# ipyflex Documentation

*Release 0.2.5*

**Trung Le**

**Oct 25, 2022**

# INSTALLATION AND USAGE

Version: 0.2.5

**ipyflex** aims to help users transform existing Jupyter widgets into an interactive dashboard with a sophisticated layout without coding.

By being a Jupyter widget itself, **ipyflex** can be easily integrated with Voila to deploy the dashboard.

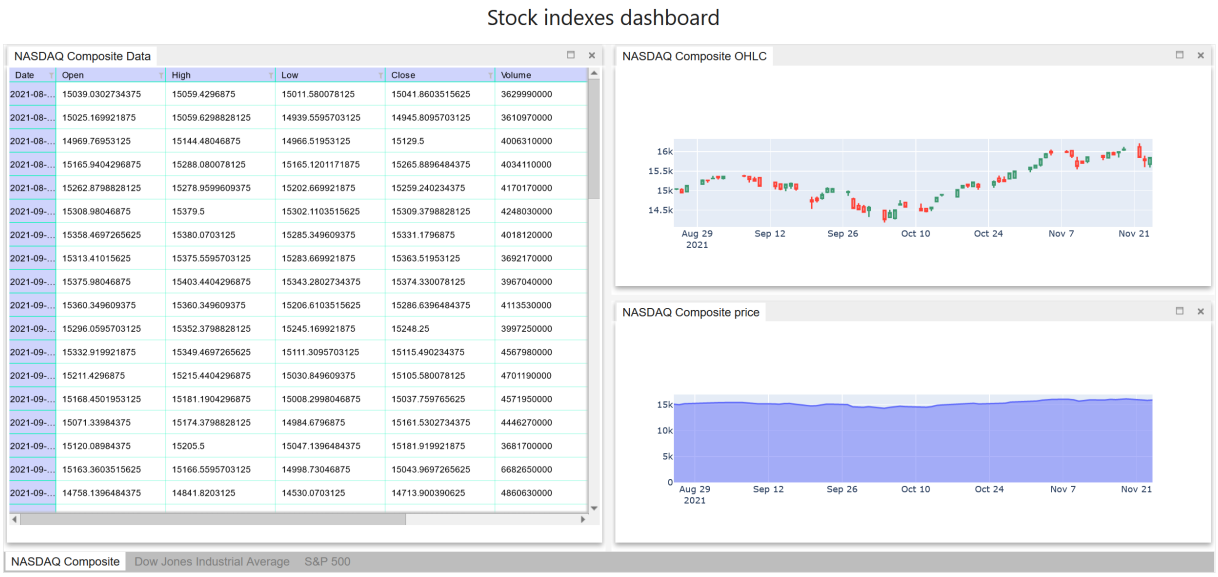



Fig. 1: A **ipyflex** dashboard deployed with *Voila*.

CHAPTER

ONE

# QUICKSTART

To get started with ipyflex, install with pip:

```
pip install ipyflex
```

or with conda:

```
conda install -c conda-forge  ipyflex
```

CHAPTER

TWO

# CONTENTS

## 2.1 Installation

The simplest way to install ipyflex is via pip:

```
pip install ipyflex
```

or via conda:

```
conda install -c conda-forge  ipyflex
```

- If you installed via pip, and notebook version < 5.3, you will also have to install / configure the front-end extension as well. If you are using classic notebook (as opposed to Jupyterlab), run:

```
jupyter nbextension install [--sys-prefix / --user / --system] --py ipyflex
jupyter nbextension enable [--sys-prefix / --user / --system] --py ipyflex
```

with the appropriate flag. If you are installing using conda, these commands should be unnecessary, but If you need to run them the commands should be the same (just make sure you choose the *–sys-prefix* flag).

- If you are using Jupyterlab <= 2, install the extension with:

```
conda install -c conda-forge yarn
jupyter labextension install @jupyter-widgets/jupyterlab-manager ipyflex
```

## 2.2 Usage

**ipyflex** is meant to be used with widgets based on ipywidgets. The entry point of **ipyflex** is the *FlexLayout* class, it allows users to dynamically customize the layout and fill their dashboard from the existing widgets.

