Performance tips

This document covers some techniques you can use to improve the performance of your application. In some cases, examples from other APIs or generic APIs are used to illustrate the ideas presented. However, the same concepts are applicable to the Google Calendar API.

Compression using gzip

An easy and convenient way to reduce the bandwidth needed for each request is to enable gzip compression. Although this requires additional CPU time to uncompress the results, the trade-off with network costs usually makes it very worthwhile.

In order to receive a gzip-encoded response you must do two things: Set an **`Accept-Encoding`** header, and modify your user agent to contain the string **`gzip`**. Here is an example of properly formed HTTP headers for enabling gzip compression:

```
Accept-Encoding: gzip
User-Agent: my program (gzip)
```

Working with partial resources

Another way to improve the performance of your API calls is by sending and receiving only the portion of the data that you're interested in. This lets your application avoid transferring, parsing, and storing unneeded fields, so it can use resources including network, CPU, and memory more efficiently.

There are two types of partial requests:

- Partial response (#partial-response): in the response (use the **`fields`** requ
- Patch (#patch): An update request v use the **`PATCH`** HTTP verb).

More details on making partial requests are provided in the following sections.

Partial response

By default, the server sends back the full representation of a resource after processing requests. For better performance, you can ask the server to send only the fields you really need and get a *partial response* instead.

To request a partial response, use the **`fields`** request parameter to specify the fields you want returned. You can use this parameter with any request that returns response data.

Note that the **`fields`** parameter only affects the response data; it does not affect the data that you need to send, if any. To reduce the amount of data you send when modifying resources, use a patch (#patch) request.

Example

The following example shows the use of the **`fields`** parameter with a generic (fictional) "Demo" API.

Simple request: This HTTP **`GET`** request omits the **`fields`** parameter and returns the full resource.

```
https://www.googleapis.com/demo/v1
```

Full resource response: The full resource data includes the following fields, along with many others that have been omitted for brevity.

```
{
  "kind": "demo",
  ...
  "items": [
  {
    "title": "First title",
    "comment": "First comment.",
    "characteristics": {
      "length": "short",
      "accuracy": "high",
      "followers": ["Jo", "Will"],
    },
    "status": "active",
    ...
  },
  {
    "title": "Second title",
    "comment": "Second comment.",
    "characteristics": {
      "length": "long",
      "accuracy": "medium"
      "followers": [ ],
    },
    "status": "pending",
    ...
  },
  ...
  ]
}
```

Request for a partial response: The following request for this same resource uses the `fields` parameter to significantly reduce the amount of data returned.

```
https://www.googleapis.com/demo/v1?fields=kind,items(title,characteristics/lengt
```

Partial response: In response to the request above, the server sends back a response that contains only the kind information along with a pared-down items array that includes only HTML title and length characteristic information in each item.

```
200 OK
```

```
{
  "kind": "demo",
  "items": [{
    "title": "First title",
    "characteristics": {
      "length": "short"
    }
  }, {
    "title": "Second title",
    "characteristics": {
      "length": "long"
    }
  },
  ...
  ]
}
```

Note that the response is a JSON object that includes only the selected fields and their enclosing parent objects.

Details on how to format the `fields` parameter is covered next, followed by more details about what exactly gets returned in the response.

Fields parameter syntax summary

The format of the `fields` request parameter value is loosely based on XPath syntax. The supported syntax is summarized below, and additional examples are provided in the following section.

- Use a comma-separated list to select multiple fields.
- Use `a/b` to select a field `b` that is nested within field `a`; use `a/b/c` to select a field `c` nested within `b`.

  Exception: For API responses that use "data" wrappers, where the response is nested within a `data` object that looks like `data: { ... }`, do not include "`data`" in the `fields` specification. Including the data object with a fields specification like `data/a/b` causes an error. Instead, just use a `fields` specification like `a/b`.
- Use a sub-selector to request a set of specific sub-fields of arrays or objects by placing expressions in parentheses "`( )`".

  For example: `fields=items(id,author/email)` returns only the item ID and author's email for each element in the items array. You can also specify a single sub-field, where `fields=items(id)` is equivalent to `fields=items/id`.
- Use wildcards in field selections, if needed.
  For example: `fields=items/pagemap/*` selects all objects in a pagemap.

More examples of using the fields parameter

The examples below include descriptions of how the `fields` parameter value affects the response.

Note: As with all query parameter values, the `fields` parameter value must be URL encoded. For better readability, the examples in this document omit the encoding.

Identify the fields you want returned, or make *field selections*.

