Journal of Information Technology and Computer Science
Volume 8, Number 3, December 2023, pp. 286-298
Journal Homepage: www.jitecs.ub.ac.id

Architectural Design of Representational State Transfer Application Programming Interface with Application-Level Base64-Encoding and Zlib Data Compression

Aryo Pinandito[a], Agi Putra Kharisma[b], and Eriq Muhammad Adams Jonemaro[b]
[a] Information System Department, Faculty of Computer Science,
Universitas Brawijaya, Malang, Indonesia
aryo@ub.ac.id
[b] Media, Game, and Mobile Laboratory,
Informatics Engineering Department,
Faculty of Computer Science,
Universitas Brawijaya Malang, Indonesia
{agi,eriq.adams}@ub.ac.id

Received 18 September 2023; accepted 13 December 2023.

Abstract. Representational State Transfer (REST) is an architectural style that underlies the protocol of HyperText Transfer Protocol (HTTP) and is used by web applications. The implementation of REST principles in web services is often known as RESTful API. The standard communication used in RESTful APIs uses HTTP without involving data compression. There are quite a lot of RESTful API clients in the form of mobile applications that use metered networks. Thus, reducing the required bandwidth during transmission is suggested to be beneficial. The data compression technique is widely known to reduce data size, but this additional step may yield side effects such as increased memory usage and processing time. This study aims to investigate how the data compression process, which specifically uses Zlib and Base64 encoding, may put additional load on the whole process of delivering content to a RESTful API. The performance and characteristics in terms of bandwidth saved when distributing JSON data from a RESTful API in compressed format are also be investigated. According to the experiment result, it is suggested that the compression process can reduce network bandwidth by up to 66% with negligible additional memory usage for the compression and decompression processes.

Keywords: API, Base64, compression, design, REST, Zlib.

1 Introduction

Nowadays, the data that mobile and web applications use is retrieved from the Internet. Almost all people in the world have their own personal handheld devices, which are most of the time used to support their activities in daily life [1]. Now, many computer applications are specifically designed to run on a mobile device with a relatively small screen, e.g., a smartphone or tablet computer, instead of a desktop Personal Computer (PC) to boost the application's user experience and business engagement [2][3].

Client-server architecture becomes one of the most popular approaches to data communications in a mobile application [4][5]. Such an architecture allows the separation of data processing and user interaction into two distinct parts. It allows client applications to be separately developed from server applications. Moreover, the client-server architecture style offers great flexibility in how it is implemented on both the client and server sides [6]. It allows both server and client mobile applications to be built and run on many different platforms and operating systems, e.g., Android, iOS, Linux, Windows, and macOS, to provide data, services, and other functionalities over the Internet.

One popular implementation of client-server architecture that uses Web technology is the Representational Stateful Transfer Application Programming Interface (RESTful API). It is basically an API built using the same technology that delivers web pages and applications. Thus, a web server, which serves web applications or dynamic web pages, could also serve as a RESTful API server [7]. REST is not a specification released by the World Wide Web Consortium (W3C) for data communication on the Internet. Similarly, when delivering web pages on the Internet, REST runs on the existing HyperText Transfer Protocol (HTTP) protocol to provide data and services. Due to its simplicity in implementation, it has been adopted as the de facto standard [5] for developing web Application Programming Interface (API) to deliver and distribute data over the Internet since the early popularity of the Internet.

Most RESTful APIs are focused on providing online services and data. The Twitter API is one example of a successful public RESTful API that benefits many of its users from its data archive with a high success rate [8]. Even though most RESTful APIs provide only data, depending on the use case and how the API is used, the length and amount of data served by the API may vary. This raises potential problems in the handling of large data sets on both the client and server sides. Additionally, larger data consumes more bandwidth and space in memory for the application to handle. Thus, one potential solution to the problem is to apply lossless compression to the data before responding to a request or API call.

This study evaluates the client-server communication of a RESTful API where data compression and decompression processes were applied in the communication process. This study aims to answer and discover the characteristics of data compression for RESTful API performance, as in the following research questions:

1. How are the performance and characteristic of the proposed communication architecture of a RESTful API in terms of bandwidth saved when delivering JSON data in a compressed format?
2. How will the Zlib compression and Base64-encoding processes put additional load on the whole process of delivering content to a RESTful API?

