Extracted from:

Deploying with JRuby 9k

Deliver Scalable Web Apps Using the JVM

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Deploying with JRuby 9k

Deliver Scalable Web Apps Using the JVM

Joe Kutner

Edited by Brian P. Hogan

Deploying with JRuby 9k

Deliver Scalable Web Apps Using the JVM

Joe Kutner

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Brian P. Hogan (editor)
Potomac Indexing, LLC (index)
Linda Recktenwald (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

Printed in the United States of America.
ISBN-13: 978-1-68050-169-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2016

Storing Sessions in Memcached

Twitalytics needs to track how many posts users create between the time they log in and the time they log out. This count is stateful and must be carried across transactions, survive any restarts of the application process, and be accessible from multiple processes when the app is scaled out. For these reasons, the count must be stored in a user's session.

Each time a user starts an interaction with a web page, a session is created. The session stores state that's carried over from one request to another for the same user. It usually includes things like username, breadcrumbs to track where they've been in the app, and even security tokens.

The default session storage mechanism in Rails is cookie based, which means the session state is stored on the client machine. This is an ideal place to put sensitive information, and it has a limited storage capacity. A better system will store session state server-side.

When storing session state on the server, you have the option to keep it in memory or in an external backing service. Keeping the session in memory is convenient, but it has some serious drawbacks. If the server process is restarted, all users will lose their current state. They may even need to log into the app again. If they were about to make a credit card transaction, they would be most unhappy.

But storing session state in memory is also a problem for scalability. In-memory session data cannot easily be distributed among multiple processes. If you need to stand up additional instances of your server to handle high volumes of traffic or ensure redundancy, you'll be in for trouble.

The best way to store session state, even for the simplest of apps, is in a backing service. One example is Memcached, which you'll use for Twitalytics. Memcached is a free, open source, high-performance, distributed-memory, object-caching system. It's useful as a key-value store to cache results of database calls, API calls, page rendering, and session state.

Installing Memcached

The easiest way to run Memcached locally is with Docker. Download the official Memcached image from DockerHub by running this command:

```
$ docker pull memcached
Using default tag: latest
latest: Pulling from library/memcached
dbacfa057b30: Pull complete
...
```

```
Digest: sha256:b335e191aac685a7ee7b9e3b4bfceef184f315412733f1ea099463fc5dcdb25e
Status: Downloaded newer image for memcached:latest
```

Now launch a new Docker container from the image and publish port 11211 by running this command:

```
$ docker run -p 11211:11211 --name memcached-server -d memcached
```

The server was started in a container that's running in the background. You can check its status with docker ps:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
3d6a392002f6        memcached           "/entrypoint.sh memca"   ...
```

And you can use telnet to test that Memcached itself is working. If you're not running Linux, you'll need to capture your Docker Machine IP address first. Run these commands:

```
$ docker-machine ip default
192.168.99.100
```

```
$ telnet 192.168.99.100 11211
Trying 192.168.99.100...
Connected to 192.168.99.100.
Escape character is '^]'.
```

Memcached is ready to store your session state. Enter quit at the prompt to end the Telnet session. Now connect Twitalytics to the Memcached server.

Using Memcached with Rails

Before you make any changes to Twitalytics, branch your code base by running these commands:

```
$ cd ~/code/twitalytics
$ git checkout -b services
Switched to a new branch 'services'
```

Connecting to Memcached from any kind of Ruby code requires a client library that knows how to speak the Memcached protocol. The de facto standard in the Ruby ecosystem is Dalli,[1] which is one of the many modern Ruby gems that works equally well with MRI and JRuby. To install Dalli, add these lines to the Twitalytics Gemfile:

Services/twitalytics/Gemfile

```
gem 'dalli'
gem 'connection_pool'
```

1. https://github.com/petergoldstein/dalli

The first gem is Dalli itself. The second gem, connection_pool, is what Dalli uses to pool Memcached connections, which ensures the Rails.cache singleton doesn't become a source of thread contention. This is important when using JRuby because it's a multithreaded runtime.

Save the Gemfile and run these commands to download and install the gems.

```
$ bundle install --binstubs
$ bin/rake rails:update:bin
```

Next, configure Rails to use Dalli as the default caching mechanism. Open the config/environments/development.rb file and add this line of code inside the Rails.application.configure block (but use the IP address of your Docker Machine in place of the IP address shown here):

Services/twitalytics/config/environments/development.rb

```
config.cache_store = :dalli_store, "192.168.99.100"
```

This sets Rails to use the Dalli client for all caching purposes in the app. But you also need to configure the session storage mechanism to use the cache instead of cookies. Open the config/initializers/session_store.rb file, and replace its contents with this code:

Services/twitalytics/config/initializers/session_store.rb

```
Rails.application.config.
  session_store :cache_store, key: '_twitalytics_session'
```

Now you can add the feature that tracks how many posts a user has created in a given session. Open the app/controllers/posts_controller.rb file, and add these lines of code to the create() method:

Services/twitalytics/app/controllers/posts_controller.rb

```
count = session[:count] || 0
session[:count] = count + 1
```

This increments a counter each time the Post#create action is executed.

To show the current count, open the app/views/posts/index.html.erb file, and add this code above the main table:

Services/twitalytics/app/views/posts/index.html.erb

```
<p>
  Created: <%= session[:count] || 0 %>
</p>
```

Now start the Puma server, and browse to http://localhost:3000/posts. You'll see the count set at zero. Create a few posts, and you'll see the count increase.

To confirm that the values are being stored in Memcached, check it with Telnet. Start a Telnet session and run the stats items command like this:

```
$ telnet 192.168.99.100 11211
Trying 192.168.99.100...
Connected to 192.168.99.100.
Escape character is '^]'.
stats items
STAT items:5:number 1
STAT items:5:age 5
STAT items:5:evicted 0
STAT items:5:evicted_nonzero 0
STAT items:5:evicted_time 0
STAT items:5:outofmemory 0
STAT items:5:tailrepairs 0
STAT items:5:reclaimed 0
STAT items:5:expired_unfetched 0
STAT items:5:evicted_unfetched 0
STAT items:5:crawler_reclaimed 0
STAT items:5:crawler_items_checked 0
STAT items:5:lrutail_reflocked 0
```

This displays an overview of the *items* in the cache. The number after the keyword items is the slab ID of the record. You can dump the record by running this command in the Telnet session:

```
stats cachedump 5 100
ITEM _session_id:fff8686f323aa547c936649e5a2f2131 [88 b; 1454260137 s]
```

It shows the item is keyed by a _session_id. The data isn't readable here, but it's enough to confirm that Rails is writing its session data to Memcached.

Your backing service is almost ready for production. The only problem is the hardcoded IP address of your Docker Machine in the configuration file and the lack of configuration for authentication credentials, which you'll need to secure your cache in production.

The IP address of the Memcached server and its credentials will be environment specific. That is, they will change depending on if the app is running locally, in test, or in production. For that reason, it's necessary to extract this information from environment variables rather than hard-coding it.

Open the config/environments/production.rb file, and add this code to the Rails.application.configure block:

Services/twitalytics/config/environments/production.rb

```
if ENV["MEMCACHEDCLOUD_SERVERS"]
  config.cache_store = :dalli_store,
                       ENV["MEMCACHEDCLOUD_SERVERS"].split(','),
                       { :username => ENV["MEMCACHEDCLOUD_USERNAME"],
                         :password => ENV["MEMCACHEDCLOUD_PASSWORD"],
                         :pool_size => ENV["MAX_PUMA_THREADS"] || 1 }
end
```

The MEMCACHEDCLOUD_SERVERS environment variable can contain multiple IP addresses because in production you'll want some kind of failover for this service. The other environment variables provide the username and password. MemcachedCloud is a cloud-based Memcached as a service provider that you'll use in a moment, which is why we've chosen the MEMCACHEDCLOUD_ prefix for these variables. The last parameter is the connection pool size, which is set to the same value as Puma's maximum thread count.

Shut down the Twitalytics server by pressing Ctrl-C. Then commit all of your changes to Git by running these commands:

```
$ git add .
$ git commit -m "memcached"
```

Now you can deploy.

Deploying with Memcached

The code you used in development is ready to run on Heroku. You just need to add a Memcached service to your app by running the following command:

```
$ heroku addons:create memcachedcloud:30
```

This creates a free MemcachedCloud backing service attached your Heroku app. It also sets the MEMCACHEDCLOUD_SERVERS environment variable and the other environment variables you need to connect to it with the configuration in your config/environments/production.rb file.

Now deploy the app by running this command:

```
$ git push heroku services:master
```

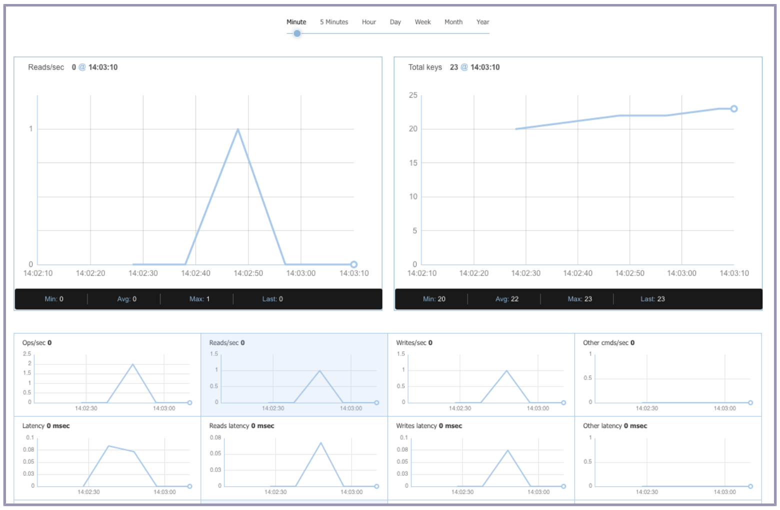
When the build is finished, open the Posts page with this command:

```
$ heroku open posts
```

Make a few requests to ensure that the session is exercised. Then verify that the session data is getting to Memcached by viewing the MemcachedCloud dashboard. Run this command to open it:

```
$ heroku addons:open memcachedcloud
```

From the dashboard, drill down into the Advanced Metrics view, and you'll see something like the following figure:



If you're not running on Heroku, you can use the Docker container you ran locally with the Rancher setup you created in Chapter 3, *Deploying a Rails Application*, on page ?. Log in to your Rancher virtual server by running `vagrant ssh`. Then pull the Docker image and run a new container just as you did locally. When you start your application containers, add the appropriate environment variable to the command options like this:

```
[rancher@rancher-server ~]$ docker run \
-e MEMCACHEDCLOUD_SERVERS=192.168.99.100:11211 \
-e DATABASE_URL=postgres://user:password@host:port/db \
-e PORT=3000 --publish=3000:3000 \
-dit username/twitalytics
```

In this case, you're reusing the `MEMCACHEDCLOUD_SERVERS` variable name, but you can use any name you'd like.

Memcached is an essential service—similar in importance to a relational database. Almost every web application will need to store session data, and doing so with a backing service ensures better failover and scalability. The next service you'll deploy is nearly as essential.