PAYTER
DATA API MANUAL

June 2024

Contents

Revision History

Version	Date	Author	Notes
1.0	14/6/2024	Product Delivery	Initial externally shareable version

Introduction

The MyPayter API is a HTTP REST api, used to access the MyPayter back-end. It contains several sub-sections, including the Data API which can be used to retrieve reporting data. The API can be reached at *https://api.mypayter.com/API*.

This manual will cover how to perform requests, and how to retrieve the data you need from the Data API.

Authentication

Before performing any data API request, an authenticate request must be completed. A successful authentication results in a authentication token, which needs to be included in subsequent requests.

Authentication is done by a **POST** request to the **/Auth** URL, supplying the credentials in the body, as shown below:

```
{
        "username":"USERNAME",
        "password":"PASSWORD",
        "domain":"DOMAIN"
}
```

A successful authentication will result in a response containing the authentication token:

```
{
   "tokenId": "3ea2714d-c445-4c5e-8590-28d806065784",
   "userId": 1,
   "username": "USERNAME",
   "domain": "DOMAIN",
   "created": 1541154878996,
   "expiresIn": 86400
}
```

The response contains the token, some information about the authenticated user, and a created timestamp and expiry for the token (in seconds). The token value must be supplied in the header of the other API requests to bind to this authentication context.

In case of an error during authentication, the response will have a HTTP error status, and will contain an error description:

```
{
   "code": -1,
   "url": "/API/Auth",
   "description": "failed to authenticate"
}
```

It is strongly recommended to use a single token for multiple requests. Token expiry can depend on several factors, but the token will be valid for at least several hours.

To use the authentication token add the following HTTP header any request, including the token you retrieved from the authentication request of course.

```
Authorization: CURO-TOKEN token="3ea2714d-c445-4c5e-8590-28d806065784"
```

The current status of the token can be retrieved by making a **GET** request to the **/Auth/{token}** URL. This can be used to examine the

After completion, it is recommended to end the session by closing the authentication context. This can be done with a **DELETE** request to the **/Auth/{token}** URL, supplying the token to delete in **both** the header **and** the URL.

For sample HTTP requests and responses concerning authentication, please see *Appendix A.1*.

General Data Requests

The Data API can be used to retrieve specific data records in the different indexes, but also to calculate aggregations over data sets (or both).

Access the data with a **POST** of a data request to the **/Data** URL. The data request is shown below.

```
{
        "index": "Events",
        "query": "",
        "maxResults": 10,
        "offset": 0,
        "sorts": [],
        "filters": [],
        "aggregations": []
}
```

The data request contains:

- the index to query
- a Lucene query to filter the data
- the maximum number of documents to return
- an optional offset (used for paging results)
- a list of filters (allowing more complex filtering than the query)
- a list of sort fields
- a list of data aggregations

Index

The available indexes and their relevant fields will be covered in the next chapter.

Query

The query parameter is used to filter the data elements using a Lucene query. This query is performed before the other filters. It follows the Lucene query syntax.

```
<query> ::= <part>|<part> <query>
<part>  ::= <modifier><term>
<modifier> ::= "+" | "-" | ""
<term>  ::= <field>":"<matcher> | "(" <query> ")"
<field> ::= [a-z]+
<matcher> ::= [a-z*? ]
```

For example: *+domain:DOMAIN +(field:exactvalue field2:wildcard*)*

Every term prefixed by a "+" must be matched in the result. Conversely every term prefixed by a "-" should not be matched. A term without a prefix can be matched. The query *+fieldA:AAAA +(fieldB:BBBB fieldC:CCCCC)* will match if fieldA matches and either of fieldB or fieldC matches. Wildcard symbol "*" will match any length of any character, while "?" will match any single character. Spaces in the matcher must be escaped using a "\" backslash i.e. *term:ABCD\ EFGH*.

Sort

The sort parameter contains a list of fields to sort by, and the order of the sort. For example:

```
"sorts": [
        { "field": "FIELD1", "asc": true },
        { "field": "FIELD2", "asc": false}
]
```

Where setting "*asc"* to true means ascending sort, and setting it to false means descending sort.

Depending on the type of field in the index, the data is sorted either alphabetically of numerically.

Filters

Filters can be used to refine the data set.

