Day 4

Created	@January 15, 2024 3:27 PM
Status	Open
Updated	@January 21, 2024 5:57 PM

Reports:

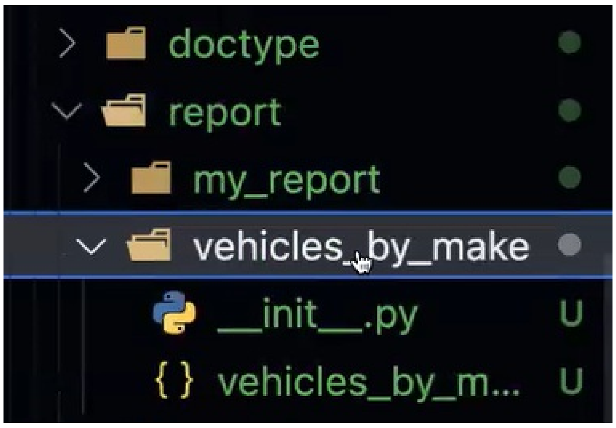
Report Builder → no-code report

Using Report Builder, we can only use the Ref Doctype, meaning we can't use other doctypes through sql join etc.

When we click `Show Report`, it will redirect the user to the Report View of the doctype by default. When we create/customize report through the Report View, we will need to save the report.

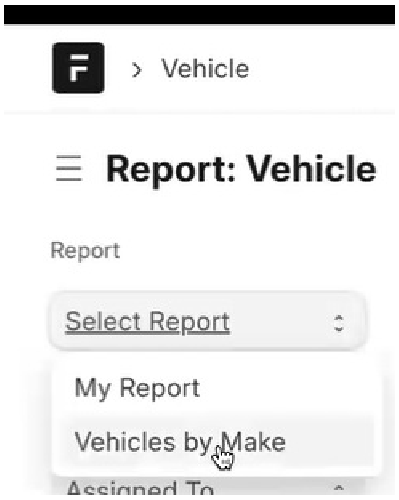
Files Generated (Report Builder & Query Report)

If we checked the standard checkbox, the files will be generated inside the `report` directory:



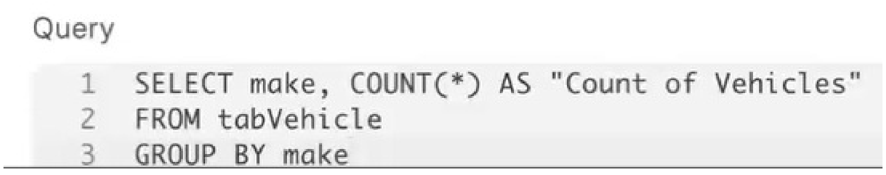
```
> doctype
v report
  > my_report
  v vehicles_by_make
      __init__.py           U
      vehicles_by_m...      U
```

The Ref Doctype will also affect the Report view selection on the doctype:



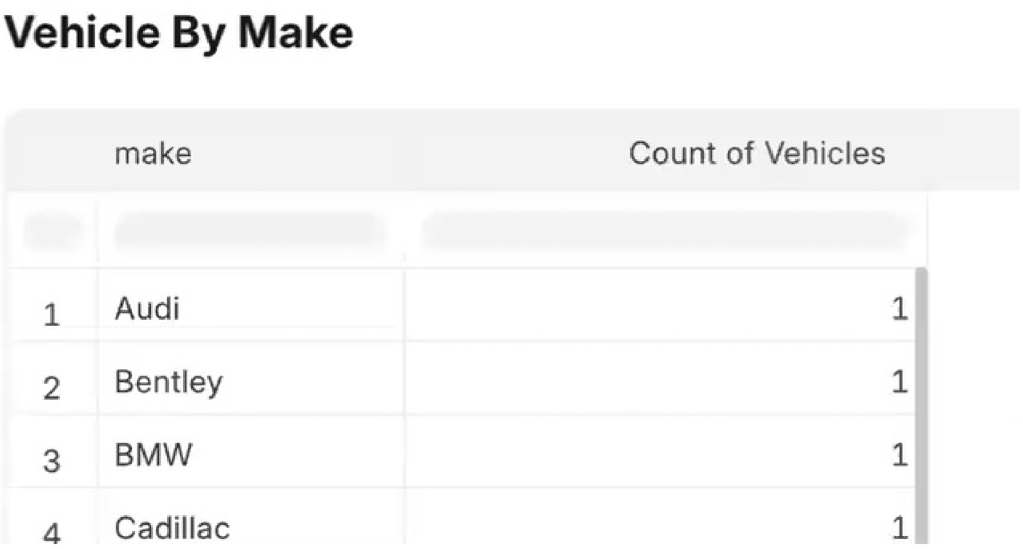
Query Report → it lets us write SQL query, and then generates a view on top of it.

Even though we have to input Ref DocType, it doesn't mean that we can't use other doctype since we can just query them.



Query

```sql
SELECT make, COUNT(*) AS "Count of Vehicles"
FROM tabVehicle
GROUP BY make
```

