plotly

Star 23,447

Dash Python > ***Dash Testing***

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

Dash Testing

`dash.testing` provides custom Dash **pytest** fixtures and a set of testing APIs for unit and end-to-end testing. This tutorial shows how to write and run tests for a Dash app.

Installing `dash.testing`

Install `dash.testing` with:

```
python -m pip install dash[testing]
```

In some shells (for example, Zsh), you may need to escape the opening bracket, `[`:

```
python -m pip install dash\[testing]
```

Unit Tests

> New in Dash 2.6

`dash.testing` supports unit testing of callbacks. Use callback unit tests to confirm that a callback's outputs are as expected.

In the following example, we write two tests for our app. We have an app, `app.py`, that has two callbacks. The `update` callback outputs to an `html.Div` the number of times `btn-1` and `btn-2` have been clicked. The `display` callback uses **callback_context** to determine which input triggered the callback, adds the `id` to a string, and outputs it to an `html.Div`.

`app.py`

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

app = Dash()

app.layout = html.Div([
    html.Button('Button 1', id='btn-1'),
    html.Button('Button 2', id='btn-2'),
    html.Button('Button 3', id='btn-3'),
    html.Div(id='container'),
    html.Div(id='container-no-ctx')
])

@callback(
    Output('container-no-ctx', 'children'),
    Input('btn-1', 'n_clicks'),
    Input('btn-2', 'n_clicks'))
def update(btn1, btn2):
    return f'button 1: {btn1} & button 2: {btn2}'
```

```
@callback(Output('container','children'),
              Input('btn-1', 'n_clicks'),
              Input('btn-2', 'n_clicks'),
              Input('btn-3', 'n_clicks'))
def display(btn1, btn2, btn3):
    button_clicked = ctx.triggered_id
    return f'You last clicked button with ID {button_clicked}'

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

In the test file, `test_app_callbacks.py`, we create two test cases, `test_update_callback` and `test_display_callback`.

```
from contextvars import copy_context
from dash._callback_context import context_value
from dash._utils import AttributeDict

# Import the names of callback functions you want to test
from app import display, update

def test_update_callback():
    output = update(1, 0)
    assert output == 'button 1: 1 & button 2: 0'

def test_display_callback():
    def run_callback():
        context_value.set(AttributeDict(**{"triggered_inputs": [{"prop_id": "btn-1-ctx-example.
        return display(1, 0, 0)

    ctx = copy_context()
    output = ctx.run(run_callback)
    assert output == f'You last clicked button with ID btn-1-ctx-example'
```

Run the tests with the command:

```
pytest
```

Notes:

- Our app is a regular app. We don't need to add anything extra to `app.py` for our tests.
- In the test file, `test_app_callbacks.py`, we import the callback functions to test, with `from app import display, update`.
- The first test case, `test_update_callback`, calls the `update` function with the values `1` and `0` and stores the result in the variable `output`.

  Looking at our app (`app.py`), we can see these inputs are the values of `n_clicks` for each of the buttons:

  ```
  @callback(
      Output('container-no-ctx', 'children'),
      Input('btn-1', 'n_clicks'),
      Input('btn-2', 'n_clicks'))
  def update(btn1, btn2):
      return f'button 1: {btn1} & button 2: {btn2}'
  ```

  The test function then uses `assert` to confirm that the `output` is as expected, that it is equal to `'button 1: 1 & button 2: 0'`.

Mocking Callback Context

The second test case uses `callback_context`. To test the callback, we need to mock `callback_context`.

To do this, we need the following additional imports:

```
from contextvars import copy_context
from dash._callback_context import context_value
from dash._utils import AttributeDict
```

In the example:

- We use **contextvars.copy_context** to create a copy of the current context and save it to the variable `ctx`.
- We then use `ctx.run` to call our `run_callback` function in that context and save it to the `output` variable.
- Within the `run_callback` function, we use `context_value.set` to define which inputs trigger the callback in our test, and call the `display` callback. `context_value.set` takes an `AttributeDict` (from `dash._utils`).
- The final part of the test checks if the `output` equals `'You last clicked button with ID btn-1-ctx-example'`.

