Optimising clients with API gateways

Anton Fagerberg
anton@antonfagerberg.com

June 5, 2015

Master's thesis work carried out at Jayway AB.

Supervisors: Roger Henriksson, Roger.Henriksson@cs.lth.se
Nils-Olof Bankell, Nils-Olof.Bankell@jayway.com

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

Abstract

This thesis investigates the benefits and complications around working with API (Application Programming Interface) gateways. When we say API gateway, we mean to proxy and potentially enhance the communication between servers and clients, such as browsers, by transforming the data. We do this by examining the underlying protocol HTTP/1.1 and the general theory regarding API gateways.

An API gateway framework was developed to help further understand some of the common problems and provide a way to rapidly develop prototype solutions to them. The framework was applied in three case studies in order to discover potential problematic areas and solve them in real world production systems. We could from these results see that the benefits gained from using an API gateway varied from case to case, and with results in hand, predict in which scenarios API gateways are the most beneficial.

Keywords: API, gateway, proxy, communication, optimisation, performance, HTTP

Acknowledgements

TODO...

Contents

Chapter 1
Introduction

This thesis work started with the assumption that the network traffic between back-end server APIs (Application Programming Interfaces) and the clients using them was not properly optimised. The reason behind this was thought to be a mismatch between the client expectations and the predefined server responses. If, for example, a back-end API was developed with a desktop client in mind and a mobile client was introduced later on, the traffic to the mobile client would not be properly adapted to fit its specific needs.

There are many reasons why the back-end servers themselves cannot be rewritten. It can be because of cost factors, risk of breaking existing clients, ownership and licensing issues or even lack of proper knowledge. Because of reasons like these, we wanted to investigate whether the introduction of a new software layer between the client and server could mitigate these issues.

1.1 Method

The previously mentioned new software layer between the client and server corresponds to the concept of an API gateway. This thesis has been designed to consist of four majors chapters, all of which build upon the previous chapters—these chapters are briefly introduced below. Finally, we end the thesis with a conclusion chapter which ties the acquired knowledge from the four major chapters together.

Performance issues with HTTP/1.1

First we look at the transport protocol HTTP, especially HTTP/1.1, and what problems it introduces when the server and client does not communicate in an efficient manner. We look at the problematic areas in the protocol and how they, by utilising clever tricks from the industry, have been mitigated over the years.

API gateways in theory

Secondly, we theorise around the broad subject of API gateways. Here we try to define some of the different ways the API gateway can improve the relationship between clients and servers. We investigate how the problems explored in the previous chapter about HTTP can be solved by utilising an API gateway.

API gateway framework

Thirdly, an API gateway framework was created to in order to better understand the API gateway problems from a practical and a more technical point of view. This framework allows us to not only understand but also benchmark the problems defined in earlier chapters and provide real applicable solutions to them.

Case studies

Finally, we did case studies on three real-world production systems. Each case study consist of an analysis to determine whether the system had any issues which could be improved with the introduction of an API gateway. A solution was created for a selected part of each system with the framework described in the previous chapter. This was done in order to verify that not only could the framework be used in real-world scenarios, but also to provide a method for benchmarking the results before and after the introduction of the API gateway. By doing so, we can determine in which scenarios it is practical to implement an API gateway, how it can be done from a practical point of view and what the expected results will be.

Chapter 2
Performance issues with HTTP/1.1

Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems [1]. The first standardised version of HTTP/1.1 was released in January 1997 [2]. The subsequent version, HTTP/2 (originally named HTTP/2.0), was approved for publication as a proposed standard on Feb 17, 2015 by the Internet Engineering Steering Group (IESG) and the HTTP/2 specification was published in May 2015 [3]. Although HTTP/2 addresses several of the HTTP/1.1 performance issues, it is reasonable to assume that it will take at least a decade [4, page 21] before HTTP/2 fully replaces HTTP/1.1 as the default protocol used on all web servers and middle-boxes such as proxies and firewalls—and even longer for many legacy back-end systems and clients used in the slow-moving enterprise environment. It is therefore necessary to acknowledge and mitigate the performance issues related to HTTP/1.1 even many years after the release of HTTP/2.

2.1 Headers

It is common in modern web applications to send a lot of HTTP requests, consisting of headers and a payload, toward one or many back-end APIs. The payload of these requests can be very small, such as a PUT request with the intention of updating a single field or even a GET request to retrieve a user profile. It is very noticeable, when the payload is small, just how much data has to be transferred along with it in order to perform a HTTP request.

There are typically plenty of headers transferred with every HTTP request and these headers can end up being a substantial amount of the total data in every request. The data stored in the headers may end up being the performance bottle neck in many HTTP requests [5]—especially if a lot of small requests has to be transmitted on a frequent basis.

As an example, consider the Instagram API [6] which has an end-point where you can get information about a certain user account. The response from the API is encoded in

JSON[1] format as seen in Figure 2.1.

```
{
    "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 2.1: The user profile response from the Instagram API, encoded in JSON format.

[1] JavaScript Object Notation, see appendix.

Suppose a client was built with the intention of showing details about ten users. We can benchmark how making ten separate API requests would differ, in transmitted HTTP data size, from how it would behave if we could fetch all ten users with one request—the difference being that the headers are sent only once in comparison with ten times.

The HTTP requests can be benchmarked with the command-line tool cURL [7]. In this example, we let the browser Firefox generate the cURL command in order to make the request look like it was sent from a regular browser. This includes setting headers such as the accepted media types, the user agent, cookies and so on.

```
curl --trace-ascii -
    'https://api.instagram.com/v1/users/1574083/?access_token=[...]'
  -H 'Host: api.instagram.com'
  -H 'User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10;
    rv:40.0) Gecko/20100101 Firefox/40.0'
  -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 'DNT: 1'
  -H 'Cookie: [...]'
  -H 'Connection: keep-alive'
  -H 'Pragma: no-cache'
  -H 'Cache-Control: no-cache'
```

Figure 2.2: The cURL command used in the benchmark—the access token and cookies have been omitted.

