OpenStack on the JVM

Jim Baker, Werner Mendizabal, Jorge Williams

{jim.baker,werner.mendizabal,jorge.williams}@rackspace.com

Overview

- What is OpenStack, especially Keystone
- **Problem**: need to deploy Keystone v3
- But continue working with existing Java-based identity infrastructure
- **Solution**: OpenStack is an extensible framework
- Can run Keystone on Jython (demo!)
- Can map Keystone models to our DAOs with a custom Keystone backend
- Deployment concerns: Spring DI, container, logging
- Applicability to other OpenStack components

About the team

Rackspace engineers:

- Jim Baker - Jython core developer
- Werner Mendizabal - Keystone Jython technical lead
- Jorge Williams - Principal Architect

About Rackspace "The Open Cloud Company"

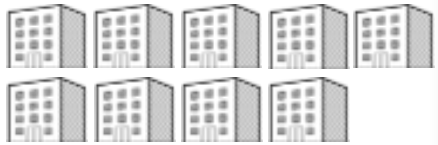
Our cloud offering is based on OpenStack:

5,000+ RACKERS

200,000+ CUSTOMERS
90,000+ SERVERS
26,000+ VM
≅70 PB STORED

GLOBAL FOOTPRINT
CUSTOMERS IN 120+ COUNTRIES

9 WORLDWIDE
DATA CENTERS



PORTFOLIO OF HOSTED SOLUTIONS
Dedicated - Cloud - Hybrid

Section 1

OpenStack

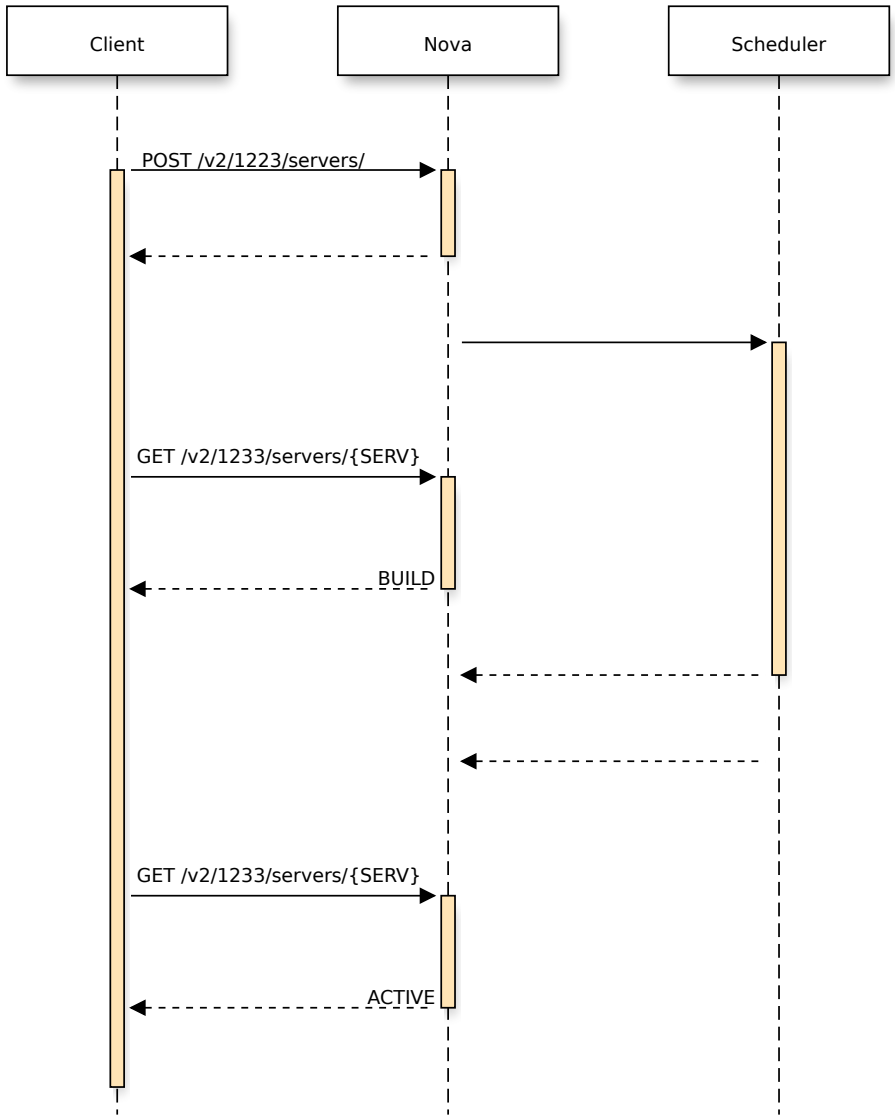
OpenStack

- Cloud computing platform providing infrastructure as a service (IaaS)
- Open sourced from Rackspace and NASA collaboration
- Now industry standard to build clouds, from small to large
- Written in Python

