

CSC570 NoSQL Databases

Lab 2: Key-Value Database / Riak

Key-Value Databases:

A Key-Value Database is focused on speed so it is a great choice for datacenters like Amazon that must serve many requests with low latency. If every millisecond spent waiting is a potential customer loss, Key-Value is hard to beat.

Why Riak:

Riak is a distributed key-value database where values can be anything—from plain text, JSON, or XML to images or video clips—all accessible through a simple HTTP interface. Whatever data you have, Riak can store it. We'll be using Riak from Basho and, if you'd like more Riak you can find it here: <http://docs.basho.com/riak/latest/>

Riak is also fault-tolerant. Servers can go up or down at any moment with no single point of failure. Your cluster continues running along as servers are added, removed, or (ideally not) crash. A failed node is not an emergency, the DB will continue to function with the remaining nodes. But this flexibility has some trade-offs. Riak lacks robust support for ad hoc queries, and key-value stores, by design, have trouble linking values together (in other words, they have no foreign keys).

You query via URLs, headers, and verbs, and Riak returns assets and standard HTTP response codes. In this lab, we'll investigate how Riak stores and retrieves values and how distributed data is aggregated using Links. Then we'll explore a data-retrieval concept used heavily in Big Data: MapReduce. We will also examine how Riak clusters its servers and handles requests, even in the face of server failure. Finally, we'll look at how Riak resolves conflict that arises from writing to distributed servers.

Starting Riak and Creating a Cluster:

Power on your Lab 2 Riak VM (Ubuntu Linux), logon to the student account (password is: p@ssw0rd) and open a terminal window.

Change directory to your Riak developer installation directory: `$ cd riak-1.2.1/dev`

```
student@student-virtual-machine:~$ cd riak-1.2.1/dev
student@student-virtual-machine:~/riak-1.2.1/dev$
```

The installation of Riak includes some example servers that we will be using to create our lab cluster. These servers will all run on your local machine (LocalHost IP 127.0.0.1). If you had distributed servers, separate machines running across a network, you would use the machine's public IP to cluster the nodes.

Displaying the contents of the /dev directory should show 4 node directories and an image file:

```
student@student-virtual-machine:~/riak-1.2.1/dev$ ls
dev1 dev2 dev3 dev4 vaultboy.jpeg
student@student-virtual-machine:~/riak-1.2.1/dev$
```

Each of these directories is a complete, self-contained package containing a Riak node. We need to start each node individually using the start command in its /bin directory. For example, let's start node dev1 with: `$ dev1/bin/riak start`:

```
student@student-virtual-machine:~/riak-1.2.1/dev$ dev1/bin/riak start
student@student-virtual-machine:~/riak-1.2.1/dev$
```

Now start the other three:

```
student@student-virtual-machine:~/riak-1.2.1/dev$ dev2/bin/riak start
student@student-virtual-machine:~/riak-1.2.1/dev$ dev3/bin/riak start
student@student-virtual-machine:~/riak-1.2.1/dev$ dev4/bin/riak start
student@student-virtual-machine:~/riak-1.2.1/dev$
```

You now have four servers (nodes) up and running but they are not yet connected to one another, i.e. they are not members of a Riak cluster. The next step is to join the nodes together into a cohesive unit. You can do this using the riak-admin command interface. The riak-admin script, like the riak script used above, is found in the /bin directory of each Riak node. To create a cluster, we need to join the nodes using each server's riak-admin command named join and point them to any other cluster node.

Let's join node dev2 to node dev1:

```
$ dev2/bin/riak-admin cluster dev1@127.0.0.1
```

```
student@student-virtual-machine:~/riak-1.2.1/dev$ dev2/bin/riak-admin cluster join dev1@127.0.0.1
Success: staged join request for 'dev2@127.0.0.1' to 'dev1@127.0.0.1'
student@student-virtual-machine:~/riak-1.2.1/dev$
```

If the response states that the cluster join is successfully staged, everything went well. We'll get into staging below, but we still have three running nodes that have not yet been joined, so let's join those as well:

```
$ dev3/bin/riak-admin cluster join dev1@127.0.0.1
```

```
$ dev4/bin/riak-admin cluster join dev1@127.0.0.1
```

It doesn't really matter which servers we point them at—in Riak, all nodes are equal. Now that dev1 and dev2 are in a cluster, we can point dev3 and dev4 at either one.

Once two or more nodes have been joined, they will share all the information necessary to join all of the nodes into a unity. Thus, if dev1 is joined to dev2 and also to dev3, dev2 and dev3 will be able to communicate with one another.