End-to-End Tests

`dash.testing` also supports end-to-end tests. End-to-end tests run programmatically, start a real browser session, and click through the Dash app UI. They are slower to run than unit tests and more complex to set up but more closely mimic the end user's experience.

If you're running end-to-end tests, you'll also need to install a **WebDriver**, unless you're planning on running your tests with a **Selenium Grid**. Your tests will use either your locally installed WebDriver or the remote WebDriver on your grid to interact with the browser. See the **Running Tests** section below for details on how to run tests using a Selenium Grid.

Installing a WebDriver

We recommend the ChromeDriver WebDriver, which we use for `dash` end-to-end tests. Install ChromeDriver by following the **ChromeDriver Getting Started Guide**. Ensure you install the correct version of ChromeDriver for the version of Chrome you have installed.

Note: **Mozilla Firefox geckodriver** is also supported. To run your tests with geckodriver, you'll need to add a flag when running tests. See the **Running Tests** section below for details on running tests with geckodriver.

> Note: The Gecko (Marionette) driver from Mozilla is not fully compatible with Selenium specifications. Some features may not work as expected.

Example - Basic Test

Here we create a test case where the browser driver opens the app, waits for an element with the `id`

`nully-wrapper` to be displayed, confirms its text equals `"0"`, and that there are no errors in the browser console.

```
# 1. imports of your dash app
import dash
from dash import html
# 2. give each testcase a test case ID, and pass the fixture
# dash_duo as a function argument
def test_001_child_with_0(dash_duo):
    # 3. define your app inside the test function
    app = dash.Dash()
    app.layout = html.Div(id="nully-wrapper", children=0)
    # 4. host the app locally in a thread, all dash server configs could be
    # passed after the first app argument
    dash_duo.start_server(app)
    # 5. use wait_for_* if your target element is the result of a callback,
    # keep in mind even the initial rendering can trigger callbacks
    dash_duo.wait_for_text_to_equal("#nully-wrapper", "0", timeout=4)
    # 6. use this form if its present is expected at the action point
    assert dash_duo.find_element("#nully-wrapper").text == "0"
```

```
    # 7. to make the checkpoint more readable, you can describe the
    # acceptance criterion as an assert message after the comma.
    assert dash_duo.get_logs() == [], "browser console should contain no error"
    # 8. visual testing with percy snapshot
    dash_duo.percy_snapshot("test_001_child_with_0-layout")
```

Notes:

1. For most test scenarios, you don't need to import any modules for the test; just import what you need for the Dash app itself.
2. A test case is a regular Python function. The function name follows this pattern: `test_{tcid}_{test title}`. The `tcid` (test case ID) is an abbreviation pattern of `mmffddd => module + file + three digits`. The `tcid` facilitates the test selection by just running `pytest -k {tcid}`. Its naming convention also helps code navigation with modern editors.
3. Here we just define our app inside a test function. All the rules still apply as in your app file.
4. We start the test by calling the `start_server` API from `dash_duo`. When the test starts:
    1. The defined app is hosted inside a light Python `threading.Thread`.
    2. A Selenium WebDriver is initialized and navigates to the local server URL using `server_url`.
    3. We first wait until the Flask server is responsive to an HTTP request, and then make sure the Dash app is fully rendered inside the browser.
5. A test case is composed of preparation, actions, and checkpoints. Both #5 and #6 are doing the same check in this example; we are expecting that the defined `Div` component's text is identical to `children`. #5 will wait for the expected state to be reached within a 4 seconds timeout. It's a safer way to write the action steps when you are doing an element check related to callbacks, as it normally happens under Dash context: the element is already present in the DOM, but not necessarily the props.
6. The `find_element` API call has an implicit global timeout of two seconds set at the driver level, i.e. the driver waits at most two seconds to find the element by the locator, **HOWEVER** it will compare the text as soon as the driver returns an element. The API `find_element('#nully-wrapper')` is a shortcut to `driver.find_element_by_css_selector('#nully-wrapper')`.
7. Unlike `unittest`, `pytest` allows you to use the standard Python `assert` for verifying expectations and values. It also puts more introspection information into the assertion failure message by overriding the `assert` behavior. It's good practice to expose your *acceptance criteria* directly in the test case rather than wrapping the `assert` inside another helper API, also to write these messages with SHOULD/SHOULD NOT without failure confusion. By looking at the test name, the app definition, the actions, and the checkpoints, reviewers should figure out easily the purpose of the test.
8. We use **Percy** as our Visual Regression Testing tool. It's a good alternative to assertions when your checkpoint is about the graphical aspects of a Dash App, such as the whole layout or a `dcc.Graph` component. We integrate the Percy service with a `PERCY_TOKEN` variable, so the regression result is only available in Plotly's CircleCI setup.

