HTTP Scalability/Performance Best Practices

COMP 120 – Team 7

Abstract

Good performance is a fundamental tenant of a successful web application. No matter if the consumer of data is human or machine, information transfer between client and server must be done quickly and efficiently. Web applications using HTTP/HTTPS as an underling protocols can
quite easily be made to perform and scale using caching, compression and proper page design.

Audience

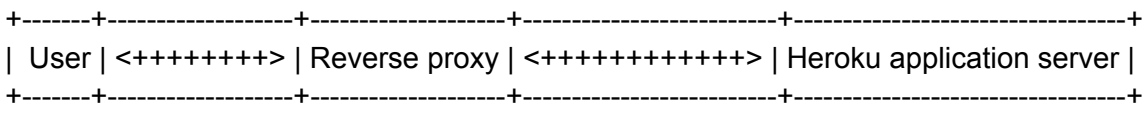
The Audience for this document is web application architects and developers.

HTTP Caching

Making use of HTTP's caching features is a great way to dramatically increase the performance of any web application. The main purpose of HTTP caching is to decrease transaction latency by reducing or eliminating the need to send responses between a server and a client. RFC 2616 explains it best: "Caching would be useless if it did not significantly improve performance. The goal of caching in HTTP/1.1 is to eliminate the need to send requests in many cases, and to eliminate the need to send full responses in many other cases. The former reduces the number of network round-trips required for many operations; we use an "expiration" mechanism for this purpose (see section 13.2). The latter reduces network bandwidth requirements; we use a "validation" mechanism for this purpose (see section 13.3)."

A Rails application is no exception. As an HTTP-based app being hosted on the internet, both users and the application server benefit greatly from the reduced load offered by caching. Being hosted on Heroku, the benefits of sending proper HTTP caching directives are even more profound as their entire infrastructure is fronted by an open source reverse proxy called Varnish to provide HTTP acceleration closer to the end user. This proxy caches content based on standard HTTP caching directives outlined in RFC 2616. This is an extremely powerful tool that allows Heroku users to serve out content directly from a cache rather than regenerating it at the application layer, which is slow, computationally expensive and usually much more expensive.

Simplified Heroku Architecture



Both static and dynamic content is candidate for caching by Varnish. Getting Varnish to cache requires attaching the following headers to GET responses:Cache-Control
The Cache-Control header is quite simple. A developer simply has to specify the header and add, in seconds, how long a proxy should serve the content from cache to the requester using the “max-age=” specifier. In the example, a proxy would keep in cache the response for 86400 seconds (1 day).

“*Cache-Control: max-age=86400*” => Instruct caches to store content for 1 day

Cache-Control headers can also be used to instruct caches not to store data
This Depaul University handout explains these functions quite well.

Here is another example:

“*Cache-Control: private, no-cache*” => Instructs cache that content is private, and thus cannot be stored on a shared cache

-“*Cache-Control: public, max-age=300*”_ => Instructs cache that content is public, and cacheable by any cache for for 300 seconds

RoR Implementation

Crafting HTTP headers in RoR is very simple. In the controller, place the following code:

`*response.headers[‘Cache-Control’] = ‘($PARAMETERS GO HERE)*’

$PARAMETERS contains whatever values are appropriate for the content. A good rule of thumb is to keep max-age for dynamic content very low (5-10 seconds) and high for static content (a few hours). Actual value must, however, be crafted with care as improper configuration can cause unwanted behavior (stale data being served out, user specific information leaking, etc.). Use carefully and test!

Compression

Using Heroku, compression is enabled by default. No action in necessary on the part of the developer.

Heroku Compression