The `fields` request parameter value is a comma-separated list of fields, and each field is specified relative to the root of the response. Thus, if you are performing a list operation, the response is a collection, and it generally includes an array of resources. If you are performing an operation that returns a single resource, fields are specified relative to that resource. If the field you select is (or is part of) an array, the server returns the selected portion of all elements in the array.

Here are some collection-level examples:

Examples	Effect
`items`	Returns all elements in the items array, including all fields in each element, but no other fields.
`etag,items`	Returns both the `etag` field and all elements in the items array.
`items/title`	Returns only the `title` field for all elements in the items array. Whenever a nested field is returned, the response includes the enclosing parent objects. The parent fields do not include any other child fields unless they are also selected explicitly.
`context/facets/label`	Returns only the `label` field for all members of the `facets` array, which is itself nested under the `context` object.
`items/pagemap/*/title`	For each element in the items array, returns only the `title` field (if present) of all objects that are children of `pagemap`.

Here are some resource-level examples:

Examples	Effect
`title`	Returns the `title` field of the requested resource.
`author/uri`	Returns the `uri` sub-field of the `author` object in the requested resource.
`links/*/href`	Returns the `href` field of all objects that are children of `links`.

Request only parts of specific fields using *sub-selections*.

By default, if your request specifies particular fields, the server returns the objects or array elements in their entirety. You can specify a response that includes only certain sub-fields. You do this using "`( )`" sub-selection syntax, as in the example below.

Example	Effect
`items(title,author/uri)`	Returns only the values of the `title` and author's `uri` for each element in the items array.

Handling partial responses

After a server processes a valid request that includes the `fields` query parameter, it sends back an HTTP `200 OK` status code, along with the requested data. If the `fields` query parameter has an error or is otherwise invalid, the server returns an HTTP `400 Bad Request` status code, along with an error message telling the user what was wrong with their fields selection (for example, "`Invalid field selection a/b`").

Here is the partial response example shown in the introductory section (#partial-response) above. The request uses the `fields` parameter to specify which fields to return.

```
https://www.googleapis.com/demo/v1?fields=kind,items(title,characteristics/lengt
```

The partial response looks like this:

```
200 OK
```

```
{
  "kind": "demo",
  "items": [{
    "title": "First title",
    "characteristics": {
      "length": "short"
    }
  }, {
    "title": "Second title",
    "characteristics": {
      "length": "long"
    }
  },
  ...
  ]
}
```

Note: For APIs that support query parameters for data pagination (**`maxResults`** and **`nextPageToken`**, for example), use those parameters to reduce the results of each query to a manageable size. Otherwise, the performance gains possible with partial response might not be realized.

Patch (partial update)

You can also avoid sending unnecessary data when modifying resources. To send updated data only for the specific fields that you're changing, use the HTTP `PATCH` verb. The patch semantics described in this document are different (and simpler) than they were for the older, GData implementation of partial update.

The short example below shows how using patch minimizes the data you need to send to make a small update.

Example

This example shows a simple patch request to update only the title of a generic (fictional) "Demo" API resource. The resource also has a comment, a set of characteristics, status, and many other fields, but this request only sends the `title` field, since that's the only field being modified:

```
PATCH https://www.googleapis.com/demo/v1/324
Authorization: Bearer your_auth_token
Content-Type: application/json

{
  "title": "New title"
}
```

Response:

```
200 OK
```

```
{
  "title": "New title",
  "comment": "First comment.",
  "characteristics": {
    "length": "short",
    "accuracy": "high",
    "followers": ["Jo", "Will"],
  },
  "status": "active",
  ...
}
```

The server returns a `200 OK` status code, along with the full representation of the updated resource. Since only the `title` field was included in the patch request, that's the only value that is different from before.

Note: If you use the partial response (#partial-response) **`fields`** parameter in combination with patch, you can increase the efficiency of your update requests even further. A patch request only reduces the size of the request. A partial response reduces the size of the response. So to reduce the amount of data sent in both directions, use a patch request with the **`fields`** parameter.

Semantics of a patch request

The body of the patch request includes only the resource fields you want to modify. When you specify a field, you must include any enclosing parent objects, just as the enclosing parents are returned with a partial response (#example-partial-response). The modified data you send is merged into the data for the parent object, if there is one.

- **Add:** To add a field that doesn't already exist, specify the new field and its value.
- **Modify:** To change the value of an existing field, specify the field and set it to the new value.
- **Delete:** To delete a field, specify the field and set it to `null`. For example, `"comment": null`. You can also delete an entire object (if it is mutable) by setting it to `null`. If you are using the Java API Client Library (/api-client-library/java), use `Data.NULL_STRING` instead; for details, see JSON null (/api-client-library/java/google-http-java-client/json#json_null).