At this point, the nodes have not yet been actually joined. Instead, the join operations have been staged and are ready to be committed. To make those joins take effect, you first must review the planned cluster changes by issuing a cluster plan command: `$ dev1/bin/riak-admin cluster plan`

The plan command will print out a synopsis of what changes will be made to the cluster on commit and how the cluster will look after the changes are complete. The output should look like this:

```

student@student-virtual-machine:~/riak-1.2.1/dev$ dev1/bin/riak-admin cluster plan
===== Staged Changes =====
Action          Nodes(s)
-----
join            'dev2@127.0.0.1'
join            'dev3@127.0.0.1'
join            'dev4@127.0.0.1'
-----

NOTE: Applying these changes will result in 1 cluster transition

#####
                After cluster transition 1/1
#####

===== Membership =====
Status    Ring    Pending    Node
-----
valid     100.0%    25.0%     'dev1@127.0.0.1'
valid      0.0%     25.0%     'dev2@127.0.0.1'
valid      0.0%     25.0%     'dev3@127.0.0.1'
valid      0.0%     25.0%     'dev4@127.0.0.1'
-----
Valid:4 / Leaving:0 / Exiting:0 / Joining:0 / Down:0

Transfers resulting from cluster changes: 48
16 transfers from 'dev1@127.0.0.1' to 'dev4@127.0.0.1'

```

If your plan looks good, you can then commit the changes that you staged:

```
$ dev2/bin/riak-admin cluster commit
```

NOTE: All changes to a cluster can be committed from any node.

```

student@student-virtual-machine:~/riak-1.2.1/dev$ dev1/bin/riak-admin cluster commit
Cluster changes committed
student@student-virtual-machine:~/riak-1.2.1/dev$

```

To test our cluster, we can now issue a status command: \$ dev1/bin/riak-admin member-status

You should see your four-node cluster valid and equally sharing the workload:

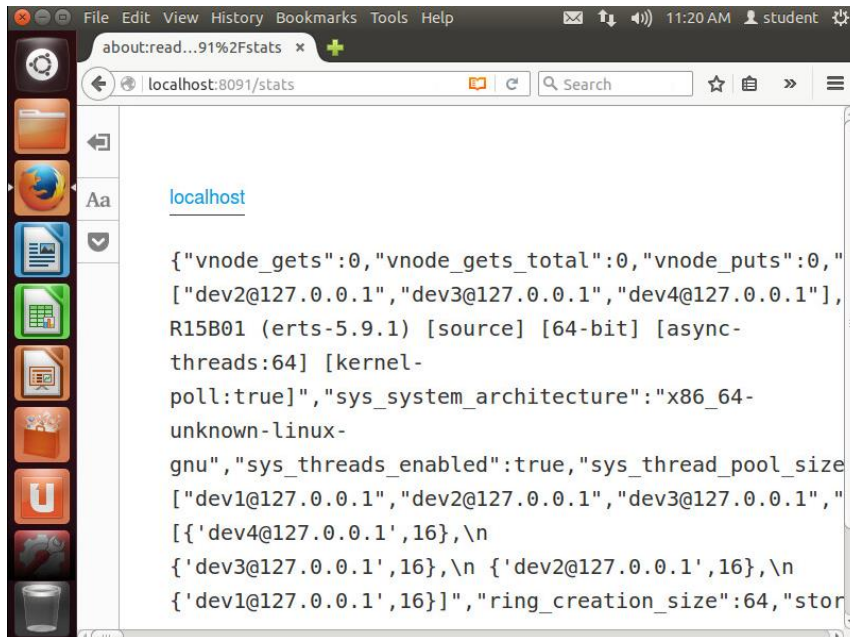
```

student@student-virtual-machine:~/riak-1.2.1/dev$ dev1/bin/riak-admin member-status
===== Membership =====
Status    Ring    Pending    Node
-----
valid     25.0%     --         'dev1@127.0.0.1'
valid     25.0%     --         'dev2@127.0.0.1'
valid     25.0%     --         'dev3@127.0.0.1'
valid     25.0%     --         'dev4@127.0.0.1'
-----
Valid:4 / Leaving:0 / Exiting:0 / Joining:0 / Down:0
student@student-virtual-machine:~/riak-1.2.1/dev$ █

```

If you were on a distributed cluster and managing the cluster remotely, you could use your local browser to verify the health of your servers by checking their stats. Do this by opening a browser window and entering the IP address or URL of your cluster, followed by the admin port number, 8091, and the status page /stats: <http://localhost:8091/stats>

It should display lots of information about the cluster and should look something like this



Riak CRUD

Now, let's begin interacting with our cluster using a REST interface. Riak supports a large number of client languages: Its official distribution maintains client libraries for Java, Ruby, Python, .NET, Erlang, and PHP. Riak supplies an HTTP REST API, so we're going to interact with it via the command line browser cURL. Using cURL allows us to look at the underlying Riak API without having to install a specific programming language driver.

REST and cURLs

REST stands for REpresentational State Transfer. It has become the de facto architecture of web applications (so it's worth knowing). REST is a guideline for mapping resources to URLs and interacting with them using CRUD verbs: POST (Create), GET (Read), PUT (Update), and DELETE (Delete).

The HTTP browser program cURL has been installed for you and we will use it as our REST interface, because it's easy to specify verbs (like GET and PUT) and HTTP header information (like Content-Type). With the cURL command, we speak directly to the Riak server's HTTP REST interface.

You can validate the cURL command works with Riak by pinging a node: \$ curl http://localhost:8091/ping

```
student@student-virtual-machine:~/riak-1.2.1/dev$ curl http://localhost:8091/ping
OKstudent@student-virtual-machine:~/riak-1.2.1/dev$
```

Let's issue a bad query. -I tells cURL that we want only the header response.

