Galyleo

Release 0.5

Andreas Bergen, Mahdi Biazi, Tim Braman, Matthew Hemmings, F

Nov 22, 2022

CONTENTS


galyleo

CHAPTER

ONE

GETTING STARTED

1.1 Overview

Galyleo is a package for the drag-and-drop design and publication of interactive dashboards driven by Jupyter Notebooks. It is available as an extension to JupyterLab, and is also available for other Jupyter enviroments or other programs. A similar dashboard can be found here: elections.

This dashboard is designed entirely in Galyleo:

Galyleo was designed to fit seamlessly into the Jupyter workflow. A Galyleo document is stored on the Jupyter Hub's storage system, and the Galyleo Editor is simply another tab in a standard JupyterLab environment. In other words, it is simply another tool in JupyterLab to complement the Notebook environment.

Galileo, of course, was the inventor of the telescope and the person who first say Jupiter's moons; we are a visualization solution and we hope that our users will use our technology to make profound discoveries.

1.1.1 Tutorials and Demos

A number of tutorial and example projects can be found at: Galyleo Examples.

1.1.2 Reporting an issue

Please use the bug-reporting button on the Galyleo toolbar to report an issue to us, or write galyleo_support@engagelively.com

1.1.3 Frequently Asked Questions

1. Is Galyleo free and open-source?

Yes. Galyleo is released under a BSD 3-clause license.

2. Can I use Galyleo with a Notebook using an R, Julia, or other kernel, or must I use Python?

The connection between the dashboard and the kernel is given by the Galyleo client, which is currently a Python module. So if the kernel can use Python modules, it can be used today. The Galyleo module is very simple, so we expect to implement it in other languages. Of course, it is also open-source, and the protocol is documented; so others are free to implement this as well.

3. Can I use other editors besides the Galyleo editor to edit my dashboard?

Yes. The dashboard's disk and wire format is a JSON document, and the format is specified in the interchange format section.

4. What's the underlying technology for the editor?

The underlying technology is lively.next, an MIT-licensed environment for developing browser-hosted graphical, interactive applications. You can find its repo here: Lively Next:

5. Can I publish my dashboards to the web, or on my local intranet?

We've developed, and are in beta, with a web application which takes the URL for a Galyleo Dashboard document as the parameter dashboard and renders the document in the browser. The page is at https://galyleobeta.engageLively.com/public/galyleo/index.html. That page can be served from any convenient web server, on the web or on an intranet. So publication is as easy as storing the dashboard file on any resource that has an URL (e.g., in a github repo, or for that matter Google Drive with the appropriate permissions).

Following that, on our roadmap is a feature which will drop a complete web page as a directory into your JupyterLab directory, and from there you can export it to any convenient web page.

6. What browsers are required to run Galyleo Studio?

We've tested it on Chrome, Firefox, Edge, and Safari. We believe that a keybnoard and mouse or touchpad are required to get the best use out of Galyleo Studio, so we recommend the use of a computer or netbook to build dashboards. Dashboards can be viewed on any modern browser on any device.

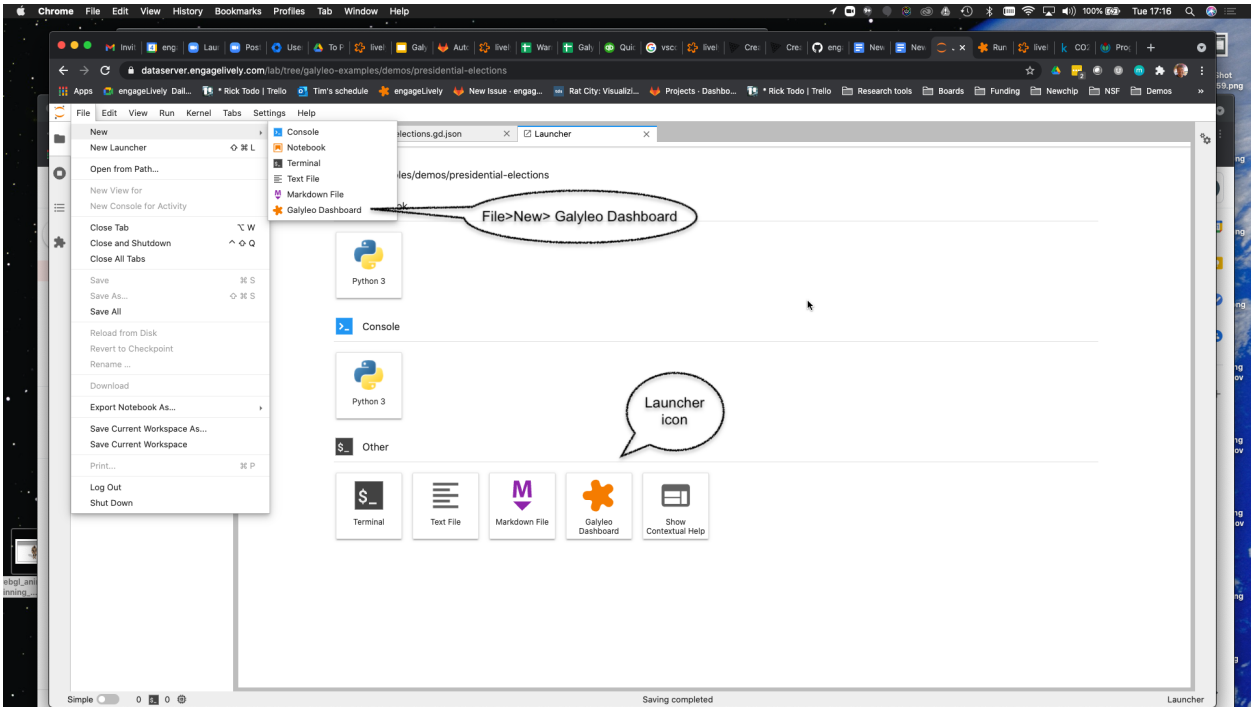
CHAPTER

TWO

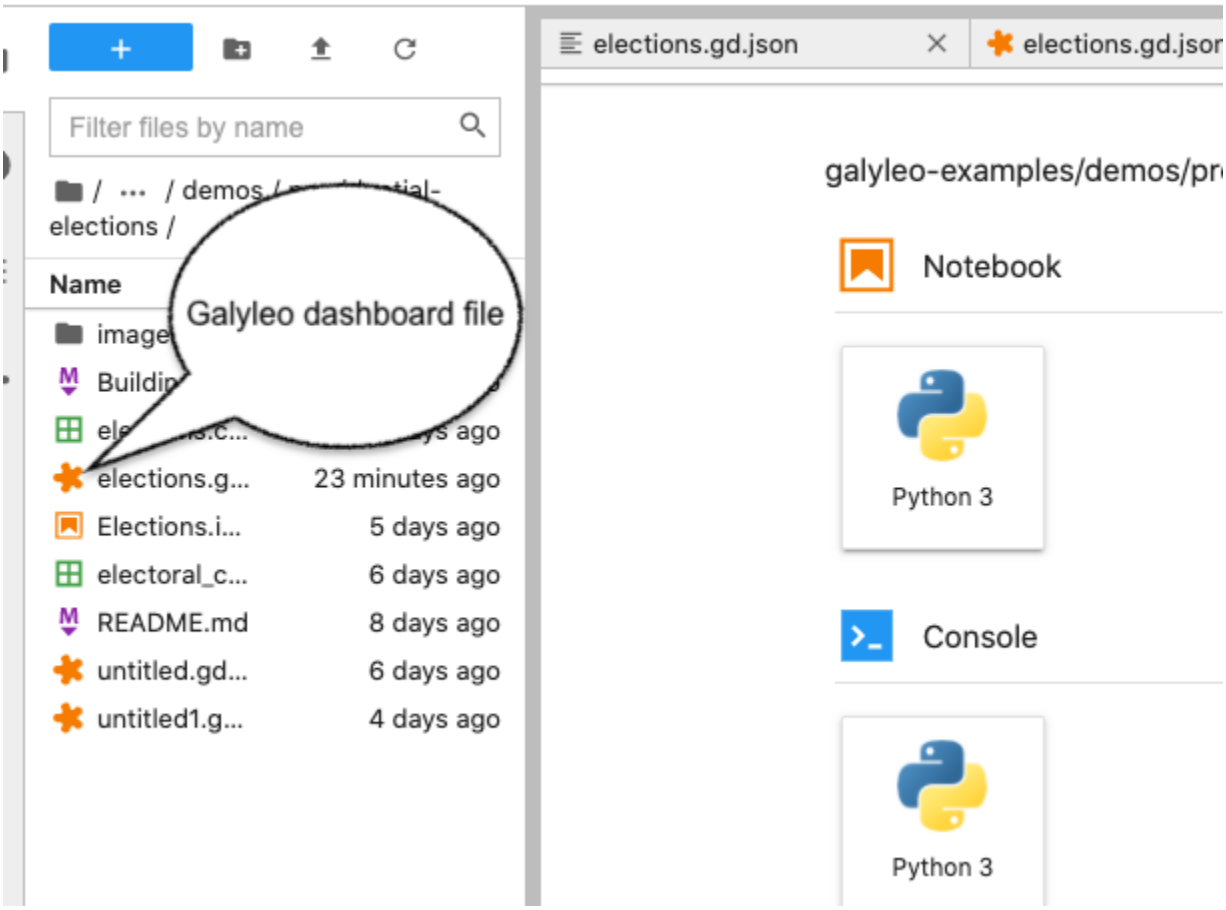
USER GUIDE

2.1 Launching A Dashboard

A new Galyleo Dashboard can be launched from the JupyterLab launcher or from the File>New menu:



An existing dashboard is saved as a .gd.json file, and is denoted with the Galyleo star logo:



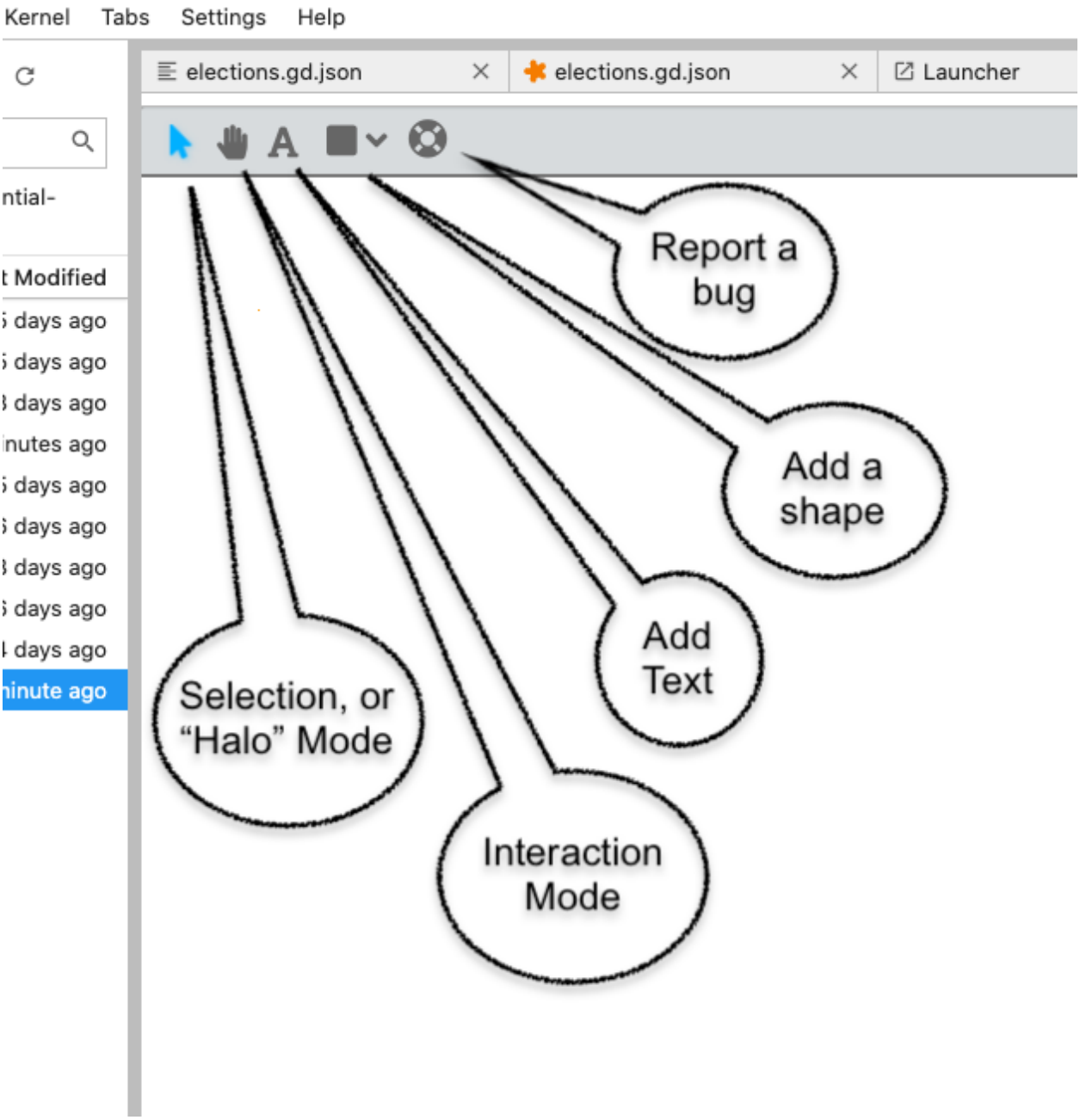
It can be opened in the usual way, with a double-click.