OpenStack and REST

- Sevice Oriented Architecture via REST architectural style
- **Resources**: Compute Nodes, Load Balancers, Databases, Storage Volumes etc. . .
- Accessed via URLs
- HTTP provides uniform API with `POST`, `GET`, `PUT`, `PATCH`, `DELETE`
- Benefit: simplified dynamic provisioning and configuration

Example: Creating a compute node



Example: Creating a compute node

Create a compute node by using `POST` against https://myhost/v2/1223/servers/

with the following body in JSON:

```
{
  "server": {
    "name": "server-test-1",
    "imageRef": "https://myhost/v2/1223/images/b566..
    "flavorRef": "https://myhost/v2/1223/flavors/2"
  }
}
```

Example: Creating a compute node

Resulting body will contain a URL for the compute node:


```
{
 "server": {
   "name": "server-test-1",
   "imageRef": "https://myhost/v2/1223/images/b566...
   "flavorRef": "https://myhost/v2/1223/flavors/2",
   "OS-DCF:diskConfig": "MANUAL",
   "id": "c6d04159-9bfc-4ab8-823d-0d5ca2abe152",
   "links": [
     {
       "href": "http://myhost/v2/4fd...",
       "rel": "self"
     }
   ]
}
```

OpenStack services

Services provided by OpenStack include

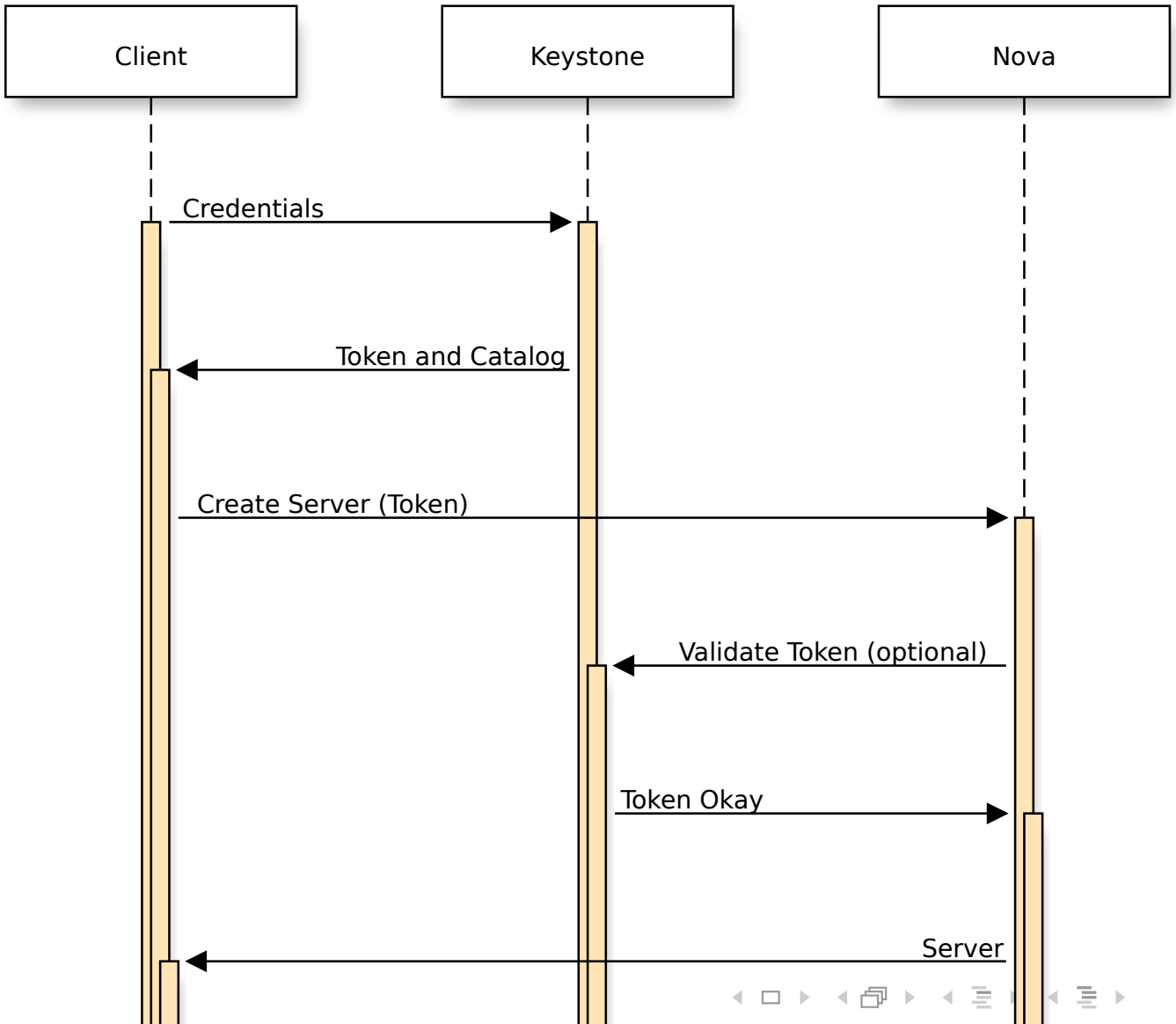
- **Nova** for managing servers
- **Swift** for object storage
- **Cinder** for block-based storage
- **Neutron** for software-define networking
- **Keystone** for identity service

