Optimizing Large Data API Response in ABAP Cloud: JSON vs CSV and Compression

Natanael Marian

July 6, 2025

Summary

I started with a weekly data pull that moved between my team and another team in the same company ~**0.6 GB** of JSON, 3 million flat rows, through almost six hundred tiny calls that each waited about 16 s. End-to-end, the job dragged on for ~3 hours. The section below shows at a glance how a few format and compression tweaks collapsed that window to seconds while keeping the code readable and the infra happy.

- **Format switch**: dropping JSON keys and emitting plain vanilla CSV trimmed ~77% of the raw payload right off the bat.(final delivery format is gzip-compressed JSON – see Conclusion)
- **Compression punch**: one `CL_ABAP_GZIP` call shrank the total response from ~0.6GB to **9 MB** a ~98 % cut, so the network became a non-issue.
- **Faster serialization**: CSV generation in ABAP ran ~85 % quicker than the original JSON routine, however the lean `CALL TRANSFORMATION` JSON stays only 18 % behind.
- **Chunking without streaming**: processing 1 million row chunks avoided memory blow-ups yet still produced a single contiguous gzip member that every HTTP client can unzip automatically.
- **Throughput leap**: three parallel requests now pull the whole dataset in **6–7 seconds** instead of **3 hours** a ~450× speed-up compared with the first cut.

The rest of the report walks through the numbers, code snippets, and trade-offs that led to this result, why the choice between CSV and JSON isn't just about format anymore, it depends on performance, payload size, and how the data is consumed.

1 Introduction and Background

Our team oversees the integration of several systems and manage a large dataset that needs to be exposed to other consumers. The current architecture for delivering this data is an **OData service in ABAP in Cloud**, proxied through a **CAP (Cloud Application Programming) service**, and then exposed to our consumers via a centralized **Kong API gateway**. The dataset resides in an SAP HANA cloud database table and consists of roughly **3 million records** with **12 fields** each (all stored as strings). Example fields include `CalMonth`, `CalWeek`, `MaterialId`, `ForecastVersion`, etc. The naive approach of returning the entire dataset as one JSON payload would result in an enormous ~0.6 GB response, which is impractical.

To make the data transfer robust, the consumer currently retrieves the data *in batches of 5,000 records* per API call.

With ~2.965 million records, that means about **593 API calls**, each taking ~16 seconds to respond, for a total data load time of roughly **3 hours**.

This is done once a week, and while the weekly schedule and available time window make it *technically acceptable*, it is far from optimal.

Waiting 16 seconds for $< 0.2\%$ of the data per call is inefficient and raised concerns about performance and scalability. My goal was to significantly reduce the payload size and improve throughput, thereby reducing the total time to retrieve the full dataset.

In this report, it is presented a systematic exploration of strategies to shrink the response size (and generation time) for our API, including reformatting JSON data, using CSV, applying compression algorithms, and adjusting our implementation to overcome platform limitations. By applying these optimizations, we were able to **cut down the data transfer time** from **3 hours to couple of seconds**, with drastic reductions in data size.

2 Analysing JSON Overhead and Considering CSV Format

The first place to seek improvements was the **payload format itself**. The OData service originally returns the data in JSON. JSON is human-readable and convenient for structured data, but it is verbose: every record repeats the same field names and includes structural characters (quotes, braces, commas) that add significant overhead. In my case, each record has 12 properties, and a quick inspection showed that the property names have lengths ranging from 3 to 15 characters. For every record, JSON includes each property name in quotes followed by a colon (e.g., `"CalMonth": "202301"`), plus curly braces {} around the object. These repeated keys and JSON syntax inflate the payload size without adding new information for each record. I estimated 168 bytes of overhead per record just from repeating property names and JSON syntax. Given that all records share the same schema and order of fields, this overhead is completely redundant data that the client does not actually need for interpretation.

If we remove the property names and just send the values in order, we can still understand each field by its position (as long as the schema is agreed upon). This essentially transforms the output into a delimited format, which is very close to CSV (Comma-Separated Values). CSV stores data in a compact form without repeating keys, making it much more size-efficient for flat tabular data. Indeed, it's well-known that JSON files are typically larger than CSV because they contain repeated key names and structural brackets, whereas CSV files are more compact for flat data.

JSON Size Comparison (5k records, minified)	
Original JSON size	1,150,806 bytes
No-property JSON size	330,428 bytes
CSV size	265,552 bytes

Reduction from Original	
Property names removed	820,378 bytes saved (71.3% reduction)
Converted to CSV	885,254 bytes saved (76.9% reduction)

This shows that simply removing JSON keys cut the size by about 71%, and converting fully to CSV cut it by 77%. Extrapolating to the full dataset (~3 million records), a JSON export of ~600 MB could potentially be reduced to around 140 MB just by switching to CSV format.

