Couchbase Ruby Client Manual

Sergey Avseyev

Couchbase Ruby Client Manual

Sergey Avseyev

Table of Contents

List of Tables

Chapter 1. Abstract

This is the manual for 1.2 of the Couchbase Ruby client library, which is compatible with Couchbase Server 2.0.

This manual provides a reference to the key features and best practices for using the Ruby Couchbase Client Library . The content constitutes a reference to the core API, not a complete guide to the entire API functionality.

Table 1.1. Product Compatibility for Couchbase Ruby SDK

Product	Compatible	All Features
Couchbase Server 1.8	✓	✓
Couchbase Server 2.0	✓	✓

External Community Resources

- Home page on couchbase.com [http://www.couchbase.com/develope/ruby/next]
- Download client library [https://rubygems.org/gems/couchbase]
- RDoc [http://rubydoc.info/gems/couchbase]
- SDK forum [http://www.couchbase.com/forums/sdks/sdks]
- Issue tracker [http://couchbase.com/issues/browse/RCBC]
- Sourse code repository [https://github.com/couchbase/couchbase-ruby-client]

Warning

The following document is still in production, and is not considered complete or exhaustive.

Chapter 2. Getting Started

Awesome that you want to learn more about Couchbase! This is the right place to start your journey. This chapter will teach you the basics of Couchbase and how to interact with it through the Ruby Client SDK.

If you haven't already, download the latest Couchbase Server 2.0 release and install it. While following the download instructions and setup wizard, make sure install the `beer-sample` default bucket. It contains sample data of beers and breweries, which we'll be using in our examples here. If you've already installed Couchbase Server 2.0 and didn't install the `beer-sample` bucket (or if you deleted it), just open the Web-UI and navigate to "Settings/Sample Buckets". Activate the `beer-sample` checkbox and click "Create". In the right hand corner you'll see a notification box that will disappear once the bucket is ready to be used.

Here's a quick outline of what you'll learn in this chapter:

1. Install the library with its dependencies.
2. Write a simple program to demonstrate connecting to Couchbase and saving some documents.

From here on, we'll assume that you have a Couchbase Server 2.0 release running and the `beer-sample` bucket configured. If you need any help on setting up everything, there is plenty of documentation available:

- Using the Couchbase Web Console [http://couchbase.com/docs/couchbase-manual-2.0/couchbase-introduction.html], for information on using the Couchbase Administrative Console,
- Couchbase CLI [http://couchbase.com/docs/couchbase-manual-2.0/couchbase-admin-web-console.html], for the command line interface,
- Couchbase REST API [http://couchbase.com/docs/couchbase-manual-2.0/couchbase-admin-restapi.html], for creating and managing Couchbase resources.

2.1. Installing the Couchbase Client Libraries

Before continuing you should ensure you have a working Ruby environment up and running. We recommend Ruby 1.9.2 or 1.8.7 http://ruby-lang.org.

You can verify that Ruby is installed by typing the following command:

```
shell> ruby -v
ruby 1.9.3p286 (2012-10-12 revision 37165) [x86_64-linux]
```

Another dependency needed for client is libcouchbase. Please consult [C Client Library [http://www.couchbase.com/develop/c/current]] page about ways to get it on your system. Here we will assume you are using the Ubuntu/Debian GNU/Linux family and have the `apt` tool.

Note that the libcouchbase dependency is not needed if you are on Microsoft Windows, as all dependencies are bundled in the source.