Example – Deployed Dash App

Use the `dash_br` fixture to test a deployed app. Set `dash_br.server_url` to the URL of the app:

```
def test_002_nav_bar(dash_br):
    dash_br.server_url = "https://dash-example-index.herokuapp.com/"
    dash_br.wait_for_text_to_equal(".navbar-brand", "Dash Example Index", timeout=20)
    assert dash_br.find_element(".navbar-brand").text == "Dash Example Index"
```

Running Tests

There are many ways to run your tests, and you can change how your tests run by adding flags to the `pytest` command.

All Tests

You can run all tests in the current working directory (and its subdirectories) with:

```
pytest
```

Any tests in .py files with names that start with *test_* or end with *_test* are run.

Specific Tests

In the **Basic Test** example above, we gave our test case a test case ID, `001`. We can use this to run that specific test:

```
pytest -k 001
```

WebDriver Options

There are different ways to configure your WebDriver when running end-to-end tests.

With an Alternative Web Driver

ChromeDriver is the default WebDriver, but `dash.testing` also supports geckodriver for Firefox. Add the `--webdriver Firefox` flag when running tests to use it:

```
pytest --webdriver Firefox -k 001
```

In Headless Mode

You can run tests in **headless** mode, if you don't need to see the tests in a browser UI:

```
pytest --headless -k 001
```

Using a Selenium Grid

You can use Selenium Grid to run tests across multiple machines. To run tests with a local hosted grid at `http://localhost:4444/wd/hu`:

```
pytest --remote -k 001
```

`http://localhost:4444/wd/hu` is the default value. To add an different remote, use the `--remote-url` flag:

```
pytest --webdriver Firefox --remote-url https://grid_provider_endpoints
```

Note: If you set `--remote-url`, and the value is different to the default value, `http://localhost:4444/wd/hu`, you don't need to include the `--remote` flag.

Caveats

We can't guarantee that the above examples work with every Selenium Grid. There may be limitations because of how your network is configured. For example, because of:

- A different hosting OS
- How docker-compose is configured

If you encounter issues, it may be because you need to do some auxiliary WebDriver options tuning to run the tests. Here are some things to try:

- Change the default logging level with `--log-cli-level DEBUG`.
- Customize your browser options. There is a back door for browser option customization by a `pytest_setup_options` hook defined in `plugin.py`. The example below is to use the headless mode with Chrome WebDriver in Windows. There is a workaround by adding `--disable-gpu` in the options.

```
from selenium.webdriver.chrome.options import Options

def pytest_setup_options():
        options = Options()
        options.add_argument('--disable-gpu')
        return options
```

Fixtures

To avoid accidental name collision with other pytest plugins, all Dash test fixtures start with the prefix `dash`.

