Pagination

_p & _size

_p indicates page and _size indicates size of response rows

By default 20 records and max of 100 are returned per GET request on a table.

```
/api/payments?_size=50

/api/payments?_p=2

/api/payments?_p=2&_size=50
```

When _size is greater than 100 - number of records defaults to 100 (i.e maximum)

When _size is less than or equal to 0 - number of records defaults to 20 (i.e minimum)

Order by / Sorting

ASC

```
/api/payments?_sort=column1
```

eg: sorts ascending by column1

DESC

```
/api/payments?_sort=-column1
```

eg: sorts descending by column1

Multiple fields in sort

```
/api/payments?_sort=column1,-column2
```

eg: sorts ascending by column1 and descending by column2

Column filtering / Fields

```
/api/payments?_fields=customerNumber,checkNumber
```

eg: gets only customerNumber and checkNumber in response of each record

```
/api/payments?_fields=-checkNumber
```

eg: gets all fields in table row but not checkNumber

Row filtering / Where

Comparison operators

```
eq      -   '='         -  (colName,eq,colValue)

ne      -   '!='        -  (colName,ne,colValue)

gt      -   '>'         -  (colName,gt,colValue)

gte     -   '>='        -  (colName,gte,colValue)

lt      -   '<'         -  (colName,lt,colValue)

lte     -   '<='        -  (colName,lte,colValue)

is      -   'is'        -  (colName,is,true/false/null)

in      -   'in'        -  (colName,in,val1,val2,val3,val4)

bw      -   'between'   -  (colName,bw,val1,val2)

like    -   'like'      -  (colName,like,~name)   note: use ~ in place of %

nlike   -   'not like'  -  (colName,nlike,~name)  note: use ~ in place of %
```

Use of comparison operators

```
/api/payments?_where=(checkNumber,eq,JM555205)~or((amount,gt,200)~and(amount,l
t,2000))
```

Logical operators

```
~or     -   'or'

~and    -   'and'

~xor    -   'xor'
```

Use of logical operators

eg: simple logical expression

```
/api/payments?_where=(checkNumber,eq,JM555205)~or(checkNumber,eq,OM314933)
```

eg: complex logical expression

```
/api/payments?_where=((checkNumber,eq,JM555205)~or(checkNumber,eq,OM314933))~a
nd(amount,gt,100)
```

eg: logical expression with sorting(_sort), pagination(_p), column filtering (_fields)

```
/api/payments?_where=(amount,gte,1000)&_sort=-
amount&p=2&_fields=customerNumber
```

eg: filter of rows using _where is available for relational route URLs too.

```
/api/offices/1/employees?_where=(jobTitle,eq,Sales%20Rep)
```

FindOne

```
/api/tableName/findOne?_where=(id,eq,1)
```

Works similar to list but only returns top/one result. Used in conjunction with _where

Count

```
/api/tableName/count
```

Returns number of rows in table

Exists

```
/api/tableName/1/exists
```

Returns true or false depending on whether record exists

Group By Having as query params

```
/api/offices?_groupby=country
```

eg: SELECT country,count(*) FROM offices GROUP BY country

```
/api/offices?_groupby=country&_having=(_count,gt,1)
```

eg: SELECT country,count(1) as _count FROM offices GROUP BY country having _count > 1

Group By Having as API

```
/api/offices/groupby?_fields=country
```

eg: SELECT country,count(*) FROM offices GROUP BY country

```
/api/offices/groupby?_fields=country,city
```

eg: SELECT country,city,count(*) FROM offices GROUP BY country,city

```
/api/offices/groupby?_fields=country,city&_having=(_count,gt,1)
```

eg: SELECT country,city,count(*) as _count FROM offices GROUP BY country,city having _count > 1

Group By, Order By

```
/api/offices/groupby?_fields=country,city&_sort=city
```

eg: SELECT country,city,count(*) FROM offices GROUP BY country,city ORDER BY city ASC

```
/api/offices/groupby?_fields=country,city&_sort=city,country
```

eg: SELECT country,city,count(*) FROM offices GROUP BY country,city ORDER BY city ASC, country ASC

```
/api/offices/groupby?_fields=country,city&_sort=city,-country
```

eg: SELECT country,city,count(*) FROM offices GROUP BY country,city ORDER BY city ASC, country DESC