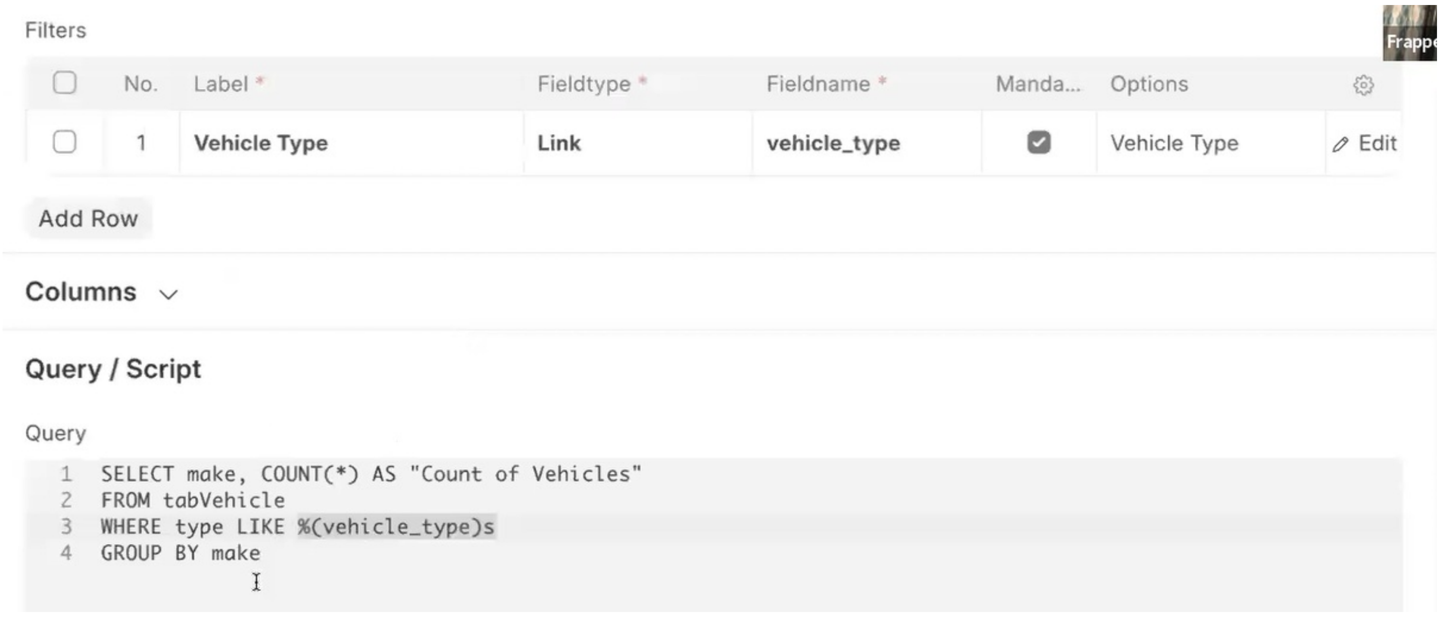


Vehicle By Make

	make	Count of Vehicles
1	Audi	1
2	Bentley	1
3	BMW	1
4	Cadillac	1

Add Filters & Columns

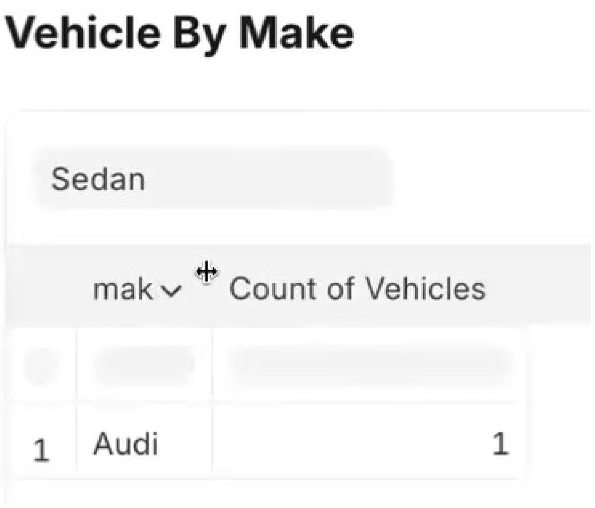
If we want to add flexibility for users to filter the report, we can use `Filters` so they can do it without us having to code it. To access the filter field, use `%(filter_key)s`. `Fieldname` in this case is the fieldname that we can use in the query to grab the filter.



Filters

No.	Label *	Fieldtype *	Fieldname *	Manda...	Options
1	Vehicle Type	Link	vehicle_type	✓	Vehicle Type

```
SELECT make, COUNT(*) AS "Count of Vehicles"
FROM tabVehicle
WHERE type LIKE %(vehicle_type)s
GROUP BY make
```

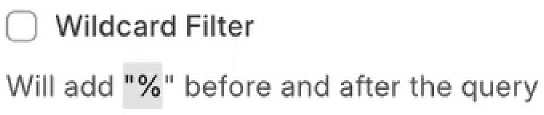


Vehicle By Make

Sedan

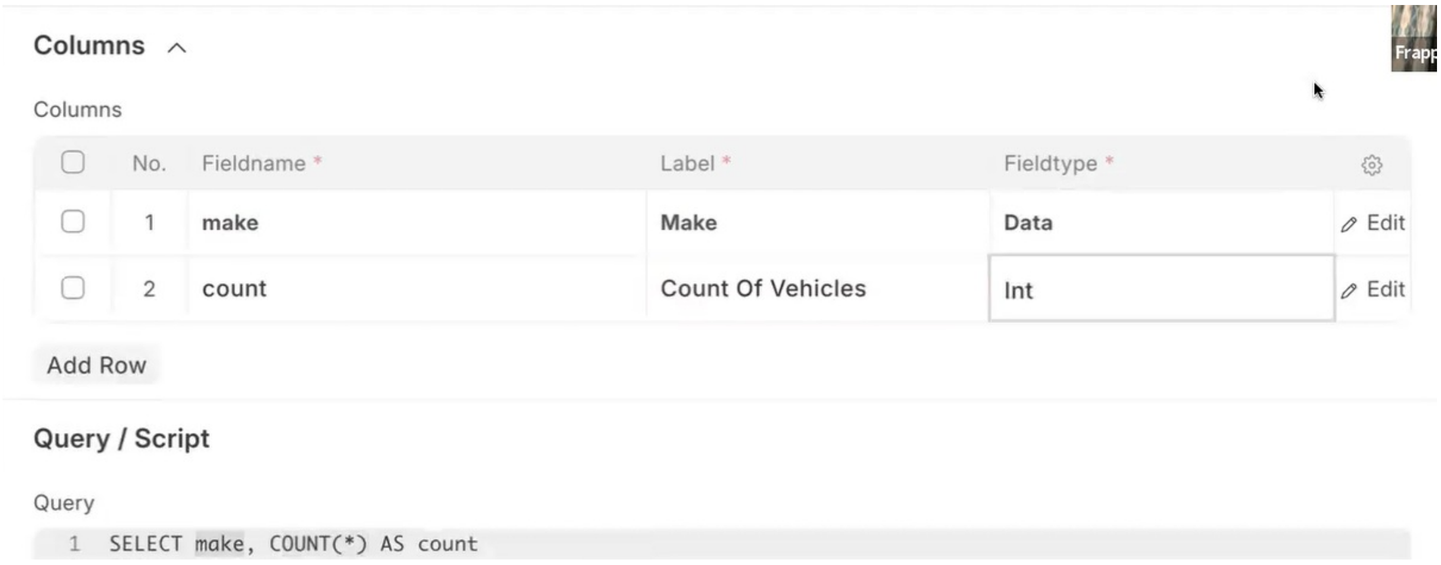
	make	Count of Vehicles
1	Audi	1

If we want the filter to not be exactly the same, there is a `Wildcard Filter` option:


☐ Wildcard Filter

Will add "%" before and after the query

If Frappe does not render our query columns properly, we can use the `Columns` child table to properly format them:


Columns

Columns

☐	No.	Fieldname *	Label *	Fieldtype *	
☐	1	make	Make	Data	Edit
☐	2	count	Count Of Vehicles	Int	Edit