3 Impact of Compression Algorithms on Payload Size

Text-based formats like JSON and CSV compress extremely well using standard compression algorithms. Since the data is highly repetitive (especially the JSON with repeated keys, and even CSV with recurring values or patterns), applying compression can yield dramatic reductions in size. The ABAP environment provides a built-in class `CL_ABAP_GZIP` for **gzip** compression of strings, so I first evaluated **gzip**, which is widely supported by HTTP clients.

GZIP Compressed Sizes	
Gzipped JSON	16,255 bytes (98.59% reduction)
Gzipped CSV	13,586 bytes (98.82% reduction)

Amazingly, GZIP reduced the JSON payload by 98.6 % (to about 1.4 % of the original size), down to approximately 16 kB. The CSV, being smaller to begin with, compressed to roughly 13.6 kB, a 98.8 % reduction. The absolute difference between gzipped JSON and gzipped CSV for the sample was only about 2.7 kB (approximately a 16 % difference in favor of CSV).

This modest gap highlights that GZIP is so effective on JSON that it largely nullifies the savings gained by stripping out property names. The DEFLATE algorithm used by GZIP identifies repeated substrings such as JSON field names appearing in every record, and replaces them with short back-references. In other words, the compressor itself handles the redundancy of JSON keys very efficiently, which is why gzipped JSON shrank almost as much as gzipped CSV. The CSV still edges out slightly because it contains only raw values and separators, making it marginally more compressible.

Encouraged by GZIP's effectiveness, I considered whether other compression algorithms could do even better. Modern algorithms like **Brotli** and **Zstandard (zstd)** often achieve higher compression ratios than GZIP. LZ4 is known for very fast compression at lower ratios. Although ABAP in Cloud doesn't natively support these algorithms, I performed an offline experiment by exporting the data and compressing it with various methods(including converting to **Parquet**, a columnar binary format, for curiosity):
From this comparison, Zstandard (zstd) appeared to have the best balance of high compression and speed, but given the lack of native support in ABAP and the already excellent results with GZIP,

Method	Size (bytes)	Saved (%)	Compress	Decompress
brotli	8,520	99.26	3865.91 ms	0.49 ms
parquet	9,972	99.13	3.42 ms	1630.81 ms
zstd	**10,252**	**99.11**	**0.38 ms**	**0.28 ms**
gzip	16,255	98.59	28.69 ms	0.62 ms
lz4	42,418	96.31	0.17 ms	0.32 ms

I decided that GZIP was the most practical choice for the current system.

4 JSON vs. CSV Generation Performance in ABAP

Another critical factor is the **serialization time** on the server. The data in the HANA table isn't JSON to begin with – it's relational rows. In ABAP, to produce the JSON output, the system must iterate through 3 million records and construct JSON text (adding braces, quotes, escaping special characters, etc.). I suspected that generating CSV text would be significantly faster, because it's a simpler concatenation without all the JSON syntax. To verify this, I wrote a simple ABAP method to convert an internal table of 1,000,000 records to JSON and another method to convert to CSV, then measured the execution time of each. See Appendix A and D for full details of the implemenation.

Called Method	Total Time [µs]
COMPARE_JSON_VS_CSV	19 378 428
CONVERT_JSON_UI2	15 703 135
CONVERT_CSV_STATIC	2 466 328

Converting to CSV was about 85% faster than converting to JSON.

This is a massive difference. The JSON generation in ABAP is expensive, likely due to large string handling and formatting overhead for each record, whereas CSV generation is close to just string concatenation of fields with separators. At the first glance it looked like we should output the data in CSV format (and compress it), rather than JSON, to save both network time and server processing time.

However I was amazed by how much time it can take to create a JSON string from an internal table in ABAP. This led me to look deeper into JSON generation performance. In our earlier comparison of CSV vs. JSON output, we used the `/ui2/cl_json` class to generate JSON. However, ABAP in the Cloud offers multiple ways to produce JSON, each with its own advantages and disadvantages. Here, I explore three different approaches:

1. Using the built-in `/ui2/cl_json` class (as used in the initial comparison).
2. Using the `xco_cp_json` class from the XCO library.
3. Using the native `CALL TRANSFORMATION id` operation to convert data to JSON.

To compare these, I implemented three simple methods that each take an internal table as input and convert it to a JSON string using one of the above approaches. A driver program loads data from the HANA database into an internal table and then calls each conversion method in turn.

I profiled the performance of these methods using a dataset of 100,000 records. The results are shown in Table 1.