2.2 The Galyleo User Interface

The Galyleo User interface consists of three components: the top bar, the side bar, and the Halo and Context Menu. We discuss each of these in turn. The mission of the Top Bar is to switch between global modes (interacting and selecting) and added non-chart elements (shapes, images, and text) to the dashboard. The Halo and Side Bar is where individual objects are positioned and configured: where shape and text properties are set, borders defined, and images chosen. The Halo permits the copying, deletion, resizing, and rotation rotation, of objects front-to-back. The Context Menu, brought up by a right-click on the object, permits its reordering.

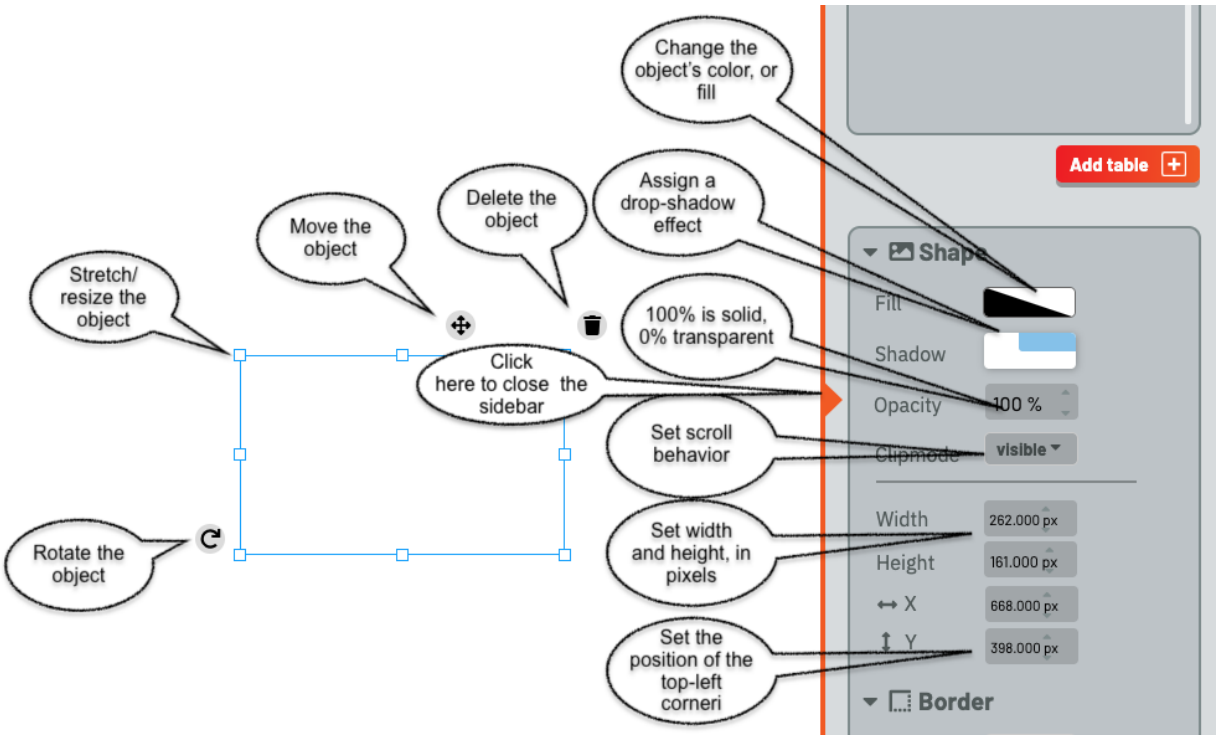
2.2.1 The Top Bar

The top bar controls are in the top left of the dashboard. They are primarily used to choose between selection mode (when the user is designing the dashboard) and interaction mode (when the user is interacting with the dashboard, e.g, manipulating a slider). The practical difference is that when the user is in interaction mode (the arrow is highlighted) a Halo appears over the clicked item and the sidebar is shown; when in interaction mode (the hand is selected) the object is manipulated on click. When Text (the A) is selected, a user click brings up a text box. When a shape is selected (one of Rectangle, Ellipse, Image, or Label) the appropriate shape is drawn in response to a user click. The last item on the top bar is a lifesaver icon, which brings up a bug-report dialog.


Kernel
Tabs
Settings
Help
elections.gd.json
elections.gd.json
Launcher
Report a bug
Add a shape
Add Text
Selection, or "Halo" Mode
Interaction Mode

2.2.2 The Halo and the Side Bar

The Halo and the Side Bar are used to configure an object when it's created, and can also be used to configure physical properties of charts and filters. The Halo automatically appears when an object is clicked on and Selection mode is enabled (the arrow icon in the top bar. The side bar automatically appears when a shape or text is created, or when the knob in the middle of the sidebar is clicked.

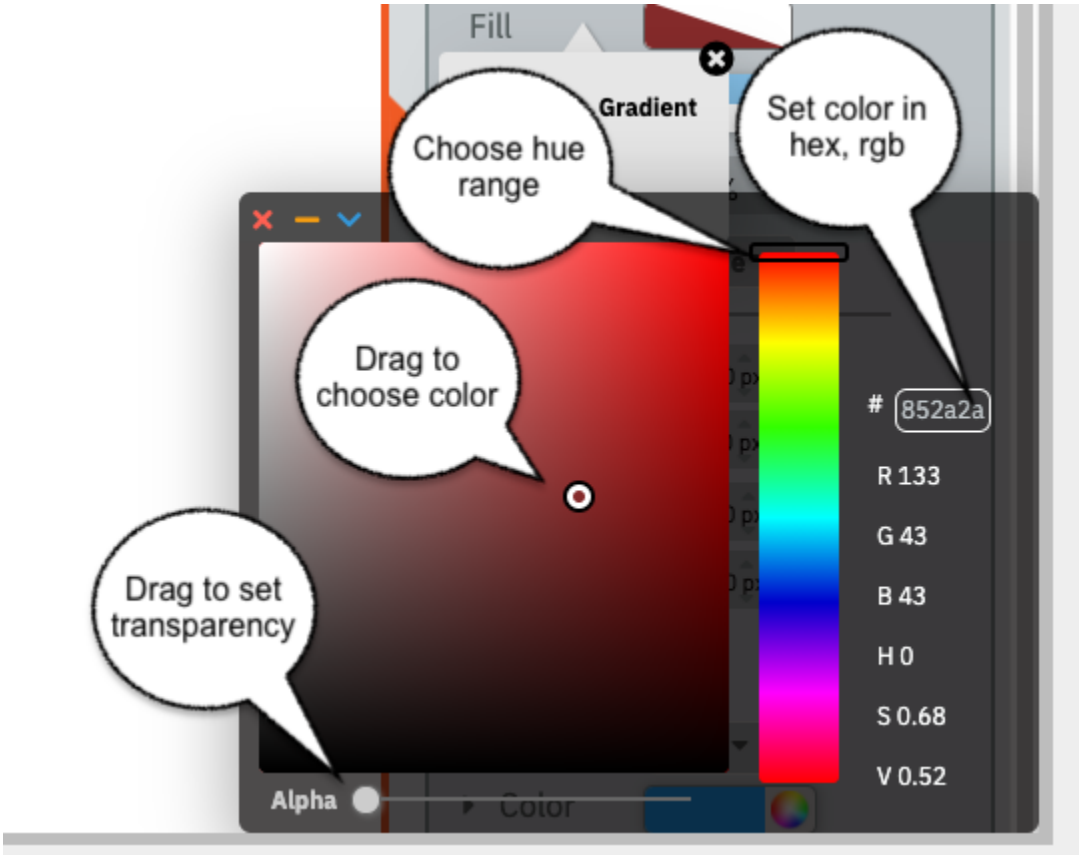


The Halo shows control points and tools around the selected object. The eight control points in the inner halo are used to change the width and height of the object. The tool in the bottom-left corner is used to rotate the object. The cross on the top bar is used to move it. The trash can on the bottom left is used to delete the object.

The Side bar consists of two parts. The top one, which we'll return to later, manages Tables, Filters, Views, and Charts. The bottom part is used to configure objects. It has three sections, each activated by clicking on the chevron next to the name.

The *Shape* configurer is shown in the image, with its eight components shown. They are used to configure the fill (color) and opacity of the object, as well as whether the object casts a drop shadow.

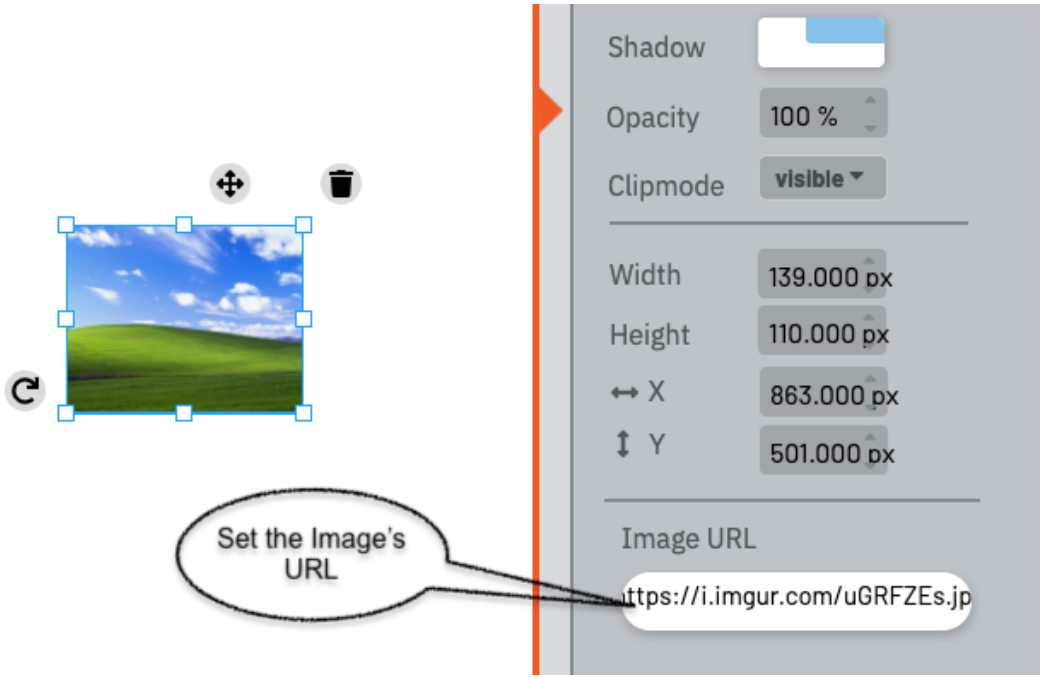
The Color Chooser offers two modes to choose the color of an object. The first, found by clicking the left-hand rectangle, brings up a palette of colors to choose from.



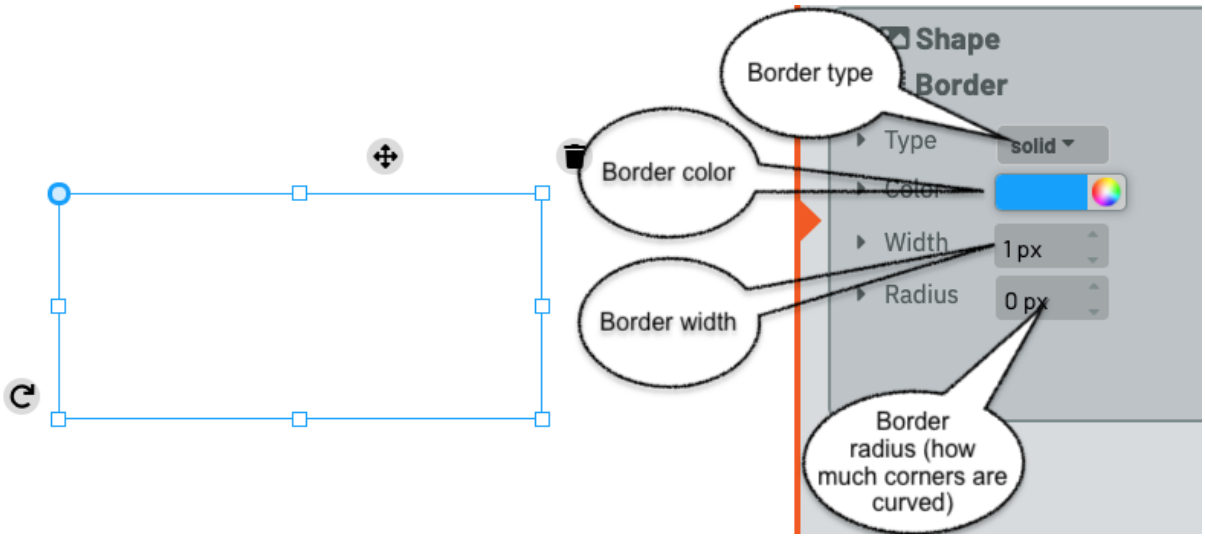
The second permits a fine-grained choice. The Hue bar is used to set the area of the rainbow to pick a color from; dragging the dot in the left-hand square gives the user the ability to choose a specific color. The slider at the bottom controls opacity/transparency of the color (as opposed to the object itself). Finally, the text box gives the user the ability to specify an RGB color, entered as six hex digits.

The "Clipmode" menu gives the user the ability to control what happens when the object is too big for its bounding box. The choices are "visible" (overflows), "hidden" (cut off), "scroll" (scrollbars appear at the right and/or bottom of the object, and "auto" – the system chooses.