### 2.2.1 Create a dashboard from existing widgets

The simplest way to create an **ipyflex** dashboard is to create a dictionary of existing widgets with the *keys* are the names of the widget and *values* are the instances of widgets and then use *FlexLayout* to compose the layout.

```
from ipyflex import FlexLayout
import ipywidgets as ipw
widgets = { 'Widget 1': ipw.HTML('<h1> Widget 1</h1>'),
            'Widget 2': ipw.HTML('<h1> Widget 2</h1>'),
            'Widget 3': ipw.HTML('<h1> Widget 3</h1>'),
            'Widget 4': ipw.HTML('<h1> Widget 4</h1>')
        }
dashboard = FlexLayout(widgets)
dashboard
```

#### Advanced configuration

Users can pass some configurations to the constructor of *FlexLayout* to set the template or the style of the dashboard:

```
dashboard = FlexLayout(widgets,
    template = 'saved.json',
    style = {'height': '50vh', 'borderTop': '5px'},
    header= True,
    layout_config = {'borderLeft': False, 'borderRight': False, 'enableSection': False},
    editable = False)
```

- **template**: the path to save template file, this file can be generated from the dashboard interface.
- **style**: CSS styles to be passed to the root element of the dashboard, it accepts any CSS rules but the keys need to be in *camelCase* format.
- **header**: set to *True* to activate the default header, pass a dictionary to create a configurable header.
- **layout_config**: dashboard layout configuration, users can show or hide left/right border, enable or disable the section tab.
- **editable**: flag to enable or disable the editable mode. In non-editable mode, the *Save template* button in the header is removed, tabs can not be removed, dragged, or renamed.

#### Create widgets from factory functions

In the case of using existing widgets in *FlexLayout* dashboard, users can create multiple views of a widget, so all tabs are linked. If users want to have the independent widget in each tab, *FlexLayout* allows users to define the factories to create widgets from the interface.

```
def slider_factory(label: 'Label of slider', value: 'Initial value'):
    return ipywidgets.FloatSlider(value=float(value), description=label )

factories = {"Slider factory": slider_factory}

dashboard = FlexLayout(widgets, factories=factories)
```

If the factory function needs parameters, *FlexLayout* will build an input form to get parameters from the interface. Users can define annotations to have the label of the input form.

---

**Note:** *FlexLayout* will pass all parameters as string, users need to convert the inputs to their appropriate type in the factory function.