The *exists* filter tests if a specific field exists in the data row. If the field does not exist, this data row will be filtered out. If a field exists but is empty, the exists filter will still match. The optional *negative* parameter can be used to invert the filter.

```
"filters": [
        {
                "type": "exists",
                "field": "fieldA",
                "negative": false
        }
]
```

The *term* filter acts much like the query parameter, in that it matches a specific field to a specific value. The optional *negative* parameter can be used to invert the filter.

```
"filters": [
        {
                "type": "term",
                "field": "fieldA",
                "matches": "AAAAA",
                "negative": false
        }
]
```

The *terms* filter matches to a list of values. If the field value does not match any of the listed value, the data row is filtered out. The optional *negative* parameter can be used to invert the filter.

```
"filters": [
        {
                "type": "terms",
                "field": "fieldA",
                "matches": [
                        "AAAAA",
                        "BBBBB",
                        "CCCCC"
                ],
                "negative": false
        }
]
```

The *range* filter allows checking of values between a specific range of values. It is especially useful when checking date ranges. the range filter works best on numeric fields, but can also be applied to other field. Use the optional *date* parameter to indicate the values are date values.

```
"filters": [
        {
                "type": "range",
                "field": "@timestamp",
                "from": 1505911108649,
                "to": 1508503108650,
                "date": false
        }
]
```

Multiple filters can be combined in a single list. These filters are applied in an **AND** fashion, meaning the filters are cumulative.

The *or* filter can be used to match any of a list of filters.

```
"filters": [
        {
                "type": "or",
                "subFilters": [{
                        "type": "range",
                        "field": "@timestamp",
                        "from": 1505911108649,
                        "to": 1508503108650
                }, {
                        "type": "exists",
                        "field": "fieldA",
                        "negative": false
                }]
        }
]
```

Aggregations

Data aggregations can be used to calculate derived values, by grouping rows into buckets, and calculating total count, in addition to sums, averages, etc.

Aggregations can be nested, using a parameter *subAggregations*, to allow very specific bucketing. Every aggregation is also given a *name*, which can be used to distinguish them in the response.

The *terms* aggregation creates a bucket for each unique value of a specifically defined field, for example calculating transaction totals per currency. It can also be used to match the top hits for a open field, for example the top 50 serial numbers in transaction count. Due to the potential number of different values, the size parameter of a *terms* aggregation is always required.

```
"aggregations": [
        {
                "name": "serials",
                "type": "terms",
                "field": "serialNumber",
                "size": 10,
                "subAggregations":[]
        }
]
```

The *date_histogram* aggregation is used to bucket results per time slot. It contains a parameter *timezone* to convert from UTC time to a specific timezone. It also contains a parameter *interval* which indicates the size of the bucket in time units, e.g. 1d or 30m. The *showEmpty* parameter indicates whether to list empty buckets, in which no events occur. The *offset* parameter can be used as a modifier to the actual interval. For example offset = "+4h" with an interval of "1d" will create buckets from 4 a.m. till 4 a.m. instead of 12 p.m. till 12 p.m.

```
"aggregations": [
        {
                "name": "dhisto",
                "type": "date_histogram",
                "field": "@timestamp",
                "interval": "30m",
                "timezone": "Europe/Berlin",
                "offset": 0,
                "showEmpty": true,
                "subAggregations":[]
        }
]
```

The following interval units are available:

- ms → milliseconds
- s → seconds
- m → minutes
- h → hours
- d → days
- w → weeks
- M → months
- q → quarters
- y → years

The *terms* aggregation creates a bucket for each possible value for that term in the result set. By default the results are sorted by highest to lowest count. The *size* parameter can be used to restrict the maximum number of buckets returned. The optional *missing* parameter supplies a value to be used if a record does not contain the field. The optional *order* parameter can be used if an alternative sorting of the buckets is required.

```
"aggregations":[
        {
                "name":"field values",
                "type": "terms",,
                "field": "fieldA"
                "size": 5,
                "order": [{
                        "field": "fieldB",
                        "asc": true
                }],
                "missing": "UNKNOWN",
                "subAggregations":[]
        }
]
```

The *filter* aggregation filters the underlying buckets. This can be useful when combining multiple aggregations into a single data request, whereby you might want to filter a sub aggregation more that the whole request. The *filters* and *query* parameters can be used to restrict which data rows are counted in the sub aggregations.

```
"aggregations":[
        {
                "name":"average value",
                "type": "filter",
                "filters": [...],
                "query": "",
                "subAggregations":[...]
        }
]
```

The *composite* aggregation can be used to combine multiple fields into a single set of buckets. This can be especially useful when retrieving a large number of buckets from the data. The composite aggregation will create a bucket for every combination of values of the specified fields, and all sub-aggregations will match all the values of the specific bucket.

The optional *after* parameter can be used to specify what the last bucket was that was retrieved in the previous request, so that the request can return the next bucket that would be enumerated.