Answers to the research questions would suggest RESTful API and mobile/web application developers determine when to apply compression techniques in their applications to save bandwidth and optimize memory usage in their use case scenarios.

2 Literature Review

2.1 RESTful Application Programming Interface

Representational Stateful Transfer (RESTful) Application Programming Interface (API) is a programming interface that is used in data exchange between different systems. In a RESTful API, data is exchanged through the Internet

with application-layer HyperText Transfer Protocol (HTTP) or HyperText Transfer Protocol over Secure Socket Layer (HTTPS) protocols. JavaScript Object Notation (JSON) and Extensible Markup Language (XML) were two common formats for distributing data over the Internet in RESTful APIs.

Many popular websites on the Internet claim to provide REST web service APIs. However, only a small percentage of the APIs comply with all REST principles [7]. Due to the nature and characteristics of RESTful APIs, many major websites and businesses have moved their APIs from Remote Procedure Call (RPC) or Simple Object Access Protocol (SOAP) to RESTful web services [7][9–12]. Even though more websites and online services provide APIs for their services, most APIs require account registration to start using them. Thus, allow service providers to monitor, control, and manage their APIs [12]. For a complying RESTful API, REST defines six constraints [13], i.e., uniform interface, code on demand, stateless, cacheable, layered system, and client-server. There is no requirement for data compression for a RESTful API to fully comply with REST. This opens the opportunity for API developers to extend their APIs with data compression features to save bandwidth.

2.2 Architectural Design of A RESTful API

The common architectural design of a RESTful API on the Internet is shown in Fig. 1. The client of the API may be a mobile application, a web application, or even IoT devices that are capable of performing HTTP requests. The returned data from the API is either in text or binary format. When the data is in the form of an object or collection of objects, it will most likely be delivered in either JSON or XML format. Because both JSON and XML-formatted data are basically formed in plain text, they can be simply represented by a String data type, so the data becomes consumable by most—if not all—types of programming languages, clients, applications, and devices, while also being easily implemented on various web application frameworks that are available on the Internet. The architectural model of a RESTful API is very general; the communication model has been adopted by many public APIs for distributing data and services over the Internet [7] and several other techniques may also be adopted to improve its performance [14], such as pagination, caching, rate limiting, using HTTP/2, and data compression, which investigated in this study.

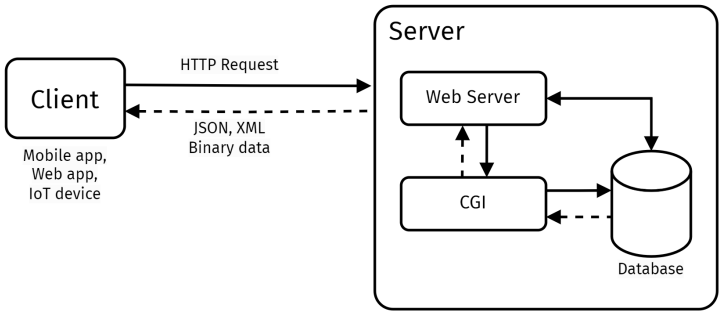


Figure 1. Common architectural style of RESTful API

The nature of a RESTful API uses resource-oriented URIs, where the requested URIs are structured around resources rather than operation requests as in RPC requests. Even though operation-oriented RPC-style URIs work with REST, they

could be easily changed to REST-style URIs that could be handled with four HTTP verbs, i.e., GET, POST, PUT, and DELETE [15]. One HTTP verb that may deliver large amounts of data from an API call is GET. The GET verb is where most clients use it to request data or services. When large amounts of data are requested, the demand for data compression and decompression at both the client and server during the call is necessary to save bandwidth. However, how much bandwidth the data compression process would save in practical use is yet to be investigated. Such research problems will be investigated in this study. Additionally, this study will also investigate how much additional memory and processing load are required to perform such tasks on both the client and server sides.

2.3 Zlib Compression and Base64 Encoding on RESTful API Response Data

There are several lossless compression methods that can be applied to data. Regardless of the data type, one of the most popular lossless compression programs to use is Zlib. Zlib can compress a chunk of data—both in plain-text and binary—and return the compressed form in binary. Therefore, Zlib can be used as a method to compress messages in XML and JSON format in RESTful API communications. Research shows that Zlib can be used to compress XML data in a simple and effective way [16]. The XML compression using Zlib could outperform the existing standard in client-server communication for a RESTful API.

