
Work in progress title

(A L^AT_EX class)

Anton Fagerberg

`ada10afa@student.lu.se`

February 3, 2015

Master's thesis work carried out at Jayway.

Supervisor: Roger Henriksson, `Roger.Henriksson@cs.lth.se`

Examiner: Görel Hedin, `Gorel.Hedin@cs.lth.se`

Abstract

This document describes the Master's Thesis format for the theses carried out at the Department of Computer Science, Lund University.

Your abstract should capture, in English, the whole thesis with focus on the problem and solution in 150 words. It should be placed on a separate right-hand page, with an additional 1cm margin on both left and right. Avoid acronyms, footnotes, and references in the abstract if possible.

Leave a 2cm vertical space after the abstract and provide a few keywords relevant for your report. Use five to six words, of which at most two should be from the title.

Keywords: MSc, template, report, style, structure

Acknowledgements

If you want to thank people, do it here, on a separate right-hand page. Both the U.S. acknowledgments and the British acknowledgements spellings are acceptable.

We would like to thank Lennart Andersson for his feedback on this template.

Contents

1	Performance issues with HTTP 1.1	7
1.1	Headers	7
1.2	Maximum TCP connections	11
1.2.1	Chunked responses	12
1.3	Compression	13
2	API Gateway Theory	15
2.1	What is an API Gateway?	15
2.2	Multiple resources & requests	16
2.3	Duplicate & Unnecessary items	16
2.4	Format transformation	17
2.5	Pure REST	18
2.6	Compression	18
2.7	Authentication	19
2.8	Caching	19
2.9	Support older API versions	20
2.10	SSL	20
2.11	Metrics & analytics	20
	Bibliography	21

Chapter 1

Performance issues with HTTP 1.1

Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems[3]. The first standard version of HTTP 1.1 was released in January 1997[10]. The next version, HTTP/2 (originally named HTTP 2.0), is expected to be released in 2015 (ändra här om det publiceras innan ex-jobbet är klart). Although HTTP/2 addresses several of the HTTP 1.1 performance issues, it is reasonable to assume that it will take many years before HTTP/2 fully replaces HTTP 1.1 as the protocol used on all web servers and even longer for many legacy systems and clients. It is therefore relevant to acknowledge and mitigate the performance issues related to HTTP 1.1 even many years after the HTTP/2 release.

1.1 Headers

It is common in modern web applications to send a lot of HTTP requests toward a back-end API. These requests can be very small, such as an PUT request to update a single field but the actual payload can also be considered small when retrieving data from the server when it is compared to the total amount of data transmitted. Along with every HTTP request are typically plenty of headers. These headers can be a substantial part of every request and may therefore end up being the performance bottle neck if many small requests has to be transmitted.

As an example, consider the Instagram API[7] which has an end-point where you can get information about a certain user account with a response serialised in JSON format. If a client was built which is supposed to show details about, for example, the 10 specific users. We can then benchmark how making 10 separate API requests would differ, in transmitted data size, from how it would behave if we could fetch all 10 users with one request.

```
{
  "data": {
    "id": "1574083",
    "username": "snoopdogg",
    "full_name": "Snoop Dogg",
    "profile_picture": "http://distillery...",
    "bio": "This is my bio",
    "website": "http://snoopdogg.com",
    "counts": {
      "media": 1320,
      "follows": 420,
      "followed_by": 3410
    }
  }
}
```

Figure 1.1: User data response from the Instagram API in JSON format.

HTTP requests can be benchmarked using cURL[13]. To make the requests authentic and look like it was made from an actual browser, we tell cURL to use the default headers provided by the browser Firefox. These headers includes among other things browsers the User-Agent, media types which are acceptable responses and so on. A local server running on port 9000 is used to simulate the the Instagram API.

```
curl --trace-ascii - 'http://localhost:9000/user/snoopdogg'
-H 'Host: localhost:9000' -H 'User-Agent: Mozilla/5.0 (
Macintosh; Intel Mac OS X 10.10; rv:36.0) Gecko/20100101
Firefox/36.0' -H 'Accept: text/html,application/xhtml+
xml,application/xml;q=0.9,*/*;q=0.8' -H 'Accept-Language
: en-US,en;q=0.5' --compressed -H 'Connection: keep-
alive' -H 'Pragma: no-cache' -H 'Cache-Control: no-cache
,
```

Figure 1.2: cURL command used in the benchmarks.

