plotly

Star 23,447

Dash Python > ***Multi-Page Apps and URL Support***

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

Multi-Page Apps and URL Support

Dash renders web applications as a "single-page app". When using `dcc.Link`, the application does not completely reload when navigating, making browsing very fast. Using Dash you can build multi-page apps using `dcc.Location` and `dcc.Link` components and callbacks.

Dash Pages uses these components and abstracts away the callback logic required for URL routing, making it easy to get up and running with a multi-page app. If you want to build a multi-page app without Pages, see the **Multi-Page Apps without Pages** section below.

> Dash Pages is not currently compatible with Dash Snapshot Engine. For details on how to make a multi-page app that is compatible with Dash Snapshot Engine, see the Multi-Page Apps without Pages section below.

Dash Pages

> Dash Pages is new in Dash 2.5. Check your version with: `print(dash.__version__)`
>
> This feature was developed in the open with collaboration from the Dash Community. Many thanks to everyone! **View the original discussion & announcement**.

Dash Pages is available from Dash version 2.5.0. It implements features to simplify creating a multi-page app, handling URL routing and offering an easy way to structure and define the pages in your app.

There are three basic steps for creating a multi-page app with Dash Pages:

1. Create individual `.py` files for each page in your app, and put them in a `/pages` directory.
2. In each of these page files:
    - Add a `dash.register_page(__name__)`, which tells Dash that this is a page in your app.
    - Define the page's content within a variable called `layout` or a function called `layout` that returns the content.
3. In your main app file, `app.py`:
    - When declaring your app, set `use_pages` to `True`: `app = Dash(__name__, use_pages=True)`
    - Add `dash.page_container` in your app layout where you want the page content to be displayed when a user visits one of the app's page paths.

Example: Simple Multi-Page App with Pages

Here is what a three-page app structure looks like with Dash Pages:

```
- app.py
- pages
   |-- analytics.py
```

```
    |-- home.py
    |-- archive.py
```

It has the main `app.py` file which is the entry point to our multi-page app (and in which we include `dash.page_container`) and three pages in our `pages` directory.

`pages/analytics.py`

```
import dash
from dash import html, dcc, callback, Input, Output

dash.register_page(__name__)

layout = html.Div([
    html.H1('This is our Analytics page'),
    html.Div([
        "Select a city: ",
        dcc.RadioItems(
            options=['New York City', 'Montreal', 'San Francisco'],
            value='Montreal',
            id='analytics-input'
        )
    ]),
    html.Br(),
    html.Div(id='analytics-output'),
])


@callback(
    Output('analytics-output', 'children'),
    Input('analytics-input', 'value')
)
def update_city_selected(input_value):
    return f'You selected: {input_value}'
```

`pages/home.py`

```
import dash
from dash import html

dash.register_page(__name__, path='/')

layout = html.Div([
    html.H1('This is our Home page'),
    html.Div('This is our Home page content.'),
])
```

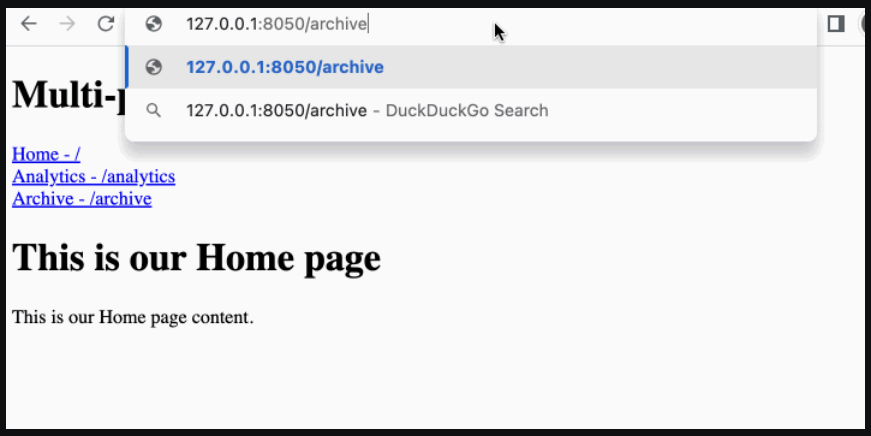
`pages/archive.py`

```
import dash
from dash import html

dash.register_page(__name__)

layout = html.Div([
    html.H1('This is our Archive page'),
    html.Div('This is our Archive page content.'),
])
```

`app.py`

```
import dash
from dash import Dash, html, dcc

app = Dash(__name__, use_pages=True)

app.layout = html.Div([
    html.H1('Multi-page app with Dash Pages'),
    html.Div([
```

```
        html.Div(
            dcc.Link(f"{page['name']} - {page['path']}", href=page["relative_path"])
        ) for page in dash.page_registry.values()
    ]),
    dash.page_container
])

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



Notes:

- **path** — We call `dash.register_page` on each of the three pages in our app. For two of the pages, we don't set a `path` property. If you don't set the `path` property, it is autogenerated based on the module name. So `archives.py`'s layout is served when a user goes to `/archives`. Similarly, the layout for `analytics.py` is served when the user goes to `/analytics`. When we call `dash.register_page` for `home.py`, we do set the `path` property. For `home.py` we set the path property because we don't want the content to be displayed when the user goes to `/home`, but when the user goes to the homepage: `/`
- **page_registry** — Pages that include a call to `dash.register_page` are added to the page registry for our app. This is an `OrderedDict` that we can extract information from about our app's pages. In our `app.py` we loop through all of our app's pages (in `dash.page_registry.values()`) and add links for each one. We can also select these links individually from the `dash.page_registry`. The page with `/` as the `path` is always in index `0` in the dict. Other pages are in alphabetical order.
- **page_container** — `app.py` has a `dash.page_container`. This is where page content is displayed when a user navigates to that page's path.

Layout

In the above example, we've defined the layout in each page using a variable called `layout`. For example, in `home.py` above:

```
layout = html.Div([
    html.H1('This is our Home page'),
    html.Div('This is our Home page content.'),
])
```

You can also use a function called `layout` that returns your page content:

```
def layout(**kwargs):
    return html.Div([
        html.H1('This is our Home page'),
        html.Div('This is our Home page content.'),
    ])
```

> Page layouts must be defined with a variable or function called `layout`. When creating an app with Pages, only use `app.layout` in your main `app.py` file.
>
> Pages captures **query strings** and **path variables** from the URL and passes them to the layout function as keyword arguments. It's recommended to include `**kwargs` in case the layout receives unexpected query strings.

dash.register_page

Calling `dash.register_page` within a file is how Dash knows to include the file as a page in your multi-page app.

As we've seen, it can be called with just the module name:

```
dash.register_page(__name__)
```

In this case, Dash generates the `path` the page is for, its `title`, and the link `name` based on the module name.

> The `title` is the **HTML `<title>`**. The `name` is the key for this page in the Dash Registry and can be used when creating links for pages. The `path` is the URL pathname of the page.

So, if we have a file called `analytics.py`, the page's path is `/analytics`, the title is `Analytics`, and the link name is `Analytics`.

We can also specify these if we don't want them to be autogenerated based on the module name, as we did in the example above with our home page.

Setting a path, title, and link name:

`pages/analytics.py`

```
dash.register_page(
    __name__,
    path='/analytics-dashboard',
    title='Our Analytics Dashboard',
    name='Our Analytics Dashboard'
)
```

See the **Reference for dash.register_page** section below for a detailed list of properties.

Dash Page Registry

Any pages that call `dash.register_page` are added to a page registry for your app.

The page registry is an `OrderedDict` called `dash.page_registry`. Each registry entry has information for a page, including property values set when `dash.register_page` was called, and values inferred by Dash. As with any dict, you can access and use its data in your code.

Here we access the `path` of our `analytics` and use it in a `dcc.Link` in `app.py`:

```
html.Div(dcc.Link('Dashboard', href=dash.page_registry['pages.analytics']['path']))
```

▶ What the Dash Page Registry looks like for our initial example, **Simple Multi-page App with Pages**

To access `dash.page_registry` from within a file in the `pages` directory, you'll need to use it within a function.

Here, we have two files within the `pages` directory: `side_bar.py` and `topic_1.py`. The `topic_1` page imports a sidebar from `side_bar.py`. Note how the function within `side_bar.py` accesses `dash.page_registry`. If this wasn't within a function, the app wouldn't work because the `dash.page_registry` wouldn't be ready when the page loads.

`side_bar.py`

```
import dash
from dash import html
import dash_bootstrap_components as dbc
```

```
def sidebar():
    return html.Div(
        dbc.Nav(
            [
                dbc.NavLink(
                    html.Div(page["name"], className="ms-2"),
                    href=page["path"],
                    active="exact",
                )
                for page in dash.page_registry.values()
                if page["path"].startswith("/topic")
            ],
            vertical=True,
            pills=True,
            className="bg-light",
        )
    )
```

`topic_1.py`

```
import dash
from dash import html

import dash_bootstrap_components as dbc

from .side_bar import sidebar

dash.register_page(__name__, name="Topics")

def layout(**kwargs):
    return dbc.Row(
        [dbc.Col(sidebar(), width=2), dbc.Col(html.Div("Topics Home Page"), width=10)]
    )
```

Dash Page Registry Order

By default, a page with a path defined as `/` is added to the registry at index `0`. Other pages are then added in alphabetical order based on file name.

You can also specify the order of pages in `dash.page_registry` by setting the `order` property on each page:

`pages/analytics.py`

```
dash.register_page(__name__, order=3)
```

If you set the `order` property on one or more pages, pages are added to the registry:

- In the order they are specified with the `order` property.
- In alphabetical order after that (for pages without the `order` property set.

Setting the order can be useful when you want to be able to loop through the links when creating a sidebar or header dynamically.

Default and Custom 404

If a user goes to a path that hasn't been declared in one of your app's pages, Pages shows a default **'404 - Page not found message'** to the user.

This page can be customized. Place a file called `not_found_404.py` in your app's `pages` directory, add `dash.register_page(__name__)` to the file, and define the content for the custom 404 within a `layout` variable or function:

```
import dash
from dash import html
```

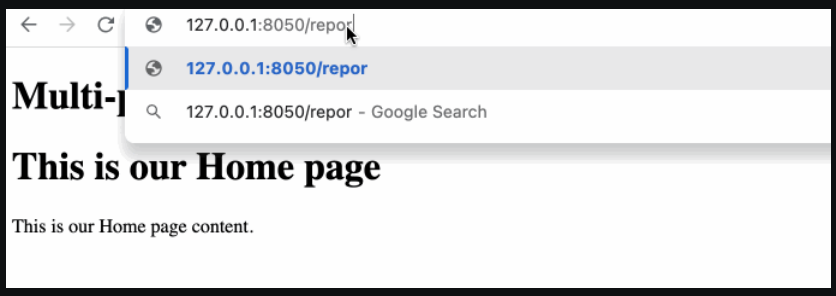
```
dash.register_page(__name__)

layout = html.H1("This is our custom 404 content")
```

Variable Paths

You can capture dynamic variables in the path by using the `path_template` parameter. Specify dynamic parts of your URL by placing it within `<variable_name>`. `variable_name` will be the named keyword argument passed into your layout function. Values that the layout function receives from the URL are always of type `str`.

Example - Single Variable Path

```
import dash
from dash import html

dash.register_page(__name__, path_template="/report/<report_id>")


def layout(report_id=None, **kwargs):
    return html.Div(
        f"The user requested report ID: {report_id}."
    )
```



Example - Two Path Variables and Update Title & Description

The path variables can also be used to update the page's `title` (what you see in the browser tab) and the page's meta `description` (information used by search engines when indexing and displaying search results and also displayed in social media when sharing links; otherwise not visible).

More information on these parameters can be found in the **Reference for dash.register_page** section below.