Because Zlib compresses the original data into binary data, this raises a new problem for how to handle the data on the client side. Not all clients and programming languages can handle data streams in binary format. Some are limited to delivering data in the HTTP GET verb, and some are limited to storing and processing data in simple String format. Thus, transformation of the compressed binary data to String-compatible data is suggested for compatibility. Base64 is one popular scheme for binary-to-text conversion on the World Wide Web. Binary data that needs to be handled (stored and transferred) over media that are designed to deal with ASCII characters can be protected during transport. Base64 can be used to embed binary data, e.g., images and other binary multimedia files, directly in HTML, XML, and JSON documents, eliminating the dependency of the document on external files.

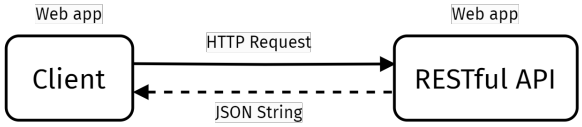
Base64-encoded binary data can be safely decoded into its original form without loss of information [17]. Furthermore, all popular programming languages for use in web applications, e.g., JavaScript, PHP, NodeJS, Python, and Java, support Base64 functionality out of the box [18]. Thus, incorporating compression and decompression processes into a RESTful API becomes an easy task for developers to realize [19]. The additional load on the data compression and encoding processes involved in an API request will be investigated as one of the research questions addressed in this study.

3 Methodology

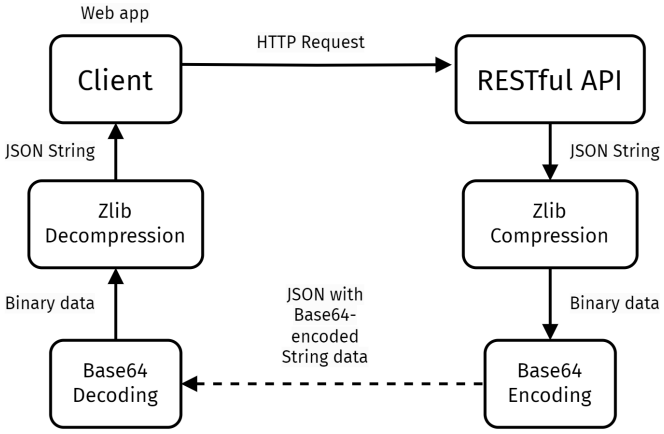
In order to answer the two research questions in this study, a web application that requests data from a custom RESTful API server is implemented as both the client and server prototype. An experiment is conducted and designed to capture the required data for analysis of the study.

3.1 Prototype of RESTful API Architecture with data compression and decompression feature

The RESTful API server is designed and implemented as a web application that runs on a server as a service. Because of the data compression requirement, both client and server applications add additional data compression and decompression processing. The server application encodes the compressed data in Base64 string format as the request response. Zlib compresses data into binary format. To maintain the compatibility level of client-server communication, before transmitting to the client, the data is encoded into a readable plain text format using Base64 encoding. On the client side, a web application decodes the Base64-encoded String data back into compressed binary data, which is then decompressed into a plain JSON String for the standard business logic processing in the application. The proposed architectural design of a RESTful API with data compression implemented in this study is depicted in Fig. 2.



(a) Standard RESTful API communication design



(b) Proposed architectural design with data compression

Figure 2. Common architectureal style design (a) and proposed architectural style design with data compression process (b) of a RESTful API

Most modern web browsers allow the response to a request to be compressed prior to transfer by including the Accept-Encoding HTTP header in the request. However, depending on the capability of the web server receiving the request, the server may or may not compress the corresponding response. The web server may have to be configured to allow compression when responding to gzip requests. In the proposed architecture of this study, the data compression process is laid out at the application level, making it applicable to a wider configuration regardless of the web server's capability to serve HTTP requests in compressed form.

3.2 Data and Implementation Context

The experiment utilizes both client and server application prototypes that implement the proposed architectural design. The client prototype is implemented as a web browser-based application; developed using HyperText Markup Language (HTML), Cascading Style Sheet (CSS), and JavaScript. The HTTP requests to the RESTful API server, which were done by the client, were run by JavaScript. The server prototype is run by a web server using the PHP programming language. The server prototype serves the data in JSON format via an API with a case study of public transportation service.