Method	Total Time [µs]
`CONVERT_JSON_XCO`	197,287,161
`CONVERT_JSON_UI2`	1,907,204
`CONVERT_JSON_TRANSFORMATION`	349,263

Table 1: Performance of different JSON generation methods for 100,000 records

As we can see, the transformation method (`CALL TRANSFORMATION`) is the fastest by a large margin. The UI2 class method is roughly 5–6 times slower than the transformation, and the XCO class method is **extremely** slow in this scenario (about 565 times slower than the transformation approach).

Besides that one important factor that we skipped is the fact that the CSV conversion from Appendix D is relying on fixed static column names, which might work in our scenario, but is definitely not robust, scalable, and practical. However all 3 JSON conversion methods are working dynamically without the need for specifying the column names, so in order to "compare apple to apple" I decided to write a dynamic CSV conversion method(Appendix J) to profile it against the JSON winner `CALL TRANSFORMATION`.

Method	Total Time [µs]
`CONVERT_JSON_TRANSFORMATION`	1,364,231
`CONVERT_CSV_DYNAMIC`	11,961,237

Table 2: Performance of JSON vs CSV generation for 1,000,000 records

These findings indicate that our initial CSV vs. JSON comparison, which used the `/ui2/cl_json` method, was not entirely fair, CSV appeared much faster because we weren't using the most efficient JSON generation method available, also because of the advantage of the CSV generation of using statically defined column names. But because JSON is now smaller and faster to generate with CALL TRANSFORMATION, we selected gzip-JSON as the production format.

So far, everything pointed to a solution where the ABAP service would: (1) read the 3M rows into an internal table, (2) serialize that table to a JSON text, and (3) GZIP the text and send it as the HTTP response. The next step was to implement and test this in a prototype.

5 Implementation: Chunking Data to Manage Memory

My first naive attempt to load all 3 million rows and convert to JSON failed due to running out of session memory. The memory session can of course be extended, but in any situation this wouldn't scale well.

To solve this, we rethought the process in terms of streaming or chunk processing. An analogy helps here: imagine you have a 1000-liter tank of liquid that needs to be boiled down to 100 liters of concentrate, but you only have a 10-liter pot and a 100-liter container for the concentrate. You can't boil all 1000 liters at once, so you would take 10 liters at a time, boil it down to ∼1 liter, pour it into the concentrate container, and repeat this 100 times. In our case, the "liquid" is the

raw data, and "boiling down" is the act of converting to JSON and compressing it. We need to do it in chunks.

Chunking approach: I decided to read the data in manageable batches (chunks of a few hundred thousand rows), convert each chunk to JSON, compress that chunk, and write it to the HTTP response output sequentially. This way, we never have more than one chunk in memory at a time (aside from a buffer for the accumulating compressed output, which is much smaller). The idea was to stream the response out incrementally. In this way while we are processing the second batch, the consumer is already downloading the first batch, similarily how it is depicted in the Figure 1

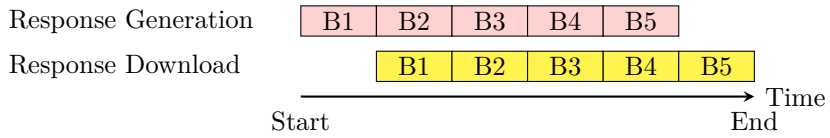

Figure 1: Streaming responses

However, we hit another obstacle: the ABAP server's ICF (Internet Communication Framework) in cloud does not support true streaming of the response in HTTP/1.1 chunked transfer mode. The ICF follows HTTP/1.0 semantics where it expects to know the full content length or have the complete response assembled before sending. (In fact, chunked transfer encoding was introduced in HTTP/1.1 and isn't supported by ABAP's ICF, which still behaves like HTTP/1.0) This meant we couldn't flush chunks to the client one by one in a live stream, the platform would still try to build the entire response internally before sending it, negating our memory-saving and time-saving strategy. Therefore the flow will be the one presented in the approach showed in the Figure 2

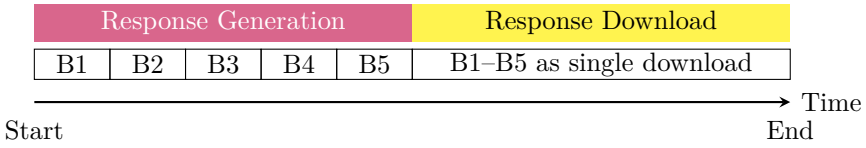

Figure 2: No-streaming responses

To work within this limitation, I slightly adjusted the approach: we still process in chunks internally, but instead of trying to send each chunk out immediately, we append each compressed chunk's bytes to a growing output buffer. In other words, we build the final compressed file incrementally in memory by concatenating compressed segments. This is not as good as true streaming (we still need memory to hold the compressed result), but note that the compressed result is much smaller (around 9 MB for the whole dataset) which is feasible to hold in memory.

