Star 23,446

Dash Python ***> Clientside Callbacks***

Plotly Studio: Transform any dataset into an interactive data application in minutes with AI. **Sign up for early access now.**

Clientside Callbacks

To get the most out of this page, make sure you've read about **Basic Callbacks** in the Dash Fundamentals.

Sometimes callbacks can incur a significant overhead, especially when they:

- receive and/or return very large quantities of data (transfer time)
- are called very often (network latency, queuing, handshake)
- are part of a callback chain that requires multiple roundtrips between the browser and Dash

When the overhead cost of a callback becomes too great and no other optimization is possible, the callback can be modified to be run directly in the browser instead of a making a request to Dash.

The syntax for the callback is almost exactly the same; you use `Input` and `Output` as you normally would when declaring a callback, but you also define a JavaScript function as the first argument to the `@callback` decorator.

For example, the following callback:

```
@callback(
    Output('out-component', 'value'),
    Input('in-component1', 'value'),
    Input('in-component2', 'value')
)
def large_params_function(largeValue1, largeValue2):
    largeValueOutput = someTransform(largeValue1, largeValue2)

    return largeValueOutput
```

Can be rewritten to use JavaScript like so:

```
from dash import clientside_callback, Input, Output

clientside_callback(
    """
    function(largeValue1, largeValue2) {
        return someTransform(largeValue1, largeValue2);
    }
    """,
    Output('out-component', 'value'),
    Input('in-component1', 'value'),
    Input('in-component2', 'value')
)
```

You also have the option of defining the function in a `.js` file in your `assets/` folder. To achieve the same result as the code above, the contents of the `.js` file would look like this:

```
window.dash_clientside = Object.assign({}, window.dash_clientside, {
    clientside: {
        large_params_function: function(largeValue1, largeValue2) {
            return someTransform(largeValue1, largeValue2);
        }
    }
});
```

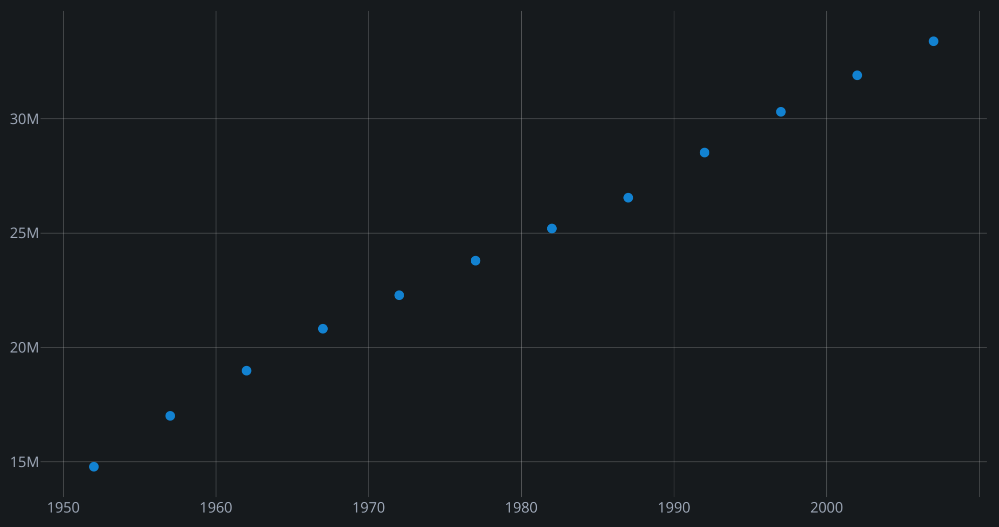
In Dash, the callback would now be written as:

```python
from dash import clientside_callback, ClientsideFunction, Input, Output

clientside_callback(
    ClientsideFunction(
        namespace='clientside',
        function_name='large_params_function'
    ),
    Output('out-component', 'value'),
    Input('in-component1', 'value'),
    Input('in-component2', 'value')
)
```

