Little Riak Core Book

Release 1.0

Mariano Guerra

November 01, 2015

CHAPTER

ONE

STARTING

We are going to build a system on top of riak_core, for that we will use some tools and to avoid copy paste and boilerplate we will use a template to get started.

1.1 Tools

- rebar3: The build tool, click on the link to see how to install it.
- erlang: Our programming language, we assume Erlang version to be at least 17.0

I also assume a unix-like environment with a shell similar to bash or zsh.

1.2 Installing the Template

At this point you should have erlang and rebar3 installed, now let's install the template we are going to use.

```
mkdir -p ~/.config/rebar3/templates
git clone https://github.com/marianoguerra/rebar3_template_riak_core/ ~/.config/rebar3/templates/
```

We just created the folder *~/.config/rebar3/templates* for the templates in case it wasn't there and cloned our template inside of it.

You can read more about rebar3 templates here.

1.3 Creating our Project

Now that we have our tools and our template installed we can start by asking rebar3 to create a new project we will call tanodb using the riak_core template we just installed:

```
rebar3 new rebar3_riak_core name=tanodb
```

If it fails saying it can't find rebar3 check that it's in your *$PATH* environment variable.

The output should be something like this:

```
===> Writing tanodb/apps/tanodb/src/tanodb.app.src
===> Writing tanodb/apps/tanodb/src/tanodb.erl
===> Writing tanodb/apps/tanodb/src/tanodb_app.erl
===> Writing tanodb/apps/tanodb/src/tanodb_sup.erl
===> Writing tanodb/apps/tanodb/src/tanodb_console.erl
===> Writing tanodb/apps/tanodb/src/tanodb_vnode.erl
===> Writing tanodb/rebar.config
===> Writing tanodb/.editorconfig
===> Writing tanodb/.gitignore
===> Writing tanodb/README.rst
===> Writing tanodb/Makefile
===> Writing tanodb/config/nodetool
===> Writing tanodb/config/extended_bin
===> Writing tanodb/config/admin_bin
===> Writing tanodb/config/config.schema
===> Writing tanodb/config/advanced.config
===> Writing tanodb/config/sys.config
===> Writing tanodb/config/vars.config
===> Writing tanodb/config/vars_dev1.config
===> Writing tanodb/config/vars_dev2.config
===> Writing tanodb/config/vars_dev3.config
===> Writing tanodb/config/vm.args
===> Writing tanodb/config/dev1_vm.args
===> Writing tanodb/config/dev2_vm.args
===> Writing tanodb/config/dev3_vm.args
```

1.4 Building and Running

Before explaining what the files mean so you get an idea what just happened let's run it!

```
cd tanodb
rebar3 release
rebar3 run
```

rebar3 release asks rebar3 to build a release of our project, for that it uses a tool called relx.

The initial build may take a while since it has to fetch all the dependencies and build them.

After the release is built (you can check the result by inspecting the folder *_build/default/rel/tanodb/*) we can run it, for this we use a rebar3 plugin called rebar3_run

When we run *rebar3 run* we get some noisy output that should end with something like this:

```
Eshell V7.0  (abort with ^G)
(tanodb@127.0.0.1)1>
```

This is the Erlang shell, something like a REPL connected to our system, we now can test our system by calling *tanodb:ping()* on it.

```
(tanodb@127.0.0.1)1> tanodb:ping().
{pong,1347321821914426127719021955160323408745312813056}
```

The response is the atom *pong* and a huge number that we will explain later, but to make it short, it's the id of the process that replied to us.

1.5 Exploring the Template Files

The template created a lot of files and you are like me, you don't like things that make magic and don't explain what's going on, that's why we will get a brief overview of the files created here.

First this files are created:

```
apps/tanodb/src/tanodb.app.src
apps/tanodb/src/tanodb.erl
apps/tanodb/src/tanodb_app.erl
apps/tanodb/src/tanodb_sup.erl
apps/tanodb/src/tanodb_console.erl
apps/tanodb/src/tanodb_vnode.erl
```

Those are the meat of this project, the source code we start with, if you know a little of Erlang you will recognize them many of them, let's explain them briefly, if you think you need more information I recommend you this awesome book which you can read online: Learn You Some Erlang for great good!

tanodb.app.src This file is "The Application Resource File", you can read it, it's quite self descriptive. You can read more about it in the Building OTP Applications Section of Learn You Some Erlang or in the man page for app in the Erlang documentation.

tanodb.erl This file is the main API of our application, here we expose all the things you can ask our application to do, for now it can only handle the *ping()* command but we will add some more in the future.

tanodb_app.erl This file implements the application behavior it's a set of callbacks that the Erlang runtime calls to start and stop our application.

tanodb_sup.erl This file implements the supervisor behavior it's a set of callbacks that the Erlang runtime calls to build the supervisor hierarchy.

tanodb_console.erl This file is specific to riak_core, it's a set of callbacks that will be called by the *tanodb-admin* command.

tanodb_vnode.erl This file is specific to riak_core, it implements the riak_code_vnode behavior, which is a set of callbacks that riak_core will call to accomplish different tasks, it's the main file we will edit to add new features.

Those were the source code files, but the template also created other files, let's review them

rebar.config This is the file that rebar3 reads to get information about our project like dependencies and build configuration, you can read more about it on the rebar3 documentation

.editorconfig This file describes the coding style for this project, if your text editor understands editorconfig files then it will change it's setting for this project to the ones described in this file, read more about editor config on the editorconfig website

.gitignore A file to tell git which files to ignore from the repository.

README.rst The README of the project

Makefile A make file with some targets that will make it easier to achieve some complex tasks without copying and pasting to much.

config/nodetool An escript that makes it easier to interact with an erlang node from the command line, it will be used by the *tanodb* and *tanodb-admin* commands.

config/extended_bin A template for the *tanodb* command with some changes to support cuttlefish which is the library we use to load and validate our configuration

config/admin_bin A template for the *tanodb-admin* command.

config/config.schema The cuttlefish schema file that describes what configuration our application supports, it starts with some example configuration fields that we will use as the application grows.

config/advanced.config This file is where we configure some advanced things of our application that don't go on our *tanodb.config* file, here we configure riak_core and our logging library

config/sys.config This is a standard Erlang application file, you can read more about it in the Erlang documentation for sys.config

config/vars.config This file contains variables used by relx to build a release, you can read more about it in the rebar3 release documentation

The following files are like vars.config but with slight differences to allow running more than one node on the same machine:

```
config/vars_dev1.config
config/vars_dev2.config
config/vars_dev3.config
```

Normally when you have a cluster for your application one operating system instance runs one instance of your application and you have many operating system instances, but to test the clustering features of riak_core we will build 3 releases of our application using offsets for ports and changing the application name to avoid collisions.

config/vm.args A file used to pass options to the Erlang VM when starting our application.

The following files are like vars_dev*.config but for vm.args:

```
config/dev1_vm.args
config/dev2_vm.args
config/dev3_vm.args
```

Those are all the files, follow the links to know more about them.

1.6 Playing with Clustering

Before starting to add features, let's first play with clustering so we understand all those config files above work.

Build 3 releases that can run on the same machine:

```
make devrel
```

This will build 3 releases of the application using different parameters (the dev1, dev2 and dev3 files we saw earlier) and will place them under:

```
_build/dev1
_build/dev2
_build/dev3
```

This is achived by using the profiles feature from rebar3.

Now open 3 consoles and run the following commands one on each console:

```
make dev1-console
make dev2-console
make dev3-console
```

This will start the 3 nodes but the won't know about each other, for them to know about each other we need to "join" them, that is to tell one of them about the other two, this is achieved using the tanodb-admin command, here is how you should run it manually (don't run them):

```
_build/dev2/rel/tanodb/bin/tanodb-admin cluster join tanodb1@127.0.0.1
_build/dev3/rel/tanodb/bin/tanodb-admin cluster join tanodb1@127.0.0.1
```

We tell dev2 and dev3 to join tanodb1 (dev1), to make this easier and less error prone run the following command:

```
make devrel-join
```

Now let's check the status of the cluster:

```
make devrel-status
```

You can read the Makefile to get an idea of what those commands do, in this case devrel-status does the following:

```
_build/dev1/rel/tanodb/bin/tanodb-admin member-status
```

You should see something like this:

```
================================= Membership ==================================
Status     Ring    Pending    Node
-------------------------------------------------------------------------------
joining     0.0%      --      'tanodb2@127.0.0.1'
joining     0.0%      --      'tanodb3@127.0.0.1'
valid     100.0%      --      'tanodb1@127.0.0.1'
-------------------------------------------------------------------------------
Valid:1 / Leaving:0 / Exiting:0 / Joining:2 / Down:0
```

It should say that 3 nodes are joining, now check the cluster plan:

```
make devrel-cluster-plan
```

The output should be something like this:

```
================================ Staged Changes ================================
Action         Details(s)
-------------------------------------------------------------------------------
join           'tanodb2@127.0.0.1'
join           'tanodb3@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%     34.4%    'tanodb1@127.0.0.1'
valid       0.0%     32.8%    'tanodb2@127.0.0.1'
valid       0.0%     32.8%    'tanodb3@127.0.0.1'
-------------------------------------------------------------------------------
Valid:3 / Leaving:0 / Exiting:0 / Joining:0 / Down:0

WARNING: Not all replicas will be on distinct nodes

Transfers resulting from cluster changes: 42
  21 transfers from 'tanodb1@127.0.0.1' to 'tanodb3@127.0.0.1'
  21 transfers from 'tanodb1@127.0.0.1' to 'tanodb2@127.0.0.1'
```

Now we can commit the plan:

```
make devrel-cluster-commit
```

Which should say something like:

```
Cluster changes committed
```

Now riak_core started an internal process to join the nodes to the cluster, this involve some complex processes that we will explore in the following chapters.

You should see on the consoles where the nodes are running that some logging is happening describing the process.

Check the status of the cluster again:

```
make devrel-status
```

You can see the vnodes transferring, this means the content of some virtual nodes on one tanodb node are being transfered to another tanodb node:

```
================================ Membership ==================================
Status     Ring    Pending    Node
-------------------------------------------------------------------------------
valid      75.0%     34.4%    'tanodb1@127.0.0.1'
valid       9.4%     32.8%    'tanodb2@127.0.0.1'
valid       7.8%     32.8%    'tanodb3@127.0.0.1'
-------------------------------------------------------------------------------
Valid:3 / Leaving:0 / Exiting:0 / Joining:0 / Down:0
```

At some point you should see something like this, which means that the nodes are joined and balanced:

```
================================ Membership ==================================
Status     Ring    Pending    Node
-------------------------------------------------------------------------------
valid      34.4%      --      'tanodb1@127.0.0.1'
valid      32.8%      --      'tanodb2@127.0.0.1'
valid      32.8%      --      'tanodb3@127.0.0.1'
-------------------------------------------------------------------------------
Valid:3 / Leaving:0 / Exiting:0 / Joining:0 / Down:0
```

When you are bored you can stop them:

```
make devrel-stop
```

1.7 Building a Production Release

Even when our application doesn't have the features to merit a production release we are going to learn how to do it here since you can later do it at any step and get a full release of the app:

```
rebar3 as prod release
```

In that command we as rebar3 to run the release task using the prod profile, which has some configuration differences with the dev profiles we use so that it builds something we can unpack and run on another operating system without installing anything.

In my case I'm developing this on ubuntu, to show you that it works I will copy the release to a clean ubuntu 15.04 Virtualbox and run it there:

```
mkdir vm-ubuntu-1504
cd vm-ubuntu-1504
```

Inside I will create a file called *Vagrantfile* with the following content:

```
Vagrant.configure(2) do |config|
  config.vm.box = "ubuntu/vivid64"
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "1024"
  end
end
```

And then run:

```
vagrant up
```

To start the virtual machine.