We couldn't hold the whole internal table easily, but 9 MB of compressed data is not a problem. So effectively we traded one problem (large raw data in memory) for a manageable one (compressed data in memory).

One implication of this approach is that the final HTTP response we send is a single compressed file that actually consists of multiple concatenated gzip streams (one per chunk). A standards-compliant decompression should handle this fine (the gzip format allows multiple members concatenated). But not all clients automatically handle concatenated streams. For example, the testing in Postman revealed that if we simply concatenated gzip outputs, Postman did not decode it automatically. We'd need custom code on the client side to stitch the pieces together. I even

wrote a Python snippet to demonstrate how to manually decompress such a concatenated deflate stream:

Listing 1: Decompressing raw DEFLATE

```
out = bytearray()
buf = compressed_data  # this is the concatenated compressed chunks
while buf:
    d = zlib.decompressobj(-zlib.MAX_WBITS)  # raw DEFLATE decompressor
    out.extend(d.decompress(buf))
    buf = d.unused_data
```

This loop takes a `buf` containing multiple compressed blocks and iteratively decodes each block, using the `unused_data` to find where one compressed chunk ended. This is clearly more involved than a single call to `decompress`. Also, tools like Postman or many HTTP libraries might not do this automatically for the developer.

Fortunately, if we use one single chunk per response (i.e., compress the entire payload as one stream) and set the HTTP header `Content-Encoding: gzip`, most HTTP clients (browsers, Postman, etc.) will automatically decompress the content on the fly. That's the standard mechanism for content compression over HTTP. I realized that if we could avoid splitting the response into multiple compressed parts, things would be simpler for the consumers.

I tested various chunk sizes to see how it affected performance and if there was any significant overhead in splitting the data, the results are presented in the following table

# of pages	# records per page	total # of records	time (s)	size (MB)
1	3,000,000	3,000,000	error: the program ran out of memory	
2	1,500,000	3,000,000	19.7	9.21
3	1,000,000	3,000,000	20.5	9.20
6	500,000	3,000,000	19.8	9.24
1	1,000,000	1,000,000	6.5	3.07

Table 3: Performance metrics for different paging configurations

Key Insights

- Single request for all 3 mil. records was **not possible** due to memory (the process ran out of memory before completion).
- The **response size** stays relatively **constant** regardless of the *# of records per page*.
- The **response time** is also **not** influenced by the number of pages, as long as the *total # of records* stays constant.

And in order to compare how does the *total # of records* scales I tried fetching only 1,000,000 records in a single page in a single request. And from this we can draw one more insight.

- The response size and time grows linearly with respect to the *total # of records* regardless of the *# of records per page*

Given these results, I chose 1 million records per call as the sweet spot. With that size, a single request completes in ∼6–7 seconds. If the consumer makes 3 requests in parallel (each for 1 million rows), the entire 3 million rows can be retrieved in roughly 6–7 seconds total, assuming the infrastructure can handle three parallel threads. Even if done serially, it would be ∼3 × 6.5 s = ∼19.5 s, which is already a huge improvement over 3 hours! In practice, parallelizing the calls brings it down to essentially the time of one chunk, since the calls overlap.

By using one chunk per response, we also keep the compression stream contiguous, which allowed us to simply send the response with `Content-Encoding: deflate` and offload the decompression to standard HTTP mechanisms. In the tests, when we called the new API from Postman with the `content-encoding` header set, Postman was able to automatically decompress and display the JSON content immediately, confirming this approach works smoothly for clients.

Besides that one of the most important features of sending a single chunk of JSON records is the fact that we're able to enable pagination for this API, which is extremely important, more on that in the next chapter.

6 Pagination

Given the fact that we plan to make this solution scalable, our go-to is to send responses of 1 million records, this way in our case we can fetch the full dataset in just 3 API calls, assuming that each API call is not very intense for the whole ABAP infrastructure, we can make all 3 calls in parallel so we can reduce the total amount of time to the time of one single call, which in our case is under 7 seconds.

However it's pretty hard to do that, and to stick to the REST best practices without a proper pagination mechanism, that's why in a production scenario we'd have to provide the implementation for pagination as well.

One way of doing it can be inspired from the OData pattern where we can send a specific parameter to fetch the total amount of records. In order to enhance a bit this functionality with more information we can provide the customer as well with some recommended page sizes, having a pagination object as following:

Listing 2: Pagination Object

```
{
  "number_of_records" : 2496434,
  "batch_size" : {
    "maximum" : 1500000,
    "recommended" : 1000000
  },
  "recommended_pages" : [
    "/entity?offset=0&count=1000000",
    "/entity?offset=1000000&count=1000000",
    "/entity?offset=2000000&count=1000000"
  ]
}
```

Therefore the overall flow of the calls will be as demonstrated in Figure 3.