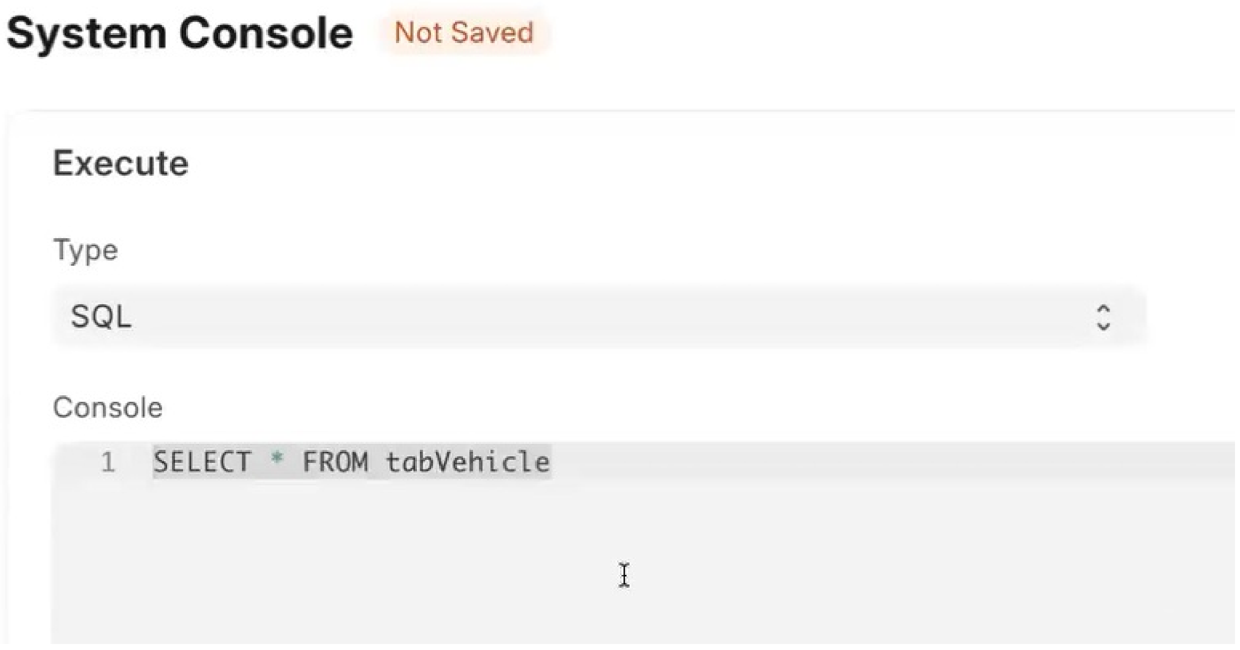
Add Row

Query / Script

Query

```
SELECT make, COUNT(*) AS count
```

There is also a `System Console` on the Desk where we can quickly tests our SQL/Python script:


System Console Not Saved

Execute

Type: SQL

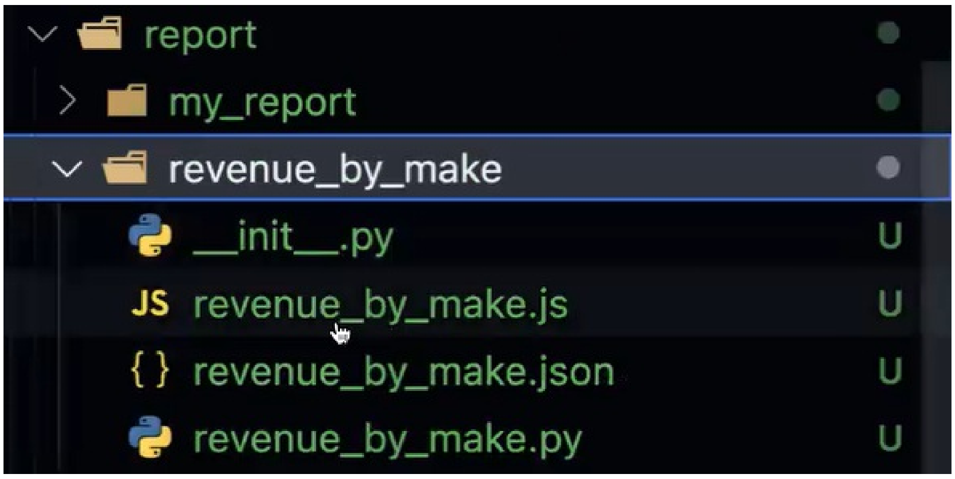
Console

```
SELECT * FROM tabVehicle
```

Script Report → Build report through script (js/py). We can even call an API to build the script report. We can also return charts & report_summary.

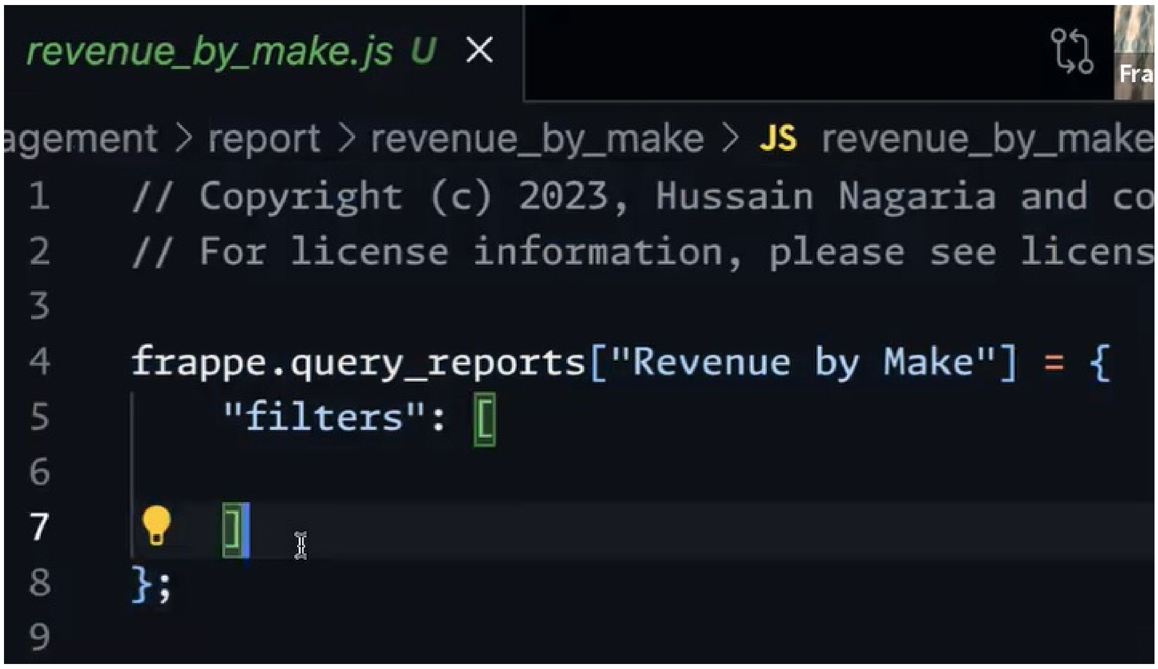
Files Generated (Script Report)

When we create a standard script report, a folder of that report will be generated with 3 files inside:



```
report
  my_report
  revenue_by_make
    __init__.py            U
    revenue_by_make.js     U
    revenue_by_make.json   U
    revenue_by_make.py     U
```

On the .js file, we can add filters, custom buttons, do formatting etc.

