plotly

Star 23,446

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

Background Callbacks

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

Most web servers have a 30 second timeout by default, which is an issue for callbacks that take longer to complete. While you can increase the timeout on the web server, you risk allowing long-running callbacks to use all of your app's workers, preventing other requests from going through. Background callbacks offer a scalable solution for using long-running callbacks by running them in a separate background queue. In the background queue, the callbacks are executed one-by-one in the order that they came in by dedicated queue worker(s).

You can configure a callback to run in the background by setting `background=True` on the callback. Callbacks with `background=True` use a backend configured by you to run the callback logic. There are currently two options:

- A **DiskCache** backend that runs callback logic in a separate process and stores the results to disk using the `diskcache` library. This is the easiest backend to use for local development, but is **not recommended for production**.
- A **Celery** backend that runs callback logic in a Celery worker and returns results to the Dash app through a Celery broker like **Redis**. This is recommended for production as, unlike Disk Cache, it queues the background callbacks, running them one-by-one in the order that they were received by dedicated Celery worker(s). Celery is a widely adopted, production-ready job queue library. For further information on the benefits of job queues, see the **Why Job Queues?** section below.

Dash Enterprise makes it easy to deploy Celery and Redis for using background callbacks in production. **Get Pricing** or see Dash in action at our next **demo session**.

Getting Started

The following examples use the `diskcache` manager when running locally. Install with:

```
pip install dash[diskcache]
```

When these examples are deployed to Dash Enterprise, they use `celery`.

```
pip install dash[celery]
```

Basic Steps

To use a background callback, you first need to configure a `manager` using your chosen backend. The `@dash.callback` decorator requires this manager instance. You can provide the manager instance to the `dash.Dash` app constructor as the `background_callback_manager` keyword argument, or as the `manager` argument to the `@dash.callback` decorator.

In the next five examples, we'll discuss in more detail how to implement background callbacks.

Simple Example

Here is a simple example of a background callback that updates an `html.P` element with the number of times that a button has been clicked. The callback uses `time.sleep` to simulate a long-running operation.

```
import time
import os

from dash import Dash, DiskcacheManager, CeleryManager, Input, Output, html, callback

if 'REDIS_URL' in os.environ:
    # Use Redis & Celery if REDIS_URL set as an env variable
    from celery import Celery
    celery_app = Celery(__name__, broker=os.environ['REDIS_URL'], backend=os.environ['REDIS_URL
    background_callback_manager = CeleryManager(celery_app)

else:
    # Diskcache for non-production apps when developing locally
    import diskcache
    cache = diskcache.Cache("./cache")
    background_callback_manager = DiskcacheManager(cache)

app = Dash()

app.layout = html.Div(
    [
        html.Div([html.P(id="paragraph_id", children=["Button not clicked"])]),
        html.Button(id="button_id", children="Run Job!"),
    ]
)

@callback(
    output=Output("paragraph_id", "children"),
    inputs=Input("button_id", "n_clicks"),
    background=True,
    manager=background_callback_manager,
)
def update_clicks(n_clicks):
    time.sleep(2.0)
    return [f"Clicked {n_clicks} times"]


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

Disable Button While Callback Is Running

Notice how in the previous example, there is no visual indication that the background callback is running. A user might click the "Run Job!" button multiple times before the original job can complete. You can also disable the button while the callback is running and re-enable it when the callback completes.

To do this, use the `running` argument to `@dash.callback`. This argument accepts a list of 3-element tuples. The first element of each tuple must be an `Output` dependency object referencing a property of a component in the app layout. The second element is the value that the property should be set to while the callback is running, and the third element is the value the property should be set to when the callback completes.

This example uses `running` to set the `disabled` property of the button to `True` while the callback is running, and `False` when it completes.

Note: In this example, the `background_callback_manager` is provided to the `dash.Dash` app constructor instead of the `@dash.callback` decorator.