Once you have installed libcouchbase, you are then ready to install the most recent client using rubygems.

```
shell> gem install couchbase
Fetching: couchbase-1.2.0.gem (100%)
Building native extensions.  This could take a while...
Successfully installed couchbase-1.2.0
1 gem installed
```

Lets load and verify library version.

```
shell> ruby -rrubygems -rcouchbase -e 'puts Couchbase::VERSION'
1.2.0
```

2.2. Hello Couchbase

To follow the tradition of programming tutorials, we'll start with "Hello Couchbase". In the first example, we'll connect to the Cluster, retrieve the document, print it out and modify it. This first example contains the full sourcecode, but in later example we'll omit the preamble and assume we're already connected to the cluster.

```
require 'rubygems'
require 'couchbase'

client = Couchbase.connect(:bucket => "beer-sample",
                           :host => "localhost")

beer = client.get("aass_brewery-juleol")
puts "#{beer['name']}, ABV: #{beer['abv']}"

beer['comment'] = "Random beer from Norway"
client.replace("aass_brewery-juleol", beer)

client.disconnect
```

While this code should be very easy to grasp, there is a lot going on worth a little more discussion:

- Connecting: the `Couchbase.connect` basically creates an instance of `Couchbase::Bucket` class internally passing all arguments to its contructor. You can see complete list of options on the API documentation site [http://rdoc.info/gems/couchbase/Couchbase/Bucket#initialize-instance_method]. In our example the most interesting option is `:bucket`. Because our data bucket isn't "default" we must specify it during connection. The bucket is the container for all your documents. Inside a bucket, a key — the identifier for a document — must be unique. In production environments, it is recommended to use a password on a bucket (this can be configured during bucket creation), but when you are just starting out using the default bucket without a password is fine. Note that the `beer-sample` bucket also doesn't have a password, so just change the bucket name and you're set. Another option is `:host` which tells the client library the address of the cluster. While passing in only one host is fine, it is strongly recommended to add two or three (of course, if your cluster has more than one node) and use `:node_list` option instead. It is important to understand that this list does not have to contain all nodes in the cluster — you just need to provide a few so that during the initial bootstrap phase the Client is able to connect to the server. Any two or three nodes will be fine, but maintain this list. After this has happened, the Client automatically fetches the cluster configuration and keeps it up to date, even when the cluster topology changes. This means that you don't need to change your application config at all when you resize your cluster.

- Set and get: these two operations are the most fundamental ones. You can use set to create or completely replace a document inside your bucket and get to read it back afterwards. There are lots of arguments and variations, but if you just use them as shown in the previous example will get you pretty far. The sample is using the `Couchbase::Bucket#replace` operation. It behaves exactly like `#set` but will raise an error if the document isn't in the bucket. Note that by default all operations are using JSON to store your documents, so make sure it is possible to represent your values in this format. If not, you might use `:marshal` format. Find more info about the formats in the API documentation.

- Disconnecting: at the end of the program (or when you shutdown your server instance), you should use the `Couchbase::Bucket#disconnect` method. But you should know that the instance will be disconnected properly if it is destroyed by garbage collector.

That's it. We're ready to run our first Couchbase program.

```
shell> ruby hello.rb
Juleøl, ABV: 5.9
```

Chapter 3. Working with Documents

A document in Couchbase Server consists of a value and meta information, like a unique key, a CAS value, flags etc. These are all stored in a bucket. A document can be anything, but it is recommended to use the JSON format. JSON is very convenient for storing structured data with little overhead, and is also used inside the View engine. This means that if you want to get most out of Couchbase Server 2.0, use JSON.

The couchbase client will use any of accessible JSON libraries supported by multi_json gem [https://rubygems.org/gems/multi_json]. This mean if your values are serializable with `MultiJson.dump`, you can pass them to mutator methods and be sure you will get them later in the same form.

The following chapter introduces the basic operations that you can use as the fundamental building blocks of your application.

Here's a quick outline of what you'll learn in this chapter:

1. Write a program to demonstrate using Create, Read, Update, Delete (CRUD) operations on documents.

2. Explore some of the API methods that will take you further than what you've seen previously.

3.1. Creating and Updating Documents

Couchbase Server provides a set of commands to store documents. The commands are very similar to each other and differ only in their meaning on the server-side. These are:

`set`	Stores a document in Couchbase Server (identified by its unique key) and overrides the previous document (if there was one).
`add`	Adds a document in Couchbase Server (identified by its unique key) and fails if there is already a document with the same key stored.
`replace`	Replaces a document in Couchbase Server (identified by its unique key) and fails if there is no document with the given key already in place.