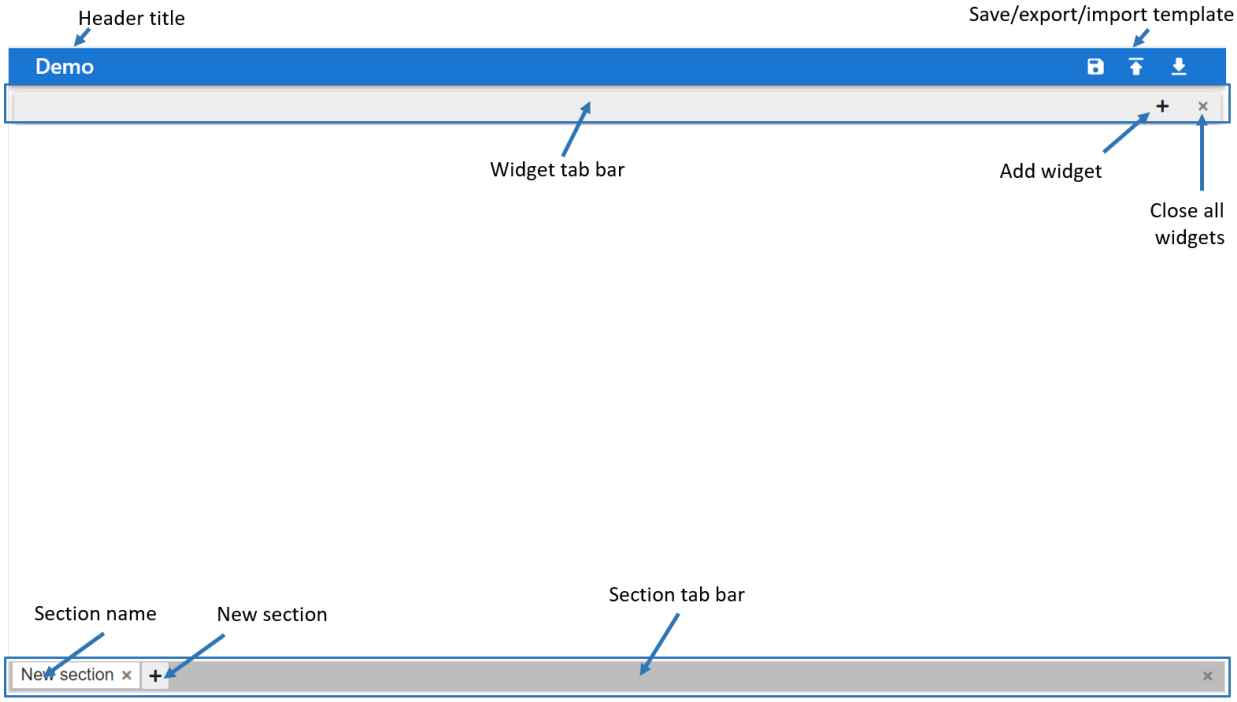
---

### FlexLayout interface

*FlexLayout* interface is composed of three components:

- Toolbar: located at bottom of the interface, it contains the button to save the current layout template to disk.
- Section tab bar: a bar to hold the section tabs, it is located on top of the toolbar. A *FlexLayout* dashboard can contain multiple sections.
- Section display window: the activated section is shown in this window. Each section is can be composed of multiple widgets.

A typical interface is displayed in the figure below:

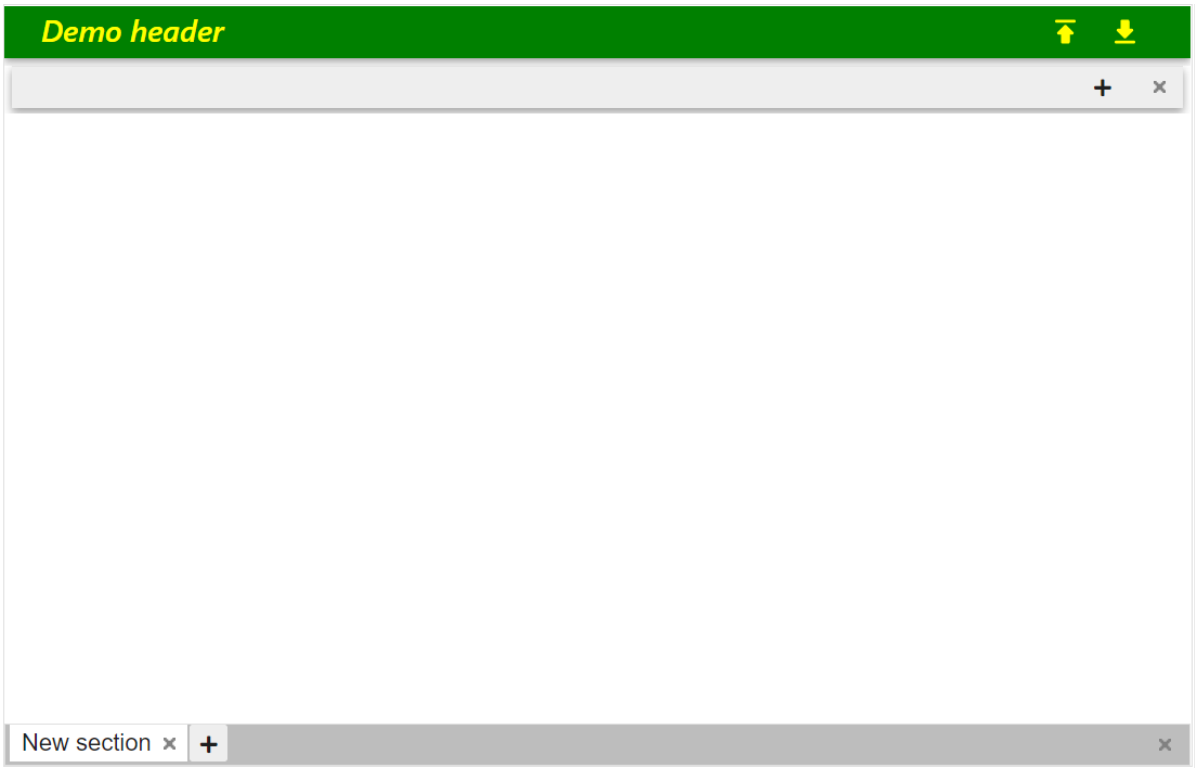


### Header

A header can be customized by setting the *header* parameter of the *FlexLayout* constructor with a dictionary of three keys.