As for the compression method, this study uses the built-in PHP functions gzdecode() and gzencode() to decode and encode data in JSON String format from and to a compressed format, respectively. On the client side, the JavaScript web application uses Pako [20][21], a JavaScript library that is used to simplify the process of Zlib data compression and decompression of a JavaScript program.

3.3 Experiment Design

The experiment utilizes both client and server application prototypes that implement the proposed architectural design. The experiment was run on a closed client-server network in the laboratory. As shown in the flow of the experiment in Fig. 3, several tests were conducted to measure compressed data length, processing time, and memory usage on 1500 different JSON String sizes as the processing data input for distribution over the network. On the client side, only processing time and memory usage during decompression are measured because the Zlib compression type is lossless. Thus, the length of compressed and decompressed data will always be exactly the same as the length of data measured on the server side.

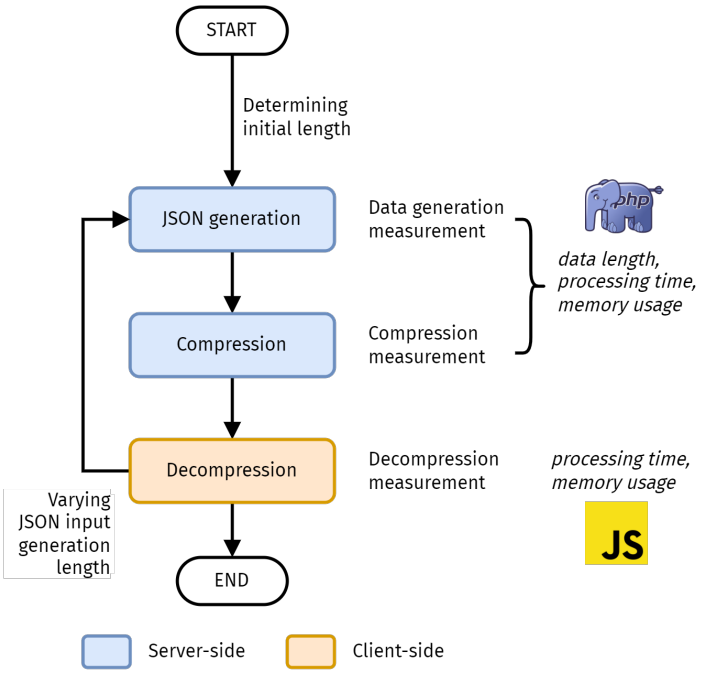


Figure 3. Experiment flow for performance measurement

3.4 Data Measurement Environment

The data measurement and analysis of this study are conducted in an empirical manner, i.e., run and measured based on physical measurement on a computer machine of a particular specification. Because the aim of this study is to evaluate the impact of data compression towards API performance, the impact of network distance and delay in the network communication between client and server to the performance may be disregarded. As described in Fig. 3, the impact of the design in this study is evaluated on three performance measurements, i.e., the length of generated JSON data both in decompressed and compressed form, processing time, and memory usage at the server and client sides. The server runs a Nginx web server that communicates with PHP-FastCGI Processing Manager (PHP-FPM) to process requests on PHP scripts of the RESTful API. The following hardware and software specifications apply to the server machine used during measurement:

- Hardware: 4 Cores Intel Family 15 Model 6 2.4GHz 3768 MB RAM, KVM-based Virtual Machine.
- Software: Nginx 1.18.0 (Ubuntu), PHP 8.2.6, Ubuntu 22.04 operating system.

On the client side, the following hardware and software specifications of a desktop computer running were used for the measurement:

- Hardware: 8 Core Intel Core™ i7-11700 2.5GHz, 32GB DDR4-3600 RAM.
- Software: 64-bit Google Chrome web browser 113.0.5672.127 running on 64-bit Windows 11 Pro 22H2 operating system Build 22621.1702.

Thus, the obtained performance measurement values were measured on the client-server applications that run on the specified hardware and software.

4 Result and Analysis

To answer the first research question regarding additional load on the generation process of compressed data at the server side, memory usage, processing time, and the length of compressed and uncompressed JSON data were m,easured. Table 1 shows a sample of uncompressed and compressed data served by the RESTful API server and their respective lengths. The encoded data shown in Table 1 contains information for three points of location for a transportation navigation system in this study.

4.1 Compressed Data Length

Table 2 describes several samples of data measurements regarding the API's plain JSON String generation time and its uncompressed and compressed sizes. The table also shows the compression rate of the yielded output data size compared to the original data size in JSON String format. The original data was delivered in JSON format without white spaces for visual formatting purposes.