```
=> Send header, 705 bytes (0x2c1)
0000: GET /v1/users/1574083/?access_token=[...]
0040: [...] HTTP/1.1
0060: Accept-Encoding: deflate, gzip
0080: Host: api.instagram.com
0099: User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:40.
00d9: 0) Gecko/20100101 Firefox/40.0
00f9: Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/
0139: *;q=0.8
0142: Accept-Language: en-US,en;q=0.5
0163: DNT: 1
016b: Cookie: [...]
02bf:
<= Recv header, 17 bytes (0x11)
0000: HTTP/1.1 200 OK
<= Recv header, 29 bytes (0x1d)
0000: X-Ratelimit-Remaining: 4996
<= Recv header, 22 bytes (0x16)
0000: Content-Language: en
<= Recv header, 24 bytes (0x18)
0000: Content-Encoding: gzip
<= Recv header, 40 bytes (0x28)
0000: Expires: Sat, 01 Jan 2000 00:00:00 GMT
<= Recv header, 48 bytes (0x30)
0000: Vary: Cookie, Accept-Language, Accept-Encoding
<= Recv header, 25 bytes (0x19)
0000: X-Ratelimit-Limit: 5000
<= Recv header, 18 bytes (0x12)
0000: Pragma: no-cache
<= Recv header, 61 bytes (0x3d)
0000: Cache-Control: private, no-cache, no-store, must-revalidate
<= Recv header, 37 bytes (0x25)
0000: Date: Wed, 27 May 2015 10:28:24 GMT
<= Recv header, 47 bytes (0x2f)
0000: Content-Type: application/json; charset=utf-8
<= Recv header, 121 bytes (0x79)
0000: Set-Cookie: [...]
0040: [...]
<= Recv header, 24 bytes (0x18)
0000: Connection: keep-alive
<= Recv header, 21 bytes (0x15)
0000: Content-Length: 271
<= Recv header, 2 bytes (0x2)
0000:
<= Recv data, 271 bytes (0x10f)
0000: ....H.eU..m.Mo. ..............7....V.E .........z........Zd..}..
0040: ..."..3...E/W.....k.....hP..............&.....!...../|.q...]v.~~
0080: .C\.u..*-{.F.?3.7..Q...6X........f...g....|q..,T8-(.....cLkV..55
00c0: %....<.=..H,/...7w..F..-..6........qd....v....QK.c....tLh..S....
0100: !9-....,...a...
```

Figure 2.3: Result from cURL. Cookies and access token omitted.

We can from the output in Figure 2.3 see that 705 bytes are sent as request header data (line 1), 536 bytes are received as response header data (line 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 39, 41 and 43) and the actual response payload is 271 bytes (line 45). This means that 82% of the data, in every request, sent to this Instagram API end-point is nothing but header data.

The header data is often useful and in many cases required so we can not just discard it. Consider instead if we could expose a new end-point where all ten users could be requested simultaneously with one HTTP request instead of one request per user. In that case, we would get away with only transmitting the header data once and not ten times.

#	10 users, 10 request	10 users, 1 request
Total bytes headers	12,410	1,241
Total bytes payload	2,710	2,710
% headers of total data	82%	31%

Figure 2.4: The headers-payload ratio when executing one request per user compared to ten users in one request. The overhead percentage will continue to decrease linearly in relation to the number of users requested simultaneously.

The header data used in this example can be viewed as a lower bound. A large quantity of HTTP cookies, which are used for personalisation, analytics and session management, are also sent with every HTTP request as part of the header data and can add up to multiple kilobytes of protocol overhead for every single HTTP request [8, page 200].

Another interesting thing in Figure 2.3 happens on line 46–50 which is where the payload is. While it looks like gibberish, it is actually the user profile data in JSON format seen in Figure 2.1 but compressed using Gzip (more about compression on page 16). Compressing responses is a great bandwidth saving technique—however, only the payload can be compressed in HTTP/1.1, not the headers.

This is one of the issues that can be mitigated by using HTTP/2 since it allows the header fields to be encoded with a static Huffman code which reduces the transfer size. HTTP/2 also require that the client and the server remember the previously seen header fields so that they can be used as a reference when encoding previously sent values [4, page 19].

2.2 Maximum TCP connections

The HTTP/1.1 protocol does not allow data to be multiplexed over the same connection [8, p.194]. For this reason, most browser vendors have introduced a connection pool of six TCP connections per host (the HTTP/1.1 specification limits the pool to two connections [1] per host, but modern browsers have refused to conform to this standard in order to decrease the load times). The limitation caused by the fixed number of connections in the connection pool is known as the “maximum TCP connections”.

A common way to deal with the connection limit is to use domain sharding [9] . Since the limit of six TCP connections is on a host name basis, it is possible to create multiple

subdomains to get around this limitation. If the subdomains {shard1, shard2, ...}.example.com are created and they all pointed to the same server, then more than six TCP connections can be used in parallel at the same time to the same server from a browser.

This approach is unfortunately not without its downsides as every new hostname requires a new DNS lookup and each new TCP stream requires a TCP three-way handshake and a TCP slow start, all of which can have a negative impact on the load times [8, page 199]—just the DNS lookup typically takes 20-120 ms [10, page 63]. Another problem with domain sharding is the fact that the browser always establishes six connections per shard even if not all, or even any of them are used. In addition to these problems, domain sharding is a complicated manual process to set-up and it is hard to determine how many shards to use for achieving optimal performance. Yahoo investigated this problem and they concluded that you should, as a rule of thumb, use at least two, but no more than four domain shards [9].

To illustrate this problem with an example, we can benchmark the impact of the connection pool limit when downloading thumbnails for an image gallery. Suppose we want to download 60 thumbnails and that the connection we are using has a lot of bandwidth but suffers from high latency.

We can see in Figure 2.5 that the six TCP connection limit will become a bottleneck if all images are retrieved with one HTTP request per image. The resulting network graph will typically look like "stairs" where the requests wait in groups of six for a free TCP connection.

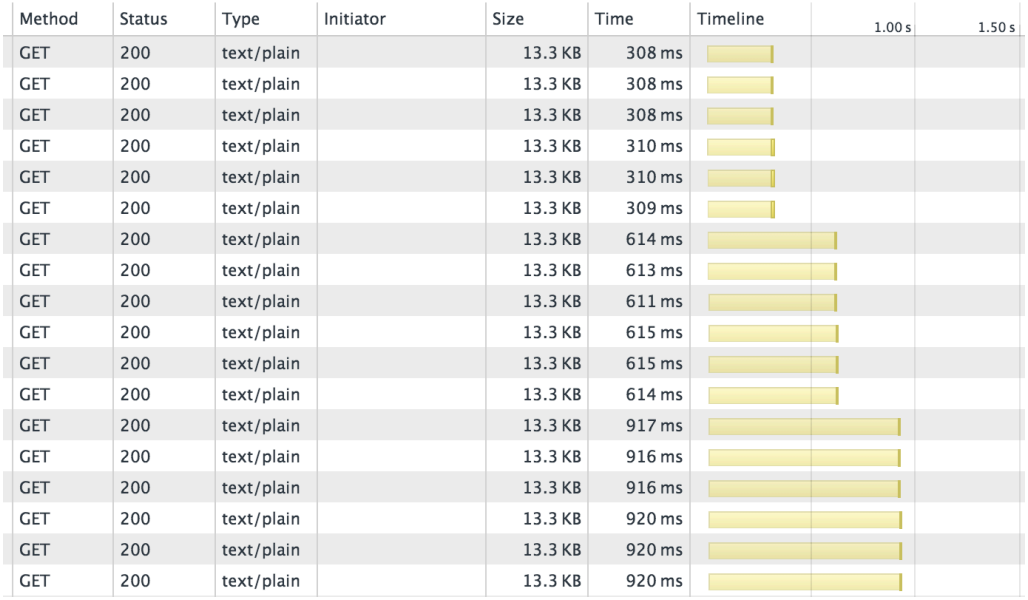

Figure 2.5: Chrome developer tools showing how the six TCP connection limit becomes a bottle neck on a connection with 300 ms of latency.

We can calculate the total amount of delay caused by latency in our example with the following formula:

$$\text{total latency} = \text{number of thumbnails} * \frac{\text{latency per request}}{\text{number of parallel requests}} \tag{2.1}$$

In our example, we fetch 60 thumbnails on a connection which has a latency of 300 ms to the server. Our browser (Google Chrome) can handle six parallel TCP connections which gives us the following equation:

$$\text{total latency} = 60 * \frac{300}{6} \text{ ms} = 3,000 \text{ ms} = 3 \text{ seconds} \tag{2.2}$$

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. This would reduce the total amount of latency by an order of magnitude, from 3,000 ms to 300 ms, since we only have to pay the price for the latency once and not for every six thumbnails.

Other similar approaches to the same problem include CSS Sprites [11] where a predefined set of images, such as icons, are merged in to one large image file. Individual images are then displayed by rendering parts of the larger image across the website. Note that this approach only works on a predefined set of images since the image merging process is costly and it would therefore not work in the thumbnail example above.

Text-based files such as JavaScript source code files and CSS stylesheets can also be concatenated into larger files during the build process in order to decrease the amount of HTTP requests needed [12].

As a side-note, it is worth pointing out that increasing the bandwidth of the connection would not resolve this problem as the latency is the only bottleneck in this example. We often focus on increasing the bandwidth as our connections to the internet improve when we perhaps should focus more on the latency instead.

It is not uncommon for browsers to wait idle for 100–150 ms before spending 5 ms actually downloading an image. This means that latency often accounts for 90–95% of the total waiting time for HTTP requests [13].

This issue has been addressed in HTTP/2 which utilises one multiplexing connection instead of a connection pool [3]. This means that a single TCP connection is able to handle all the requests and responses concurrently and by doing so removes the need for a fixed number of blocking TCP connections.

2.2.1 Chunked responses

In the previous example where we fetched thumbnails, we often want to display each thumbnail as soon as each individual image has been loaded. This could cause problems now that we are using one concatenated request instead of a separate request for each image. Fortunately we can utilise chunked responses for this—that is sending the currently available parts of the response as individual pieces, or chunks, to the client before the entire response is known.

The HTTP server can utilise chunked transfer encoding in the HTTP responses in order to send the individual thumbnail data in chunks to the client [14]. By doing so, images, or any other type of data, can be rendered in the client as soon as each chunk is available, even out of order if necessary.

This approach, with concatenated requests and chunked responses, has been successfully been implemented at Dropbox in their gallery software implementation [15]—much in the same fashion as the previous example.

Chunked transfer encoding is the only encoding which HTTP/1.1 clients are required to understand [1]. This makes it very attractive to use—especially in the use cases where data chunks can be separated in to logical pieces.

2.3 Compression

All requested data, especially text based data, can be compressed before it is sent to the client in order to reduce the transferred data size. A common compression algorithm used together with HTTP requests is GNU Zip (Gzip). Gzip works best on text-based files such as HTML, CSS and JavaScript and has an expected compression rate of 60–80% when used on text-based files [8, 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 [16].

As an example, arbitrary user data[2] for 50 users was created and encoded in JSON format. When this data was requested from a server without compression, the total size of the HTTP request payload amounted to 55,205 bytes of data. By applying Gzip compression to the same data, the content length was reduced to 16,563 bytes of data which amounts to a 70% space saving.

$$\text{Space Saving} = 1 - \frac{\text{Compressed size}}{\text{Uncompressed size}} = 1 - \frac{16,563}{55,205} \approx 70\% \tag{2.3}$$

An important thing to note about Gzip compression is that only the payload is compressed in HTTP/1.1 [1]. This means that the headers, including cookies, are not compressed which would have otherwise been an additional performance gain. This is one of the improvements which have been addressed in the development of HTTP/2 [4, page 19].

2.4 Further reading

High Performance Browser Networking - What every web developer should know about networking and web performance, Ilya Grigorik, 2013, O'Reilly Media

The essential book about browser networking performance. It covers many aspects around browser networking and the limitations within the HTTP protocols which are essential to understan in the development of performance increasing API gateways.

[2] `https://gist.github.com/AntonFagerberg/32ddde695fb0e2581176`

HTTP/2: A New Excerpt from High Performance Browser Networking, Ilya Grigorik, 2015, O'Reilly Media

A new excerpt from the High Performance Browser Networking book. The new excerpt describes the new features in the HTTP/2 protocol such as the header compression and request/response multiplexing which has to be taken into consideration in API gateways.

High Performance Web Sites - Essential Knowledge for Front-End Engineers, Ilya Grigorik, 2007, O'Reilly Media

Techniques for building high performing web sites. These techniques can also be applied inside API gateways.

Even Faster Web Sites - Performance Best Practices for Web Developers, Ilya Grigorik, 2009, O'Reilly Media

Follow-up book to High performance Web Sites with additional techniques which can be applied inside API gateways.

Nine Things to Expect from HTTP/2, Mark Nottingham, 2014, mnot's blog

A blog post from Mark Nottingham, chair of the IETF HTTP Working Group and a member of W3C TAG, in which he briefly explores nine tings to expect from HTTP/2.
`https://www.mnot.net/blog/2014/01/30/http2_expectations`

Chapter 3
API gateways in theory

When developing clients for back-end APIs, you often find that the client's needs and the back-end APIs functionality is not a perfect match. On top of that, 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 based on what kind of product is being developed.

Not being able to optimise the API for each individual client's needs can hurt the performance since it has to do a lot of extra work to refit the back-end's model to its own model—but it can also strain the developer who may have to refit the API for each new client.

Changing the back-end API is often not possible, perhaps especially in the enterprise environment where things tend to move slowly. The back-end can be a legacy system where changes are not allowed to take place—it might be impractical to adapt the back-end for different client types without breaking existing clients or the back-end development team might be strained for any other reason.

One approach to mitigate these problems is by utilising an API gateway [17]—a new software layer between the clients and the back-end APIs. By introducing an API gateway, API calls can be modified in many different ways when they are transported between the client and the back-end API.

In this chapter, we will walk through what an API gateway is and what it can do. Many of the potential usage areas described here are derived from problems encountered in the industry while other usage areas are adaptions of existing similar concepts which can seen in section 3.19.

3.1 What is an API gateway?

An API gateway works on a new layer between the clients and the back-end API servers. For an API gateway to be efficient, it has to be able to modify the communication between

the clients and the servers, and by doing so, improve the clients and potentially also the back-end API servers performance.

The focus in this thesis is to see how the clients can be optimised in terms of performance but also regarding code complexity and developer productivity. Little effort is taken to optimise the back-end API server—the goal is however not to put more strain on the server after introducing the API gateway but rather to keep it on the same level as before.

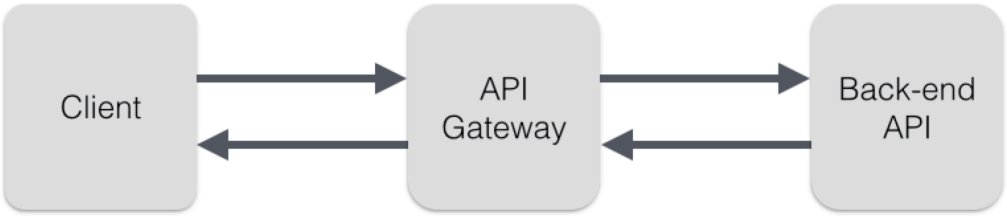

Figure 3.1: The API gateway is placed as a new separate layer between a client and a back-end API server.

One might suspect that the amount of code will increase as we introduce yet another software component. However, in many scenarios, the code we write in the API gateway had to be written in the client if the API gateway did not exist. With this reasoning, we can use, to some extent, the same amount of code but gain a lot from just placing it closer to the back-end APIs.

3.2 Differing client needs

We now, perhaps more than ever, have a vast variety of consumer devices such as mobile phones, tablets, desktop computers and other smart devices such as TVs—all of which often utilise the same APIs. One can imagine an API which returns a collection of the latest uploaded images to some service.

Since the screen size is drastically different on a mobile phone compared to a TV, the number of images the client wants to retrieve from the API can vary a lot. Depending on the type of client which is requesting the data, the API gateway can adapt the number of returned images.

A similar approach has been implemented at Netflix where each client development team write their own “adaptor” code to fully optimise the underlying API for their client’s specific needs [18]. This concept works much like how an API gateway works—the main difference being whether the adaptors are actually part of the back-end system or in a new software layer.

3.3 Multiple resources and requests

A client often want to perform many requests simultaneously, either to one or multiple back-end APIs. A typical scenarios is when a user loads a single-page web application for

the first time and the application's initial state has to be retrieved. Another typical example is when multiple resources, which are connected in some fashion, has to be loaded.

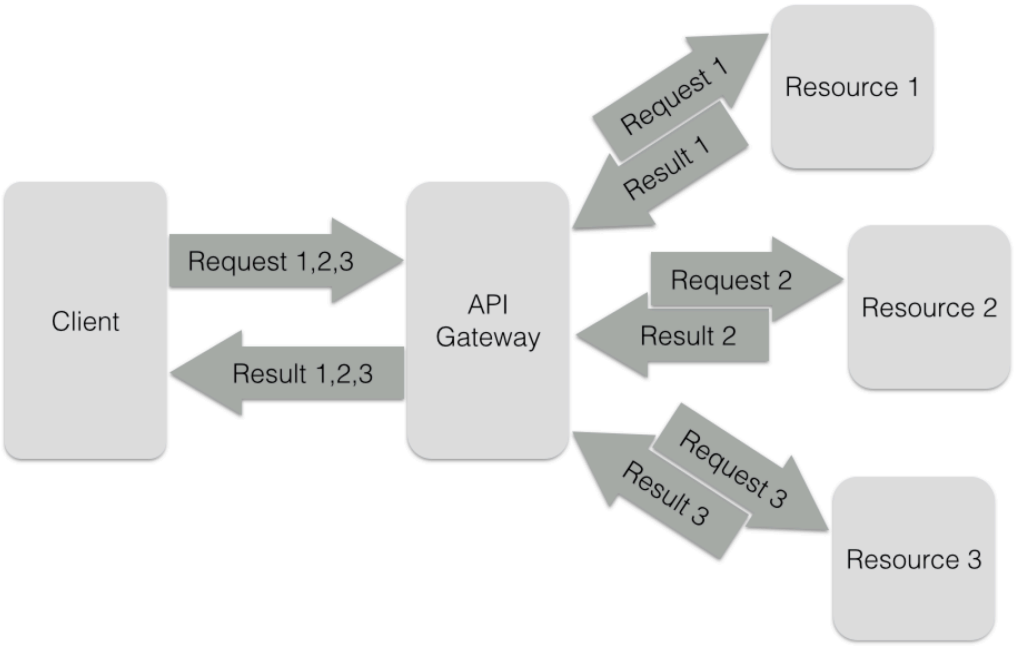

Figure 3.2: The API gateway receives a concatenated request which it distributes to multiple resources, the responses are then concatenated into a single response. The resources can either belong to one or several back-end systems.

When working with HTTP requests, there are multiple penalties for executing many requests compared to one concatenated request. These penalties includes the previously mentioned limit of maximum TCP connections (page 13) and the overhead from HTTP headers (page 9)—but there are also other issues such as the increased battery drain on mobile devices which occur when the wireless radio needs to become active multiple times [19].

We can, by utilising an API gateway to concatenate many HTTP requests, avoid these common problems. Concatenating requests can be seen directly in many modern API designs such as the Facebook Graph API [20]—but for the APIs which lacks this feature, an API gateway can effectively mitigate these problems.

3.4 Duplicate and unnecessary items

