The Basics

Contents

The basic actions of Riak are the same CRUD (Create, Read, Update, Delete) operations as any key/value store.

Object/Key Operations

Riak organizes data into Buckets, Keys, and Values. Values (or objects) are identifiable by a unique key, and each key/value pair is stored in a bucket. Buckets are essentially a flat namespace in Riak and have little significance beyond their ability to allow the same key name to exist in multiple buckets and to provide some per-bucket configurability for things like replication factor and pre/post-commit hooks.

Most of the interactions you'll have with Riak will be setting or retrieving the value of a key. Although we'll use the Riak HTTP API for illustrative purposes, Riak has supported client libraries for Erlang, Java, PHP, Python, Ruby and C/C++. In addition, there arecommunity-supported projects for .NET, Node.js, Python, Perl, Clojure, Scala, Smalltalk, and many others.

Read an Object

Here is the basic command form for retrieving a specific key from a bucket.

Shell

```
GET /buckets/BUCKET/keys/KEY
```

The body of the response will contain the contents of the object (if it exists).

Riak understands many HTTP-defined headers, like `Accept` for content-type negotiation (relevant when dealing with siblings, see the sibling examples for the HTTP API, and `If-None-Match`/`ETag` and `If-Modified-Since`/`Last-Modified` for conditional requests.

Riak also accepts many query parameters, including `r` for setting the R-value for this GET request (R values describe how many replicas need to agree when retrieving an existing object in order to return a successful response). If you omit the the `r` query parameter, Riak defaults to `r=2`.

Normal response codes:

- `200 OK`
- `300 Multiple Choices`
- `304 Not Modified`

Typical error codes:

- `404 Not Found`

So, with that in mind, try this command. This will request (GET) the key `doc2` from the bucket `test.`

HTTP

```
$ curl -v http://localhost:8098/buckets/test/keys/doc2
```

This should return a `404 Not Found` as the key `doc2` does not exist (you haven't created it yet).

Store an object

Your application will often have its own method of generating the keys for its data. If so, storing that data is easy. The basic request looks like this.

Note that this is not the only URL format available. Alternate forms can be found in the HTTP API.

Shell

```
PUT /buckets/BUCKET/keys/KEY
```

`POST` is also a valid verb, for compatibility's sake.

There is no need to intentionally create buckets in Riak. They pop into existence when keys are added to them, and disappear when all keys have been removed from them.

Some request headers are required for PUTs:

- `Content-Type` must be set for the stored object. Set what you expect to receive back when next requesting it.
- `X-Riak-Vclock` if the object already exists, the vector clock attached to the object when read; if the object is new, this header may be omitted

Other request headers are optional for PUTs:

- `X-Riak-Meta-YOUR_HEADER` any additional metadata headers that should be stored with the object (eg. `X-Riak-Meta-FirstName`).
- `Link` user and system-defined links to other resources. Read more about Links.

Similar to how GET requests support the `r` query parameter, `PUT` requests also support these parameters:

- `r` (default is `2`) how many replicas need to agree when retrieving an existing object before the write
- `w` (default is `2`) how many replicas to write to before returning a successful response
- `dw` (default is `0`) how many replicas to commit to durable storage before returning a successful response
- `returnbody` (boolean, default is `false`) whether to return the contents of the stored object

Normal status codes:

- `200 OK`
- `204 No Content`
- `300 Multiple Choices`

If `returnbody=true`, any of the response headers expected from a `GET` request may be present. Like a `GET` request, `300 Multiple Choices` may be returned if siblings existed or were created as part of the operation, and the response can be dealt with similarly.

Let's give it a shot. Try running this in a terminal.

```
$ curl -v -XPUT http://localhost:8098/buckets/test/keys/doc?retu
  -H "X-Riak-Vclock: a85hYGBgzGDKBVIszMk55zKYEhnzWBlKIniO8mUBAA=
  -H "Content-Type: application/json" \
  -d '{"bar":"baz"}'
```

Store a new object and assign random key

If your application would rather leave key-generation up to Riak, issue a `POST` request to the bucket URL instead of a PUT to a bucket/key pair:

Shell

```
POST /buckets/BUCKET/keys
```

If you don't pass Riak a “key” name after the bucket, it will know to create one for you.

