# Work in progress title

(A LaTeX class)

Anton Fagerberg
ada10afa@student.lu.se

February 26, 2015

Master's thesis work carried out at Jayway.

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

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

Abstract

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

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

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

Keywords: MSc, template, report, style, structure

# Acknowledgements

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

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

# Contents

# Chapter 1

# Performance issues with HTTP 1.1

Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems[9]. The first standard version of HTTP 1.1 was released in January 1997[18]. The next version, HTTP/2 (originally named HTTP 2.0), is expected to be released in 2015 (ändra här om det publiceras innan ex-jobbet är klart). Although HTTP/2 addresses several of the HTTP 1.1 performance issues, it is reasonable to assume that it will take many years before HTTP/2 fully replaces HTTP 1.1 as the protocol used on all web servers and even longer for many legacy systems and clients. It is therefore relevant to acknowledge and mitigate the performance issues related to HTTP 1.1 even many years after the HTTP/2 release.

## 1.1 Headers

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

As an example, consider the Instagram API[14] which has an end-point where you can get information about a certain user account with a response serialised in JSON format. If a client was built which is supposed to show details about, for example, the 10 specific users. We can then benchmark how making 10 separate API requests would differ, in transmitted data size, from how it would behave if we could fetch all 10 users with one request.

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

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

HTTP requests can be benchmarked using cURL[23]. To make the requests authentic and look like it was made from an actual browser, we tell cURL to use the default headers provided by the browser Firefox. These headers includes among other things browsers the User-Agent, media types which are acceptable responses and so on. A local server running on port 9000 is used to simulate the the Instagram API.

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

Figure 1.2: cURL command used in the benchmarks.

Performing this request will give us the following results:

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

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

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

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

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

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

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

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

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

The data displayed in this example should be viewed as a lower bound. In practise, HTTP cookies which are used for personalisation, analytics and session management are also sent with every HTTP request as headers and can add up to multiple kilobytes of protocol overhead for every single HTTP request[11, page 200].

This is one of the issues that may (ändra "may" om det stämmer när http2 släpps) be mitigated by using HTTP/2 which remembers which headers that has already been sent and therefore doesn't need to retransmit them on subsequent requests[11, page 222].

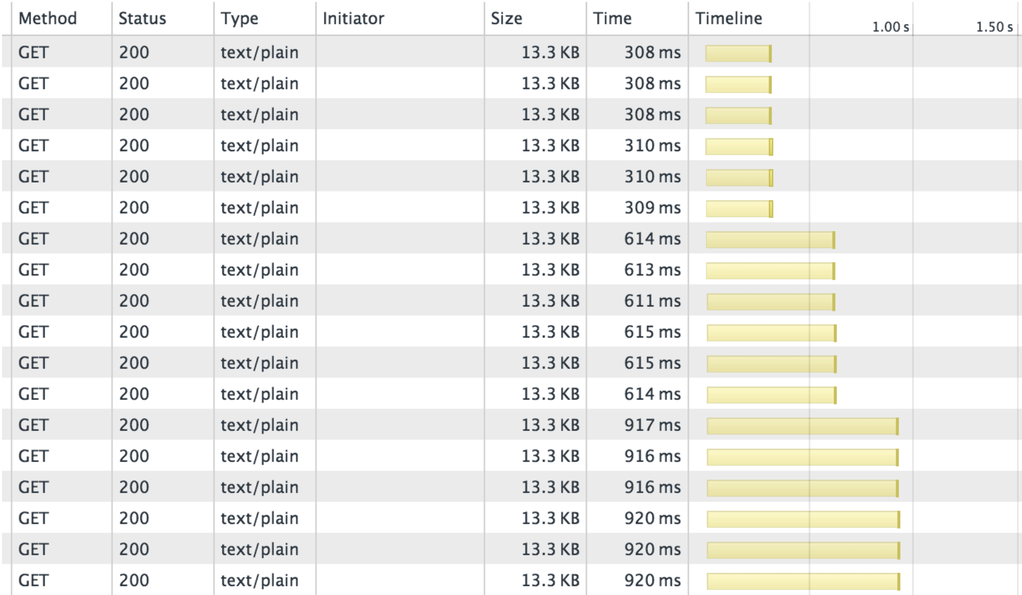
## 1.2 Maximum TCP connections

The HTTP 1.X protocol doesn't allow data to be multiplexed over the same connection[11, p.194]. For this reason, browser vendors has introduced a connection pool of 6 TCP connections per host (the HTTP 1.1 specification limits the pool to 2 connections[8] per host, but modern browsers has refused to conform to this standard in order to decrease the load times).

A common way to deal with the connection limit is to use domain sharding. Since the limit of six TCP connections are on a host name basis, it is possible to create multiple subdomains to avoid this limit. If the subdomains {shard1, shard2, …}.example.com where created and pointed to the same server, more than six TCP connections can be used in parallel at the same time from that machine. This approach is not without downsides as every new host requires a new DNS lookup, a TCP three-Way handshake and a slow start from TCP which can have negative impact on the load times[11, page 199] - just the DNS lookup typically takes 20-120 ms[21, page 63]. Another problem with domain sharding is that the browser always opens six connections per shard even if not all of them are used. In addition, domain sharding can be a complicated manual process and it is hard to calculate how many shards to use for optimal performance. When Yahoo investigated this problem they reached to conclusion that you should, as a rule of thumb, use at least two, but no more than four domain shards[22].

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

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



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

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

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

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

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

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

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

It is worth pointing out that increasing the bandwidth would not resolve this problem as the latency is the only bottleneck in this scenario. It is not uncommon for browsers to wait idle for 100 - 150 ms before spending 5 ms to download an image which means that latency is accounting for 90-95% of the total time for the HTTP requests[13].

### 1.2.1 Chunked responses

When fetching thumbnails, you often want to display them as soon as each individual image has been loaded and not wait for the entire concatenated response. When using a concatenated request to fetch multiple resources at the same time, the server can use chunked transfer encoding in the HTTP response to send the thumbnail data in chunks[7]. By doing so, images can be loaded as soon as they are available in the client, even if loaded out of order[16].

## 1.3 Compression

All requested data should be compressed before it is sent to the client. A common compression algorithm used in HTTP requests is Gzip (GNU Zip) which works best on text-based files such as HTML, CSS and JavaScript. Gzip has an expected compression rate of 60-80% when used on text-based files[11, 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[10].