Inferring from the data shown in Table 2 and Fig. 4, it can be said that for very small data sizes, the compression process does not yield smaller data. This happens because the Base64 encoding process encodes the Zlib-compressed binary data into Base64 text format, which yields a longer 64-alphanumeric-character document. Therefore, it is not recommended to compress JSON data of less than about 150 bytes in length. According to the measurement result, most of the time the compression-encoding process could yield data sizes up to 33% of their original size.

Table 1. Sample of Uncompressed and Compressed Data

Data	Output	Length
JSON-encoded uncompressed data	["idpoint":"637", "lat":"-7.9346600068216", "lng":"112.65868753195", "idpoint":"638", "lat":"-7.9345271803044", "lng":"112.65807330608", "idpoint":"639", "lat":"-7.9355871347163", "lng":"112.65798211098"]	202 Byte
Zlib-compressed and Base64-encode data	H4sIAAAAAAAAA2XOSwqDQBCE4bv 02kj19PQrVxEXgUAQgrrILuTuGZeOu4If Pmr60vLct2X90J1MnAZ6P4598zGlmgGw KGxHWF8tMJfRNCxchVPpN5yJ6AgtzgF BrT0BF4EhLkSeCdVwlups0hGe7RojGzH\ /Ac0R7U\/KAAAA	168 Byte

Table 2. Sample of Uncompressed and Compressed Data Length and Compression Rate

JSON Data Length (bytes)	Generation Time (ms)	Compressed Length (bytes)	Compression Rate
68	2.59	112	164.70%
135	2.73	144	106.66%
1003	1.49	424	42.27%
1069	1.70	448	41.9%
12543	2.83	4312	34.37%
12609	2.15	4336	34.38%
32585	5.13	11020	33.81%
32653	4.10	11044	33.82%
72621	4.82	24148	33.25%
72689	4.57	24172	33.25%
72757	5.08	24192	33.25%
100879	7.16	33324	33.03%
100946	7.13	33344	33.03%
101014	6.59	33364	33.02%

4.2 Processing Time and Memory Usage in Data Compression Process

The API retrieves data from a MySQL database server on the same computer machine; hence, localhost. Table 3 shows several measurements of how long the process of JSON data generation and compression took the Central Processing Unit (CPU) time and memory space. According to the data shown in Table 3, it can be said that the time required to process and compress the data increases as the data size increases. Similarly, memory usage also increases linearly with data size. However, in the tests conducted in this study, the time consumed for the compression process increased.

As shown in Fig. 5, compressing larger JSON data took more CPU processing time. Even though the process took milliseconds to complete, it tends to require half the original processing time of the data generation process as the data

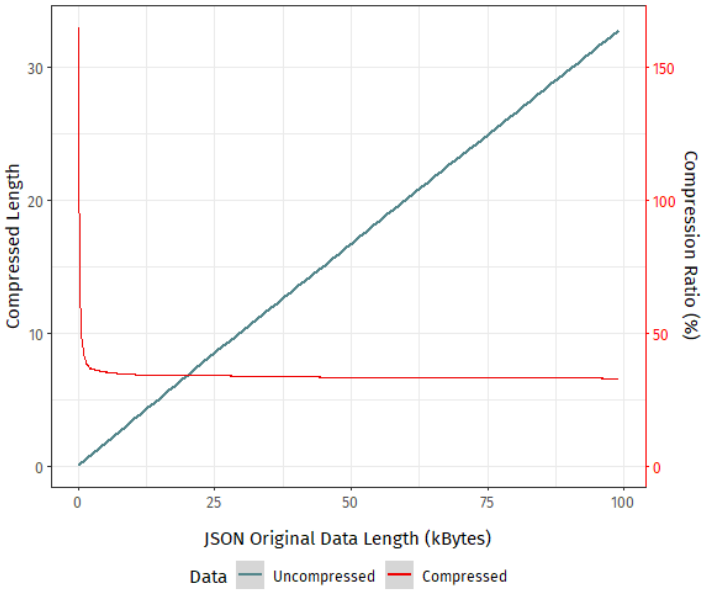


Figure 4. Data compression rate

being processed grows. Even though the data compression process contributes to a small percentage of additional processing time on relatively small data, the Mann-Whitney U analysis of the memory usage measurement data yielded a p-value of 2.2×10^{-6}. The yielded p-value is less than the significance level of 0.05; hence, different [22]. Regarding memory usage, it is expected to increase as the amount of data to be processed grows. As shown in Fig. 6, the compression process requires less additional memory when the data size increases. Thus, it suggested developers to apply data compression to large data.