Performing this request will give us the following results:

```
=> Send header , 355 bytes (0x163)
0000: GET /user/snoopdogg HTTP/1.1
001e: Accept-Encoding: deflate , gzip
003e: Host: localhost:9000
0054: User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
      10.10; rv:36.
0094: 0) Gecko/20100101 Firefox/36.0
00b4: Accept: text/html , application/xhtml+xml , application /
      xml;q=0.9 ,*/
00f4: *;q=0.8
00fd: Accept-Language: en-US,en;q=0.5
011e: Connection: keep-alive
0136: Pragma: no-cache
0148: Cache-Control: no-cache
0161:
<= Recv header , 17 bytes (0x11)
0000: HTTP/1.1 200 OK
<= Recv header , 47 bytes (0x2f)
0000: Content-Type: application/json; charset=utf-8
<= Recv header , 21 bytes (0x15)
0000: Content-Length: 286
<= Recv header , 2 bytes (0x2)
0000:
<= Recv data , 286 bytes (0x11e)
```

Figure 1.3: Results from cURL when sending an HTTP request to fetch one user. The actual response payload has been omitted.

We can from this information see that 355 bytes are sent as header data, 87 bytes are then received as header data (17 + 47 + 21 + 2) and the actual payload is 286 bytes. This means 61% of every request to this user end-point are nothing but header data.

If we instead would expose an endpoint where all 10 users could be requested with one HTTP request which returned an array of JSON objects, we would get the following result:

```
=> Send header , 446 bytes (0x1be)
0000: GET /users/snoopdog1,snoopdog2,snoopdog3,snoopdog4,
      snoopdog5,sno
0040: opdog6,snoopdog7,snoopdog8,snoopdog9,snoopdo10 HTTP
      /1.1
0079: Accept-Encoding: deflate , gzip
0099: Host: localhost:9000
00af: User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
      10.10; rv:36.
00ef: 0) Gecko/20100101 Firefox/36.0
010f: Accept: text/html,application/xhtml+xml,application/
      xml;q=0.9,*/
014f: *;q=0.8
0158: Accept-Language: en-US,en;q=0.5
0179: Connection: keep-alive
0191: Pragma: no-cache
01a3: Cache-Control: no-cache
01bc:
<= Recv header , 17 bytes (0x11)
0000: HTTP/1.1 200 OK
<= Recv header , 47 bytes (0x2f)
0000: Content-Type: application/json; charset=utf-8
<= Recv header , 22 bytes (0x16)
0000: Content-Length: 2871
<= Recv header , 2 bytes (0x2)
0000:
<= Recv data , 2871 bytes (0xb37)
```

Figure 1.4: Results from cURL when performing a HTTP request to fetch 10 users. The actual payload has been omitted.

From the results, we can see that the size of the sent headers has increased from 355 to 446 bytes because of the longer URL which specifies all users to fetch. The received headers are increased with just one byte from 87 to 88 because of the increased Content-Length field. This results in a total header size of 534 bytes. The actual payload has increased from 286 bytes to 2871 bytes - about 10 fold which is expected since we request 10 users at once instead of one at a time. A very small increase in data is added because of the array-characters in the JSON format. By avoiding doing 10 separate requests and instead do one concatenated request, the overhead added because of all the headers have now been reduced from 61% to 16%. This number will scale with to the number of requests concatenated - the more requests concatenated, the lesser amount of overhead from HTTP headers.

#	10 users, 10 request	10 users, 1 request
Headers	4,420 B	534 B
Payload	2,860 B	2,871 B
% headers of total data	61%	16%

Figure 1.5: Header and payload ratio when doing 10 separate or one concatenated request to fetch users.

The data displayed in this example should be viewed as a lower bound. In practise, HTTP cookies which are used for personalisation, analytics and session management are also sent with every HTTP request as headers and can add up to multiple kilobytes of protocol overhead for every single HTTP request[5, page 200].

This is one of the issues that may (ändra ”may” om det stämmer när http2 släpps) be mitigated by using HTTP/2 which remembers which headers that has already been sent and therefore doesn’t need to retransmit them on subsequent requests[5, page 222].