- *title*: Title of the header
- *style*: CSS styles to be passed to the header HTML element, it accepts any CSS rules but the keys need to be in *camelCase* format.
- *buttons*: a list of buttons to be shown on the header, users can choose from *save*, *import*, *export*.
  - Save button: Save the dashboard template to the same folder of the notebook, this feature requires a kernel to handler saving function.
  - Export button: Export the dashboard template to disk, this feature does not require the kernel, so it can be used in a pure static page.
  - Import button: Load the dashboard template from a *json* file, this feature does not require the kernel, so it can be used in a pure static page.

```
header = dict(title='Demo header',
              style={'background':'green',
                     'color':'yellow',
                     'fontStyle': 'italic'},
              buttons=['import','export']
              )
FlexLayout(header=header)
```



### Toolbar

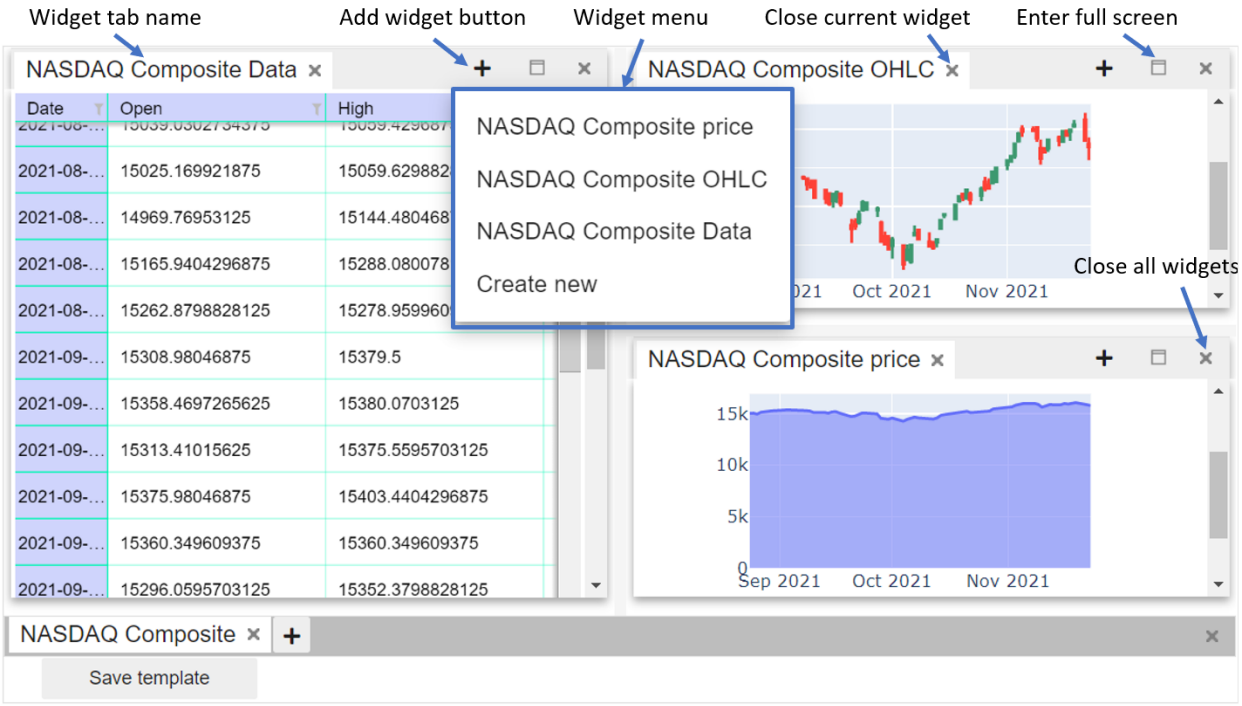
- **Save template**: save dashboard configuration into a *json* file in the current working folder. If *FlexLayout* is started with a template, the current template will be overwritten.