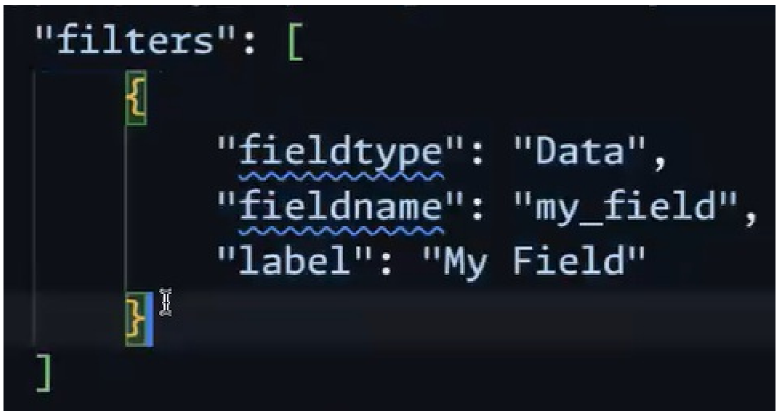


revenue_by_make.js U

agement > report > revenue_by_make > JS revenue_by_make

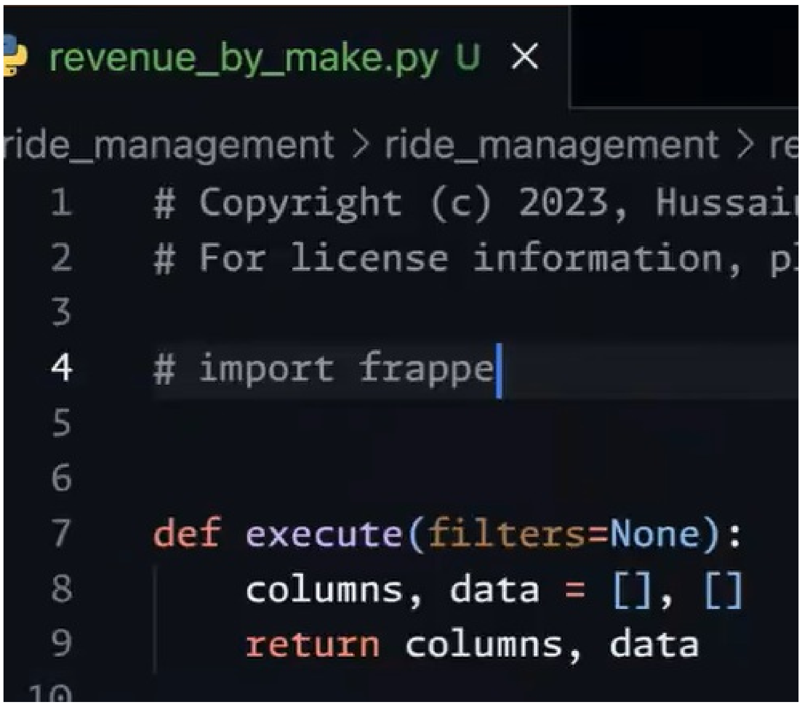
```
// Copyright (c) 2023, Hussain Nagaria and co
// For license information, please see licens

frappe.query_reports["Revenue by Make"] = {
    "filters": [

    ]
};

```

To add filters, add a list of dictionaries the same way as adding columns in the .py:



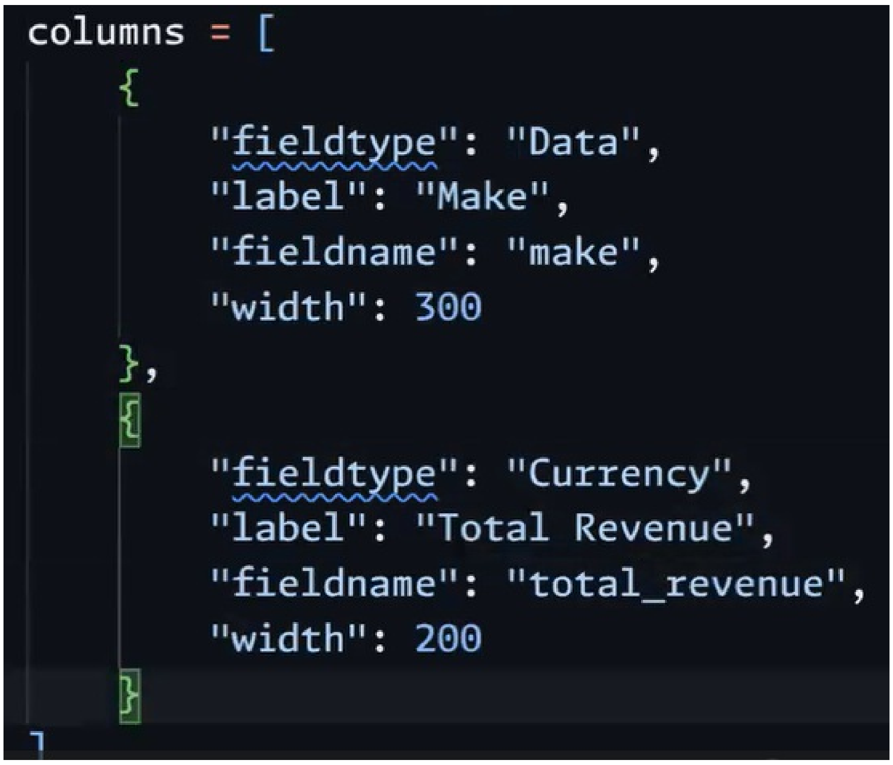
```
"filters": [
    {
        "fieldtype": "Data",
        "fieldname": "my_field",
        "label": "My Field"
    }
]
```

On the .py file, we can do anything that is possible with Python. This file is responsible to return the columns & data of the report.