```
import dash
from dash import html


def title(asset_id=None, dept_id=None):
    return f"Asset Analysis: {asset_id} {dept_id}"


def description(asset_id=None, dept_id=None):
    return f"This is the AVN Industries Asset Analysis: {asset_id} in {dept_id}"


dash.register_page(
    __name__,
    path_template="/asset/<asset_id>/department/hello-<dept_id>",
    title=title,
    description=description,
)


def layout(asset_id=None, dept_id=None, **other_unknown_query_strings):
    return html.Div(
        f"variables from pathname:  asset_id: {asset_id} dept_id: {dept_id}"
    )
```

Query Strings

Query string parameters in a URL are captured by Pages and passed to the layout function as keyword arguments.

Example - Single Query String Parameter

In this example, when the user goes to `/archive?report_id=9`, the value `9` is captured by the layout function and displayed on the page. Values that the layout function receives from the URL are always of type `str`.

`pages/archive.py`

```
import dash
from dash import html

dash.register_page(__name__)


def layout(report_id=None, **other_unknown_query_strings):
    return html.Div([
        html.H1('This is our Archive page'),
        html.Div(f'This is report: {report_id}.'),
    ])
```

Example - Two Query String Parameters

In this example, when the user goes to `/archive?report_id=9&department_id=55`, the values `9` and `55` are captured by the layout function and displayed on the page.

`pages/archive.py`

```
import dash
from dash import html

dash.register_page(__name__)


def layout(report_id=None, department_id=None, **other_unknown_query_strings):
    return html.Div([
        html.H1('This is our Archive page'),
        html.Div(f'''
            This is report: {report_id}.\n
            This is department: {department_id}.
            '''),
    ])
```

Redirects

If you change a page's path, it's best practice to define a redirect so users that go to old links don't get a '404 – Page not found'. You can set additional paths to direct to a page using the `redirects` parameter. This takes a `list` of all paths that redirect to this page.

Here we have a page called `archive`. It is displayed when a user goes to `/archive`, `/archive-2021`, or `/archive-2020`

```
import dash
from dash import html

dash.register_page(
    __name__,
    path="/archive",
    redirect_from=["/archive-2021", "/archive-2020"]
)

layout = html.Div([
    html.H1('This is our Archive page'),
    html.Div('This is our Archive page content.'),
])
```

Meta Tags

> Not sure what meta tags are? **Check out this tutorial on meta tags and why you might want to use them.**

Each page you add to your app has page meta tags stored for it: a `title`, `image`, and `description`.

The `title` is used as the page title in the browser, but together with the `image` and `description`, it is also often used by social media sites and chat clients to create a card to display when someone shares a link to a page.

You can set the values for these properties with `title=`, `description=`, `image=`:

```
dash.register_page(
    __name__,
    title='Custom Page Title',
    description='Custom Page Description',
    image='logo.png'
)
```

Image types of `apng`, `avif`, `gif`, `jpeg`, `jpg`, `png`, `svg`, and `webp` are supported.

The `image` value must be the name of a file inside the `assets` folder. To set the `image` to a file that is not in the `assets` folder, such as an image hosted externally on a CDN, change `image=` to `image_url=` and provide the URL.

If you don't specify `title`, it is derived from the module name. If you don't specify a `description`, it defaults to `None`. Lastly, if you don't specify `image`, Pages checks for an image that meets one of these criteria (in order) and uses the first one it finds:

- A page-specific image at `assets/<module>.<extension>`
- A generic app image at `assets/app.<extension>`
- A logo at `assets/logo.<extension>`

For example, placing a file `analytics.png` in the `assets` folder sets this file as the `image` for `pages/analytics.py`, because the first criterion is met.

A more complete example for setting meta tags with Pages might look like:

```
import dash
from dash import html

dash.register_page(
    __name__,
    title='Explore the archive',
    image='archive_image_2022.png',
    description='The archive page shows previously generated reports.'
)

layout = html.Div([
    html.H1('This is our Archive page'),
    html.Div('This is our Archive page content.'),
])
```

The `title` and `description` properties can also be set as functions. If provided as functions, Pages calls these functions on page load and uses the values that they return. The second example in **Variable Paths** illustrates this.

Additional Keywords with Dash Page Registry

You can use additional custom key-value pairs when calling `dash.register_page` to add those to the Dash Page Registry.

For example, if you want to add information to a page about where its links appear on the home page, you could add a " location" keyword with a value of "sidebar".

```
dash.register_page(__name__, location="sidebar")
```

In your `app.py` page, in the sidebar, you can then loop through the pages that have that location set:

```
html.Div([
    html.Div(
        dcc.Link(f"{page['name']}", href=page["path"])
    ) for page in dash.page_registry.values() if page["location"] == "sidebar"
])
```

When you add new pages with `dash.register_page(__name__, location = "sidebar")`, they'll automatically be included in the sidebar.

Nested Pages

Dash Pages also recursively searches directories in the `pages` directory for additional app pages. For example, if we add a `reports` (this name is arbitrary!) directory within pages, put two pages, `summary_2020.py` and `summary_2021.py`, in that directory, and call `dash.register_page(__name__)`, they will be included in our app.