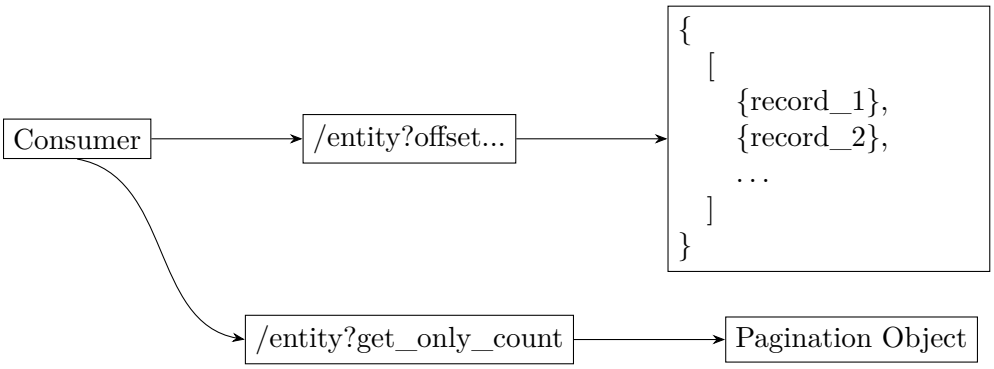

Figure 3: API Call flow

7 Results and Discussion

The transformation is dramatic. What used to be a 3-hour data load involving nearly 600 small requests is now a process of a few seconds via three highly compressed large responses. More specifically:

- **Original approach:** ∼3 hours to fetch ∼600 MB (in 593 calls of 5k records each). Each 5k payload was ∼1.15 MB JSON (uncompressed) and took ∼16 seconds.
- **New approach:** ∼7 seconds to fetch ∼9 MB (in 3 calls of 1M records each, compressed JSON). Each 1M payload is ∼3 MB gzipped JSON and takes ∼6–7 seconds.

To put it another way, one 5,000-record JSON call (0.2% of data) originally took 16 s, whereas now we can retrieve 1,000,000 records (33% of data) in under 7 s. That's over 450× the throughput per second. And the entire dataset can be obtained in fractions of the original time (or faster, if parallelized). The key factors enabling this improvement are:

- Leveraged compression (GZIP) to reduce transfer size by about 98%.
- Improved serialization efficiency (CALL TRANSFORMATION generation in ABAP is much faster).
- Adjusted to ABAP's constraints by using single-chunk responses and parallel client requests.

8 Trade-offs and caveats

1. Giving up on `OData` over plain HTTP
    - The first trade-off that we are making is giving up on OData, over plain HTTP communication, OData is amazing for the integration with Fiori apps, and in many otehr usecases, however when it comes to huge data loads it might not be the best solution.

- The give up on the OData, comes with multiple consequences, for example if we decide now to use filtering, pagination, and other features that are provided out of the box by the OData, we'd have to implement these manually with custom code.

2. Client changes and CPU overhead

   - Another consideration is client changes: the consumer originally expecting JSON now has to handle a gzipped JSON response. Most languages and tools can handle gzipped data automatically or with minor adjustments (e.g. setting an Accept-Encoding: gzip header and then parsing CSV).
   - We should mention that using compression does add some CPU overhead on both server (to compress) and client (to decompress), but these proved to be minimal relative to the cost of data transfer. These CPU costs are well worth the network time saved. Moreover, as our tests showed, even faster algorithms like zstd exist that could reduce CPU at same compression levels – in the future, if ABAP supports them or we implement them, we could lower that overhead further. It's worth noting that many modern web systems are adopting Zstandard because it achieves similar compression to GZIP but with about 40-50% faster compression speed.
   - One very important item is the fact that the client should be ready to handle this amount of decompressed data, meaning that if previously these calls were done and processed by an AWS Lambda with a configured memory of 128MB, this might no longer be enough, as one decompressed response is only ∼200MB. But let's make a quick calculation of cost for this scenario.

     We will assume that we have a `lambda` that calls the API, fetches the data and then it does some simple operations with the data, for example storing it into an S3; we will neglect the runtime for pushing the data to S3 for now.
     So we have 2 scenarios: compressed and uncompressed, and we have the next requirements: for uncompressed, which is the status quo, we have to keep a lambda running for `3 h`, and the requirement in terms of memory is minimal because we will flush the memory after each call and the minimal `128 MB` lambda will be enough. However, for the compressed scenario we will need a lambda with higher memory, because we will need to be able to handle the whole response; so in this case, given that the response is `200 MB` decompressed, considering other packages and variables, we can go safe with a `1024 MB` lambda, and the runtime for it will be `7 s * 3 = 21 s`.
     We will abstractize from the real cost of running a lambda and we will speak only in units of costs (`uc`), so assuming that a `128 MB` lambda costs `1 uc` per second, the relationship between cost and memory is linear at AWS; therefore the `1024 MB` lambda will cost `8 uc` per second. Running the `128 MB` lambda will cost `1 uc * 3 h * 3600 s/h = 10800 uc`. Running the `1024 MB` lambda will cost `8 uc * 21 s = 160 uc`. Which concludes that running the compressed scenario is `68×` times cheaper for the client rather than running the decompressed scenario. But let's keep in mind that this is a theoretical calculation for one single type of client infrastructure; in real life this can widely differ.