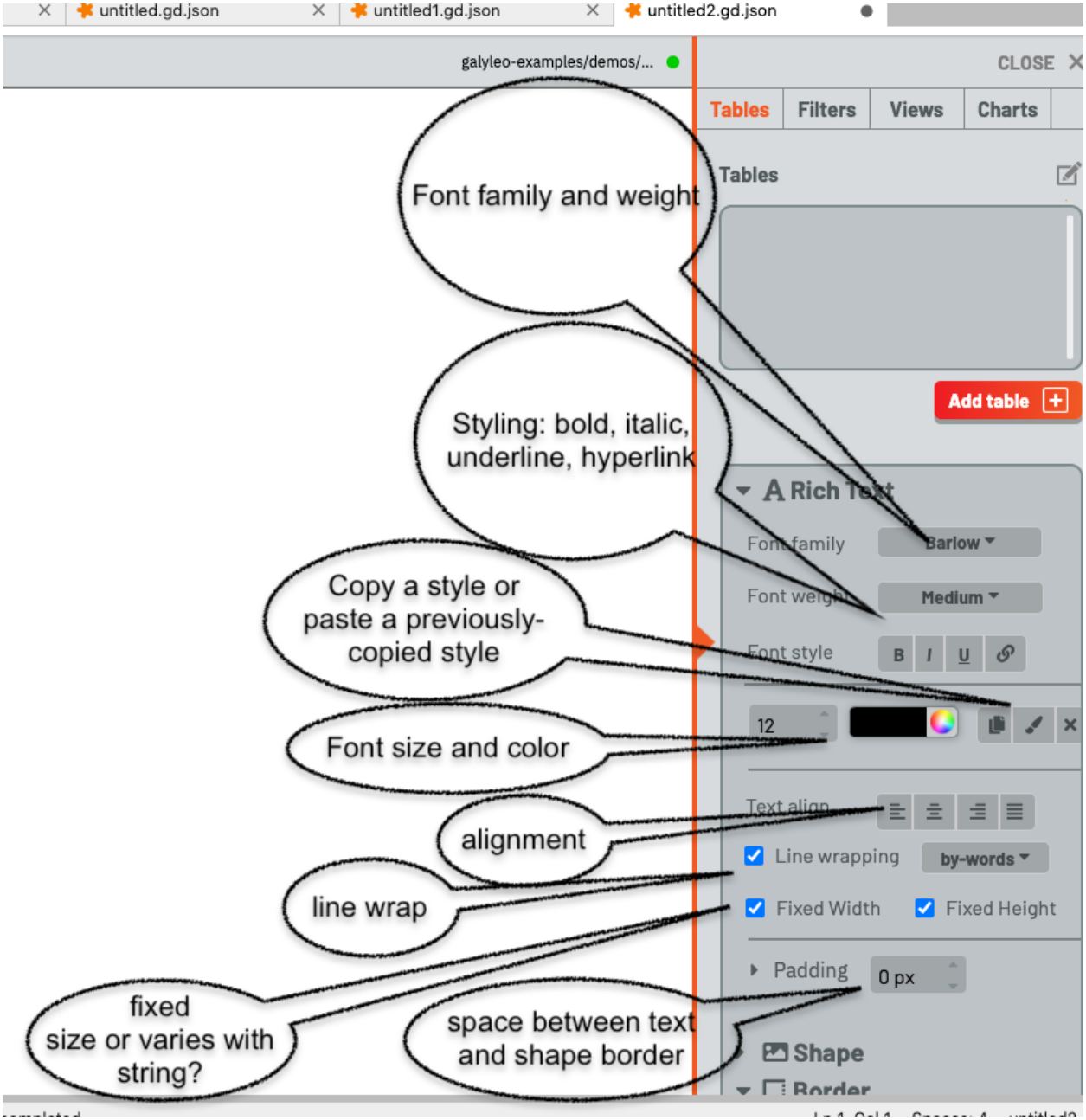
The position is the x,y coordinate of the top-left corner of the object, where (0,0) is the top-left corner of the dashboard; x increases left-to-right, and y top-to-bottom.



When an image is selected, a dialog appears at the bottom of the Shape configurer, permitting the user to choose the URL for the image.



The *Borders* configurer, below the shape configurer, offers the user the ability to control the color, width, and radius of corners, and the type of line that forms the borders (solid, dotted, dashed, ridged, double, groove, or inset). Hidden and none, two other border options, are equivalent to no border.

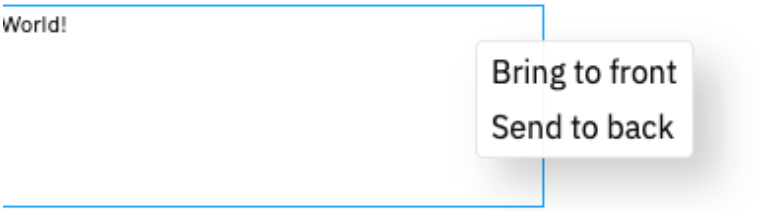


The *Text* controller only appears when a Text item is selected, and it is used to control the textual properties of a text object. These are:

- The font family and weight (fine to extra-bold)
- The font style (bold, italic, underline, hyperlink)
- The font size and color
- Alignment
- Whether and how lines are wrapped (by words, anywhere, only by words (cannot break a word, or by characters)
- Whether the text box size is set by the user or grows and shrinks with the storing
- The padding control gives the spacing between the text boundary and the boundary of the object, in pixels

There are also three buttons, next to the color chooser, which permit a user to copy style (the standard copy icon), paste a copied style (the paste brush), or clear all formatting (the x).

2.2.3 The Context Menu

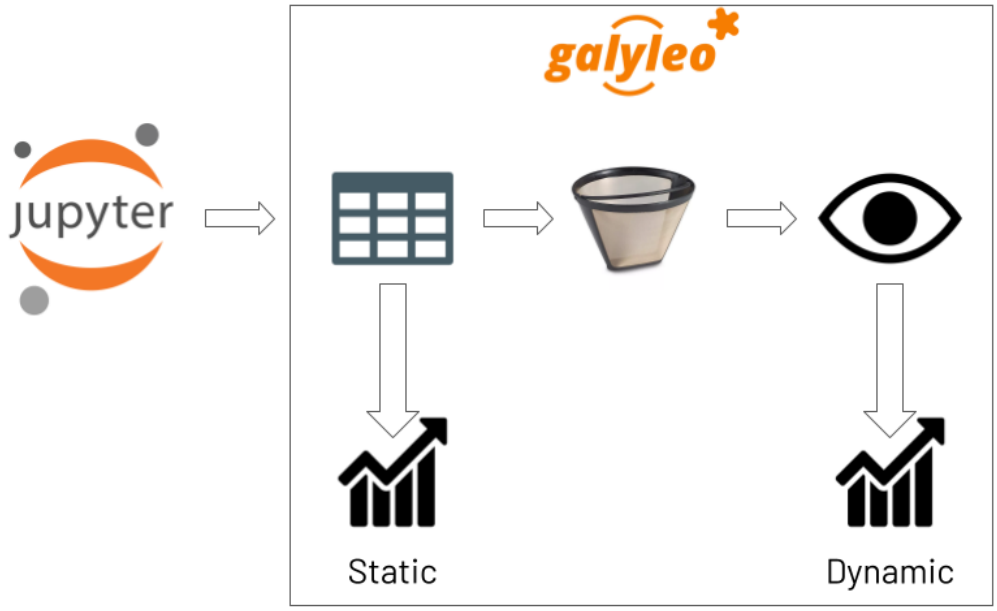


The context menu appears when the object is right-clicked and selection mode is enabled. It controls the ordering of objects, front-to-back.

2.3 Galyleo Data Architecture And Flow

The data flow in Galyleo is shown here. Data is produced in a Jupyter Notebook, and then sent to a dashboard via the Galyleo library. The object sent to a dashboard is a Table, which is conceptually a SQL database table – a list of columns, each with a type, and a list of list of rows, which form the Table's data. The data is then optionally passed through *filters*. A *Filter* is a user interface element that selects a value (or range of values) from a column. This can be used to choose subsets of rows from a particular table, to create what we call a *View*.

A *View* is a subset of a table; a selection (and, potentially, a reordering) of the columns of a table, and a subset of its rows, chosen by one or more Filters. Static charts can take as input *Table*; these charts display the same data, independent of user actions. Dynamic charts take as input a View, which shows the data as filtered by the user through user inputs.



2.3.1 Tables

A Table is equivalent to a SQL database table – a list of columns, each with a type, and a list of list of rows, which form the Table's *data*.* A table has a name, which must be unique among tables and views, a source, a schema, and data. A *schema* is a list of records of the form `{"name": <name>, "type": <type>}`, where `<name>` is the column name and type is the column type, which is chosen from the set `{"number", "string", "boolean", "date", "datetime", "timeofday"}`. These are captured in the `galyleoconstants` library `GALYLEO_STRING, GALYLEO_NUMBER, GALYLEO_BOOLEAN, GALYLEO_DATE, GALYLEO_DATETIME, GALYLEO_TIME_OF_DAY`.

The Table data is a list of lists, where each list is a row of the table. Each row must meet two conditions:

- The entry in column *i* must be of the type of schema entry *i*
- It must have the same length as the schema

Tables are formed using the `GalyleoTable` class in the `galyleo_table` Python module.

Here's a simple example of a Table, which we'll use throughout this tutorial:

Table 1: Cereal Example

name	mfr	type	calories	fiber	rating
100% Bran	N	C	70	10	68.402973
100% Natural Bran	Q	C	120	2	33.983679
All-Bran	K	C	70	9	59.425505
All-Bran with Extra Fiber	K	C	50	14	93.704912
Almond Delight	R	C	110	1	34.384843
Apple Cinnamon Cheerios	G	C	110	1.5	29.509541
Apple Jacks	K	C	110	1	33.174094
Basic 4	G	C	130	2	37.038562
Bran Chex	R	C	90	4	49.120253
Bran Flakes	P	C	90	5	53.313813
Cap'n'Crunch	Q	C	120	0	18.042851
Cheerios	G	C	110	2	50.764999
Cream of Wheat (Quick)	N	H	100	1	64.533816
Maypo	A	H	100	0	54.850917

This is a formatted version of the table. The schema is:

```
[
    {"name": "name", "type": GALYLEO_STRING},
    {"name": "mfr", "type": GALYLEO_STRING},
    {"name": "type", "type": GALYLEO_STRING},
    {"name": "calories", "type": GALYLEO_NUMBER},
    {"name": "fiber", "type": GALYLEO_NUMBER},
    {"name": "rating", "type": GALYLEO_NUMBER}
]
```

And the first data row is:

```
["100% Bran","N","C",70,10,68.402973]
```

2.3.2 Filters

A Filter is a user-interface element that selects rows from tables, based on values from an individual, named column. A *Select* Filter chooses rows whose value in the named column is equal to the filter's value. For example, a Select Filter over the type column in our example whose value is "H" would select rows:

name	mfr	type	calories	fiber	rating
Cream of Wheat (Quick)	N	H	100	1	64.533816
Maypo	A	H	100	0	54.850917

A *Range* filter chooses rows whose value lies between the two values of the filter. For example, a Range Filter over the calories column whose minimum is 50 and whose maximum is 70 would select the rows

name	mfr	type	calories	fiber	rating
100% Bran	N	C	70	10	68.402973
All-Bran	K	C	70	9	59.425505
All-Bran with Extra Fiber	K	C	50	14	93.704912

Range and *Select* specify the functional properties of filters (whether the filter selects a specific value or all values in a range). The physical properties of a filter are dependent on the functional properties of the filter, the data type of the column, and user experience factors. For example, a spinner and a slider are both Select filters over numeric columns, but are very different widgets. At this writing, the *current* set of supported filters are:

Filter	Filter Type	Column Type
List	Select	any
Dropdown	Select	any
Spinner	Select	Number
Slider	Select	Number
Min/Max	Range	Number
Double Slider	Range	Number
Toggle	Select	Boolean

2.3.3 Views

A _View_ is a subset of a table; a selection (and, potentially, a reordering) of the columns of a table, and a subset of its rows, chosen by one or more Filters. While a chart can take as input a _Table_, such a chart wouldn't respond to user inputs (because a user selects the rows he's interested in by adjusting a Filter, and filters only affect the rows in Views). A View is chosen with: - a source table; - a fixed subset (and potential reordering) of columns - a set of filters which select the rows of the table. The filters are considered to have acted in sequence, and thus the rows preserved are the logical AND of the applied filters. For example, suppose we wanted to construct a View with columns name, rating from our table, and had a range filter on column calories and a select filter on column mfr. The View would be:

```
{
  "table": "cereal",
  "columns": ["name", "rating"],
  "filters": ["mfrFilter", "calorieFilter"]
}
```

And, if mfrFilter was set to "N" (Nabisco) and calorieFilter to [50, 90], the data in the view would be:

name	rating
100% Bran	68.402973

2.3.4 Charts

Charts are, well, charts. Each chart takes its input data from a View or a Table. The category, or X axis (place on geocharts, X axis on column charts or line charts, Y axis on bar charts, wedge labels on donut or pie charts) is the first column in the view or table. This is why an important part of constructing a view is reoordering columns. The current set of Chart types supported by Galyleo are Google Charts; however, we intend to extend these chart types in the near future, to include OpenLayers, Leaflet.js, Chart.js, Cytoscape.js, and others. It is the intent of the Galyleo system that *any* JavaScript/HTML5-hosted charts be available under Galyleo.