The following example would result in buckets for each combination of fieldA and fieldB.

```
"aggregations":[
        {
                "name":"my composite",
                "type": "composite",
                "sources": [ {
                        "field": "fieldA",
                        "missing": true
                },{
                        "field": "fieldB",
                        "missing": false
                }],
                "size": 10,
                "after": { "fieldA": "VAL1", "fieldB", "AnotherValue" },
                "subAggregations":[]
        }
]
```

The *metric* aggregation is used to calculate specific metrics for each bucket. It cannot have sub-aggregations.

```
"aggregations":[
        {
                "name":"average value",
                "type": "metric",,
                "field": "fieldA"
                "operation": "avg"
        }
]
```

The following metrics are available:

- cardinality → unique count
- avg → average value
- sum → summed value
- max → maximum value
- min → minimum value

By default every bucket will also contain a total result count value.

Paging

The *maxResults* and *offset* parameters can be used to restrict the returned data rows. It does not restrict what data to use in the aggregations. The *maxResults* parameter specifies the maximum number of rows to return. The *offset* parameter indicated how many data rows to skip.

Since it it possible to be only interested in the aggregations, a *maxResults* value of 0 will actually return 0 data rows. Every data lookup will return the actual number of matched results.

It is recommended to prefer large data sets (5k-10k rows), over multiple requests. So 1x1000 rows is better than 10x100 rows.

Data lookup response

The response from a data lookup request contains the following elements:

- count → the total number of matched data rows
- documents → a list of all the matched data rows, see section on indexes for document examples.
- aggregations → a nested map of the aggregations from the request, by name

```
{
        "count": 6,
        "documents": [],
        "aggregations": {
                "dhisto": {
                        "type": "date_histogram",
                        "name": "dhisto",
                        "buckets": [
                                {
                                        "name": "2017-10-06T00:00:00.000+02:00",
                                        "key": "1507240800000",
                                        "count": 3,
                                        "aggregations": {
                                                "unique": {
                                                        "type": "cardinality",
                                                        "name": "unique",
                                                        "value": "3.0",
                                                        "buckets": []
                            }}
                                },
                                ...
                                {
                                        "name": "2017-10-19T00:00:00.000+02:00",
                                        "key": "1508364000000",
                                        "count": 3,
                                        "aggregations": {
                                                "unique": {
                                                        "type": "cardinality",
                                                        "name": "unique",
                                                        "value": "3.0",
                                                        "buckets": []
                                        }}
                                }
                        ]
                }
        }
}
```

Every bucket will be given a name and a key. In the case of a terms aggregation, these will be equal. In the case of a date histogram the key will be the timestamp, and the name the timezone value of the timestamp.

Metric aggregations contain a *value* result with the numeric value of the metric.

Indexes

The primary indexes available via the data API are:

- Transactions → all payment transactions
- Events → all terminal events

Transaction Document


```
@timestamp: "2020-07-30T08:55:14.000Z"
@version: "1"
authCode: "945852"
authTime: 5081
authTimestamp: "2020-07-30T08:55:20.000Z"
authorizedAmount: 90
brand: "A0000000043060"
cardTime: 554
cardTimestamp: "2020-07-30T08:55:14.000Z"
commitTime: 3052
commitTimestamp: "2020-07-30T08:55:29.000Z"
committedAmount: 90
complete: true
currency: "EUR"
disposition: "APPROVED"
errorIndicator: "0000000000FF"
errors: []
event-authorized: "P6820200000000:1596099320000:421614"
event-card: "P6820200000000:1596099314000:421612"
event-committed: "P6820200000000:1596099329000:421618"
extra-CARD-EASE-CARD-HASH: "qb16M4mwnwD6A31baG1xSEAIg98="
extra-CARD-EASE-CARD-REFERENCE: "ab11aed9-42d2-ea11-80ca-ecf4bbe4a9fb"
extra-MDB-SESSION-DATA: "0301F4"
extra-TXN-AUTH-CODE: "945852"
extra-TXN-AUTH-RESP-CODE: "00"
extra-TXN-MASKED-PAN: "670343xxxxxxx8127"
extra-Timezone: "Europe/Amsterdam"
host: "CREDITCALL"
hostTransactionId: "e4b14a63-42d2-ea11-80c4-00505a4161f2"
id: "P6820200000000:26386"
ifd: "CONTACTLESS"
ingestSequence: 11808917
ingestTime: "2020-07-30T08:55:49.340Z"
ingestTopic: "json-terminal-transactions"
itemRef: "96"
labels: []
posReference: "10782467"
serialNumber: "P6820200000000"
siteDomain: "DOMAIN_CODE"
siteReference: "c-s-1234"
state: "COMMITTED"
terminalDomain: "DOMAIN_CODE"
terminalReference: "c-t-15234"
terminalResult: "OK"
transactionId: "26386"
txnTimestamp: "2020-07-30T08:55:14.000Z"
type: "ONLINE"
unexpectedEventStream: false
```