\$ curl -I http://localhost:8091/riak/no_bucket/_no_key

```
student@student-virtual-machine:~/riak-1.2.1/dev$ curl -I http://localhost:8091/riak/no_bucket/no_key
HTTP/1.1 404 Object Not Found
Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted it blue)
Date: Fri, 12 Feb 2016 23:26:58 GMT
Content-Type: text/plain
Content-Length: 10
```

Since Riak is designed to use HTTP URLs and actions, it uses HTTP headers and error codes. The 404 response means the same as a 404 when you encounter a missing web page: "nothing found".

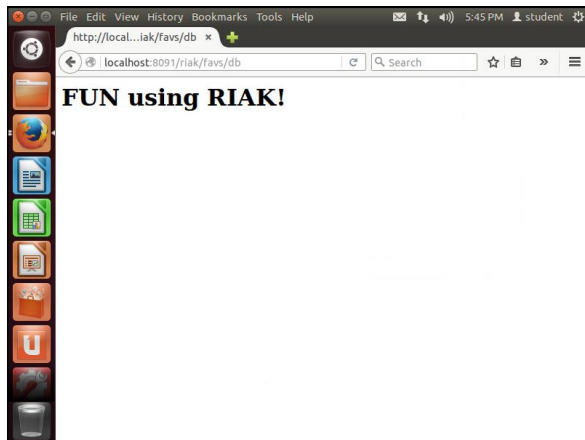
The PUT instruction

The -X PUT parameter tells cURL that we want to perform an HTTP PUT action to store or modify on an explicit key. The -H attribute sets the following text as HTTP header information. In this case, we set the MIME content type to HTML. Everything passed to -d (also known as the body data) is what Riak will add as a new value.

\$ curl -v -X PUT http://localhost:8091/riak/favs/db \
-H "Content-Type: text/html" \
-d "<html><body><h1>Fun Using Riak! </h1></body></html>"

```
student@student-virtual-machine:~/riak-1.2.1/dev$ curl -v -X PUT http://localhost:8091/riak/favs/db -H "Content-Type: text/html" -d "<html><body><h1> FUN using RI AK! </h1></body></html>"
* About to connect() to localhost port 8091 (#0)
* Trying 127.0.0.1... connected
> PUT /riak/favs/db HTTP/1.1
> User-Agent: curl/7.22.0 (x86_64-pc-linux-gnu) libcurl/7.22.0 OpenSSL/1.0.1 zlib/1.2.3.4 libidn/1.23 librtmp/2.3
> Host: localhost:8091
> Accept: */*
> Content-Type: text/html
> Content-Length: 52
>
* upload completely sent off: 52out of 52 bytes
< HTTP/1.1 204 No Content
< Vary: Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted it blue)
< Date: Fri, 12 Feb 2016 23:42:29 GMT
< Content-Type: text/html
< Content-Length: 0
<
* Connection #0 to host localhost left intact
* Closing connection #0
student@student-virtual-machine:~/riak-1.2.1/dev$
```

Navigate to `http://localhost:8091/riak/favs/db` in a browser to see the message you stored.



Riak is a key-value store, so it expects you to pass in a key to retrieve a value. Riak breaks up classes of keys into buckets to avoid key collisions—for example, a key for java the language will not collide with java the drink.

We're going to create a system to keep track of video games in a game collection. We'll start by creating a bucket of video games that contain each title's details (genre, platform).

The URL follows this pattern:

`http://SERVER:PORT/riak/BUCKET/KEY`

A straightforward way of populating a Riak bucket is to know your key in advance. We'll first add FIFA2016 and give it the key `fifa2016` with the value `{"genre": "sport", "platform": "PS4"}`. You don't need to explicitly create a bucket—putting a first value into a bucket name will create that bucket.

```
$ curl -v -X PUT http://localhost:8091/riak/games/fifa2016 \
-H "Content-Type: application/json" \
-d '{"genre": "sport", "platform": "PS4"}'
```

The `-v` (verbose) attribute in the cURL command outputs a complete header line, including the `>HTTP/1.1 204 No Content`, meaning the server has fulfilled the request but does not need to return an entity-body.

```
student@student-virtual-machine:~$ curl -v -X PUT http://localhost:8091/riak/games/fifa2016 -H "Content-Type: application/json" -d '{"genre": "sport", "platform": "PS4"}'
* About to connect() to localhost port 8091 (#0)
* Trying 127.0.0.1... connected
* PUT /riak/games/fifa2016 HTTP/1.1
> User-Agent: curl/7.22.0 (x86_64-pc-linux-gnu) libcurl/7.22.0 OpenSSL/1.0.1 zlib/1.2.3.4 libidn/1.23 librtmp/2.3
> Host: localhost:8091
> Accept: */*
> Content-Type: application/json
> Content-Length: 39
>
* upload completely sent off: 39out of 39 bytes
< HTTP/1.1 204 No Content
< Vary: Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted it blue)
< Date: Sat, 13 Feb 2016 00:00:17 GMT
< Content-Type: application/json
< Content-Length: 0
<
* Connection #0 to host localhost left intact
* Closing connection #0
student@student-virtual-machine:~$
```

We can view the list of buckets that have been created by entering

```
$ curl -X GET http://localhost:8091/riak?buckets=true
```

```
student@student-virtual-machine:~$ curl -X GET http://localhost:8091/riak?buckets=true
{"buckets":["favs","games"]}student@student-virtual-machine:~$
```

({"buckets":["favs","games"]} is what we are looking to be returned, and it was)