2.3.5 Charts as Filters

One common operation in Dashboards is to use Charts as filters. This enables drill-down and detail operations on particular categories. Consider, for example, a table that gives average rating by manufacturer on the cereals example, where the data is shown on the dashboard as a column chart. What we'd like is to see the detailed rating, by cereal brand, on another chart, filtered by manufacturer, and when the user clicks on the bar for a particular manufacturer on the average-rating chart the detail for that manufacturer is shown on the detail-rating chart. Or consider the Presidential Election database example; when we click on a state, we see the vote for that state for the chosen year and the voting history for that state. In both these cases, the chart is being used as a filter; it selects the manufacturer for the rating-detail chart and the state in the vote-history and vote-detail charts. This is such a common use case that it is made a feature in Galyleo: every chart is a filter. Specifically, it is a select filter on the category column of the View or Table that is input to the chart. As we'll see below, charts show up in the same UI sections as filters.

2.3.6 Names and Namespaces

References are by name in Galyleo; each object (Table, Filter, View, or Chart) has a name. Since a Chart can take input from a View or a Table, Views and Tables share the same namespace (Data Source) and a Table cannot have the same name as a View. Similarly, since every Chart is also a Filter, Charts and Filters share the same namespace (Data Selectors), and a Chart cannot have the same name as a Filter or another Chart. Objects in different namespaces can share a name. For example, it's quite common for a View and a Chart to share a name, when the View is the data source for the Chart and isn't otherwise used.

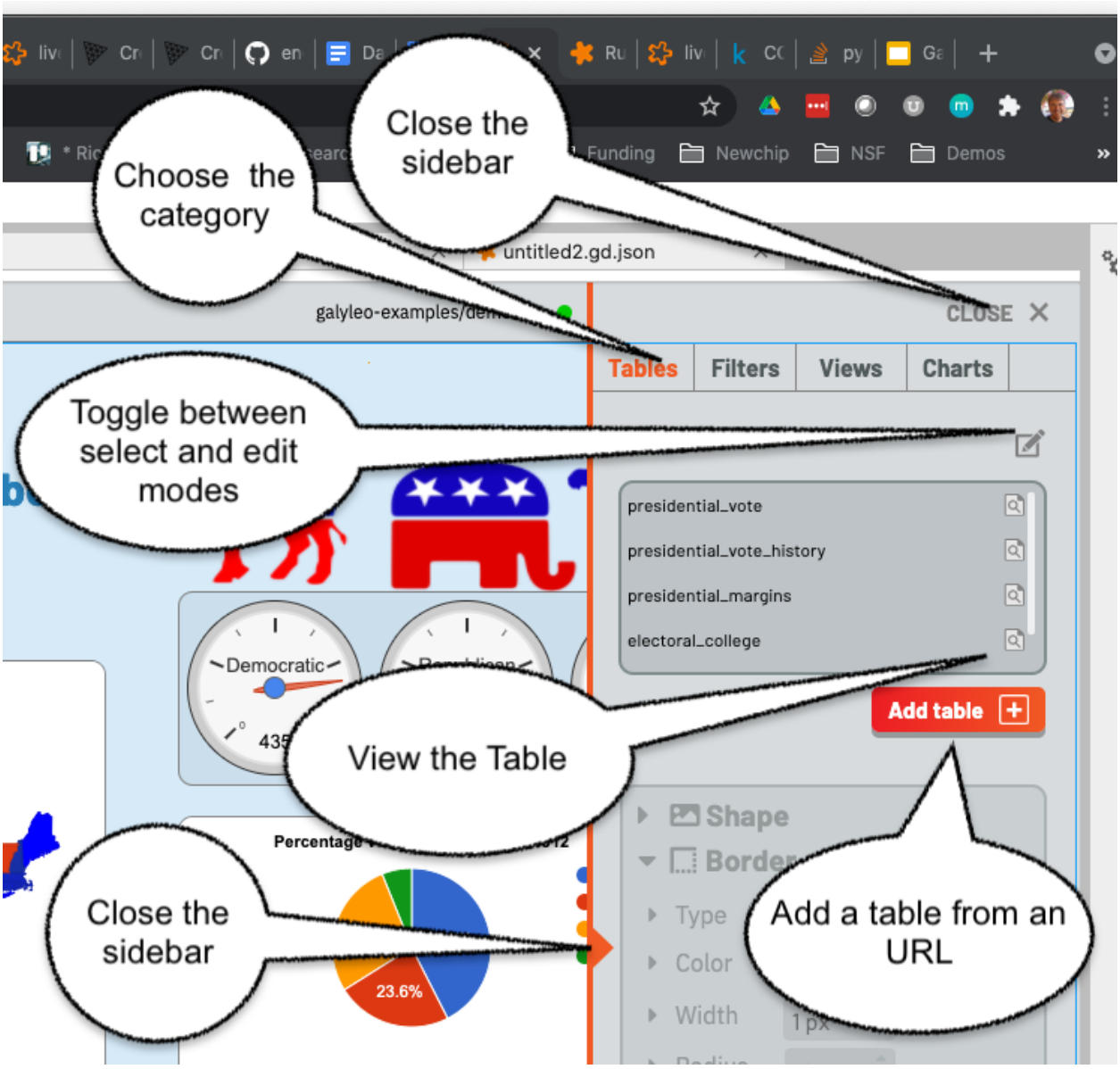
Namespace	Objects
Data Source	Tables, Views
Data Selector	Charts, Filters

2.4 Using Galyleo

This section covers the library and user interface elements for sending Tables from Jupyter Notebooks to Galyleo Dashboards, and using the Galyleo UI to add Filters, Views, Charts, and explanatory elements (Text, Shapes, and Images) to the Dashboard. The UI for Shapes, Images, and Text was largely covered above, so we'll focus on tables, filters, views, and charts here.

2.4.1 The Galyleo UI

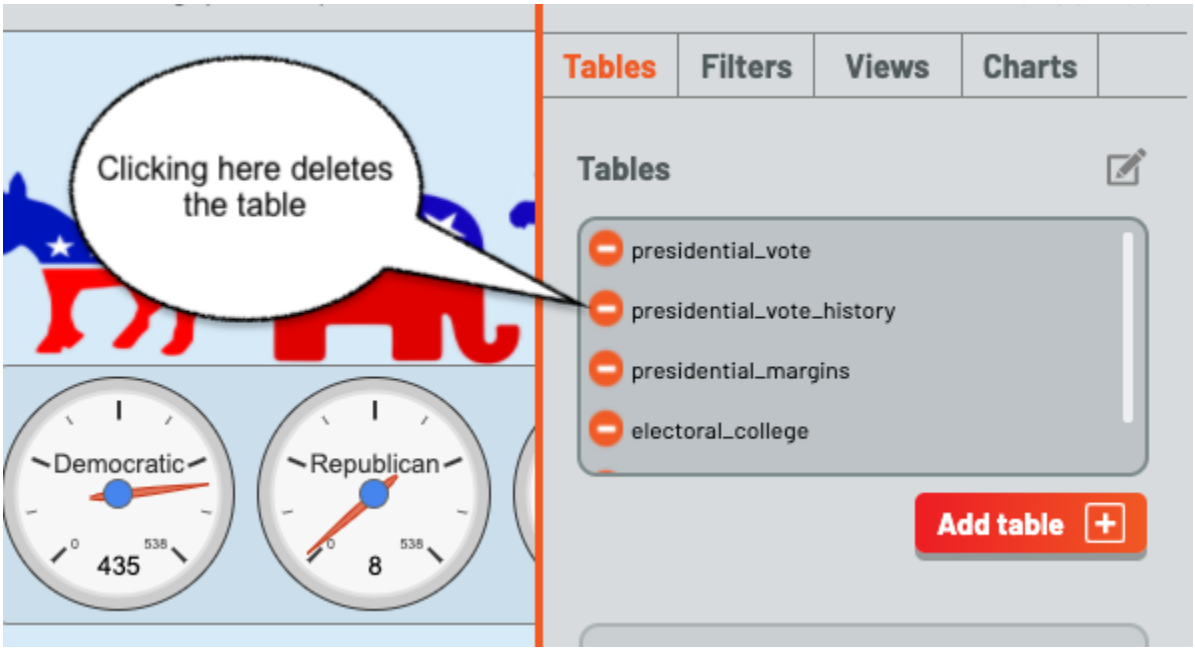
Key elements of the Galyleo UI can be seen in the Tables section of the sidebar, shown here with annotations.



The tab selectors choose the category of item being viewed. Here, it is the list of Tables (the Tables tab is highlighted in orange). To the right of each Table name is an inspection icon. Clicking on this gives a preview of the selected table in a popup window. *Warning:* this should be done carefully, since viewing large tables can cause performance issues. Clicking on the "Add Table" button brings up a popup, inviting a load of a Table in intermediate form from an URL.

The sidebar is closed either by clicking on the orange triangle in the center of the sidebar's left edge, or on the close button on the top right.

Clicking on the pen icon on the top right of the Table list toggles between inspection mode and edit mode.



When in edit mode, clicking on the circle to the left of a table name deletes the circle. Clicking on the pen icon again restores inspection mode.

Every element of the Table UI is present for all classes of element, (Tables, Filters, Views, and Charts). The pen is present in all lists to switch between inspection/configuration mode for all classes, each class has an Add button, and the close-sidebar buttons are always present.

2.4.2 Sending Tables to the Dashboard

The anticipated method of loading a table is to send it from a Notebook. The Galyleo Client document has a detailed description of how to do that. The brief version is to collect the data in a tabular forma, either a list of lists or a Pandas dataframe, create a `GalyleoTable` from the `galyleo.galyleo_table` module, load the data into it, create a `GalyleoClient` from the `galyleo.galyleo_client` module, and then use the `client.send_data_to_dashboard()` method to send the data.

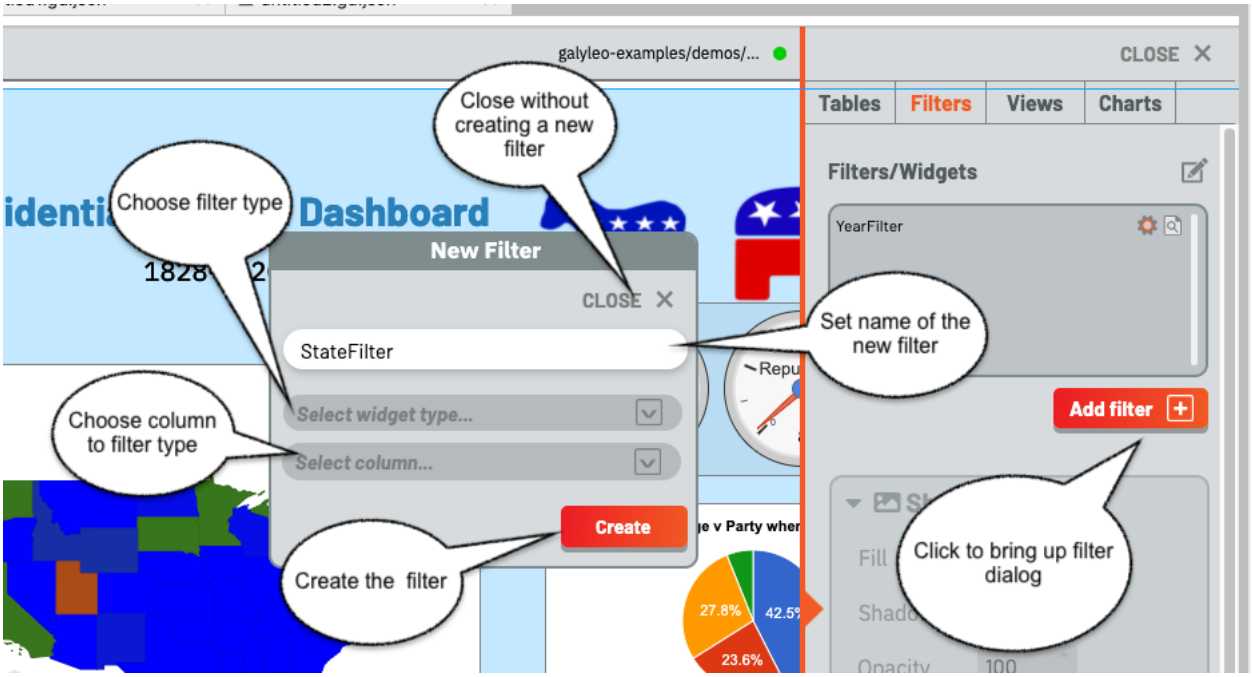
`send_data_to_dashboard` sends data to *open* dashboards in the JupyterLab editor. Data can be sent to a *specific* dashboard by naming it in the call to `send_data_to_dashboard`. Here is a short snippet which sends the cereals data we've used above to a dashboard, assuming the file is in `cereals.csv`:

```
from galyleo.galyleo_jupyterlab_client import GalyleoClient
from galyleo.galyleo_table import GalyleoTable
from galyleo.galyleoconstants import GALYLEO_STRING, GALYLEO_NUMBER
import csv
f = open('cereals.csv', 'r')
reader = csv.csv_reader(f)
data = [row for row in reader][:1]
table = GalyleoTable('cereals')
schema = [("name", GALYLEO_STRING), ("mfr",  GALYLEO_STRING), ("type",  GALYLEO_STRING),␣
↪("calories",  GALYLEO_NUMBER), ("fiber,  GALYLEO_NUMBER), ("rating",  GALYLEO_NUMBER)]
table.load_from_schema_and_data(schema, data)
client = GalyleoClient()
client.send_data_to_dashboard(table)
```

Other methods of loading data and schemas can be found in the documentation for the GalyleoTable class.