```
- app.py
- pages
    - reports
        |-- summary_2020.py
        |-- summary_2021.py
   |-- analytics.py
   |-- home.py
   |-- archive.py
```

`pages/reports/summary_2020.py`

```
import dash
from dash import html

dash.register_page(__name__)

layout = html.Div([
    html.H1('2020 Summary'),
    html.Div("This is our page's content."),
])
```

As we haven't set the `path` property, Pages will display this page when the user visits the app at the URL path `/reports/summary-2020`.

Changing the Default Pages Directory

By default, Pages checks for a directory called `pages` for your app files. You can change this when declaring your Dash app:

```
app = Dash(__name__, use_pages=True, pages_folder="my_apps")
```

Multiple Pages in One File

So far, we've built a multi-page app where we've declared each page in a separate `.py` file in our `pages` directory. It's also possible to declare multiple pages within `app.py`.

To do this, we register the page within `app.py` and pass the layout directly to `dash.register_page`. In this example, we define two pages within our app.py file: a `home` page, and an `analytics` page. For `module`, the first argument, we give each of our pages a unique name (as these names are used as keys in the Dash Page Registry). We also add `pages_folder=""` when creating our Dash app instance to specify that we are not using a folder for our app's pages.

```
from dash import Dash, html, dcc
import dash
```

```
app = Dash(__name__, use_pages=True, pages_folder="")

dash.register_page("home", path='/', layout=html.Div('Home Page'))
dash.register_page("analytics", layout=html.Div('Analytics'))

app.layout = html.Div([
    html.Div([
        html.Div(
            dcc.Link(f"{page['name']} - {page['path']}", href=page["relative_path"])
        ) for page in dash.page_registry.values()
    ]),
    dash.page_container,
])

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

Circular Imports

When using Pages, the file that declares `Dash(__name__, use_pages=True)` recursively imports all files within the `pages` folder. If any of those pages import a function from the Dash file (usually `app.py`), then you will get a circular import error.

For example, this will cause a circular import error:

`app.py`

```
import dash
from dash import Dash, html

app = Dash(__name__, use_pages=True)

app.layout = html.Div(
    dash.page_container
)

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

`analytics.py`

```
import dash
from dash import Input, Output, html, dcc
from app import app

dash.register_page(__name__)

layout = html.Div([dcc.Input(id='input'), html.Div(id='output')])


@app.callback(Output('output', 'children'), Input('input', 'value'))
def update(value):
    return value
```

Running `python app.py` displays the error `KeyError: 'pages.analytics'`

- If you are trying to use a callback within one of your files in the `pages` directory, you can instead use `callback` imported from `dash`:

```
import dash
from dash import Input, Output, html, dcc, callback

dash.register_page(__name__)

layout = html.Div([dcc.Input(id='input'), html.Div(id='output')])

@callback(Output('output', 'children'), Input('input', 'value'))
```

```
    def update(value):
        return value
```

- In cases where you need access to the `app` in one of your page layouts, you can access it with `dash.get_app`.
- If you want to use `app.get_asset_url` in one of your pages, use `dash.get_asset_url`.

App Validation

By default, Pages validates the layout as described in the **Dynamically Create a Layout for Multi-Page App Validation** section. For faster loading on large apps, you can prevent the validation by setting `suppress_callback_exceptions=True` when declaring your Dash app:

```
app = Dash(__name__, use_pages=True, suppress_callback_exceptions=True)
```

Additional Inputs to the Pages Routing

> This feature is new in Dash 2.14

By default, the routing to different pages is triggered by 2 Inputs: 1. The URL pathname 2. The URL search params

In some cases you may want to pass more information to the page layout functions, or trigger the layout rendering from other inputs. For instance you may want to:

- Re-render the whole page content when a user changes the language dropdown
- Access additional information when the page renders, without re-rendering the page everytime the value is updated (for example, serializing the app state in the URL hash)

You can pass whatever Inputs/States you want through this mechanism, but here are a few things to keep in mind:

- This will be used in a server-side callback, so passing very large amounts of data may increase the page load time
- The new Inputs/States passed to your pages will be passed to every page. Make sure that the input components are available on every page or use an `ALL` id pattern

Example 1: Updating the Language Contents of the Page

`app.py`

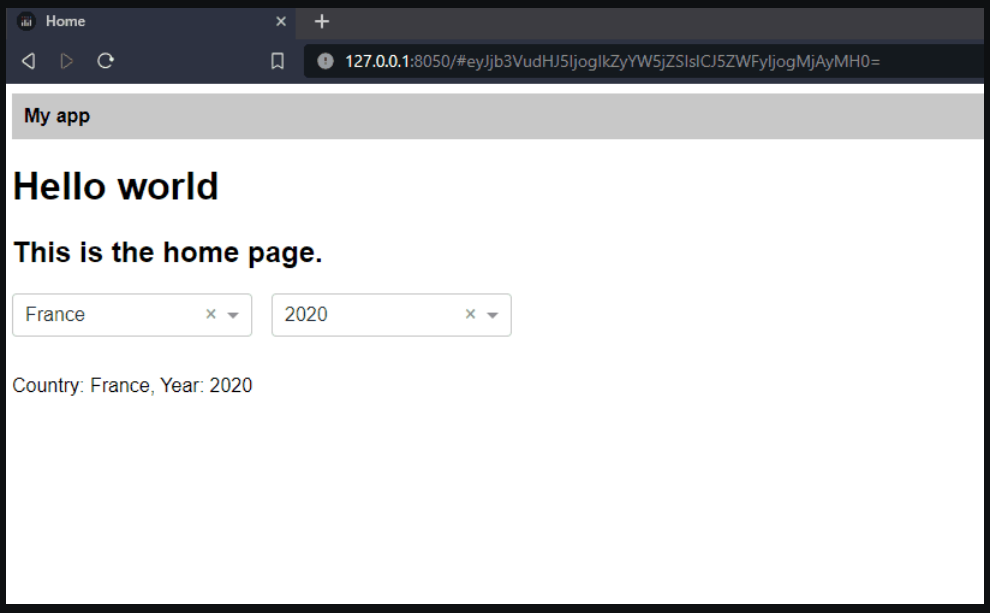
```
from dash import Dash, Input, Output, State, callback, dcc, html, page_container