Keystone Identity

Keystone binds the OpenStack services together:

- **Identity** for managing users, groups, roles
- **Catalog** for endpoint discovery
- **Policy** for rules-based authorization
- **Token** for validating tokens

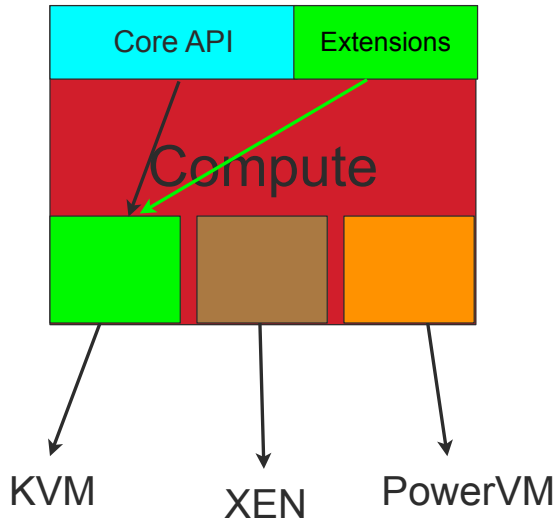
How Keystone fits in



OpenStack extensibility

All OpenStack services are extensible

- Different choice of tech, etc.
- OpenStack services are composed using Paste of WSGI components
- Extensions may expose API calls, implemented with backend drivers
- => Can be used to create different types of clouds



https://wiki.openstack.org/wiki/HypervisorSupportMatrix

Keystone Jython extensibility

Specific extensiblily options we are using:

- REST API - can intercept with tools like Rackspace's Repose (servlet filters, global counts for rate limiting)
- Can plug in choice of WSGI middleware - or not
- Can choose our own backend driver - Java with Spring dependency injection

Section 2

Why Jython?

Why Jython?

- OpenStack is run using CPython
- Complex project that supports many extensions and backends
- Why run OpenStack on the JVM?

Global Auth

- Rackspace's own identity implementation: Global Auth
- We proposed Keystone API based on our implementation of Global Auth
- Today Global Auth is an alternate implementation of Keystone API v1.1 and v2.0
- Written in Java
- Currently serving production requests:
  - 600 req/s
  - $<$ 100 ms response time
  - for hundreds of thousands of users

Issues maintaining our own implementation

- OpenStack does not have a formal test suite to test API conformance
- The Keystone community revolves around the Python implementation rather than the REST API
- It's difficult to contribute our ideas without contributing code
- We are not receiving benefits of community involvement lag behind on some features
- The OpenStack foundation may not allow us to use OpenStack trademark if we are not running upstream code
- Other organizations may face similar issues