1.2 Maximum TCP connections

The HTTP 1.X protocol doesn’t allow data to be multiplexed over the same connection[5, p.194]. For this reason, browser vendors has introduced a connection pool of 6 TCP connections per host (the HTTP 1.1 specification limits the pool to 2 connections[2] per host, but modern browsers has refused to conform to this standard in order to decrease the load times).

A common way to deal with the connection limit is to use domain sharding. Since the limit of six TCP connections are on a host name basis, it is possible to create multiple subdomains to avoid this limit. If the subdomains {shard1, shard2, ...}.example.com where created and pointed to the same server, more than six TCP connections can be used in parallel at the same time from that machine. This approach is not without downsides as every new host requires a new DNS lookup, a TCP three-Way handshake and a slow start from TCP which can have negative impact on the load times[5, page 199] - just the DNS lookup typically takes 20-120 ms[11, page 63]. Another problem with domain sharding is that the browser always opens six connections per shard even if not all of them are used. In addition, domain sharding can be a complicated manual process and it is hard to calculate how many shards to use for optimal performance. When Yahoo investigated this problem they reached to conclusion that you should, as a rule of thumb, use at least two, but no more than four domain shards[12].

As an example, we can benchmark downloading thumbnails for an image gallery. Suppose we want to download 60 thumbnails, encoded in base-64 format, and the connection we are using has a lot of bandwidth but suffers from high latency.

If all images were downloaded by using a single HTTP request per thumbnail, we can see that the six TCP connection limit will become a bottleneck.

Method	Status	Type	Initiator	Size	Time	Timeline	1.00 s	1.50 s
GET	200	text/plain		13.3 KB	308 ms			
GET	200	text/plain		13.3 KB	308 ms			
GET	200	text/plain		13.3 KB	308 ms			
GET	200	text/plain		13.3 KB	310 ms			
GET	200	text/plain		13.3 KB	310 ms			
GET	200	text/plain		13.3 KB	309 ms			
GET	200	text/plain		13.3 KB	614 ms			
GET	200	text/plain		13.3 KB	613 ms			
GET	200	text/plain		13.3 KB	611 ms			
GET	200	text/plain		13.3 KB	615 ms			
GET	200	text/plain		13.3 KB	615 ms			
GET	200	text/plain		13.3 KB	614 ms			
GET	200	text/plain		13.3 KB	917 ms			
GET	200	text/plain		13.3 KB	916 ms			
GET	200	text/plain		13.3 KB	916 ms			
GET	200	text/plain		13.3 KB	920 ms			
GET	200	text/plain		13.3 KB	920 ms			
GET	200	text/plain		13.3 KB	920 ms			

Figure 1.6: Chrome developer tools showing how the six TCP connection limit becomes a bottle neck on a connection with 300 ms of latency.

We can calculate the total delay caused by latency in our example by this formula:

$$\text{total latency} = \text{number of thumbnails} * \frac{\text{latency per request}}{\text{number of parallel requests}} \quad (1.1)$$

In our example, we fetch 60 thumbnails and with a latency of 300 ms per request. Our browser can handle six parallel TCP connections which gives us the following result:

$$\text{total latency} = 60 * \frac{300}{6} = 3,000 \text{ ms} \quad (1.2)$$

If we instead could concatenate these 60 thumbnail requests into one request and the response would contain all of the thumbnails - then we would only have to pay the latency cost once which would reduce the total latency by an order of magnitude from 3,000 ms to 300 ms.

It is worth pointing out that increasing the bandwidth would not resolve this problem as the latency is the only bottleneck in this scenario. It is not uncommon for browsers to wait idle for 100 - 150 ms before spending 5 ms to download an image which means that latency is accounting for 90-95% of the total time for the HTTP requests[6].

1.2.1 Chunked responses

When fetching thumbnails, you often want to display them as soon as each individual image has been loaded and not wait for the entire concatenated response. When using a concatenated request to fetch multiple resources at the same time, the server can use chunked transfer encoding in the HTTP response to send the thumbnail data in chunks[1]. By doing so, images can be loaded as soon as they are available in the client, even if loaded out of order[9].

1.3 Compression

All requested data should be compressed before it is sent to the client. A common compression algorithm used in HTTP requests is Gzip (GNU Zip) which works best on text-based files such as HTML, CSS and JavaScript. Gzip has an expected compression rate of 60-80% when used on text-based files[5, page 237].