2.4.3 Adding a Filter

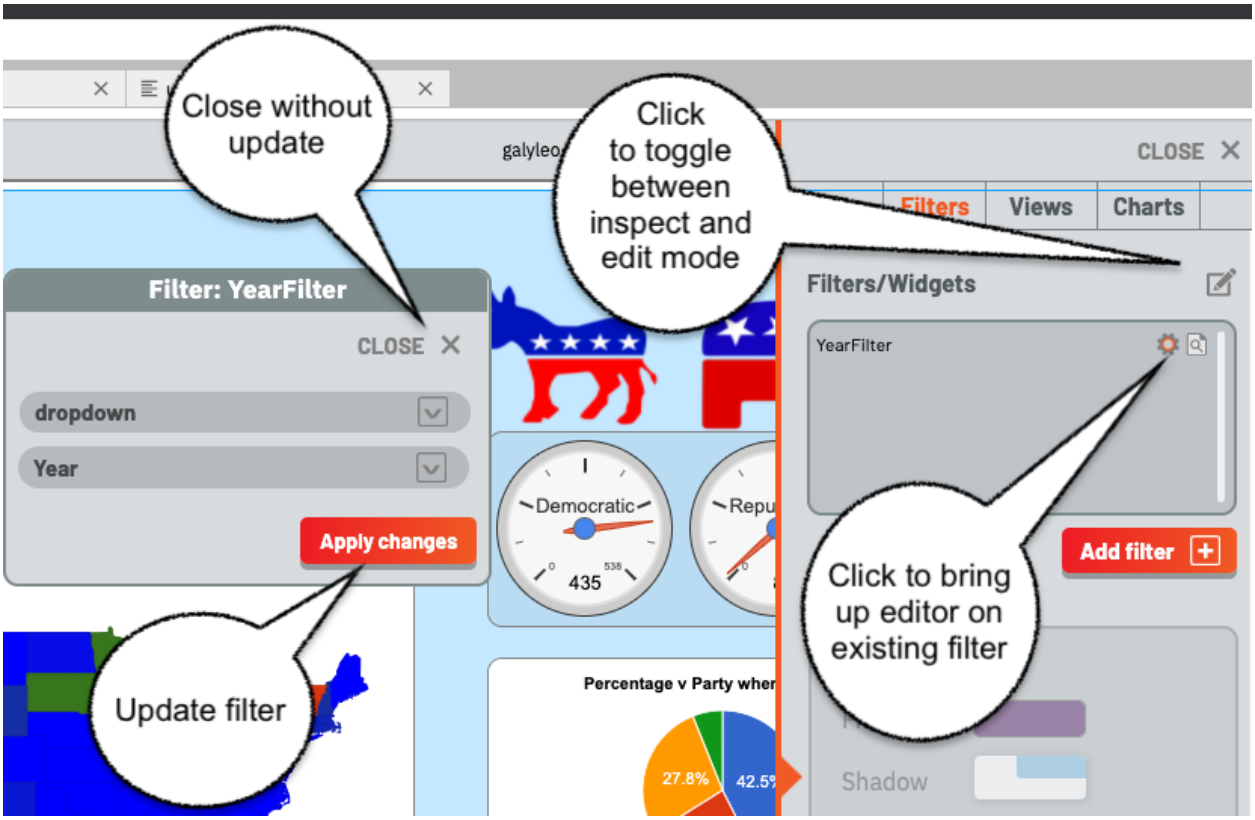
Once tables are in the dashboard, filters can be created and edited. This is done in the Filters tab, found by clicking filters. Once again, there is an Add button below the lower-right corner of the filter list. Click this, and a popup is brought up, permitting the user to create a filter.



The filter must have a name, which cannot be the name of another filter or chart. Type this in the input box, and select a widget type from the upper drop-down and a column name from the lower drop-down, then click create. Clicking "Close" closes the dialog without creating a new filter.

Various errors can occur during this process. In particular, *Range* filters are only valid over numeric columns, and if a mismatched column is selected an error message will appear; the same message appears if a column is not chosen or a widget type is not chosen. An error will display if a name is not entered, or if the name of another filter or chart is chosen.

Once the filter is created, it appears in the top-left corner of the dashboard. The Filter is a physical object, and can be manipulated as with any other physical object on the dashboard, using the Halo and Sidebar as described above. Put the dashboard into selection mode and move the filter as desired.



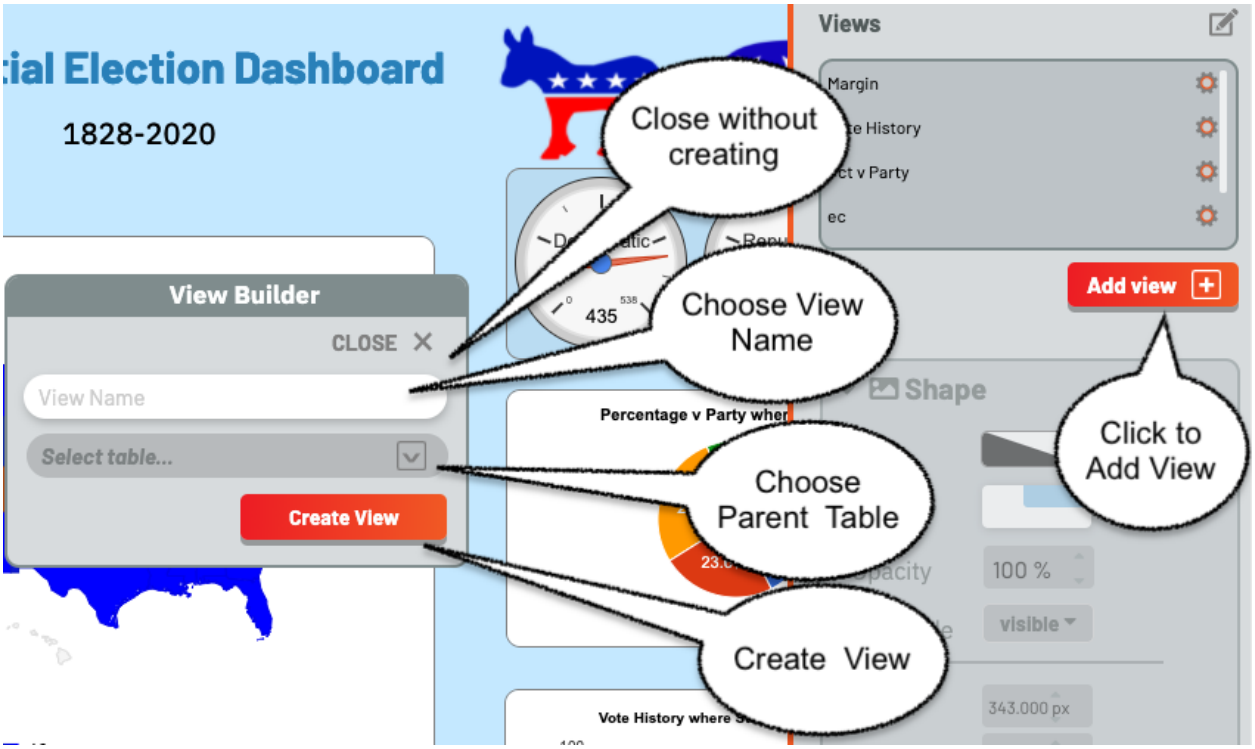
Clicking on the gear icon beside the name of an existing filter brings up a filter editor, as shown here. The filter editor is very similar to the filter creator; it simply lacks an input for the filter name. Choose column and widget, then Apply Changes to update the filter, or Close to close without update.

Notice the pen icon is at the top right; once again, it is used to switch between inspection and edit modes, and filters are deleted in edit mode just as tables are, and with the same icon.

Note: Using the Halo to delete the Filter from the dashboard has the same effect as deleting it from the filter list.

2.4.4 Creating a View

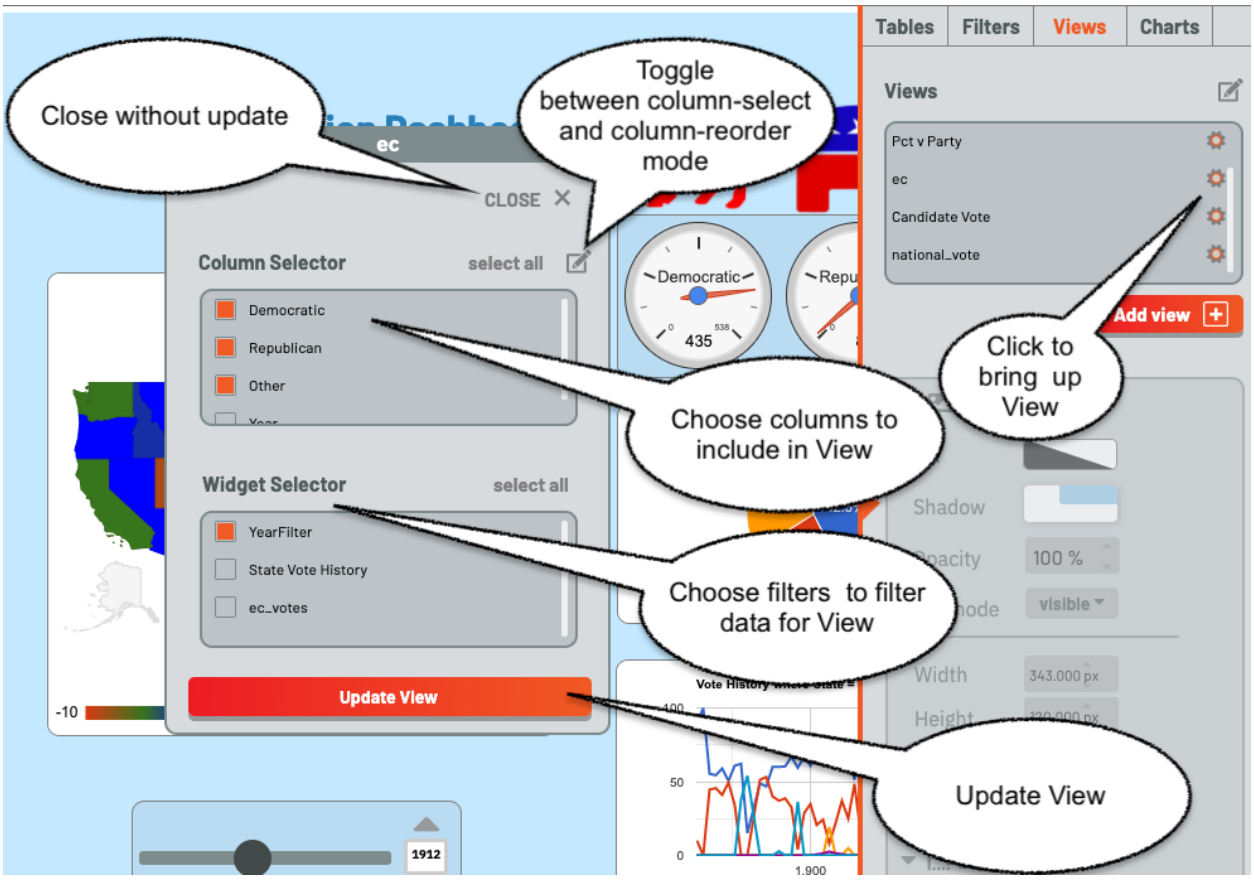
Creating a View is very similar to creating a Filter, under the Views Tab. Once again, there is an Add button below the lower-right corner of the views list. Click this, and a popup is brought up, permitting the user to create a view.