A Simple Example

Below are two examples of using clientside callbacks to update a graph in conjunction with a `dcc.Store` component. In these examples, we update a `dcc.Store` component on the backend; to create and display the graph, we have a clientside callback in the frontend that adds some extra information about the layout that we specify using the radio buttons under "Graph scale".

```python
from dash import Dash, dcc, html, Input, Output, callback, clientside_callback
import pandas as pd

import json

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = Dash(__name__, external_stylesheets=external_stylesheets)

df = pd.read_csv('https://raw.githubusercontent.com/plotly/datasets/master/gapminderDataFiveYe

available_countries = df['country'].unique()

app.layout = html.Div([
    dcc.Graph(
        id='clientside-graph'
    ),
    dcc.Store(
        id='clientside-figure-store',
        data=[{
            'x': df[df['country'] == 'Canada']['year'],
            'y': df[df['country'] == 'Canada']['pop']
        }]
    ),
    'Indicator',
    dcc.Dropdown(
        {'pop' : 'Population', 'lifeExp': 'Life Expectancy', 'gdpPercap': 'GDP per Capita'},
        'pop',
        id='clientside-graph-indicator'
    ),
    'Country',
    dcc.Dropdown(available_countries, 'Canada', id='clientside-graph-country'),
    'Graph scale',
    dcc.RadioItems(
        ['linear', 'log'],
        'linear',
        id='clientside-graph-scale'
    ),
    html.Hr(),
    html.Details([
        html.Summary('Contents of figure storage'),
        dcc.Markdown(
            id='clientside-figure-json'
        )
    ])
])


@callback(
    Output('clientside-figure-store', 'data'),
    Input('clientside-graph-indicator', 'value'),
    Input('clientside-graph-country', 'value')
```



Indicator

Population

Country

Canada

Graph scale

[✓ linear]
log

▶ Contents of figure storage

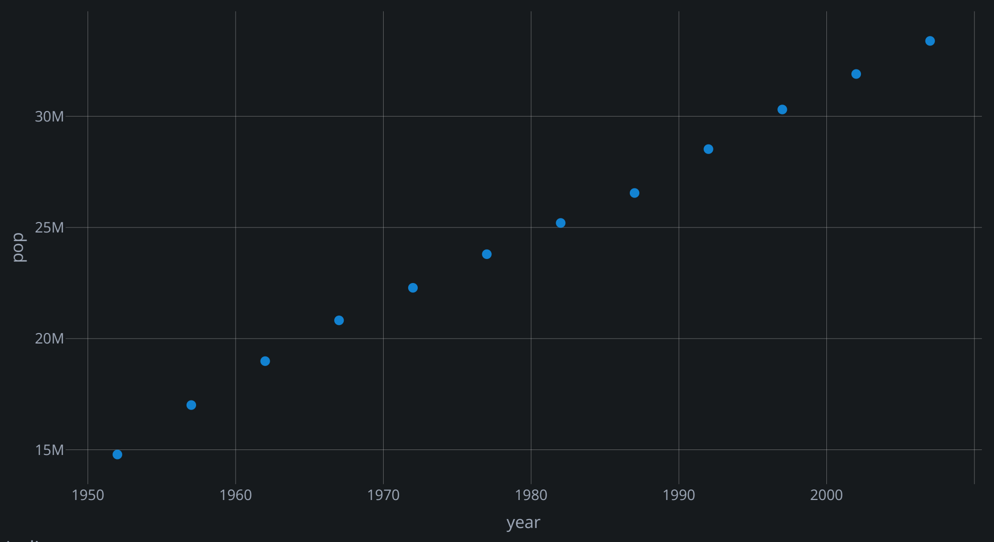
Note that, in this example, we are manually creating the `figure` dictionary by extracting the relevant data from the dataframe. This is what gets stored in our `dcc.Store` component; expand the "Contents of figure storage" above to see exactly what is used to construct the graph.

Using Plotly Express to Generate a Figure

Plotly Express enables you to create one-line declarations of figures. When you create a graph with, for example, `plotly_express.Scatter`, you get a dictionary as a return value. This dictionary is in the same shape as the `figure` argument to a `dcc.Graph` component. (See **here** for more information about the shape of `figure`s.)

We can rework the example above to use Plotly Express.

```
from dash import Dash, dcc, html, Input, Output, callback, clientside_callback
import pandas as pd
import json

import plotly.express as px

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = Dash(__name__, external_stylesheets=external_stylesheets)

df = pd.read_csv('https://raw.githubusercontent.com/plotly/datasets/master/gapminderDataFiveYe

available_countries = df['country'].unique()

app.layout = html.Div([
    dcc.Graph(
        id='clientside-graph-px'
    ),
    dcc.Store(
        id='clientside-figure-store-px'
    ),
    'Indicator',
    dcc.Dropdown(
        {'pop' : 'Population', 'lifeExp': 'Life Expectancy', 'gdpPercap': 'GDP per Capita'},
        'pop',
        id='clientside-graph-indicator-px'
    ),
    'Country',
    dcc.Dropdown(available_countries, 'Canada', id='clientside-graph-country-px'),
    'Graph scale',
    dcc.RadioItems(
        ['linear', 'log'],
        'linear',
        id='clientside-graph-scale-px'
    ),
    html.Hr(),
    html.Details([
        html.Summary('Contents of figure storage'),
        dcc.Markdown(
            id='clientside-figure-json-px'
        )
    ])
])


@callback(
    Output('clientside-figure-store-px', 'data'),
    Input('clientside-graph-indicator-px', 'value'),
    Input('clientside-graph-country-px', 'value')
)
def update_store_data(indicator, country):
```