```
import time
import os

from dash import Dash, DiskcacheManager, CeleryManager, Input, Output, html, callback

if 'REDIS_URL' in os.environ:
    # Use Redis & Celery if REDIS_URL set as an env variable
    from celery import Celery
    celery_app = Celery(__name__, broker=os.environ['REDIS_URL'], backend=os.environ['REDIS_URL
    background_callback_manager = CeleryManager(celery_app)

else:
    # Diskcache for non-production apps when developing locally
    import diskcache
    cache = diskcache.Cache("./cache")
    background_callback_manager = DiskcacheManager(cache)

app = Dash(__name__, background_callback_manager=background_callback_manager)

app.layout = html.Div(
    [
        html.Div([html.P(id="paragraph_id", children=["Button not clicked"])]),
        html.Button(id="button_id", children="Run Job!"),
    ]
)

@callback(
    output=Output("paragraph_id", "children"),
    inputs=Input("button_id", "n_clicks"),
    background=True,
    running=[
        (Output("button_id", "disabled"), True, False),
    ],
)
def update_clicks(n_clicks):
    time.sleep(2.0)
    return [f"Clicked {n_clicks} times"]


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

> There is a known issue where using `running` with a multi-pages app doesn't work as expected when a user changes page when the callback is running.

Cancelable Callback

This example builds on the previous example, adding support for canceling a long-running callback using the `cancel` argument to the `@dash.callback` decorator. We set the `cancel` argument to a list of `Input` dependency objects that reference a property of a component in the app's layout. When the value of this property changes while a callback is running, the callback is canceled. Note that the value of the property is not significant — any change in value cancels the running job (if any).

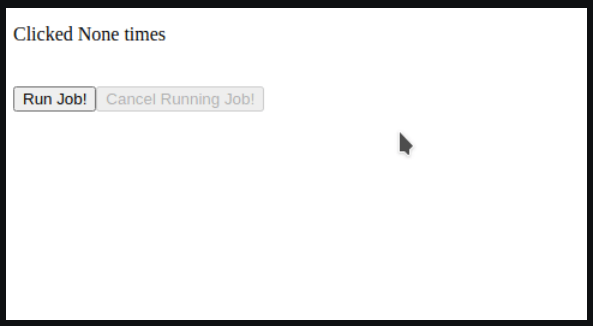
```
import time
import os

from dash import Dash, DiskcacheManager, CeleryManager, Input, Output, html, callback

if 'REDIS_URL' in os.environ:
    # Use Redis & Celery if REDIS_URL set as an env variable
    from celery import Celery
    celery_app = Celery(__name__, broker=os.environ['REDIS_URL'], backend=os.environ['REDIS_URL
    background_callback_manager = CeleryManager(celery_app)

else:
    # Diskcache for non-production apps when developing locally
    import diskcache
    cache = diskcache.Cache("./cache")
    background_callback_manager = DiskcacheManager(cache)

app = Dash(__name__, background_callback_manager=background_callback_manager)

app.layout = html.Div(
    [
        html.Div([html.P(id="paragraph_id", children=["Button not clicked"])]),
        html.Button(id="button_id", children="Run Job!"),
        html.Button(id="cancel_button_id", children="Cancel Running Job!"),
    ]
)

@callback(
    output=Output("paragraph_id", "children"),
    inputs=Input("button_id", "n_clicks"),
    background=True,
    running=[
        (Output("button_id", "disabled"), True, False),
        (Output("cancel_button_id", "disabled"), False, True),
    ],
    cancel=[Input("cancel_button_id", "n_clicks")],
)
def update_clicks(n_clicks):
    time.sleep(2.0)
    return [f"Clicked {n_clicks} times"]


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

Progress Bar

This example uses the `progress` argument to the `@dash.callback` decorator to update a progress bar while the callback is running. We set the `progress` argument to an `Output` dependency grouping that references properties of components in the app's layout.

When a dependency grouping is assigned to the `progress` argument of `@dash.callback`, the decorated function is called with a new special argument as the first argument to the function. This special argument, named `set_progress` in the example below, is a function handle that the decorated function calls in order to provide updates to the app on its current progress. The `set_progress` function accepts a single argument, which corresponds to the grouping of properties specified in the `Output` dependency grouping passed to the `progress` argument of `@dash.callback`.