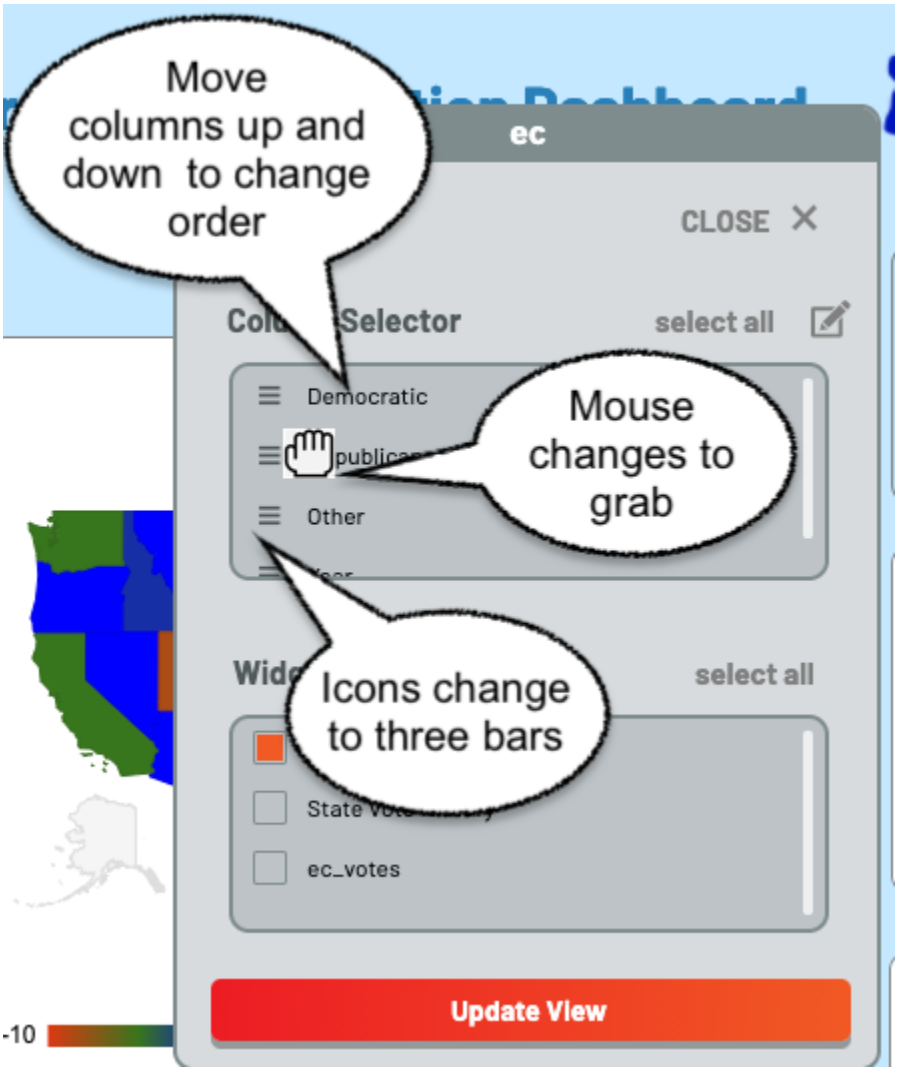
The view must have a name, which cannot be the name of another view or table. Type this in the input box, and choose the underlying table from the drop-down, then click create. Clicking "Close" closes the dialog without creating a new view.

An error will display if a name is not entered, or if the name of another view or table is chosen.

Once a View is created, it is immediately added to the View List, and a View editor is brought up.



The View Editor is also brought up by clicking on the gear icon beside the name of a View. It consists of two panels, a Column Chooser and a Filter Chooser. The Column Chooser chooses the columns for the View, and the Filter Chooser chooses the filters which will be applied to the underlying table to get the rows for the View.

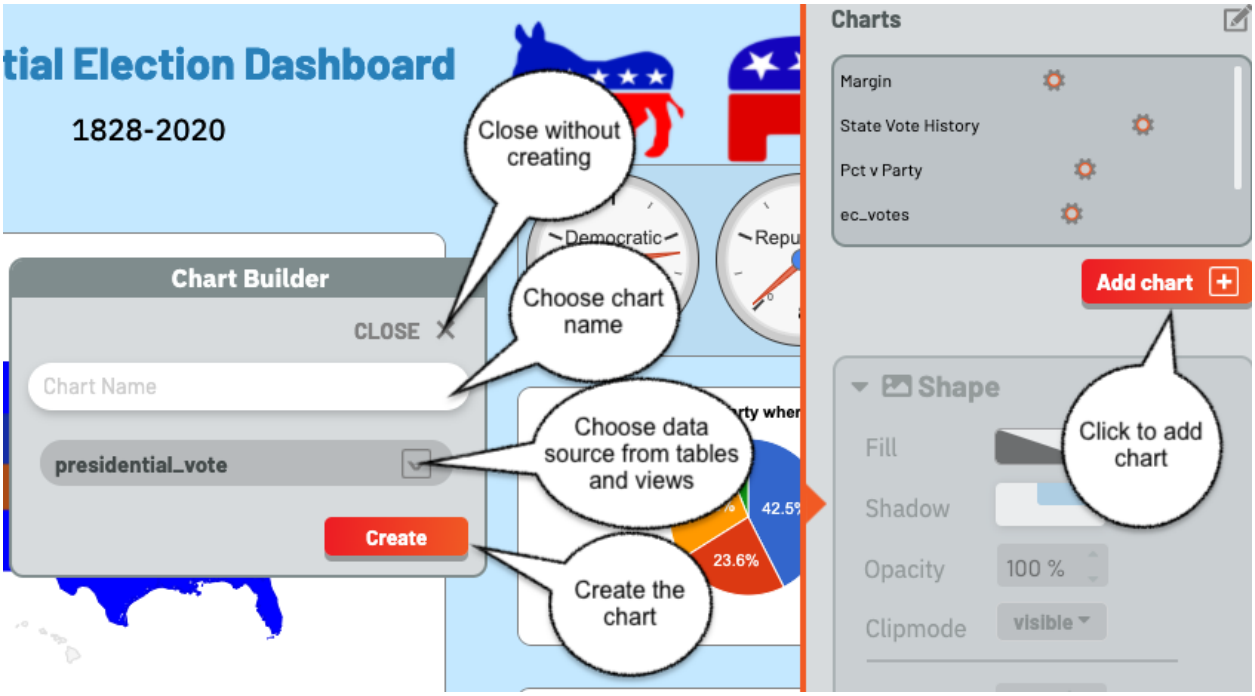


Since column order is important for a View, there is a column-order mode. It is toggled by choosing the pen icon above the Columns list. When it is toggled, the icons beside the column names change to three horizontal bars and the mouse changes to a grab icon. The columns can then be dragged into order with the mouse. Note that while all columns are displayed, only the order of selected columns are important.

As with tables and filters, views can be deleted using the pen icon above the view list to switch to edit mode, then deleting views in the same way tables and filters are deleted.

2.4.5 Creating a Chart

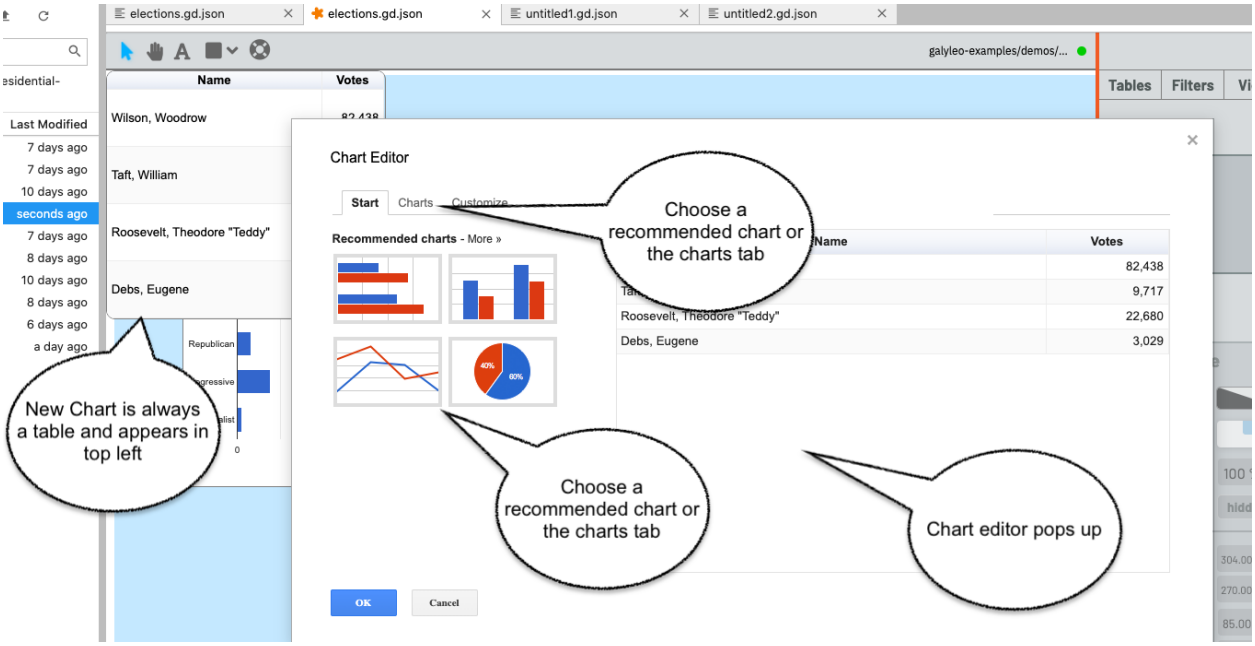
Creating a Chart is very similar to creating a View, under the Charts Tab. Once again, there is an Add button below the lower-right corner of the chart list. Click this, and a popup is brought up, permitting the user to create a chart.



The chart must have a name, which cannot be the name of another chart or filter. Type this in the input box, and choose the view or table to use as a data source from the drop-down, then click create. Clicking "Close" closes the dialog without creating a new chart.

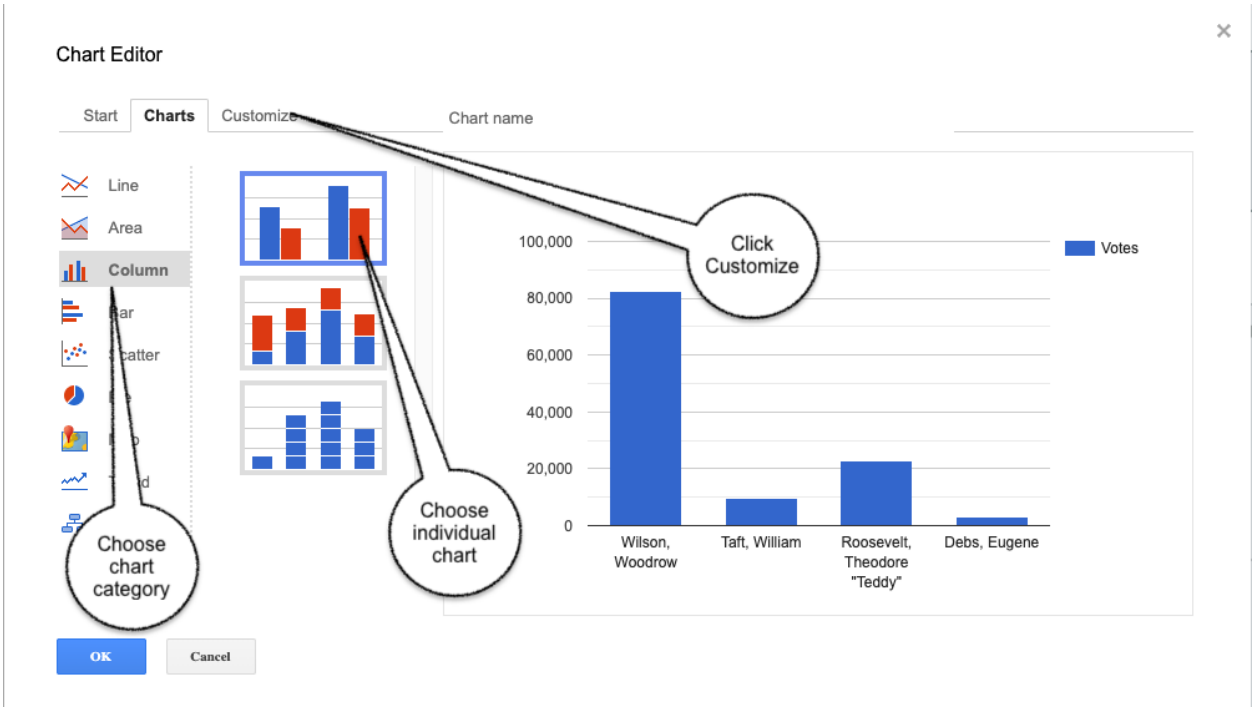
An error will display if a name is not entered, or if the name of another filter or chart is chosen.

Once a chart is created, it is immediately added to the chart List, the chart is brought up as a table on the dashboard, and the Chart Editor pops up.

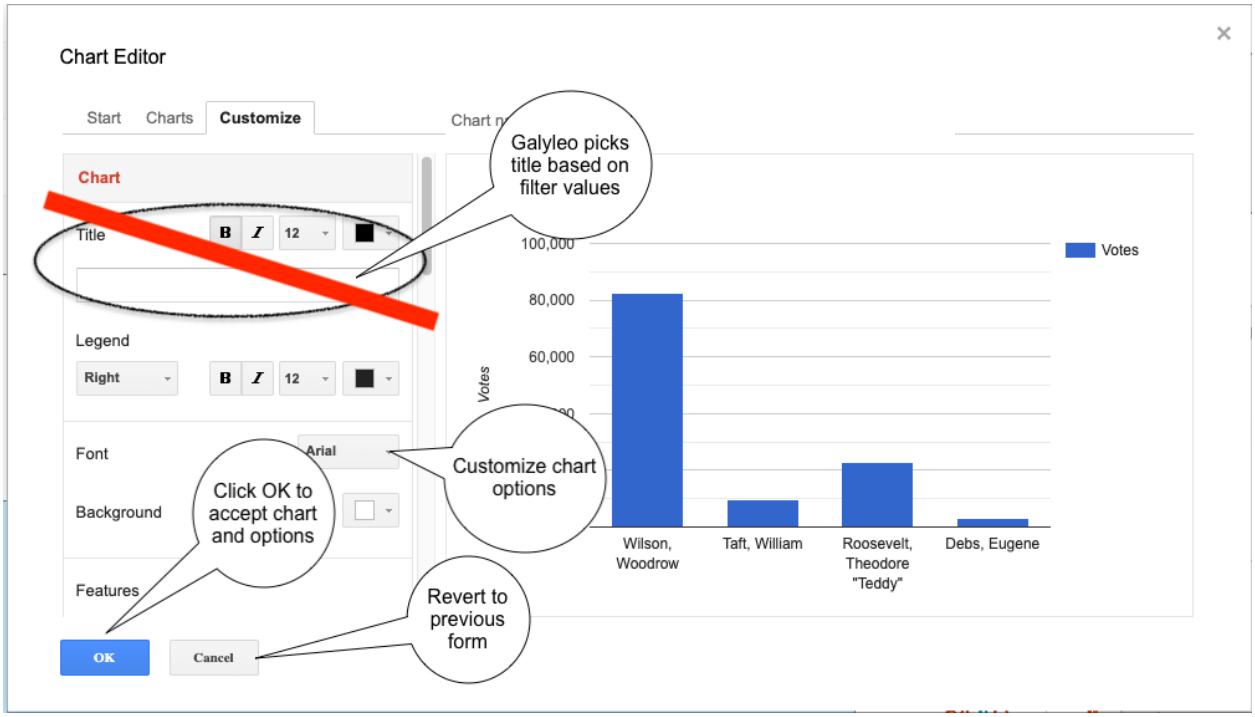


The Chart Editor is also brought up by clicking on the gear icon beside the name of a Chart.