When requesting data from a back-end API, the responses may contain unnecessary data which the client do not need. In a similar fashion, if a client performs several similar requests, it is possible that all the responses contains some amount of duplicate data. By utilising an API gateway, the results from the back-end API can be modified to remove the items which the different clients do not need.

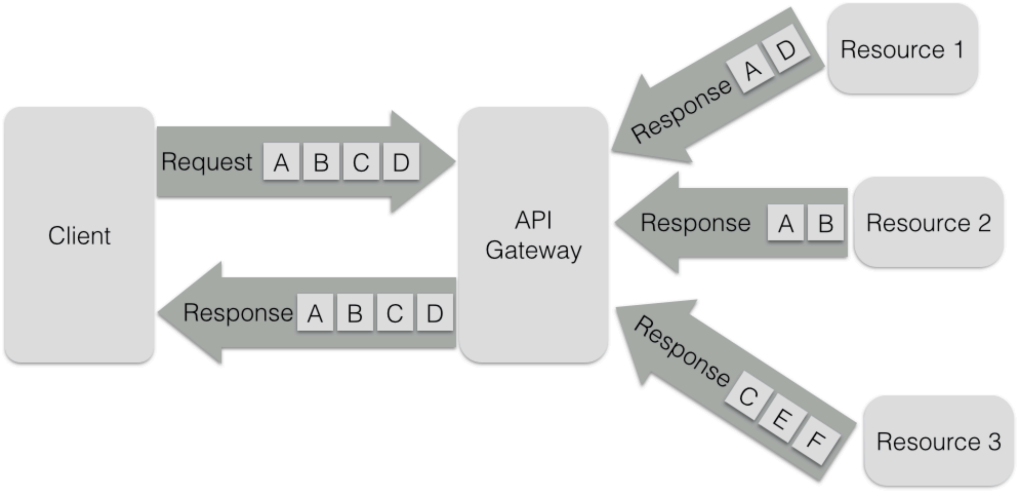

Figure 3.3: The client requests the items A, B, C, D. The API gateway fetches A, D from Resource 1—item A from the Resource 2's response can then be discarded since it is duplicate data. Item E, F from Resource 3 can be discarded since they are not wanted by the client at all. The API gateway can after retrieval respond with just the requested items A, B, C, D.

3.5 Format transformation

When working with older legacy systems, the data can be formatted in a way which is not suitable for modern clients. When looking at clients written in JavaScript, many browsers and developers prefer to work with JSON rather than XML since the translation between JavaScript objects to JSON is a 1:1 mapping—more about this on page 48. API gateways can convert the request and response data to a format more appropriate for the requesting client or the responding back-end.

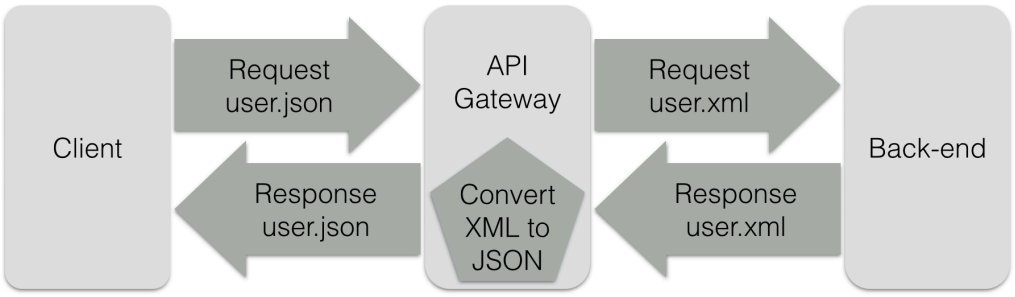

Figure 3.4: The client requests "user" in JSON-format. The API gateway fetches "user" in XML-format from the back-end, converts it to JSON and responds to the client with it.

The approach of using an API Gateway has the additional benefit that the conversion code does not have to be rewritten in every client. Rewriting the same conversion code,

potentially in a new language or by using a different library, for each client increases the risk of introducing bugs. The reason for this is that different libraries works in different ways even though they solve the same problem—especially when there is no standardised mapping between two formats. Bugs are also introduced as the size of the code base grows such as when the same task has to be rewritten several times [21, page 521].

By performing the transformations in the gateway, the processing work is moved away from the clients which can improve its performance as well as reducing the code size and its complexity.

3.6 Pure REST and HATEOAS

REST, Representational State Transfer, consists of guidelines and best practices for creating scalable web services. The style was developed by the W3C Technical Architecture Group (TAG) in parallel with HTTP/1.1. RESTful systems often communicate over HTTP using so called HTTP verbs such as GET, POST, PUT and DELETE to send and retrieve data between clients and servers.

If an API follows the strict rules of REST, it must utilise the concept of HATEOAS, Hypermedia as the Engine of Application State, which is a constraint in the REST architecture. Instead of defining and explicitly sharing a collection of end-points which the client can call, it requires the client to discover the resources itself by first performing a GET HTTP request to the APIs root URL. The back-end will respond with all the resources available from the root such as "users". The client then has to query the "user root" to discover which requests can be made in regards to the user resource—and so forth.

By forcing the client to discover all resources, the client developer has to do a lot of demanding work in the implementation phase [22, page 62]. This approach also introduces a lot of HTTP requests which increases the network traffic significantly.

API gateways can be utilised to transform a "Pure REST API" with HATEOAS to a simpler API which only follows some of the restrictions put in place by the REST architectural principles. This can significantly lower the amount of traffic between the client and the back-end, which can be a big performance gain, especially in cases such as when there is a high latency between the client and the back-end—assuming that the latency between the API gateway and the back-end is low such as when they are placed inside the same LAN[1].

[1] Local Area Network, see appendix.

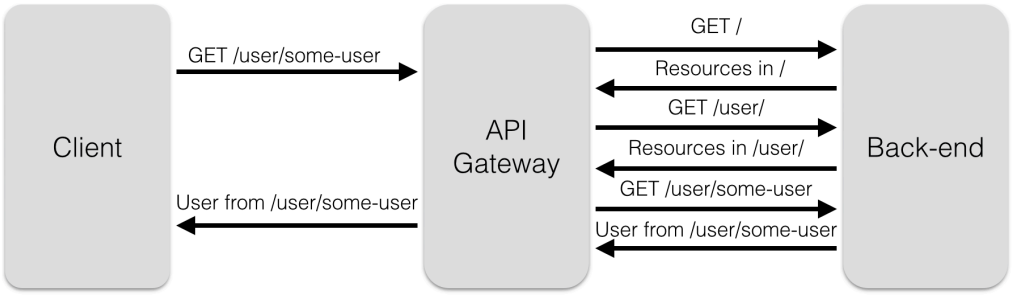

Figure 3.5: The API gateway performs the pure REST HATEOAS communication. At the same, the API gateway exposes a simpler end-point which the clients can call.

3.7 Compression

API gateways can be utilised to compress responses in the cases where no compression is present on the back-end API servers. This can significantly reduce the amount of traffic the client has to receive which increases the performance, especially on mobile devices with low bandwidth. HTTP compression was explored on page 24 where it was noted that Gzip has an expected compression level of 60-80% on text-based media.

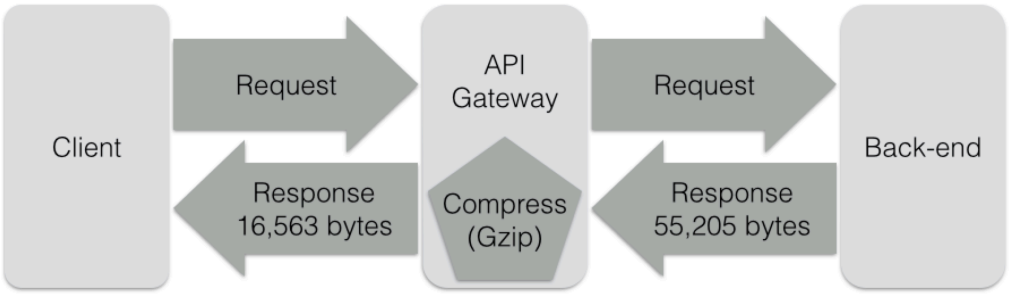

Figure 3.6: The API gateway compresses the response from the back-end API by utilising the Gzip algorithm. This reduces the response traffic in the client by 70%. Numbers taken from the example on page 24.

3.8 Caching

Responses from frequent API calls can be cached using the API gateway in order to reduce the load on the back-end system [22, page 107]. The cache can have a specified lifetime or be invalidated based on certain events. There are several different caching strategies and many popular third-party systems which the API gateway can utilise—caching is a vast and complex topic in itself and is therefore not explored in further detail here.

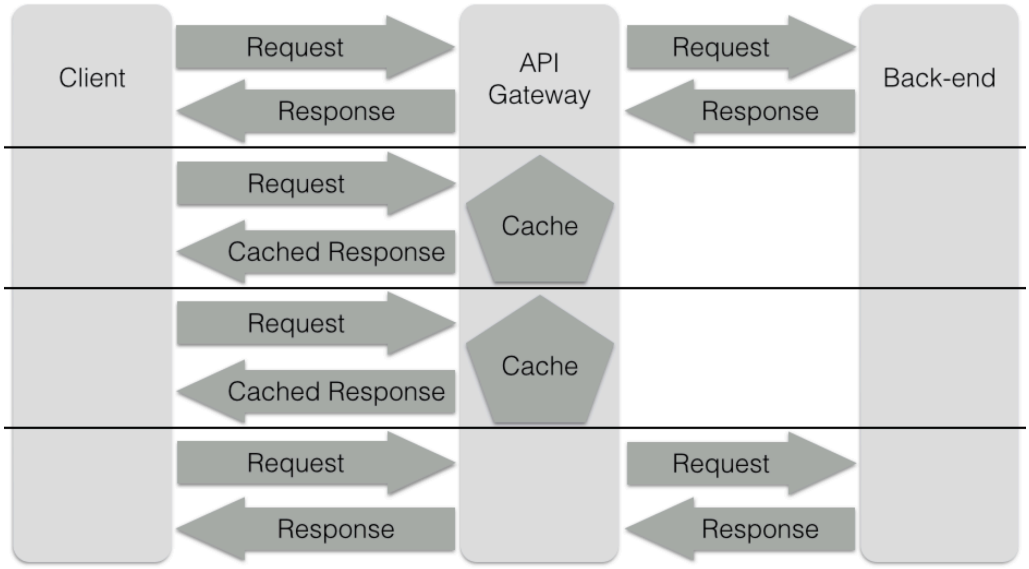

Figure 3.7: Frequent API calls to the same end-point can be cached in the API gateway to reduce the load on the back-end servers.

3.9 Decreasing bandwidth and cost

Cloud providers, such as Amazon [23] and Microsoft [24], do not charge for used bandwidth as long as data is transfered between servers in the same cloud regions. When utilising an API gateway in the cloud, bandwidth and its costs, to and from the client, can be reduced by placing the API gateway in the same cloud region as the back-end servers and apply bandwidth saving techniques such as the previously in this chapter mentioned: compression, duplicate & unnecessary items, pure REST and in some cases even format transformation.

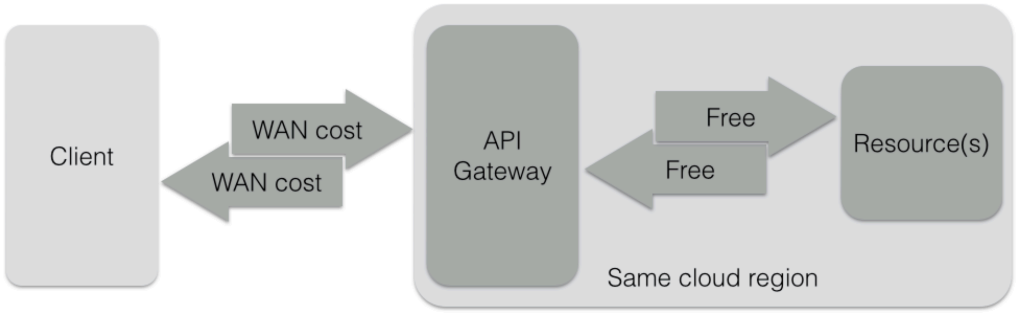

Figure 3.8: Cloud providers such as Amazon [23] and Microsoft [24] charges based on whether the traffic is over WAN or in the same cloud region.

3.10 Secure point of entry for private networks

Corporations usually uses several internal services with APIs that are protected inside a private network. A VPN[2] can be utilised to give clients on the outside access to services inside the private network. A VPN can however have the undesired side effect of exposing too much of the private network to the external client machines.

Another approach to solve this is to place an API gateway inside the DMZ[3] of the private network. By doing so, external clients can access the API gateway as a single point of entry for all internal APIs. The API gateway can be configured to only expose a predefined collection of the internal APIs and proxy them to the appropriate external clients.

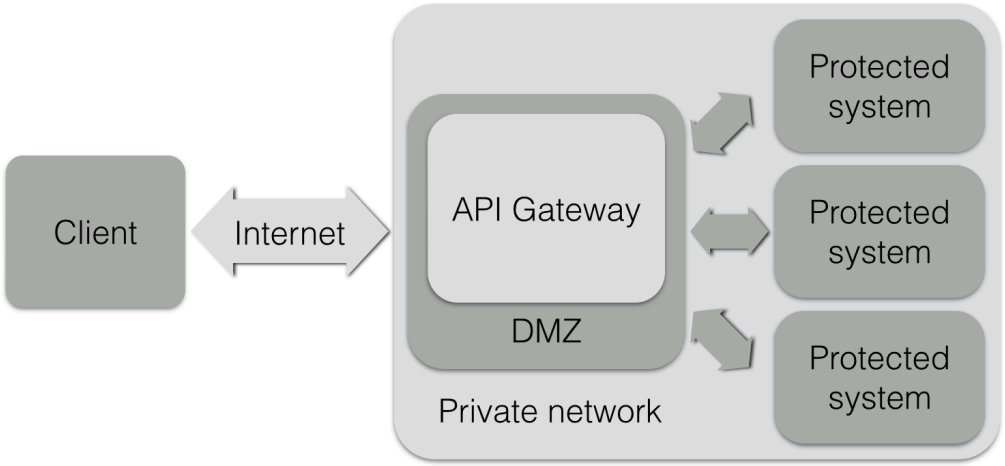

Figure 3.9: An API gateway used as a secure way of exposing internal services in a private network to the outside world.

3.11 Latency

One important goal of an API gateway is to reduce, or at least not significantly increase, the latency experienced in the communication between the client and the back-end server. Because of this, the placement of the API gateway from a network point of view is very important. (In all of the following scenarios, we treat LAN latency as negligible which should be a fair assumption.)

The first approach we look at is placing the gateway on the same LAN as the client. Placing the API gateway on the same machine as the client is rarely possible or practical—it complicates updating the gateway and defeats much its purpose of introducing a new layer between the client and server.

Placing the API gateway inside the same LAN as the client can be a good solution, for example when it is used inside a corporation's private network. The constraint with this approach is that no outside clients, such as smartphones not connected to the internal

[2]Virtual Private Network, see appendix.
[3]DeMilitarized Zone, see appendix.

network, will be able to avoid the extra latency introduced over WAN[4]—or may not be able to connect to it at all based on the LAN security. This is however an approach which does not introduce double latency, but it does not decrease it either.

The second approach is to place the API gateway as a separate application in its own cloud or on a LAN separated from the back-end and client. While this may be the only solution for certain hosting setups, this introduces the problem of double latency. Since the TCP-packets has to go through two WAN connections, both of them can introduce a substantial amount of latency which can worsen the response times.

Finally, the third approach is to place the API gateway on the same LAN as the back-end system. This is in many cases the best approach as it avoids the problem regarding double latency while it at the same time provides access for external clients and introduces flexibility in regards to updates.

The problem with double latency can however arise, and be unavoidable, if the API gateway is communicating with several back-end systems which are placed on different LANs. In such a scenario, several factors have to be considered before deciding which LAN to place the gateway in. Such factors include which back-end API has the most traffic, bandwidth costs between LANs, the latency between the different LANs and so forth.

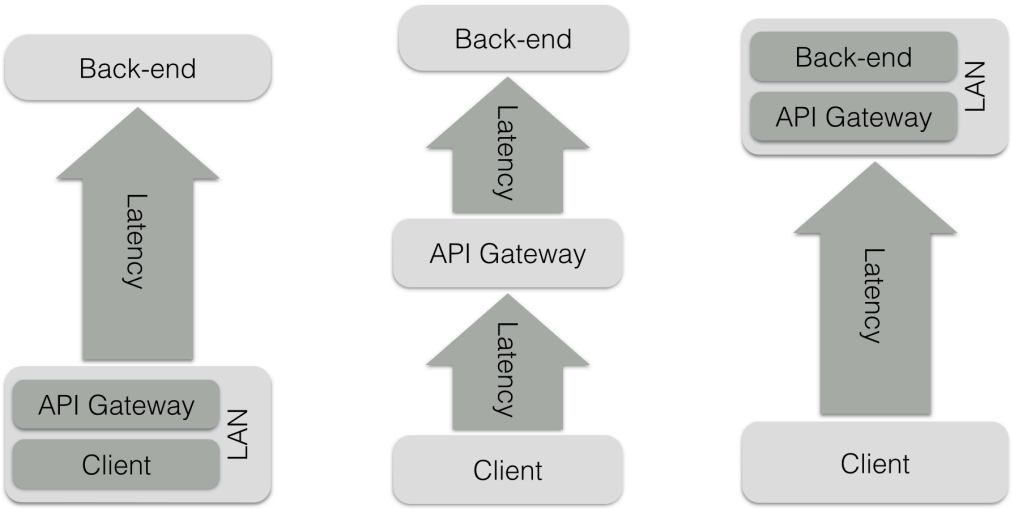

Figure 3.10: How latency affects the different placement strategies for the API gateway.

3.12 Error handling

An API gateway should be able to handle potential errors with different strategies. In the simple scenarios where a single request is proxied and potentially transformed, the API gateway can chose to either resubmit the failing request to the back-end API a number of times, potentially after a small delay, or to simply relay the error to the client.

[4]Wide Area Network, see appendix.

Deciding what to do in more complex cases where several requests are concatenated or transformed together is much harder. The developer of the API gateway's end-point has to decide if a partial result is relevant for the client or if one failure should invalidate the entire combined result.

Deciding to invalidate the entire result based on one request failure is problematic if the API gateway uses chunked responses. Ideally, the API gateway want to transmit data to the client as soon as it is available but since chunks can not be retracted, the API gateway either has to wait for all back-end results to arrive before responding or introduce some kind of an error chunk which tells the client to discard the previously sent data.

The API gateway developer has to decide whether to handle much of the error complexity in the API gateway itself or delegate this responsibility to the client. These factors has to be considered on a case to case basis—there is no correct answer.

3.13 Security—authentication & authorisation

API gateway security is, like all security scenarios, a very complex problem. All but the very simplest of cases should be solved outside the implementation of the gateway itself. What makes an API gateway complex from a security point of view is the fact that an end-point exposed from the API gateway can communicate with several back-end systems, all of which can utilise different authentication and authorisation protocols. Because of this, a single sign-on service provided outside the API gateway itself is a good approach for the more complex setups which communicates with several back-end systems.

Any further in-depth discussion regarding this topic is outside the scope of this thesis and has therefore been excluded intentionally.

3.14 Conditional back-ends

By utilising an API gateway, several different back-ends can be exposed as one single end-point. If we, for example, wanted to provide an API with weather reports from Sweden and Denmark but we have noticed that two different back-end APIs provide better reports for each country—one is better for Denmark and one is better for Sweden. With an API gateway, we can translate the incoming API-calls to the format required by the different back-ends and delegate the call based on certain inputs such as from where the API-call is made.

3.15 Rate limiting

API gateways can mitigate traffic spikes on back-end services by implementing a rate limit for API-calls. This can usually be done in many different ways as seen in Azure [25] and Apigee [26]. Rate limiting strategies include a global rate limit, a per client rate limit or a per token rate limit. This functionality is often used from a business perspective where a certain number of calls are free but a fee has to be paid for subsequent calls.

3.16 Support from old API versions

It happens that API developers make changes which break backward compatibility when moving on to newer improved versions of the API. Fields can be added, renamed or removed. In such scenarios, old clients may be forced to update in order to work with these breaking API changes.

Instead of rewriting many of the already released clients to fit the new API-version, an API gateway can, in some cases, be used to translate the new API format back to the old one. How feasible this is depends on what kind of changes that have been introduced and whether they are destructive or not.

3.17 Analytics

API gateways are in a perfect position to collect data that can be used for analytics. This is because the API gateway is able to monitor all the traffic between the clients and the back-ends.

API gateways can collect a lot of analytic data from HTTP requests and responses such as:

- Client technology: the browsers user-agent which is sent with request headers is one way to collect a variety of data. The user-agent normally includes the browsers name and version, rendering engine, computer architecture and operating system.
- Request-response time for both the client and each individual back-end API calls.
- Latency from different back-end APIs.
- Geolocation from the HTML 5 geolocation API [27] or by geolocating the requesting IP address.
- Errors and failure rates for the back-end servers.
- Invalid client requests.
- Traffic peak hours.
- Suspicious client behaviour such password or denial of service attacks.

Since performance is usually a top priority in API gateways, the collected data should preferably be delegated, stored and processed using a third-party analytics engine.

3.18 Load balancing

API gateways can be used as load balancers to distribute workloads across multiple back-end systems. This can be achieved by implementing different scheduling algorithms—either by doing a simple round-robin or by implementing a more complex algorithm which takes additional factors into account such as the back-end systems reported load, response time, geolocation and so forth.

3.19 Similar concepts

Netflix API

Netflix has applied a concept, similar to API gateways, where each client's team develop their own end-points adapted to the client's specific needs.

Optimizing the Netflix API, Ben Christensen, 2013, The Netflix Tech Blog
`http://techblog.netflix.com/2013/01/optimizing-netflix-api.html`

The Netflix API Optimization Story, Jeevak Kasarkod, 2013, InfoQ
`http://www.infoq.com/news/2013/02/netflix-api-optimization`

Managing API Performance, Apigee

Apigee, who works with API tool development, have put together a collection of articles with focus on optimising API performance in common scenarios which are also applicable in API gateways.

Managing API Performance, Apigee
`http://apigee.com/docs/content/managing-api-performance`

Chapter 4
Rackla: API gateway framework

As part of this thesis work, we developed the framework Rackla[1] in order to better understand, and be able to rapidly develop, custom API gateways. Other API gateway technologies, such as Microsoft Azure API Management [28], Apigee Edge [29] and IBM API Management [30], mainly focuses on expanding existing public APIs from a business point of view with focus on monetisation, security and BaaS (backend as a service) using drag-and-drop graphical interfaces.

Rackla's goal is to help developers create their own custom API gateways programatically with a high degree of freedom and a small amount of abstractions which otherwise could limit the use cases.

4.1 Technologies: language and libraries

Rackla was developed using the programming language Elixir which runs on the Erlang VM. It utilises the existing libraries Plug, Hackney and Poison. In this section, we will explain what they are and what purpose they have.

4.1.1 Elixir

Elixir is a functional language designed for building scalable and maintainable applications which run on the Erlang Virtual Machine. The Erlang VM is known for running low-latency, distributed and fault-tolerant systems while also being successfully used in web development [31]. We consider all of these properties important when developing a successful API gateway.

Other important factors for choosing to use the language Elixir in this framework is the pipe operator, the asynchronous behaviour defined by Elixir processes and the functional programming aspect of the language.

[1] `https://github.com/AntonFagerberg/rackla`

4.1.2 The pipe operator

An important concept used in Elixir is the pipe operator: `|>`. The pipe operator takes the result from the expression on the left side of the operator and "pipes" it into the first argument of the right hand side function. People who are accustomed to Unix may see a similarity to the Unix pipe operator: `|`.

As an example, we can take a look at the following nested and hard to read code. The code will take a list of all integers from 1 to 100,000, multiply all the integers with 3, remove all even numbers and finally summarise them:

```
1 Enum.sum(Stream.filter(Stream.map(1..100_000, &(&1 * 3)), odd?))
```

Figure 4.1: Elixir code written without the pipe operator.

The code from the figure above can be rewritten using the pipe operator which results in a more easily read version:

```
1 1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?) |> Enum.sum
```

Figure 4.2: The same code as seen in Figure 4.1 expressed with the pipe operator.

Another benefit of using the pipe-operator is that it makes you reason about the code in a more structured way. When you read it, you might say: "First I have the range of numbers, then I map over it, then I filter them, then I sum them"—which corresponds to how the code is written.

The pipe operator is an important part in how Rackla works as it pipes requests to a response, potentially through transformations along the way. For developers who are not accustomed to Elixir, the minimalistic syntax used inside Rackla can make it look like an easy to read DSL (Domain Specific Language) in which they can expose end-points and pipe requests to the clients without writing a lot of boilerplate code.

4.1.3 Elixir processes

Processes are Elixir's term for utilising the "actor model" as its concurrency model. In Elixir, processes are extremely lightweight (in comparison with operating system processes) which means that it is not uncommon to have thousands of them running simultaneously. Elixir processes run concurrently, isolated from each other and can only communicate with message passing [32].

4.1.4 Plug

Plug is a specification for composable modules in between web applications—but also as connection adapters for different web servers in the Erlang VM [33]. In Rackla, Plug is utilised for exposing end-points to which the client can send requests, and a as a way for the API gateway to send responses back to the clients over the HTTP protocol. An additional benefit of using Plug is that already existing third-party code for handling things like cookie management and cross-origin requests can be reused inside Rackla.

4.1.5 Hackney

Internally, Rackla uses the Erlang HTTP client library Hackney [34] to send HTTP requests to back-end systems. Hackney is only used internally and it is therefore abstracted away from the Rackla framework users in order to simplify the API gateway development and ensure that Hackney can be removed or replaced in future versions if necessary.

4.1.6 Poison

Poison is an Elixir library used to convert JSON strings to native Elixir data structures and vice versa. In Rackla, Poison is used to convert responses to JSON format but it can also be used by the users of Rackla in order to decode JSON responses from the back-end APIs and transform them with the built-in Elixir functions.

4.2 Rackla overview

4.2.1 Pipeline

A pipeline is the result from using the pipe operator to tie the different functions together. The goal, when writing an API gateway using Rackla, is to describe each end-point as a sequence of steps defined as a pipeline. The simplest scenario is to proxy a request within the API gateway—that is to relay a request without modifying it. We can do that by first defining a URL to call, pipe it to the request function and then pipe it to the response function:

```
1 "www.example.com"
2 |> request
3 |> response
```

Figure 4.3: A simple proxy which relays the entire payload untouched.

4.2.2 Monads and Functional programming

A new type, which can be passed between the different functions, has to be defined in order for the pipeline to work—but also for letting us reason about it. This type has to be able to handle any number of requests asynchronously while still enable us to transform the eventual results before they are sent to the client.

With these constraints, a new type was defined with inspiration taken from the concept of monads (a monad is a structure, often used in functional programming, which represents computations as a sequence of steps).

The new `Rackla` type defines what it means to chain operations together using well established concepts commonly found in functional programming such as `map`, `flat_map` and `reduce` with focus on still being asynchronous.

To illustrate this, we can use Rackla to request two images over HTTP, transform the image's binary data by applying a Base64 encoding to them and finally responding to the client with the encoded data:

```
1 ["url-to-image-1", "url-to-image-2"]
2 |> request
3 |> map(&Base.encode64/1)
4 |> response
```

Figure 4.4: Asynchronously transforming two images requested over HTTP to Base64 encoded data.

It is important to point out that it is the new `Rackla` type that allows this pipeline to be fully asynchronous—the order of the response chunks sent to the client will depend on which of the URLs that is responding first and which of the Base64 encodings is performed the fastest. It can very well be the case that one Base64 encoded image has been fully sent to the client while the other image is still being requested in the API gateway.

4.2.3 Function overview

In this section we will go through and explain the most common functions defined in the Rackla framework. These functions enable us to build very flexible and powerful API gateways.

Request

The function `request`, in the simple case, takes one or many URLs as strings and returns a `Rackla` type. The data eventually contained inside the `Rackla` type will correspond to either the results from the HTTP requests or error tuples if any error occur such as DNS lookup failures, non-responding back-end or thrown exceptions.

More advanced HTTP requests can be executed by instead passing one or many `Request` types to the `request` function. By using a `Request` type, we can define which HTTP verb to use such as GET or POST, what timeout limits we want to use and how we want to

handle insecure SSL connections. We can also define custom request headers and attach a request body (payload) to each individual request.

In addition to this, we can in the `request` function define if we want the `Rackla` type to contain just the response body, the HTTP responses payload, or a `Response` type containing, in addition to the response body, the response status code and response headers.

As we noted in section 2.2, the HTTP specification and the browsers introduce a limitation in the number of TCP connections that the client can establish to the back-end API. Rackla, in contrast to this, does conform to this limitation which means that the API gateway designer can chose to establish any number TCP connections to the back-end API.

Just

The function `just` takes any existing type and encapsulates it inside a Rackla type. This function is useful when mixing already available data with eventually available data from HTTP requests.

Response

The function `response` takes a `Rackla` type and converts the underlying types to an HTTP response and sends it to the client. In order to conform to the asynchronous nature of Rackla, we only use chunked responses [14]. We can pass in options in order to customise the response, available options are: `json` which automatically encodes the response as a JSON data structure, `compress` which compresses the response using GZip compression, `sync` which guarantees that the responses are sent in a deterministic order, `status` which defines the response's status code and `headers` which defines the headers that will be sent with the HTTP response.

Map

The function `map` is used in order to transform the contents inside the `Rackla` type from one type to another type. As an example, we saw in Figure 4.4 that we can apply a Base64 encoding to each individual image's data from many HTTP requests, contained inside a single `Rackla` type.

Flat map

The function `flat_map` works like `map` with the exception that the returned type inside its anonymous lambda function has to be a new `Rackla` type. The naive approach is to wrap the returned type using `just`, but a more appropriate usage is to utilise this function when you want to initiate another `request` pipeline inside a transformation.

```
"example-url"
|> request
|> flat_map(fn(response) ->
  "url-get-more-info-#{response}"
  |> request
end)
|> response
```

Figure 4.5: A new `request` pipeline can be constructed inside the `flat_map` function.

In the figure above we make an HTTP request to an end-point which will return some response. By using `flat_map`, we can create one or more additional requests from the received response by creating a new `request` pipeline inside the `flat_map`'s anonymous lambda function. This enables us to define recursive request structures which later boils down to a single response.

Reduce

The function `reduce`, or sometimes called `fold`, is a way to take a collection of values and "reduce" them into to a single value. Rackla provides a `reduce` function which reduces the values contained inside its own `Rackla` type into a single value (still contained inside a `Rackla` type).

```
Enum.reduce([1,2,3], &(&1 + &2))
```

Figure 4.6: Illustration of how the `reduce` function works inside the built in `Enum` module in Elixir—the reduction will result in the value 6 after performing the addition $1+2+3$. The `reduce` defined by Rackla works in the same way but on a `Rackla` type instead of an enumerable type.

Collect

Sometimes we may want to to break out of the asynchronous `Rackla` type and convert it to a native Elixir type. In these cases, we can use the blocking function `collect` which takes a `Rackla` type and returns the containing value—or all values in a list if there are more than one underlying value.

4.2.4 A complete example

To conclude this section, we will look at a complete end-point which uses the Instagram API [6]. We will first present the code below and then explain what happens line by line. This is a complex end-point which uses many of the techniques defined in Rackla.

```
get "/instagram" do
  "<!doctype html><html lang=\"en\"><head></head><body>"
  |> just
  |> response


   "https://api.instagram.com/v1/users/self/feed?count=50&access_token="
   <> conn.query_string
  |> request
  |> flat_map(fn(response_data) ->
    case response_data do
      {:error, reason} ->
        just(reason)

      http_response ->
        case Poison.decode(http_response) do
          {:error, reason} ->
            just(reason)

          {:ok, decoded_data} ->
            decoded_data
            |> Map.get("data")
            |> Enum.map(&(&1["images"]["standard_resolution"]["url"]))
            |> request
            |> map(fn(img_response) ->
              case img_response do
                {:error, reason} ->
                  reason

                img_data ->
                  "<img
   src=\"#{Base.encode64(img_data)}\"
   height=\"150px\" width=\"150px\">"
              end
            end)
        end
    end
  end)
  |> response