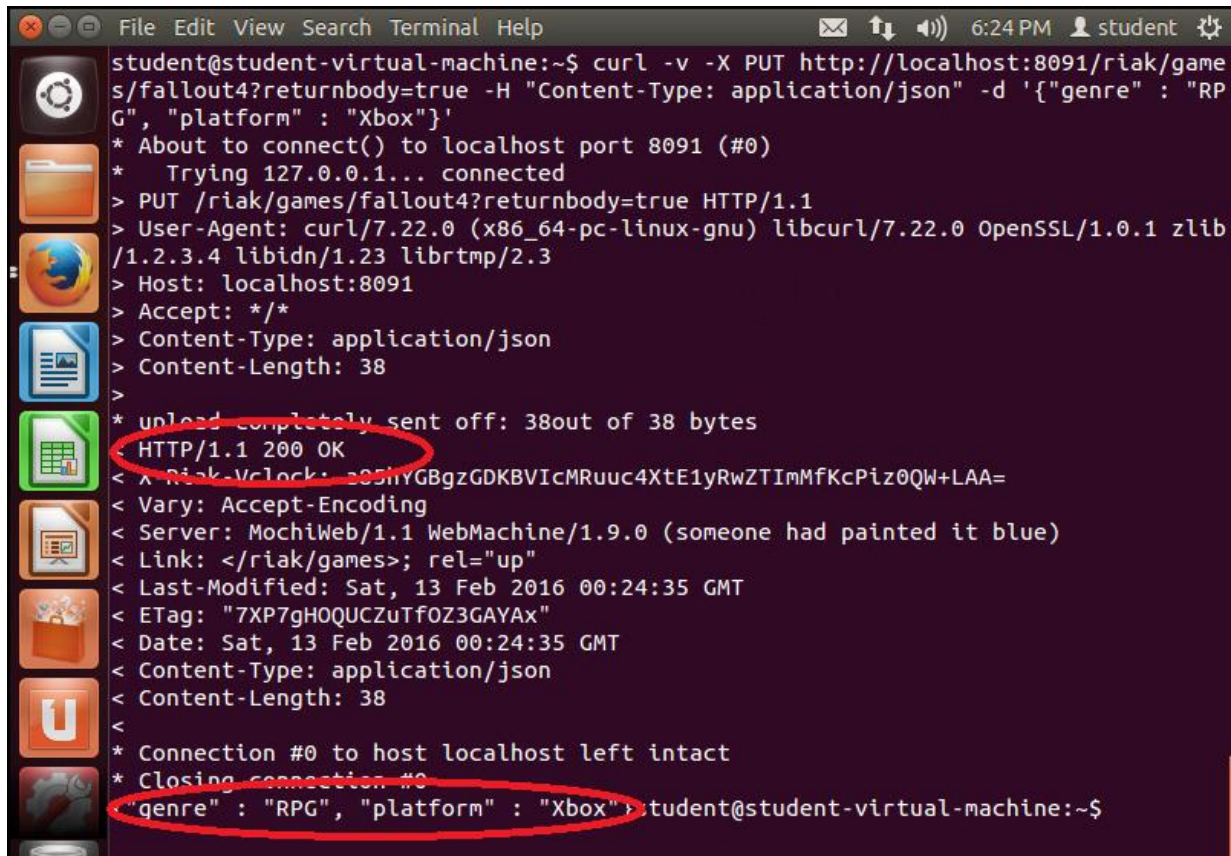
Now, let's further populate our games bucket using some variants of the PUT. Each of these can be useful when working with Riak.

Returning the set results

Optionally, you can return the set results when you create the bucket entry by adding the ?returnbody=true parameter, which we'll test by adding another game, Fallout 4:

```
$ curl -v -X PUT http://localhost:8091/riak/games/fallout4?returnbody=true \
-H "Content-Type: application/json" \
-d '{"genre": "RPG", "platform": "XBox"}'
```

This time you'll see a 200 code and it will return what was added to the bucket.