Now let's package our release and copy it to a place where the VM can see it:

```
cd _build/prod/rel
tar -czf tanodb.tgz tanodb
cd -
mv _build/prod/rel/tanodb.tgz vm-ubuntu-1504
```

Let's ssh into the virtual machine:

```
export TERM=xterm
vagrant ssh
```

Inside the virtual machine run:

```
cp /vagrant/tanodb.tgz .
tar -xzf tanodb.tgz
./tanodb/bin/tanodb console
```

And it runs!

Note: You should build the production release on the same operating system version you are intending to run it to avoid version problems, the main source of headaches are C extensions disagreeing on libc versions and similar.

So, even when you could build it on a version that is close and test it it's better to build releases on the same version to avoid problems. More so if you are packaging the Erlang runtime with the release as we are doing here.

1.8 Wrapping Up

Now you know how to create a riak_core app from a template, how to build a release and run it, how to build releases for a development cluster, run the nodes, join them and inspect the cluster status and how to build a production release and run it on a fresh server.

Quite a lot for the first chapter I would say...

CHAPTER

TWO

PING AS A SERVICE (PAAS)

Note: This chapter and the following ones will reference a real project in a github repository, follow the links to see the details of what is written here.

2.1 Setting Up

After setting up our project we will now expose ping as a REST API, for that we will use The Cowboy Web Server.

For JSON parsing we will use jsx.

Here is the commit to add the dependencies.

We also add the rebar.lock file to make our builds reproducible.

Just in case we want to use another json library later and to simplify the calls we wrap the json library in our own module.

Finally we create a cowboy rest handler for our ping resource, tanodb_http_ping.erl and initialize cowboy in tanodb_app.

2.2 Testing it

To interact with the REST API we will use httpie since it's simpler to read (and write) than curl, check how to install it on the httpie website

First we build it and run it as usual (this may be the last time I show you explicitly how to do it, so learn it :):

```
rebar3 release
rebar3 run
```

Now on another shell we will make an HTTP request to our ping resource:

```
http localhost:8080/ping
```

And this is what I get:

```
HTTP/1.1 200 OK
content-length: 59
content-type: application/json
date: Thu, 29 Oct 2015 19:07:23 GMT
server: Cowboy

{
"pong": "98194641258170039816810074698125265383132967731 2"
}
```

If you run it more times the value of the *pong* attribute should change, since the vnode that handles the request is defined by the time that the request is made.

2.3 Changing Some Configuration

Let's say we would like to run the server on another port, for that we need to change the configuration, we can do this by editing the file:

```
_build/default/rel/tanodb/etc/tanodb.conf
```

Search for 8080 and change it for 8081, save and close and stop the server if you are running it.

Now we will run it again but manually to avoid rebar3 from overriding our change:

```
./_build/default/rel/tanodb/bin/tanodb console
```

And try a request to see if the port is actually changed:

```
http localhost:8081/ping
```

And this is what I get:

```
HTTP/1.1 200 OK
content-length: 60
content-type: application/json
date: Thu, 29 Oct 2015 19:18:03 GMT
server: Cowboy

{
    "pong": "1187470080331358621040493926581979953470445191168"
}
```

Read tanodb.config to see all the available options, this file is generated using cuttlefish which takes a schema we define and uses it to generate the default config file and later to validate the config file on startup and generate configuration files that the Erlang runtime understands.

If you are curious you can see the generated config files after running the server at least once under *_build/default/rel/tanodb/generated.configs/*

CHAPTER

THREE

METRICS

3.1 API Metrics

Since this is meant to be a production system we can't be far along until we add metrics, for this we will use exometer which is already a dependency of riak_core so we don't need to add it.

We start by defining a module named tanodb_metrics.

The main functions we care about are :

init/0 which will initialize all the metrics when the app starts, we will add more metrics here as we add more features.

core_ping/0 should be called to register metrics about calls to tanodb:ping/0

all/0 returns the current status of all metrics.

To make the metrics actually work we need to call tanodb_metrics:init/0 when we start the application and tanodb_metrics:core_ping/0 each time tanodb:ping/0 is called.

3.1.1 Test It

Stop, build a release and run the server (I won't tell you how from now on, check previous chapters to see how).

On the server shell run:

```
(tanodb@127.0.0.1)1> tanodb_metrics:all().
[{tanodb,[

 ...

 {core,[{ping,[{count,0},{one,0}]}]}]

(tanodb@127.0.0.1)2> tanodb:ping().
{pong,593735040165679310520246963290989976735222595584}

(tanodb@127.0.0.1)3> tanodb_metrics:all().
[{tanodb,[

 ...

 {core,[{ping,[{count,1},{one,1}]}]}]

(tanodb@127.0.0.1)4>
```

The ... are there to skip a lot of metrics about riak_core itself that are quite useful but not important at this point.

Let's see the shell session step by step, first we call tanodb_metrics:all() and get the core ping metrics, in this case count and one are 0 since we didn't call ping yet.

```
(tanodb@127.0.0.1)1> tanodb_metrics:all().
[{tanodb,[

 ...

 {core,[{ping,[{count,0},{one,0}]}]}]
```

Then we call ping once.

```
(tanodb@127.0.0.1)2> tanodb:ping().
{pong,593735040165679310520246963290989976735222595584}
```

And ask for the metrics again, we can see that now it registered our call.

```
(tanodb@127.0.0.1)3> tanodb_metrics:all().
[{tanodb,[

 ...

 {core,[{ping,[{count,1},{one,1}]}]}]
```

3.2 Erlang Runtime Metrics

Until now we have metrics for riak_core and for our API, it would be useful to have some metrics about the Erlang Runtime, like memory, GC, processes, schedulers etc. For that we will use a really nice library called recon which unified all the information gathering behind a nice API.

We start by adding recon as a dependency, then we create the function tanodb_metrics:node_stats/0 and add it to tanodb_metrics:all/0.

3.2.1 Test it

Stop, build a release and run. In the shell run:

```
(tanodb@127.0.0.1)1> tanodb_metrics:all().
[{tanodb,[

    ...

 {node,[{abs,[{process_count,377},
              {run_queue,0},
              {error_logger_queue_len,0},
              {memory_total,30418240},
              {memory_procs,11745496},
              {memory_atoms,458994},
              {memory_bin,232112},
              {memory_ets,1470872}]},
        {inc,[{bytes_in,11737},
              {bytes_out,2470},
              {gc_count,7},
              {gc_words_reclaimed,29948},
              {reductions,2601390},
              {scheduler_usage,[{1,0.9291112866248371},
                                {2,0.04754016011809648},
                                {3,0.04615958261183974},
                                {4,0.03682005933534583}]}]}]},
 {core,[{ping,[{count,0},{one,0}]}]}]
```

The metrics should be self explanatory, check the recon documentation for details.

3.3 Web Server Metrics (Cowboy)

We will start with some generic web server metrics, you can add specific ones with what you have learned in this chapter and by reading the exometer docs.

For the generic metrics we will use cowboy_exometer which is a module I just wrote since it was quite generic :)

We start by adding the cowboy_exometer dependency, this module exposes a middleware and a response hook to register metrics on all requests, for that we need to initialize it providing the endpoints we care about and when we want to collect the metrics we call cowboy_exometer:stats/1 passing the same endpoints we passed on init.

Finally we need to tell cowboy that we will add a middleware and a response hook.

3.3.1 Test it

After all of this, stop, build, run and make some requests:

```
http localhost:8080/ping
```

and then on the node shell ask for the metrics:

```
(tanodb@127.0.0.1)1> tanodb_metrics:all().
[{tanodb,[

    ...

 {http,[{resp,[{by_code,[{200,[{count,1},{one,1}]},
                         {201,[{count,0},{one,0}]},
                         {202,[{count,0},{one,0}]},
                         {203,[{count,0},{one,0}]},
                         {204,[{count,0},{one,0}]},
                         {205,[{count,0},{one,0}]},
                         {206,[{count,0},{one,0}]},
                         {300,[{count,0},{one,0}]},
                         {301,[{count,0},{one,0}]},
                         {302,[{count,0},{one,0}]},
                         {303,[{count,0},{one,0}]},
                         {304,[{count,0},{one,0}]},
                         {305,[{count,0},{one,0}]},
                         {306,[{count,0},{one,...}]},
                         {307,[{count,...},{...}]},
                         {308,[{...}|...]},
                         {400,[...]},
                         {401,...},
                         {...}|...]}]},
        {req,[{time,[{<<"ping">>,
                      [{n,3},
                       {mean,44126},
                       {min,44126},
                       {max,44126},
                       {median,44126},
                       {50,0},
                       {75,44126},
                       {90,44126},
                       {95,44126},
                       {99,44126},
                       {999,44126}]}]},
              {active,[{value,0},{ms_since_reset,11546}]},
              {count,[{<<"ping">>,[{count,1},{one,1}]}]}]}]},
 {node,[{abs,[{process_count,428},
              {run_queue,0},
              {error_logger_queue_len,0},
              {memory_total,50301760},
              {memory_procs,30854096},
              {memory_atoms,471201},
              {memory_bin,222648},
              {memory_ets,1574728}]},
        {inc,[{bytes_in,11737},
              {bytes_out,2470},
              {gc_count,6},
              {gc_words_reclaimed,29747},
              {reductions,2848780},
              {scheduler_usage,[{1,0.05329944038387727},
                                {2,0.8991375098414373},
                                {3,0.03932163131802264},
                                {4,0.05719991628720056}]}]}]},
 {core,[{ping,[{count,1},{one,1}]}]}]
```

You can see on this line that I made one request to ping and it returned 200:

```
{http,[{resp,[{by_code,[{200,[{count,1},{one,1}]},
```

You can also see request time stats per endpoint:

```
{req,[{time,[{<<"ping">>,
              [{n,3},
               {mean,44126},
               {min,44126},
               {max,44126},
               {median,44126},
               {50,0},
               {75,44126},
               {90,44126},
               {95,44126},
               {99,44126},
               {999,44126}]}]},
```

And request count by endpoint:

```
{count,[{<<"ping">>,[{count,1},{one,1}]}]}]}]}]},
```

3.3.2 Exposing Metrics as a REST resource

This one will be simple, first we add the route to cowboy then add the metrics endpoint to the list of endpoints we want to collect metrics (metricception) and finally we implement the cowboy handler to return the json.

Test it

Stop, build, start and make some requests:

```
http localhost:8080/ping
```

And then make a request for the metrics (result edited since it's quite big):

```
$ http localhost:8080/metrics

HTTP/1.1 200 OK
content-length: 8079
content-type: application/json
date: Fri, 30 Oct 2015 10:39:27 GMT
server: Cowboy

{
    "core": {
        "ping": { "count": 2, "one": 1 }
    },
    "http": {
        "req": {
            "active": { "ms_since_reset": 279958, "value": 1 },
            "count": {
                "metrics": { "count": 1, "one": 0 },
                "ping": { "count": 2, "one": 1 }
            },
            "time": {
                "metrics": {
                    "50": 0,
                    "75": 0,
                    "90": 0,
                    "95": 0,
                    "99": 0,
                    "999": 0,
                    "max": 0,
                    "mean": 0,
                    "median": 0,
                    "min": 0,
```

```
                    "n": 0
                },
                "ping": {
                    "50": 0,
                    "75": 349,
                    "90": 349,
                    "95": 349,
                    "99": 349,
                    "999": 349,
                    "max": 349,
                    "mean": 349,
                    "median": 349,
                    "min": 349,
                    "n": 3
                }
            }
        },
        "resp": {
            "by_code": {
                "200": { "count": 3, "one": 1 },
                "201": { "count": 0, "one": 0 },
                ...
                "400": { "count": 0, "one": 0 },
                "401": { "count": 0, "one": 0 },
                ...
                "404": { "count": 0, "one": 0 },
                ...
                "500": { "count": 0, "one": 0 },
                ...
            }
        }
    },
    "node": {
        "abs": {
            "error_logger_queue_len": 0,
            "memory_atoms": 471362,
            "memory_bin": 224392,
            "memory_ets": 1579592,
            "memory_procs": 31886248,
            "memory_total": 51342840,
            "process_count": 432,
            "run_queue": 0
        },
        "inc": {
            "bytes_in": 0,
            "bytes_out": 0,
            "gc_count": 2,
            "gc_words_reclaimed": 6624,
            "reductions": 695770,
            "scheduler_usage": {
                "1": 0.16108125753314584,
                "2": 0.5187896583972728,
                "3": 0.18046079477682214,
                "4": 0.15292436095407036
            }
        }
    },
    "tanodb": {
        ...
    }
}
```

CHAPTER

FOUR

USERS, GROUPS AND PERMISSIONS

Now that we exposed metrics through our REST API we may want to restrict its access to only some users, for that we will need at least users and permissions.