### Section tab bar

- Uses can use **+** button to add a new section into the dashboard, a section is displayed as a tab in the section tab bar. Each section can be dragged to modify its position, double-clicked to rename, and removed with the **x** button.

### Section display window

- A section is composed of multiple widgets, users can use the *add widget* button to add the predefined widgets into the section. The added widget will be displayed in the widget tab bar with the name taken from its key in the widget dictionary.
- A typical layout of a section with annotation for buttons is shown in the image below:




- The widget menu can be opened by the *add widget* button, it contains the keys of the widget dictionary defined in the constructor of *FlexLayout*. The *Create new* item in the widget menu is always available, it will be detailed in the next section.
- Right-click on any widget will give users options to show or hide the tab bar of this widget.
- Users can customize the layout of a section by using drag and drop on each widget. The widgets can also be resized by dragging their borders.
- Users can change the name of the widget tab by double-clicking on the tab name.

### 2.2.2 Create a dashboard layout without widgets

Even without widgets, users can still define a dashboard layout with *FlexLayout* and then fill the dashboard progressively. To do so, just use the *Create new* button in the widget menu to add widgets to the dashboard, the placeholder tabs will be created for the new widgets. Once the real widgets are ready, users can update the dashboard with `add` method:

```
dashboard = FlexLayout() # Create an empty dashboard
#Add a widget named `foo` to the dashboard by using `Create new` button
#Now add the real widget `foo_widget` to dashboard
dashboard.add('foo', foo_widget)
#The dashboard will be updated with the real widget.
```

### 2.2.3 Load and save template programmatically

The template of a *FlexLayout* dashboard can be save or load from notebook by using *save_template* and *load_template* method.

This feature is useful if you want to prepare the widgets and only create the dashboard when a user connected with some specific data about the template.

## 2.3 Examples

This section contains several examples generated from Jupyter notebooks. The widgets have been embedded into the page for demonstrative purposes.

### 2.3.1 Ipygany example

[1]:
```
# This notebook is taken from https://github.com/QuantStack/ipygany/blob/master/examples/
↪isocolor.ipynb
# with adaptation to use with ipyflex

import numpy as np
from ipywidgets import FloatSlider, FloatRangeSlider, Dropdown, Select, VBox, AppLayout,␣
↪jslink
from ipygany import Scene, IsoColor, PolyMesh, Component, ColorBar, colormaps
from ipyflex import FlexLayout

# Create triangle indices
nx = 100
ny = 100

triangle_indices = np.empty((ny - 1, nx - 1, 2, 3), dtype=int)

r = np.arange(nx * ny).reshape(ny, nx)

triangle_indices[:, :, 0, 0] = r[:-1, :-1]
triangle_indices[:, :, 1, 0] = r[:-1, 1:]
```

(continues on next page)

(continued from previous page)

```
triangle_indices[:, :, 0, 1] = r[:-1, 1:]

triangle_indices[:, :, 1, 1] = r[1:, 1:]
triangle_indices[:, :, :, 2] = r[1:, :-1, None]

triangle_indices.shape = (-1, 3)

# Create vertices
x = np.arange(-5, 5, 10/nx)
y = np.arange(-5, 5, 10/ny)

xx, yy = np.meshgrid(x, y, sparse=True)

z = np.sin(xx**2 + yy**2) / (xx**2 + yy**2)

vertices = np.empty((ny, nx, 3))
vertices[:, :, 0] = xx
vertices[:, :, 1] = yy
vertices[:, :, 2] = z
vertices = vertices.reshape(nx * ny, 3)

height_component = Component(name='value', array=z)

mesh = PolyMesh(
    vertices=vertices,
    triangle_indices=triangle_indices,
    data={'height': [height_component]}
)

height_min = np.min(z)
height_max = np.max(z)

# Colorize by height
colored_mesh = IsoColor(mesh, input='height', min=height_min, max=height_max)

# Create a slider that will dynamically change the boundaries of the colormap
colormap_slider_range = FloatRangeSlider(value=[height_min, height_max], min=height_min,␣
↪max=height_max, step=(height_max - height_min) / 100.)

jslink((colored_mesh, 'range'), (colormap_slider_range, 'value'))

# Create a colorbar widget
colorbar = ColorBar(colored_mesh)

# Colormap choice widget
colormap = Dropdown(
    options=colormaps,
    description='colormap:'
)

jslink((colored_mesh, 'colormap'), (colormap, 'index'))
scene = Scene([colored_mesh])
```