It is worth mentioning that there are scenarios where Gzip compression applied to very small files can increase the total size because of the Gzip dictionary overhead. This problem can be mitigated by defining a minimum file size threshold[4].

As an example, arbitrary user data for 50 users was created and stored in JSON format. When this data was requested from the server without compression, the total content-length of the HTTP request payload amounted to 55,205 bytes. When applying Gzip compression to the same data, the content length was reduced to 16,563, which amounts to a space saving of 70%.

$$\text{Space Saving} = 1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}} = 1 - \frac{16,563}{55,205} = 70\% \quad (1.3)$$

An important thing to note about Gzip compression is that only the payload is compressed in HTTP 1.1[14]. This means that the headers including cookies are not compressed which would've otherwise been an additional performance gain. This is one of the improvements which have been addressed in HTTP/2[5, page 222].

Chapter 2

API Gateway Theory

When developing clients for a back-end API, you often find that the clients need and the API's functionality isn't a perfect match. Different functionality is often required based on whether the client is a mobile application, a desktop application or something entirely different. The way the clients want to use the API can also radically differ. Not being able to optimise the API for each client's need can hurt the client's performance which has to do a lot of extra work but can also strain the developer whom may have to do extra work to fit the API for every client. One approach to solve this problem is by utilising an API gateway.

2.1 What is an API Gateway?

An API gateway works as an additional layer between the client and the server. For an API gateway to be efficient, it has to be able to augment the communication between the client and the server, and by doing so, improve the client performance and developer implementation of it.

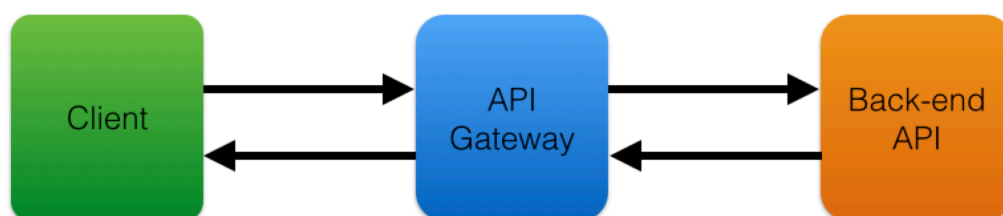


Figure 2.1: A simple scenario using an API gateway between a client and a back-end API.

2.2 Multiple resources & requests

A client often has to perform many requests simultaneously, either to one or multiple back-end APIs. Typical scenarios are when a user loads a web application for the first time and the applications initial state and data has to be retrieved or when multiple connected resources has to be loaded.