Supported headers are the same as for bucket/key PUT requests, though `X-Riak-Vclock` will never be relevant for these POST requests. Supported query parameters are also the same as for bucket/key PUT requests.

Normal status codes:

- `201 Created`

This command will store an object, in the bucket `test` and assign it a key:

```
$ curl -v -XPOST http://localhost:8098/buckets/test/keys \
  -H 'Content-Type: text/plain' \
  -d 'this is a test'
```

In the output, the `Location` header will give the you key for that object. To view the newly created object, go to `http://localhost:8098/*_Location_*` in your browser.

If you've done it correctly, you should see the value (which is “this is a test”).

Delete an object

The delete command, as you can probably guess, follows a predictable pattern and looks like this:

Shell

```
DELETE /buckets/BUCKET/keys/KEY
```

The normal response codes for a `DELETE` operations are `204 No Content` and `404 Not Found`

404 responses are *normal*, in the sense that `DELETE` operations are idempotent and not finding the resource has the same effect as deleting it.

Try this:

HTTP

```
$ curl -v -XDELETE http://localhost:8098/buckets/test/keys/t
```

Bucket Properties and Operations

Buckets are essentially a flat namespace in Riak. They allow the same key name to exist in multiple buckets and provide some per-bucket configurability.

How Many Buckets Can I Have?

Buckets come with virtually no cost *except for when you modify the default bucket properties*. Modified bucket properties are gossiped around the cluster and therefore add to the amount of data sent around the network. In other words, buckets using the default bucket properties are free.

Setting a Bucket's Properties

However, in addition to providing a namespace for keys, the properties of a bucket also define some of the behaviors that Riak will implement for the values stored in the bucket.

To set these properties, issue a `PUT` to the bucket's URL:

Shell

```
PUT /buckets/BUCKET/props
```

The body of the request should be a JSON object with a single entry “props”. Unmodified bucket properties may be omitted.

Important headers:

- `Content-Type: application/json`

The most important properties to consider for your bucket are:

- `n_val` (defaults to `3`) the number of replicas for objects in this bucket; `n_val` should be an integer greater than 0 and less than the number of partitions in the ring.

  > If you change `n_val` after keys have been added to the bucket it may result in failed reads. The new value may not be replicated to all of the appropriate partitions.

- `allow_mult` (boolean, defaults to `false`) With `allow_mult` set to `false`, clients will only get the most-recent-by-timestamp object. Otherwise, Riak maintains any sibling objects caused by concurrent writes (or network partitions).

Let's go ahead and alter the properties of a bucket. The following `PUT` will create a new bucket called `test` with a modified `n_val` of `5`.

```
$ curl -v -XPUT http://localhost:8098/buckets/test/props \
  -H "Content-Type: application/json" \
  -d '{"props":{"n_val":5}}'
```

GET Buckets

Here is how you use the HTTP API to retrieve (or `GET`) the bucket properties and/or keys:

Shell

```
GET /buckets/BUCKET/props
```

The optional query parameters are:

- `props`: `true`|`false` - whether to return the bucket properties (defaults to `true`)
- `keys`: `true`|`false`|`stream` - whether to return the keys stored in

the bucket (defaults to `false`); see the HTTP API's list keys for details about dealing with a `keys=stream` response

With that in mind, go ahead and run this command. This will `GET` the bucket information that we just set with the sample command above:

HTTP

```
$ curl -v http://localhost:8098/buckets/test/props
```

You can also view this bucket information through any browser by going to `http://localhost:8098/buckets/test/props`

So, that's the basics of how the HTTP API works. An in depth reading of the HTTP API page is highly recommended. This will give you details on the headers, parameters, and status that you should keep in mind, even when using a client library.

These May Also Interest You

- Five-Minute Install
- Advanced Secondary Indexes
- Replication Properties
- Advanced Commit Hooks
- Advanced MapReduce
- Advanced Search