```
import time
import os

from dash import Dash, DiskcacheManager, CeleryManager, Input, Output, html, callback

if 'REDIS_URL' in os.environ:
    # Use Redis & Celery if REDIS_URL set as an env variable
    from celery import Celery
    celery_app = Celery(__name__, broker=os.environ['REDIS_URL'], backend=os.environ['REDIS_UR
    background_callback_manager = CeleryManager(celery_app)

else:
    # Diskcache for non-production apps when developing locally
    import diskcache
    cache = diskcache.Cache("./cache")
    background_callback_manager = DiskcacheManager(cache)

app = Dash(__name__, background_callback_manager=background_callback_manager)

app.layout = html.Div(
    [
        html.Div(
            [
                html.P(id="paragraph_id", children=["Button not clicked"]),
                html.Progress(id="progress_bar", value="0"),
            ]
        ),
        html.Button(id="button_id", children="Run Job!"),
        html.Button(id="cancel_button_id", children="Cancel Running Job!"),
    ]
)

@callback(
    output=Output("paragraph_id", "children"),
    inputs=Input("button_id", "n_clicks"),
    background=True,
    running=[
        (Output("button_id", "disabled"), True, False),
        (Output("cancel_button_id", "disabled"), False, True),
        (
            Output("paragraph_id", "style"),
            {"visibility": "hidden"},
            {"visibility": "visible"},
        ),
        (
            Output("progress_bar", "style"),
            {"visibility": "visible"},
            {"visibility": "hidden"},
        ),
    ],
    cancel=Input("cancel_button_id", "n_clicks"),
    progress=[Output("progress_bar", "value"), Output("progress_bar", "max")],
```



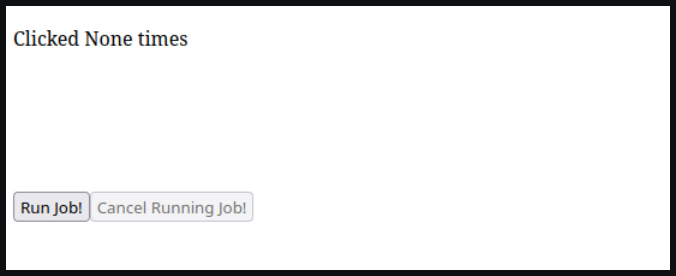
Progress Bar Chart Graph

The `progress` argument to the `@dash.callback` decorator can be used to update arbitrary component properties. This example creates and updates a Plotly bar graph to display the current calculation status.

This example also uses the `progress_default` argument to specify a grouping of values that should be assigned to the components specified by the `progress` argument when the callback is not in progress. If

`progress_default` is not provided, all the dependency properties specified in `progress` are set to `None` when the callback is not running. In this case, `progress_default` is set to a figure with a zero width bar.

```
import time
import os

from dash import Dash, DiskcacheManager, CeleryManager, Input, Output, html, dcc, callback
import plotly.graph_objects as go

if 'REDIS_URL' in os.environ:
    # Use Redis & Celery if REDIS_URL set as an env variable
    from celery import Celery
    celery_app = Celery(__name__, broker=os.environ['REDIS_URL'], backend=os.environ['REDIS_UR
    background_callback_manager = CeleryManager(celery_app)

else:
    # Diskcache for non-production apps when developing locally
    import diskcache
    cache = diskcache.Cache("./cache")
    background_callback_manager = DiskcacheManager(cache)

def make_progress_graph(progress, total):
    progress_graph = (
        go.Figure(data=[go.Bar(x=[progress])])
        .update_xaxes(range=[0, total])
        .update_yaxes(
            showticklabels=False,
        )
        .update_layout(height=100, margin=dict(t=20, b=40))
    )
    return progress_graph


app = Dash(__name__, background_callback_manager=background_callback_manager)

app.layout = html.Div(
    [
        html.Div(
            [
                html.P(id="paragraph_id", children=["Button not clicked"]),
                dcc.Graph(id="progress_bar_graph", figure=make_progress_graph(0, 10)),
            ]
        ),
        html.Button(id="button_id", children="Run Job!"),
        html.Button(id="cancel_button_id", children="Cancel Running Job!"),
    ]
)