Aggregate functions

```
http://localhost:3000/api/payments/aggregate?_fields=amount



response body

[

    {

        "min_of_amount": 615.45,

        "max_of_amount": 120166.58,

        "avg_of_amount": 32431.645531,

        "sum_of_amount": 8853839.23,

        "stddev_of_amount": 20958.625377426568,

        "variance_of_amount": 439263977.71130896

    }

]
```

eg: retrieves all numeric aggregate of a column in a table

```
http://localhost:3000/api/orderDetails/aggregate?_fields=priceEach,quantityOrd
ered



response body

[

    {

        "min_of_priceEach": 26.55,

        "max_of_priceEach": 214.3,

        "avg_of_priceEach": 90.769499,

        "sum_of_priceEach": 271945.42,

        "stddev_of_priceEach": 36.576811252187795,

        "variance_of_priceEach": 1337.8631213781719,

        "min_of_quantityOrdered": 6,

        "max_of_quantityOrdered": 97,

        "avg_of_quantityOrdered": 35.219,

        "sum_of_quantityOrdered": 105516,

        "stddev_of_quantityOrdered": 9.832243813502942,

        "variance_of_quantityOrdered": 96.67301840816688

    }

]
```

eg: retrieves numeric aggregate can be done for multiple columns too

Union of multiple group by statements

??**[HOTNESS ALERT]**

Group by multiple columns in one API call using _fields query params - comes really handy

```
http://localhost:3000/api/employees/ugroupby?_fields=jobTitle,reportsTo



response body

{

   "jobTitle":[

      {

         "Sales Rep":17

      },

      {

         "President":1

      },

      {

         "Sale Manager (EMEA)":1

      },

      {

         "Sales Manager (APAC)":1

      },

      {
         "Sales Manager (NA)":1

      },

      {

         "VP Marketing":1

      },

      {

         "VP Sales":1

      }

   ],

   "reportsTo":[

      {

         "1002":2

      },

      {

         "1056":4

      },

      {

         "1088":3

      },

      {

         "1102":6

      },
   {

      "1143":6

   },

   {

      "1621":1

   }

   {

      "":1

   },

]

}
```

Chart

?????? **[HOTNESS ALERT]**

Chart API returns distribution of a numeric column in a table

It comes in **SEVEN** powerful flavours

Chart : With min, max, step in query params ??

This API returns the number of rows where amount is between (0,25000), (25001,50000) ...

```
/api/payments/chart?_fields=amount&min=0&max=131000&step=25000



Response

```

```
[
  {
    "amount": "0 to 25000",
    "_count": 107
  },
  {
    "amount": "25001 to 50000",
    "_count": 124
  },
  {
    "amount": "50001 to 75000",
    "_count": 30
  },
  {
    "amount": "75001 to 100000",
    "_count": 7
  },
  {
    "amount": "100001 to 125000",
    "_count": 5
  },
  {
    "amount": "125001 to 150000",

    "_count": 0

  }

]
```

Chart : With step array in params ??

This API returns distribution between the step array specified

```
/api/payments/chart?_fields=amount&steparray=0,10000,20000,70000,140000



Response



[

  {

    "amount": "0 to 10000",

    "_count": 42

  },

  {

    "amount": "10001 to 20000",

    "_count": 36

  },

  {

    "amount": "20001 to 70000",

    "_count": 183
  },

  {

    "amount": "70001 to 140000",

    "_count": 12

  }

]
```

??. Chart : With step pairs in params ??

This API returns distribution between each step pair

```
/api/payments/chart?_fields=amount&steppair=0,50000,40000,100000



Response



[

    {"amount":"0 to 50000","_count":231},

    {"amount":"40000 to 100000","_count":80}

]
```

??. Chart : with no params ??

This API figures out even distribution of a numeric column in table and returns the data

```
/api/payments/chart?_fields=amount
```

```
Response

[

  {

    "amount": "-9860 to 11100",

    "_count": 45

  },

  {

    "amount": "11101 to 32060",

    "_count": 91

  },

  {

    "amount": "32061 to 53020",

    "_count": 109

  },

  {

    "amount": "53021 to 73980",

    "_count": 16

  },

  {

    "amount": "73981 to 94940",

    "_count": 7
  },

  {

    "amount": "94941 to 115900",

    "_count": 3

  },

  {

    "amount": "115901 to 130650",

    "_count": 2

  }

]
```

??. Chart : range, min, max, step in query params ??