There are also additional commands mutation commands, which do make sense when you are working in `:plain` mode, because they are implmented on the server and not JSON-aware. But still they might be useful in your application:

`prepend`	Prepend given string to the value. The concatenation is done on the server side.
`append`	Append given string to the value. The concatenation is also done on the server side.
`increment`	Increment, atomically, the value. The value is a string representation of an unsigned integer. The new value is returned by the operation. By default it will increment by one. See API reference for other options.
`decrement`	Decrement, atomically, the value. The value is a string representation of an unsigned integer. The new value is returned by the operation. By default it will decrement by one. See API reference for other options.

The SDK provides several options for these operations, but to start out here are the simplest forms:

```
key = "aass_brewery-juleol"
doc = {"name" => "Juleøl", "abv" => 5.9}

client.add(key, doc);
client.set(key, doc);
client.replace(key, doc);
```

3.2. Reading Documents

With Couchbase Server 2.0, you have two ways of fetching your documents: either by the unique key through the get method, or through Views. Since Views are more complex, let's just look at a simple get first:

```
doc = client.get("aass_brewery-juleol")

keys = ["foo", "bar"]
docs = client.get(keys, :quiet => true)
```

In this case you will receve the Hash document you stored earlier. If there no such key in the bucket, the exception `Couchbase::Error:NotFound` will be raised. But you can suppress all `NotFound` errors by using option `:quiet => true` and the method will return `nil` instead. The `Couchbase::Bucket#get` method can also accept list of keys returning list of documents.

With Couchbase Server 2.0, the very powerful ability to query your documents across this distributed system through secondary indexes (Views) has been added to your toolbelt. This guide gets you started on how to use them through the Ruby SDK, if you want to learn more please refer to the chapter in the Couchbase Server 2.0 documentation [http://www.couchbase.com/docs/couchbase-manual-2.0/couchbase-views.html].

Once you created your View in the UI, you can query it from the SDK in two steps. First, you grab the design document definition from the cluster, second query view with options you need and use results. In its simplest form, it looks like this:

```
# 1: Get the design document definition
ddoc = client.design_docs["beer"]
ddoc.views      #=> ["brewery_beers", "by_location"]
```

```
# 2: Query the view and use results
ddoc.brewery_beers.each do |row|
  puts row.key
  puts row.value
  puts row.id
  puts row.doc
end
```

Note that the view request won't be executed until you will try to access the results. This means that you can pass view object (`ddoc.brewery_beers` here) without executing it.

Views can be queried with a large amount of options to change what the results of the query will contain. All supported options are available as items in options Hash accepted either by the view method or by `#each` iterator on the view. Here are some of them:

include_docs (Boolean)	Used to define if the complete documents should be fetched with the result (`false` by default). Note this will actually fetch the document itself from the cache, so if it has been changed or deleted you may not receive a document that matches the view, or any at all.
reduce (Boolean)	Used to enable/disable the reduce function (if there is one defined on the server). `true` by default.
limit (Fixnum)	Limit the number of results that should be returned.
descending (Boolean)	Revert the sorting order of the result set. (`false` by default)
stale (Boolean, Symbol)	Can be used to define the tradeoff between performance and freshness of the data. (`:update_after` by default)

Now that we have our View information in place, we can issue the query, which actually triggers the scatter-gather data loading process on the Cluster. We can use it to iterate over the results and print out some details (here is a more complete example which also includes the full documents and only fetches the first five results). The resulting information is encapsulated inside the `ViewRow` object.

```
view = client.design_docs["beer"].brewery_beers

# Include all docs and limit to 5
view.each(:include_docs => true, :limit => 5) do |row|
  puts row.id
  # The full document (as a Hash) is available through row.doc
end
```

In the logs, you can see the corresponding document keys automatically sorted (ascending):

```
21st_amendment_brewery_cafe
21st_amendment_brewery_cafe-21a_ipa
21st_amendment_brewery_cafe-563_stout
21st_amendment_brewery_cafe-amendment_pale_ale
21st_amendment_brewery_cafe-bitter_american
```

3.3. Deleting Documents

If you want to get rid of a document, you can use the delete operation:

```
client.delete("aass_brewery-juleol");
```

Chapter 4. Advanced Topics

This chapter introduces some techniques topics that you can use to further extend your Couchbase vocabulary.