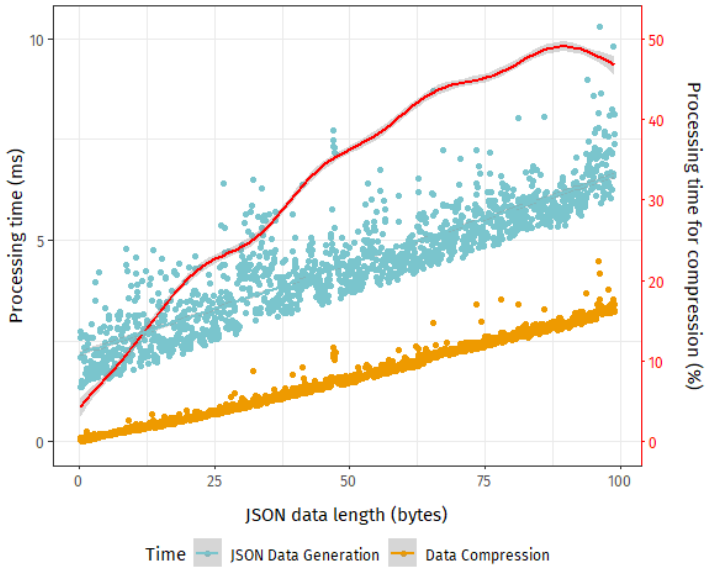


Figure 5. Data compression processing time measurement

Table 3. Compression Processing Time and Memory Usage

JSON Data Length (bytes)	Generation Time (ms)		Memory Usage (kbytes)	
	JSON Data	Zlib+Base64 (additional time)	JSON Data	Zlib+Base64 (additional memory)
68	2.59	0.064 (2.49%)	414.38	0.16 (0.04%)
135	2.73	0.102 (3.74%)	414.99	0.19 (0.05%)
1003	1.49	0.058 (3.87%)	423.40	0.50 (0.12%)
1069	1.70	0.066 (3.87%)	423.92	0.50 (0.12%)
12543	2.83	0.345 (12.20%)	543.74	8 (1.47%)
12609	2.15	0.328 (15.27%)	544.26	8 (1.47%)
32585	5.13	0.940 (18.34%)	722.95	12 (1.66%)
32653	4.10	0.931 (22.74%)	723.47	12 (1.66%)
72621	4.82	2.305 (47.78%)	1115.77	24 (2.15%)
72689	4.57	2.255 (49.36%)	1116.29	24 (2.15%)
72757	5.08	2.258 (44.45%)	1116.81	24 (2.15%)
100879	7.16	3.332 (46.53%)	1373.67	36 (2.62%)
100946	7.13	3.319 (46.56%)	1374.19	36 (2.62%)
101014	6.59	3.269 (49.62%)	1374.71	36 (2.62%)

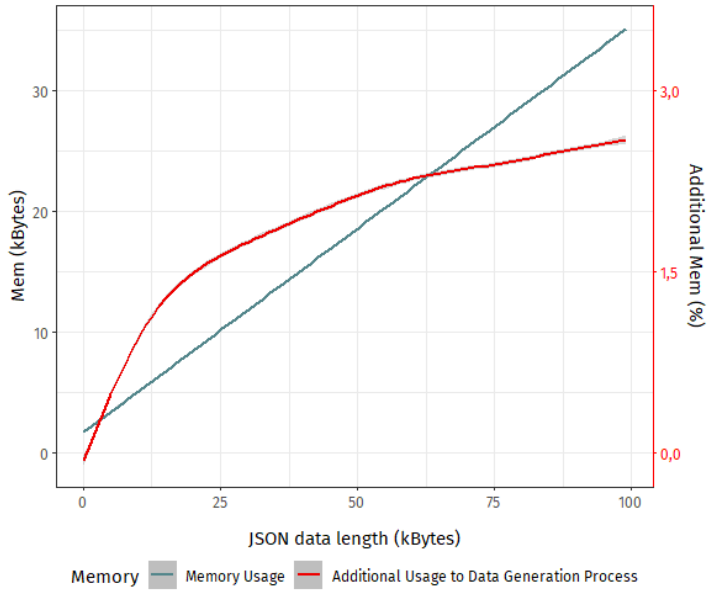


Figure 6. Memory usage of data compression process

4.3 Processing Time and Memory Usage in Data Decompression Process at Client Side Using JavaScript

In this study, the data is received on the client side using a JavaScript-based web application that communicates with the RESTful API and the Google Chrome web