TRANSLATIONS = {
    "en": {
        "title": "My app",
    },
    "fr": {
        "title": "Mon app",
    },
}
DEFAULT_LANGUAGE = "en"

app = Dash(
   __name__,
   use_pages=True,
   routing_callback_inputs={
        # The language will be passed as a `layout` keyword argument to page layout functions
        "language": Input("language", "value"),
   },
)

app.layout = html.Div(
    [
        html.Div(
            [
                # We will need to update the title when the language changes as it is
                # rendered outside the page layout function
                html.Div(TRANSLATIONS[DEFAULT_LANGUAGE]["title"], id="app-title", style={"font
                html.Div(
                    [
                        # Language dropdown
                        dcc.Dropdown(
                            id="language",
                            options=[
                                {"label": "English", "value": "en"},
                                {"label": "Français", "value": "fr"},
                            ],
                            value=DEFAULT_LANGUAGE,
                            persistence=True,
                            clearable=False,
                            searchable=False,
                            style={"minWidth": 150},
                        ),
                    ],
                    style={"marginLeft": "auto", "display": "flex", "gap": 10}
                )
            ],
            style={
                "background": "#CCC",
                "padding": 10,
                "marginBottom": 20,
                "display": "flex",
```

pages/home.py

```
from dash import Input, Output, State, callback, dcc, html, register_page

register_page(__name__, "/")

TRANSLATIONS = {
    "en": {
        "title": "Hello world",
        "subtitle": "This is the home page.",
        "country": "Country",
        "year": "Year",
    },
    "fr": {
        "title": "Bonjour le monde",
        "subtitle": "Ceci est la page d'accueil.",
        "country": "Pays",
        "year": "Année",
    }
}
```

```
def layout(language: str = "en", **_kwargs):
    """Home page layout

    It takes in a keyword arguments defined in `routing_callback_inputs`:
    * language (input coming from the language dropdown), it triggers re-render
    """
    default_country = "USA"
    default_year = 2020

    return [
        html.H1(TRANSLATIONS.get(language, {}).get("title")),
        html.H2(TRANSLATIONS.get(language, {}).get("subtitle")),
        html.Div(
            [
                dcc.Dropdown(
                    id={"type": "control", "id": "country"},
                    value=default_country,
                    options=["France", "USA", "Canada"],
                    style={"minWidth": 200},
                ),
                dcc.Dropdown(
                    id={"type": "control", "id": "year"},
                    value=default_year,
                    options=[{"label": str(y), "value": y} for y in range(1980, 2021)],
                    style={"minWidth": 200},
                ),
            ],
            style={"display": "flex", "gap": "1rem", "marginBottom": "2rem"},
        ),
        html.Div(contents(language, country=default_country, year=default_year), id="contents'
    ]

def contents(language: str, country: str, year: int, **_kwargs):
```

Example 2: Serializing the App State in the URL Hash



app.py

```
from dash import Dash, Input, Output, State, callback, dcc, html, page_container

app = Dash(
    __name__,
    use_pages=True,
    routing_callback_inputs={
        # The app state is serialised in the URL hash without refreshing the page
        # This URL can be copied and then parsed on page load
        "state": State("main-url", "hash"),
    },
)

app.layout = html.Div(
```

```
    [
        dcc.Location(id="main-url"),
        html.Div(
            html.Div("My app", style={"fontWeight": "bold"}),
            style={"background": "#CCC", "padding": 10, "marginBottom": 20},
        ),
        page_container,
    ],
    style={"fontFamily": "sans-serif"}
)

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

`pages/home.py`

```
import base64
import json

from dash import ALL, Input, Output, callback, html, dcc, register_page, ctx

register_page(__name__, "/", title="Home")

def layout(state: str = None, **_kwargs):
    """Home page layout

    It takes in a keyword arguments defined in `routing_callback_inputs`:
    * state (serialised state in the URL hash), it does not trigger re-render
    """

    # Define default state values
    defaults = {"country": "France", "year": 2020}
    # Decode the state from the hash
    state = defaults | (json.loads(base64.b64decode(state)) if state else {})

    return [
        html.H1("Hello world"),
        html.H2("This is the home page."),
        html.Div(
            [
                dcc.Dropdown(
                    id={"type": "control", "id": "country"},
                    value=state.get("country"),
                    options=["France", "USA", "Canada"],
                    style={"minWidth": 200},
                ),
                dcc.Dropdown(
                    id={"type": "control", "id": "year"},
                    value=state.get("year"),
                    options=[{"label": str(y), "value": y} for y in range(1980, 2021)],
                    style={"minWidth": 200},
                ),
            ],
            style={"display": "flex", "gap": "1rem", "marginBottom": "2rem"},
        ),
        html.Div(contents(**state), id="contents")
    ]

def contents(country: str, year: int, **_kwargs):
    return f"Country: {country}, Year: {year}"

@callback(
    Output("main-url", "hash", allow_duplicate=True),
    Input({"type": "control", "id": ALL}, "value"),
    prevent_initial_call=True,
)
def update_hash(_values):
    """Update the hash in the URL Location component to represent the app state.
```

Reference for dash.register_page

Assigns the variables to `dash.page_registry` as an `OrderedDict` (ordered by `order`).