4.1. CAS and Locking

If you need to coordinate shared access on documents, Couchbase helps you with two approaches. Depending on the application you may need to use both of them, but in general it is better (if feasible) to lean towards CAS because it provides the better performance characteristics.

Optimistic Locking. Each document has a unique identifier associated with it (the CAS value), which changes when the document itself is mutated. You can fetch the CAS value for a given key and pass it to any mutator operation to protect it. The update will only succeed, when the CAS value is still the same. This is why it's called optimistic locking. Someone else can still read and try to update the document, but it will fail once the CAS value has changed. Here is a example on how to do it with the Ruby SDK:

```
key = "eagle_brewing-golden"
# Reads the document with the CAS value.
beer, flags, cas = client.get(key, :extended => true)

# Updates the document and tries to store it back.
beer["name"] = "Some other Name"
client.set(key, beer, :cas => cas, :flags => flags)
```

Note that this also means that all your application need to follow the same code path (cooperative locking). If you use `#set` somewhere else in the code on the same document, it will work even if the CAS itself is out of date (that's because the normal `#set` method doesn't care about those values at all). Of course, the CAS itself changes then and the mutation operation would fail afterwards.

There is also shortcut operation for doing optimistic locking `Bucket#cas`. Internally it does the same thing but abstract you from storing and passing meta information. Here is the previous example rewritten to use this operation:

```
key = "eagle_brewing-golden"
client.cas(key) do |beer|
  beer["name"] = "Some other Name"
  # return new value from block
  beer
end
```

Note that you should return new value from the block. If you will skip it, it will use `"Some other Name"` as new value.

Pessimistic Locking. If you want to lock a document completely (or an object graph), you can use the `Bucket#get` operation with `:lock` option. The option accepts either boolean (where truth does make sense really) or Fixnum meaning the time period where the lock is valid. The server will release lock after the that period (or maximum value, which configured on the server). Other threads can still run `get` queries queries against the document, but mutation operations without a CAS will fail.

You can determine actual default and maximum values calling `Bucket#stats` without arguments and inspecting keys `"ep_getl_default_timeout"` and `"ep_getl_max_timeout"` correspondingly.

```
key = "eagle_brewing-golden";

# Get with Lock
beer, flags, cas = client.get(key, :lock => true, :extended => true);

# Update the document
beer["name"] = "Some other Name"

# Try to set without the lock
client.set(key, beer, :flags => flags)
#=> will raise Couchbase::Error::KeyExists

# Try to set with the CAS aquired, will be OK
```

```
client.set(key, beer, :flags => flags, :cas => cas)
```

Once you update the document, the lock will be released. There is also the `Bucket#unlock` method available through which you can unlock the document.

4.2. Persistence and Replication

By default, the mutation operations return when Couchbase Server has accepted the command and stored it in memory (disk persistence and replication is handled asynchronously by the cluster). That's one of the reason why it's so fast. For most use-cases, that's the behavior that you need. Sometimes though, you want to trade in performance for data-safety and wait until the document has been saved to disk and/or replicated to other hosts.

The Ruby SDK provides `:observe` option for all mutation operations. You can claim various persistence conditions using this option. Basically its argument is a Hash with three possible keys, describing the condition when the mutator will yield the result:

1. `:replicated` (Fixnum) describe how many nodes should receive replicated copy of the document.

2. `:persisted` (Fixnum) describe how many nodes should persist the document to the disk. The nodes include master node, where the key resides and all replica nodes.

3. `:timeout` (Fixnum) the timeout period in microseconds. After passing, the operation condition will be considered timed out and appropriate exception will be thrown. Default value could be addressed using `Bucket#default_observe_timeout`.