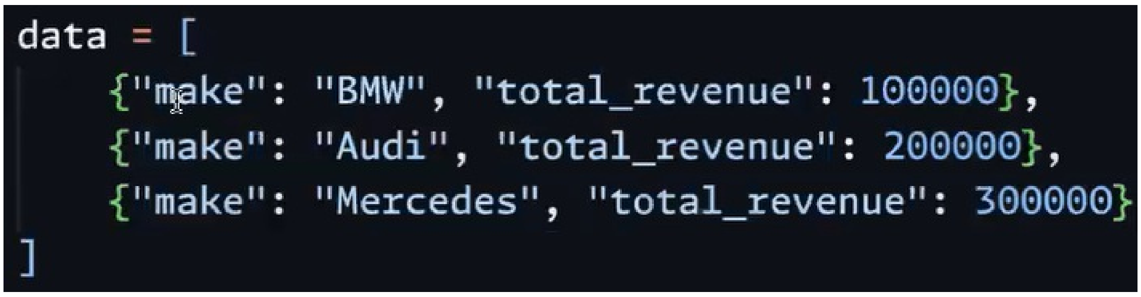
```
# Copyright (c) 2023, Hussai
# For license information, pl

# import frappe


def execute(filters=None):
    columns, data = [], []
    return columns, data
```

Columns must be a list of dictionaries that contains the info of the fields we want to display:



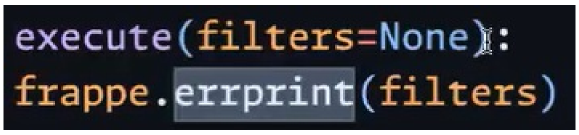
```
columns = [
    {
        "fieldtype": "Data",
        "label": "Make",
        "fieldname": "make",
        "width": 300
    },
    {
        "fieldtype": "Currency",
        "label": "Total Revenue",
        "fieldname": "total_revenue",
        "width": 200
    }
]
```

`data` is also a list of dictionaries. But they key:value pairs should be fieldname:value.



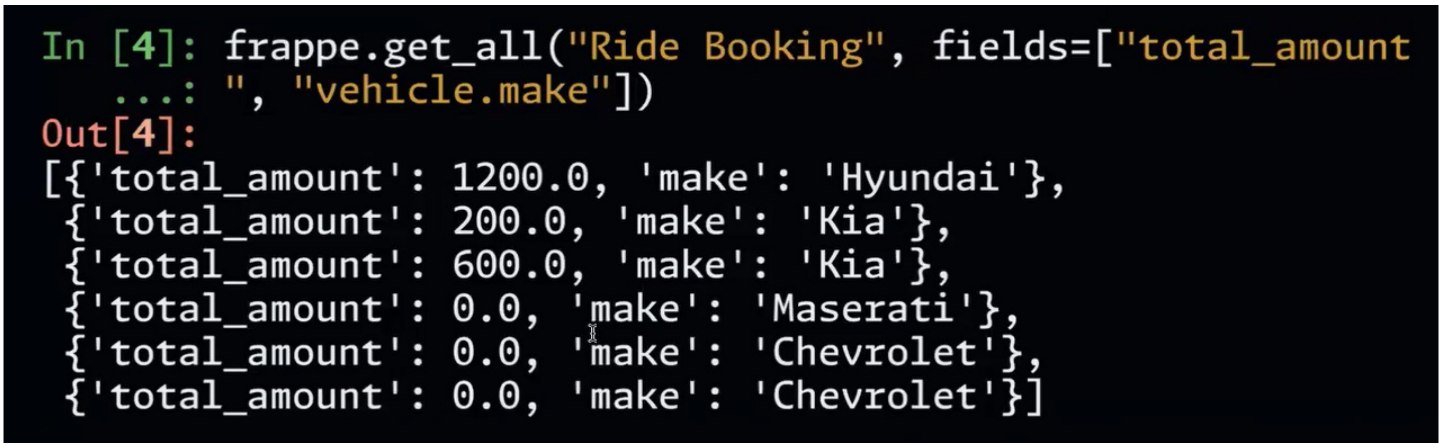
```
data = [
    {"make": "BMW", "total_revenue": 100000},
    {"make": "Audi", "total_revenue": 200000},
    {"make": "Mercedes", "total_revenue": 300000}
]
```

To access the filters by the user, we can use the `filters` variable:



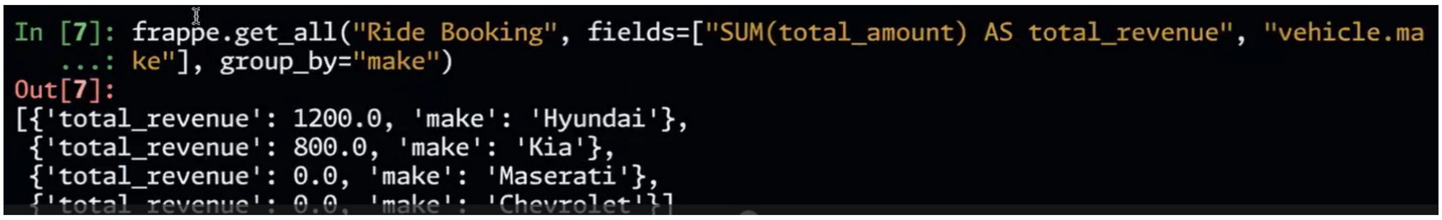
```
execute(filters=None):
frappe.errprint(filters)
```

A nice trick using frappe python API is that we can directly access a link field of a doctype record:



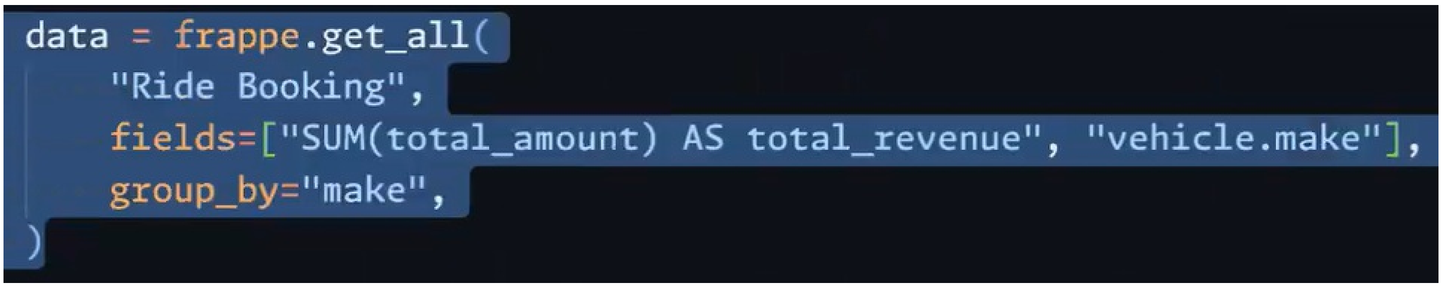
```
In [4]: frappe.get_all("Ride Booking", fields=["total_amount
   ...: ", "vehicle.make"])
Out[4]:
[{'total_amount': 1200.0, 'make': 'Hyundai'},
 {'total_amount': 200.0, 'make': 'Kia'},
 {'total_amount': 600.0, 'make': 'Kia'},
 {'total_amount': 0.0, 'make': 'Maserati'},
 {'total_amount': 0.0, 'make': 'Chevrolet'},
 {'total_amount': 0.0, 'make': 'Chevrolet'}]
```