  "</body></html>"
  |> just
  |> response
end
```

Figure 4.7: A complete example end-point created with Rackla.

- Line 1: we define our new HTTP end-point which we name `/instagram`. We use the macro `get` defined in `Plug` in order to listen to GET HTTP requests.
- Line 2–4: we create our first pipeline. What we want to do is to first send some HTML tags which will make the response a valid HTML document (normally we would not do this since we are working with API responses, but this serves as a good example). We do this by storing the HTML code in a string, use the function `just` to turn the string into a `Rackla` type and pipe it to the `response` function. Since the pipeline is ended with a `response` function, it is immediately sent to the client before moving on to the subsequent pipelines.
- Line 6: we define the underlying back-end URL which is the end-point that we wish to use. The end-point requires an access token which we pass to it by letting the client supply it as the query string. The URL will return the current user's feed with a lot of data encoded in JSON format.
- Line 7: we pipe our URL string to the `request` function which will immediately return a `Rackla` type while in the background executing the HTTP request.
- Line 8–34: we use `flat_map` with an anonymous lambda function—described below.
- Line 9: we pattern match on the response data from the HTTP request. We know that the response will either be the actual response body from the API endpoint or an error tuple if case an error has occurred since this behaviour is defined in Rackla.
- Line 10–11: if we receive an error, we will transmit the reason why the error occurred to the client. We have to call the `just` function to convert the reason type into a `Rackla` type since it is required in the definition of `flat_map`.
- Line 13: if we have reached this point in the code, then we know that we have gotten a valid HTTP response. We store the valid HTTP response in a variable called `http_response`.
- Line 14: we pass the HTTP response data to the function `Poison.decode` which will decode the response data from a JSON string to an Elixir data structure by using the library Poison [35]. This conversion can also fail if the response is not a valid JSON string, so we will yet again pattern match on the outcome of this operation.
- Line 15–16: we handle the potential failing JSON decoding by transmitting the failure reason to the client.
- Line 18: we have successfully decoded the JSON string to an Elixir data structure. We store the decoded data in a variable called `decoded_data`.
- Line 19–21: we transform the data, from the JSON response, by extracting the URLs which points to images—all other data will be discarded. This will leave us with a list of URLs pointing to images. It is worth pointing out that we are using different types in the different stages of the pipeline—we are actually working with both the built in Elixir types and the `Rackla` type seamlessly.

- Line 22: we pipe the list of URLs pointing to images to the `request` function. It is important to notice what is happening here: we transform the response, from the outer pipeline's request, to be a new request pipeline which execute many requests. Also note that we are not providing any `response` function in the inner pipeline, we will simply use the `response` function from the outer pipeline.
- Line 23: we use `map` to transform the results from our internal pipeline.
- Line 24: once again we pattern match on the response data. Since we have executed HTTP requests once more we may end up with failing DNS lookups and other problems which we want to handle.
- Line 25–26: we handle the potential failing requests as before by transmitting the reason to the client if any error occur.
- Line 28: we know that we have received a valid HTTP response so we store the response in the variable `img_data`. In this case the response will contain binary image data since the URLs pointed to images.
- Line 29: we Base64 encode the image data and wrap it in a HTML image tag which enables it to render directly in the browser when we call our new end-point.
- Line 35: we end up in the `response` function defined in the outer pipeline. This function will asynchronously receive the HTML image tags from our internal pipeline and transmit them directly to the client. It is important to point out the asynchronous behaviour we have defined here—the images will be sent in a nondeterministic order which depends on the response order of the images requested in the internal pipeline.
- Line 37–39: we define a final pipeline which we will use to send the closing HTML tags to the client. We are at this point guaranteed that these tags will be sent after every image has been transmitted since this final pipeline is defined after, and separately from, the previous pipelines—the asynchronous behaviour is only applicable inside the same pipeline.

When we visit our new endpoint `/instagram`, we will see a valid HTML page which will contain images fetched from Instagram. Since we use chunked transfer encoding, we will see each new image as soon as it is sent from our API gateway to the browser. The ordering will be nondeterministic since we will send every image as soon as it is available in the API gateway—we will most likely get a different ordering of the images every time we refresh the page.

It is worth pointing out that even though we are requesting several images, there is only one request and one response sent between the client and the API gateway—even though there are several requests and responses sent between the API gateway and the back-end API.

To conclude, we should point out that we would, most likely, in a real world application only send the image data as Base64 encoded chunks and apply the HTML markup inside of the client's code instead—by doing so we would not have to mix in any HTML markup in our API gateway. If we, in the API gateway, would attach the ordering when we send each chunks, the client could also render each received image in the appropriate position in a grid even though the images arrive out of order.

4.2.5 Asynchronous process overview

To fully understand how the asynchronous behaviour is defined "under the hood" is out of scope for this thesis. However, we can look at a rough sketch over how the Elixir processes communicate asynchronously, as defined by the `Rackla` type. We will use the example seen in Figure 4.7—more specifically, the largest pipeline defined on line 6–35.

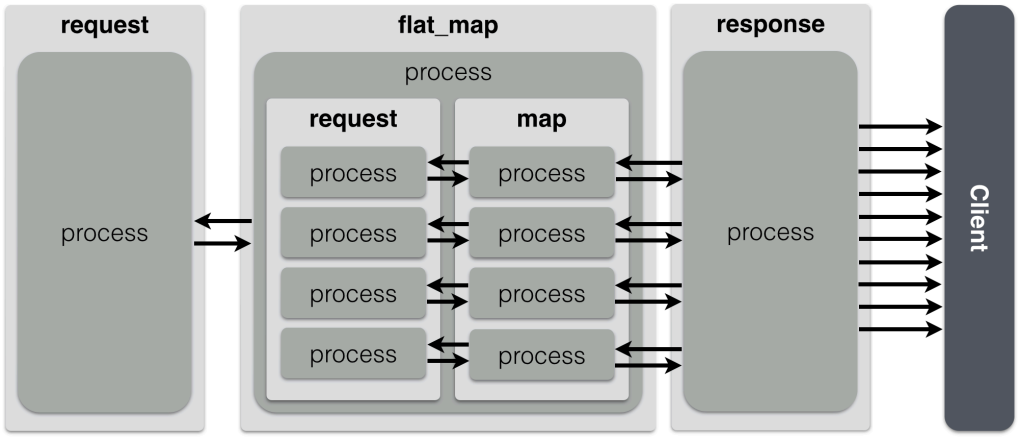

Figure 4.8: Sketched overview of how the processes communicate in the middle pipeline defined in Figure 4.7, line 6–35.

When we call the first `request` function on line 7, it will spawn a new process which will execute the HTTP request to the defined URL. We then we call `flat_map` on line 8 and it will spawn a matching new process which will send a message to the process defined in `request` telling it that it is ready to receive data as soon as it is ready (as soon as the HTTP request is done). When the HTTP request is done, the process in `request` will send the results to the process defined in `flat_map`.

A new pipeline is created inside the `flat_map` function's process—spawning a process inside another process means that the internally defined processes has access to the outer process's data via the scope.

On line 22, we call `request` with a list of URLs. The internal `request` function will spawn a new process for each URL so that each HTTP request will be executed by a unique process. When we call `map` on line 23, it knows how many processes that have been spawned inside `request`, and thus, it will spawn the same amount of new processes to receive the data and transform it.

The `response` function defined in the outer pipeline will spawn just one new process even though it will communicate with an arbitrary amount of processes. This is because its job is to create one HTTP response by consolidating the responses from all processes.

It is important to understand that the arrows drawn in Figure 4.8 are independent of each other and that they are not ordered in any way. We can think of the arrows as open roads where the messages race like cars to the finish line—the processes spawned by the `request` function defined inside `flat_map` are independent of each other, the first responding HTTP request has the opportunity to be sent first to the client.

4.3 Related work

Tyk

Tyk is an open source API gateway written in Go which enables you to control who accesses your API, when they access it and how they access it. It can, like Rackla, transform requests but it uses templates instead of code. This approach can make the end-point development easier with the downside of being less powerful since the templates are not as expressive as a full featured programming language. The focus in Rackla is mainly on transforming the data between the client and back-end API while Tyk's, on the other hand, is more about adding additional functionality around existing APIs.

`https://tyk.io/`

LoopBack-Gateway

LoopBack-Gateway is an experimental, minimum viable product, API gateway developed by StrongLoop. It is written in JavaScript using Node.js and focuses on rate limiting (limiting the amount of calls a client can make), reverse proxying (retrieves resources and then relays them to the client) and security (forcing the client to authenticate itself before making calls to the API). While LoopBack-Gateway in theory could do many of the things Rackla does, it would require the users of the framework to implement many of these features themselves while Rackla provides many helping functions from the starts already optimised for being used concurrently. LoopBack-Gateway, like Tyk, also add functionality around existing APIs such as rate limiting which is something that Rackla does not.

`https://github.com/strongloop/loopback-gateway`

Chapter 5

Case studies

Three systems were evaluated in order to find potential usage areas for API gateways in real-world products. The goal was to look at the three systems from different view points; if two systems have a common problem, then it will only be mentioned in one of the case studies. The reasoning behind this was to highlight the different usage potential for API gateways.

5.1 Streamflow

Streamflow [36] is a system used within municipalities in order to communicate with its citizens and inside organisations to communicate with their customers. It is primarily used to register and track customer cases and works as a central case management hub.

Streamflow exposes a HATEOAS REST API with JSON-encoded responses. The desktop client for Streamflow was written in Java using Swing—however, a new web-client written using AngularJS, a JavaScript Framework developed by Google, is currently under development. The following case study used the in-development AngularJS client with the current production API over HTTP/1.1.

5.1.1 Case lists

In Streamflow, incoming cases are automatically categorised according to rules defined by the municipality or organisation. Each category has two folders: “inbox” and “my cases”. When clicking on the “inbox” or “my cases” for a category, all cases in that folder will be fetched from the server and the results will be displayed in a list.

5.1.2 Evaluation

When viewing the case list, several recursive requests will be executed from the client in order to collect all the required information and in order to follow the HATEOAS specification (the first steps in the request chain has been omitted):

1. Request a list of all cases in the selected category and folder. This will return a list of case-objects which are 0.9 KB per object.

2. For each case in the list, request the case information. This will return the same case-object once more with additional meta data. The reason for executing this request is to discover the next hypermedia resource called "general". The wanted payload, the "general" resource, is 94 bytes while the total response is 3.5–4 KB. This results in a overhead of roughly 97.5% unnecessary data for each request.

3. Request the "general" resource for each case. From this response, the client wants two fields: a date and a note. If a priority is present, the client also wants the next resource called priorities. The total response is 1.5–2 KB and the wanted data is 130 B resulting in a unwanted overhead of roughly 92.5% for each request.

4. If the cases has a defined priority, that priority has to be requested. This response is 293 B and will contain information about all priority levels, usually four levels. A case can only have one priority which results in a 75% overhead for each request.

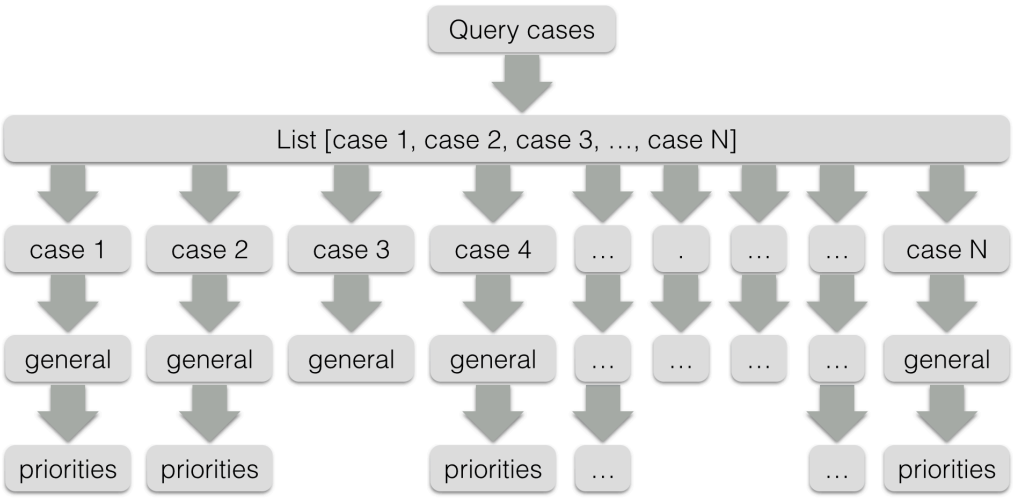

Figure 5.1: How the recursive requests are executed in the Streamflow web client in order to fetch all needed resources and to comply with HATEOAS in the REST architectural specification.

By placing an API gateway developed with Rackla between the Streamflow web-client and the Streamflow API, all recursive requests can be concatenated, for the client, to one request with a single response. By doing so, a lot of unnecessary data can also be discarded before it is sent to the client. This unnecessary data is duplicate data such as the duplicate

case-information, irrelevant data for the client such as HATEOAS discovery information and unneeded data such as unused priorities.

In the test environment, measurements were made on a list which contained 156 cases. For this list, the client had to execute 373 requests: 1 request for the list, 156 requests for each case to get the location of the "general" resource, 156 requests for the "general" resource for each case, and 60 "priority" requests for the cases which needed that information. All these request where replaced with one single request to the API gateway which took care of the HATEOAS communication. By doing so, the total transmitted data was reduced from 1,100 KB to 159 KB—a 86% decrease of transmitted data.

The Streamflow API does not compress any of the responses. By adding Gzip compression to the payload inside the API gateway before responding to the client, the data could be reduced even more, from 1,100 KB to 9.9 KB, which amounts to a 99% decrease of transmitted data.

In addition to this, the client also had to transform certain data types after retrieving them from the server so that it would fit in its internal model. For example, the field "dueOn" was truncated from "2015-02-17T23:59:59.000Z" to "2015-02-17" since the time part was not relevant for the client. By utilising the API gateway, these transformations could be taken care of before replying to the client. This means that no, potentially demanding and error prone, transformations had to be performed on the client—instead the data from the response could be used directly.

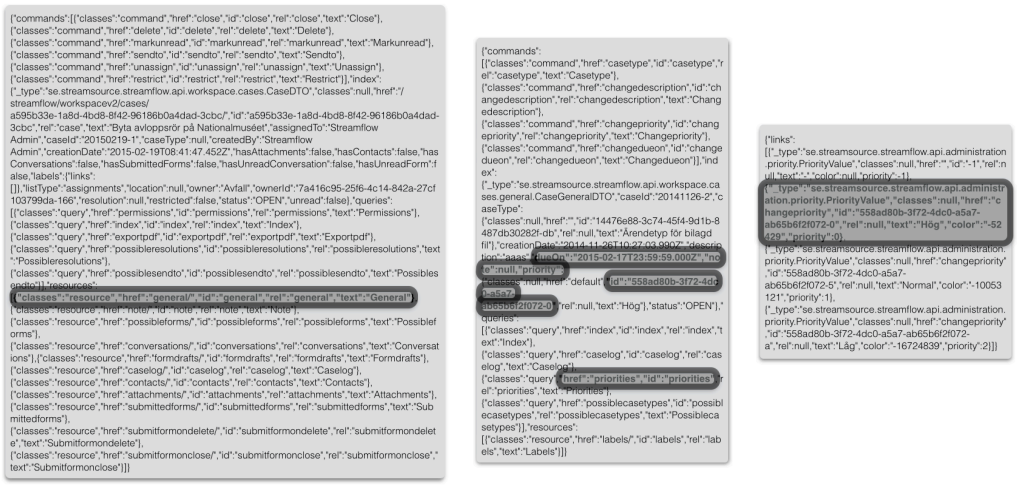

Figure 5.2: Illustration of the responses (case, general and priorities) from Streamflow. The actual data needed in the list view is highlighted in order to illustrate how much unnecessary data that was transmitted.

In the production environment, a measurement was made in order to determine the number of cases present in the municipality of Jönköping at a given time of the day. On average, the number of cases in the non-empty inboxes was 19 and the maximum number of cases in one inbox was 296. This means that on average, the number of requests performed, every time an inbox is checked from the client, is roughly 40–60. When the largest inbox is viewed, the number of requests will be somewhere between 600–900—every time

it is clicked. This is a substantial performance bottleneck for all clients, especially browsers using HTTP/1.1 considering the TCP max-connection limit and the various textual overheads.

It should be noted that the final version of the web-client will most likely be limited to displaying 10–20 cases at a time using pagination. This would reduce the number of requests to 20–60 for any given inbox view. This is however still a substantial amount of HTTP requests to perform every time a user checks an inbox. This approach will neither address the problem that 86% of the transferred data is unnecessary overhead.

5.2 Bank App

The second system evaluated was a banking app for an undisclosed major bank in Sweden, here after simply called "Bank App". Bank App is a mobile app with clients for iOS, Android and Windows Phone. What made the Bank App interesting, from the API gateway point of view, was that it already had a modern and well designed API, which was adapted to the needs of the clients, from the start. Additionally, the clients were written as hybrid apps in which common web technologies could be shared among the different platforms and only a small amount of native components had to be rewritten.

5.2.1 Transaction overview

An essential part of the application is the transaction overview which is where the users can view their balance, orders, fund orders, trades and transactions.

When the transaction overview screen is loaded, the client uses promises (asynchronous computations) to collect and transform the results from the different back-end API endpoints. The results are then bound to the scope variable which in turn makes sure that the information is rendered in the view.

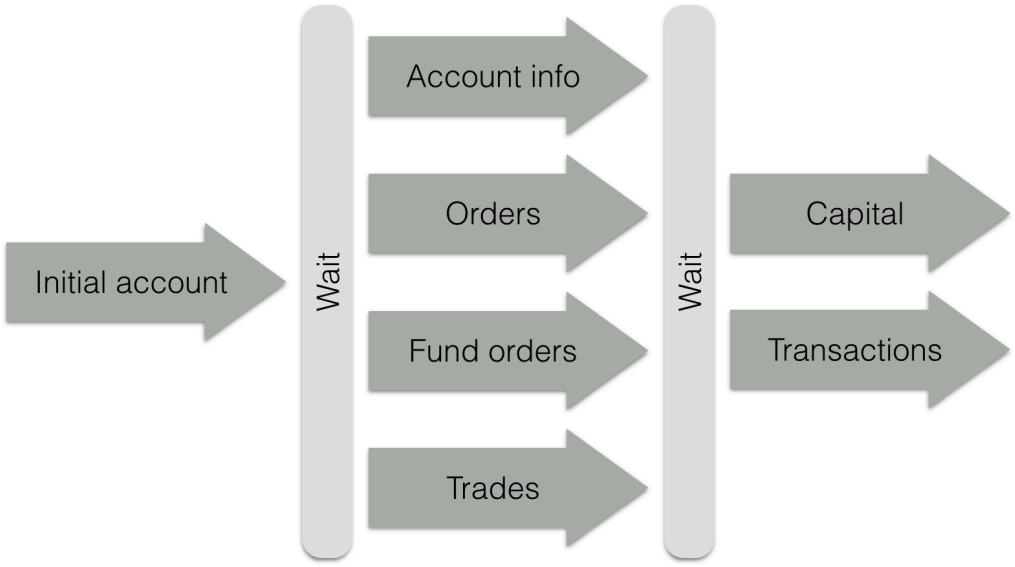

Figure 5.3: How the asynchronous calls to the API and the transformation works. There are two places where the code waits for previous computations before continuing.

To load the required data, the initial account information has to be fetched first. Using that response, the client will then load the account information, orders, fund orders and trades for that account. The client then waits for the four requests to respond, perform some transformations to the data and then merge it with capital and transactions data so that the fetched information fits the clients internal data model.

5.2.2 Evaluation

All in all, the client requests data from seven different API end-points and uses about twice as many functions to fetch and transform these results in the transaction overview. Since the number of parallel requests are less than six in the transaction overview, it will have no problem with the TCP-connection limit.

The data served from the API only had a small amount of information which the client discarded and so the data transfer before and after the introduction of the gateway was almost identical—about 1 KB less after the introduction of the API gateway.

It is possible that the overhead of unnecessary data can differ from customer to customer. One example of this is the fact that the API sends all of the customer's accounts, including inactive accounts, but the client is only interesting in the active accounts. It is however hard to argue that this amount of overhead is so substantial that it will cause performance issues.

What made the Bank App special was that the different mobile platforms shared a common code base for the overview screen. This meant that iOS, Android and Windows Phone could utilise the same JavaScript code for transforming, filtering and sorting the requested data.

If the Bank App instead would have been developed using native code, we would see that the fetching and transforming code had to be rewritten for each platform in a new

language. If we draw a parallel to the industry average defect rate, about 1–25 errors per 1000 lines of code [21, page 521], this means that every new client platform introduced to the transaction overview would potentially create 0.1–2.5 new bugs—a number which could be reduced if the code only had to be written once in the API gateway instead.

Perhaps more importantly, moving the code to the API gateway means that the combined codebase would not just be less error prone but it would also be more maintainable. If we, to take one example, wanted to sort the accounts in descending order rather than ascending, we could change this code once in the API gateway and avoid updating every client.

If this code was instead located in the clients, all of them had to be updated individually, probably by different teams, and each of them submitted to the various app stores for a potentially long review process.

Having different code bases for common tasks in the clients would also increase the risk of introducing discrepancies by mistake despite the goal of having identically working clients on all platforms.

In the end, the code in the client to perform these requests and transform the results amounted to roughly 100 lines of code and the corresponding end-point in the Rackla API gateway amounted to roughly the same line count.

It is hard to argue for the inclusion of an API gateway in the current state of this project based on the facts that the Bank App already has a shared cross-platform code base and a well suited API with mostly optimised end-points. One can imagine that the need for an API gateway can increase over time if the API is not moving as fast enough or is as flexible as desired and therefore can not meet the clients needs. It is also a possibility that new clients will be introduced later on, clients move to native code bases instead of hybrid technologies or that they will fork the existing cross-platform code base. However, in the current state of the project, the inclusion of an API gateway will not provide any substantial benefit.

5.3 Accountant System

Accountant System is a code name for a system used by a large Swedish accountant firm to help them keep track of important documents, tasks and internal priorities. The client is a single-page web application written in AngularJS which uses an existing legacy back-end API. The back-end API communicates entirely with XML encoded messages over HTTP—but the client only works with JSON internally. Working with JSON in web applications can be considered very beneficial since JSON and its syntax is a subset of the JavaScript language. JSON support is also included in the ECMAScript standard, which JavaScript implements, since version 5 [37]. This enables easy (de)serialisation of JavaScript objects to JSON in all modern browsers.

5.3.1 Working with XML in JSON clients

When it comes to converting XML to JSON, and vice versa, there is no standardised approach which can be applied to make the conversions uniform. Even though the formats do solve some of the same problems in regards to data encapsulation, the semantics and fea-

tures are inherently different and it is therefore impossible to create a 1:1 mapping between the two formats.

In an article from XML.com [38] which was published by O'Reilly Media, a conversion algorithm was developed to highlight some of the issues regarding this topic. One of the examples started with a very simple XML structure defined in Figure 5.4.