`dash.page_registry` is used by `pages_plugin` to set up the layouts as a multi-page Dash app. This includes the URL routing callbacks (using `dcc.Location`) and the HTML templates to include title, meta description, and the meta description image.

`dash.page_registry` can also be used by Dash developers to create the page navigation links or by template authors.

- `module`: The module path where this page's `layout` is defined. Often `__name__`.
- `path`: URL Path, e.g. `/` or `/home-page`. If not supplied, will be inferred from the `path_template` or `module`, e.g. based on path_template: `/asset/<asset_id` to `/asset/none` e.g. based on module: `pages.weekly_analytics` to `/weekly-analytics`
- `relative_path`: The path with `requests_pathname_prefix` prefixed before it. Use this path when specifying local URL paths that will work in environments regardless of what `requests_pathname_prefix` is. In some deployment environments, like Dash Enterprise, `requests_pathname_prefix` is set to the application name, e.g. `my-dash-app`. When working locally, `requests_pathname_prefix` might be unset and so a relative URL like `/page-2` can just be `/page-2`. However, when the app is deployed to a URL like `/my-dash-app`, then `relative_path` will be `/my-dash-app/page-2`.
- `path_template`: Add variables to a URL by marking sections with <variable_name>. The layout function then receives the <variable_name> as a keyword argument. e.g. path_template= "/asset/<asset_id>" then if pathname in browser is "/assets/a100" then layout will receive **{"asset_id":"a100"}
- `name`: The name of the link. If not supplied, will be inferred from `module`, e.g. `pages.weekly_analytics` to `Weekly analytics`
- `order`: The order of the pages in `page_registry`. If not supplied, then the filename is used and the page with path `/` has order `0`
- `title`: (string or function) Specifies the page title displayed in the browser tab. If not supplied, the app's title is used if different from the default "Dash". Otherwise, the title is the given `name` or inferred from the module name. For example, `pages.weekly_analytics` is inferred as "Weekly Analytics".
- `description`: (string or function) The . If not defined, the application description will be used if available.
- `image`: The meta description image used by social media platforms. If not supplied, then it looks for the following images in `assets/`:
    - A page specific image: `assets/<module>.<extension>` is used, e.g. `assets/weekly_analytics.png`
    - A generic app image at `assets/app.<extension>`
    - A logo at `assets/logo.<extension>` When inferring the image file, it will look for the following extensions: APNG, AVIF, GIF, JPEG, JPG, PNG, SVG, WebP.
- `image_url`: Overrides the image property and sets the `<image>` meta tag to the provided image URL.
- `redirect_from`: A list of paths that should redirect to this page. For example: `redirect_from=['/v2', '/v3']`
- `layout`: The layout function or component for this page. If not supplied, then looks for `layout` from within the supplied `module`.
- `**kwargs`: Arbitrary keyword arguments that can be stored

`page_registry` stores the original property that was passed in under `supplied_<property>` and the coerced property under `<property>`. For example, if this was called:

```
register_page(
    'pages.historical_outlook',
    name='Our historical view',
    custom_key='custom value'
)
```

Then this will appear in `page_registry`:

```
OrderedDict([
    (
        'pages.historical_outlook',
```

```
        dict(
            module='pages.historical_outlook',

            supplied_path=None,
            path='/historical-outlook',

            supplied_name='Our historical view',
            name='Our historical view',

            supplied_title=None,
            title='Our historical view'

            supplied_layout=None,
            layout=<function pages.historical_outlook.layout>,

            custom_key='custom value'
        )
    ),
])
```

Multi-Page Apps without Pages

> Dash Pages (available in Dash 2.5 and later) is the easiest way to build a multi-page app in Dash. If you are using an earlier version of Dash 2.x, you can build a multi-page app using the following guide.
>
> Dash Pages uses a `dcc.Location` callback under-the-hood as described in the method below. Dash Pages also automatically:
>
> - Sets configurable `title`, `description`, and `image` meta tags using `interpolate_index` and clientside callbacks under-the-hood
> - Sets configurable redirects using `flask.redirect` under-the-hood
> - Sets configurable 404 content
> - Sets `validate_layout` under-the-hood to avoid callback exceptions See the **community announcement for the original discussion of this feature**.

The components **dcc.Location** and **dcc.Link** aid page navigation: `dcc.Location` represents the web browser address bar. You can access the current pathname in the user's browser with `dcc.Location`'s `pathname` property. `dcc.Link` updates the `pathname` in the browser.

In the following examples, we demonstrate using these components to build multi-page apps, without using Dash Pages.

Simple Example

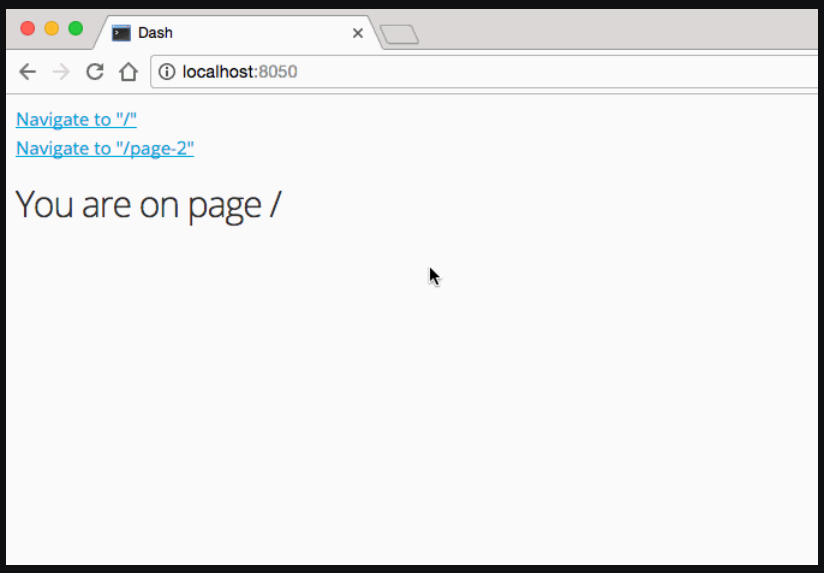
```
from dash import Dash, dcc, html, callback, Input, Output

app = Dash()

app.layout = html.Div([
    # represents the browser address bar and doesn't render anything
    dcc.Location(id='url', refresh=False),

    dcc.Link('Navigate to "/"', href='/'),
    html.Br(),
    dcc.Link('Navigate to "/page-2"', href='/page-2'),

    # content will be rendered in this element
    html.Div(id='page-content')
])


@callback(Output('page-content', 'children'), Input('url', 'pathname'))
def display_page(pathname):
    return html.Div([
        html.H3(f'You are on page {pathname}')
    ])
```