Indicator

Population

Country

Canada

Graph scale

[✓ linear] log

▶ Contents of figure storage

Again, you can expand the "Contents of figure storage" section above to see what gets generated. You may notice that this is quite a bit more extensive than the previous example; in particular, a `layout` is already defined. So, instead of creating a `layout` as we did previously, we have to mutate the existing layout in our JavaScript code.

Clientside Callbacks with Promises

Dash 2.4 and later supports clientside callbacks that return promises.

Fetching Data Example

In this example, we fetch data (based on the value of the dropdown) using an async clientside callback function that outputs it to a `dash_table.DataTable` component.

```
from dash import Dash, dcc, html, Input, Output, dash_table, clientside_callback

app = Dash()

app.layout = html.Div(
    [
        dcc.Dropdown(
            options=[
                {
                    "label": "Car-sharing data",
                    "value": "https://raw.githubusercontent.com/plotly/datasets/master/carshare
                },
                {
                    "label": "Iris data",
                    "value": "https://raw.githubusercontent.com/plotly/datasets/master/iris_dat
                },
            ],
            value="https://raw.githubusercontent.com/plotly/datasets/master/iris_data.json",
            id="data-select",
        ),
        html.Br(),
        dash_table.DataTable(id="my-table-promises", page_size=10),
    ]
)

clientside_callback(
    """
    async function(value) {
    const response = await fetch(value);
    const data = await response.json();
    return data;
    }
    """,
    Output("my-table-promises", "data"),
    Input("data-select", "value"),
)

if __name__ == "__main__":
    app.run(debug=True)
```

Iris data ×

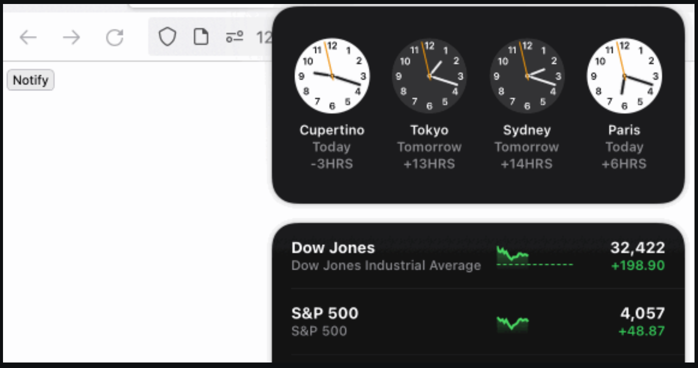
sepal length	sepal width	petal length	petal width	class
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5	3.6	1.4	0.2	Iris-setosa
5.4	3.9	1.7	0.4	Iris-setosa
4.6	3.4	1.4	0.3	Iris-setosa
5	3.4	1.5	0.2	Iris-setosa
4.4	2.9	1.4	0.2	Iris-setosa
4.9	3.1	1.5	0.1	Iris-setosa

« ‹ 1 / 15 › »

Notifications Example

This example uses promises and sends desktop notifications to the user once they grant permission and select the **Notify** button:

```
from dash import Dash, dcc, html, Input, Output, clientside_callback

app = Dash()

app.layout = html.Div(
    [
        dcc.Store(id="notification-permission"),
        html.Button("Notify", id="notify-btn"),
        html.Div(id="notification-output"),
    ]
)


clientside_callback(
    """
    function() {
        return navigator.permissions.query({name:'notifications'})
    }
    """,
    Output("notification-permission", "data"),
    Input("notify-btn", "n_clicks"),
    prevent_initial_call=True,
)

clientside_callback(
    """
    function(result) {
        if (result.state == 'granted') {
            new Notification("Dash notification", { body: "Notification already granted!"});
            return null;
        } else if (result.state == 'prompt') {
            return new Promise((resolve, reject) => {
                Notification.requestPermission().then(res => {
                    if (res == 'granted') {
                        new Notification("Dash notification", { body: "Notification granted!"
                        resolve();
                    } else {
                        reject(`Permission not granted: ${res}`)
                    }
                })
            });
        } else {
            return result.state;
        }
    }
    """,
    Output("notification-output", "children"),
    Input("notification-permission", "data"),
    prevent_initial_call=True,
)

if __name__ == "__main__":
```



Callback Context

You can use `dash_clientside.callback_context.triggered_id` within a clientside callback to access the ID of the component that triggered the callback.