- `dash_br` A standalone WebDriver wrapped with high-level Dash testing APIs. This is suitable for testing a Dash App in a deployed environment (Dash for Python or R), i.e. when your Dash App is accessible from a URL.
- `dash_duo` The default fixture for Dash Python integration tests, it contains a `thread_server` and a WebDriver wrapped with high-level Dash testing APIs.
- `dash_thread_server` Start your Dash App locally in a Python `threading.Thread`, which is lighter and faster than a process.
- `dash_process_server` This is close to your production/deployed environment. Start your Dash App with `waitress` (by default if `raw_command` is not provided) in a Python `subprocess`. You can control the process runner with two supplemental arguments. To run the application with alternative deployment options, use the `raw_command` argument; to extend the timeout if your application needs more than the default three seconds to launch, use the `start_timeout` argument. Note: *You need to configure your PYTHONPATH so that the Dash app source file is directly importable.*

APIs

Selenium Overview

Both `dash_duo` and `dash_br` expose the Selenium WebDriver via the property `driver`, e.g. `dash_duo.driver`, which gives you full access to the **Python Selenium API**. (*Note that this is not the official Selenium documentation site, but has somehow become the de facto Python community reference*) One of the core components of Selenium testing is finding the **web element** with a `locator`, and performing some actions like `click` or `send_keys` on it, and waiting to verify if the expected state is met after those actions. The check is considered as an acceptance criterion, for which you can write in a standard Python `assert` statement.

Element Locators

There are several strategies to **locate elements**; CSS selector and XPATH are the two most versatile ways. We recommend using the **CSS Selector** in most cases due to its **better performance and robustness** across browsers. If you are new to using CSS Selectors, these **SauceLab tips** are a great start. Also, remember that **Chrome Dev Tools Console** is always your good friend and playground.

Waits

This link covers this topic nicely. For impatient readers, a quick take away is quoted as follows: The Selenium WebDriver provides two types of waits:

- **explicit wait** Makes WebDriver wait for a certain condition to occur before proceeding further with execution. All our APIs with `wait_for_*` falls into this category.
- **implicit wait** Makes WebDriver poll the DOM for a certain amount of time when trying to locate an element. We set a global two-second timeout at the `driver` level. **Note** *all custom wait conditions are defined in `dash.testing.wait` and there are two extra APIs `until` and `until_not` which are similar to the explicit wait*

with WebDriver, but they are not binding to WebDriver context, i.e. they abstract a more generic mechanism to poll and wait for certain condition to happen

Browser APIs

This section lists a minimal set of browser testing APIs. They are convenient shortcuts to Selenium APIs and have been approved in our daily integration tests. The following table might grow as we start migrating more legacy tests in the near future. But we have no intention to build a comprehensive list, the goal is to make writing Dash tests concise and error-free. Please feel free to submit a community PR to add any missing ingredient, we would be happy to accept that if it's adequate for Dash testing.