We can treat it somewhat like SQL query as well:



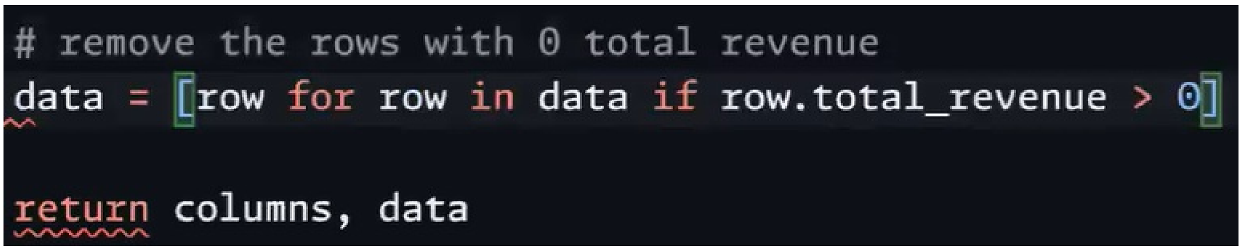
```
In [7]: frappe.get_all("Ride Booking", fields=["SUM(total_amount) AS total_revenue", "vehicle.ma
   ...: ke"], group_by="make")
Out[7]:
[{'total_revenue': 1200.0, 'make': 'Hyundai'},
 {'total_revenue': 800.0, 'make': 'Kia'},
 {'total_revenue': 0.0, 'make': 'Maserati'},
 {'total_revenue': 0.0, 'make': 'Chevrolet'}]
```

We can now use it as the data in our report:



```
data = frappe.get_all(
    "Ride Booking",
    fields=["SUM(total_amount) AS total_revenue", "vehicle.make"],
    group_by="make",
)
```

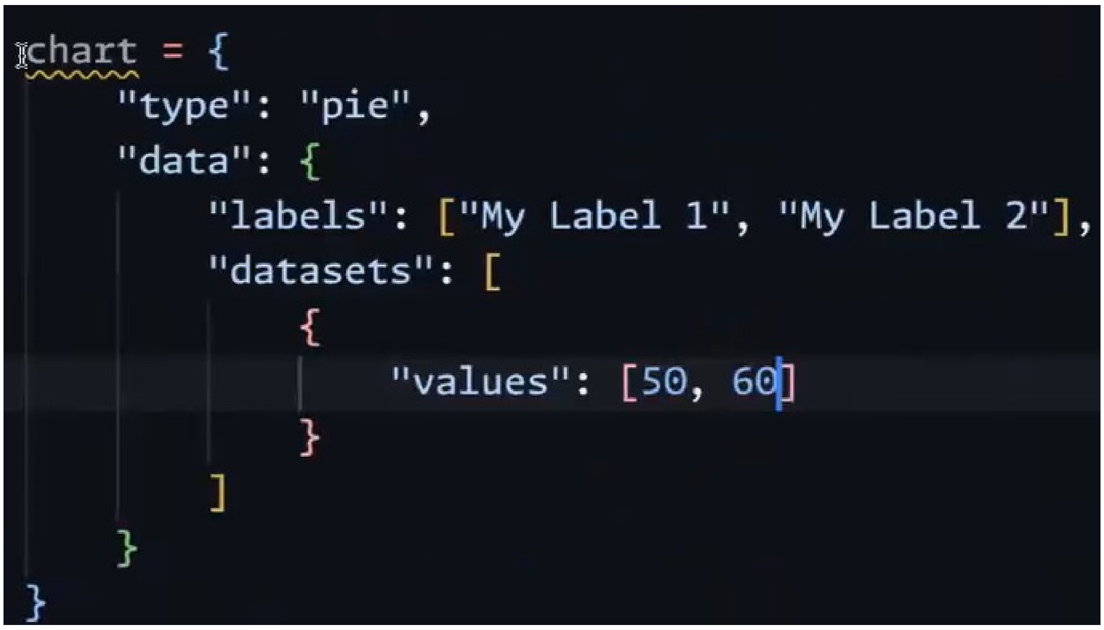
And we can manipulate it using Python:



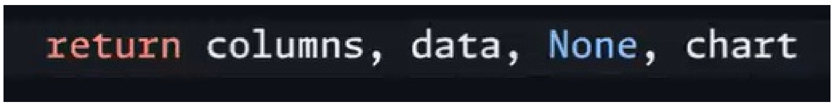
```
# remove the rows with 0 total revenue
data = [row for row in data if row.total_revenue > 0]

return columns, data
```

Reports are also searchable through AwesomeBar.

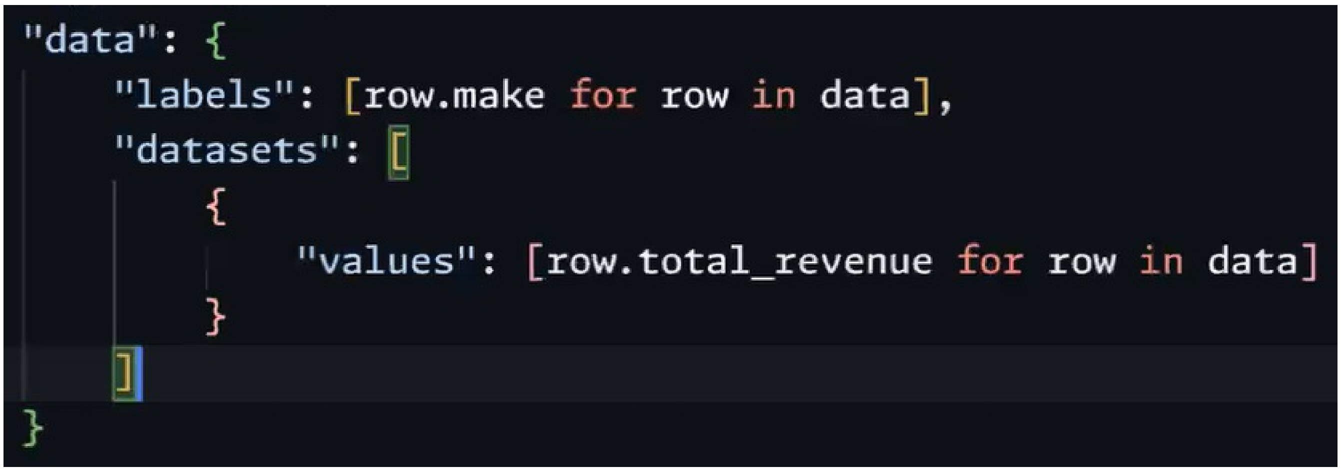
We can also return `chart` on our script report. Behind the scene, frappe uses `Frappe Charts` to render charts.


```
chart = {
    "type": "pie",
    "data": {
        "labels": ["My Label 1", "My Label 2"],
        "datasets": [
            {
                "values": [50, 60]
            }
        ]
    }
}
```