@callback(
    output=Output("paragraph_id", "children"),
    inputs=Input("button_id", "n_clicks"),
    background=True,
    running=[
        (Output("button_id", "disabled"), True, False),
        (Output("cancel_button_id", "disabled"), False, True),
```



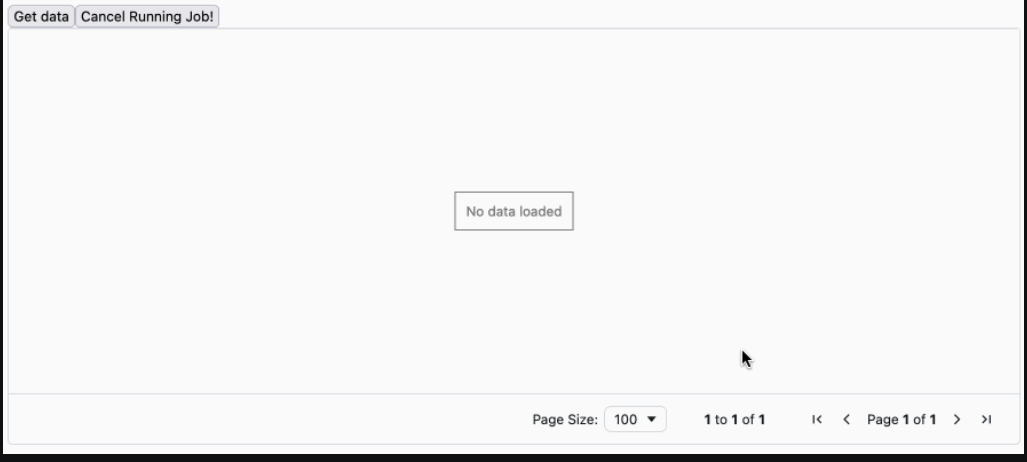
Using `set_props` Within a Callback

New in 2.17

By using `set_props` inside a callback, you can update a component property that isn't included as an output of the callback. Updates using `set_props` inside a background callback take place immediately. In the following

example, we update a Dash AG Grid's `rowData` using `set_props` every two seconds, gradually adding more data.

```
import time
import os

from dash import (
    Dash,
    DiskcacheManager,
    CeleryManager,
    Input,
    Output,
    html,
    callback,
    set_props,
)

import dash_ag_grid as dag

from plotly.express import data

if "REDIS_URL" in os.environ:
    # Use Redis & Celery if REDIS_URL set as an env variable
    from celery import Celery

    celery_app = Celery(
        __name__, broker=os.environ["REDIS_URL"], backend=os.environ["REDIS_URL"]
    )
    background_callback_manager = CeleryManager(celery_app)

else:
    # Diskcache for non-production apps when developing locally
    import diskcache

    cache = diskcache.Cache("./cache")
    background_callback_manager = DiskcacheManager(cache)

app = Dash(background_callback_manager=background_callback_manager)

app.layout = [
    html.Button(id="button_id", children="Get data"),
    html.Button(id="cancel_button_id", children="Cancel Running Job!"),
    dag.AgGrid(
        id="ag-grid-updating",
        dashGridOptions={
            "pagination": True,
        },
    ),
]