Scenario	Lambda size	Cost per second (uc)	Runtime	Total cost (uc)
Uncompressed	`128 MB`	`1`	`3 h`	`10800`
Compressed	`1024 MB`	`8`	`21 s`	`160`

9 Conclusion

Through this exploration and prototyping, we achieved a vastly more efficient data export mechanism for our large dataset. By applying GZIP compression, we reduced the payload size by over 98%, and by optimizing server-side generation and adjusting to constraints, we cut down the total data transfer time by **from ∼3 hours to 7 seconds**. The final solution was implemented as an HTTP service in the ABAP cloud environment that returns a gzip-compressed JSON. The consumer can retrieve the data with a few parallel calls and ingest the JSON content directly or via standard libraries.

Going forward, this approach can be extended or refined. If even larger volumes or higher frequencies are needed, we might look into splitting by logical partitions (so consumers only fetch subsets they need), or utilizing more advanced compression like Zstandard if it becomes available in our environment. This also gives a framework for other usecases in other teams, so for example if they want to highly optimize the server response time, and strip at maximum the size of the respone the CSV is the go-to, however the team will have to look into a more efficient CSV generation mechanism, or using a less scalable statical column names. But for now, the solution of JSON + Gzip meets the requirements efficiently.

All column names in the following code and examples have been obfuscated to preserve data privacy.

Appendix A UI2 JSON serialization

```
METHODS convert_json_ui2 IMPORTING data TYPE tt_data_subset
                         RETURNING VALUE(string) TYPE string.

METHOD convert_json_ui2.
  RETURN /ui2/cl_json⇒serialize(
    EXPORTING
      data = data
  ).
ENDMETHOD.
```

Appendix B XCO JSON serialization

```
METHODS convert_json_xco IMPORTING data TYPE tt_data_subset
                         RETURNING VALUE(string) TYPE string.

METHOD convert_json_xco.
  RETURN xco_cp_json⇒data→from_abap( data )→to_string( ).
ENDMETHOD.
```

Appendix C TRANSFORMATION JSON serialization

```
METHODS convert_json_transformation IMPORTING data TYPE tt_data_subset
                                    RETURNING VALUE(string) TYPE xstring.

METHOD convert_json_transformation.
  DATA(lo_writer) = cl_sxml_string_writer⇒create(
                    type = if_sxml⇒co_xt_json ).

  CALL TRANSFORMATION id
    SOURCE itab  = data
    RESULT XML   lo_writer.

  string = lo_writer→get_output( ).
ENDMETHOD.
```

Appendix D CSV serialization

```
METHODS convert_csv_static IMPORTING data TYPE tt_data_subset
                    RETURNING VALUE(string) TYPE string.

METHOD convert_csv_static.
  string = |column_1;column_2;column_3;column_4;column_5;column_6;|
         &&|column_7;column_8;column_9;|
         &&|column_10;column_11;column_12\n|.

  LOOP AT data INTO DATA(line).
    string = string &&
      |{ line-column_1 };{ line-column_2 };{ line-column_3 };|
   &&|{ line-column_4 };{ line-column_5 };{ line-column_6 };|
   &&|{ line-column_7 };{ line-column_8 };{ line-column_9 };|
   &&|{ line-column_10 };{ line-column_11 };{ line-column_12 }\n|.
  ENDLOOP.
ENDMETHOD.
```

Appendix E Profiling: JSON vs CSV

```
METHODS compare_json_vs_csv.

METHOD compare_json_vs_csv.
  SELECT column_1 column_2 column_3 column_4 column_5 column_6
         column_7 column_8 column_9 column_10 column_11 column_12
    FROM /namespace/dbtable
    INTO TABLE @DATA(page)
    UP TO 1000000 ROWS.

  convert_json_ui2( page ).
  convert_csv_static(       page ).
ENDMETHOD.
```

Appendix F HTTP service handler

```
METHOD if_http_service_extension~handle_request.

METHOD if_http_service_extension~handle_request.
  " choose one of these
  gzip_json_single_page(    CHANGING request = request response = response ).

  " only for demonstration
  "gzip_csv_single_page(    CHANGING request = request response = response ).

  " only for demonstration
  "gzip_csv_multiple_pages( CHANGING request = request response = response ).
```

```
ENDMETHOD.
```