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

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

An important thing to note about Gzip compression is that only the payload is compressed in HTTP 1.1[24]. This means that the headers including cookies are not compressed which would've otherwise been an additional performance gain. This is one of the improvements which have been addressed in HTTP/2[11, page 222].

# Chapter 2

# API Gateways in theory

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

## 2.1 What is an API Gateway?

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

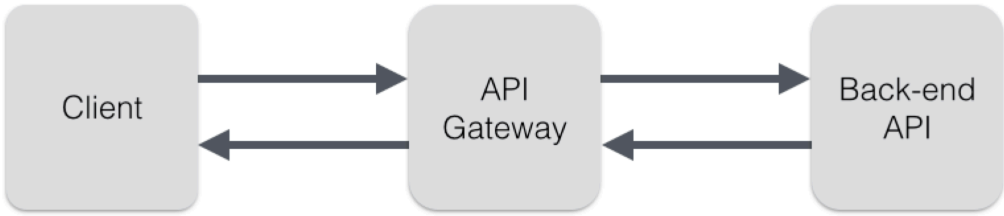



Figure 2.1: A simple scenario using an API gateway between a client and a back-end API.

## 2.2 Different client needs

Nowadays it is very common for different types of clients to consume the same API. One can imagine an image service with a API that returns a list of the latest uploaded images. Depending on what type of client is requesting this data, the API gateway can regulate the number of returned images. If the requesting device is a small mobile phone, the API gateway can chose to return a lot fewer images than if the requesting device was connected to a large wall monitor.

A similar approach has been implemented at Netflix where each client development team write their own adaptor code at the back-end to fully optimise the underlying API for that client needs, much like in the same fashion as an API-gateway works[2].

## 2.3 Multiple resources & requests

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

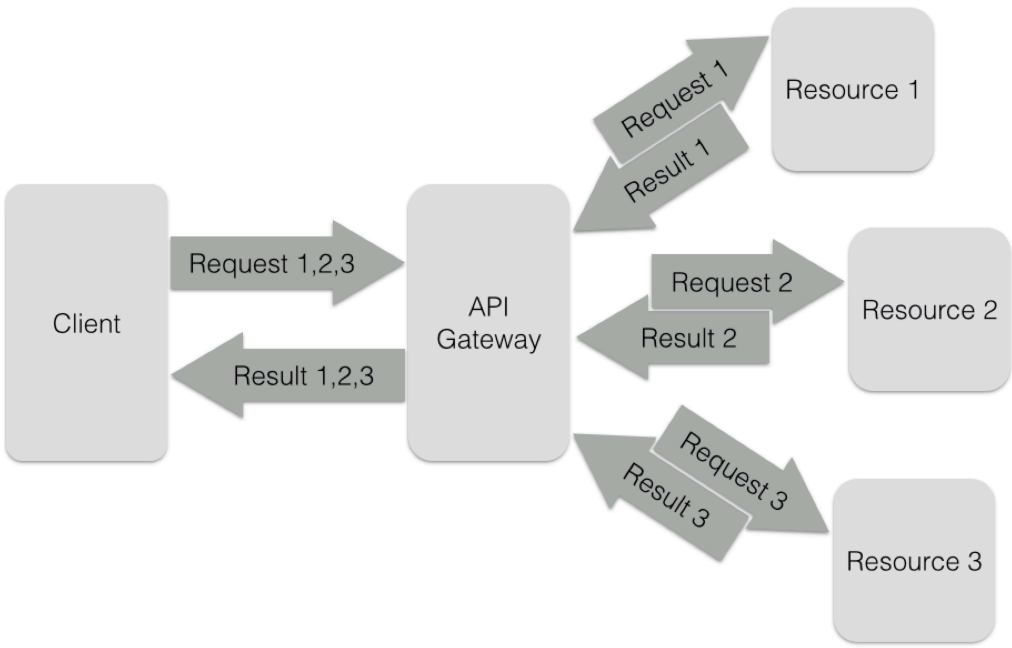



Figure 2.2: An API gateway receives a concatenated request which it distributes to multiple resources, the responses are then concatenated into a single response. The resources can either be one or several back-end APIs.

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

## 2.4 Duplicate & Unnecessary items

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

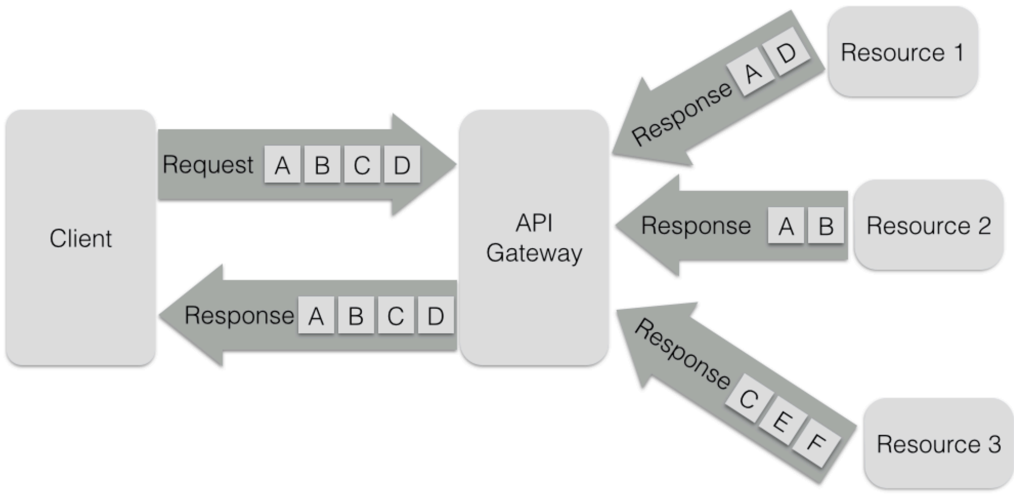



Figure 2.3: Client requests item A, B, C, D. The API gateway fetches A, D from Resource 1. Item A from the Resource 2 response can be discarded since it's duplicate data. Item E, F from Resource 3 can be discarded since they are unnecessary. The API gateway can after retrieval respond with just A, B,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

## 2.5 Format transformation

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

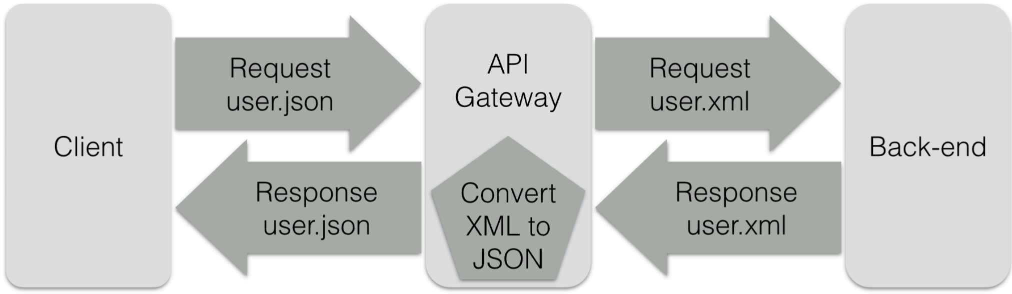



Figure 2.4: The client requests ”user” in JSON-format. The API gateway fetches ”user” in XML-format from the back-end, converts it to JSON and responds to the client in the appropriate format.