Luckily riak_core provides support for users, groups and permissions built in, the module that provides it is riak_core_security, in a later chapter we explore the module in detail, but for now let's use some libraries to get ourself started fast.

For this we will a library called rcs_cowboy which exposes riak_core_security API through cowboy. But since doing HTTP calls to handle our users is a little bit annoying we will also use a project called iorioui which is a ui for the API exposed by rcs_cowboy.

Let's get started!

As usual we start by adding the dependency for rcs_cowboy

Note: At the moment there's a hack where we add a compiler flag to export all functions from riak_core since riak_core_security doesn't export the get_context function and rcs_cowboy needs it.

Alternatives would be to fork riak_core or copy the riak_core_security module and export the function or replicate get_context in another module, for now to avoid complications we use this hack.

Since we need permissions to manage we need to uncomment the configuration in config/advanced.config to list the permissions that our app allows.

We also need to add the cowboy routes for rcs_cowboy, read the readme of rcs_cowboy for details.

Since we need to login to the admin ui, we need to ensure that at least there's a user available, for that we ensure that there are 2 users and 2 groups by default and we call the setup function during startup

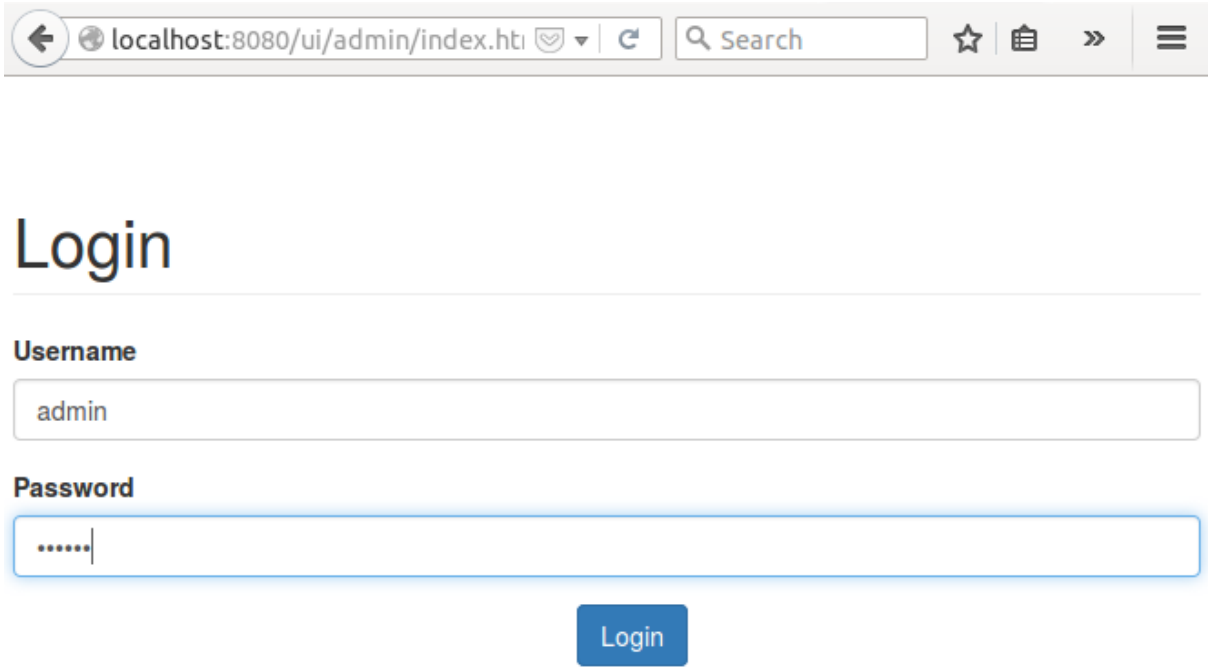
For now session management will be weak since it's not our focus at the moment, this means that when rcs_cowboy calls us to provide a response with a session token we return a fixed token and when rcs_cowboy calls us to check if a request is authorized (the ui will send us back the token we returned in the X-Session header), we say always that it's authorized.

Finally we copy the files from iorioui under apps/tanodb/priv/ui/admin and add a cowboy static route to serve the static files. Read more about cowboy static file serving in the cowboy's page for cowboy_static.

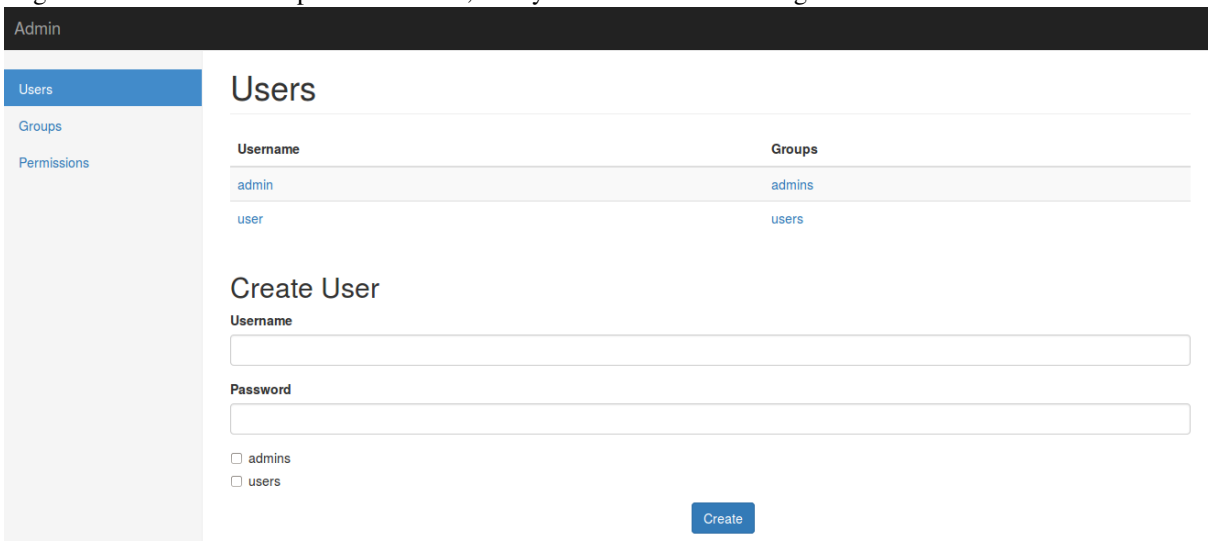
Now do the usual process of stop, build and run, but before running, in case you were playing with users and groups you can remove the folder that contains that data so you start from scratch and users and groups are created correctly, to do this run:

```
rm -r _build/default/rel/tanodb_data/cluster_meta
```

Once the server is running open http://http://localhost:8080/ui/admin/index.html with your browser, you should see something like this:



Login with user *admin* and password *secret*, then you should see something like this:



4.1 Riak Core Security Model

The riak_core_security module provides a security model that contains 4 main elements:

Users Usernames and passwords as usual, users can belong to groups, a user inherits the permissions of the groups it belongs.

Groups A group has a name and can belong to other groups, a group inherits the permissions of the groups it belongs.

Grants A grant is a rule that says that a user or group has permission to do something on a resource, a permission is a string composed of the app name and the permission name joined by a dot as a string, for example "tanodb.get".

The meaning of permissions is given by your application logic by checking for specific permissions when an operation is requested on a resource.

The list of valid permissions for your app is defined in the file config/advanced.config.

Resources Riak Core allows you to apply grants to a bucket or a bucket/key pair, a bucket is like a namespace for keys, you can have the same key in two different buckets and they mean different things.

For example you could assign one bucket to each user or something else, the use of buckets and keys is defined by your application.

CHAPTER

FIVE

HOW A COMMAND WORKS

Enough with the setup, let's see how ping works under the covers.

Its entry point and public API is the tanodb module, that means we have to look into tanodb.erl:

```
-module(tanodb).
-include_lib("riak_core/include/riak_core_vnode.hrl").

-export([ping/0]).

-ignore_xref([ping/0]).

%% Public API

%% @doc Pings a random vnode to make sure communication is functional
ping() ->
    tanodb_metrics:core_ping(),
    DocIdx = riak_core_util:chash_key({<<"ping">>, term_to_binary(os:timestamp())}),
    PrefList = riak_core_apl:get_primary_apl(DocIdx, 1, tanodb),
    [{IndexNode, _Type}] = PrefList,
    riak_core_vnode_master:sync_spawn_command(IndexNode, ping, tanodb_vnode_master).
```

We see we have our ping function there as the only public API and it does some funny stuff.

I won't go into much riak_core details that are described elsewhere since here we cover the practical aspects, there are many useful talks about riak_core internals and theory around, you can watch them:

- Rusty Klophaus - Masterless Distributed Computing with Riak Core
- Andy Gross - Riak Core - An Erlang Distributed Systems Toolkit

There are also some detailed articles about it:

- Ryan Zezeski's "working" blog
- Riak Core Wiki
- Where To Start With Riak Core

But let's look at what it does line by line:

```
tanodb_metrics:core_ping(),
```

First we register the operation in our metrics, we covered this in previous chapters.

```
DocIdx = riak_core_util:chash_key({<<"ping">>, term_to_binary(os:timestamp())}),
```

The line above hashes a key to decide to which vnode the call should go, a riak_core app has a fixed number of vnodes that are distributed across all the instances of your app's physical nodes, vnodes move from instance to instance when the number of instances change to balance the load and provide fault tolerance and scalability.

The call above will allow us to ask for vnodes that can handle that hashed key, let's run it in the app console to see what it does:

```
(tanodb@127.0.0.1)1> DocIdx = riak_core_util:chash_key({<<"ping">>, term_to_binary(os:timestamp()

<<126,9,218,77,97,108,38,92,0,155,160,26,161,3,200,87,134,213,167,168>>
```

We seem to get a binary back, in the next line we ask for a list of vnodes that can handle that hashed key:

```
PrefList = riak_core_apl:get_primary_apl(DocIdx, 1, tanodb),
```

let's run it to see what it does:

```
(tanodb@127.0.0.1)2> PrefList = riak_core_apl:get_primary_apl(DocIdx, 1, tanodb).

[{{730750818665451459101842416358141509827966271488, 'tanodb@127.0.0.1'},
     primary}]
```

We get a list with one tuple that has 3 items, a long number, something that looks like a host and an atom, let's try changing the number 1:

```
(tanodb@127.0.0.1)3> PrefList2 = riak_core_apl:get_primary_apl(DocIdx, 2, tanodb).

[{{730750818665451459101842416358141509827966271488,
   'tanodb@127.0.0.1'},
  primary},
 {{753586781748746817198774991869333432010090217472,
   'tanodb@127.0.0.1'},
  primary}]
```

Now we get two tuples, the first one is the same, so what this does is to return the number of vnodes that can handle the request from the hashed key by priority.