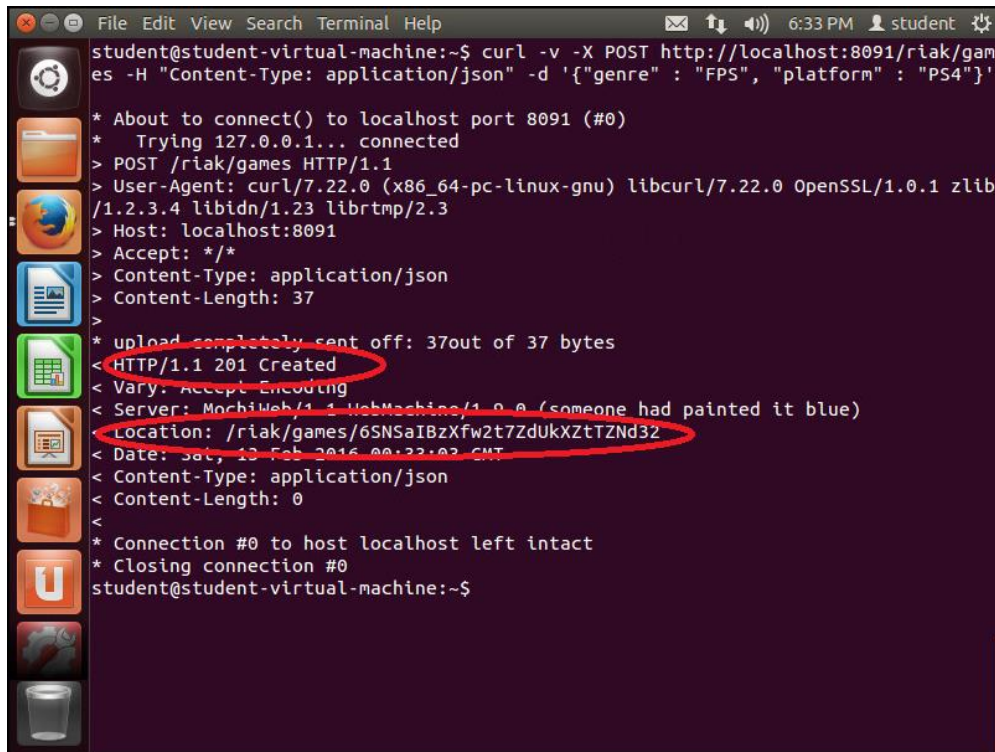
```
student@student-virtual-machine:~$ curl -v -X PUT http://localhost:8091/riak/games/fallout4?returnbody=true -H "Content-Type: application/json" -d '{"genre": "RPG", "platform": "Xbox"}'
* About to connect() to localhost port 8091 (#0)
*   Trying 127.0.0.1... connected
> PUT /riak/games/fallout4?returnbody=true HTTP/1.1
> User-Agent: curl/7.22.0 (x86_64-pc-linux-gnu) libcurl/7.22.0 OpenSSL/1.0.1 zlib/1.2.3.4 libidn/1.23 librtmp/2.3
> Host: localhost:8091
> Accept: */*
> Content-Type: application/json
> Content-Length: 38
>
* upload completely sent off: 38out of 38 bytes
< HTTP/1.1 200 OK
< X-Riak-Checksum: 205hyGBgzGDKBVIcMRuuc4XtE1yRwZTImMfKcPiz0QW+LAA=
< Vary: Accept-Encoding
< Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted it blue)
< Link: </riak/games>; rel="up"
< Last-Modified: Sat, 13 Feb 2016 00:24:35 GMT
< ETag: "7XP7gH0QUCZuTfOZ3GAYAx"
< Date: Sat, 13 Feb 2016 00:24:35 GMT
< Content-Type: application/json
< Content-Length: 38
<
* Connection #0 to host localhost left intact
* Closing connection #0
{"genre": "RPG", "platform": "Xbox"}student@student-virtual-machine:~$
```

Auto-generating a key name

If you do not care about the key name, Riak will generate one when using POST (instead of PUT).

```
$ curl -i -X POST http://localhost:8091/riak/games \
-H "Content-Type: application/json" \
-d '{"genre": "FPS", "Platform": "PS4"}'
```

The generated key will be in the header under Location—also note the 201 success code in the header.



```
student@student-virtual-machine:~$ curl -v -X POST http://localhost:8091/riak/games -H "Content-Type: application/json" -d '{"genre": "FPS", "platform": "PS4"}'
* About to connect() to localhost port 8091 (#0)
*   Trying 127.0.0.1... connected
> POST /riak/games HTTP/1.1
> User-Agent: curl/7.22.0 (x86_64-pc-linux-gnu) libcurl/7.22.0 OpenSSL/1.0.1 zlib/1.2.3.4 libidn/1.23 librtmp/2.3
> Host: localhost:8091
> Accept: */*
> Content-Type: application/json
> Content-Length: 37
* upload completely sent off: 37out of 37 bytes
< HTTP/1.1 201 Created
< Vary: Accept-Encoding
< Server: MochiWeb/1.1-WebMachine/1.0.0 (someone had painted it blue)
< Location: /riak/games/6SNSaIBzXfw2t7ZdUkXZtTZNd32
< Date: Sat, 13 Feb 2016 00:22:02 GMT
< Content-Type: application/json
< Content-Length: 0
* Connection #0 to host localhost left intact
* Closing connection #0
student@student-virtual-machine:~$
```

A GET request (cURL's default if left unspecified) to that location will retrieve the value.

```
$ curl http://localhost:8091/riak/games/6SNSaIBzXfw2t7ZdUkXZtTZNd32
student@student-virtual-machine:~$ curl http://localhost:8091/riak/games/6SNSaIBzXfw2t7ZdUkXZtTZNd32
{"genre": "FPS", "platform": "PS4"}student@student-virtual-machine:~$
```

DELETE will remove it.

```
$ curl -i -X DELETE http://localhost:8091/riak/games/6SNSaIBzXfw2t7ZdUkXZtTZNd32
```


DELETE won't return any body, but the HTTP code will be 204 if successful. (Otherwise, as you'd expect, it returns a 404.)

```
student@student-virtual-machine:~$ curl -i -X DELETE http://localhost:8091/riak/games/6SMS5TP2vFw2+77dUkXZtTZNd32
HTTP/1.1 204 No Content
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted it blue)
Date: Sat, 13 Feb 2016 00:40:47 GMT
Content-Type: application/json
Content-Length: 0
```

LINKS

Links are metadata that associate one key to other keys. The basic structure is this:

Link: </riak/bucket/key>; riaktag=\"whatever\"

The key to where this value links is in pointy brackets (<...>), followed by a semicolon and then a tag describing how the link relates to this value (it can be whatever string we like).