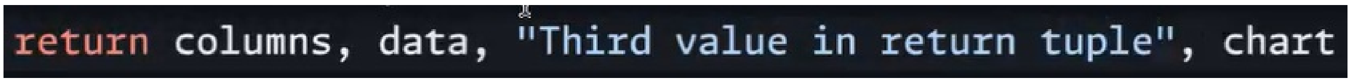
So, we can choose the `type` of the chart (pie, bar, etc.), then we need to give it a `data`, which is a dictionary with 2 keys: `labels` & `datasets`. To render it, just return the chart as the 4th return:


```
return columns, data, None, chart
```

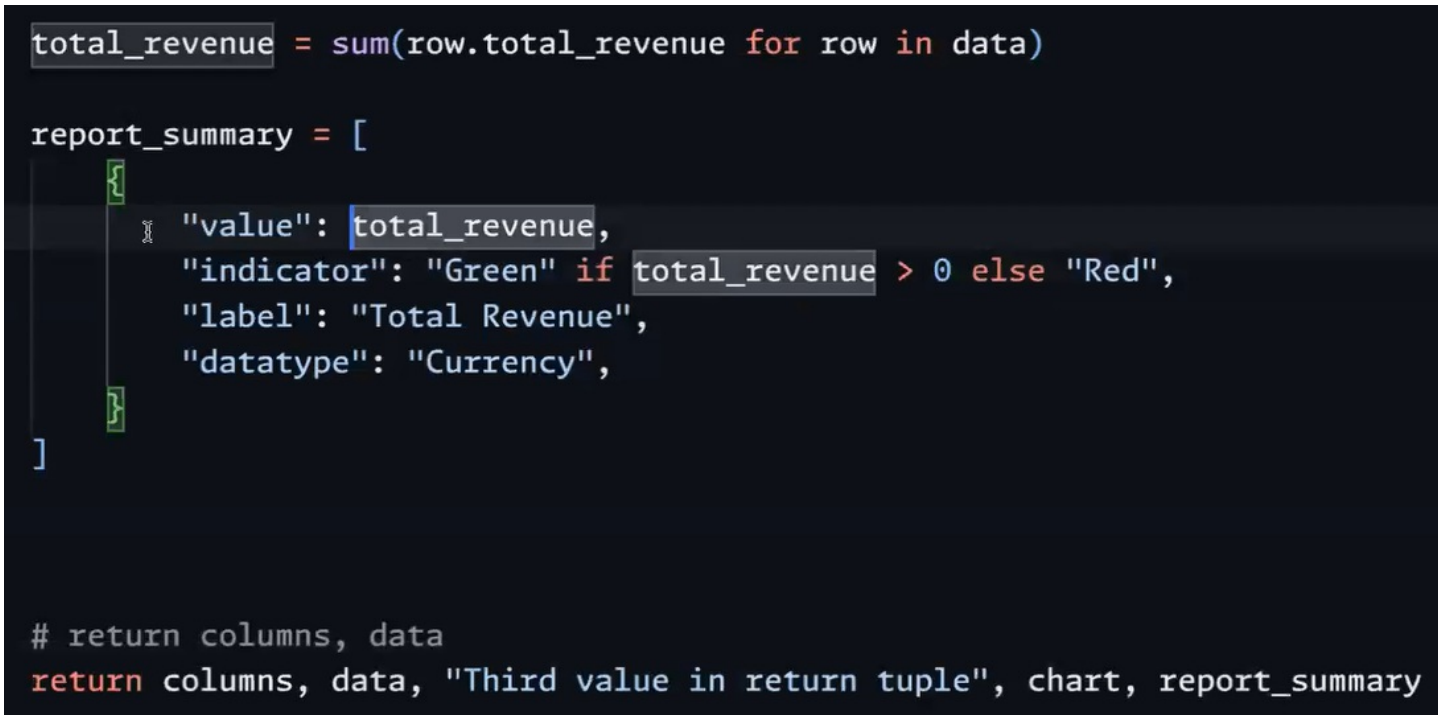
We can give it dynamic data as well:


```
"data": {
    "labels": [row.make for row in data],
    "datasets": [
        {
            "values": [row.total_revenue for row in data]
        }
    ]
}
```