Btw, the first number is the vnode id, it's what we get on the ping response :)

Next line just unpacks the pref list to get the vnode id and ignore the other part:

```
[{IndexNode, _Type}] = PrefList,
```

And finally we ask riak_core to call the ping command on the IndexNode we got back:

```
riak_core_vnode_master:sync_spawn_command(IndexNode, ping, tanodb_vnode_master).
```

Let's try it on the console:

```
(tanodb@127.0.0.1)5> [{IndexNode, _Type}] = PrefList.

[{{730750818665451459101842416358141509827966271488,
   'tanodb@127.0.0.1'}, primary}]

(tanodb@127.0.0.1)6> riak_core_vnode_master:sync_spawn_command(IndexNode, ping, tanodb_vnode_maste

{pong,730750818665451459101842416358141509827966271488}
```

You can see we get IndexNode back in the pong response, now let's try passing the second IndexNode:

```
(tanodb@127.0.0.1)7> [{IndexNode1, _Type1}, {IndexNode2, _Type2}] = PrefList2.

[{{730750818665451459101842416358141509827966271488,
   'tanodb@127.0.0.1'}, primary},
 {{753586781748746817198774991869333432010090217472,
   'tanodb@127.0.0.1'}, primary}]


(tanodb@127.0.0.1)9> riak_core_vnode_master:sync_spawn_command(IndexNode2, ping, tanodb_vnode_mast

{pong,753586781748746817198774991869333432010090217472}
```

We get the IndexNode2 back, that means that the request was sent to the second vnode instead of the first one.

But where does the command go? the road is explained in this scientific chart:

```
tano.erl -> riak_core magic -> tano_vnode.erl
```

let's see the content of tanodb_vnode.erl (just the useful parts):

```
-module(tanodb_vnode).
-behaviour(riak_core_vnode).

-export([start_vnode/1,
         init/1,
         terminate/2,
         handle_command/3,
         is_empty/1,
         delete/1,
         handle_handoff_command/3,
         handoff_starting/2,
         handoff_cancelled/1,
         handoff_finished/2,
         handle_handoff_data/2,
         encode_handoff_item/2,
         handle_coverage/4,
         handle_exit/3]).

-record(state, {partition}).

%% API
start_vnode(I) ->
    riak_core_vnode_master:get_vnode_pid(I, ?MODULE).

init([Partition]) ->
    {ok, #state { partition=Partition }}.

%% Sample command: respond to a ping
handle_command(ping, _Sender, State) ->
    {reply, {pong, State#state.partition}, State};
handle_command(Message, _Sender, State) ->
    lager:warning("unhandled_command ~p", [Message]),
    {noreply, State}.
```

OK, let's go by parts, first we declare our module:

```
-module(tanodb_vnode).
```

Then we specify that we want to implement the riak_core_vnode behavior:

```
-behaviour(riak_core_vnode).
```

Behaviors in Erlang are like interfaces, a set of functions that a module must implement to satisfy the behaviour specification, you can read more in the Erlang documentation.

In this case riak_core defines a behavior with a set of functions we must implement to be a valid riak_core vnode, you can get an idea of the kind of functionality we need by looking at the exported functions:

```
-export([start_vnode/1,
         init/1,
         terminate/2,
         handle_command/3,
         is_empty/1,
         delete/1,
         handle_handoff_command/3,
         handoff_starting/2,
         handoff_cancelled/1,
         handoff_finished/2,
         handle_handoff_data/2,
         encode_handoff_item/2,
         handle_coverage/4,
         handle_exit/3]).
```

For the moment most of them have a "dummy" implementation where they just to the minimal amount of work to satisfy the behavior and not more, it's our job to change the default implementation to fit our needs.

We will have a record called state to keep info between callbacks, this is typical Erlang way of managing state so I won't cover it here:

```
-record(state, {partition}).
```

Then we implement the api to start the vnode, nothing fancy:

```
%% API
start_vnode(I) ->
    riak_core_vnode_master:get_vnode_pid(I, ?MODULE).
```

Note that on init we store the Partition value on state so we can use it later, this is what I referred above as vnode id, it's the big number you saw before:

```
init([Partition]) ->
    {ok, #state { partition=Partition }}.
```

And now for the interesting part, here we have our ping command implementation, we match for ping in the Message position (the first argument):

```
handle_command(ping, _Sender, State) ->
```

And return a reply response with the second item in the tuple being the actual response that the caller will get where we reply with the atom pong and the partition number of this vnode, the last item in the tuple is the new state we want to have for this vnode, since we didn't change anything we pass the current value:

```
{reply, {pong, State#state.partition}, State};
```

And then we implement a catch all that will just log the unknown command and give no reply back:

```
handle_command(Message, _Sender, State) ->
    lager:warning("unhandled_command ~p", [Message]),
    {noreply, State}.
```

So, this is the roundtrip of the ping call, our task to add more commands will be:

- Add a function on tanodb.erl that hides the internal work done to distribute the work
- Add a new match on handle_command to match the command we added on tanodb.erl and provide a reply

CHAPTER

SIX

ADDING OUR FIRST COMMANDS

Now that we have everything set up and we know how commands are implemented it's time to implement our own.

To start we are going to implement a simple in memory key value store, the first two commands we are going to implement are the basic ones we need to see if it works: put and get

To hold the values we are going to use the Erlang Term Storage (ETS).

6.1 Riak Core API

First we start by creating the two new metrics for our new commands.

Then we add the commands to tanodb.erl, get and put we extract the common code to hash the key to a vnode to a private function called send_to_one.

On the riak_core side, that is, in the tanodb_vnode module, on init we create our ETS table, the name is tanodb_<partition> where <partition> is the partition id.

Then we add two new function clauses to handle_command, one for put and one for get. The logic is quite simple.

6.1.1 Test it

Stop, build, start and in the console we run some commands.

First try getting the key "k1" from bucket "mybucket", which doesn't exist:

```
(tanodb@127.0.0.1)2> tanodb:get({<<"mybucket">>, <<"k1">>}).

{not_found,22835963083295358096932575511191922182123945984O,
           {<<"mybucket">>,<<"k1">>}}
```

We get not_found back with the partition that handled the command and the bucket and key that wasn't found.

Now let's put that key:

```
(tanodb@127.0.0.1)3> tanodb:put({<<"mybucket">>, <<"k1">>}, 42).

{ok,22835963083295358096932575511191922182123945984O}
```

We just get ok back, let's try to get it again:

```
(tanodb@127.0.0.1)4> tanodb:get({<<"mybucket">>, <<"k1">>}).

{found,22835963083295358096932575511191922182123945984O,
       {{<<"mybucket">>,<<"k1">>},{{<<"mybucket">>,<<"k1">>},42}}}
```

Now we get the value back.

Let's try the same with another key:

```
(tanodb@127.0.0.1)5> tanodb:get({<<"mybucket">>, <<"k2">>}).

{not_found,1210306043414653979137426502093171875652569137152,
           {<<"mybucket">>,<<"k2">>}}
```

Notice that the partition id changed, this is because the key hashed to a different vnode.

```
(tanodb@127.0.0.1)6> tanodb:put({<<"mybucket">>, <<"k2">>}, 42).

{ok,1210306043414653979137426502093171875652569137152}

(tanodb@127.0.0.1)7> tanodb:get({<<"mybucket">>, <<"k2">>}).

{found,1210306043414653979137426502093171875652569137152,
       {{<<"mybucket">>,<<"k2">>},{{<<"mybucket">>,<<"k2">>},42}}}
```

6.2 REST API

Let's expose our new functions as a REST API, first we add a new route to cowboy for our store, the API will be like this:

- POST /store/:bucket/:key <json-body>: stores <json-body> under {:bucket, :key}
  - returns 204 No Content on success
- GET /store/:bucket/:key:
  - returns 404 if :key doesn't exist on :bucket
  - returns 200 and the value stored under {:bucket, :key} if found

The implementation of the store api is quite simple if you know cowboy, it's in the tanodb_http_store.erl file.

6.2.1 Test it

Do the usual stop, build, run and then from another shell:

```
$ http localhost:8080/store/mybucket/bob
```

Returns

```
HTTP/1.1 404 Not Found
content-length: 0
content-type: application/json
date: Fri, 30 Oct 2015 17:16:16 GMT
server: Cowboy
```

Let's put something on that bucket/key:

```
$ http post localhost:8080/store/mybucket/bob name=bob color=yellow

HTTP/1.1 204 No Content
content-length: 0
content-type: application/json
date: Fri, 30 Oct 2015 17:17:25 GMT
server: Cowboy
```

And try to get it again:

```
$ http localhost:8080/store/mybucket/bob
```

```
HTTP/1.1 200 OK
content-length: 31
content-type: application/json
date: Fri, 30 Oct 2015 17:18:06 GMT
server: Cowboy

{
    "color": "yellow",
    "name": "bob"
}
```

6.3 Implementing Delete

Let's implement the delete command and REST API so our API is complete.

We start as usual adding the metrics for the delete command, then add the delete function on the tanodb module which is really similar to get.

After that we add the new function clause in handle_command in our vnode, notice that it returns the same values as get, this is to get back the last value in case it was found or inform us that there wasn't a value with that bucket and key.

Finally we handle the DELETE HTTP method in our cowboy handler.

6.3.1 Test it

Let's start by testing the core API, we get a key that is not there:

```
(tanodb@127.0.0.1)1> tanodb:get({<<"mybucket">>, <<"k1">>}).

{not_found,22835963083295358096932575511191922182123945984O,
           {<<"mybucket">>,<<"k1">>}}
```

Then set it to the value 42:

```
(tanodb@127.0.0.1)2> tanodb:put({<<"mybucket">>, <<"k1">>}, 42).

{ok,22835963083295358096932575511191922182123945984O}
```

Get it to make sure it's there:

```
(tanodb@127.0.0.1)3> tanodb:get({<<"mybucket">>, <<"k1">>}).

{found,22835963083295358096932575511191922182123945984O,
       {{<<"mybucket">>,<<"k1">>},{{<<"mybucket">>,<<"k1">>},42}}}
```

Proceed to delete it, notice that it returns the last seen value and the result has the same shape as a get call:

```
(tanodb@127.0.0.1)4> tanodb:delete({<<"mybucket">>, <<"k1">>}).

{found,22835963083295358096932575511191922182123945984O,
       {{<<"mybucket">>,<<"k1">>},{{<<"mybucket">>,<<"k1">>},42}}}
```

We get it again to make sure it was deleted:

```
(tanodb@127.0.0.1)5> tanodb:get({<<"mybucket">>, <<"k1">>}).

{not_found,22835963083295358096932575511191922182123945984O,
           {<<"mybucket">>,<<"k1">>}}
```

And try to delete it again to see how it handles trying to delete a key that is not there:

```
(tanodb@127.0.0.1)6> tanodb:delete({<<"mybucket">>, <<"k1">>}).

{not_found,22835963083295358096932575511191922182123945984O,
           {<<"mybucket">>,<<"k1">>}}
```

Now that we checked it works on the Erlang shell, let's try the REST API, we will do the same as before, first get and expect not found:

```
$ http localhost:8080/store/mybucket/bob

HTTP/1.1 404 Not Found
content-length: 0
content-type: application/json
date: Fri, 30 Oct 2015 17:32:17 GMT
server: Cowboy
```

Then POST a value:

```
$ http post localhost:8080/store/mybucket/bob name=bob color=yellow

HTTP/1.1 204 No Content
content-length: 0
content-type: application/json
date: Fri, 30 Oct 2015 17:32:21 GMT
server: Cowboy
```

GET it to make sure it's there:

```
$ http localhost:8080/store/mybucket/bob

HTTP/1.1 200 OK
content-length: 31
content-type: application/json
date: Fri, 30 Oct 2015 17:32:23 GMT
server: Cowboy

{
    "color": "yellow",
    "name": "bob"
}
```

DELETE it:

```
$ http delete localhost:8080/store/mybucket/bob

HTTP/1.1 204 No Content
content-length: 0
content-type: application/json
date: Fri, 30 Oct 2015 17:32:27 GMT
server: Cowboy
```

GET it back to make sure it's actually deleted:

```
$ http localhost:8080/store/mybucket/bob

HTTP/1.1 404 Not Found
content-length: 0
content-type: application/json
date: Fri, 30 Oct 2015 17:32:28 GMT
server: Cowboy
```

DELETE it again to see how it handles a missing delete:

```
$ http delete localhost:8080/store/mybucket/bob

HTTP/1.1 404 Not Found
content-length: 0
content-type: application/json
date: Fri, 30 Oct 2015 17:43:03 GMT
server: Cowboy
```

CHAPTER

SEVEN

LISTING KEYS FROM A BUCKET

Since we already implemented some commands you may be asking yourself, why do we need a full chapter for another command? well, think again...