Link Walking

Our Video Game collection is stored on shelves. To keep track of which game is on which shelf, we'll use a link. Shelf 1 contains Fallout 4 by linking to its key (this also creates a new bucket named shelves). The shelf is labeled 1, so include that value as JSON data.

```
$ curl -X PUT http://localhost:8091/riak/shelves/1 \
-H "Content-Type: application/json" \
-H "Link: </riak/games/fallout4>; riaktag=\"contains\"" \
-d '{"shelf" : 1}'
```

```
student@student-virtual-machine:~$ curl -X PUT http://localhost:8091/riak/shelves/1 -H "Content-Type: application/json" -H "Link:</riak/games/fallout4>; riaktag=\"contains\"" -d '{"shelf" : 1}'
student@student-virtual-machine:~$
```

Note that this link relationship is one-directional. In effect, the shelf we've just created knows that Fallout 4 is stored on it, but no changes have been made to our fallout4 bucket. We can confirm this by pulling up fallout4's data and checking that there have been no changes to the Link headers.

```
$ curl -i http://localhost:8091/riak/games/fallout4
```

```
student@student-virtual-machine:~$ curl -i http://localhost:8091/riak/games/fallout4
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVIcMRuuc4XtE1yRwZTImMfKcPiz0QW+LAA=
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.9.0 (someone had painted it blue)
Link: </riak/games>; rel="up"
Last-Modified: Sat, 13 Feb 2016 00:24:35 GMT
ETag: "7XP7gHOQUCZuTf0Z3GAYAx"
Date: Sat, 13 Feb 2016 01:05:42 GMT
Content-Type: application/json
Content-Length: 38
{"genre" : "RPG", "platform" : "Xbox"}
```

You can have as many metadata Links as necessary, separated by commas. We'll put FIFA2016 on shelf 2 and also point to shelf 1 tagged with next_to so we know that it's nearby.

```
$ curl -X PUT http://localhost:8091/riak/shelves/2 \
-H "Content-Type: application/json" \
-H "Link:</riak/games/fifa2016>;riaktag=\"contains\",
</riak/shelves/1>;riaktag=\"next_to\""" \
-d '{"shelf" : 2}'
```

```
student@student-virtual-machine:~$ curl -X PUT http://localhost:8091/riak/shelves
/2 -H "Content-Type: application/json" -H "Link:</riak/games/fifa2016>; riaktag=
\"contains\", </riak/shelves/1>;riaktag=\"next_to\""" -d '{"shelf" : 1}'
student@student-virtual-machine:~$
```

What makes Links special in Riak is *link walking* (and a more powerful variant, linked MapReduce queries, which we investigate in part 2). Getting the linked data is achieved by appending a *link spec* to the URL that is structured like this: `/_/_/_`.

The underscores (`_`) in the URL represent wildcards to each of the link criteria: bucket, tag, keep. So, let's retrieve all links from shelf 1.

```
$ curl http://localhost:8091/riak/shelves/1/_/_/_
```

```
student@student-virtual-machine:~$ curl http://localhost:8091/riak/shelves/1/_/_/_
-
--aK9rG0Ss71CIUDVfNw0zRBxbpu9
Content-Type: multipart/mixed; boundary=NfAoiF01bNloiCHTwCrIlyPzChF

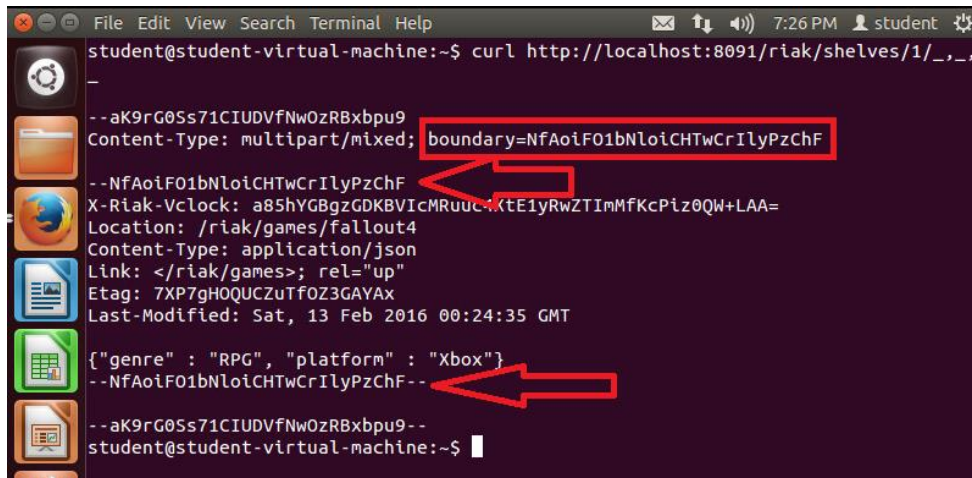
--NfAoiF01bNloiCHTwCrIlyPzChF
X-Riak-Vclock: a85hYGBgzGDKBVIcMRuuc4XtE1yRwZTImMfKcPiz0QW+LAA=
Location: /riak/games/fallout4
Content-Type: application/json
Link: </riak/games>; rel="up"
Etag: 7XP7gH0QUCZuTfOZ3GAYAx
Last-Modified: Sat, 13 Feb 2016 00:24:35 GMT

{"genre" : "RPG", "platform" : "Xbox"}
--NfAoiF01bNloiCHTwCrIlyPzChF--

--aK9rG0Ss71CIUDVfNw0zRBxbpu9--
student@student-virtual-machine:~$
```

It returns a multipart/mixed dump of headers plus bodies of all linked keys/values.