Our goal

Use Python implementation to support v3 API

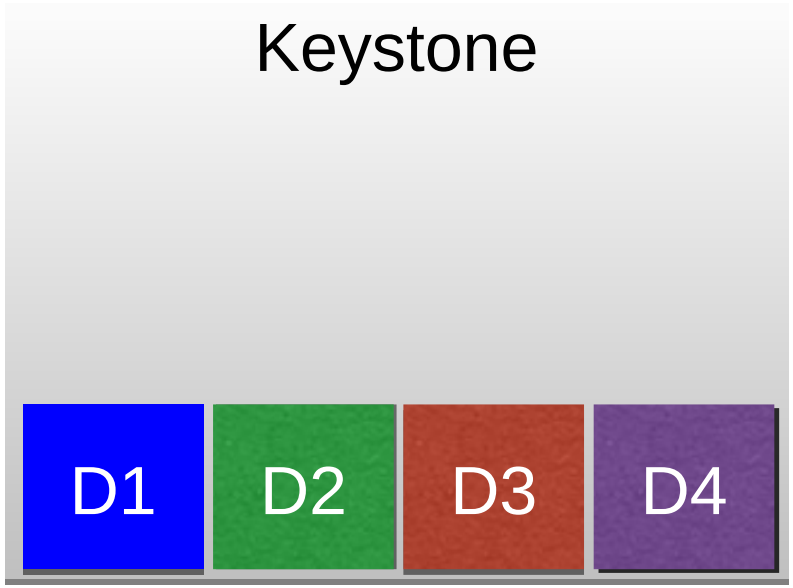
Requirements

- We need to keep existing API support (v1.1, v2) with our current implementation. We have specific extensions that may not be worth converting to Python
- Maintain our current performance (600 req/s $<$ 100 ms per request)
- It should be possible to use the same credentials and tokens across all of our APIs (v1.1, v2, v3)

Requirements: **single writer**

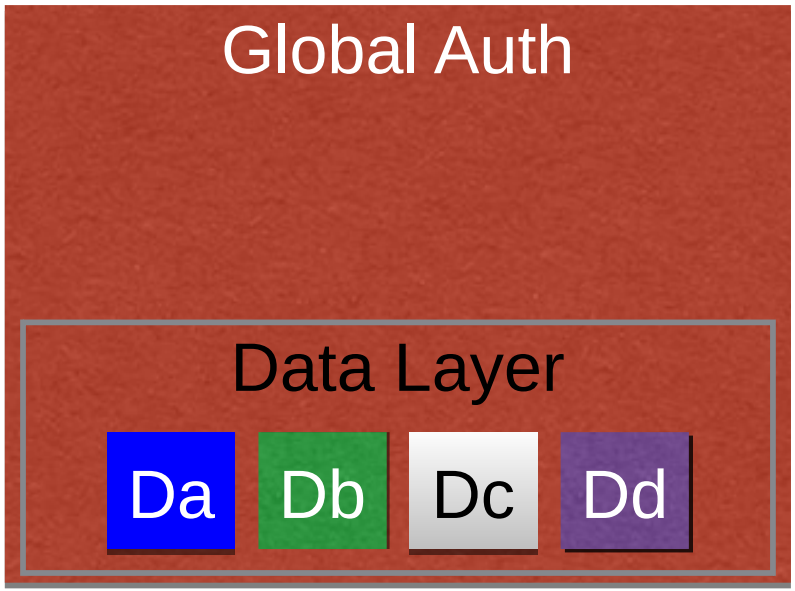
- Global Auth is truly global: we can't upgrade all instances at once
- Need to maintain forwards and backwards compatibility
- One piece of software should be responsible for accessing the backend

Keystone



Keystone is backed by a collection of “drivers” that allow integration with a backend datastore

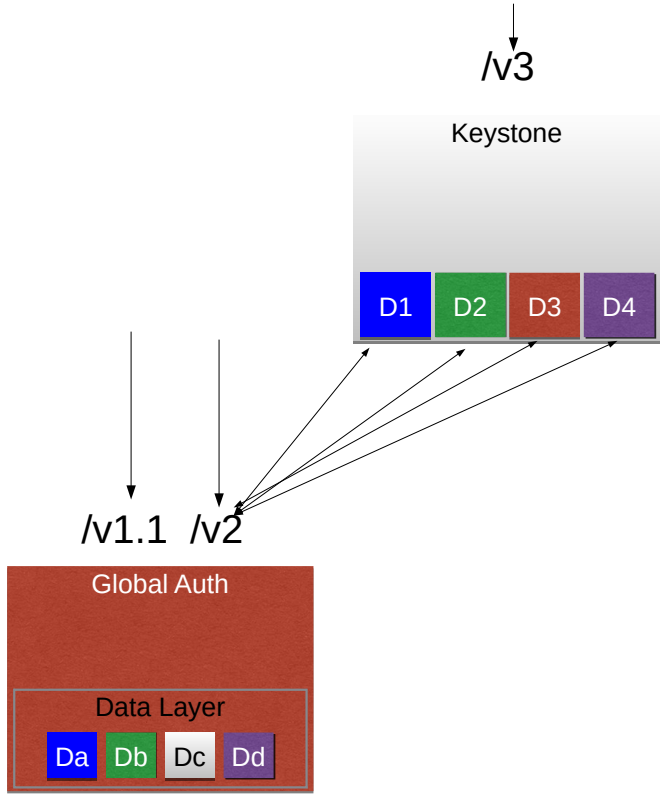
Global Auth



Likewise, Global Auth has a single data layer that is responsible for communication to the backend datastore

The data layer is a collection of Data Access Objects (DAOs) which correspond roughly to drivers in Keystone