API	Description
`find_element(selector)`	return the first found element by the `CSS selector`, shortcut to `driver.find_element_by_css_selector`. *note that this API will raise exceptions if not found, the `find_elements` API returns an empty list instead*
`find_elements(selector)`	return a list of all elements matching by the `CSS selector`, shortcut to `driver.find_elements_by_css_selector`
`multiple_click(selector, clicks)`	find the element with the `CSS selector` and clicks it with number of `clicks`
`wait_for_element(selector, timeout=None)`	shortcut to `wait_for_element_by_css_selector` the long version is kept for back compatibility. `timeout` if not set, equals to the fixture's `wait_timeout`
`wait_for_element_by_css_selector(selector, timeout=None)`	explicit wait until the element is present, shortcut to `WebDriverWait` with `EC.presence_of_element_located`
`wait_for_element_by_id(element_id, timeout=None)`	explicit wait until the element is present, shortcut to `WebDriverWait` with `EC.presence_of_element_located`
`wait_for_style_to_equal(selector, style, value, timeout=None)`	explicit wait until the element's style has expected `value`. shortcut to `WebDriverWait` with custom wait condition `style_to_equal`. `timeout` if not set, equals to the fixture's `wait_timeout`
`wait_for_text_to_equal(selector, text, timeout=None)`	explicit wait until the element's text equals the expected `text`. shortcut to `WebDriverWait` with custom wait condition `text_to_equal`. `timeout` if not set, equals to the fixture's `wait_timeout`
`wait_for_contains_text(selector, text, timeout=None)`	explicit wait until the element's text contains the expected `text`. shortcut to `WebDriverWait` with custom wait condition `contains_text` condition. `timeout` if not set, equals to the fixture's `wait_timeout`
`wait_for_class_to_equal(selector, classname, timeout=None)`	explicit wait until the element's class has expected `value`. timeout if not set, equals to the fixture's `wait_timeout`. shortcut to `WebDriverWait` with custom `class_to_equal` condition.
`wait_for_contains_class(selector, classname, timeout=None)`	explicit wait until the element's classes contains the expected `classname`. timeout if not set, equals to the fixture's `wait_timeout`. shortcut to `WebDriverWait` with custom `contains_class` condition.
`wait_for_page(url=None, timeout=10)`	navigate to the `url` in webdriver and wait until the dash renderer is loaded in browser. use `server_url`

API	Description
	if `url` is None
`toggle_window()`	switch between the current working window and the newly opened one.
`switch_window(idx)`	switch to window by window index. shortcut to `driver.switch_to.window`. raise `BrowserError` if no second window present in browser
`open_new_tab(url=None)`	open a new tab in browser with window name `new window`. `url` if not set, equals to `server_url`
`percy_snapshot(name, wait_for_callbacks=False)`	visual test API shortcut to `percy_runner.snapshot`. it also combines the snapshot `name` with the actual python versions. The `wait_for_callbacks` parameter controls whether the snapshot is taken only after all callbacks have fired; the default is `False`.
`visit_and_snapshot(resource_path, hook_id, wait_for_callbacks=True, assert_check=True)`	This method automates a common task during dash-docs testing: the URL described by `resource_path` is visited, and completion of page loading is assured by waiting until the element described by `hook_id` is fetched. Once `hook_id` is available, `visit_and_snapshot` acquires a snapshot of the page and returns to the main page. `wait_for_callbacks` controls if the snapshot is taken until all dash callbacks are fired, default True. `assert_check` is a switch to enable/disable an assertion that there is no devtools error alert icon.
`take_snapshot(name)`	hook method to take a snapshot while Selenium test fails. the snapshot is placed under `/tmp/dash_artifacts` in Linux or `%TEMP` in windows with a filename combining test case `name` and the running Selenium session id
`zoom_in_graph_by_ratio(elem_or_selector, start_fraction=0.5, zoom_box_fraction=0.2, compare=True)`	zoom out a graph (provided with either a Selenium WebElement or CSS selector) with a zoom box fraction of component dimension, default start at middle with a rectangle of 1/5 of the dimension use `compare` to control if we check the SVG get changed
`click_at_coord_fractions(elem_or_selector, fx, fy)`	Use `ActionChains` to click a Selenium WebElement at a location a given fraction of the way `fx` between its left (0) and right (1) edges, and `fy` between its top (0) and bottom (1) edges.
`get_logs()`	return a list of `SEVERE` level logs after last reset time stamps (default to 0, resettable by `reset_log_timestamp`. **Chrome only**
`clear_input()`	simulate key press to clear the input
`driver`	property exposes the Selenium WebDriver as fixture property
`session_id`	property returns the Selenium session_id, shortcut to `driver.session_id`
`server_url`	set the server_url as setter so the Selenium is aware of the local server port, it also implicitly calls `wait_for_page`. return the server_url as property
`download_path`	property returns the download_path, note that dash fixtures are initialized with a temporary path from pytest `tmpdir`

Dash APIs

This section enumerates a full list of Dash App related properties and APIs apart from the previous browser ones.