If you're not familiar with reading the multipart/mixed MIME type, the Content-Type definition describes a boundary string, which denotes the beginning and end of some HTTP header and body data:



```
student@student-virtual-machine:~$ curl http://localhost:8091/riak/shelves/1/_,_
--aK9rG0Ss71CIUDVfNwOzRBxbpu9
Content-Type: multipart/mixed; boundary=NfAoiF01bNloiCHTwCrIlyPzChF
--NfAoiF01bNloiCHTwCrIlyPzChF
X-Riak-Vclock: a85hYGBgzGDKBVicMRUUC...tE1yRwZTImMfKcPiz0QW+LAA=
Location: /riak/games/fallout4
Content-Type: application/json
Link: </riak/games>; rel="up"
Etag: 7XP7gh0QUCZuTfOZ3GAYAx
Last-Modified: Sat, 13 Feb 2016 00:24:35 GMT
{"genre" : "RPG", "platform" : "Xbox"}
--NfAoiF01bNloiCHTwCrIlyPzChF--
--aK9rG0Ss71CIUDVfNwOzRBxbpu9--
student@student-virtual-machine:~$
```

In our case, the data is what shelf 1 links to: Fallout 4. (You may have noticed that the headers returned don't actually display the link information. This is OK; that data is still stored under the linked-to key.)

When link walking, we can replace the underscores in the link spec to filter only values we want. Shelf 2 has two links, so performing a link spec request will return both the game FIFA2016 contained on the shelf and the shelf 1 next_to it. To specify only following the games bucket, replace the first underscore with the bucket name.

```
$ curl http://localhost:8091/riak/shelves/2/games,_,_
```

Or follow the shelf next to this one by populating the tag criteria.

```
$ curl http://localhost:8091/riak/shelves/2/_next_to,_
```

The final underscore—*keep*—accepts a 1 or 0. *keep* is useful when following second-order links, or links following other links, which you can do by just appending another link spec. Let's follow the keys next_to shelf 2, which will return shelf 1.

Next, we walk to the games linked to shelf 1. Since we set keep to 0, Riak will not return the intermediate step (the shelf 1 data). It will return only Fallout 4's information, which is next to FIFA2016's shelf.

```
$ curl http://localhost:8091/riak/shelves/2/_next_to,0/games,_,_
student@student-virtual-machine:~$ curl http://localhost:8091/riak/shelves/2/_next_to,0/games,_,_

--NqRJbXHEOGiumxCwBmdsWjBtJwb
Content-Type: multipart/mixed; boundary=K6Pv9nn1TveK9NSwxZpEfEkUc2q

--K6Pv9nn1TveK9NSwxZpEfEkUc2q
X-Riak-Vclock: a85hVGBqzGDKBVTcMRuuc4XtE1yRwZTImMfKcPiz0QW+LAA=
Location: /riak/games/fallout4
Content-Type: application/json
Link: </riak/games>; rel="up"
Etag: 7XP7gH0QUCZuTfOZ3GAYAx
Last-Modified: Sat, 13 Feb 2016 00:24:35 GMT

{"genre" : "RPG", "platform" : "Xbox"}
--K6Pv9nn1TveK9NSwxZpEfEkUc2q--

--NqRJbXHEOGiumxCwBmdsWjBtJwb--
```

MIME types in Riak

Riak stores everything as a binary-encoded value, just like normal HTTP. The MIME type gives the binary data context. (All of our examples so far have only used text.) MIME types are stored on the Riak server but are really just a flag to the client so that when it downloads the binary data, it knows how to render it.

If we wanted our video game library to keep an image of our games, we need only use the data-binary flag on the curl command to upload an image to the server and specify the MIME type as image/jpeg. Then add a link back to the /games/fifa2016 key so we know which game we are looking at.