```
<e>
  <a>some</a>
  <b>textual</b>
  <a>content</a>
</e>
```

Figure 5.4: Simple XML data structure.

An algorithm was developed to convert the XML-structure to JSON notation. When this algorithm was tested, the first naive approach to convert the structure from Figure 5.4 to JSON would result in the following invalid JSON structure as seen in Figure 5.5.

```
"e": {
  "a": "some",
  "b": "textual",
  "a": "content"
}
```

Figure 5.5: The first attempt to transform XML to JSON. The result is an invalid JSON data structure because of the duplicate key "a".

The problem with the JSON structure in Figure 5.5 is that we can not have "a" as the key in two places in an associative array—"a" has to be unique. If we try to solve this by converting the values for the key "a" to a list instead, then we get a syntactical valid JSON structure as seen in Figure 5.6.

```
"e": {
  "a": [ "some", "content" ],
  "b": "textual"
}
```

Figure 5.6: The second approach for transforming XML to JSON. The result is a valid JSON structure but the ordering problem has now been introduced.

However, another problem has been introduced with this approach which is that the element order is no longer perserved. If we would iterate over the values in the XML from Figure 5.4, we would end up with “some, textual, content” but when we iterate over our JSON-structure from Figure 5.6 we would end up with “some, content, textual” which is not the desired result.

Based on this, the following conclusion was made by XML.com:

> “A structured XML element can be converted to a reversible JSON structure, if all subelement names occur exactly once, or subelements with identical names are in sequence. A structured XML element can be converted to an irreversible but semantically equivalent JSON structure, if multiple homonymous subelements occur nonsequentially, and element order doesn’t matter.” [38]

Note that the algorithm from XML.com is just one approach to solve some of the problem around XML-JSON conversion—there are many additional issues which makes this conversion very complex. As with many things where there is no 1:1 mapping, different library developers and corporations are developing their own standards to handle the conversion.

To humorously illustrate this problem, a tool was created [39] which converted JSON to XML with IBM’s JSONx [40] standard and then back to JSON from XML with JsonML’s [41] standard. These tools follow their own defined conversion standards with different syntactical data. When these conversions are performed recursively with each output added to the others input, you would expect that the formats would stay the same, switching back and forth between JSON and XML—instead the data structure will grow indefinitely until the browser crashes since each format adds their own custom syntactical data to it.

In the end—the point is that converting XML to JSON, and vice versa, is troublesome and since there is no 1:1 mapping, it is handled differently in the variety of libraries used today.

5.3.2 Translating XML APIs

In Accountant System, the XML-JSON conversion was handled in two different ways—one way for requests and one for responses.

Response

When working with XML responses from the API, the client utilised a third-party library which converted the XML responses to JSON which then could be used as JavaScript objects in the views, often after some transformations. As in the previous examples, this adds additional complexity to the client which now has to transform the responses before it can handle them properly.

Request

More interesting is how the requests are made to the API. When we look at a typical REST end-point which is accessed over HTTP, we first have a HTTP verb such as GET, POST,

PUT or DELETE which indicates how the underlying resource should be manipulated. In addition to this, we have an end-point which we communicate with via a URL. Lastly, we add a payload which either contains the data we want to submit or additional parameters which can make the request more specific than what can be expressed by the URL itself.

In the case of Accountant System, we can look at the simplest scenario which is saving a note. To do this, the client has to send the XML data seen in Figure 5.7 to the back-end.