Making REST Calls to Archive Single Writer



We can write backend Keystone drivers that allow us to essentially translate V3 calls into V2 calls

We would start here first because we already have good drivers for speaking to our backend

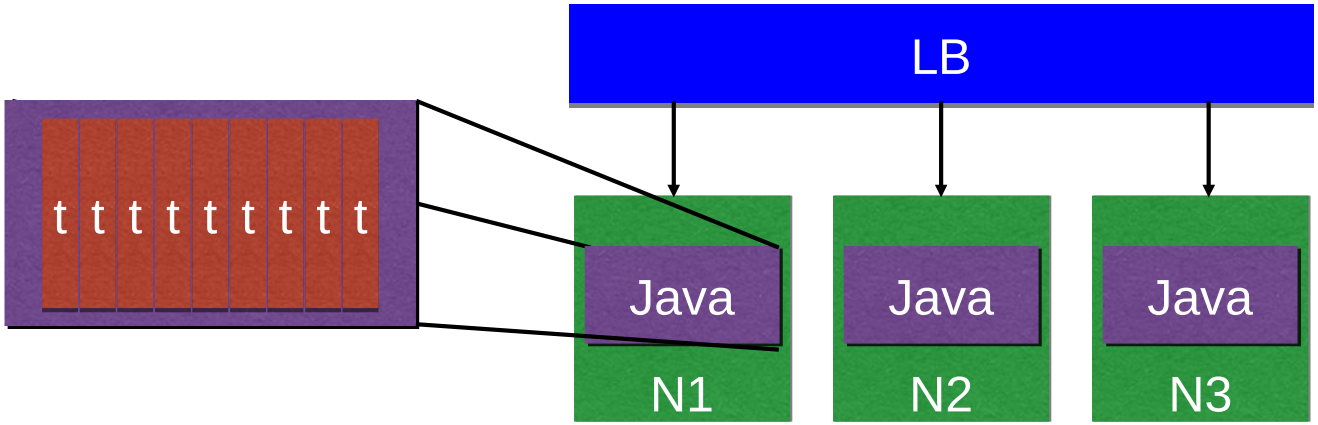
Issues with REST approach

There are a number of issues with the REST approach:

1. The approach adds overhead
2. Internal models don't always align single call in V3 may mean multiple calls in v2
3. There are operational challenges with running a Python service at scale. This is the biggest issue!

Operational challenges of running Keystone in CPython

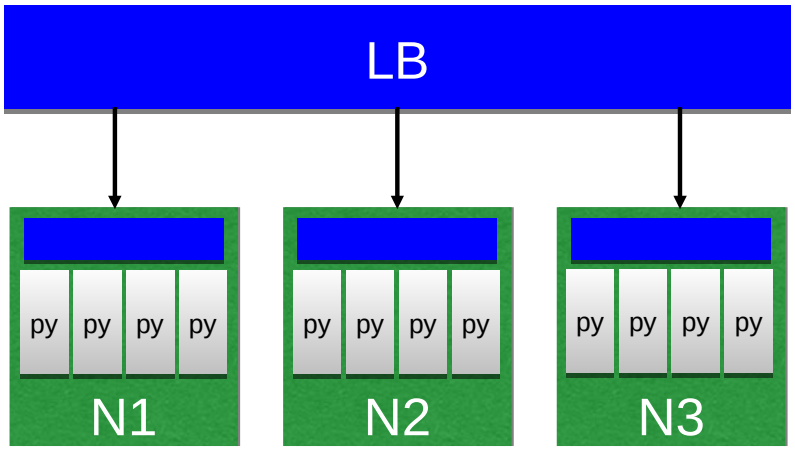
Differences in concurrency model means different ways to scale (processes vs threads)



concurrency achieved via threading (Single Process per Node)

Operational challenges of running Keystone in CPython

- Differences in concurrency model means different ways to scale (processes vs threads)