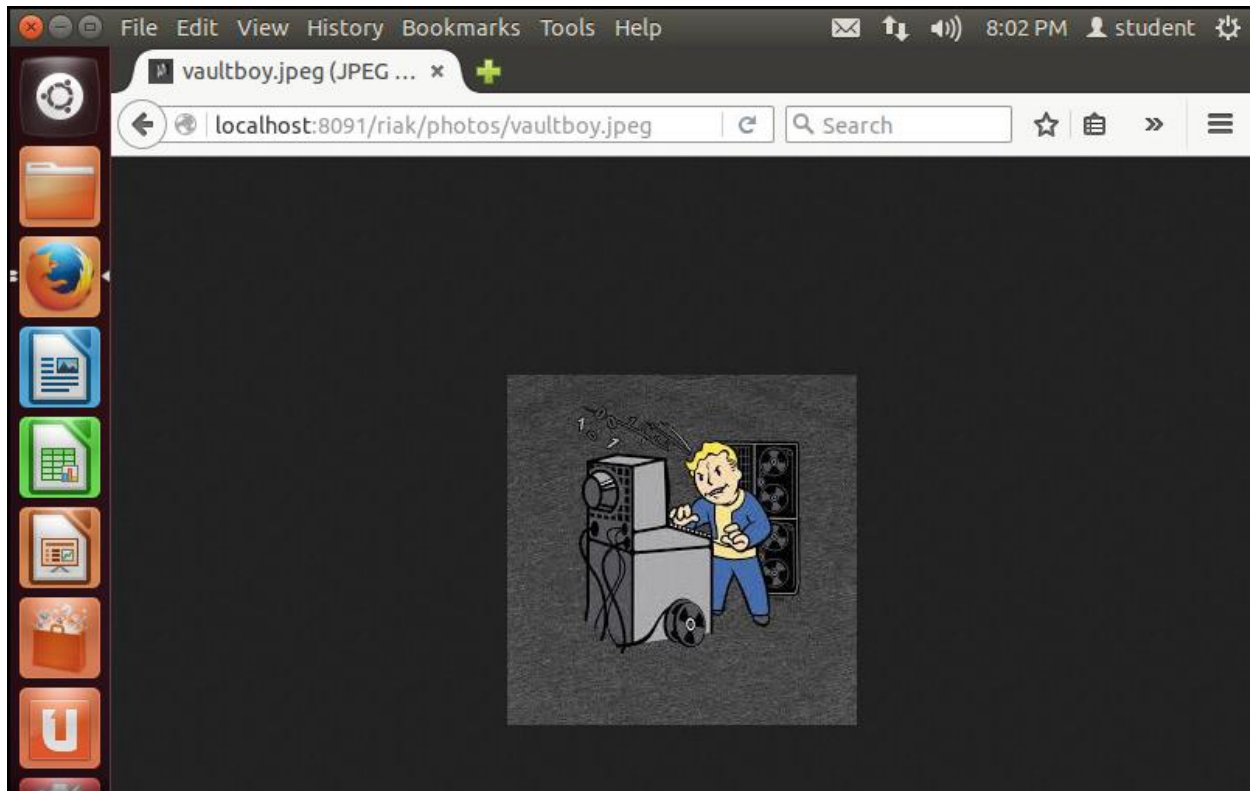
Change your directory to ~/riak-1.2.1/dev and you will find an image called vaultboy.jpeg. Upload it to our Riak DB by entering the following: (

```
$ curl -X PUT http://localhost:8091/riak/photos/vaultboy.jpeg \
-H "Content-type: image/jpeg" \
-H "Link: </riak/games/fallout4>; riaktag=\"photo\"" \
--data-binary @vaultboy.jpeg

student@student-virtual-machine:~/riak-1.2.1/dev$ curl -X PUT http://localhost:8091/riak/photos/vaultboy.jpeg \
> -H "Content-type: image/jpeg" \
> -H "Link:</riak/games/fallout4>;riaktag=\"photo\"" \
> --data-binary @vaultboy.jpeg
student@student-virtual-machine:~/riak-1.2.1/dev$
```

Now visit the URL in a web browser, which will be delivered and rendered exactly as you'd expect any web client-server request to function.

<http://localhost:8091/riak/photos/vaultboy.jpeg>



Since we pointed the image to `/games/fallout4`, we could link walk from the image key to Fallout 4 but not vice versa. Unlike a relational database, there is no “has a” or “is a” rule concerning links. You link the direction you need to walk. If your use case will require accessing image data from the games bucket, a link should exist on that object instead (or in addition to).

So far, we’ve covered standard key-value practice with some links thrown in. When designing a Riak schema, your plan should be somewhere in between a caching system and PostgreSQL. You will break up your data into different logical classifications (buckets), and values can tacitly relate to each other. But you will not go so far as to normalize into fine components like you would in a relational database, since Riak performs no sense of relational joins to recompose values.

CONTINUE TO NEXT PAGE FOR WORK TO SUBMIT

Work on your own to submit

You will create a movie rental Riak key-value store. The bucket should be movies, and the value should be json with releasedate (i.e. 2016), runningtime (i.e. 1:15), and genre (i.e. comedy, drama, horror). The key should be the title of the movie all together in Pascal case (TheLordOfTheRings).

1. Add 6 movies to the database from at least 2 genres (comedy, drama, horror). You can use imdb.com to find the information about your selected movies.
2. Delete one of the movie records.
3. Our movie rental business has 3 branches (East, West, South). Create these branches. The bucket should be branches. The value should be json with the name of the branch. Link each of the remaining five movies to at least one of the branches. At least one of the movies should link to two branches (i.e. that movie can be found at 2 stores). Come up with an intuitive riaktag such as holds. For example, we can lookup if a customer calls looking for a particular movie, and that movie is available at another branch.
4. Download a picture for one of the movies and add it to a bucket named images with the key being the picture name. Then link it to the movie. Images can be found by searching Google Images and using the Advanced Search feature set to usage rights to "free to use/share" (so as not to violate any copyrights).
5. Run queries listing all the buckets, all the movies found in each branch, and finally of the movie with the picture and its branch.
6. All the curl commands need to be tested first on your system and then saved as lab2a.sh. The hostname in the file should be changed to http://riak:8098/riak/ This file and the picture used in Step 4 will be pushed to GitHub for submission.