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

In this example, the callback `display_page` receives the current pathname (the last part of the URL) of the page. The callback simply displays the `pathname` on page, but it could use the `pathname` to display different content.

The `Link` element updates the `pathname` of the browser *without refreshing the page*. If you used a `html.A` element instead, the `pathname` would update but the page would refresh.

Here is what this example running looks like. Note how clicking on the `Link` doesn't refresh the page even though it updates the URL!

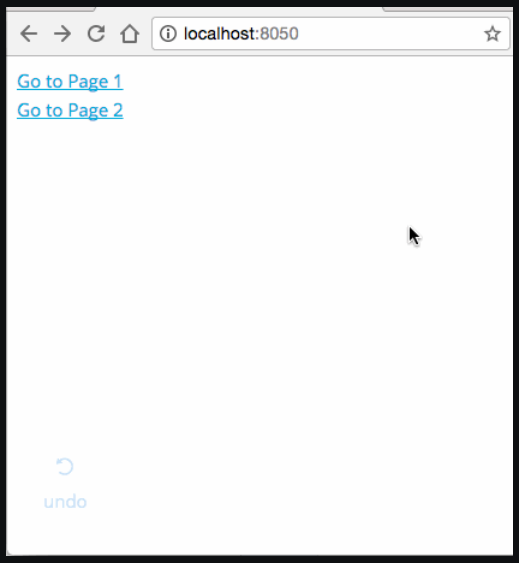


Example With Different Pages

You can modify the previous example to display different pages depending on the URL:

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

app = Dash(__name__, suppress_callback_exceptions=True)

app.layout = html.Div([
    dcc.Location(id='url', refresh=False),
    html.Div(id='page-content')
])

index_page = html.Div([
    dcc.Link('Go to Page 1', href='/page-1'),
    html.Br(),
    dcc.Link('Go to Page 2', href='/page-2'),
])

page_1_layout = html.Div([
    html.H1('Page 1'),
    dcc.Dropdown(['LA', 'NYC', 'MTL'], 'LA', id='page-1-dropdown'),
    html.Div(id='page-1-content'),
    html.Br(),
    dcc.Link('Go to Page 2', href='/page-2'),
    html.Br(),
    dcc.Link('Go back to home', href='/'),
])


@callback(Output('page-1-content', 'children'),
          Input('page-1-dropdown', 'value'))
def page_1_dropdown(value):
    return f'You have selected {value}'


page_2_layout = html.Div([
    html.H1('Page 2'),
```

```
    dcc.RadioItems(['Orange', 'Blue', 'Red'], 'Orange', id='page-2-radios'),
    html.Div(id='page-2-content'),
    html.Br(),
    dcc.Link('Go to Page 1', href='/page-1'),
    html.Br(),
    dcc.Link('Go back to home', href='/')
])


@callback(Output('page-2-content', 'children'), Input('page-2-radios', 'value'))
def page_2_radios(value):
    return f'You have selected {value}'


# Update the index
@callback(Output('page-content', 'children'), Input('url', 'pathname'))
def display_page(pathname):
```



In this example, we're displaying different layouts through the `display_page` function. A few notes:

- Each page can have interactive elements even though those elements may not be in the initial view. Dash handles these "dynamically generated" components gracefully: as they are rendered, they will trigger the callbacks with their initial values.
- Since we're adding callbacks to elements that don't exist in the app.layout, Dash will raise an exception to warn us that we might be doing something wrong. In this case, we're adding the elements through a callback, so we can ignore the exception by setting `suppress_callback_exceptions=True`. It is also possible to do this without suppressing callback exceptions. See the example below for details.
- You can modify this example to import the different page's `layout`s in different files.
- This Dash Userguide that you're looking at is itself a multi-page Dash app, using these same principles.

Dynamically Create a Layout for Multi-Page App Validation

Dash applies validation to your callbacks, which performs checks such as validating the types of callback arguments and checking to see whether the specified Input and Output components actually have the specified properties.

For full validation, all components within your callback must therefore appear in the initial layout of your app, and you will see an error if they do not. However, in the case of more complex Dash apps that involve dynamic modification of the layout (such as multi-page apps), not every component appearing in your callbacks will be included in the initial layout.

New in Dash 1.12 You can set `app.validation_layout` to a "complete" layout that contains all the components you'll use in any of the pages / sections. `app.validation_layout` must be a Dash component, not a function. Then set `app.layout` to just the index layout. In previous Dash versions there was a trick you could use to achieve the same result, checking `flask.has_request_context` inside a layout function - that will still work but is no longer recommended.