Event Document


```
{
      "sourceId": 512,
      "sequenceNumber": 577,
      "extra-Timezone": "Europe/Amsterdam",
      "syncTimestamp": "2018-11-30T09:13:14.449Z",
      "terminalReference": "c-t-8329",
      "serialNumber": "P6X20182200216",
      "siteReference": "c-s-9559",
      "ingestSequence": 1528123,
      "terminalAccessDomain":       [
         "PAYTER_RD",
         "PAYTER",
         "CURO"
      ],
      "ingestTopic": "json-terminal-events",
      "ingestTime": "2018-11-30T09:13:25.863Z",
      "parseErrors": [],
      "body": "DA0104",
      "LOG-Type": "ROTATE_RTP",
      "siteDomain": "PAYTER_RD",
      "@timestamp": "2018-11-30T09:12:51.000Z",
      "@version": "1",
      "siteAccessDomain":       [
         "PAYTER_RD",
         "PAYTER",
         "CURO"
      ],
      "sourceName": "LOG",
      "id": "P6X20182200216:1543569171000:577",
      "terminalDomain": "PAYTER_RD",
      "timestamp": "2018-11-30T09:12:51.000Z"
}
```

Common Use Cases

Import Transaction Data

```
A common use case is to use the API to ingest all transaction data into an external system, the
best way to perform this is using the following request repeated until no more documents are
returned.
{
        "index":"Transactions",
        "maxResults":1000,
        "query":"",
        "sorts": [
                {
                        "field": "ingestTime",
                        "asc": true
                }
        ],
        "filters": [
                {
                        "type": "range",
                        "field": "ingestTime",
                        "from": <LAST INGESTED TIME>
                },
                {
                        "type": "term",
                        "field": "complete",
                        "matches": "true"
                }
        ],
        "aggregations": []
}
```

This will return all records which have a sequence number after the given <LAST INGESTED TIME> and only return records that are complete and will no longer change. Transaction records can change overtime as updates to the transaction occur, for example offline transactions that are posted for settlement will still change after first appearing in the data. They are however only marked as complete once the final result is known.

Due to the distributed nature the backend the requester needs to be prepared to deal with multiple transactions on the same ingested time, there is no monotonically increasing sequence number so a transaction that was the last transaction returned in previous request will be returned again as the first transaction when its ingestTime is used as the <LAST INGESTED TIME>. This is required to ensure any other transactions on the same ingestTime will be returned.

Appendix

Sample HTTP requests

Authentication

Authenticate

```
POST https://api.mypayter.com/API/Auth HTTP/1.1
Content-Type: application/json
Content-Length: 66
Host: api.mypayter.com
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)

{
        "username":"USER",
        "password":"PASSWORD",
        "domain":"DOMAIN"
}
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=utf-8
Transfer-Encoding: chunked
Server: Jetty(9.3.21.v20170918)

{
        "tokenId":"3ea2714d-c445-4c5e-8590-28d806065784",
        "userId":1,
        "username":"USER",
        "domain":"DOMAIN",
        "created":1541154878996,
        "expiresIn":86400
}
```

View token

```
GET https://api.mypayter.com/API/Auth/3ea2714d-c445-4c5e-8590-28d806065784 HTTP/1.1
Accept-Encoding: gzip,deflate
Authorization: CURO-TOKEN token="3ea2714d-c445-4c5e-8590-28d806065784"
Host: api.mypayter.com
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=utf-8
Transfer-Encoding: chunked
Server: Jetty(9.3.21.v20170918)

{
        "tokenId":"3ea2714d-c445-4c5e-8590-28d806065784",
        "userId":1,
        "username":"USER",
        "domain":"DOMAIN",
        "created":1541154878996,
        "expiresIn":86400
}
```

Delete token

```
DELETE https://api.mypayter.com/API/Auth/3ea2714d-c445-4c5e-8590-28d806065784 HTTP/1.1
Accept-Encoding: gzip,deflate
Authorization: CURO-TOKEN token="3ea2714d-c445-4c5e-8590-28d806065784"
Content-Type: application/json
Content-Length: 0
Host: api.mypayter.com
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
```

Response

```
HTTP/1.1 204 No Content
Server: Jetty(9.3.21.v20170918)
```