This has the benefit that the conversion code doesn’t has to be rewritten for different types of clients. By performing the transformation in the gateway, the processing load is moved away from the client which can improve its performance.

## 2.6 Pure REST

If an API follows the strict rules of REST, it uses the concept of Hypermedia as the Engine of Application State (HATEOAS). Instead of defining a bunch of end-points which the client can utilise, it requires the client to discover the resources itself by first performing a GET request to the root. To root will respond with the resources available, such as users. The client then has to query the users root to discover which requests can be made in regards to users - and so on. Working in this pure way places a very high bar for the client developer[15, page 61].

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

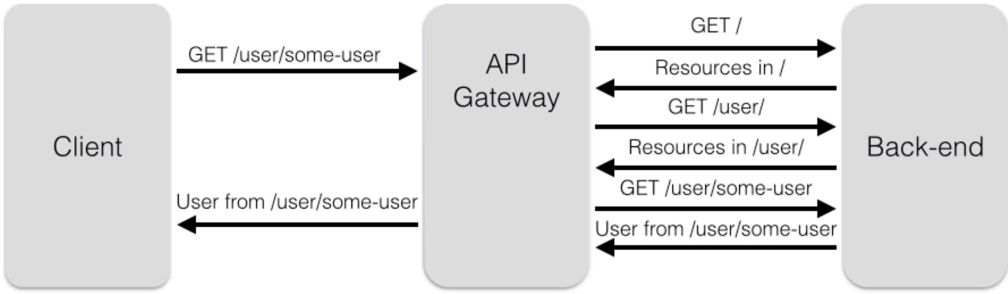



Figure 2.5: Scenario where the API gateway performs the pure REST HATEOAS-communication and at the same time exposes a simple end-point for the client.

## 2.7 Compression

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

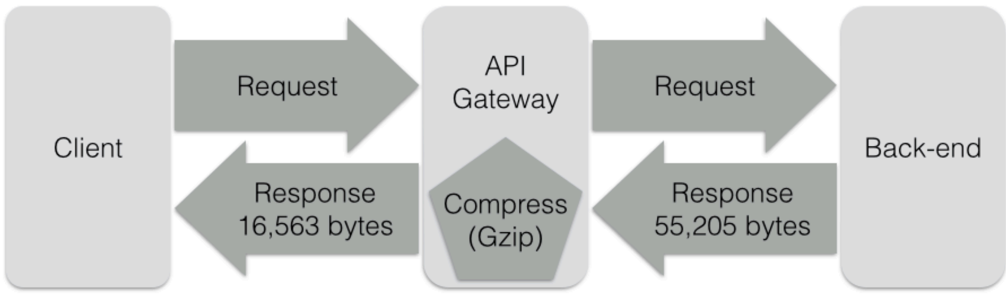



Figure 2.6: The API gateway compresses the response from the back-end API with Gzip which reduces the response traffic in the client by 70%. Numbers taken from the example on page 13.

## 2.8 Caching

Responses from frequent API calls can be cached in the API gateway to reduce the load on the back-end system[15, page 107]. The cache can have a specified lifetime or be invalidated based on certain events such as a table update on a database.

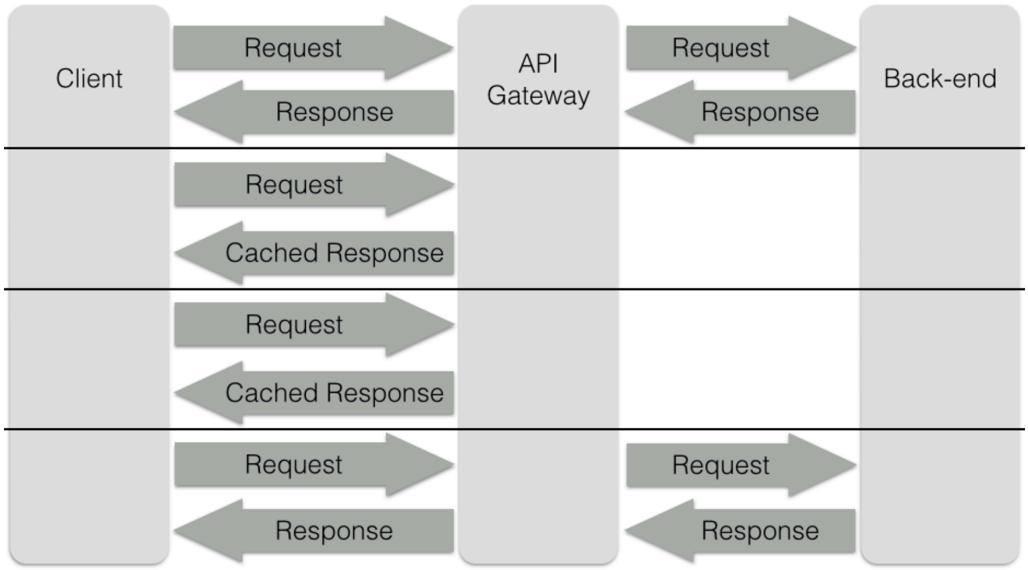



Figure 2.7: Frequent API calls to the same end-point can be cached to reduce the load on the back-end.

## 2.9 Decreasing bandwidth & cost

Cloud providers, such as Amazon[4] and Microsoft[17], does not charge for bandwidth as long as data is transfered between servers in the same regions. When using an API-gateway in the cloud, bandwidth & its costs can be reduced by placing the API-gateway in the same cloud region 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.