In this example, we display the `triggered_id` in the app when a button is clicked.

```python
from dash import Dash, html, Input, Output

app = Dash(prevent_initial_callbacks=True)

app.layout = html.Div(
    [
        html.Button("Button 1", id="btn1"),
        html.Button("Button 2", id="btn2"),
        html.Button("Button 3", id="btn3"),
        html.Div(id="log"),
    ]
)

app.clientside_callback(
    """
    function(){
        console.log(dash_clientside.callback_context);
        const triggered_id = dash_clientside.callback_context.triggered_id;
        return "triggered id: " + triggered_id
    }
    """,
    Output("log", "children"),
    Input("btn1", "n_clicks"),
    Input("btn2", "n_clicks"),
    Input("btn3", "n_clicks"),
)

if __name__ == "__main__":
    app.run()
```

BUTTON 1 BUTTON 2 BUTTON 3

triggered id: undefined

Set Props

New in 2.16

`dash_clientside.set_props` allows you to update a Dash component property directly instead of updating it by having it as an output of a clientside callback. This can be useful if you have a non-Dash component (for example, a custom JavaScript component) that you want to update a Dash component property from, or if you want to implement custom functionality that is not available directly within Dash but that interacts with Dash.

> For an example of using `set_props` with a custom JavaScript component, go to the community-driven **Dash Example Index**.

The following example adds an event listener to the page. This event listener responds to the user pressing **Ctrl** + **R** by updating a `dcc.Store` component's `data`. Another callback has the `dcc.Store` component's `data` property as an input so runs each time it changes, outputting the updated `data` to an `html.Div` component.

```python
from dash import Dash, html, dcc, Input, Output

app = Dash()

app.layout = html.Div(
    [
        html.Span(
            [
                "Press ",
                html.Kbd("Ctrl"),
                " + ",
                html.Kbd("R"),
                " to refresh the app's data",
            ]
        ),
        dcc.Store(id="store-events", data={}),
        html.Div(id="container-events"),
    ],
    id="document",
)


app.clientside_callback(
    """
    function () {
        document.addEventListener('keydown', function(e) {

            if (e.ctrlKey && e.keyCode == 82) {
                // Simulate getting new data
                newData = JSON.stringify(new Date())

                // Update dcc.Store with ID store-events
                dash_clientside.set_props("store-events", {data: newData})

                event.preventDefault()
                event.stopPropagation()
                return dash_clientside.no_update;
            }
        });
        return dash_clientside.no_update;
    }
    """,
    Output('document', 'id'),
    Input('document', 'id'),
)


@app.callback(
    Output('container-events', 'children'),
    Input('store-events', 'data'),
    prevent_initial_call=True
)
```

Press **Ctrl** + **R** to refresh the app's data

Notes about this example

- `dash_clientside.set_props` takes two arguments. The first is the ID of the Dash component to update. The second is an object with the name of the property to update as a key, and the value as the new value to update that property to. In this example `dash_clientside.set_props("store-events", {data: newData})` updates the `data` property of the Dash component with ID `store-events`, with a new value of `newData`, which here is a variable that contains a string representation of the current date and time.
- The clientside callback returns `dash_clientside.no_update`, meaning it doesn't update any Dash component specified as an `Output`. The only update that happens to the page from the clientside callback is via `dash_clientside.set_props`.
- The `Input` for the callback that adds the event listener is the ID of the app's main container `html.Div`. We use the `id` property as this won't change after our app loads, meaning this clientside callback only runs when the app loads.
- In this example, the `Output` is the same as the `Input`, but it could be anything because we don't update the `Output`.

Limitations

There are a few limitations to keep in mind:

1. Clientside callbacks execute on the browser's main thread and will block rendering and events processing while being executed.
2. Clientside callbacks are not possible if you need to refer to global variables on the server or a DB call is required.

3. Dash versions prior to 2.4.0 do not support asynchronous clientside callbacks and will fail if a `Promise` is returned.

Dash Python ***> Clientside Callbacks***