Here is an example on how to make sure that the document has been persisted on its master node, but also replicated to at least one of its replicas.

```
key = "important"
value = "document"
client.set(key, value, :observe => {:persisted => 1, :replicated => 1})
```

You can also separate persistence requirement from actual operations, and in this case, you can wait for several keys:

```
keys = []
(1..5).each do |nn|
  key = "important-#{nn}"
  keys << key
  client.set(key, "document-#{nn}")
end
client.observe_and_wait(keys, :persisted => 1, :replicated => 1)
```

Chapter 5. Couchbase Rails Tutorial

The goal of this chapter is to show how to write more advanced application using Couchbase and Rails framework.

We assume here that you are passed "Getting Started" section already, if not, we recommend to take a look, because it describes how to install and verify Ruby SDK.

5.1. TL;DR

For the purposes of this tutorial, we have specially prepared an example application for you to follow along with. The application uses a bucket with one of the sample datasets which come with Couchbase Server itself: `beer-sample`. If you haven't already, download the latest Couchbase Server 2.0 release and install it. While following the download instructions and setup wizard, make sure you install the `beer-sample` default bucket. It contains a sample data set of beers and breweries, which we'll use in our examples here. If you've already installed Couchbase Server 2.0 and didn't install the `beer-sample` bucket (or if you deleted it), just open the Web-UI and navigate to "Settings/Sample Buckets". Select the `beer-sample` checkbox and click "Create". In the right hand corner you'll see a notification box that will disappear once the bucket is ready to be used.

After that you can clone the complete repository from couchbaselabs on github:

```
shell> git clone git://github.com/couchbaselabs/couchbase-beer.rb.git
Cloning into 'couchbase-beer.rb'...
remote: Counting objects: 409, done.
remote: Compressing objects: 100% (254/254), done.
remote: Total 409 (delta 183), reused 340 (delta 114)
Receiving objects: 100% (409/409), 235.17 KiB | 130 KiB/s, done.
```

```
Resolving deltas: 100% (183/183), done.
```

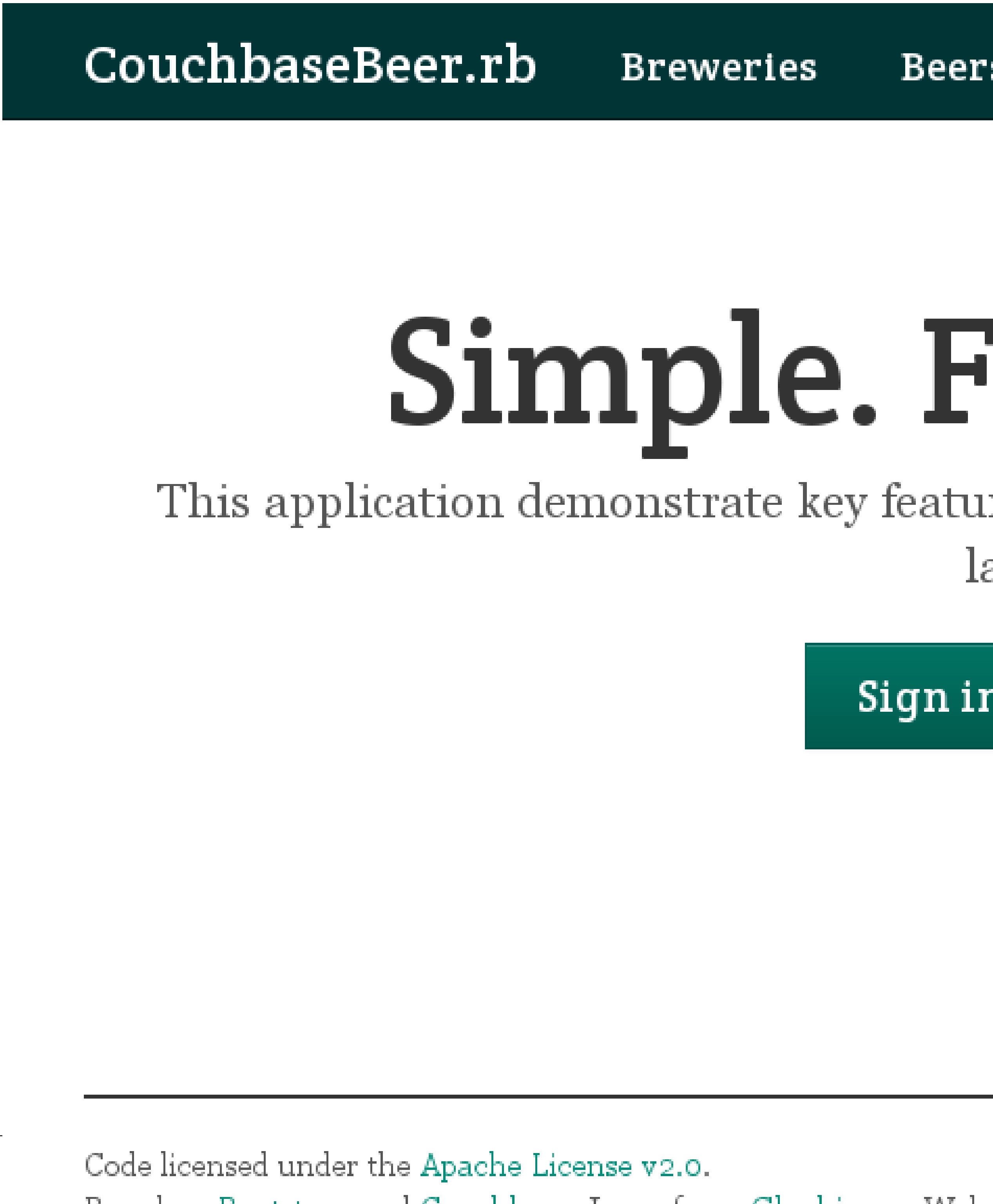
Navigate to the directory and install all application dependencies:

```
shell> cd couchbase-beer.rb/
shell> bundle install
...snip...
Your bundle is complete! Use `bundle show [gemname]` to see where a bundled gem
```