(continues on next page)

(continued from previous page)

```
color = VBox([colormap, colorbar])
```

[2]:
```
# Create the dictionary of widgets
widgets = {'Viewer': scene,'Slider range': colormap_slider_range, 'Color map': color}
```

[3]:
```
w = FlexLayout(widgets, style={'height':'620px'}, template = '3d.json', editable=False)
```

[4]:
```
w
```

[4]:
```
FlexLayout(children={'Viewer': Scene(children=[IsoColor(input='height', max=1.0, min=-0.
↪21723236496763176, par...
```

[ ]:
```
```

### 2.3.2 A widget factory example

[2]:
```
import ipywidgets
from ipyflex import FlexLayout
```

[3]:
```
slider = ipywidgets.FloatSlider(description='Linked slider')
```

[4]:
```
def slider_factory(label: 'Label of slider', value: 'Initial value'):
    return ipywidgets.FloatSlider(value=float(value), description=label )
```

[5]:
```
widgets = {'Linked slider': slider}
factories = {"Slider factory": slider_factory}
```

#### Create an empty dashboard with factory

[6]:
```
FlexLayout(widgets, factories=factories, style={'height':'300px'})
```

```
FlexLayout(children={'Linked slider': FloatSlider(value=0.0, description='Linked slider
↪')}, layout_config={'bo...
```

#### Load dashboard from template

[ ]:
```
FlexLayout(widgets, factories=factories, style={'height':'300px'}, template = 'widget_
↪factory.json')
```

[ ]:
```
```

### 2.3.3 A quick start example

[1]:
```
from ipyflex import FlexLayout
import ipywidgets as ipw
```

#### Create the widget dictionary for FlexLayout

[2]:
```
widgets = {
    'Widget 1' : ipw.HTML('<h1>Widget 1</h1>'),
    'Widget 2' : ipw.HTML('<h1>Widget 2</h1>'),
    'Widget 3' : ipw.HTML('<h1>Widget 3</h1>'),
    'Widget 4' : ipw.HTML('<h1>Widget 4</h1>')
}
```

#### Initialize an empty dashboard

[3]:
```
FlexLayout(widgets, style={'height':'400px'})
```

[3]:
```
FlexLayout(children={'Widget 1': HTML(value='<h1>Widget 1</h1>'), 'Widget 2': HTML(value=
↪'<h1>Widget 2</h1>'),...
```

#### Load saved template with a non-editable dashboard

[4]:
```
FlexLayout(widgets, style={'height':'400px'}, template='quickstart.json', editable=False,
↪ header=True)
```

[4]:
```
FlexLayout(children={'Widget 1': HTML(value='<h1>Widget 1</h1>'), 'Widget 2': HTML(value=
↪'<h1>Widget 2</h1>'),...
```

[ ]:
```

```

### 2.3.4 Simple dashboard with chart

[1]:
```
from ipyflex import FlexLayout
import ipywidgets as widgets
import plotly.graph_objects as go
import numpy
import math
```

[2]:
```
slider = widgets.FloatRangeSlider(description = 'Range')
```

[3]:
```
omega = widgets.FloatSlider(description = 'Omega',value = 1)
```

[4]:
```
fig = go.FigureWidget()
fig.add_trace(go.Scatter(x=[],y=[]))
fig.update_layout(title = 'Hello Ipyflex')
```

(continues on next page)

(continued from previous page)

```
fig.update_traces

f = lambda t: math.sin(t)
def compute(*ignore):
    min = slider.value[0]
    max =  slider.value[1]
    x = numpy.arange(min, max, (max-min)/100)
    y =  [f(omega.value* _) for _ in x]
    fig.data[0].x= x
    fig.data[0].y= y

slider.observe(compute, 'value')
omega.observe(compute, 'value')
```