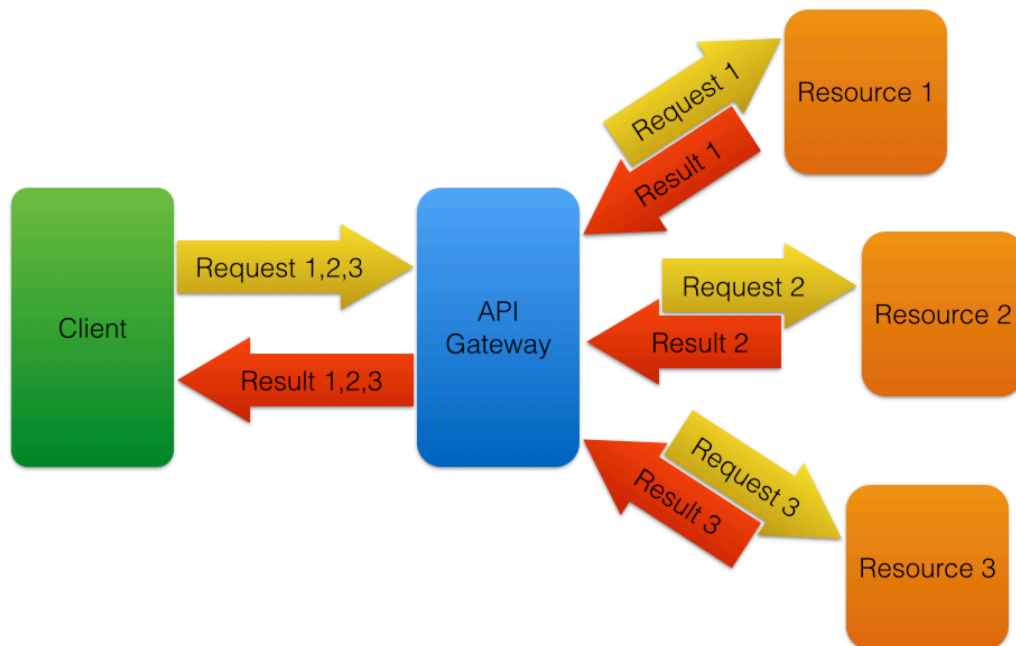


Figure 2.2: An API gateway receives a concatenated request which it distributes to multiple resources, the responses are then concatenated into a single response. The resources can either be one or several back-end APIs.

When working with HTTP requests, there are multiple penalties for executing many small requests compared to one concatenated request. These penalties includes the previously mentioned limit of maximum TCP connections (page 11) and overhead from http headers (page 7).

2.3 Duplicate & Unnecessary items

When requesting data from a back-end API, the response may contain unnecessary data which the client doesn't want. In a similar fashion, if a client performs several similar requests, it is possible the all the responses contains some duplicate data. By using an API gateway, the results from the back-end API(s) can be augmented to better fit the individual clients needs.

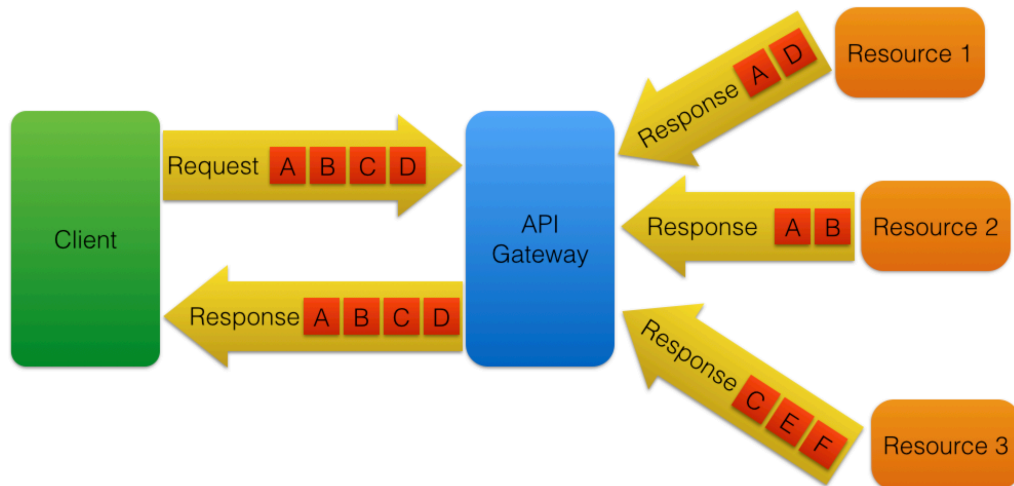


Figure 2.3: Client requests item A, B, C, D. The API gateway fetches A, D from Resource 1. Item A from the Resource 2 response can be discarded since it's duplicate data. Item E, F from Resource 3 can be discarded since they are unnecessary. The API gateway can after retrieval respond with just A, B, C, D. The resources can either be one or several back-end APIs.

2.4 Format transformation

It is common when working with older legacy systems that the data is formatted in a way which is not suitable for modern clients. In the case of JavaScript, the browsers have excellent built in support for JSON but not for XML. An API gateway can on the fly convert the data from the back-end to a format more appropriate for the client and respond with it.

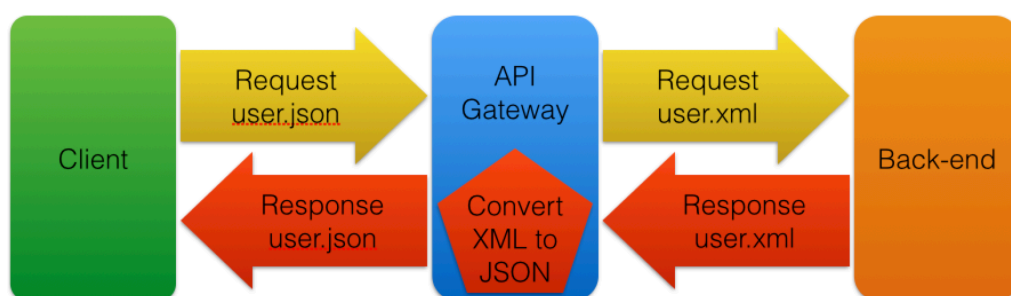


Figure 2.4: The client requests "user" in JSON-format. The API gateway fetches "user" in XML-format from the back-end, converts it to JSON and responds to the client in the appropriate format.