@callback(
    Input("button_id", "n_clicks"),
    background=True,
    running=[
```



In the above example, `set_props` works similarly to using `progress`, but by using `set_props`, we don't have to add the component-properties we are updating up front when defining the callback. This means, we could also

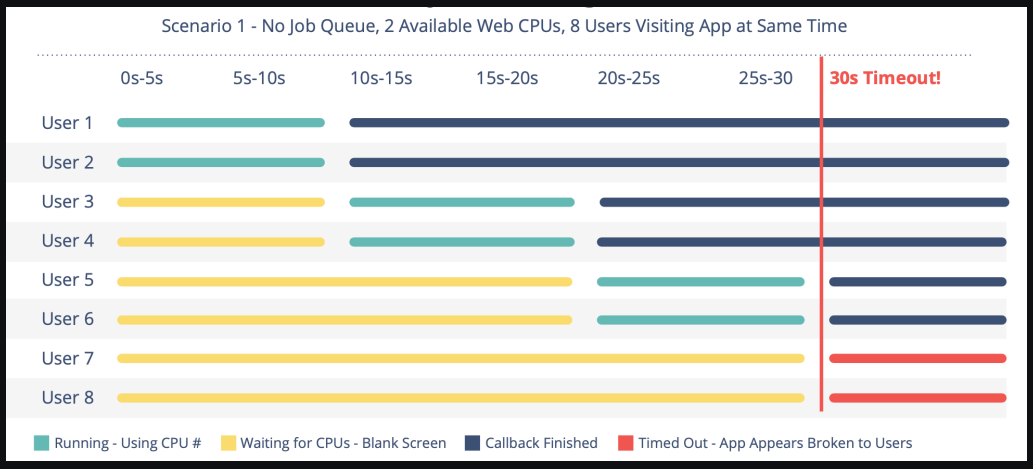
use `set_props` to update several different component-property pairs within our callback, instead of one set of component-property pairs allowed with the `progress` parameter.

Limitations

- Component properties updated using `set_props` won't appear in the **callback graph** for debugging.
- Component properties updated using `set_props` won't appear as loading when they are wrapped with a **`dcc.Loading` component**.
- `set_props` doesn't validate the `id` or `property` names provided, so no error will be displayed if they contain typos. This can make apps that use `set_props` harder to debug.
- Using `set_props` with chained callbacks may lead to unexpected results.

Why Job Queues?

When your app is deployed in production, a finite number of CPUs serve requests for that app. Callbacks that take longer than 30 seconds often experience timeouts when deployed in production. And even callbacks that take less than 30 seconds can tie up all available server resources when multiple users access your app at the same time. When all CPUs are processing callbacks, new visitors to your app see a blank screen and eventually a "Server Timed Out" message.



Job queues are a solution to these timeout issues. Like the web processes serving your Dash app, job queues run with a dedicated number of CPU workers. These workers go through the jobs one at a time and aren't subject to timeouts. While the job queue workers are processing the data, the web processes serving the Dash app and the regular callbacks display informative loading screens, progress bars, and the results of the job queues. End users never see a timeout and always see a responsive app.

Running in Dash Enterprise Workspaces

To run an app that uses background callbacks with a `celery` backend in a Dash Enterprise workspace:

1. Add a **Redis database** to your app.
2. Install Dash with `celery` in the workspace with `pip install dash[celery]`.

3. In the workspace terminal, run your app with `python app.py`
4. In a separate workspace terminal, run a `celery` worker. In the code examples above, we declared a `celery` instance with `celery_app = Celery(__name__...)` in an `app.py` file. We reference this in the command to run the worker with `app:celery_app` and also set the number of worker processes with `--concurrency`.

```
celery -A app:celery_app worker --loglevel=INFO --concurrency=2
```

> If you make changes to your app's code, you'll need to restart the `celery` worker process in the workspace terminal for those changes to apply.

Deploying to Dash Enterprise

To deploy an app that uses background callbacks with a `celery` backend to Dash Enterprise:

1. Add a **Redis database** to your app.
2. Update your app's `requirements.txt` file to include `celery` when installing Dash.

```
dash[celery]==3.1.1
```

3. Add a line to your Procfile to run a `celery` worker. In the code examples above, we declared a `celery` instance with `celery_app = Celery(__name__...)` in an `app.py` file. We reference this in the Procfile with `app:celery_app` and set the number of worker processes with `--concurrency`.

```
web: gunicorn app:server --workers 4
queue: celery -A app:celery_app worker --loglevel=INFO --concurrency=2
```

4. *If you're deploying your app to Dash Enterprise 4.x*, include a DOKKU_SCALE file with the `celery` process:

```
web=1
queue=1
```

5. **Deploy** your app.

Number of workers

In production apps, you can tune the number of workers you want to process your web requests versus process background jobs in the queue using command line flags in Gunicorn and Celery.

Here is an example of a Procfile with 4 CPUs dedicated to regular Dash callbacks and 2 CPUs dedicated to processing background callbacks in a queue.

```
web: gunicorn app:server --workers 4
queue: celery -A app:celery_app worker --loglevel=INFO --concurrency=2
```

The ratio of Gunicorn web workers to Celery queue workers will depend on your app. You'll want enough web workers that your app remains responsive to new users opening your app and enough background queue workers so tasks don't wait too long in the queue.

If your regular callbacks respond quickly (less than 500ms), consider configuring fewer web gunicorn workers.

Additional Resources

- **Background Callback Caching**
- **Introduction to Celery**
- **Redis documentation**
- **DiskCache tutorial**

Products

Dash

Consulting and Training

Pricing

Enterprise Pricing

About Us

Careers

Resources

Blog

Support

Community Support

Graphing Documentation

Join our mailing list

Sign up to stay in the loop with all things Plotly — from Dash Club to product updates, webinars, and more!

SUBSCRIBE

Copyright © 2025 Plotly. All rights reserved.

Terms of Service Privacy Policy