That’s it. Assuming that the server with `beer-sample` bucket is up and running on localhost, you can just start ruby web server:

```
shell> rails server
=> Booting Thin
=> Rails 3.2.8 application starting in development on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
>> Thin web server (v1.5.0 codename Knife)
>> Maximum connections set to 1024
>> Listening on 0.0.0.0:3000, CTRL+C to stop
```

Then navigate to http://localhost:3000/. You should see something like that:



Simple. F

This application demonstrate key featu

la

Code licensed under the Apache License v2.0.

5.2. Create Application Skeleton

If you would like to learn how to create this application from scratch just continue reading. As with any rails application we will use generators a lot. Since we don't need ActiveRecord we'll let `rails new` know about it.

```
shell> rails new couchbase-beer.rb -O --old-style-hash
```

Now navigate to the project root and open up `Gemfile` in your favorite editor. First, we need to add the Couchbase client libraries there:

```
gem 'couchbase'
gem 'couchbase-model'
```

Skipping the version will get the latest stable versions of those gems. We will use the couchbase-model [https://rubygems.org/gems/couchbase-model] gem to define our models in a declarative way much like all rails developers describe their models with ActiveRecord. Apart from that we will use `yajl-ruby`, a high-performance JSON parser/generator, `rdiscount` to render descriptions as Markdown, and `omniauth-twitter` for authentication.

```
gem 'yajl-ruby'
gem 'rdiscount'
gem 'omniauth-twitter'
```

The complete `Gemfile` will looks like this:

Gemfile.

```
source 'https://rubygems.org'

gem 'rails', '3.2.8'

gem "eventmachine", "~> 1.0.0"
gem 'thin', "~> 1.5.0"
gem 'jquery-rails'
gem 'yajl-ruby'
gem 'couchbase'
gem 'couchbase-model'
gem 'rdiscount'
gem 'omniauth'
gem 'omniauth-twitter'

gem 'capistrano'

group :development, :test do
  gem 'debugger'
end

group :assets do
  gem 'sass-rails', '~> 3.2.3'
  gem 'uglifier', '>= 1.0.3'
end
```