API	Description
`devtools_error_count_locator`	property returns the selector of the error count number in the devtool UI
`dash_entry_locator`	property returns the selector of react entry point, it can be used to verify if an Dash app is loaded
`dash_outerhtml_dom`	property returns the BeautifulSoup parsed Dash DOM from outerHTML
`dash_innerhtml_dom`	property returns the BeautifulSoup parsed Dash DOM from innerHTML
`redux_state_paths`	property returns the `window.store.getState().paths`
`redux_state_rqs`	property returns `window.store.getState().requestQueue`
`window_store`	property returns `window.store`
`get_local_storage(store_id="local")`	get the value of local storage item by the id, default is `local`
`get_session_storage(session_id="session")`	get the value of session storage item by the id, default is `session`
`clear_local_storage()`	shortcut to `window.localStorage.clear()`
`clear_session_storage()`	shortcut to `window.sessionStorage.clear()`
`clear_storage()`	clears both local and session storages

Debugging

Verify Your Test Environment

If you run the integration in a virtual environment, make sure you are getting the latest commit in the **master** branch from each component, and that the installed `pip` versions are correct. Note: *We have some enhancement initiatives tracking in this **issue***

Run the CI Job Locally

The **CircleCI Local CLI** is a handy tool to execute some jobs locally. It gives you an earlier warning before even pushing your commits to remote. For example, it's always recommended to pass lint and unit tests job first on your local machine. So we can make sure there are no simple mistakes in the commit.

```
# install the cli (first time only)
$ curl -fLSs https://circle.ci/cli | bash && circleci version
# run at least the lint & unit test job on both python 2 and 3
# note: the current config requires all tests pass on python 2.7, 3.6 and 3.7.
$ circleci local execute --job lint-unit-27 && $ circleci local execute --job lint-unit-37
```

Increase the Verbosity of pytest Logging Level

`pytest --log-cli-level DEBUG -k bsly001`

You can get more logging information from Selenium WebDriver, Flask server, and our test APIs.

```
14:05:41 | DEBUG | selenium.webdriver.remote.remote_connection:388 | DELETE http://127.0.0.1:5
14:05:41 | DEBUG | urllib3.connectionpool:393 | http://127.0.0.1:53672 "DELETE /session/87b6f1e
14:05:41 | DEBUG | selenium.webdriver.remote.remote_connection:440 | Finished Request
14:05:41 | INFO | dash.testing.application_runners:80 | killing the app runner
14:05:41 | DEBUG | urllib3.connectionpool:205 | Starting new HTTP connection (1): localhost:80
14:05:41 | DEBUG | urllib3.connectionpool:393 | http://localhost:8050 "GET /_stop-3ef0e64e8688
```

Selenium Snapshots

If you run your tests with CircleCI dockers (locally with `CircleCI CLI` and/or remotely with `CircleCI`). Inside a docker run or VM instance where there is no direct access to the video card, there is a known limitation that you cannot see anything from the Selenium browser on your screen. Automation developers use **Xvfb** as a workaround to solve this issue. It enables you to run graphical applications without a display (e.g., browser tests on a CI server) while also having the ability to take screenshots. We implemented an automatic hook at the test report stage, it checks if a test case failed with a Selenium test fixture. Before tearing down every instance, it will take a snapshot at the moment where your assertion is `False` or having a runtime error. refer to **Browser APIs** *Note: you can also check the snapshot directly in CircleCI web page under `Artifacts` Tab*

Percy Snapshots

There are two customized `pytest` arguments to tune Percy runner: 1. `--nopercyfinalize` disables the Percy finalize in dash fixtures. This is required if you run your tests in parallel, then you add an extra `percy finalize --all` step at the end. For more details, please visit **Percy Documents**. 2. `--percy-assets` lets Percy know where to collect additional assets such as CSS files.

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