Operational Challenges of Running Keystone in CPython

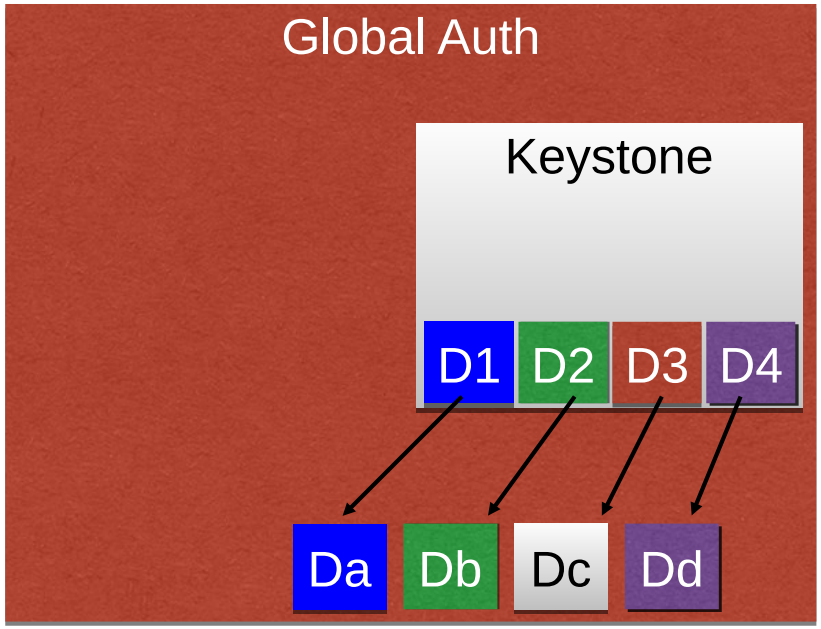
- New operational tools for monitoring etc (JMX is out)
- New requirements (Apache?, uWSGI?, more LB layers)
- Different ways of dealing with connection pools to our backend

Operational Challenges of Running Keystone in CPython

With the REST approach we need to address operational challenges *before* we can tackle anything else. . .

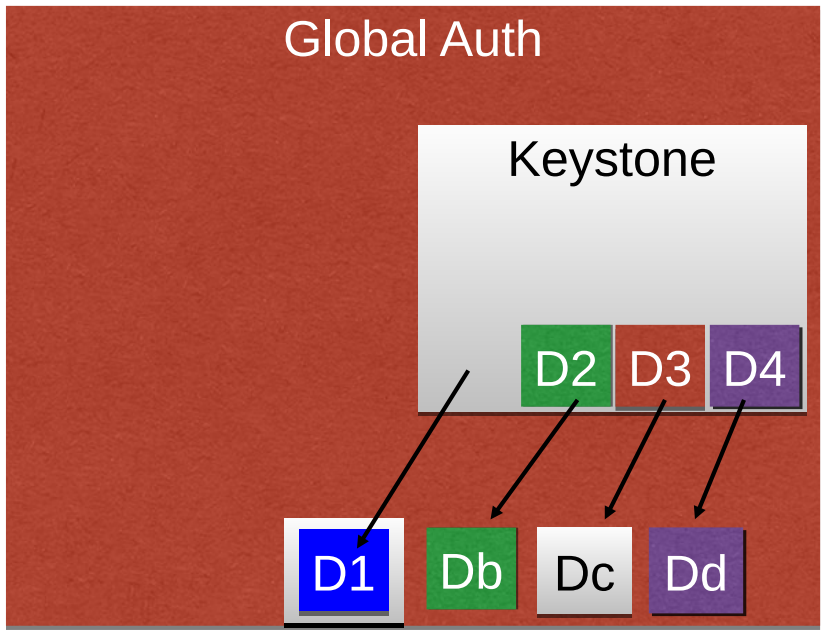
Big Idea: Use Jython to eliminate operational challenges

- We can compile the keystone code into a JAR
- We can add that JAR as a dependency that's consumed by our WAR
- We can create Keystone drivers that simply talk to our datalayer

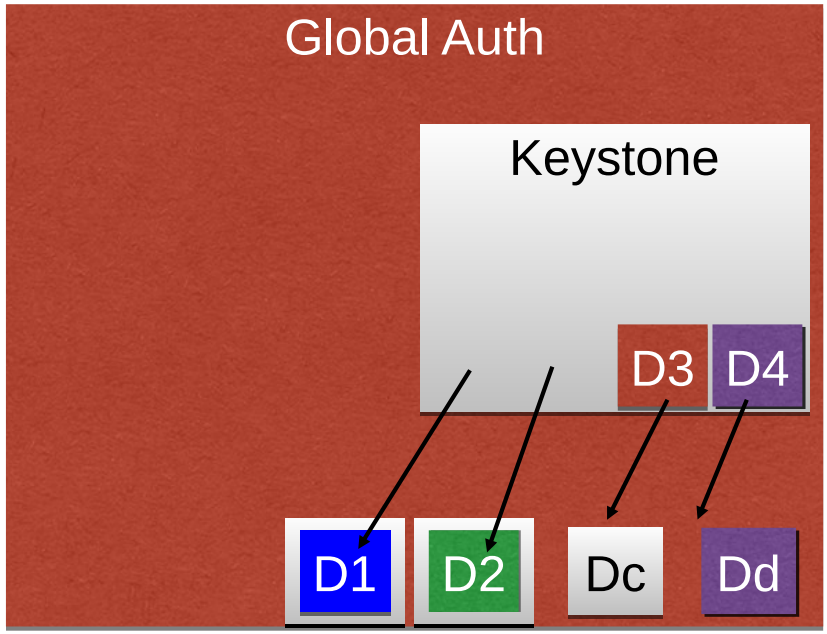


Reimplement drivers in Python

- We want to reimplement drivers in Python so we can contribute to the OpenStack community
- These new drivers replace parts of our existing DAO layer
- Notice that the same code is always writing to the backend