```
<SaveNotesRequest>
  <Notes>
    <JPTEXT>This is the actual note.</JPTEXT>
  </Notes>
</SaveNotesRequest>
```

Figure 5.7: XML data used for creating a new note.

This XML data has to be sent with a POST HTTP request to a specific URL with ends with "/note/create". We can reason about what the purpose of the XML is. From the URL we can deduce where the information should be sent and what information we are sending—a note. We can from the HTTP verb POST see that we want to create a new note and in this case, the URL also contains this information since it ends with "/create". The only thing missing to complete this action is the actual payload which is the note content, in this case "This is the actual note.".

When working with this API, the XML is entirely redundant from the clients point of view and it does not add any value, but for historical reasons the back-end API can not change. This puts additional strain and a new layer of complexity in every new client which interacts with the API. Not only do the client have to know about the normal interaction methods such as the URL and HTTP verbs but it also needs to maintain a collection of XML templates and use a different XML template for each type of request it wants to execute. It is also worth pointing out that the the actual note amounted to roughly 1/3 of the total payload data and about 2/3 was structural XML data in the example from Figure 5.7.

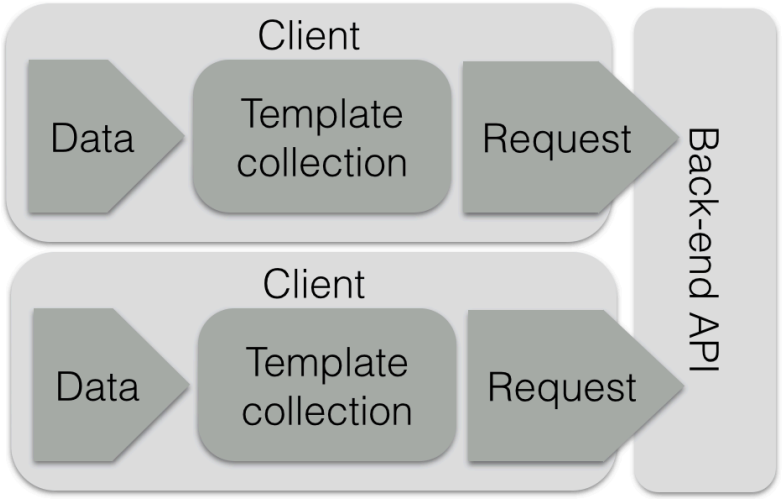

Figure 5.8: Each client has to maintain its own collection of XML templates in order to make requests to the API.

To avoid having to maintain a collection of XML templates in each client, we can introduce an API gateway to do that instead. By introducing an API gateway, we can expose corresponding new end-points which can be used without any XML in the client. The API gateway will maintain the only collection of XML templates which it uses for translating the client's API-requests to back-end API requests. This makes the development of clients a lot easier from the API-call point of view but also has the benefit that there is only one, easily maintainable, collection of XML templates.

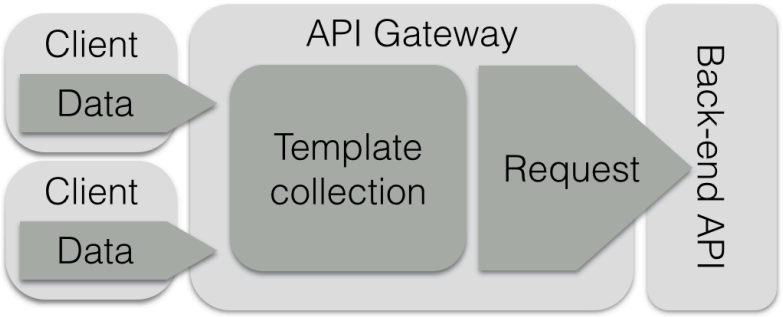

Figure 5.9: The API gateway maintains a collection of XML templates so the client can communicate to the back-end without them.

5.3.3 Evaluation

In the case of Accountant Software, we can see a decrease in used bandwidth when dropping the XML format in the client since XML is a verbose, an in this case unnecessary, format. It is worth pointing out that the XML templates were not only transmitted upon each request but the entire collection of XML templates also had to be downloaded to the browser every time the single-page application was initially loaded.

A study in 2011 compared the XML-based protocol SOAP[1] with FIX (Financial Information Exchange) and CDR (Common Data Representation) in financial messaging systems and concluded that SOAP had 2–4 times larger messages on average [42]. It is however hard to draw a fair parallel with that study to this case study.

The biggest gain in utilising an API gateway in this scenario is likely to be found in developer productivity and code stability. The JSON versus XML is an ongoing debate which has been active for several years. Jeff Barr who is the Chief Evangelist for Amazon Web Services stated back in 2003 that 85% of their API users utilised REST while only 15% wanted the XML-based SOAP interface [43]. The comparison here is not entirely fair either since SOAP is a protocol, not just using XML, while REST is an architectural style which can utilise XML—and that is the case for Accountant Software.

What we can do is to look at the limitations in XML for Accountant Software in particular. The first thing to note is that all modern browsers has built in support for a standardised way of parsing JSON and that there is a natural 1:1 relationship between JavaScript objects encapsulating data and the JSON format. For Accountant Software to work with XML, a third-party library was introduced and all API-responses has to be validated to make sure that the JSON-to-XML parsing works in a decent manner as there are pitfalls to watch out for. For clients written in JavaScript, it would be a more natural approach to use an JSON-based API.

An API gateway can, like the client already does, automatically translate the XML-based responses from the API to JSON. The benefit of placing the translation step in the API gateway is that all clients will have translation done in exactly the same way instead of relying on different local translation libraries.

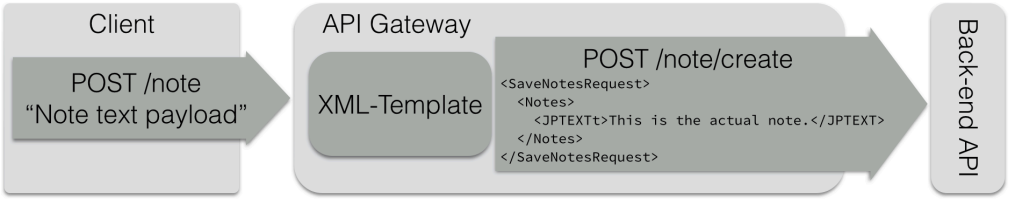

Figure 5.10: The API gateway exposes a simpler REST interface to the client and converts this to the more complex XML-based interface to the back-end.

When looking at the requests the client sends to the back-end API, we earlier concluded that the XML in many cases was unnecessary since all information describing data was already present in the URL in combination with the HTTP verb. By moving all XML-templates to the API gateway, all clients can utilise a much simpler API. An additional benefit to this approach is that all XML-templates are gathered in one place in contrast to the current approach where every client has to keep track of their own collection of templates.

[1] Simple Object Access protocol, see appendix.

Chapter 6
Conclusions

In this thesis we have defined many use cases where API gateways are useful. We did this by examining problems encountered in the industry in combination with solutions developed in concepts similar to API gateways such as the adapters used at Netflix [18].

We hypothesised that API gateways could improve the performance of clients when they are using APIs over HTTP/1.1—especially when the APIs are not customised for the clients. This hypothesis was proven true in the Streamflow case study where we in the API gateway implemented request concatenation, removal of duplicate an unnecessary items and simplified the communication by not conforming to HATEOAS specification. These techniques reduced the total amount of transferred data with 99% when using compressed responses and 86% with uncompressed responses.

In the case study regarding "Bank App", we saw that API gateways are not only useful for improving the performance. Instead, we saw that the clients code complexity could be reduced by moving code away from the clients and into the API gateway. This made the code in the clients simpler and avoided the code duplication which is normally encountered when the same problems has to be solved on different client platforms.

But we also noticed that API gateways are not always useful, as seen in the "Accountant System" case study. The two main factors that we should look for is how well the API conforms to the client's needs as well as the potential limitations introduced by HTTP/1.1—especially the maximum TCP connections and the headers overhead which we described in the first chapter.

HTTP/2 addresses the performance issues discussed in regards to HTTP/1.1 with header compression and a multiplexing connection which removes the fixed TCP connection limit [4]. Despite this, HTTP/2 will not solve all problems such as the amount of duplicate and unnecessary data sent from many APIs or the complicated client implementation required by using the REST constraints HATEOAS. In addition to this, HTTP/2 will neither solve the problems with code duplication encountered in the clients, nor the complexity they must be able to handle, in order to work with API responses that are not suited for their specific needs.

It should also be pointed out that migrating all clients, servers and middle boxes from HTTP/1.1 to HTTP/2 will take at least another decade [4]. This makes the HTTP/1.1 performance enhancing techniques used in API gateways relevant for at least another decade to come and perhaps even longer for legacy systems used in the industry.

The focus of the case studies has been to be able to transform existing APIs in order to suit the client's needs better. However, we have seen that a this concept can be applied in the beginning of the development process. If we know that we will deal with a lot of different clients with different needs, then we can use API gateways in the same way that Netflix optimises its API [18]—their approach is that the back-end API team develops general, unoptimised end-points while the client teams are responsible both for developing the clients as well as develop their own customised end-points using a software layer similar to API gateways.

6.1 Future work

The API gateway concept is very broad and there are many unexplored areas to which the concept can be applied. There are many topics in this thesis which are only briefly touched upon, such security and caching, which could fill an entire thesis of their own in order to be fully explored.

The framework Rackla developed in this thesis used Elixir but there is no doubt that the same functionality could be translated to many other programming languages as well. By migrating the framework to other languages, new challenges and solutions would doubtless appear and shine new light on the topic.

Bibliography

[1] R. Fielding, UC. Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. IETF. RFC: 2616, 1999. https://tools.ietf.org/html/rfc2616.

[2] R. Fielding, UC. Irvine, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. RFC: 2068, 1997. http://tools.ietf.org/html/rfc2068.

[3] M. Belshe, R. Peaon, and M. Ed. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC: 7540, 2015. https://tools.ietf.org/html/rfc7540.

[4] I. Grigorik. HTTP/2: A New Excerpt from High Performance Browser Networking. O'Reilly Media, Inc., 2015.

[5] S. Lorento, P. Saint-Andre, S. Salsano, and G. Wilkins. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. RFC: 6202, 2011. https://tools.ietf.org/html/rfc6202#section-2.2.

[6] Instagram. User Endpoints, 2015. http://instagram.com/developer/endpoints/users/#get_users.

[7] D. Stenberg. curl groks URLs, 2015. http://curl.haxx.se/.

[8] I. Grigorik. High Performance Browser Networking. O'Reilly Media, Inc., 2013.

[9] S. Souders and YUI Team. Performance Research, Part 4: Maximizing Parallel Downloads in the Carpool Lane, 2007. http://yuiblog.com/blog/2007/04/11/performance-research-part-4/.

[10] S. Souders. High Performance Web Sites. O'Reilly Media, Inc., 2007.

[11] S. Lennartz. The Mystery Of CSS Sprites: Techniques, Tools And Tutorials, 2009. http://www.smashingmagazine.com/2009/04/27/the-mystery-of-css-sprites-techniques-tools-and-tutorials/.

[12] S. Tomlinson. Fantastic front-end performance Part 1, 2012. https://hacks.mozilla.org/2012/12/fantastic-front-end-performance-part-1-concatenate-compress-cache-a-node-js-holiday-season-part-4/.

[13] B. Hoffman. Bandwidth, Latency, and the Size of your Pipe, 2011. http://zoompf.com/blog/2011/12/i-dont-care-how-big-yours-is.

[14] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing—Chunked Transfer Coding. RFC: 7230, 2014. http://tools.ietf.org/html/rfc7230#section-4.1.

[15] Z. Mahkovec. Improving Dropbox Performance: Retrieving Thumbnails, 2014. https://tech.dropbox.com/2014/01/retrieving-thumbnails/.

[16] I. Grigorik. Optimizing encoding and transfer size of text-based assets, 2014. https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer.

[17] C. Richardson. Pattern: API Gateway, 2014. http://microservices.io/patterns/apigateway.html.

[18] B. Christensen. Optimizing the Netflix API, 2014. http://techblog.netflix.com/2013/01/optimizing-netflix-api.html.

[19] Inc. Google. Optimizing Downloads for Efficient Network Access—Reduce Connections, 2015. http://developer.android.com/training/efficient-downloads/efficient-network-access.html.

[20] Facebook. Graph API, Making Batch Requests, 2015. https://developers.facebook.com/docs/graph-api/making-multiple-requests.

[21] S. McConnell. Code Complete, 2nd edition. Microsoft Press, 2004.

[22] D. Jacobson, G. Brail, and D. Woods. APIs: A Strategy Guide. O'Reilly Media, Inc., 2012.

[23] Amazon Web Services Inc. Amazon EC2 Pricing, 2015. http://aws.amazon.com/ec2/pricing/.

[24] Microsoft. Microsoft Azure—Data Transfers Pricing Details, 2015. http://azure.microsoft.com/en-us/pricing/details/data-transfers/.

[25] Microsoft. API Management access restriction policies, 2015. https://msdn.microsoft.com/library/azure/dn894078.aspx.

[26] Apigee Corp. Comparing Quota, Spike Arrest, and Concurrent Rate Limit Policies, 2015. http://apigee.com/docs/api-services/content/comparing-quota-spike-arrest-and-concurrent-rate-limit-policies.

[27] A. Popescu. Geolocation API Specification, 2010. http://www.w3.org/TR/geolocation-API/.

[28] Microsoft. Microsoft Azure API Management, 2015. http://azure.microsoft.com/en-us/services/api-management/?b=15-05.

[29] Apigee Corp. Apigee Edge, 2015. http://apigee.com/about/products/apis-and-edge.

[30] IBM. IBM API Management, 2015. http://www-03.ibm.com/software/products/sv/api-management.

[31] Plataformatec. Elixir, 2015. http://elixir-lang.org.

[32] Plataformatec. Elixir—Getting started—Processes, 2015. http://elixir-lang.org/getting-started/processes.html.

[33] Elixir-lang. Plug—A specification and conveniences for composable modules in between web applications, 2015. https://github.com/elixir-lang/plug.

[34] B. Chesneau. Hackney—simple HTTP client in Erlang, 2015. https://github.com/benoitc/hackney.

[35] D. Torres. Poison—An incredibly fast, pure Elixir JSON library, 2015. https://github.com/devinus/poison.

[36] Jayway. Streamflow-–with citizen service in focus, 2015. http://www.jayway.com/portfolio/streamflow/.

[37] Standard Ecma. Ecma-262 ecmascript language specification, 2009. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[38] S. Goessner. Converting Between XML and JSON, 2006. http://www.xml.com/lpt/a/1658.

[39] Convert Json to JsonX to JsonML and so on.., 2015. http://orihoch.uumpa.com/jsonxml/.

[40] IBM. JSONx, 2015. http://www-01.ibm.com/support/knowledgecenter/SS9H2Y_6.0.0/com.ibm.dp.xm.doc/json_jsonx.html.

[41] M. S. McKamey. JsonML, 2015. http://www.jsonml.org/.

[42] C. Kohlhoff and R. Steele. Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems, 2003. http://www2003.org/cdrom/papers/alternate/P872/p872-kohlhoff.html.

[43] T. O'Reilly. REST vs. SOAP at Amazon, 2003. http://archive.oreilly.com/lpt/wlg/3005.

Appendices

Appendix A
Definitions

A.1 JSON

JSON, JavaScript Object Notation, is a data-interchange text format based on a subset of the JavaScript Programming Language. It is an open standard format which uses human-readable text. JSON is often used as an alternative to XML.

A.2 XML

XML, Extensible Markup Language, is a markup language used for encoding documents. It can be used as an alternative to JSON for data communication but it is also used in other areas and document formats.

A.3 REST

REST, Representational State Transfer, consists of guidelines and best practices for creating scalable web services. The style was developed by the W3C Technical Architecture Group (TAG) in parallel with HTTP/1.1. RESTful systems often communicate over HTTP using so called HTTP verbs such as GET, POST, PUT and DELETE to send and retrieve data between clients and servers.

A.4 HATEOAS

HATEOAS, Hypermedia as the Engine of Application State, is a constraint in the REST architecture. The clients enter a REST application through a fixed URL and all future actions are discovered dynamically within resource representations sent from the server.

A.5 DMZ

DMZ, DeMilitarised Zone, is an isolated subnet located outside the protected LAN where workstations and internal back-end systems are located. It is common that machines, which have to be directly exposed from the internet, are placed inside the DMZ.

A.6 SOAP

SOAP, Simple Object Access protocol, is an XML-based protocol used for data exchange. SOAP is primarily transported using HTTP but can also be used with other protocol such as the e-mail protocol SMTP.

A.7 Proxy

A proxy server acts as the intermediary between clients and servers by relaying the data between them.

A.8 LAN

LAN, Local Area Network, is a network limited to a smaller area such as a building or an office.

A.9 WAN

WAN, Wide Area Network, is a network consisting of a large region such as a country or many countries. The internet is considered to be a WAN.

A.10 VPN

A VPN, Virtual Private Network, is used to securely extend a private network, such as the network inside a corporation, to an outside public network such as the internet.