Since bucket and key are hashed together to decide to which vnode a request will go it means that the keys for a given bucket may be distributed in multiple vnodes, and in case you are running in a cluster this means your keys are distributed in multiple physical nodes.

This means that to list all the keys from a bucket we have to ask all the vnodes for the keys on a given bucket and then put the responses together and return the set of all responses.

For this Riak Core provides something called coverage calls, which are a way to handle this process of running a command on all vnodes and gathering the responses.

In this chapter we are going to implement the *tanodb:keys(Bucket)* function using coverage calls.

7.1 Implementing the CORE API

We start as usual by adding the metric for the keys function.

Then implement tanodb:keys/1, but as you may notice it's not similar to the previous ones because of what we talked about in the introduction.

In this case we call *tanodb_coverage_fsm:start({keys, Bucket}, Timeout)*, which is a new module, it implements a behavior called *riak_core_coverage_fsm*, short for riak_core_coverage finite state machine, it implements some predefined callbacks that are called on different states of a finite state machine.

The start function calls tanodb_coverage_fsm_sup:start_fsm([ReqId, self(), Request, Timeout]) which starts a supervisor for this new process.

We also need to register the supervisor in the supervisor tree.

As a side note, tanodb_coverage_fsm uses a module called time_compat to avoid problems with deprecated uses of time in Erlang, for that we need to add the module as a dependency.

When we start the fsm with a command ({keys, Bucket}) and a timeout in milliseconds, it starts a supervisor that starts the finite state machine process, it first calls the init function which initializes the state of the process and returns some information to riak_core so it knows what kind of coverage call we want to do, then riak_core calls the handle_coverage function on each vnode and with each response it calls process_results in our process, when all the results are received or if an error happens (such as a timeout) it will call the finish callback there we send the results to the calling process which is waiting for it.

The handle_coverage implementation is really simple, it uses the ets:match/2 function to match against all the entries with the given bucket and returns the key from the matched results.

You can read more about ets match specs in the match spec chapter on the Erlang documentation.

7.1.1 Test It

Let's start by checking keys on an empty bucket.

```
(tanodb@127.0.0.1)1> tanodb:keys(<<"mybucket">>).

{ok,[{1347321821914426127719021955160323408745312813056,
      'tanodb@127.0.0.1',[]},

     ...

     {959110449498405040071168171470060731649205731328,
      'tanodb@127.0.0.1',...},
     {411047335499316445744786359201454599278231027712,...},
     {...}|...]}
```

The output is quite verbose, here is redacted for clarity, but we get back:

```
{ok, [{Partition, Node, ListOfKeys}*64]}
```

That means 64 3-item tuples (one for each vnode) with the partition id, the node where the partition is and the list of keys for that vnode, in this case all of them are empty and in the following cases most of them will be empty so we will filter them to clean the output.

Now let's put a value:

```
(tanodb@127.0.0.1)2> tanodb:put({<<"mybucket">>, <<"k1">>}, 42).

{ok,22835963083295358096932575511191922182123945984O}
```

And try again listing keys but this time filtering the empty results:

```
(tanodb@127.0.0.1)3> lists:filter(fun ({_, _, []}) -> false;
                                      (_) -> true
                                  end,
                                  element(2, tanodb:keys(<<"mybucket">>))).

[{22835963083295358096932575511191922182123945984O,
  'tanodb@127.0.0.1', [<<"k1">>]}]
```

We get one partition that returns the key that we just inserted, you can also check that the partition id is the same as the result from the put call before.

Now let's insert another value:

```
(tanodb@127.0.0.1)4> tanodb:put({<<"mybucket">>, <<"k2">>}, 43).

{ok,1210306043414653979137426502093171875652569137152}
```

And list again, now we get two partitions with keys:

```
(tanodb@127.0.0.1)5> lists:filter(fun ({_, _, []}) -> false;
                                      (_) -> true
                                  end,
                                  element(2, tanodb:keys(<<"mybucket">>))).

[{1210306043414653979137426502093171875652569137152,
  'tanodb@127.0.0.1', [<<"k2">>]},
 {22835963083295358096932575511191922182123945984O,
  'tanodb@127.0.0.1', [<<"k1">>]}]
```

Yet another value:

```
(tanodb@127.0.0.1)6> tanodb:put({<<"mybucket">>, <<"k3">>}, 44).

{ok,1073290264914881830555831049026020342559825461248}
```

And the list again:

```
(tanodb@127.0.0.1)7> lists:filter(fun ({_, _, []}) -> false;
                                      (_) -> true
                                  end,
                                  element(2, tanodb:keys(<<"mybucket">>))).

[{1210306043414653979137426502093171875652569137152,
  'tanodb@127.0.0.1', [<<"k2">>]},
 {1073290264914881830555831049026020342559825461248,
  'tanodb@127.0.0.1', [<<"k3">>]},
 {228359630832953580969325755111919221821239459840,
  'tanodb@127.0.0.1', [<<"k1">>]}]
```

7.2 Implementing the REST API

The REST API is quite straight forward, we add a new route to cowboy allowing to do *GET /store/:bucket* without specifying the key, we will interpret this as a request to "get the bucket" which for us means to return the keys.

Then when doing a GET and key is undefined we assume it's a request to list the bucket's keys so we request the keys and deduplicate them by using them as keys in a map with the values set to true and then collecting the keys of the map.

7.2.1 Test It

Like in the previous test, let's start listing an empty bucket:

```
$ http localhost:8080/store/mybucket

HTTP/1.1 200 OK
content-length: 2
content-type: application/json
date: Sat, 31 Oct 2015 14:12:52 GMT
server: Cowboy

[]
```

Let's put a value in that bucket:

```
$ http post localhost:8080/store/mybucket/bob name=bob color=yellow

HTTP/1.1 204 No Content
content-length: 0
content-type: application/json
date: Sat, 31 Oct 2015 14:12:58 GMT
server: Cowboy
```

And list it again:

```
$ http localhost:8080/store/mybucket

HTTP/1.1 200 OK
content-length: 7
content-type: application/json
date: Sat, 31 Oct 2015 14:13:00 GMT
server: Cowboy

[
    "bob"
]
```

Yet another one:

```
$ http post localhost:8080/store/mybucket/patrick name=patrick color=pink

HTTP/1.1 204 No Content
content-length: 0
content-type: application/json
date: Sat, 31 Oct 2015 14:13:18 GMT
server: Cowboy
```

List again:

```
$ http localhost:8080/store/mybucket

HTTP/1.1 200 OK
content-length: 17
content-type: application/json
date: Sat, 31 Oct 2015 14:13:20 GMT
server: Cowboy

[
    "bob",
    "patrick"
]
```

CHAPTER

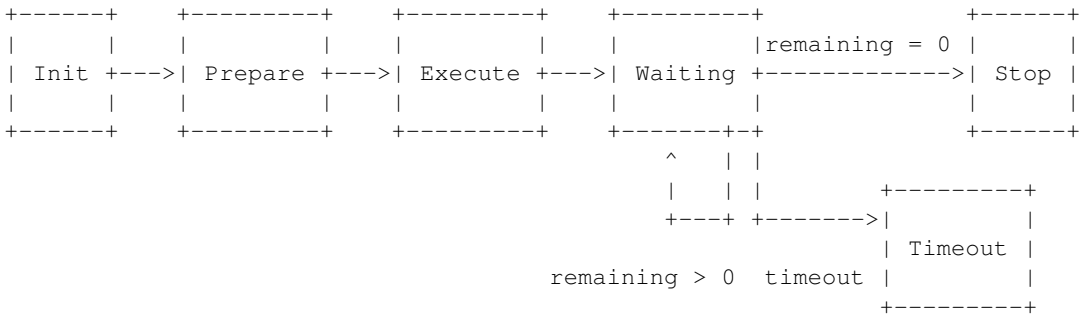
EIGHT

TOLERATING NODE FAILURES

You know computers cannot be trusted, so we may want to run our commands in more than one vnode and wait for a subset (or all of them) to finish before considering the operation to be successful, for this when a command is ran we will send the command to a number of vnodes, let's call it W and wait for a number of them to succeed, let's call it N.

To do this we will need to do something similar than what we did with coverage calls, we will need to setup a process that will send the command to a number of vnodes and accumulate the responses or timeout if it takes to long, then send the result back to the caller. We will also need a supervisor for it and to register this supervisor in our main supervisor tree.

Here is a diagram of how it works:



8.1 Quorum Based Writes and Deletes

To implement quorum based writes and deletes we will introduce two new modules, a gen_fsm implementation called tanodb_write_fsm and its supervisor, tanodb_write_fsm_sup. The supervisor is a simple supervisor behavior so I won't go into details here other than observing that we add it to the supervisor hierarchy as we did with the coverage supervisor, the gen_fsm is the one that is interesting.

On tanodb_write_fsm:write/6 and tanodb_write_fsm:delete/4 we start a supervisor that calls tanodb_write_fsm:start_link which in turn calls tanodb_write_fsm:init/1, this function initialize the state and moves the state machine to the prepare state.

The state from the fsm contains the following fields:

req_id Identifier for this request

from Process Id of the process that did the request

n Number of vnodes to send the request to

w Minimum number of responses to consider the request successful

key The key ({Bucket, Key}) that will be used for the operation

action An atom identifying the operation type, it can be *write* or *delete*

data If *action* is *write* then *data* is the value to write, if it's *delete* then it's not used

preflist A riak_core preflist

num_w Counter for current amount of responses

accum List of current response values, this field was introduced in the next commit

When we move to the *prepare* state we build the list of nodes we are going to send the request to using the value of *n*, we store the list of nodes on the preflist field and move to the execute state.

On the *execute* state we build the command we want to send depending on the value of the *action* field and we execute it, then we move to the waiting state.

On the waiting state when we receive a result we increment num_w and add the new response to accum, if *num_w* is equal to *w* we send the accumulated results to the requester with the req_id so it can distinguis it from others doing a selective receive.

On the *tanodb* module the changes are to call the delete and write functions in the write_fsm module and then do the selective receive waiting for the req_id we sent if the response doesn't come after *Timeout* milliseconds we return an error.

The changes on the *tanodb_vnode* module are that the *put* and *delete* commands now receive an extra argument, *ReqId* that is returned in the reply.