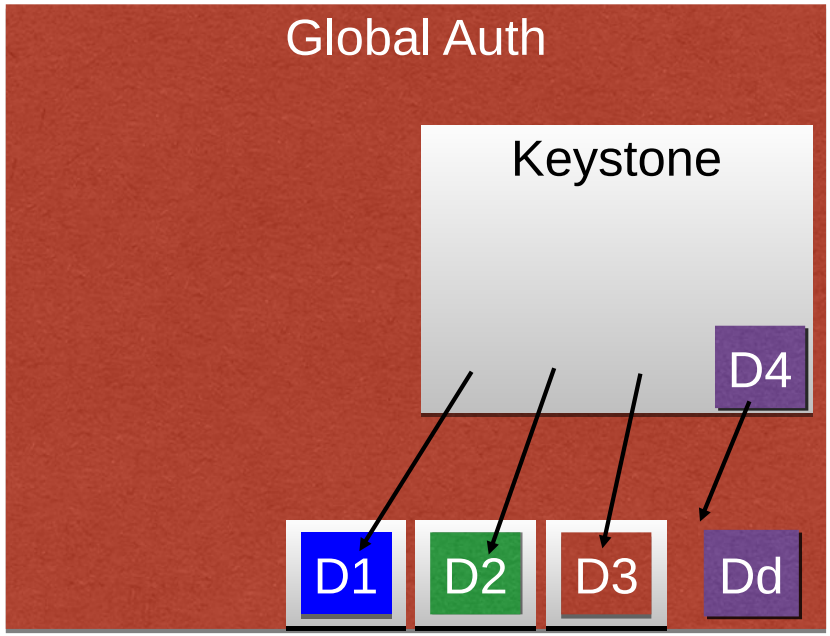


Reimplement drivers in Python



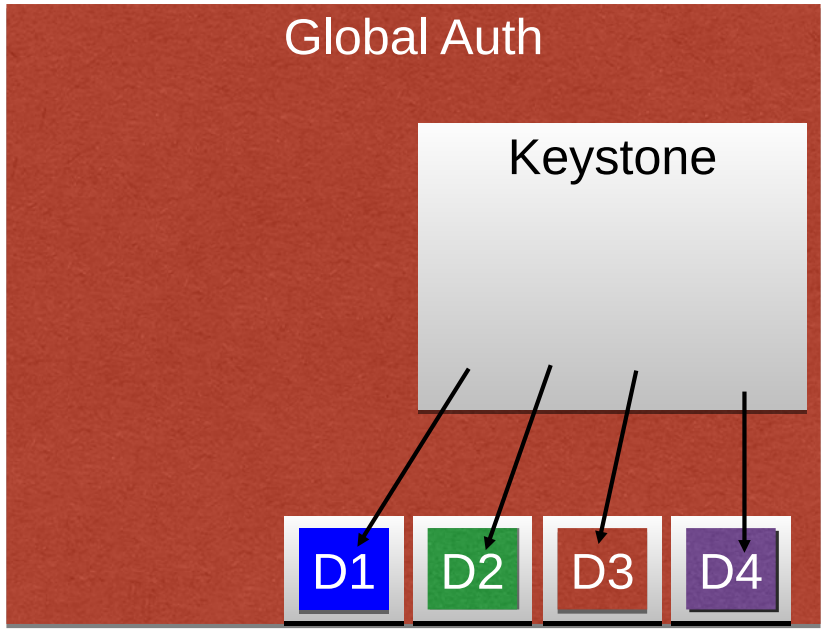
We can do this iteratively. . .