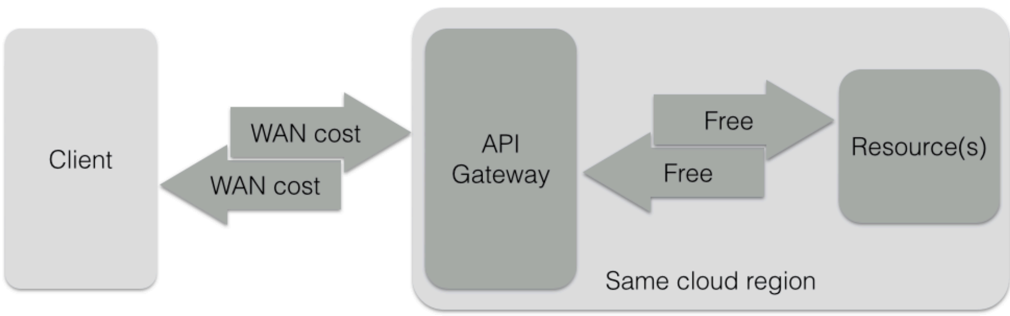



Figure 2.8: How cloud providers such as Amazon[4] and Microsoft[17] charges based if the traffic is over WAN or in the same cloud region.

## 2.10 Secure point of entry in private networks

It is not uncommon for corporations to have several internal services with APIs which are protected within a private network. A VPN, virtual private network, can be used to give outside clients access to services inside the private network but can have the side effect of exposing to much of the private network.

Another approach is to place an API-gateway inside the DMZ of the private network. By doing so, outside clients can access the API-gateway as a single point of entry for all internal APIs. The API-gateway can be configured to expose some of the internal APIs and proxy them to outside clients.

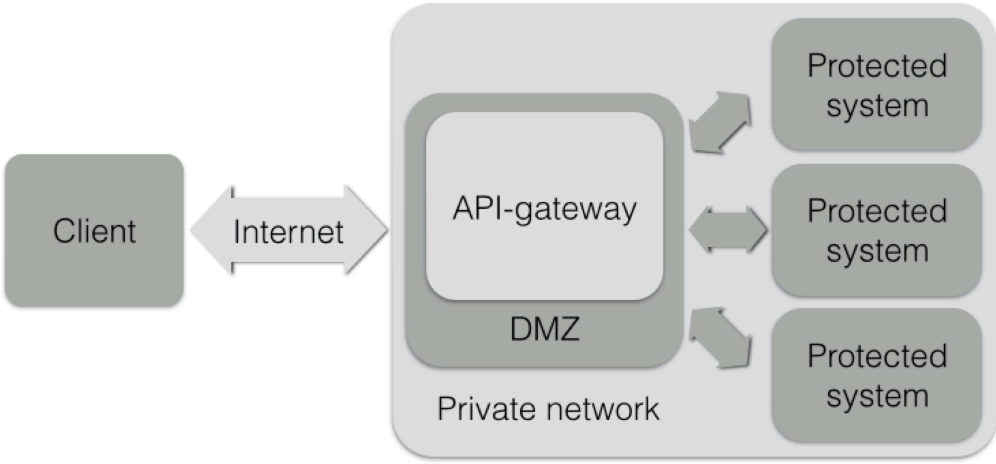



Figure 2.9: An API-gateway used as a secure way of exposing internal services in a private network to the outside world.

## 2.11 Latency

One of the goals of an API-gateway is to reduce latency in the communication between the client and the back-end. Because of this, the placement of the API-gateway, from a network point of view, is very important.

The first approach is to place the gateway on the same LAN or together with 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.

Placing the API-gateway inside the same LAN as the client can be a good solution, for example when used inside corporation. The constraint with this approach is that no outside clients, such as smartphones not connected to the LAN, will be able to avoid the extra latency - or may not be able to connect at all based on the LAN security. However, this is an approach which doesn't introduce double latency.

The second approach is to place the API-gateway as a separate application in its own cloud or server hall. While this can potentially be the only solution for certain hosting setups, this introduce 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 time instead of improving it.

The final approach placing the API-gateway on the same LAN as the back-end system. This is in many cases the best approach as it avoids the double latency, gives access to outside clients and is flexible with updates. The problem with double latency can however arise if the API-gateway is communicating with several 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 as the which back-end API has the most traffic, bandwidth costs between LANs, the latency between the different LANs and so on.

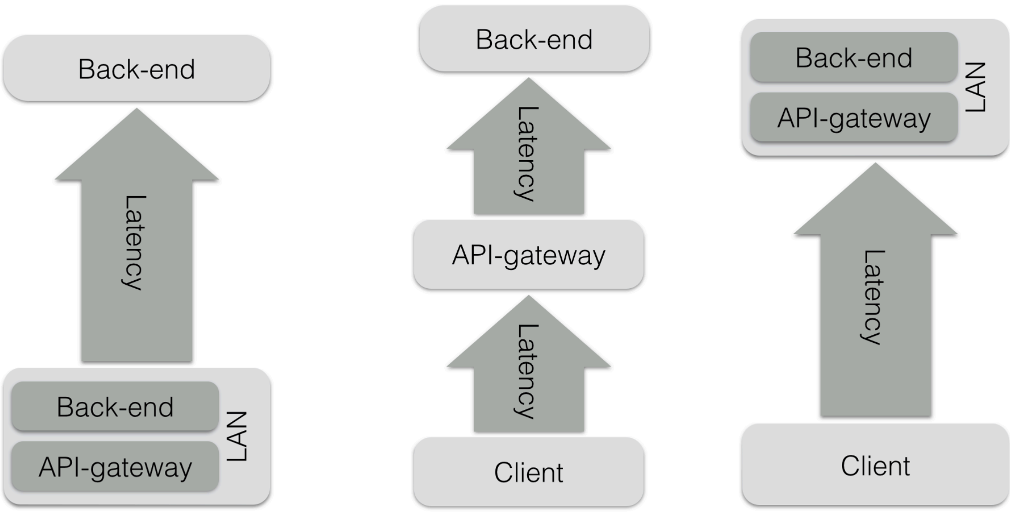



Figure 2.10: How latency affect the different placement strategies for the API-gateway.

## 2.12 Security - authentication & authorisation

API-gateway security is a complex issue which in all but the simplest 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 one end-point provided from the API-gateway can communicate with several back-end systems, all of which may have different authentication and authorisation protocols. Because of this, a single sign-on service outside the API-gateway itself is a good approach for the complex API-gateways which integrate with several back-end systems with their own security requirements.