8.1.1 Test it

We start by listing the bucket's keys to make sure it's empty:

```
(tanodb@127.0.0.1)1> lists:filter(fun ({_, _, []}) -> false;
                                      (_) -> true
                                  end,
                                  element(2, tanodb:keys(<<"mybucket">>))).

[]
```

Then we put a value on that bucket:

```
(tanodb@127.0.0.1)2> tanodb:put({<<"mybucket">>, <<"k1">>}, 42).

{ok,[{ok,274031556999544297163190906134303066185487351808},
```

```
     {ok,251195593916248939066258330623111144003363405824},
     {ok,228359630832953580969325755111919221821239459840}]}
```

Now we list the keys again, but this time there's something different, 3 vnodes returned that they have the key *k1*, this means that our put wrote to 3 vnodes instead of 1 as before.

```
(tanodb@127.0.0.1)3> lists:filter(fun ({_, _, []}) -> false;
                                      (_) -> true
                                  end,
                                  element(2, tanodb:keys(<<"mybucket">>))).

[{251195593916248939066258330623111144003363405824,
  'tanodb@127.0.0.1', [<<"k1">>]},
 {274031556999544297163190906134303066185487351808,
  'tanodb@127.0.0.1', [<<"k1">>]},
 {228359630832953580969325755111919221821239459840,
  'tanodb@127.0.0.1', [<<"k1">>]}]
```

Let's delete that key to see if it deletes in the 3 vnodes:

```
(tanodb@127.0.0.1)4> tanodb:delete({<<"mybucket">>, <<"k1">>}).

{ok,[{found,274031556999544297163190906134303066185487351808,
            {{<<"mybucket">>,<<"k1">>},{{<<"mybucket">>,<<"k1">>},42}}},
     {found,228359630832953580969325755111919221821239459840,
            {{<<"mybucket">>,<<"k1">>},{{<<"mybucket">>,<<"k1">>},42}}},
     {found,251195593916248939066258330623111144003363405824,
            {{<<"mybucket">>,<<"k1">>}, {{<<"mybucket">>,<<"k1">>},42}}}]}
```

Listing the keys from the bucket shows that the key went away from all vnodes:

```
(tanodb@127.0.0.1)5> lists:filter(fun ({_, _, []}) -> false;
                                      (_) -> true
                                  end,
                                  element(2, tanodb:keys(<<"mybucket">>))).

[]
```

8.2 Handoffs

With quorum based writes we are half there, our values are written to more than one vnode but if a node dies and another takes his work or if we add a new node and the vnodes must be rebalanced we need to handle handoff.

The reasons to start a handoff are:

- A ring update event for a ring that all other nodes have already seen.
- A secondary vnode is idle for a period of time and the primary, original owner of the partition is up again.

When this happen riak_core will inform the vnode that handoff is starting, calling handoff_starting, if it returns false it's cancelled, if it returns true it calls is_empty, that must return false to inform that the vnode has something to handoff (it's not empty) or true to inform that the vnode is empty, in our case we ask for the first element of the ets table and if it's the special value '$end_of_table' we know it's empty, if it returns true the handoff is considered finished, if false then a call is done to handle_handoff_command passing as first parameter an opaque structure that contains two fields we are insterested in, foldfun and acc0, they can be unpacked with a macro like this:

```
handle_handoff_command(?FOLD_REQ{foldfun=Fun, acc0=Acc0}, _Sender, State) ->
```

The *FOLD_REQ* macro is defined in the riak_core_vnode.hrl header file which we include.

This function must iterate through all the keys it stores and for each of them call foldfun with the key as first argument, the value as second argument and the latest acc0 value as third.

The result of the function call is the new *Acc0* you must pass to the next call to foldfun, the last *Acc0* must be returned by the handle_handoff_command.

For each call to Fun(Key, Entry, AccIn0) riak_core will send it to the new vnode, to do that it must encode the data before sending, it does this by calling encode_handoff_item(Key, Value), where you must encode the data before sending it.

When the value is received by the new vnode it must decode it and do something with it, this is done by the function handle_handoff_data, where we decode the received data and do the appropriate thing with it.

When we sent all the key/values handoff_finished will be called and then delete so we cleanup the data on the old vnode.

You can decide to handle other commands sent to the vnode while the handoff is running, you can choose to do one of the followings:

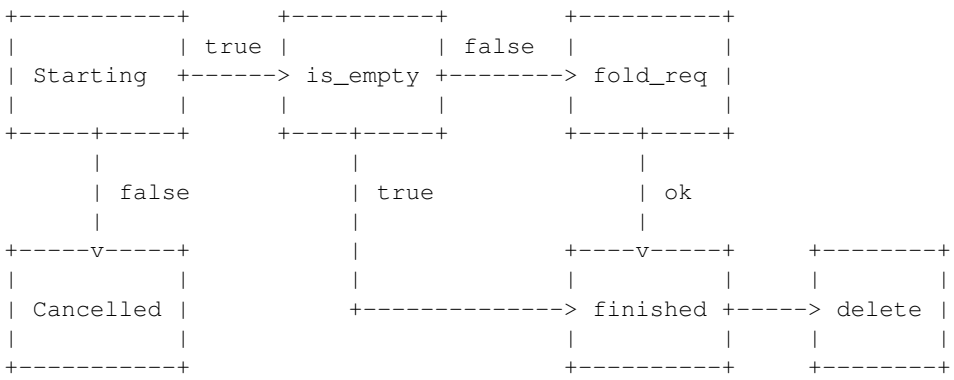
- Handle it in the current vnode
- Forward it to the vnode we are handing off
- Drop it

What to do depends on the design of you app, all of them have tradeoffs.

The signature of all the responses is:

```
-callback handle_handoff_command(Request::term(), Sender::sender(), ModState::term()) ->
{reply, Reply::term(), NewModState::term()} |
{noreply, NewModState::term()} |
{async, Work::function(), From::sender(), NewModState::term()} |
{forward, NewModState::term()} |
{drop, NewModState::term()} |
{stop, Reason::term(), NewModState::term()}.
```

A diagram of the flow is as follows:



8.2.1 Test it

To test it we will first start a devrel node, put some values and then join two other nodes and see on the console the handoff happening.

To make sure the nodes don't know about each other in case you played with clustering already we will start by removing the devrel builds:

```
rm -rf _build/dev*
```

And build the nodes again:

```
make devrel
```

Now we will start the first node and connect to its console:

```
make dev1-console
```

We generate a list of some numbers:

```
(tanodb1@127.0.0.1)1> Nums = lists:seq(1, 10).

[1,2,3,4,5,6,7,8,9,10]
```

And with it create some bucket names:

```
(tanodb1@127.0.0.1)2> Buckets = lists:map(fun (N) ->
(tanodb1@127.0.0.1)2>             list_to_binary("bucket-" ++ integer_to_list(N))
(tanodb1@127.0.0.1)2>           end, Nums).

[<<"bucket-1">>,<<"bucket-2">>,<<"bucket-3">>,
 <<"bucket-4">>,<<"bucket-5">>,<<"bucket-6">>,<<"bucket-7">>,
 <<"bucket-8">>,<<"bucket-9">>,<<"bucket-10">>]
```

And some key names:

```
(tanodb1@127.0.0.1)3> Keys = lists:map(fun (N) ->
(tanodb1@127.0.0.1)3>          list_to_binary("key-" ++ integer_to_list(N))
(tanodb1@127.0.0.1)3>        end, Nums).

[<<"key-1">>,<<"key-2">>,<<"key-3">>,<<"key-4">>,
 <<"key-5">>,<<"key-6">>,<<"key-7">>,<<"key-8">>,<<"key-9">>,
 <<"key-10">>]
```

We create a function to generate a value from a bucket and a key:

```
(tanodb1@127.0.0.1)4> GenValue = fun (Bucket, Key) ->
(tanodb1@127.0.0.1)4>                    [{bucket, Bucket}, {key, Key}]
```

```
(tanodb1@127.0.0.1)4>            end.

#Fun<erl_eval.12.54118792>
```

And then put some values to the buckets and keys we created:

```
(tanodb1@127.0.0.1)5> lists:foreach(fun (Bucket) ->
(tanodb1@127.0.0.1)5>   lists:foreach(fun (Key) ->
(tanodb1@127.0.0.1)5>     Val = GenValue(Bucket, Key),
(tanodb1@127.0.0.1)5>       tanodb:put({Bucket, Key}, Val)
(tanodb1@127.0.0.1)5>     end, Keys)
(tanodb1@127.0.0.1)5> end, Buckets).

ok
```

Now that we have some data let's start the other two nodes:

```
make dev2-console
```

In yet another shell:

```
make dev3-console
```

This part should remind you of the first chapter:

```
make devrel-join

Success: staged join request for 'tanodb2@127.0.0.1' to 'tanodb1@127.0.0.1'
Success: staged join request for 'tanodb3@127.0.0.1' to 'tanodb1@127.0.0.1'

make devrel-cluster-plan

================================ Staged Changes ============================
Action         Details(s)
--------------------------------------------------------------------------
join           'tanodb2@127.0.0.1'
join           'tanodb3@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%     34.4%    'tanodb1@127.0.0.1'
valid       0.0%     32.8%    'tanodb2@127.0.0.1'
valid       0.0%     32.8%    'tanodb3@127.0.0.1'
--------------------------------------------------------------------------
Valid:3 / Leaving:0 / Exiting:0 / Joining:0 / Down:0

WARNING: Not all replicas will be on distinct nodes

Transfers resulting from cluster changes: 42
  21 transfers from 'tanodb1@127.0.0.1' to 'tanodb3@127.0.0.1'
  21 transfers from 'tanodb1@127.0.0.1' to 'tanodb2@127.0.0.1'

make devrel-cluster-commit
```

```
Cluster changes committed
```

On the consoles from the nodes you should see some logs like the following, I will just paste some as example.

On the sending side:

```
00:17:24.240 [info] Starting ownership transfer of tanodb_vnode from
'tanodb1@127.0.0.1' 1118962191081472546749696200048404186924073353216 to
'tanodb2@127.0.0.1' 1118962191081472546749696200048404186924073353216

00:17:24.240 [info] fold req 1118962191081472546749696200048404186924073353216
00:17:24.240 [info] fold fun {<<"bucket-1">>,<<"key-1">>}:
    [{bucket,<<"bucket-1">>},{key,<<"key-1">>}]

...

00:17:24.241 [info] fold fun {<<"bucket-7">>,<<"key-8">>}:
    [{bucket,<<"bucket-7">>},{key,<<"key-8">>}]

00:17:24.281 [info] ownership transfer of tanodb_vnode from
'tanodb1@127.0.0.1' 1118962191081472546749696200048404186924073353216 to
'tanodb2@127.0.0.1' 1118962191081472546749696200048404186924073353216
    completed: sent 575.00 B bytes in 7 of 7 objects in 0.04 seconds
    (13.67 KB/second)

00:17:24.280 [info] handoff finished
    1141798154164767904846628775559596109106197299200:
    {1141798154164767904846628775559596109106197299200,
        'tanodb3@127.0.0.1'}

00:17:24.285 [info] delete
    1141798154164767904846628775559596109106197299200
```

On the receiving side:

```
00:13:59.641 [info] handoff starting
    1050454301831586472458898473514828420377701515264:
    {hinted,{1050454301831586472458898473514828420377701515264,
        'tanodb1@127.0.0.1'}}

00:13:59.641 [info] is_empty
    1826877046663628647754606040895353774569915678 72: true

00:14:34.259 [info] Receiving handoff data for partition
    tanodb_vnode:685078892498860742907977265335757665463718379 52 from
    {"127.0.0.1",47440}

00:14:34.296 [info] handoff data received
    {{<<"bucket-8">>,<<"key-1">>},
        [{bucket,<<"bucket-8">>},{key,<<"key-1">>}]}

...

00:14:34.297 [info] handoff data received
    {{<<"bucket-3">>,<<"key-7">>},
        [{bucket,<<"bucket-3">>},{key,<<"key-7">>}]}

00:14:34.298 [info] Handoff receiver for partition
    685078892498860742907977265335757665463718379 52 exited after
    processing 5 objects from {"127.0.0.1",47440}
```

CHAPTER

NINE

RIAK CORE METADATA

Note: The first 3 sections are taken from here https://gist.github.com/jrwest/d290c14e1c472e562548

9.1 1. Overview

Cluster Metadata is intended to be used by *riak_core* applications wishing to work with information stored cluster-wide. It is useful for storing application metadata or any information that needs to be read without blocking on communication over the network.