Appendix G CSV + GZip over multiple encoded chunks

```
METHODS gzip_csv_multiple_pages IMPORTING page_size TYPE i DEFAULT 1000
                                          max_pages TYPE i DEFAULT 1
                                CHANGING  response  TYPE REF TO if_web_http_response
                                          request   TYPE REF TO if_web_http_request.

METHOD gzip_csv_multiple_pages.

  CONSTANTS file_name  TYPE string VALUE 'data_subset.gz'.

  "-------------------------------------------------------------
  " HTTP response headers
  "-------------------------------------------------------------
  response→set_status( 200 ).
  response→set_content_type( 'application/gzip' ).
  response→set_header_field(
      i_name  = 'Content-Disposition'
      i_value = |attachment; filename="{ file_name }"| ).
  response→set_compression(
      options = if_web_http_response⇒co_compress_none ).

  " we cannot use this content encoding anymore because the client will
  " decompress only the first chunk, not the full response

  " response→set_header_field(
  "     i_name  = 'Content-Encoding'
  "     i_value = |deflate| ).


  "-------------------------------------------------------------
  " helpers
  "-------------------------------------------------------------
  DATA gzip_payload     TYPE xstring.
  DATA gzip_chunk       TYPE xstring.
  DATA csv_buffer       TYPE string.
  DATA page             TYPE STANDARD TABLE OF /namespace/cds.
  DATA page_counter     TYPE i VALUE 0.

  "-------------------------------------------------------------
  " CSV header  (always one GZIP member)
  "-------------------------------------------------------------
  csv_buffer = |column_1;column_2;column_3;column_4;column_5;column_6;|
             & |column_7;column_8;column_9;|
             & |column_10;column_11;column_12\n|.

  cl_abap_gzip⇒compress_text(
      EXPORTING text_in  = csv_buffer
      IMPORTING gzip_out = gzip_chunk ).
```

```
CONCATENATE gzip_payload gzip_chunk INTO gzip_payload IN BYTE MODE.

"------------------------------------------------------------
" main loop - keep going until we created max_pages
"------------------------------------------------------------
WHILE page_counter < max_pages.

  SELECT column_1,
         column_2,
         column_3,
         column_4,
         column_5,
         column_6,
         column_7,
         column_8,
         column_9,
         column_10,
         column_11,
         column_12
    FROM /namespace/dbtable
    ORDER BY column_2
    INTO TABLE @page
    UP TO @page_size ROWS.


  "------------------------------------------------------------
  " build CSV for this page
  "------------------------------------------------------------
  CLEAR csv_buffer.
  LOOP AT page INTO DATA(line).
    csv_buffer = csv_buffer &&
      |{ line-column_1 };{ line-column_2 };{ line-column_3 };|
    && |{ line-column_4 };{ line-column_5 };{ line-column_6 };|
    && |{ line-column_7 };{ line-column_8 };{ line-column_9 };|
    && |{ line-column_10 };{ line-column_11 };{ line-column_12 }\n|.
  ENDLOOP.


  "------------------------------------------------------------
  " compress & append
  "------------------------------------------------------------
  cl_abap_gzip⇒compress_text(
      EXPORTING text_in  = csv_buffer
      IMPORTING gzip_out = gzip_chunk ).

  CONCATENATE gzip_payload gzip_chunk
         INTO gzip_payload IN BYTE MODE.

  "------------------------------------------------------------
  " advance cursor & counter
  "------------------------------------------------------------
  CLEAR page.
```

```
    page_counter = page_counter + 1.

  ENDWHILE.

  "-------------------------------------------------------------
  " send the response
  "-------------------------------------------------------------
  response→set_binary( gzip_payload ).

ENDMETHOD.
```