## 2.13 Support older API versions

It happens that API makes breaking changes when moving on to newer versions. Fields can be added, renamed or removed. In such scenarios, old clients may have to be updated to work with these 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 to the old one.

How feasible this is depends on what kind of changes has been introduced and whether they are destructive or not.

## 2.14 Analytics

API-gateways are in a perfect position to collect data to be used in analytics since the API-gateway is able to monitor all traffic between the clients and the back-ends. It is easy to collect a variety of data such as:

- Client technology, the browsers user-agent which is sent with request headers is one way to collect a variety of data. Example: User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:37.0) Gecko/20100101 Firefox/37.0.
- Request-response time for both the client and individual back-end APIs.
- Latency from different back-end APIs.
- Geolocation from HTML 5 geolocation API[1] or by geolocating the requesting IP address.
- Errors from back-end APIs.
- Invalid client requests.
- Traffic peak hours.
- Suspicious client behaviour such password or denial of service attacks.

Since performance is a big factor in API-gateways, the collected data should preferably be delegated to and stored in a third-party analytics engine.

# Chapter 3

# Crocodile Pear - API-gateway prototype

## 3.1 Language & Libraries

### 3.1.1 Elixir

Elixir is a functional language designed for building scalable and maintainable applications 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[19]. Properties all of which are important for a successful API-gateway implementation.

Two other important influences for choosing Elixir in this prototype are the pipe operator and the asynchronicity achieved by utilising Elixir processes.

### 3.1.2 The pipe operator

One of the integral parts of Elixir is the pipe operator: $|>$. The pipe operator takes the output from the expression 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 the similarity with the Unix pipe operator: $|$.

As an example, we can take a look at the following nested and hard to read code:

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

Figure 3.1: 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..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?) |> Enum.sum
```

Figure 3.2: The same code as in previous figure but written with the pipe operator.

This also makes you reason about the code in a better way. When you read it, you might say something like: ”First I have the range of numbers, then I map over it, then I filter them, then I sum them”.

The pipe operator is also an important part in how Crocodile Pear works, as it pipes requests to a result, potentially through transformations along the way.

### 3.1.3 Elixir processes

Processes are Elixir’s term for using the Actors model as its concurrency model. In Elixir, processes are extremely lightweight (compared to operating system processes) which means that it is not uncommon to have thousands of them running simultaneously. They run concurrently, isolated from each other and can only communicate by message passing[20].

### 3.1.4 Plug

Plug is a specification for composable modules in between web applications - and works as connection adapters for different web servers in the Erlang VM[6]. In Crocodile Pear, Plug is utilised for exposing an end-point for the clients and a way for them to receive responses over HTTP for their requests.

### 3.1.5 HTTPoison

Internally, Crocodile Pear uses HTTPoison[12]. It is based on hackney[5], and it is used for performing HTTP requests. The HTTPoison library is only strictly used internally and it is abstracted away to ensure that it can be removed or replaced in the future. However, it is included as a project dependency and its use in encouraged for developed-defined transformers etc.

## 3.2 High-level concept

### 3.2.1 Data

The data used in Crocodile Pear corresponds to that of an HTTP request. It consists of a numeric HTTP code, such as 200 for OK, headers which is an key-value map and an arbitrary amount of chunks which makes up the actual data payload of the HTTP request/response, in this document called the body - not to be confused with the HTML body-tag, and additional meta-data mainly used within Crocodile Pear.

### 3.2.2 Producer & Consumer

The key component in Crocodile Pear's asynchronous behaviour is the relationship between producers and consumers. Both consumers and producers are fully asynchronous Elixir processes which communicate by passing messages between each other. When a consumer is ready to receive data from a producer, it sends a message to the producer with its identity and a status telling the producer that it is ready to receive.

```
send(producer, { self, :ready })
```

Figure 3.3: The consumer sends a tuple with status "ready" and the identity of itself, using the function self(), to the producer when it is ready to receive data.

The identity of the producer is always known to the consumer via an Elixir PID, process identification, which is passed around in the pipeline (explained below). The producer will only produce a response once and will terminate after a consumer has consumed the response.

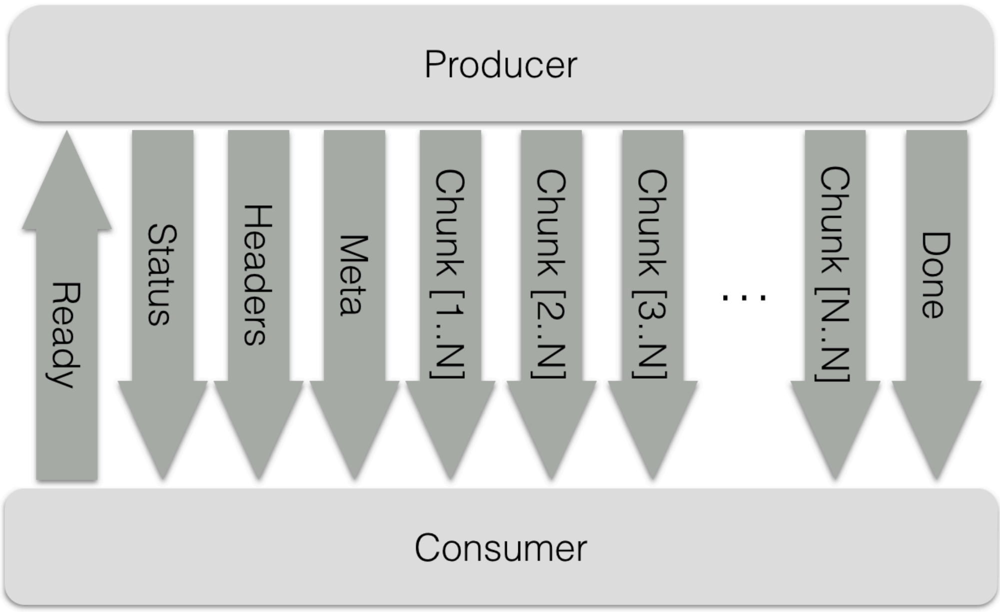



Figure 3.4: Message passing between a producer and a consumer. Time goes from left to right.

After the producer has been notified that a consumer is ready to consume its data, it must first respond with the status, the headers and the meta data. After that, the chunks must be sent in order. Finally, a done message is sent indicating that the producer will terminate and all chunks have been sent.

## 3.3 Pipeline

The pipeline is what ties all pieces in Crocodile Pear together. The pipeline follows a simple rule: every intermediate function should take a list of producer PIDs as its first argument and return a new list of producer PIDs.

The two exceptions are the two functions $request$ and $response$ which is the beginning and the end of a pipeline.

### 3.3.1 Request

The $request$ function in Crocodile Pear takes an arbitrary amount of URLs as strings or a map structure if more advanced settings is needed such as specifying request headers, the HTTP method GET, POST etc. The $request$ will always return a list of producer PIDs. Each producer will produce the result of a HTTP request to the specified URI.

### 3.3.2 Response

The $response$ function takes a list of producers and turns this in to an HTTP response by utilising the provided $conn$ in $Plug$. If more than one producer is passed to the $response$ function, the first responding producer will be consumed first which means that the response order is not deterministic. However, the first responding producer will be consumed until fully completed before any other producer is consumed which guarantees that the chunks will not be mixed together.

As soon as a chunk from a producer has been consumed by the response function, it will respond to the client with the consumed chunked via a chunked HTTP response.

### 3.3.3 Transformers

Transformers is a concept used in Crocodile Pear to manipulate the data when it flows between one, or several, request(s) and a response,

A transformer function takes a lambda function as its parameter. This lambda function pattern matches on the three types of messages: the status, the header-map and the combined chunks, the actual payload denoted body, and applies a function on these parts to manipulate them in some way.