9.1.1 1.1 Data Model

Cluster Metadata is a key-value store. It treats values as opaque Erlang terms that are fully addressed by their "Full Prefix" and "Key". A Full Prefix is a *{atom() | binary(), atom() | binary()}*, while a Key is any Erlang term. The first element of the Full Prefix is referred to as the "Prefix" and the second as the "Sub-Prefix".

9.1.2 1.2 Storage

Values are stored on-disk and a full copy is also maintained in-memory. This allows reads to be performed only from memory, while writes are affected in both mediums.

9.1.3 1.3 Consistency

Updates in Cluster Metadata are eventually consistent. Writing a value only requires acknowledgment from a single node and as previously mentioned, reads return values from the local node, only.

9.1.4 1.4 Replication

Updates are replicated to every node in the cluster, including nodes that join the cluster after the update has already reached all nodes in the previous set of members.

9.2 2. API

The interface to Cluster Metadata is provided by the riak_core_metadata module. The module's documentation is the official source for information about the API, but some details are re-iterated here.

9.2.1 2.1 Reading and Writing

Reading the local value for a key can be done with the *get/2,3* functions. Like most *riak_core_metadata* functions, the higher arity version takes a set of possible options, while the lower arity function uses the defaults.

Updating a key is done using *put/3.4*. Performing a put only blocks until the write is affected locally. The broadcast of the update is done asynchronously.

2.1.1 Deleting Keys

Deletion of keys is logical and tombstones are not reaped. *delete/2,3* act the same as *put/3,4* with respect to blocking and broadcast.

9.2.2 2.2 Iterators

Along with reading individual keys, the API also allows Full Prefixes to be iterated over. Iterators can visit both keys and values. They are not ordered, nor are they read-isolated. However, they do guarantee that each key is seen *at most once* for the lifetime of an iterator.

See *iterator/2* and the *itr_** functions.

9.2.3 2.3 Conflict Resolution

Conflict resolution can be done on read or write.

On read, if the conflict is resolved, an option, *allow_put*, passed to *get/3* or *iterator/2* controls whether or not the resolved value will be written back to local storage and broadcast asynchronously.

On write, conflicts are resolved by passing a function instead of a new value to *put/3,4*. The function is passed the list of existing values and can use this and values captured within the closure to produce a new value to store.

9.2.4 2.4 Detecting Changes in Groups of Keys

The *prefix_hash/1* function can be polled to determined when groups of keys, by Prefix or Full Prefix, have changed.

9.3 3. Common Pitfalls & Other Notes

The following is by no means a complete list of things to keep in mind when developing on top of Cluster Metadata.

9.3.1 3.1 Storage Limitations

Cluster Metadata use *dets* for on-disk storage. There is a *dets* table per Full Prefix, which limits the amount of data stored under each Full Prefix to 2GB. This size includes the overhead of information stored alongside values, such as the logical clock and key.

Since a full-copy of the data is kept in-memory, its usage must also be considered.

9.3.2 3.2 Replication Limitations

Cluster Metadata uses disterl for message delivery, like most Erlang applications. Standard caveats and issues with large and/or too frequent messages still apply.

9.3.3 3.3 Last-Write Wins

The default conflict resolution strategy on read is last-write-wins. The usual caveats about the dangers of this method apply.

9.3.4 3.4 "Pathological Eventual Consistency"

The extremely frequent writing back of resolved values after read in an eventually consistent store where acknowledgment is only required from one node for both types of operations can lead to an interesting pathological case where siblings continue to be produce (although the set does not grow unbounded). A very rough exploration of this can be found here).

If a *riak_core* application is likely to have concurrent writes and wishes to read extremely frequently, e.g. in the Riak request path, it may be advisable to use *{allow_put, false}* with *get/3*.

9.4 4. Playing in the REPL

we start by building and running our app:

```
rebar3 release
rebar3 run
```

First let's setup some variables, FullPrefix is like an identifier for the place where we are going to store related values, there can be many, some of them are used by other components of riak_core as you will see in the next sections.

```
(tanodb@127.0.0.1)1> FullPrefix = {<<"tanodb">>, <<"config">>}.
{<<"tanodb">>,<<"config">>}
```

Let's start by trying to get a value that is not set, by default we get undefined.

```
(tanodb@127.0.0.1)2> riak_core_metadata:get(FullPrefix, max_users).
undefined
```

We can change that by calling the get function that supports options, one of them is **default**, so we set it to a value that makes sense for use in case max_users is not set.

```
(tanodb@127.0.0.1)3> riak_core_metadata:get(FullPrefix, max_users, [{default, 100}]).
100
```

Now let's put the value in the store.

```
(tanodb@127.0.0.1)4> riak_core_metadata:put(FullPrefix, max_users, 150).
ok
```

And try getting it.

```
(tanodb@127.0.0.1)5> riak_core_metadata:get(FullPrefix, max_users).
150
```

Let's put another value.

```
(tanodb@127.0.0.1)6> riak_core_metadata:put(FullPrefix, max_connections, 100).
ok
```

Get all the values in this prefix as a list, the "d" there is because [100] looks like a string to erlang, don't worry, your value is safe.

```
(tanodb@127.0.0.1)7> riak_core_metadata:to_list(FullPrefix).
[{max_connections,"d"},{max_users,[150]}]
```

Now let's delete a value.

```
(tanodb@127.0.0.1)8> riak_core_metadata:delete(FullPrefix, max_users).
ok
```

And put another one.

```
(tanodb@127.0.0.1)9> riak_core_metadata:put(FullPrefix, hostname, "tanodb1").
ok
```

Now let's list them again, you will see that deleted values are still there but **marked** with a "thombstone" value (the atom '$deleted'), this means we have to handle them in our functions if we want to avoid trouble.

```
(tanodb@127.0.0.1)11> riak_core_metadata:to_list(FullPrefix).
[{max_connections,"d"},
 {max_users,['$deleted']},
 {hostname,["tanodb1"]}]
```

Now let's do something more complex, let's iterate over all the values in the prefix, count the amount of deleted values and accumulate the "alive" ones.

Notice I use a function clause to match the thombstone first and then one to handle "alive" values.

```
(tanodb@127.0.0.1)11> riak_core_metadata:fold(fun
(tanodb@127.0.0.1)11>     ({Key, ['$deleted']}, {Deleted, Alive}) ->
(tanodb@127.0.0.1)11>         {Deleted + 1, Alive};
(tanodb@127.0.0.1)11>     ({Key, [Value]}, {Deleted, Alive}) ->
(tanodb@127.0.0.1)11>         {Deleted, [{Key, Value}|Alive]}
(tanodb@127.0.0.1)11> end, {0, []}, FullPrefix).

{1,[{max_connections,100},{hostname,"tanodb1"}]}
```

There are more functions I didn't show here since this ones are the main ones you will uses, you can look at the riak_core_metadata module for the other ones, the module has good documentation for each function.

CHAPTER

TEN

RIAK CORE SECURITY

riak_core_security is a module in riak_core that provides facilities to implement user/group management, authentication and authorization.

Here we will see an overview of it.

10.1 Implementation

riak_core_security is implemented on top of riak_core_metadata, it uses the following keys to store its information:

```
{<<"security">>, <<"users">>}
{<<"security">>, <<"groups">>}
{<<"security">>, <<"sources">>}
{<<"security">>, <<"usergrants">>}
{<<"security">>, <<"groupgrants">>}
{<<"security">>, <<"status">>} -> enabled
{<<"security">>, <<"config">>} -> ciphers
```

How they are stored should be an implementation detail but sometimes you may need to fold over values to get information if it's not supported by riak_core_security's API.

10.2 Vocabulary

10.2.1 Context

Opaque information you get back from authentication, you have to pass it back in to other operations.

At the moment it's a record with three fields:

- username
- grants
- epoch

But notice that this is an implementation detail and you should handle it as an opaque value.

Contexts are only valid until the GRANT epoch changes, and it will change whenever a GRANT or a REVOKE is performed. This rule may change in the future.

10.2.2 Permission

A string that represents some action in a given application, for example tanodb.get, tanodb.put.

A permission muy be listed as valid in the environment variable {riak_core, permissions}:

```
(tanodb@127.0.0.1)1> application:get_env(riak_core, permissions).
{ok,[{riak_core,[get_bucket,set_bucket,get_bucket_type, set_bucket_type]}]}
```

You can list your permissions in config/advanced.config uncommenting the line:

```
% {permissions, [{ tanodb, [put, get, list, grant, delete]}]}
```

And changing the permissions inside the list.

Note: tanodb is the name of your app

10.2.3 Role

Something you assign permissions to, it can be a user or a group, there are some reserved roles:

- all
- on
- to
- from
- any

10.2.4 Source

The source where the user is authenticating, it can be an IP or something else, you can allow a user to authenticate from a source but not another.

10.3 Extra Features

- Certificate Authentication
- Pluggable Authentication

10.4 API Overview

10.4.1 check_permission

```
% Check a Global permission, one that is not tied to a bucket
check_permission({Permission}, Context)

% Check a permission for a specific bucket
check_permission({Permission, Bucket}, Context)
```

10.4.2 check_permissions

```
% Check that all permissions are valid
check_permissions(List, Ctx)
```

10.4.3 get_username

```
% return username from context
get_username(Context)
```

10.4.4 authenticate

If successful it will return {ok, Context}

A username can be tied to specific sources from which he can login, if you don't need this feature specify a generic source for all your users.

```
authenticate(Username, Password, ConnInfo)
```

10.4.5 add_user

Valid options:

- password
- groups: groups must be a string with comma separated groups, like "g1,g2"

```
add_user(Username, Options)
```

10.4.6 add_group

Valid options:

- password

```
add_group(Groupname, Options)
```

10.4.7 alter_user

Options passed will override options already in user's details, this means if you pass a password it will be changed, if you pass groups the new groups will be set and the old removed.

```
alter_user(Username, Options)
```

10.4.8 alter_group

Options passed will override options already in groups's details, if you pass groups the new groups will be set and the old removed.

```
alter_group(Groupname, Options)
```

10.4.9 del_user

Deletes user and associated grants

```
del_user(Username)
```

10.4.10 del_group

Deletes group and associated grants

```
del_group(Groupname)
```

10.4.11 add_grant

Add Grants to RoleList on Bucket, RoleList can be the atom **all** to assign Grants to all roles in that Bucket.

Bucket can be a binary to assign to the whole bucket or {binary(), binary()}, to assign to a key in the bucket.