This API returns the number of rows where amount is between (0,25000), (0,50000) ... (0,maxValue)

Number of records for amount is counted from min value to extended *Range* instead of incremental steps

```
/api/payments/chart?_fields=amount&min=0&max=131000&step=25000&range=1



Response



[

   {

       "amount": "0 to 25000",

       "_count": 107
```

```
    },

    {

        "amount": "0 to 50000",

        "_count": 231

    },

    {

        "amount": "0 to 75000",

        "_count": 261

    },

    {

        "amount": "0 to 100000",

        "_count": 268

    },

    {

        "amount": "0 to 125000",

        "_count": 273

    }

]
```

6. Range can be specified with step array like below

```
/api/payments/chart?_fields=amount&steparray=0,10000,20000,70000,140000&range=1
```

```
[

   {

       "amount": "0 to 10000",

       "_count": 42

   },

   {

       "amount": "0 to 20000",

       "_count": 78

   },

   {

       "amount": "0 to 70000",

       "_count": 261

   },

   {

       "amount": "0 to 140000",

       "_count": 273

   }

]
```

7. Range can be specified without any step params like below

```
/api/payments/chart?_fields=amount&range=1



[

    {
        "amount": "-9860 to 11100",

        "_count": 45

    },

    {

        "amount": "-9860 to 32060",

        "_count": 136

    },

    ...



]
```

Please Note: _fields in Chart API can only take numeric column as its argument.

Autochart

Identifies numeric columns in a table which are not any sort of key and applies chart API as before - feels like magic when there are multiple numeric columns in table while hacking/prototyping and you invoke this API.

```
http://localhost:3000/api/payments/autochart



[

    {

        "column": "amount",

        "chart": [

                    {
```

```
            "amount": "-9860 to 11100",
            "_count": 45
        },
        {
            "amount": "11101 to 32060",
            "_count": 91
        },
        {
            "amount": "32061 to 53020",
            "_count": 109
        },
        {
            "amount": "53021 to 73980",
            "_count": 16
        },
        {
            "amount": "73981 to 94940",
            "_count": 7
        },
        {
            "amount": "94941 to 115900",
            "_count": 3
```

```
                    },

                    {

                        "amount": "115901 to 130650",

                        "_count": 2

                    }

                ]

    }

]
```

XJOIN

Xjoin query params and values:

```
_join           :   List of tableNames alternated by type of join to be made
(_j, _ij,_ lj, _rj)

alias.tableName :   TableName as alias

_j              :   Join [ _j => join, _ij => ij, _lj => left join , _rj =>
right join)

_onNumber       :   Number 'n' indicates condition to be applied for 'n'th
join between (n-1) and 'n'th table in list
```

Simple example of two table join:

Sql join query:

```
SELECT pl.field1, pr.field2
FROM productlines as pl
    JOIN products as pr
        ON pl.productline = pr.productline
```

Equivalent xjoin query API:

```
/api/xjoin?_join=pl.productlines,_j,pr.products&_on1=(pl.productline,eq,pr.pro
ductline)&_fields=pl.field1,pr.field2
```

Multiple tables join

Sql join query:

```
SELECT pl.field1, pr.field2, ord.field3
FROM productlines as pl
    JOIN products as pr
        ON pl.productline = pr.productline
    JOIN orderdetails as ord
        ON pr.productcode = ord.productcode
```

Equivalent xjoin query API:

```
/api/xjoin?_join=pl.productlines,_j,pr.products,_j,ord.orderDetails&_on1=(pl.p
roductline,eq,pr.productline)&_on2=(pr.productcode,eq,ord.productcode)&_fields
=pl.field1,pr.field2,ord.field3
```

Explanation:

pl.productlines => productlines as pl
_j => join
pr.products => products as pl
_on1 => join condition between productlines and products =>
(pl.productline,eq,pr.productline)
_on2 => join condition between products and orderdetails =>
(pr.productcode,eq,ord.productcode)
Example to use : _fields, _where, _p, _size in query params

```
/api/xjoin?_join=pl.productlines,_j,pr.products&_on1=(pl.productline,eq,pr.pro
ductline)&_fields=pl.productline,pr.productName&_size=2&_where=(productName,li
ke,1972~)
```

Response:

```
[{"pl_productline":"Classic Cars","pr_productName":"1972 Alfa Romeo GTA"}]
```

Please note : Xjoin response has aliases for fields like below aliasTableName + '_' + columnName.
eg: pl.productline in _fields query params - returns as pl_productline in response.

Run dynamic queries

Dynamic queries on a database can be run by POST method to URL localhost:3000/dynamic

This is enabled **ONLY when using local mysql server** i.e -h localhost or -h 127.0.0.1 option.

Post body takes two fields : query and params.

query: SQL query or SQL prepared query (ones with ?? and ?)
params : parameters for SQL prepared query

```
POST /dynamic



    {

        "query": "select * from ?? limit 1,20",

        "params": ["customers"]

    }
```

POST /dynamic URL can have any suffix to it - which can be helpful in prototyping

eg:

```
POST /dynamic/weeklyReport

POST /dynamic/user/update
```