This has the benefit that the conversion code doesn't have to be rewritten for different types of clients. By performing the transformation in the gateway, the processing load is

moved away from the client which can improve its performance.

2.5 Pure REST

If an API follows the strict rules of REST, it uses the concept of Hypermedia as the Engine of Application State (HATEOAS). Instead of defining a bunch of end-points which the client can utilise, it requires the client to discover the resources itself by first performing a GET request to the root. To root will respond with the resources available, such as users. The client then has to query the users root to discover which requests can be made in regards to users - and so on. Working in this pure way places a very high bar for the client developer[8, page 61].

API gateways can be used to transform a Pure REST API with HATEOAS to a simpler API which only follows some of the restrictions put in place by REST. This can significantly lower the amount of traffic between the client and the back-end which can be a big performance gain in cases such when the latency is high between the client and back-end (assuming that the latency between the API gateway and back-end is low such as when they are placed in the same LAN).

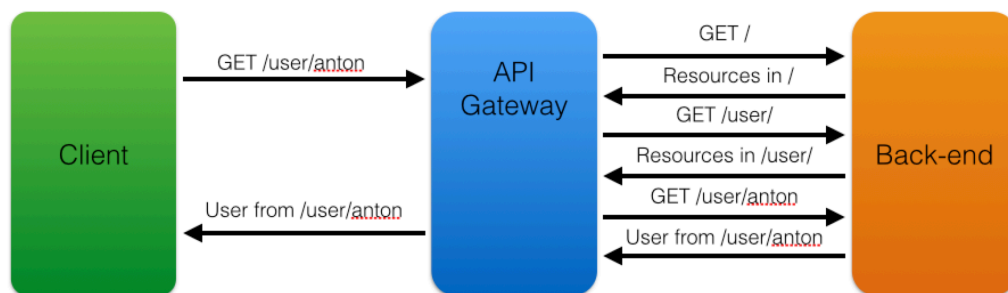


Figure 2.5: Scenario where the API gateway performs the pure REST HATEOAS-communication and at the same time exposes a simple end-point for the client.

2.6 Compression

API gateways can be used to compress responses in the cases where no compression is present on the back-end API. This can significantly reduce the amount of traffic the client has to receive which increases the performance - especially on mobile devices. HTTP compression was explored on page 13 where it was noted that Gzip has an expected compression level of 60-80% on text-based files.

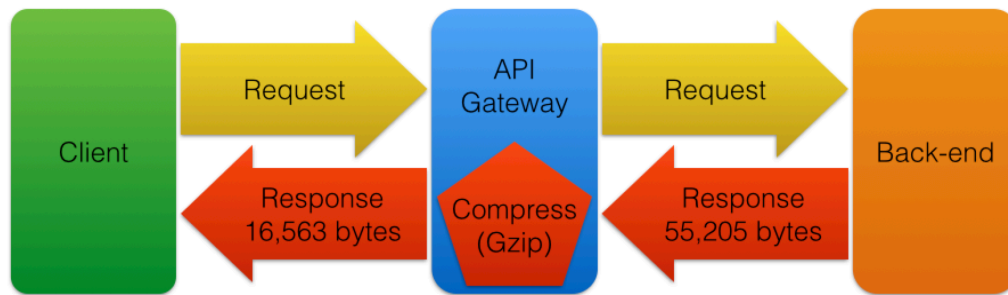


Figure 2.6: The API gateway compresses the response from the back-end API with Gzip which reduces the response traffic in the client by 70%. Numbers taken from the example on page 13.

2.7 Authentication

2.8 Caching

Responses from frequent API calls can be cached in the API gateway to reduce the load on the back-end system[8, page 107]. The cache can have a specified lifetime or be invalidated based on certain events such as a table update on a database.

