plotly

Star 23,447

Dash Python ***> Background Callback Caching***

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

Background Callback Caching

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

Caching Results

Background callbacks support caching callback function results, which saves and reuses the results of the function if it is called multiple times with the same arguments.

> Caching with background callbacks can help improve your app's response time, but if you want users to be able to save and access views of the app at a particular point in time, use **Dash Enterprise Snapshot Engine**. Dash Enterprise Snapshot Engine stores a full record of results, allowing you to track how the result for a specific set of parameters changes over time. For long-running callbacks where you don't need to have access to past results, use background callbacks.

How Caching Works

Imagine a dictionary is associated with each decorated callback function. Each time the decorated function is called, the input arguments to the function (and potentially other information about the environment) are hashed to generate a key.

When the callback is called, the callback checks if it has already been called with the same input arguments. It does this by checking the cache dictionary to see if there is already a value stored associated with the hashed key which represents the input arguments.

If the hashed key exists, then the function is not called and the cached result is returned. If not, the function is called and the result is stored in the dictionary using the associated key.

The built-in `functools.lru_cache` decorator uses a Python `dict` just like this.

Python's built-in LRU-cache is designed for caching data in a single Python process. When scaling Dash apps in production environments, the cache will be more effective if it:

- **Persists the cache data across app restarts & deployments**.
- **Shares the cache data across multiple processes**. For example, multiple `gunicorn` workers on a single server, or multiple servers behind a load balancer.

For these reasons, a simple Python `dict` is not a suitable storage container for caching Dash callbacks.

Instead, the Dash callback managers were designed to store data in a central place that is persistent and shared between processes.

The Celery manager stores this data in Redis, which is a shared memory database. The DiskCache manager stores the data to disk.

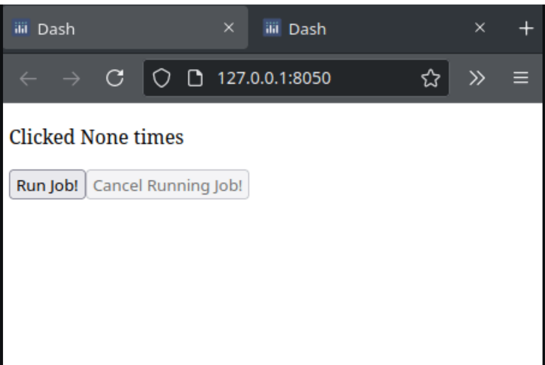
In all container-based deployment environments (including Dash Enterprise and Heroku), the filesystem is ephemeral, meaning it is only associated with the container. The cache in an ephemeral filesystem is not persisted across deploys or restarts and isn't shared between multiple replicas of the container. These are a few of the reasons why DiskCache is not suitable for production.

Enabling Caching

Caching is enabled by providing one or more zero-argument functions to the `cache_by` argument of `dash.callback`. These functions are called each time the status of a background callback function is checked, and their return values are hashed as part of the cache key.

In this example, the `cache_by` argument is set to a `lambda` function that returns a fixed UUID that is randomly generated during app initialization. The implication of this `cache_by` function is that the cache is shared across all invocations of the callback across all user sessions that are handled by a single server instance. Each time a server process is restarted, the cache is cleared and a new UUID is generated.

```
import time
import os
from uuid import uuid4

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

launch_uid = uuid4()

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, cache_by=[lambda: launch_uid], expire=60
    )

else:
    # Diskcache for non-production apps when developing locally
    import diskcache
    cache = diskcache.Cache("./cache")
    background_callback_manager = DiskcacheManager(
        cache, cache_by=[lambda: launch_uid], expire=60
    )

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"), Output("button_id", "n_clicks")),
    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"], (n_clicks or 0) % 4

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



Clicked None times

Here you can see that it takes a few seconds to run the callback function, but the cached results are used after `n_clicks` cycles back around to 0. By interacting with the app in a separate tab, you can see that the cached results are shared across user sessions.

Omitting Properties from Cache Key Calculation

The `@dash.callback` decorator has an argument `cache_args_to_ignore` that you can use to omit properties from the cache key calculation. For example, you likely won't want to include a button's `n_clicks` property in a cache key because it has a new value each time it's clicked.

If you've configured your callback with keyword arguments (`Input/State` provided in a dict), use a list of argument names as strings with `cache_args_to_ignore`.

Here we ignore the `button` argument to the callback function. This represents the first input in the dict `Input("run-button-1", "n_clicks")`

```
@callback(
    Output("result-1", "children"),
    dict(button=Input("run-button-1", "n_clicks"), value=State("input-1", "value")),
    progress=Output("status-1", "children"),
    progress_default="Finished",
    interval=500,
    cache_args_to_ignore=["button"],
    prevent_initial_call=True,
)
def update_output_1(set_progress, button, value):
    for i in range(4):
        set_progress(f"Progress {i}/4")
        time.sleep(2)
    return f"Result for '{value}'"
```

Otherwise, use a list of argument indices as integers.

Here we ignore the `number_of_clicks` argument to the callback function. This represents the first input in the list `Input("run-button-2", "n_clicks")`

```
@callback(
    Output("result-2", "children"),
    Input("run-button-2", "n_clicks"),
    State("input-2", "value"),
    background=True,
    progress=Output("status-2", "children"),
    progress_default="Finished",
    interval=500,
    cache_args_to_ignore=[0],
)
def update_output_2(set_progress, number_of_clicks, value):
    for i in range(4):
        set_progress(f"Progress {i}/4")
        time.sleep(2)
    return f"Result for '{value}'"
```

See the **Flexible Callback Signatures chapter** for more information on keyword arguments.

Cache_by Function Workflows

You can use `cache_by` functions to implement a variety of caching policies. Here are a few examples:

- **File modification cache expiry** – the `cache_by` function could return the file modification time of a dataset to automatically invalidate the cache when an input dataset changes.
- **Deployment cache expiry** – in a Heroku or **Dash Enterprise** deployment setting, the `cache_by` function could return the Git hash of the app, making it possible to persist the cache across redeploys, but invalidate it when the app's source changes.
- **User-based caching** – in a **Dash Enterprise** setting, the `cache_by` function could return user meta-data to prevent cached values from being shared across authenticated users.
- **Time-based cache expiry** – with both `CeleryManager` and `DiskcacheManager`, you can use the `expire` argument to limit how long a cache entry is retained for in the database. This is the number of seconds to keep a cache entry for after its last use. When an entry is accessed, the timer restarts.

```
celery_app = Celery(
    __name__, broker="redis://localhost:6379/0", backend="redis://localhost:6379/1"
)
background_callback_manager = CeleryManager(celery_app, expire=100)
```

Dash Python ***> Background Callback Caching***

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

Terms of Service Privacy Policy