[5]:
```
all_widgets =  {'A slider widget':slider, 'Output result': fig, 'Another slider': omega}
```

[6]:
```
header = dict(title='Demo header',
              style={'background':'green',
                     'color':'yellow',
                     'fontStyle': 'italic'},
               buttons=['import','export']
              )
w = FlexLayout(all_widgets, style={'height': '600px'}, template = './simple.json',␣
↪editable=True, header=header)
w
```

[6]:
```
FlexLayout(children={'A slider widget': FloatRangeSlider(value=(25.0, 75.0), description=
↪'Range'), 'Output res...
```

[ ]:
```

```

### 2.3.5 Stock indexes example

[1]:
```
import numpy as np
import pandas as pd
from ipydatagrid import DataGrid, TextRenderer, BarRenderer, Expr
import ipydatagrid
import time
import plotly.graph_objects as go
from ipyflex import FlexLayout
```

[2]:
```
def create_graph(x, y, title, **kwargs):
    layout = go.Layout(
        autosize=True,
        height=300,
    )
    fig = go.FigureWidget(data=go.Scatter(x=x,y=y,fill='tozeroy'), layout=layout)
    return fig
```

(continues on next page)

(continued from previous page)

```
def create_OHLC(data, title, **kwargs):
    layout = go.Layout(
        autosize=True,
        height=300,
    )
    fig = go.FigureWidget(data=go.Candlestick(x=data.index,
                        open=data['Open'],
                        high=data['High'],
                        low=data['Low'],
                        close=data['Close']), layout=layout)
    fig.update(layout_xaxis_rangeslider_visible=False)
    return fig

cotton_candy = {
    "header_background_color": "rgb(207, 212, 252, 1)",
    "header_grid_line_color": "rgb(0, 247, 181, 0.9)",
    "vertical_grid_line_color": "rgb(0, 247, 181, 0.3)",
    "horizontal_grid_line_color": "rgb(0, 247, 181, 0.3)",
    "selection_fill_color": "rgb(212, 245, 255, 0.3)",
    "selection_border_color": "rgb(78, 174, 212)",
    "header_selection_fill_color": "rgb(212, 255, 239, 0.3)",
    "header_selection_border_color": "rgb(252, 3, 115)",
    "cursor_fill_color": "rgb(186, 32, 186, 0.2)",
    "cursor_border_color": "rgb(191, 191, 78)",
}


def create_widget(data):
    widgets = {}
    for ticker, value in data.items():
        index = value.index
        _price = create_graph(index, value.Close,f'{ticker}',
                 labels=["Open","High", "Low", "Close"])
        _ohlc = create_OHLC(value, f'{ticker} OHLC')
        datagrid = DataGrid(value, base_row_size=32, base_column_size=150, layout={
↪"height": "630px"})
        datagrid.grid_style = cotton_candy
        widgets[f'{ticker} price'] = _price
        widgets[f'{ticker} OHLC'] = _ohlc
        widgets[f'{ticker} Data'] = datagrid
    return widgets
```

[3]:
```
import yfinance as yahooFinance
tickers  = ['^IXIC', '^DJI', '^GSPC']
tickers_name = {'^IXIC':'NASDAQ Composite', '^DJI':'Dow Jones Industrial Average', '^GSPC
↪':'S&P 500'}
data = {}
for ticker in tickers:
    info = yahooFinance.Ticker(ticker)
    name = tickers_name[ticker]
    data[name] = info.history(period="3mo")
```

[4]:
```
widgets = create_widget(data)
```

[5]:
```
a = FlexLayout(widgets, style={'height':'700px'}, template='stock.json', editable=False)
```

[6]:
```
a
```

[6]:
```
FlexLayout(children={'NASDAQ Composite price': FigureWidget({
    'data': [{'fill': 'tozeroy',
              '...
```

[ ]:
```

```

### 2.3.6 A widget factory example

[2]:
```
import ipywidgets
from ipyflex import FlexLayout
```

[3]:
```
slider = ipywidgets.FloatSlider(description='Linked slider')
```

[4]:
```
def slider_factory(label: 'Label of slider', value: 'Initial value'):
    return ipywidgets.FloatSlider(value=float(value), description=label )
```