browser. The measurement result on the client side for the decompression process is shown in Table 4 and Fig. 7. The decompression process added additional milliseconds to the data processing at the client side for the testing environment of this study. As a client who consumes the data, these additional milliseconds of processing time should not affect the client application's usability or put some lag in the user experience.

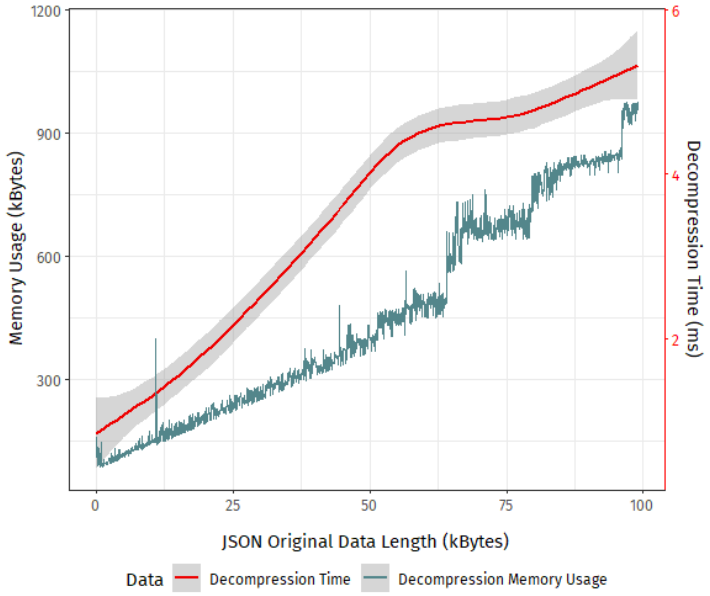


Figure 7. Processing time and memory usage measurement of data decompression process

Considering the memory usage, each request requires a small amount of additional memory, which should not be a problem on a modern device that has gigabytes of memory capacity. However, when the client has a very limited memory space, e.g., embedded systems or Internet of Things (IoT) devices, then the result of this study could become a reference for its implementation decision.

5 Limitation and Future Work

Several limitations apply to this study. First, the RESTful API function used in the measurement lists the point data of a public transportation navigation system, where the length of the output is determined by the number of points returned by the API. Thus, depending on the output of the RESTful API, the yielded performance result may vary from this study. There is also no caching or pagination technique applied during JSON generation. Caching and paginating the data may also contribute to improving the performance of the API.

As for the future work, this study demonstrated the impact of data compression process of an API performance at the application level. However, the performance analysis in contrast to the native web server data compression method of an HTTP gzip request needs to be conceived in the near future to arbitrate its effectiveness in the actual implementation.

Table 4. Decompression Processing Time and Memory Usage

JSON Data Length (bytes)	Compressed Length (bytes)	Processing Time (ms)	Decompression memory usage (kbytes)
68	112	1.30	160.48
135	144	0.30	109.95
1003	424	1	92.70
1069	448	1.30	89.58
12543	4312	0.80	163.67
12609	4336	0.70	157.67
32585	11020	3.90	290.16
32653	11044	3.20	305.03
72621	24148	2	651.97
72689	24172	5	761.17
72757	24192	1.70	677.88
100879	33324	8.10	956.97
100946	33344	2.40	929.78
101014	33364	7.20	956.02

6 Conclusion

When run on modern computers, adding compression and decompression processes to the JSON data of a RESTful API for distribution over the Internet is suggested. The process is able to reduce the amount of data distributed, thereby saving bandwidth on a metered network. Based on these study results, the compression process could save network bandwidth by up to 66%. Additionally, the compression process puts milliseconds of additional CPU processing time on both the client and server sides when implemented on modern computers; hence, minimal additional load. Regarding the memory usage during the data compression and decompression processes, the process took a small percentage of additional memory—that is, less than 2.6% of the actual required memory to generate data in plain JSON String format; hence, the additional load on memory is minimal.