```
blanker = fn(response) ->
  case response do
    {:status, _status}  -> 404
    {:headers, _headers} -> %{}
    {:body, _body} -> ""
  end
end
```

Figure 3.5: The function blanker is a simple lambda function which can be used in a transformer. It sets the status to 404 and removes all headers and the payload by simply ignoring the received values and providing empty ones instead. The actual data in status, headers and body are discarded by using an underscore at the start of the variables in the pattern matching.

### 3.3.4 Timers

Crocodile Pear include timers which can be placed anywhere in the pipeline to benchmark the time it takes to reach the different stages. They can be used anonymously or with labels attached to them and will output data to the default logger.

```
10:41:57.437 [info]  {1424, 252517, 434960} (Got URL)
10:41:58.187 [info]  {1424, 252518, 187494} [headers] (Executed request) on #PID<0.256.0>
10:41:58.187 [info]  {1424, 252518, 187591} [status] (Executed request) on #PID<0.256.0>
10:41:58.187 [info]  {1424, 252518, 187728} [chunk] (Executed request) on #PID<0.256.0>

[...]

10:41:59.529 [info]  {1424, 252519, 529096} [status] (Added transform function) on #PID<0.431.0>
10:41:59.529 [info]  {1424, 252519, 529177} [headers] (Added transform function) on #PID<0.431.0>
10:41:59.543 [info]  {1424, 252519, 543830} [chunk] (Added transform function) on #PID<0.431.0>
10:41:59.544 [info]  {1424, 252519, 543943} [done] (Added transform function) on #PID<0.431.0>
10:41:59.544 [info]  {1424, 252519, 544106} (Responded to query)
```

Figure 3.6: The output from the logger. From left to right: time when message is logged, log level, Erlang timestamp {MegaSecs, Secs, Microsecs}, [message atom] optional message (producer PID if present).

### 3.3.5 Concatenate JSON

The function $concatenate\_json$ takes a list of producers, consumes them and returns a new producer. The new producers chunks will be the responses from the previous consumers, concatenated to a JSON list. Each chunk sent by the new producer will be one item from the JSON list, along with some JSON syntactical data.

Each item in the list will be a JSON-object containing the three keys: status, headers and body (payload). Optionally, if a ”true” value is passed in to the function, each item in the list will be only the ”body” (payload) with the headers and status omitted.

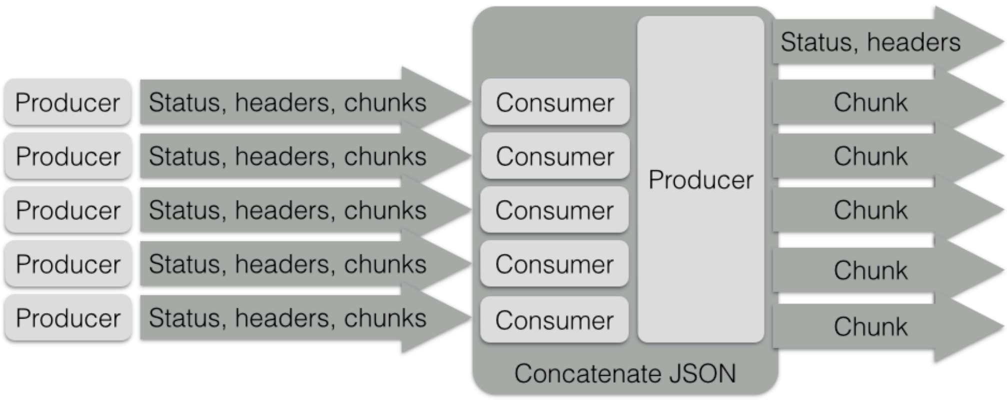



Figure 3.7: Concatenate JSON creates one consumer for each producer. As soon as one consumer has consumed an entire response, the new producer will send that as a chunk in JSON format.

Even though $concatenate\_json$ takes an arbitrary number of producer PIDs as its argument, it will itself only return one new producer as its result is one JSON list object.

## 3.4 Examples

### 3.4.1 Request proxying

A simple proxy is a good introductionary example which demonstrates how the request / response pipeline can be used. In this example, we extract an URI from the query string in the connection and pipe it to the $request$ function which in turn is piped into the $response$ function.

```
conn.query_string
|> request
|> response(conn)
```

Figure 3.8: A simple proxy.

In a similar fashion, we could take an arbitrary number of requests and proxy them into one result. Here we take several URIs from the query string, separated by the character "|" and pipe them in to the $request$ and $response$ as before. By using Elixirs String.split, a list of strings is created which is passed into the $request$ function.

```
String.split(conn.query_string, "|")
|> request
|> response(conn)
```

Figure 3.9: A simple proxy for an arbitrary number of URIs.

This will execute all requests concurrently and start responding, in indeterministic order, based on which of the requests is responding first.

### 3.4.2 Concatenate responses to JSON

The concatenate_json function can be used to concatenate several requests into a JSON list object. It will also set the "Content-Type" header to the appropriate "application/json".