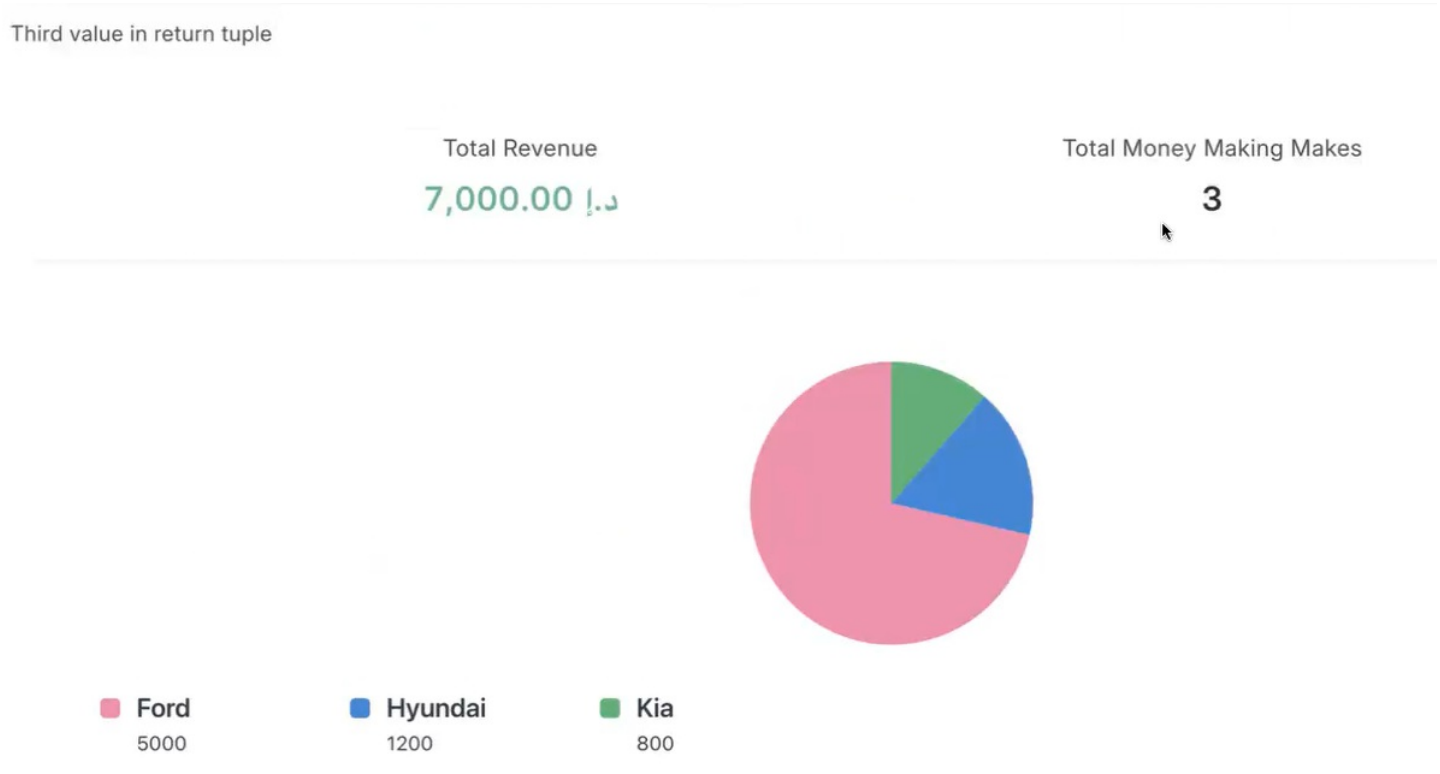
The third value in the return tuple is for the chart title:


```
return columns, data, "Third value in return tuple", chart
```

We can also return a fifth value which is a report_summary list:



```
total_revenue = sum(row.total_revenue for row in data)

report_summary = [
    {
        "value": total_revenue,
        "indicator": "Green" if total_revenue > 0 else "Red",
        "label": "Total Revenue",
        "datatype": "Currency",
    }
]



# return columns, data
return columns, data, "Third value in return tuple", chart, report_summary
```



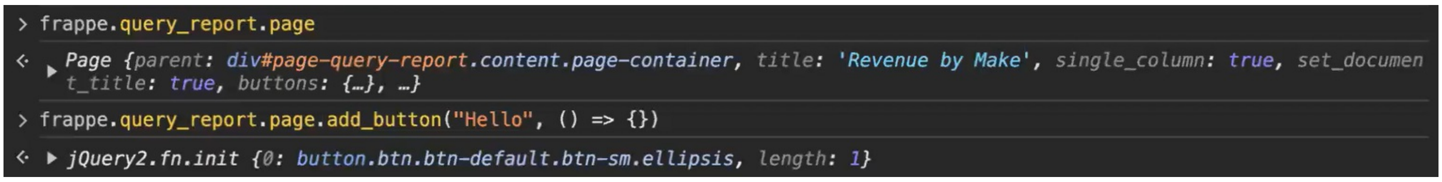
Third value in return tuple

Total Revenue
7,000.00 د.إ

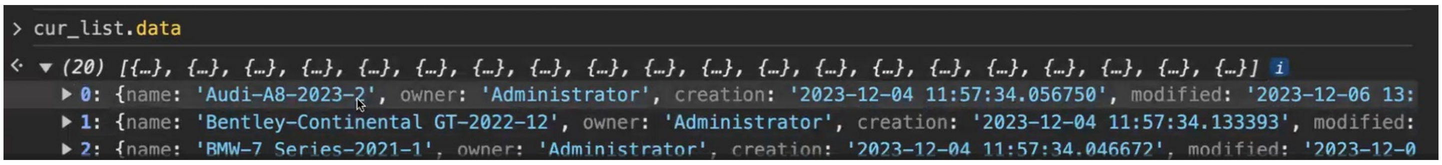
Total Money Making Makes
3

Others:

Other than `cur_frm`, there is also a `cur_list` (for list) and `frappe.query_report.page` (for report)



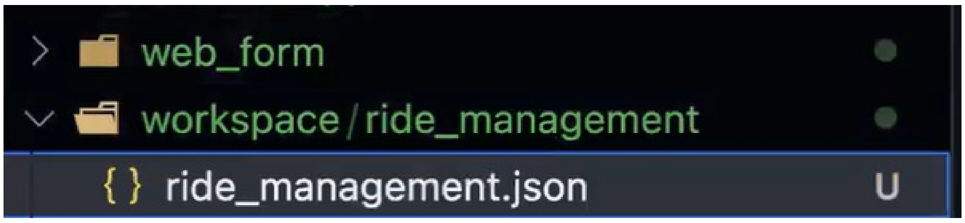
```
> frappe.query_report.page
< ▸ Page {parent: div#page-query-report.content.page-container, title: 'Revenue by Make', single_column: true, set_documen
    t_title: true, buttons: {…}, …}
> frappe.query_report.page.add_button("Hello", () => {})
< ▸ jQuery2.fn.init {0: button.btn.btn-default.btn-sm.ellipsis, length: 1}
```



```
> cur_list.data
< ▾ (20) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}] i
   ▸ 0: {name: 'Audi-A8-2023-2', owner: 'Administrator', creation: '2023-12-04 11:57:34.056750', modified: '2023-12-06 13:
   ▸ 1: {name: 'Bentley-Continental GT-2022-12', owner: 'Administrator', creation: '2023-12-04 11:57:34.133393', modified:
   ▸ 2: {name: 'BMW-7 Series-2021-1', owner: 'Administrator', creation: '2023-12-04 11:57:34.046672', modified: '2023-12-0
```

There is also `set_query` & `get_query` that helps us filter link fields using JavaScript.

If we decide to make a standard workspace, we will see the files generated as well in `workspace` directory:


```
> web_form
v workspace / ride_management
    { } ride_management.json    U
```

There are 2 types of patches:

`pre_model_sync`: runs before fields/properties are added

`post_model_sync`: runs after fields are added