```
from dash import Dash, html, dcc, Input, Output, State, callback
```

```
app = Dash()

url_bar_and_content_div = html.Div([
    dcc.Location(id='url', refresh=False),
    html.Div(id='page-content')
])

layout_index = html.Div([
    dcc.Link('Navigate to "/page-1"', href='/page-1'),
    html.Br(),
    dcc.Link('Navigate to "/page-2"', href='/page-2'),
])

layout_page_1 = html.Div([
    html.H2('Page 1'),
    dcc.Input(id='input-1-state', type='text', value='Montreal'),
    dcc.Input(id='input-2-state', type='text', value='Canada'),
    html.Button(id='submit-button', n_clicks=0, children='Submit'),
    html.Div(id='output-state'),
    html.Br(),
    dcc.Link('Navigate to "/"', href='/'),
    html.Br(),
    dcc.Link('Navigate to "/page-2"', href='/page-2'),
])

layout_page_2 = html.Div([
    html.H2('Page 2'),
    dcc.Dropdown(['LA', 'NYC', 'MTL'], 'LA', id='page-2-dropdown'),
    html.Div(id='page-2-display-value'),
    html.Br(),
    dcc.Link('Navigate to "/"', href='/'),
    html.Br(),
    dcc.Link('Navigate to "/page-1"', href='/page-1'),
])

# index layout
app.layout = url_bar_and_content_div

# "complete" layout
app.validation_layout = html.Div([
    url_bar_and_content_div,
    layout_index,
    layout_page_1,
    layout_page_2,
])


# Index callbacks
@callback(Output('page-content', 'children'),
          Input('url', 'pathname'))
```

Structuring a Multi-Page App

Earlier examples show each multi-page app created within a single Python file. For bigger apps, a structure with multiple files may make the app easier to manage.

One Page Per File

One way to structure a multi-page app is to have each page as a separate app imported in the main app (`app.py`). In the following example, we build our app with two pages `pages/page1.py` and `pages/page2.py`. More pages (for example, `pages/page3.py`) can easily be added to this structure.

File structure:

```
- app.py
- pages
   |-- __init__.py
   |-- page1.py
   |-- page2.py
```

`pages/page1.py`

```
from dash import dcc, html, Input, Output, callback

layout = html.Div([
    html.H3('Page 1'),
    dcc.Dropdown(
        {f'Page 1 - {i}': f'{i}' for i in ['New York City', 'Montreal', 'Los Angeles']},
        id='page-1-dropdown'
    ),
    html.Div(id='page-1-display-value'),
    dcc.Link('Go to Page 2', href='/page2')
])


@callback(
    Output('page-1-display-value', 'children'),
    Input('page-1-dropdown', 'value'))
def display_value(value):
    return f'You have selected {value}'
```

`pages/page2.py`

```
from dash import dcc, html, Input, Output, callback

layout = html.Div([
    html.H3('Page 2'),
    dcc.Dropdown(
        {f'Page 2 - {i}': f'{i}' for i in ['London', 'Berlin', 'Paris']},
        id='page-2-dropdown'
    ),
    html.Div(id='page-2-display-value'),
    dcc.Link('Go to Page 1', href='/page1')
])


@callback(
    Output('page-2-display-value', 'children'),
    Input('page-2-dropdown', 'value'))
def display_value(value):
    return f'You have selected {value}'
```

`app.py`

In `app.py` we import `page1` and `page2`. When you run `app.py` it loads the layout from `page1.py` if you go to the pathname `/page1` and the layout from `page2.py` if you go to `/page2`.

```
from dash import Dash, dcc, html, Input, Output, callback
from pages import page1, page2

app = Dash(__name__, suppress_callback_exceptions=True)

app.layout = html.Div([
    dcc.Location(id='url', refresh=False),
    html.Div(id='page-content')
])


@callback(Output('page-content', 'children'),
          Input('url', 'pathname'))
def display_page(pathname):
    if pathname == '/page1':
        return page1.layout
    elif pathname == '/page2':
        return page2.layout
    else:
        return '404'


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

Flat Project Structure

Another option for a multi-page structure is a flat project layout with callbacks and layouts in separate files:

File structure:

```
- app.py
- callbacks.py
- layouts.py
```

`app.py`

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

from layouts import layout1, layout2
import callbacks

app = Dash(__name__, suppress_callback_exceptions=True)

app.layout = html.Div([
    dcc.Location(id='url', refresh=False),
    html.Div(id='page-content')
])


@callback(Output('page-content', 'children'),
          Input('url', 'pathname'))
def display_page(pathname):
    if pathname == '/page1':
        return layout1
    elif pathname == '/page2':
        return layout2
    else:
        return '404'


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

`callbacks.py`

```
from dash import Input, Output, callback


@callback(
    Output('page-1-display-value', 'children'),
    Input('page-1-dropdown', 'value'))
def display_value(value):
    return f'You have selected {value}'


@callback(
    Output('page-2-display-value', 'children'),
    Input('page-2-dropdown', 'value'))
def display_value(value):
    return f'You have selected {value}'
```

`layouts.py`

```
from dash import dcc, html

layout1 = html.Div([
    html.H3('Page 1'),
    dcc.Dropdown(
        {f'Page 1 - {i}': f'{i}' for i in ['New York City', 'Montreal', 'Los Angeles']},
        id='page-1-dropdown'
    ),
    html.Div(id='page-1-display-value'),
    dcc.Link('Go to Page 2', href='/page2')
```

```
])

layout2 = html.Div([
    html.H3('Page 2'),
    dcc.Dropdown(
        {f'Page 2 - {i}': f'{i}' for i in ['London', 'Berlin', 'Paris']},
        id='page-2-dropdown'
    ),
    html.Div(id='page-2-display-value'),
    dcc.Link('Go to Page 1', href='/page1')
])
```

Dash Python **> *Multi-Page Apps and URL Support***