The call will merge previous grants with the new ones.

```
add_grant(RoleList, Bucket, Grants)
```

10.4.12 add_revoke

Revoke Grants to RoleList on Bucket, RoleList can be the atom **all** to revoke Grants to all roles in that Bucket.

```
add_revoke(RoleList, Bucket, Revokes)
```

10.4.13 add_source

Users is a list of users or the atom **all** to apply to all users. CIDR is a tuple with an IP address and a mask in bits. Source is an atom:

- trust: no password required
- password: password authentication
- certificate: certificate authentication
- Atom: Atom will be used as a custom authentication module, on auth Atom will be looked up on the env key {riak_core, auth_mods} if found the returned value will be used as a module to call AuthMod:auth(Username, Password, UserData, SourceOptions)

Options are options for the source that will be passed during auth

```
add_source(Users, CIDR, Source, Options)
```

Example calls:

```
riak_core_security:add_source(all, {{127, 0, 0, 1}, 32}, trust, [])
riak_core_security:add_source(all, {{127, 0, 0, 1}, 32}, password, [])
```

10.4.14 del_source

Delete source identified by CIDR for Users, Users can be the atom **all** to remove the source from all users. This won't apply to sources added for each users, only if the source was added explicitly for the **all** atom.

```
del_source(Users, CIDR)
```

10.4.15 is_enabled

Returns **true** if riak_core_security is enabled, **false** otherwise.

```
is_enabled()
```

10.4.16 enable

Enables riak_core_security

```
enable()
```

10.4.17 disable

Disabled riak_core_security

```
disable()
```

10.4.18 status

Returns an atom representing the status of riak_core_security:

- enabled
- enabled_but_no_capability
- disabled

```
status()
```

10.5 Playing in the REPL

First we will need to uncomment the permissions for our app in config/advanced.config

Then we build again and run it:

```
rebar3 release
rebar3 run
```

First let's setup some variables

```
(tanodb@127.0.0.1)1> User1 = <<"sandy">>.
<<"sandy">>

(tanodb@127.0.0.1)2> Pass1 = <<"secret">>.
<<"secret">>

(tanodb@127.0.0.1)3> ConnInfo = [{ip, {127, 0, 0, 1}}].
[{ip,{127,0,0,1}}]

(tanodb@127.0.0.1)4> Source1 = {{127, 0, 0, 1}, 32}.
{{127,0,0,1},32}

(tanodb@127.0.0.1)5> Bucket1 = <<"bucket_sandy">>.
<<"bucket_sandy">>

(tanodb@127.0.0.1)6> PermGet = "tanodb.get".
"tanodb.get"

(tanodb@127.0.0.1)7> PermPut = "tanodb.put".
"tanodb.put"

(tanodb@127.0.0.1)8> PermList = "tanodb.list".
"tanodb.list"

(tanodb@127.0.0.1)9> GroupWriter = <<"writers">>.
```

```
<<"writers">>

(tanodb@127.0.0.1)10> GroupReader = <<"readers">>.
<<"readers">>
```

We didn't add the user yet, so the following should fail

```
(tanodb@127.0.0.1)11> riak_core_security:authenticate(User1, Pass1, ConnInfo).
{error,unknown_user}
```

Let's add the user

```
(tanodb@127.0.0.1)12> riak_core_security:add_user(User1, [{"password", binary_to_list(Pass1)}]).
ok
```

Adding it twice should fail

```
(tanodb@127.0.0.1)13> riak_core_security:add_user(User1, [{"password", binary_to_list(Pass1)}]).
{error,role_exists}
```

We didn't add the source for the user so the following should fail

```
(tanodb@127.0.0.1)14> riak_core_security:authenticate(User1, Pass1, ConnInfo).
{error,no_matching_sources}
```

Add a local source that requires password for all users

```
(tanodb@127.0.0.1)15> riak_core_security:add_source(all, Source1, password, []).
ok
```

Now it should work

```
(tanodb@127.0.0.1)16> {ok, Ctx1} = riak_core_security:authenticate(User1, Pass1, ConnInfo).
{ok,{context,<<"sandy">>,[],{1444,659568,765253}}}
```

Checking permissions should fail, since we didn't granted any permissions yet

```
(tanodb@127.0.0.1)17> riak_core_security:check_permission({PermGet, Bucket1}, Ctx1).
{false,<<"Permission denied: User 'sandy' does not have 'tanodb.get' on bucket_sandy">>,
   {context,<<"sandy">>,[],{1444,659568,765253}}}
```

Let's grant PermGet to User1

```
(tanodb@127.0.0.1)18> riak_core_security:add_grant([User1], Bucket1, [PermGet]).
ok
```

And try again

```
(tanodb@127.0.0.1)19> riak_core_security:check_permission({PermGet, Bucket1}, Ctx1).
{true,{context,<<"sandy">>,
           [{<<"bucket_sandy">>,["tanodb.get"]}],
           {1444,659568,779759}}}
```

Create some groups, each group belongs to the previous one

```
(tanodb@127.0.0.1)20> riak_core_security:add_group(GroupReader, []).
ok

(tanodb@127.0.0.1)21> riak_core_security:add_group(GroupWriter, [{"groups", [GroupReader]}]).
ok
```

Let's grant permissions to each group

```
(tanodb@127.0.0.1)22> riak_core_security:add_grant([GroupReader], Bucket1, [PermGet]).
ok
```

```
(tanodb@127.0.0.1)23> riak_core_security:add_grant([GroupWriter], Bucket1, [PermPut]).
ok
```

Now let's join User1 to some groups and try permissions

```
(tanodb@127.0.0.1)24> riak_core_security:alter_user(User1, [{"groups", [GroupReader]}]).
ok
```

We can see User1 is a member of the group

```
(tanodb@127.0.0.1)25> riak_core_security:print_user(User1).
ok

+----------+--------------+------------------------------------------+-----------------------------
| username |   member of  |                password                  |           options
+----------+--------------+------------------------------------------+-----------------------------
|  sandy   |    readers   |9c8984b176e07eb7ba9ff1e3ada5a43ecb8a812e|              []
+----------+--------------+------------------------------------------+-----------------------------
```

She can do PermGet on Bucket1, but she could before since she has the permission explicitly set

```
(tanodb@127.0.0.1)26> riak_core_security:check_permission({PermGet, Bucket1}, Ctx1).
{true,{context,<<"sandy">>,
           [{<<"bucket_sandy">>,["tanodb.get"]}],
           {1444,659568,837358}}}
```

Let's revoke it

```
(tanodb@127.0.0.1)27> riak_core_security:add_revoke([User1], Bucket1, [PermGet]).
ok
```

Still can

```
(tanodb@127.0.0.1)28> riak_core_security:check_permission({PermGet, Bucket1}, Ctx1).
{true,{context,<<"sandy">>,
           [{<<"bucket_sandy">>,["tanodb.get"]}],
           {1444,659568,847161}}}
```

But can't put on that bucket

```
(tanodb@127.0.0.1)29> riak_core_security:check_permission({PermPut, Bucket1}, Ctx1).
{false,<<"Permission denied: User 'sandy' does not have 'tanodb.put' on bucket_sandy">>,
   {context,<<"sandy">>,
            [{<<"bucket_sandy">>,["tanodb.get"]}],
            {1444,659568,848204}}}
```

Now let's join User1 to some groups and try permissions

```
(tanodb@127.0.0.1)30> riak_core_security:alter_user(User1, [{"groups", [GroupWriter]}]).
ok
```

We can see User1 is a member of the group, but no more of GroupReader

```
(tanodb@127.0.0.1)31> riak_core_security:print_user(User1).
ok

+----------+--------------+------------------------------------------+-----------------------------
| username |   member of  |                password                  |           options
+----------+--------------+------------------------------------------+-----------------------------
|  sandy   |    writers   |9c8984b176e07eb7ba9ff1e3ada5a43ecb8a812e|              []
+----------+--------------+------------------------------------------+-----------------------------
```

User1 can now put on that bucket

```
(tanodb@127.0.0.1)32> riak_core_security:check_permission({PermPut, Bucket1}, Ctx1).
{true,{context,<<"sandy">>,
           [{<<"bucket_sandy">>,["tanodb.get","tanodb.put"]}],
           {1444,659568,859448}}}
```

Still can get since GroupWriter is member of the group GroupReader

```
(tanodb@127.0.0.1)33> riak_core_security:check_permission({PermGet, Bucket1}, Ctx1).
{true,{context,<<"sandy">>,
           [{<<"bucket_sandy">>,["tanodb.get","tanodb.put"]}],
           {1444,659568,860961}}}
```

Now let's add a new grant to GroupReader so they can list the bucket

```
(tanodb@127.0.0.1)34> riak_core_security:add_grant([GroupReader], Bucket1, [PermList]).
ok
```

Now User1 has the list permission since she is a member of GroupWriter which is a member of GroupReader who has permissions to list Bucket1

```
(tanodb@127.0.0.1)35> riak_core_security:check_permission({PermList, Bucket1}, Ctx1).
{true,{context,<<"sandy">>,
           [{<<"bucket_sandy">>,
             ["tanodb.get","tanodb.list","tanodb.put"]}],
           {1444,659568,872565}}}
```

Let's remove GroupReader membership from GroupWriter

```
(tanodb@127.0.0.1)36> riak_core_security:alter_group(GroupWriter, [{"groups", []}]).
ok
```

Now User1 can't list on Bucket1 anymore

```
(tanodb@127.0.0.1)37> riak_core_security:check_permission({PermList, Bucket1}, Ctx1).
{false,<<"Permission denied: User 'sandy' does not have 'tanodb.list' on bucket_sandy">>,
   {context,<<"sandy">>,
            [{<<"bucket_sandy">>,["tanodb.put"]}],
            {1444,659568,881585}}}
```

Let's try one more thing, add GroupWriter to GroupReader

```
(tanodb@127.0.0.1)38> riak_core_security:alter_group(GroupWriter, [{"groups", [GroupReader]}]).
ok
```

This works again

```
(tanodb@127.0.0.1)39> riak_core_security:check_permission({PermList, Bucket1}, Ctx1).
{true,{context,<<"sandy">>,
           [{<<"bucket_sandy">>,
             ["tanodb.get","tanodb.list","tanodb.put"]}],
           {1444,659568,890698}}}
```

Let's now remove GroupReader completely

```
(tanodb@127.0.0.1)40> riak_core_security:del_group(GroupReader).
ok
```

This should fail again

```
(tanodb@127.0.0.1)41> riak_core_security:check_permission({PermList, Bucket1}, Ctx1).
{false,<<"Permission denied: User 'sandy' does not have 'tanodb.list' on bucket_sandy">>,
   {context,<<"sandy">>,
            [{<<"bucket_sandy">>,["tanodb.put"]}],
            {1444,659568,914573}}}
```

Let's clean everything up

```
(tanodb@127.0.0.1)42> riak_core_security:del_group(GroupWriter).
ok

(tanodb@127.0.0.1)43> riak_core_security:del_user(User1).
ok

(tanodb@127.0.0.1)44> riak_core_security:del_source(all, Source1).
ok
```

If you want to retry from scratch removing all state you can do the following:

```
rm -rf _build/default/rel
rebar3 release
rebar3 run
```

10.6 API Gotchas

10.6.1 Groups Value is a CSV

If you want to create a user that is member a more than one group at the same time in the same add_user call you have to pass a string with comma separated names of the groups the user is going to be member of, like this:

```
riak_core_security:add_user(User1, [{"password", binary_to_list(Pass1)}, {"groups", "readers,write
```

10.6.2 Prefixing Users and Groups to avoid Potential Conflict

Since there's only one function to add grants and there's no restriction on usernames or groupnames it may happen that there's a group and a user with the same name, if this is the case then we get an error back saying that there are duplicated roles, this means riak_core doesn't know if you want to add the grant to the user or the group.

Let's try it, this assumes you have a clean state on riak_core_security and that you uncommented the permissions section in advanced.config for this app:

```
(tanodb@127.0.0.1)1> riak_core_security:add_user(<<"admin">>, [{"password", "secret"}]).
ok

(tanodb@127.0.0.1)2> riak_core_security:add_group(<<"admin">>, []).
ok

(tanodb@127.0.0.1)3> riak_core_security:add_grant([<<"admin">>], {<<"bucket">>, <<"key">>}, ["tan

{error,{duplicate_roles,[<<"admin">>]}}
```

As you can see we got the duplcate_roles error.

To solve this ambiguity we can prefix the role with the type of it, let's try it:

```
(tanodb@127.0.0.1)4> riak_core_security:add_grant([<<"group/admin">>], {<<"bucket">>, <<"key">>},
ok

(tanodb@127.0.0.1)5> riak_core_security:add_grant([<<"user/admin">>], {<<"bucket">>, <<"key">>},
ok
```

Now we assigned *tanodb.get* to the admin group and *tanodb.put* to the admin user.