Note about arrays: Patch requests that contain arrays replace the existing array with the one you provide. You cannot modify, add, or delete items in an array in a piecemeal fashion.

Using patch in a read-modify-write cycle

It can be a useful practice to start by retrieving a partial response with the data you want to modify. This is especially important for resources that use ETags, since you must provide the current ETag value in the `If-Match` HTTP header in order to update the resource successfully. After you get the data, you can then modify the values you want to change and send the modified partial representation back with a patch request. Here is an example that assumes the Demo resource uses ETags:

```
GET https://www.googleapis.com/demo/v1/324?fields=etag,title,comment,characteris
Authorization: Bearer your_auth_token
```

This is the partial response:

```
200 OK



{
  "etag": "ETagString"
  "title": "New title"
  "comment": "First comment.",
  "characteristics": {
    "length": "short",
    "level": "5",
    "followers": ["Jo", "Will"],
  }
}
```

The following patch request is based on that response. As shown below, it also uses the `fields` parameter to limit the data returned in the patch response:

```
PATCH https://www.googleapis.com/demo/v1/324?fields=etag,title,comment,character
Authorization: Bearer your_auth_token
Content-Type: application/json
If-Match: "ETagString"




{
  "etag": "ETagString"
  "title": "",                /* Clear the value of the title by setting it to
                                 Delete the comment by replacing its value wit


                                 Modify the level value.
                                 Replace the followers array to delete Will an
                                 Add a new characteristic. */
  },
}
```

The server responds with a 200 OK HTTP status code, and the partial representation of the updated resource:

```
200 OK
```

```
{
  "etag": "newETagString"
  "title": "",               /* Title is cleared; deleted comment field is mis


                                Value is updated.*/
    "followers": ["Jo" "Liz"], /* New follower Liz is present; deleted Will is m
                                New characteristic is present. */
  }
}
```

Constructing a patch request directly

For some patch requests, you need to base them on the data you previously retrieved. For example, if you want to add an item to an array and don't want to lose any of the existing array elements, you must get the existing data first. Similarly, if an API uses ETags, you need to send the previous ETag value with your request in order to update the resource successfully.

Note: You can use an "`If-Match: *`" HTTP header to force a patch to go through when ETags are in use. If you do this, you don't need to do the read before the write.

For other situations, however, you can construct the patch request directly, without first retrieving the existing data. For example, you can easily set up a patch request that updates a field to a new value or adds a new field. Here is an example:

```
PATCH https://www.googleapis.com/demo/v1/324?fields=comment,characteristics
Authorization: Bearer your_auth_token
Content-Type: application/json

{
  "comment": "A new comment",
  "characteristics": {
    "volume": "loud",
    "accuracy": null
  }
}
```

With this request, if the comment field has an existing value, the new value overwrites it; otherwise it is set to the new value. Similarly, if there was a volume characteristic, its value is overwritten; if not, it is created. The accuracy field, if set, is removed.

Handling the response to a patch

After processing a valid patch request, the API returns a `200 OK` HTTP response code along with the complete representation of the modified resource. If ETags are used by the API, the server updates ETag values when it successfully processes a patch request, just as it does with `PUT`.

The patch request returns the entire resource representation unless you use the `fields` parameter to reduce the amount of data it returns.

If a patch request results in a new resource state that is syntactically or semantically invalid, the server returns a `400 Bad Request` or `422 Unprocessable Entity` HTTP status code, and the resource state remains unchanged. For example, if you attempt to delete the value for a required field, the server returns an error.

Alternate notation when PATCH HTTP verb is not supported

If your firewall does not allow HTTP `PATCH` requests, then do an HTTP `POST` request and set the override header to `PATCH`, as shown below:

```
POST https://www.googleapis.com/...
X-HTTP-Method-Override: PATCH
...
```

Difference between patch and update

In practice, when you send data for an update request that uses the HTTP `PUT` verb, you only need to send those fields which are either required or optional; if you send values for fields that are set by the server, they are ignored. Although this might seem like another way to do a partial update, this approach has some limitations. With updates that use the HTTP `PUT` verb, the request fails if you don't supply required parameters, and it clears previously set data if you don't supply optional parameters.

It's much safer to use patch for this reason. You only supply data for the fields you want to change; fields that you omit are not cleared. The only exception to this rule occurs with repeating elements or arrays: If you omit all of them, they stay just as they are; if you provide any of them, the whole set is replaced with the set that you provide.