Next step will be to configure Couchbase connection, this step should be familiar to the rails developer, because `couchbase-model` brings YAML-styled configuration, so if you know how `config/database.yml` works, you can make some assumtions about how `config/couchbase.yml` works. To generate a config, use the `couchbase:config` generator:

```
shell> rails generate couchbase:config
      create  config/couchbase.yml
```

Since our bucket name differs from the project name, you should update the `bucket` property in the config. Also if your Couchbase Server is not running on the local machine, you should also change the hostname in the config. After you've made your modifications, your config should look like this:

config/couchbase.yml.

```
common: &common
  hostname: localhost
  port: 8091
  username:
  password:
  pool: default

development:
  <<: *common
  bucket: beer-sample

production:
  <<: *common
  bucket: beer-sample
```

That's it for configuration, let's move forward and create some models.

5.3. Define Models

To create a model, all you need is just define class and inherit from `Couchbase::Model`. Lets dissect the `Brewery` model from the application.

app/models/brewery.rb.

```
class Brewery < Couchbase::Model
  attribute :name, :description
  attribute :country, :state, :city, :address
  attribute :phone
  attribute :geo
  attribute :updated

  view :all, :limit => 31
  view :all_with_beers
  view :by_country, :include_docs => false, :group => true
  view :points, :spatial => true

  def full_address
    [country, state, city, address].reject(&:blank?).join(', ')
  end

  def location
    [country, state].reject(&:blank?).join(', ')
  end
end
```

The first part of the model contains attribute definitions. The `attribute` macro defines a pair of accessors and helps to maintain map of attributes-values. You can also specify default a value for each attribute:

```
class Post < Couchbase::Model
  attribute :title, :body
  attribute :timestamp, :default => lambda{ Time.now }
end

Post.new(:title => "Hello, World!")
#=> #<Post timestamp: 2012-12-07 16:03:56 +0300, title: "Hello, World!">
```

The next block is about defining views. One way to play with views is Couchbase Server Admin Console, but it may not be the best to keep parts of the code in different places. After all Views are implemented in Javascript and carry part of business logic. `couchbase-model` has solution for it. Each time the server starts (in development mode for each request), the application will check the filesystem changes of the map/reduce javascript files and update design document if needed. There is also the rails generator for views. It will put view stubs for you in proper places. Here, for example, is the model layout for this particular application:

```
app/models/
├── beer
│   ├── all
│   │   └── map.js
│   └── by_category
│       ├── map.js
│       └── reduce.js
├── beer.rb
├── brewery
│   ├── all
│   │   └── map.js
│   ├── all_with_beers
│   │   └── map.js
│   ├── by_country
│   │   ├── map.js
│   │   └── reduce.js
│   └── points
│       └── spatial.js
├── brewery.rb
├── favorites.rb
└── user.rb
```

For each model which has views, you should create a directory with the same name and put in the appropriate javascript files. Each should implement the corresponding parts of the view. For example `_design/brewery/_view/by_country` does require both map and reduce parts. To generate a new view you should pass model name and view name you need to get:

```
shell> rails generate couchbase:view beer test
      create  app/models/beer/test/map.js
      create  app/models/beer/test/reduce.js
```

Those automatically generated files are full of comments, so they are worth reading as quick start about how to write View indexes. The offical documentation for how to do so is in the Couchbase Server manual.