[5]:
```
widgets = {'Linked slider': slider}
factories = {"Slider factory": slider_factory}
```

#### Create an empty dashboard with factory

[6]:
```
FlexLayout(widgets, factories=factories, style={'height':'300px'})
```

```
FlexLayout(children={'Linked slider': FloatSlider(value=0.0, description='Linked slider
↪')}, layout_config={'bo...
```

#### Load dashboard from template

[ ]:
```
FlexLayout(widgets, factories=factories, style={'height':'300px'}, template = 'widget_
↪factory.json')
```

[ ]:
```

```

## 2.4 Changelog

### 2.4.1 0.2.5

(Full Changelog)

#### Merged PRs

- Support ipywidgets 8 #48 (@trungleduc)

#### Contributors to this release

(GitHub contributors page for this release)

@github-actions | @trungleduc

### 2.4.2 0.2.4

(Full Changelog)

#### Merged PRs

- Bump to 0.2.4 #45 (@trungleduc)

#### Contributors to this release

(GitHub contributors page for this release)

@trungleduc

### 2.4.3 0.2.3

(Full Changelog)

#### Merged PRs

- Fix layout config #43 (@trungleduc)

#### Contributors to this release

(GitHub contributors page for this release)

@trungleduc

### 2.4.4 0.2.2

(Full Changelog)

#### Merged PRs

- Hotfix for `layout_config` #38 (@trungleduc)

#### Contributors to this release

(GitHub contributors page for this release)

@trungleduc

### 2.4.5 0.2.1

(Full Changelog)

#### Enhancements made

- Add `layout_config` #36 (@trungleduc)

#### Contributors to this release

(GitHub contributors page for this release)

@trungleduc

### 2.4.6 0.2.0

(Full Changelog)

#### Enhancements made

- Update non-editable mode #32 (@trungleduc)
- Show or hide tabs individually #30 (@trungleduc)
- Add api to save/load template from python side. #29 (@trungleduc)
- Add header component #21 (@trungleduc)
- Create widget from factory #18 (@trungleduc)

#### Maintenance and upkeep improvements

- Add `galata` UI tests #27 (@trungleduc)
- Add Jupyter releaser #19 (@trungleduc)

#### Documentation improvements

- Update docs #33 (@trungleduc)
- ipyvuetify examples #10 (@joseberlines)
- Add a quickstart example #7 (@trungleduc)

#### Other merged PRs

#### Contributors to this release

(GitHub contributors page for this release)

@github-actions | @joseberlines | @trungleduc

### 2.4.7 v0.1.2

Initial release.

## 2.5 Developer install

To install a developer version of ipyflex, you will first need to clone the repository:

```
git clone https://github.com/trungleduc/ipyflex
cd ipyflex
```

Create a dev environment:

```
conda create -n ipyflex-dev -c conda-forge nodejs yarn python jupyterlab
conda activate ipyflex-dev
```

Install the python. This will also build the TS package:

```
pip install -e ".[test, examples]"
```

When developing your extensions, you need to manually enable your extensions with the notebook / lab frontend. For lab, this is done by the command:

```
jupyter labextension develop --overwrite .
yarn run build
```

For classic notebook, you need to run:

```
jupyter nbextension install --sys-prefix --symlink --overwrite --py ipyflex
jupyter nbextension enable --sys-prefix --py ipyflex
```

Note that the *–symlink* flag doesn't work on Windows, so you will here have to run the *install* command every time that you rebuild your extension. For certain installations you might also need another flag instead of *–sys-prefix*, but we won't cover the meaning of those flags here.

### 2.5.1 How to see your changes

#### Typescript:

If you use JupyterLab to develop then you can watch the source directory and run JupyterLab at the same time in different terminals to watch for changes in the extension's source and automatically rebuild the widget:

```
# Watch the source directory in one terminal, automatically rebuilding when needed
yarn run watch
# Run JupyterLab in another terminal
jupyter lab
```

After a change wait for the build to finish and then refresh your browser and the changes should take effect.

#### Python:

If you make a change to the python code then you will need to restart the notebook kernel to have it take effect.