On a very small amount of data, the compression-encoding process may not yield a smaller payload because of the Base64 encoding process. The Base64 encoding is applied to the process to maintain compatibility with media that handle data in simple ASCII characters. Thus, it is suggested to apply the compression and decompression processes when transfer time and network bandwidth become issues in client-server RESTful API communication.

Acknowledgment

This study is supported and partially funded by the DIPA fund of the Faculty of Computer Science, Universitas Brawijaya.

References

1. Jiang, Z., Kuang, R., Gong, J., Yin, H., Lyu, Y., Zhang, X.: What makes a great mobile app? a quantitative study using a new mobile crawler. In: 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE). (2018) 222–227
2. Stocchi, L., Pourazad, N., Michaelidou, N., Tanusondjaja, A., Harrigan, P.: Marketing research on mobile apps: past, present and future. J. of the Acad. Mark. Sci. **50** (2022) 195–225
3. Daniels, B.: Top 11 app engagement strategies (and how to make your app sticky) (2023)
4. Raducanu, A.L., Carabas, M., Barbulescu, M., Tapus, N., Suliman, G.: Client-server application with android mobile phone technology. In: 2019 18th RoEduNet Conference: Networking in Education and Research (RoEduNet). (2019) 1–6
5. Xu, A.: Why restful api so popular? (2022)
6. Ma, S.P., Hsu, M.J., Chen, H.J., Lin, C.J.: Restful api analysis, recommendation, and client code retrieval. Electronics **12**(5) (2023)
7. Neumann, A., Laranjeiro, N., Bernardino, J.: An analysis of public rest web service apis. IEEE Transactions on Services Computing **14**(4) (2021) 957–970
8. Tornes, A.: Full-archive search api success stories: Gnip customer union metrics (2016)
9. Maleshkova, M., Pedrinaci, C., Domingue, J.: Investigating web apis on the world wide web. In: 2010 Eighth IEEE European Conference on Web Services. (2010) 107–114
10. Renzel, D., Schlebusch, P., Klamma, R.: Today's top "restful" services and why they are not restful. In Wang, X.S., Cruz, I., Delis, A., Huang, G., eds.: Web Information Systems Engineering - WISE 2012, Berlin, Heidelberg, Springer Berlin Heidelberg (2012) 354–367
11. Bülthoff, F., Maleshkova, M.: Restful or restless – current state of today's top web apis. In Presutti, V., Blomqvist, E., Troncy, R., Sack, H., Papadakis, I., Tordai, A., eds.: The Semantic Web: ESWC 2014 Satellite Events, Cham, Springer International Publishing (2014) 64–74
12. Kopecký, J., Fremantle, P., Boakes, R.: A history and future of web apis. it - Information Technology **56**(3) (2014) 90–97
13. Haupt, F., Leymann, F., Pautasso, C.: A conversation based approach for modeling rest apis. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture. (2015) 165–174
14. Mohan, M.: Optimizing rest api performance: Advanced techniques (2023)
15. Rodríguez, C., Baez, M., Daniel, F., Casati, F., Trabucco, J.C., Canali, L., Percannella, G.: Rest apis: A large-scale analysis of compliance with principles and best practices. In Bozzon, A., Cudre-Maroux, P., Pautasso, C., eds.: Web Engineering, Cham, Springer International Publishing (2016) 21–39
16. Tiwary, G.P., Stroulia, E., Srivastava, A.: Compression of xml and json api responses. IEEE Access **9** (2021) 57426–57439
17. Muła, W., Lemire, D.: Base64 encoding and decoding at almost the speed of a memory copy. Software: Practice and Experience **50**(2) (2020) 89–97
18. Wen, S., Dang, W.: Research on base64 encoding algorithm and php implementation. In: 2018 26th International Conference on Geoinformatics. (2018) 1–5
19. Sohan, S.M., Maurer, F., Anslow, C., Robillard, M.P.: A study of the effectiveness of usage examples in rest api documentation. In: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). (2017) 53–61
20. Yoshikawa, N., Kubo, R., Yamamoto, K.Z.: Twitter integration of chemistry software tools. Journal of Cheminformatics **13**(1) (Jul 2021) 46
21. Tupitsin, A., Puzrin, V.: pako: zlib port to javascript, very fast! (2022)
22. LaMorte, W.W.: Mann whitney u test (wilcoxon rank sum test) (2017)