5.4. Implement Controllers

Now lets dissect the controllers from the project. This is where data is selected to display to user. For example `BreweriesController`:

```
class BreweriesController < ApplicationController
  def index
    filter = params.extract!(:start_key, :end_key).reject{|_, v| v.blank?}
    @breweries = Brewery.all(filter).to_a
    if @breweries.size > 30
      @last_key = @breweries.pop.try(:key)
    end
    respond_to do |format|
      format.html
      format.json do
        render :json => {:last_key => @last_key, :items => @breweries}
      end
    end
  end

  def show
    @brewery, *@beers = Brewery.all_with_beers(:start_key => [params[:id]],
                                               :end_key => ["#{params[:id]}\uef
  end
```

```
end
```

It has two actions:

1. "index" which is supposed to pull the list of breweries. It might look complex, but it isn't. In first line we are preparing query parameters for views. In particular, we are interested in the `start_key` and `end_key` parameters, but we need them only if they aren't blank (i.e. not nil and not empty string). The second step is to execute the view and get results immediately. Your application might want to fetch the entries from view on demand in a lazy manner, then you no need to call the `#to_a` method and iterate over them or call methods like `#next`. In this particular example we are fetching 31 records and are trying to pop last one to use it later for "infinite" scrolling. The end of the method is responsible for rendering the data in two formats, by default it will use HTML, but if the application is responding to an AJAX request with the `Accept: application/json` header, it will instead render the JSON representation of our models.

2. "show" uses another view from the `Brewery` model, which collates breweries with beer for easier access. Here is a map function which does that job:

   ```
   function(doc, meta) {
     switch(doc.type) {
     case "brewery":
       emit([meta.id]);
       break;
     case "beer":
       if (doc.brewery_id && doc.name) {
         emit([doc.brewery_id, doc.name]);
       }
       break;
     }
   }
   ```

   As you can see we are using a compound key with brewery ID in the first position and the document name in the second position. Because we are selecting only beers without null names, they will be sorted after the breweries. By doing so, when we filter result by the first key only, the `@brewery` variable will receive first element of the sequence, and `@beers` will get the rest of the collection because of the splat (`*`) operator.

5.5. Bonus: Spatial Queries, Sessions and Cache

One of the experimental features of the Couchbase Server is spatial views. These kind of views allow you to build indexes on geo attributes of your data. Sample application has part, demostrating power of spatial views. Click on the "Map" link in the menu, and if you allowed your browser to fetch your current location it will position the center of the map to your, or to Mountain View otherwise. After that it will execute spatial query using map bounds and the Couchbase Server will give you all the breweries which are nearby. Lets take a look at the implemetation. The core of this feature in `brewery/points/spatial.js`:

```
function(doc, meta) {
  if (doc.geo && doc.geo.lng && doc.geo.lat && doc.name) {
    emit({type: "Point", coordinates: [doc.geo.lng, doc.geo.lat]},
          {name: doc.name, geo: doc.geo});
  }
}
```

The function will emit Point object and the name with coordinates as the payload. The action in the controller is quite trivial, it transmit the result to the frontend in JSON representation, where google maps is rendering markers for each object.

Except nice extensions, provided by `couchbase-model` library, `couchase` gem itself has few more nice features which could be useful in the web application. For example, you can easily substitute your session or cache store in rails (or even in rack) with Couchbase Server.

To use Couchbase as cache store in rails, just put following line in your `config/application.rb` file:

```
config.cache_store = :couchbase_store
```

You can also pass additional connection options there

```
cache_options = {
  :bucket => 'protected',
  :username => 'protected',
  :password => 'secret',
  :expires_in => 30.seconds
}
config.cache_store = :couchbase_store, cache_options
```

To use Couchbase as the session store you should update your `config/initializers/session_store.rb` file

```
require 'action_dispatch/middleware/session/couchbase_store'
AppName::Application.config.session_store :couchbase_store
```

Or remove this file and add following line to your `config/application.rb`:

```
require 'action_dispatch/middleware/session/couchbase_store'
config.session_store :couchbase_store
```

You can also pass additional options:

```
require 'action_dispatch/middleware/session/couchbase_store'
session_options = {
  :expire_after => 5.minutes,
  :couchbase => {:bucket => "sessions", :default_format => :json}
}
config.session_store :couchbase_store, session_options
```

In the example above we are specifying format as JSON which allows us to share sessions in heterogenous environment, and also analyze them using Map/Reduce. But keep in the mind that not everything in typical rails application could be serialized to JSON, for example `ActionDispatch::Flash::FlashHash`. This is why the library serialize sessions using `Marshal.dump` by default.