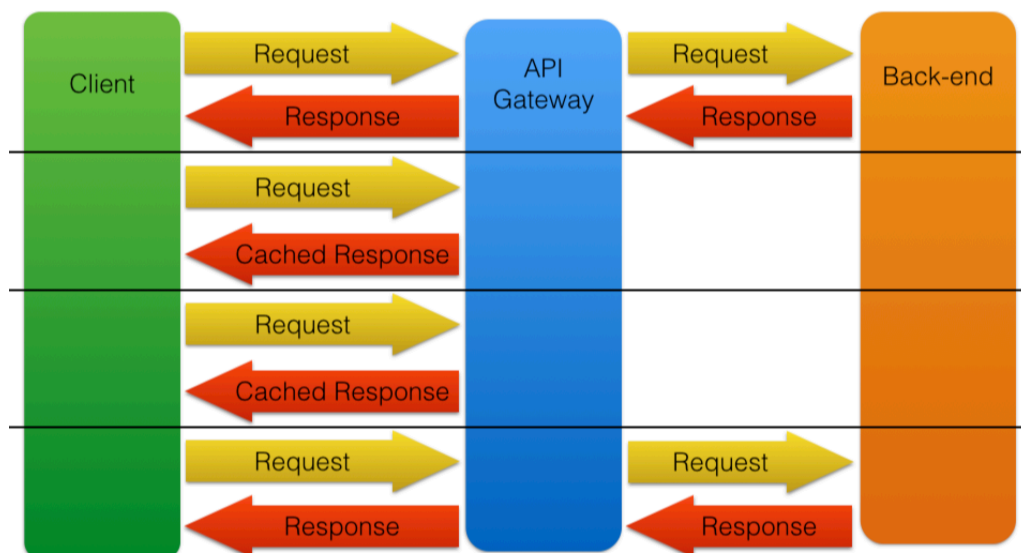


Figure 2.7: Frequent API calls to the same end-point can be cached to reduce the load on the back-end.

2.9 Support older API versions

2.10 SSL

2.11 Metrics & analytics

Bibliography

- [1] J Fielding, R.; Reschke. Hypertext transfer protocol (http/1.1): Message syntax and routing - chunked transfer coding. <http://tools.ietf.org/html/rfc7230#section-4.1>.
- [2] J.; Mogul J.; Frystyk H.; Masinter L.; Leach P.; Berners-Lee T Fielding, R.; Gettys. Hypertext transfer protocol – http/1.1. <http://www.ietf.org/rfc/rfc2616.txt>.
- [3] James; Mogul Jeffrey C.; Nielsen Henrik Frystyk; Masinter Larry; Leach Paul J.; Berners-Lee Fielding, Roy T.; Gettys. Hypertext transfer protocol – http/1.1. ietf. rfc 2616. <https://tools.ietf.org/html/rfc2616>.
- [4] I Grigorik. Optimizing encoding and transfer size of text-based assets. <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer>.
- [5] I Grigorik. High-Performance Browser Networking. O’Rilley Media, Inc., 2013.
- [6] B Hoffman. Bandwidth, latency, and the size of your pipe. <http://zoompf.com/blog/2011/12/i-dont-care-how-big-yours-is>.
- [7] instagram.com/developer. User endpoints. http://instagram.com/developer/endpoints/users/#get_users.
- [8] G.; Woods D. Jacobson, D.; Brail. APIs: A Strategy Guide. O’Rilley Media, Inc., 2012.
- [9] Z Mahkovec. Improving dropbox performance: Retrieving thumbnails. <https://tech.dropbox.com/2014/01/retrieving-thumbnails/>.
- [10] R. Fielding; UC Irvine; J. Gettys; J. Mogul; DEC; H. Frystyk; T. Berners-Lee; MIT/LCS. Hypertext transfer protocol – http/1.1.

- [11] S Souders. High Performance Web Sites. O’Rilley Media, Inc., 2007.
- [12] S.;YUI Team Souders. Performance research, part 4: Maximizing parallel downloads in the carpool lane. <http://yuiblog.com/blog/2007/04/11/performance-research-part-4/>.
- [13] D. Stenberg. curl groks urls. <http://curl.haxx.se/>.
- [14] D Stenberg. Http transfer compression. <http://daniel.haxx.se/blog/2011/04/18/http-transfer-compression/>.