```
String.split(conn.query_string, "|")
|> request
|> concatenate_json
|> response(conn)
```

Figure 3.10: Demonstration of the concatenate_json function.

The response will be one JSON list containing response's status, headers and payload as JSON objects.

### 3.4.3 Transformers

In this more complex example, we will use all previously explained techniques along with a transformer to create a more powerful end-point. We will go over the code in the following figure line by line.

```
   uris =
2    String.split(conn.query_string, "|")
     |> Enum.map(&("http://api.openweathermap.org/data/2.5/weather?q=#{&1}"))
4
   temperature_extractor = fn(item) ->
6    case item do
       { :body, body } ->
8        response_body = Poison.decode!(body)

10       Map.put(%{}, response_body["name"], response_body["main"]["temp"])
         |> Poison.encode!
12
       { _other, other } -> other
14   end
   end
16
   uris
18 |> request
   |> transform(temperature_extractor)
20 |> concatenate_json(true)
   |> response(conn)
```

Figure 3.11: A complex example with a transformer.

On line 1 - 3, we intercept cities and country codes, such as "Lund,SE|Copenhagen,DK", from the query string sent through the connection and then split them in to a list using the separating character: |. This list is then mapped over which will create appropriate URIs for the underlying API which we will use in this example.

On line 5 - 15, we create a new transformer lambda function. When we receive the actual payload data, the body, we will decode it from JSON format to native Elixir data types using the third-party library Poison[25]. We will transform the responses and create a new JSON object by extracting the name and temperature as a new key-value pair and encode it using Poison again - all other payload data is discarded. The other data, status and headers, will be left intact by defining that "other" returns itself, "other".

On line 17 - 21, we tie everything together using the pipeline. We take the URIs, pipe them into a request, pipe the request in to a transformer with our new function, pipe that in to the concatenate_json function and pipe that to the response.

In this example, we have used response concatenation, discarding of duplicate and unnecessary data, and provided a simple way for clients to request temperature data from multiple cities in one requests with a tailor made response.

### 3.4.4 Timers

Timers can be used anywhere in the pipeline to output timed logger information.

```
uris
|> timer("got URIs")
|> request
|> timer("created requests")
|> transform(temperature_extractor)
|> timer("added transformer")
|> concatenate_json(true)
|> timer("concatenated to JSON")
|> response(conn)
|> timer("started responding")
```

Figure 3.12: Using timers in the pipeline.

The logger for this pipeline would produce an output similar to:

```
10:47:49.795 [info] {1424, 252869, 793251} (got URIs)
10:47:49.815 [info] {1424, 252869, 796360} [status] (concatenated to JSON) on #PID<0.311.0>
10:47:49.821 [info] {1424, 252869, 815637} [headers] (concatenated to JSON) on #PID<0.311.0>
10:47:49.894 [info] {1424, 252869, 890575} [headers] (created requests) on #PID<0.288.0>
10:47:49.894 [info] {1424, 252869, 894745} [headers] (created requests) on #PID<0.282.0>
10:47:49.895 [info] {1424, 252869, 891617} [headers] (created requests) on #PID<0.280.0>
10:47:49.895 [info] {1424, 252869, 891539} [headers] (created requests) on #PID<0.284.0>
10:47:49.895 [info] {1424, 252869, 894894} [status] (created requests) on #PID<0.282.0>

[...]

10:47:49.903 [info] {1424, 252869, 903675} [chunk] (concatenated to JSON) on #PID<0.311.0>
10:47:49.903 [info] {1424, 252869, 903826} [done] (concatenated to JSON) on #PID<0.311.0>
10:47:49.904 [info] {1424, 252869, 903948} (started responding)
```

Figure 3.13: Example output from using timers in a pipeline.

### 3.4.5 Simple authentication

While security is mostly out of the scope for this text (see TODO chapter), a small example was created to illustrate how a simple security mechanism can work within Crocodile Pear.

In this example, a username and password is retrieved from the URL - note that this is not secure and this approach should not be used in practice! We validate the credentials using a simple if statement and if the credentials are valid, then we specify a HTTP Basic Auth header with a valid username and password to authenticate with the back-end system.

What this illustrates is that we can have different credentials for the API-gateway and the back-end APIs. Crocodile Pear uses Plug which itself, and with third-party plugins, provides several authentication mechanisms such as sessions, cookies and HTTP basic auth.

```
get "/http-basic-auth/:user/:password" do
  headers =
    if (user == "test_user" and password == "test_password") do
      [authorization: "Basic #{Base.encode64("administrator:administrator")}"]
    else
      []
    end