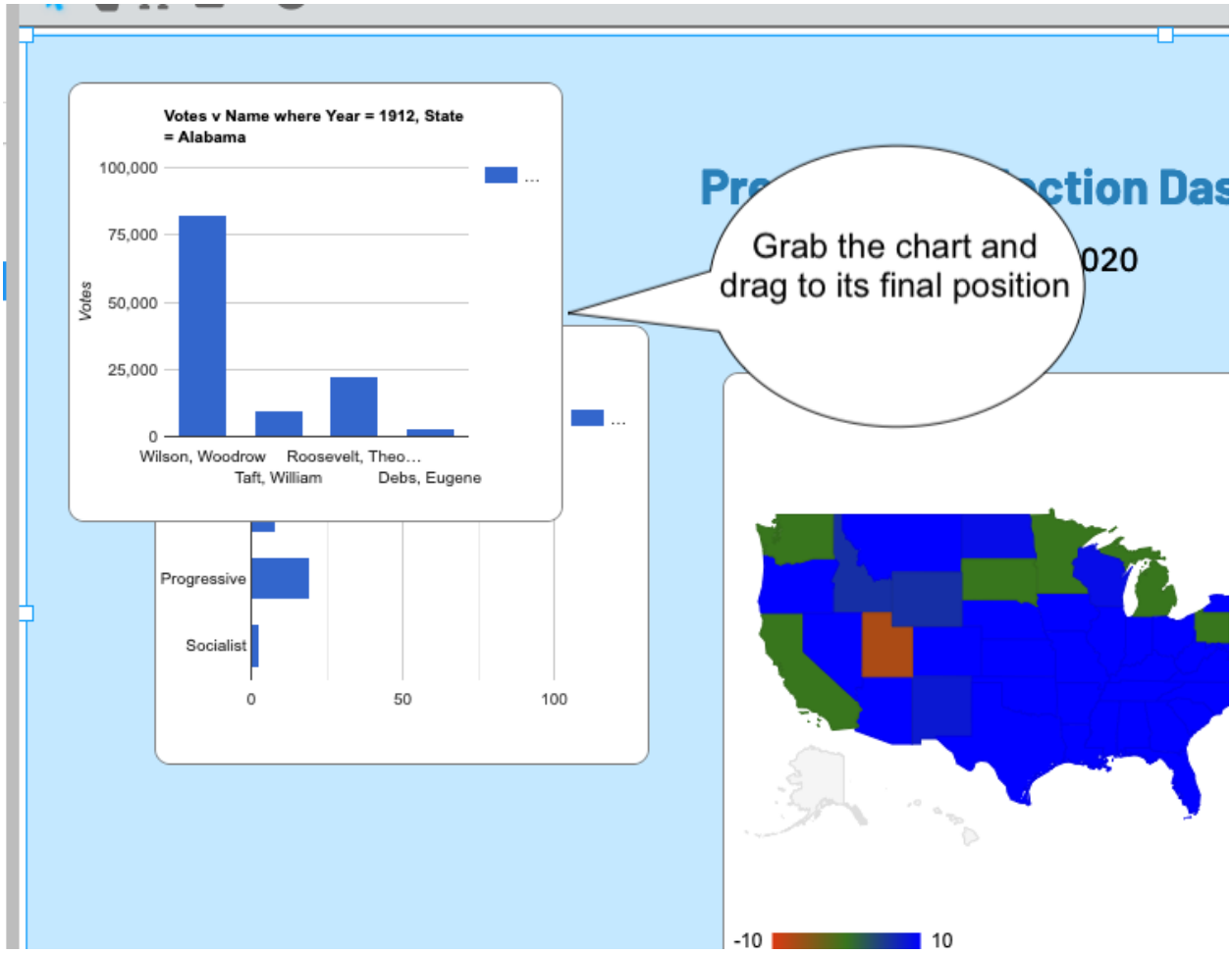
Once the Chart Editor pops up (it is the standard Google Chart Editor), choose the chart type either from the recommended charts on the start page, or click the Charts tab and then choose the chart type on the charts page.



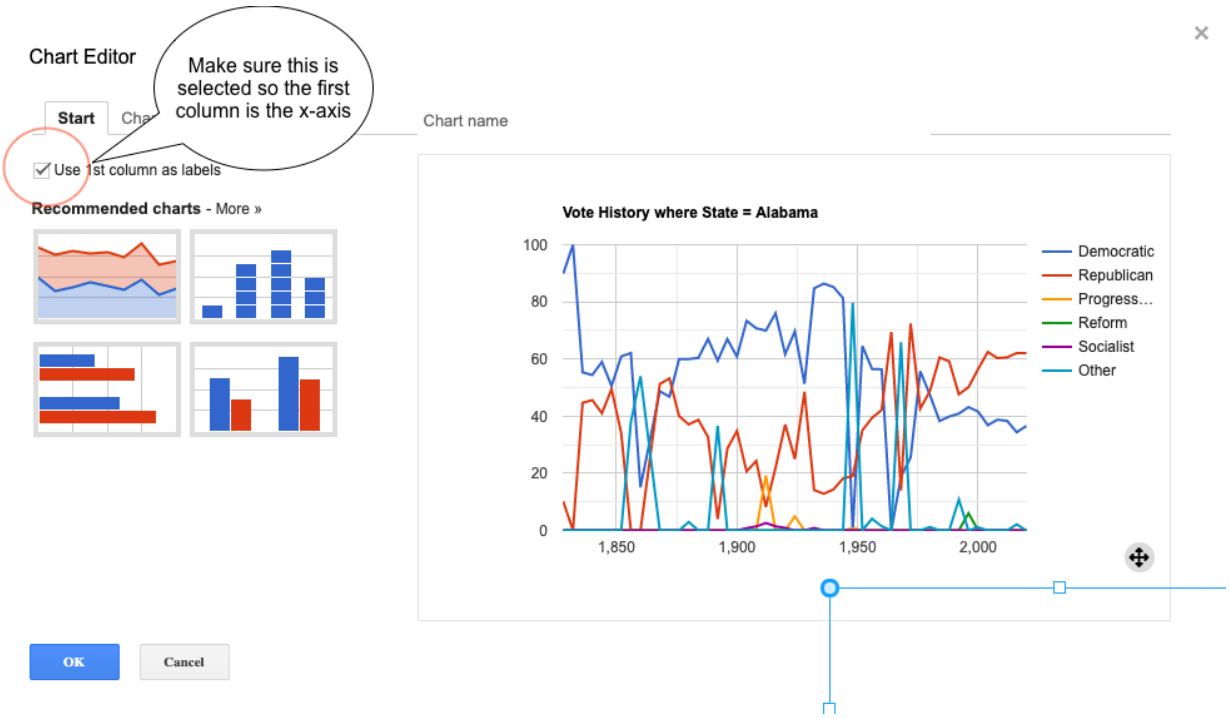
Then click customize and choose chart options. We recommend that you *not* choose a title for the chart; Galyleo automatically generates a title based on the names of the columns chosen and the values of the filters used to drive the chart.



Once you're happy with the chart, click OK



Important note. When choosing Line or Area Charts, using the first column as X-axis labels (rather than a data series) must be explicitly chosen by checking "Use 1st Column as Labels" on the Start tab in the editor.



As with tables, filters, and views, charts can be deleted using the pen icon above the chart list to switch to edit mode, then deleting charts in the same way tables, filters and charts are deleted. And, as with filters, deleting the physical chart with the Halo has the same effect as deleting charts from the chart list.

CHAPTER

THREE

THE GALYLEO PYTHON CLIENT

The Galyleo Python client is a module designed to convert Python structures into Galyleo Tables, and send them to dashboards for use with the Galyleo editor. It consists of four components:

- galyleo.galyleo_table: classes and methods to create GalyleoTables, convert Python data structures into them, and produce and read JSON versions of the tables.
- galyleo.galyleo_jupyterlab_client: classes and methods to send Galyleo Tables to Galyleo dashboards runniung under JupyterLab clients
- galyleo.galyleo_constants: Symbolic constants used by these packages and the code which uses them
- galyleo.galyleo_exceptions; Exceptions thrown by the package

3.1 Installation

The galyleo module can be installed using `pip`:

```
pip install --extra-index-url https://pypi.engagelively.com galyleo
```

When the module is more thoroughly tested, it will be put on the standard pypi servers.

3.2 License

`galyleo` is released under a standard BSD 3-Clause licence by engageLively

3.3 Galyleo Table

class galyleo.galyleo_table.**GalyleoTable**(*name: str*)

A Galyleo Dashboard Table. Used to create a Galyleo Dashboard Table from any of a number of sources, and then generate an object that is suitable for storage (as a JSON file). A GalyleoTable is very similar to a Google Visualization data table, and can be converted to a Google Visualization Data Table on either the Python or the JavaScript side. Convenience routines provided here to import data from pandas, and json format.

aggregate_by(*aggregate_column_names*, *new_column_name='count'*, *new_table_name=None*)

Create a new table by aggregating over multiple columns. The resulting table contains the aggregate column names and the new column name, and for each unique combination of values among the aggregate column names, the count of rows in this table with that unique combination of values. The new table will have name new_table_name Throws an InvalidDataException if aggregate_column_names is not a subset of the names in self.schema

Args: aggregate_column_names: names of the columns to aggregate over new_column_name: name of the column for the aggregate count. Defaults to count new_table_name: name of the new table. If omitted, defaults to None, in which case a name will be generated

Returns: A new table with name new_table_name, or a generated name if new_table_name == None

Throws: InvalidDataException if one of the column names is missing

`as_dictionary()`

Return the form of the table as a dictionary. This is a dictionary of the form: {"name": <table_name>,"table": <table_struct>} where table_struct is of the form: {"columns": [<list of schema records],"rows": [<list of rows of the table>]}

A schema record is a record of the form: {"name": < column_name>, "type": <column_type}, where type is one of the Galyleo types (GALYLEO_STRING, GALYLEO_NUMBER, GALYLEO_BOOLEAN, GALYLEO_DATE, GALYLEO_DATETIME, GALYLEO_TIME_OF_DAY). All of these are defined in galyleo_constants.

Args: None

Returns: {"name": <table_name>, "table": {"columns": <list of schema records], "rows": [<list of rows of the table>]}}

`equal(`*table*, *names_must_match=False*`)`

Test to see if this table is equal to another table, passed as an argument. Two tables are equal if their schemas are the same length and column names and types match, and if the data is the same, and in the same order. If names_must_match == True (default is False), then the names must also match

Args: table (GalyleoTable): table to be checked for equality names_must_match (bool): (default False) if True, table names must also match

Returns: True if equal, False otherwise

`filter_by_function(`*column_name*, *function*, *new_table_name*, *column_types={}*`)`

Create a new table, with name table_name, with rows such that function(row[column_name]) == True. The new table will have columns {self.columns} - {column_name}, same types, and same order Throws an InvalidDataException if: 1. new_table_name is None or not a string 2. column_name is not a name of an existing column 3. if column_types is not empty, the type of the selected column doesn't match one of the allowed types

Args: column_name: the column to filter by function: a Boolean function with a single argument of the type of columns[column_name] new_table_name: name of the new table column_types: set of the allowed column types; if empty, any type is permitted

Returns: A table with column[column_name] missing and filtered

Throws: InvalidDataException if new_table_name is empty, column_name is not a name of an existing column, or the type of column_name isn't in column_types (if column_types is non-empty)

`filter_equal(`*column_name*, *value*, *new_table_name*, *column_types*`)`

A convenience method over filter_by_function. This is identical to filter_by_function(column_name, lambda x: x == value, new_table_name, column_types)

Args: column_name: the column to filter by value: the value to march for equality new_table_name: name of the new table column_types: set of the allowed column types; if empty, any type is permitted

Returns:

A table with column[column_name] missing and filtered

Throws: InvalidDataException if new_table_name is empty, column_name is not a name of an existing column, or the type of column_name isn't in column_types (if column_types is non-empty)

filter_range(*column_name*, *range_as_tuple*, *new_table_name*, *column_types*)

A convenience method over filter_by_function. This is identical to filter_by_function(column_name, lambda x: x >= range_as_tuple[0], x <= range_as_tuple[1], new_table_name, column_types)

Args: column_name: the column to filter by range_as_tupe: the tuple representing the range new_table_name: name of the new table column_types: set of the allowed column types; if empty, any type is permitted

Returns:

A table with column[column_name] missing and filtered

Throws: InvalidDataException if new_table_name is empty, column_name is not a name of an existing column, or the type of column_name isn't in column_types (if column_types is non-empty), if len(range_as_tuple) != 2

from_json(*json_form*, *overwrite_name=True*)

Load the table from a JSON string, of the form produced by toJSON(). Note that if the overwrite_name parameter = True (the default), this will also overwrite the table name.