Appendix H CSV + GZip over single encoded chunks

```
METHODS gzip_csv_single_page IMPORTING page_size TYPE i DEFAULT 1000
                             CHANGING  response  TYPE REF TO if_web_http_response
                                       request   TYPE REF TO if_web_http_request.

METHOD gzip_csv_single_page.

  CONSTANTS file_name  TYPE string VALUE 'data_subset.gz'.

  "-------------------------------------------------------------
  " HTTP response headers
  "-------------------------------------------------------------
  response→set_status( 200 ).
  response→set_content_type( 'application/gzip' ).
  response→set_header_field(
      i_name  = 'Content-Disposition'
      i_value = |attachment; filename="{ file_name }"| ).
  response→set_compression(
      options = if_web_http_response⇒co_compress_none ).

  response→set_header_field(
      i_name  = 'Content-Encoding'
      i_value = |deflate| ).

  "-------------------------------------------------------------
  " helpers
  "-------------------------------------------------------------
  DATA csv_buffer       TYPE string.

  "-------------------------------------------------------------
  " CSV header
  "-------------------------------------------------------------
  csv_buffer = |column_1;column_2;column_3;column_4;column_5;column_6;|
             & |column_7;column_8;column_9;|
             & |column_10;column_11;column_12\n|.


  "-------------------------------------------------------------
  " data fetching
```

```
  "-------------------------------------------------------------
  SELECT column_1,
         column_2,
         column_3,
         column_4,
         column_5,
         column_6,
         column_7,
         column_8,
         column_9,
         column_10,
         column_11,
         column_12
    FROM /namespace/dbtable
    ORDER BY column_2
    INTO TABLE @DATA(page)
    UP TO @page_size ROWS.


  "-------------------------------------------------------------
  " build CSV for this page
  "-------------------------------------------------------------
  LOOP AT page INTO DATA(line).
    csv_buffer = csv_buffer &&
      |{ line-column_1 };{ line-column_2 };{ line-column_3 };|
    && |{ line-column_4 };{ line-column_5 };{ line-column_6 };|
    && |{ line-column_7 };{ line-column_8 };{ line-column_9 };|
    && |{ line-column_10 };{ line-column_11 };{ line-column_12 }\n|.
  ENDLOOP.


  "-------------------------------------------------------------
  " compress & append
  "-------------------------------------------------------------
  cl_abap_gzip⇒compress_text(
      EXPORTING text_in  = csv_buffer
      IMPORTING gzip_out = DATA(gzip) ).


  "-------------------------------------------------------------
  " send the response
  "-------------------------------------------------------------
  response→set_binary( gzip ).
ENDMETHOD.
```

Appendix I JSON + GZip over multiple encoded chunks

```
METHODS gzip_json_single_page IMPORTING page_size TYPE i DEFAULT 1000
                              CHANGING  response  TYPE REF TO if_web_http_response
                                        request   TYPE REF TO if_web_http_request.
```

```
METHOD gzip_json_single_page.

  CONSTANTS file_name  TYPE string VALUE 'data_subset.gz'.

  "-------------------------------------------------------------
  " HTTP response headers
  "-------------------------------------------------------------
  response→set_status( 200 ).
  response→set_content_type( 'application/gzip' ).
  response→set_header_field(
      i_name  = 'Content-Disposition'
      i_value = |attachment; filename="{ file_name }"| ).
  response→set_compression(
      options = if_web_http_response⇒co_compress_none ).

  response→set_header_field(
      i_name  = 'Content-Encoding'
      i_value = |deflate| ).

  "-------------------------------------------------------------
  " data fetching
  "-------------------------------------------------------------
  SELECT column_1,
         column_2,
         column_3,
         column_4,
         column_5,
         column_6,
         column_7,
         column_8,
         column_9,
         column_10,
         column_11,
         column_12
    FROM /namespace/dbtable
    ORDER BY column_2
    INTO TABLE @DATA(page)
    UP TO @page_size ROWS.


  "-------------------------------------------------------------
  " compress & append
  "-------------------------------------------------------------
  cl_abap_gzip⇒compress_binary(
      EXPORTING raw_in  = convert_json_transformation( page )
      IMPORTING gzip_out = DATA(gzip) ).

  "-------------------------------------------------------------
  " send the response
  "-------------------------------------------------------------
  response→set_binary( gzip ).


ENDMETHOD.
```

Appendix J Dynamic CSV generation

```
METHODS convert_csv IMPORTING data TYPE tt_data_subset
                    RETURNING VALUE(string) TYPE string.
METHOD convert_csv.

    DATA lt_column_names TYPE STANDARD TABLE OF string WITH EMPTY KEY.

    DATA(lo_tab_descr) = CAST cl_abap_tabledescr(
                           cl_abap_tabledescr⇒describe_by_data_ref( REF #( data ) )
                         ).
    DATA(lo_line_descr) = CAST cl_abap_structdescr( lo_tab_descr→get_table_line_type( ) ).

    LOOP AT lo_line_descr→components ASSIGNING FIELD-SYMBOL(<comp>).
      APPEND <comp>-name TO lt_column_names.
      string = string && <comp>-name && ';'.
    ENDLOOP.
    string = string && '\n'.

    LOOP AT data INTO DATA(line).
      LOOP AT lt_column_names INTO DATA(column).
        ASSIGN COMPONENT column OF STRUCTURE line TO FIELD-SYMBOL(<val>).
        string = string && |{ <val> };|.
      ENDLOOP.
      string = string && '\n'.
    ENDLOOP.

ENDMETHOD.
```