- These techniques can be applied to any ESB (e.g. ServiceMix) or Spring container
- Easy to incorporate code written by non-Java coders into an enterprise app running on a JVM!

Mule Services in Python

- We now have hot deployment without a long wait to restart the Mule container
- No dicking around with .jars, bundles, activation, deactivation
  - Save a file, test the service in less than a second
- Potentially patch production code in a hurry if necessary with no service disruption
- Call standard Java libraries when needed
- Integrate with Mule where appropriate using the Mule API, otherwise keep it separate
- Now we have modules and business logic that can run in a Java host or anywhere that Python works!

Mule Services in Python



epicenter

Mule/Spring Configuration

```
<http:endpoint name='Sample'
    address='http://localhost:8080/sample_dynamic_service' synchronous='true' />


<!--
  Force the Mule container to cache these components so that
  we can find them in import statements; otherwise the Jython
  interpreter will think that the .jar files/modules or
  .jar files/packages aren't in scope.
-->
<spring:bean id='GsonCache'       class='com.google.gson.Gson' />


<script:script name='SampleMuleComponent' engine='jython' file='article/mulescript.py' />


<model name='Core'>
  <service name='SampleService'>
    <inbound>
      <inbound-endpoint ref='Sample' />
    </inbound>
    <script:component script-ref='SampleMuleComponent' />
  </service>
</model>
```

epicenter

Service Definition

- This code is the interface between the Mule/Spring world and the Python world
- Regardless of how complex the script gets, this pattern remains almost identical

```
#!/usr/bin/env jython
#
# mulescript.py


import article.muleservice
from article.muleservice import SampleMuleComponent

reload(article.muleservice)  # For mule punching

sample = SampleMuleComponent(payload,
            log, eventContext)

result = sample.serviceRequest()
```

Service Implementation

- The implementation can be anything that fulfills a service request
- For this example, let's use a restlet-like service

```
import org.mule as mule

import article.businessmodule as businessmodule
from businessmodule import BusinessObject

# Enable dynamic updates to the script:
reload(businessmodule)

class SampleMuleComponent(object):

  # *** Public members ***

  # Get a local reference - useful for testing outside of a Mule container:
  def __init__(self, payload, log = None, eventContext = None, muleContext = None):
    self.payload      = payload
    self.muleContext  = muleContext
    self.eventContext = eventContext
    self.log          = log
    self.response     = ''
```

Service Implementation

```
def serviceRequest(self):
  if self.eventContext is not None:
    self.payload = self.eventContext.getMessage().getPayloadAsString()
    method       = self.eventContext.getMessage().getProperty('http.method')

  self.log.info('processing method = '+method)

  nStatus = 200 # OK
  if method == 'GET':
    self.response = BusinessObject().today()
  else:
    nStatus = 400 # Bad request
    self.response = 'Invalid HTTP method called!'

  responseMessage = mule.DefaultMuleMessage(self.response)
  responseMessage.setIntProperty('http.status', nStatus)

  return responseMessage
```

epicenter

Portable Business Objects

```
#!/usr/bin/env jython
#
# Listing 3
#
# Place this file in the module $MULE_HOME/lib/usr/article instead of
# in a .jar if you want to mule punch it.


from datetime import date


class BusinessObject(object):
  # *** Public members ***

  def today(self):      # Return today as a string
    return str(date.today())
```

Mule Punching Your Code

- From the Ruby/Python jargon “duck punching”, derived from duck typing. It means “punch the duck until it gives you the type you expect.”
  - Modify the code at run-time
- In a Mule/Spring container we decided to call it “mule punching” because it’d tell us that we’re modifying code intended for a Java environment
- You may dynamically add Java, Python, or any other functionality as long as the container’s class loader can find the code you’re punching and it’s dependencies

Mule Punching Your Code

```
import simplejson as json

import org.mule as mule

# Enable dynamic updates to the script:
reload(businessmodule)


class SampleMuleComponent(object):
  .
  def serviceRequest(self):
    if self.eventContext is not None:
      self.payload = self.eventContext.getMessage().getPayloadAsString()
      method       = self.eventContext.getMessage().getPayloadAsStringroperty('http.method')

    self.log.info('processing method = '+method)

    nStatus = 200 # OK
    if method == 'GET':
      self.response = BusinessObject().today()
    else:
      nStatus = 400 # Bad request
      self.response = 'Invalid HTTP method called!'

    self.response   = json.dumps({ 'nStatus' : nStatus, 'response' : self.response})
    responseMessage = mule.DefaultMuleMessage(self.response)
    responseMessage.setIntProperty('http.status', nStatus)

    return responseMessage
```

epicenter

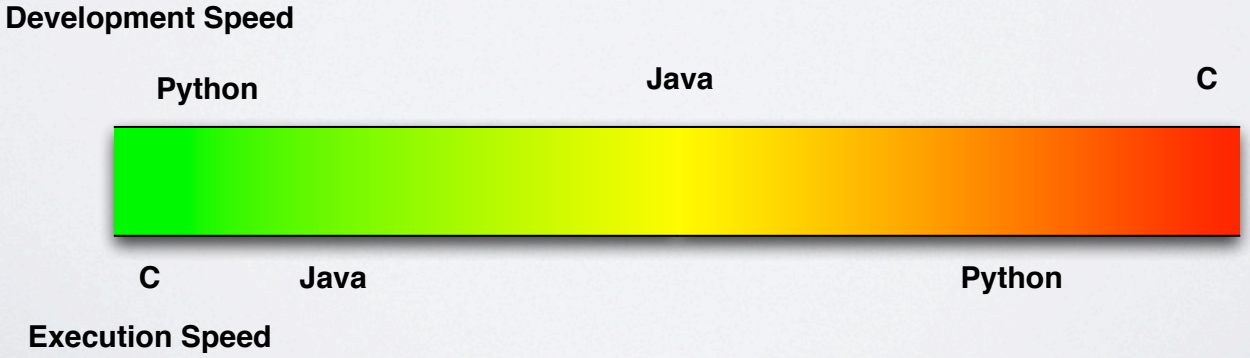
Mule Punching Your Code

```
import com.google.gson as gson
.
.
# Enable dynamic updates to the script:
reload(businessmodule)

class SampleMuleComponent(object):
  # *** Class members ***
  converter = gson.Gson()

  # *** Public members ***
  .
  def serviceRequest(self):
    if self.eventContext is not None:
      self.payload = self.eventContext.getMessage().getPayloadAsString()
      method       = self.eventContext.getMessage().getProperty('http.method')

    self.log.info('processing method = '+method)

    nStatus = 200 # OK
    if method == 'GET':
      self.response = BusinessObject().today()
    else:
      nStatus = 400 # Bad request
      self.response = 'Invalid HTTP method called!'

    self.response   = SampleMuleComponent.converter.toJson({ 'nStatus' : nStatus,
                                'response' : self.response, 'encoder' : 'Gson' })
    responseMessage = mule.DefaultMuleMessage(self.response)
    responseMessage.setIntProperty('http.status', nStatus)

    return responseMessage
```

epicenter

Results

- Coding time reduced by at least 50%
  - Developers are fluent in Java and Python
- Source code reduced by 30% to 75%
  - Algorithmic code saw the least reduction
  - Regular code + API calls average 50%
- Development/testing cycles reduced from 25% to 50%
  - Save/test vs. save/build/package/deploy/stop/start
- Can use with or without OSGi
- Language features help to come up with fresh approaches to problem solving



epicenter

Thanks for Coming!

Wanna know more about real life cloud, scalable systems?

Subscribe to the newsletter!

http://ciurana.eu/scalablesystems

http://twitter.com/ciurana **technology tweets**

Questions?

Eugene Ciurana

Open source evangelist

irishdev@ciurana.eu

+1 415 387 3800