Reimplement drivers in Python



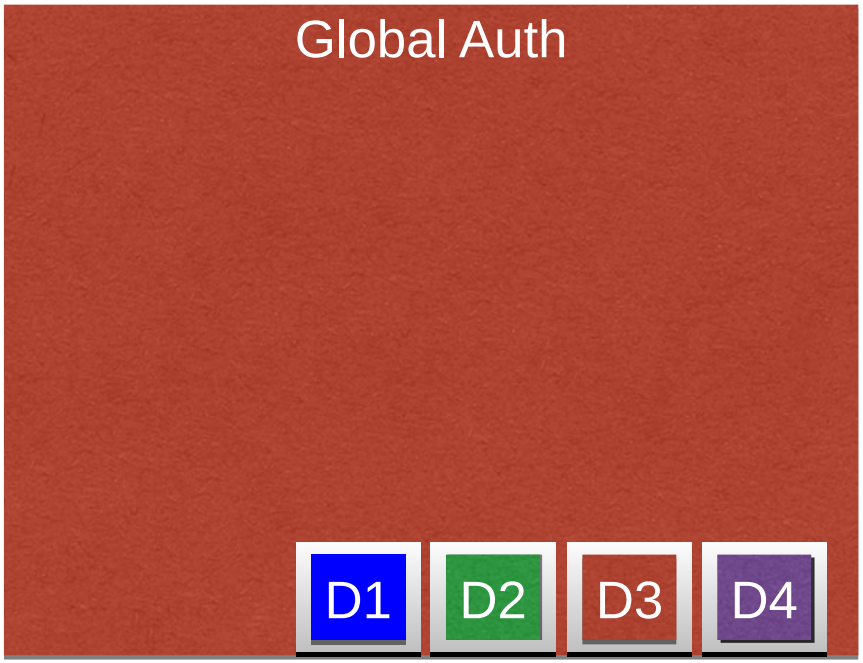
We can do this iteratively. . .

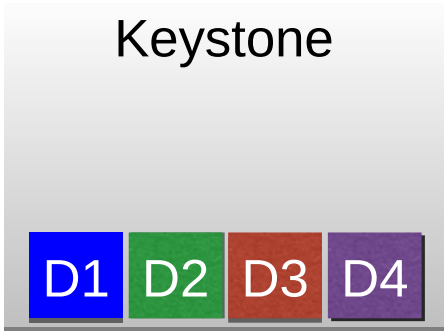
Reimplement drivers in Python



We can do this iteratively. . .

Same drivers





At this point we can decouple GA from Keystone and even run it in CPython if we wish

Jython rocks

Jython enables us to treat a Python application just as if it were a regular Java Application:

- Python is compiled to Java bytecode
- Code is run under the same concurrency model as any other Java process
- Our operations folks simply see a different WAR file, the same tools they are used to using continue to apply!

But there are some subtleties. . .

Section 3

Keystone on Jython

Demo

- Application context
- Making Java code accessible to Python
- Mapping Java objects to Python objects
- Custom Keystone Drivers
- JAR Construction

Section 4

Future work

Technology sweet spot

- Keystone does not use eventlets
- Keystone (via Paste) can be configured to not require use any C-based libraries

Keystone as a central component

- Keystone is very unique because of its central role
- Users expect all services to work with a common identity
- Users expect global authentication across data centers
- Other components can be partitioned by DC or even cell

Applicability to other OpenStack components?

- Maybe, maybe not
- Worthwhile discussion topic for today!

Eventlets in OpenStack

- Eventlets use green threads (greenlets), plus monkeypatching of networking libraries, like socket
- Idea: eventlets should scale higher than normal Python threads for concurrent I/O (not applicable as a GIL solution)
- In practice: eventlet scalability is not used
- Possible exception: Swift Object Storage, which sees much more higher concurrency
- => therefore just need to support greenlet API, not worry about its possible efficiency

Supporting greenlets

- Map greenlets to threads, much like JRuby has done for Ruby fibers
- JVM can run lots of threads!
- Implement its fundamental op - `switch` via `ArrayBlockingQueue(1)` mailbox

C library support

- Emerging standard for C library support is Python's CFFI
- CFFI is a simple foreign function interface to C, gives great possible performance
- Jiffy is now pure vaporware to provide CFFI backend for Jython
- Cursory examination of `cffi.backend_ctypes` suggests effort is straightforward/modest because of existing `jffi` package from Java Native Runtime project

Eliminate boilerplate


```
package com.rackspace.idm.jython;
import com.rackspace.idm.domain.dao.UserDao;
import org.springframework.*;

@Component
public class Objects {
  public static ApplicationContext
    applicationContext;

  @Autowired
  public void setApplicationContext(
      ApplicationContext applicationContext) {
    Objects.applicationContext = applicationContext;
  }
}
```

Future version of Clamp: Support Java annotations

- Clamp project supports precise bytecode layout
- Idea - have Java annotations be imported such that they turn into Python decorators adding metadata
- Clamp then visits classes, methods with ASM to add metadata

Section 5

Questions

Questions and discussion

Feel free to contact us for collaboration/questions:
{jim.baker,werner.mendizabal,jorge.williams}@rackspace.com

GitHub repo for starter project:
github.com/rackerlabs/keystone-jvm

Slides available in this repo as well