  %{url: "https://api-url/", headers: headers}
  |> request
  |> response(conn)
end
```

Figure 3.14: Simple authentication example.

# Chapter 4

# Cases studies

## 4.1 Streamflow

Streamflow[3] is a system used in municipalities to communicate with its citizens or organisations with their customers. It is used to register and track customer cases and works as a central case management hub.

Streamflow has a HATEOAS API which responds with JSON. The client for Streamflow is written in Swing using Java, but a new web-client written in AngularJS is currently under development. The following case study has used the in-development AngularJS client 1.1 using the existing production API over HTTP 1.1.

### 4.1.1 Case lists

In Streamflow, incoming cases are categorised by the municipality or organisation. Each category has two folders: ”inbox” and ”my cases”. When clicking ”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.

When requesting the case list, several recursive requests will be executed from the client in order to follow the HATEOAS requirements (the first steps has been excluded):

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

2. For each case in the list, request the case information. This will return the same case-object once more. The reason for executing this request is to discover the next hypermedia resource: general. The wanted payload, the ”general” resource, is 94 bytes while the total response is 3.5 - 4 KB. This results in roughly 97.5% of 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 ”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. Each case only has one priority which results in a 75% overhead for each request.

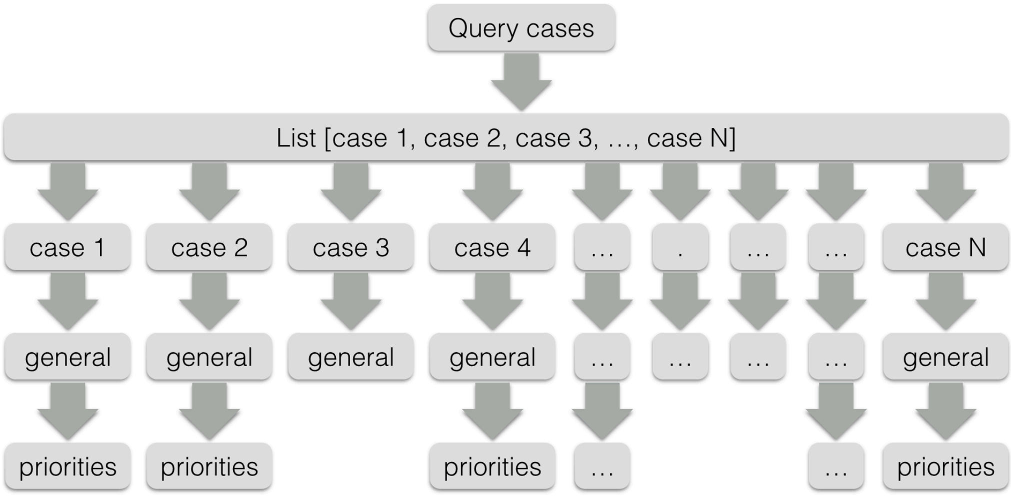



Figure 4.1: How the recursive requests are executed in the Streamflow web-client to fetch all needed resources and comply with HATEOAS.

By placing a API-gateway written with Crocodile Pear in between the Streamflow web-client and the Streamflow API, all recursive requests can be concatenated to one request with one 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 duplicate case-objects, irrelevant data for the client such as HATEOAS discovery information and unneeded data such as unused priorities.

In the test environment, measurements where made on a list which had 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 load time for the client was reduced by 55%, from 11 seconds to 5 seconds. The total transmitted data was reduced from 1,100 KB to 159 KB - a 86% decrease of transmitted data.

The Streamflow API also doesn’t compress the responses. By adding a gzip compression in the API-gateway before responding to the client, the data could be reduced even more, from 1,100 KB to 9.9 KB, which amouts to a 99% decrease of transmitted data!

In addition, the client transformed certain data types after retrieving it 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”. Using the API-gateway, these transformations could be taken care of before replying to the client. This means that no potentially demanding transformations had to be done on the client and instead the data could be used directly on response.

Figure 4.2: Illustration of the responses (case, general and priorities) from Streamflow. The actual data needed in the list view is highlighted.

In production, a measurement was made to determine the number of cases in the municipality of Jönköping at a given time of the day. On average, the number of cases in the non-empty inboxes where 19 and the maximum number of cases in one inbox where 296. This means that on average, the number of requests performed every time an inbox is checked is roughly 40 - 60. When the largest inbox is checked, the number of requests will be in between 600 - 900 every time it is clicked - a substantial bottleneck for browsers on the HTTP 1.1 stack when looking at the TCP max-connection limit and various textual overhead.

It should be noted that the final version of the web-client most likely will be limited to displaying 10 - 20 cases at a time. This would reduce the number of requests to 20 - 60 for any given inbox. This is however still a substantial amount of HTTP requests to perform every time a user check an inbox. This approach will not address the problem of the 86% data overhead which is seen with the current API design.

# Bibliography


[1] Geolocation api specification. http://dev.w3.org/geo/api/spec-source.html.

[2] Optimizing the netflix api. http://techblog.netflix.com/2013/01/optimizing-netflix-api.html.

[3] Streamflow. http://www.jayway.com/portfolio/streamflow/.

[4] Inc. Amazon Web Services. Amazon ec2 pricing. http://aws.amazon.com/ec2/pricing/.

[5] B. Chesneau. hackney - http client library in erlang. https://github.com/benoitc/hackney.

[6] Elixir-lang. Microsoft azure - data transfers pricing details. https://github.com/elixir-lang/plug.

[7] J Fielding, R.; Reschke. Hypertext transfer protocol (http/1.1): Message syntax and routing - chunked transfer coding. http://tools.ietf.org/html/rfc7230#section-4.1.

[8] J.; Mogul J.; Frystyk H.; Masinter L.; Leach P.; Berners-Lee T Fielding, R.; Gettys. Hypertext transfer protocol – http/1.1. http://www.ietf.org/rfc/rfc2616.txt.

[9] James; Mogul Jeffrey C.; Nielsen Henrik Frystyk; Masinter Larry; Leach Paul J.; Berners-Lee Fielding, Roy T.; Gettys. Hypertext transfer protocol – http/1.1. ietf. rfc 2616. https://tools.ietf.org/html/rfc2616.

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

[11] I Grigorik. High-Performance Browser Networking. O'Rilley Media, Inc., 2013.

[12] E. Gurgel. Httpoison. https://github.com/edgurgel/httpoison.

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

[14] instagram.com/developer. User endpoints. http://instagram.com/developer/endpoints/users/#get_users.

[15] G.; Woods D. Jacobson, D.; Brail. APIs: A Strategy Guide. O'Rilley Media, Inc., 2012.

[16] Z Mahkovec. Improving dropbox performance: Retrieving thumbnails. https://tech.dropbox.com/2014/01/retrieving-thumbnails/.

[17] Microsoft. Microsoft azure - data transfers pricing details. http://azure.microsoft.com/en-us/pricing/details/data-transfers/.

[18] R. Fielding; UC Irvine; J. Gettys; J. Mogul; DEC; H. Frystyk; T. Berners-Lee; MIT/LCS. Hypertext transfer protocol – http/1.1.

[19] Plataformatec. Elixir. http://elixir-lang.org.

[20] Plataformatec. Elixir - getting started - processes. http://elixir-lang.org/getting-started/processes.html.

[21] S Souders. High Performance Web Sites. O'Rilley Media, Inc., 2007.

[22] S. Souders and YUI Team. Performance research, part 4: Maximizing parallel downloads in the carpool lane. http://yuiblog.com/blog/2007/04/11/performance-research-part-4/.

[23] D. Stenberg. curl groks urls. http://curl.haxx.se/.

[24] D Stenberg. Http transfer compression. http://daniel.haxx.se/blog/2011/04/18/http-transfer-compression/.

[25] D. Torres. Poison. https://github.com/devinus/poison.

# 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 an alternative to XML.

## A.2 XML

XML, Extensible Markup Language, is a markup language used for decoding documents. It can be used as an alternative to JSON among many other 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 W3C Technical Architecture Group (TAG) in parallel with HTTP 1.1. RESTful systems often communicates over HTTP using HTTP verbs /GET, POST, PUT, DELETE, etc.) to send and retrieve data from servers.

## A.4 HATEOAS

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

## A.5 DMZ