Throws InvalidDataException id json_form is malformed

Args: json_form: A JSON form of the Dictionary

Returns: None

Throws: InvalidDataException if json_form is malformed

load_from_dataframe(*dataframe*, *schema=None*)

Load from a Pandas Dataframe. The schema is given in the optional second parameter, as a list of records {"name": <name>, "type": <type>}, where type is a Galyleo type. (GALYLEO_STRING, GALYLEO_NUMBER, GALYLEO_BOOLEAN, GALYLEO_DATE, GALYLEO_DATETIME, GALYLEO_TIME_OF_DAY). If the second parameter is not present, the schema is derived from the name and column types of the dataframe, and each row of the dataframe becomes a row of the table.

Args:

dataframe (pandas dataframe): the pandas dataframe to load from schema (list of dictionaries): if present, the schema in list of dictionary form; each dictionary is of the form {"name": <column name>, "type": <column type>}

load_from_dictionary(*dict*)

load data from a dictionary of the form: {"columns": [<list of schema records], "rows": [<list of rows of the table>]}

A schema record is a record of the form: {"name": < column_name>, "type": <column_type}, where type is one of the Galyleo types (GALYLEO_STRING, GALYLEO_NUMBER, GALYLEO_BOOLEAN, GALYLEO_DATE, GALYLEO_DATETIME, GALYLEO_TIME_OF_DAY).

Throws InvalidDataException if the dictionary is of the wrong format or the rows don't match the columns.

Args: dict: the table as a dictionary (a value returned by as_dictionary)

Throws: InvalidDataException if dict is malformed

load_from_schema_and_data(*schema: list*, *data: list*)

Load from a pair (schema, data). Schema is a list of pairs [(<column_name>, <column_type>)] where column_type is one of the Galyleo types (GALYLEO_STRING, GALYLEO_NUMBER, GALYLEO_BOOLEAN, GALYLEO_DATE, GALYLEO_DATETIME, GALYLEO_TIME_OF_DAY). All of these are defined in galyleo_constants. data is a list of lists, where each list is a row of the table. Two conditions:

(1) Each type must be one of types listed above

(2) Each list in data must have the same length as the schema, and the type of each element must match the corresponding schema type

throws an InvalidDataException if either of these are violated

Args: schema (list of pairs, (name, type)): the schema as a list of pairs data (list of lists): the data as a list of lists

pivot_on_column(*pivot_column_name*, *value_column_name*, *new_table_name*, *pivot_column_values={}*, *other_column=False*)

The pivot_on_column method breaks out value_column into n separate columns, one for each member of pivot_column_values plus (if other_column = True), an "Other" column. This is easiest to see with an example. Consider a table with columns (Year, State, Party, Percentage). pivot_on_column('Party', {'Republican', 'Democratic'}, 'Percentage', 'pivot_table', False) would create a new table with columns Year, State, Republican, Democratic, where the values in the Republican and Democratic columns are the values in the Percentage column where the Party column value was Republican or Democratic, respectively. If Other = True, an additional column, Other, is found where the value is (generally) the sum of values where Party not equal Republican or Democratic

Args: pivot_column_name: the column holding the keys to pivot on value_column_name: the column holding the values to spread out over the pivots new_table_name: name of the new table pivot_column_values: the values to pivot on. If empty, all values used other_column: if True, aggregate other values into a column

Returns:

A table as described in the comments above

Throws: InvalidDataException if new_table_name is empty, pivot_column_name is not a name of an existing column, or value_column_name is not the name of an existing column

to_json()

Return the table as a JSON string, suitable for transmitting as a message or saving to a file. This is just a JSON form of the dictionary form of the string. (See as_dictionary)

Returns: as_dictionary() as a JSON string

3.4 JupyterLab Client

class galyleo.galyleo_jupyterlab_client.**GalyleoClient**

The Dashboard Client. This is the client which sends the tables to the dashboard and handles requests coming from the dashboard for tables.

send_data_to_dashboard(*galyleo_table*, *dashboard_name: Optional[str] = None*) → None

The routine to send a GalyleoTable to the dashboard, optionally specifying a specific dashboard to send the data to. If None is specified, sends to all the dashboards. The table must not have more than galyleo_constants.MAX_NUMBER_ROWS, nor be (in JSON form) > galyleo_constants.MAX_DATA_SIZE. If either of these conditions apply, a DataSizeExceeded exception is thrown. NOTE: this sends data to one or more open dashboard editors in JupyterLab. If there are no dashboard editors open, it will have no effect.

Args: galyleo_table: the table to send to the dashboard dashboard_name: name of the dashboard editor to send it to (if None, sent to all)

3.5 Galyleo Exceptions

Galyleo specific exceptions

exception galyleo.galyleo_exceptions.**DataSizeExceeded**
Raised when the data volume is too large on a single request. The exact limitations are specified in README.md and in galyleo_constants

exception galyleo.galyleo_exceptions.**DataSizeIsZero**
Raised when the data set is empty.

exception galyleo.galyleo_exceptions.**Error**
Base class for other exceptions.

exception galyleo.galyleo_exceptions.**InvalidDataException**
An exception thrown when a data table (list of rows) doesn't match an accoompanying schema, or a bad schema is specified, or a table row is the wrong length, or..

3.6 Galyleo Constants

Constants that are used throughout the module. These include:

1. Data types for a table (GALYLEO_STRING, GALYLEO_NUMBER, GALYLEO_BOOLEAN, GALYLEO_DATE, GALYLEO_DATETIME, GALYLEO_TIME_OF_DAY)
2. GALYLEO_TYPES: The types in a list
3. MAXIMUM_DATA_SIZE: Maximum size, in bytes, of a GalyleoTable
4. MAX_TABLE_ROWS: Maximum number of rows in a GalyleoTable

galyleo.galyleo_constants.**GALYLEO_SCHEMA_TYPES = ['string', 'number', 'boolean', 'date', 'datetime', 'timeofday']**
Maximum size of a table being sent to the dashoard. Exceeding this will throw a DataSizeExceeded exception

galyleo.galyleo_constants.**GALYLEO_TIME_OF_DAY = 'timeofday'**
Types for a chart/dashboard table schema

galyleo.galyleo_constants.**MAX_DATA_SIZE = 16777216**
Maximum number of rows in a table

CHAPTER

FOUR

THE GALYLEO INTERCHANGE FORMAT

The Galyleo Interchange Format is the wire protocol between the galyleo module and a dashboard and the disk format of a Galyleo Dashboard file. The extension of the file is .gd.json

4.1 Overall Structure

The Galyleo Interchange Format is simply the JSONified form of the data structure underlying a Galyleo Dashboard. Its overall structure is:

```
{
   "tables": <dictionary of tables>,
   "filters" <dictionary of filters>,
   "views": <dictionary of views>,
   "charts": <dictionary of charts>,
   "morphs": <list of morphs>
}
```

where each dictionary is of the form {<objectNMame>: <structure>}.

4.2 Morphic Properties

All physical objects (filters, charts, text, shapes, and images) have a field `"morphIndex"` and a substructure `"morphicProperties"`. The morphIndex gives the z-order of the morph; 0 is the frontmost morph, n the rearmost. No two object should share the same morphIndex; the morphIndex should be a total order on physical objects.

The morphicProperties structure describes its physical properties. The fields of the `"morphicProperties"` structure are:

Field	Structure	Purpose	Example	Note
fill	Color string	Color Object		
position	Coordinate	Top-left corner position	pt(128, 73)	x = 128, y = 73
extent	Coordinate	Bounding Box width/height	pt(10, 5)	width: 10, height: 5
rotation	real	In radians, clockwise	1.57	rotated 90° clockwise
border	Border object	Substructure with border properties. See below.		
opacity	real	1 = solid, 0 = transparent	0.5	semi-transparent
Clipmode	Clip enum	Handle bounding box overflow	scroll	scroll bars appear

A Color Object is a four-tuple:

Field	Type	Semantics
r	0-1 real	red intensity (0 = none, 1 = full)
g	0-1 real	green intensity (0 = none, 1 = full)
b	0-1 real	blue intensity (0 = none, 1 = full)
a	0-1 real	opactiy (0 = transparent, 1 = solid)

A Coordinate is a structure of the form 'pt(x, y)', where x and y are reals.

The Clip enum is one of visible, scroll, or auto. Visible is overflow the bounds, hidden is do not show the overflow, scroll is show scrollbars

A border structure is a dictionary with a number of fields. Each field is a structure of the form *{"top": <top>, "bottom": <bottom>, "right": <right>, "left":<left>}*, where <top>, <bottom>, <right>, and <left> specify the values for each side of the structure. For example:

`{ "width": {"top": 2, "bottom": 2, "right": 4, "left": 4}}` specifies that the width, in pixels, of the top and bottom borders are 2 and the width of the side borders are 4.

The fields are here. The structure refers to the structure of each of the top, bottom, right, and left components.

Field	Structure	Purpose	Example	Note
Width	Real	Border width in pixels	2	Must be > 0
Radius	Real	Side arc radius, as a percentage of side length	10	0 = Straight side 100 = full arc
Type	BorderType	style of the border line	dotted	border composed of dots
Color	Color object	color of the border line		

A BorderType is an enum (as a string). It is one of: none, hidden, solid, dotted, dashed, ridged, double, groove, or inset

4.3 Tables

A table is a dictionary with two entries, columns and rows. A column is a list of entries of the form `{"name": <columnName>, "type": <columnType>}` where column type must be one of 'string', 'number', 'boolean', 'date', 'datetime', 'timeofday'.

The rows field is a list of lists, where each list represents a row of the table. Each component list *must*:

- Be of the same length as the list of columns
- Be type-compliant with the list of columns. The ith entry in the list must be of the type specified in the ith entry of the column list

4.4 Filters

A Filter is a structure of the form: `{"savedFilter": <filterSpecification>, "morphIndex": <index>, "morphicProperties": <properties>}`. See the discussion above for morphicIndex and morphicProperties. Each filter specification has a field "part", which gives the URL for the lively.next component used to build the slider. The filter specification is specific to the filter type, and these are given here:

- Slider

Field	Type	Role
part	part URL	Url of the prototype for the filter
columnName	String	Name of the column this filters
minVal	Number	Minimum possible value for this filter
maxVal	Number	Maximum possible value for this filter
value	Number	Current value for this filter
increment	number	distance between consecutive values
type	enum	type of filter

For as Slider, the filter type is always "NumericSelect".

- List, Dropdown

Field	Type	Role
part	part URL	Url of the prototype for the filter
columnName	String	Name of the column this filters
choices	List	List of chocies for this filter
selection		Current selection for this filter
type	enum	type of filter

Selection is the type of the choices of the list; the filter type is always "Select".

- Range, Double Slider

Field	Type	Role
part	part URL	Url of the prototype for the filter
columnName	String	Name of the column this filters
minVal	Number	Minimum possible value for this filter
maxVal	Number	Maximum possible value for this filter
min	Number	Current minimum of the range selected
max	Number	Current maximum of the range selected
increment	number	distance between consecutive values
type	enum	type of filter

The filter type is always "Range".

4.5 Views

A View is an extremely simple structure; it has three components:

Field	Type	Role
table	string	name of the underlying table
filters	list of strings	Unordered list of the names of the filters used to find the rows
columns	list of strings	*ordered* list of the names of the columns in this view

4.6 Charts

A Chart is also a simple structure. It has four fields:

Field	Type	Role/Notes
chartType	enum	type of the chart (chosen from a supported chart library)
options	object	chart options (library specific)
viewOrTable	string	name of the View or Table that is the data source for the chart
morphIndex	number	order of the chart in the scene (0 = front)
morphicProperties	object	see above

4.7 Morphs

A morph is a simple structure. Since Morphs are not stored in dictionaries, but rather in lists, the name of the morph is in the morph structure. Every morph has four fields:

Field	Type	Role
name	string	name of the morph
type	enum morphType	morph type: list below
morphIndex	number	z-order of the morph
morphicProperties	object	morphic properties

The morph types are Rectangle, Ellipse, Image, and Text. The Image morph has one additional field:

Field	Type	Role
imageUrl	URL	URL of the image (can be a data URL)

The Text morph has one additional field:

Field	Type	Role
textProperties	object	Text-specific properties

The text properties are given here:

Field	Type	Role
fontFamily	string	Name of the font family
fontSize	number	Size of the font, in pts
fontWeight	enum fontWeight	Weight, fine to bold
fontStyle	list of styles	Weight, fine to bold
fontColor	Color Object	text color
padding	number	padding between text and bounding box
textAlign	enum alignment	text alignment
textDecoration	enum decoration	underlined or not
lineWrapping	enum wrapping	whether to wrap text
fixedHeight	boolean	if true, bounding box height independent of tex
fixedWidth	boolean	if true, bounding box width independent of text
textString	string	the text string itself

- A fontWeight is one of “Fine”, “Medium”, “Bold”, “Extra Bold”
- textAlign is one of “center”, “left”, “right”, “justified”
- fontStyle is one of “normal”, “italic”, “oblique”
- textDecoration is one of “underline” or “none”
- linewrapping is one of “by words”, “anywhere”, “only by words”, “none”

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX