django-plotly-dash Documentation

Mark Gibbs

Jan 25, 2020

Contents

Plotly Dash applications served up in Django templates using tags.

CHAPTER 1

Contents

1.1 Introduction

The purpose of `django-plotly-dash` is to enable Plotly Dash applications to be served up as part of a Django application, in order to provide these features:

- Multiple dash applications can be used on a single page
- Separate instances of a dash application can persist along with internal state
- Leverage user management and access control and other parts of the Django infrastructure
- Consolidate into a single server process to simplify scaling

There is nothing here that cannot be achieved through expanding the Flask app around Plotly Dash, or indeed by using an alternative web framework. The purpose of this project is to enable the above features, given that the choice to use Django has already been made.

The source code can be found in this github repository. This repository also includes a self-contained demo application, which can also be viewed online.

1.1.1 Overview

`django_plotly_dash` works by wrapping around the `dash.Dash` object. The http endpoints exposed by the `Dash` application are mapped to Django ones, and an application is embedded into a webpage through the use of a template tag. Multiple `Dash` applications can be used in a single page.

A subset of the internal state of a `Dash` application can be persisted as a standard Django model instance, and the application with this internal state is then available at its own URL. This can then be embedded into one or more pages in the same manner as described above for stateless applications.

Also, an enhanced version of the `Dash` callback is provided, giving the callback access to the current User, the current session, and also the model instance associated with the application's internal state.

This package is compatible with version 2.0 onwards of Django. Use of the *live updating* feature requires the Django Channels extension; in turn this requires a suitable messaging backend such as Redis.

1.2 Installation

The package requires version 2.0 or greater of Django, and a minimum Python version needed of 3.5.

Use `pip` to install the package, preferably to a local `virtualenv`:

```
pip install django_plotly_dash
```

Then, add `django_plotly_dash` to `INSTALLED_APPS` in the Django `settings.py` file:

```
INSTALLED_APPS = [
    ...
    'django_plotly_dash.apps.DjangoPlotlyDashConfig',
    ...
    ]
```

The project directory name `django_plotly_dash` can also be used on its own if preferred, but this will stop the use of readable application names in the Django admin interface.

Further, if the *header and footer* tags are in use then `django_plotly_dash.middleware.BaseMiddleware` should be added to `MIDDLEWARE` in the same file. This can be safely added now even if not used.

If assets are being served locally through the use of the global `serve_locally` or on a per-app basis, then `django_plotly_dash.middleware.ExternalRedirectionMiddleware` should be added, along with the `whitenoise` package whose middleware should also be added as per the instructions for that package. In addition, `dpd_static_support` should be added to the `INSTALLED_APPS` setting.

The application's routes need to be registered within the routing structure by an appropriate `include` statement in a `urls.py` file:

```
urlpatterns = [
    ...
    path('django_plotly_dash/', include('django_plotly_dash.urls')),
]
```

The name within the URL is not important and can be changed.

For the final installation step, a migration is needed to update the database:

```
./manage.py migrate
```

The `plotly_app` tag in the `plotly_dash` tag library can then be used to render any registered dash component. See *Simple usage* for a simple example.

It is important to ensure that any applications are registered using the `DjangoDash` class. This means that any python module containing the registration code has to be known to Django and loaded at the appropriate time. An easy way to ensure this is to import these modules into a standard Django file loaded at registration time.

1.2.1 Extra steps for live state

The live updating of application state uses the Django Channels project and a suitable message-passing backend. The included demonstration uses `Redis`:

```
pip install channels daphne redis django-redis channels-redis
```

A standard installation of the Redis package is required. Assuming the use of `docker` and the current production version:

```
docker pull redis:4
docker run -p 6379:6379 -d redis
```

The `prepare_redis` script in the root of the repository performs these steps.

This will launch a container running on the localhost. Following the channels documentation, as well as adding `channels` to the `INSTALLED_APPS` list, a `CHANNEL_LAYERS` entry in `settings.py` is also needed:

```
INSTALLED_APPS = [
    ...
    'django_plotly_dash.apps.DjangoPlotlyDashConfig',
    'channels',
    ...
    ]

CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            'hosts': [('127.0.0.1', 6379),],
        },
    },
}
```

The host and port entries in `hosts` should be adjusted to match the network location of the Redis instance.

1.2.2 Further configuration

Further configuration options can be specified through the optional `PLOTLY_DASH` settings variable. The available options are detailed in the *configuration* section.

This includes arranging for Dash assets to be served using the Django `staticfiles` functionality.

1.2.3 Source code and demo

The source code repository contains a *simple demo* application.

To install and run it:

```
git clone https://github.com/GibbsConsulting/django-plotly-dash.git

cd django-plotly-dash

./make_env                 # sets up a virtual environment
                           #   with direct use of the source
                           #   code for the package

./prepare_redis            # downloads a redis docker container
                           #   and launches it with default settings
                           #   *THIS STEP IS OPTIONAL*

./prepare_demo             # prepares and launches the demo
                           #   using the Django debug server
                           #   at http://localhost:8000
```

This will launch a simple Django application. A superuser account is also configured, with both username and password set to `admin`. If the `prepare_redis` step is skipped then the fourth demo page, exhibiting live updating, will not work.

More details on setting up a development environment, which is also sufficient for running the demo, can be found in the *development* section.

Note that the current demo, along with the codebase, is in a prerelease and very raw form. An overview can be found in the *demonstration application* section.‘

1.3 Simple usage

To use existing dash applications, first register them using the `DjangoDash` class. This replaces the `Dash` class from the `dash` package.

Taking a simple example inspired by the excellent getting started guide:

```
import dash
import dash_core_components as dcc
import dash_html_components as html

from django_plotly_dash import DjangoDash

app = DjangoDash('SimpleExample')   # replaces dash.Dash

app.layout = html.Div([
    dcc.RadioItems(
        id='dropdown-color',
        options=[{'label': c, 'value': c.lower()} for c in ['Red', 'Green', 'Blue']],
        value='red'
    ),
    html.Div(id='output-color'),
    dcc.RadioItems(
        id='dropdown-size',
        options=[{'label': i,
                  'value': j} for i, j in [('L','large'), ('M','medium'), ('S','small
↪')]],
        value='medium'
    ),
    html.Div(id='output-size')

])

@app.callback(
    dash.dependencies.Output('output-color', 'children'),
    [dash.dependencies.Input('dropdown-color', 'value')])
def callback_color(dropdown_value):
    return "The selected color is %s." % dropdown_value

@app.callback(
    dash.dependencies.Output('output-size', 'children'),
    [dash.dependencies.Input('dropdown-color', 'value'),
     dash.dependencies.Input('dropdown-size', 'value')])
def callback_size(dropdown_color, dropdown_size):
    return "The chosen T-shirt is a %s %s one." %(dropdown_size,
                                                  dropdown_color)
```

Note that the `DjangoDash` constructor requires a name to be specified. This name is then used to identify the dash app in *templates*:

```
{%load plotly_dash%}

{%plotly_app name="SimpleExample"%}
```

Direct use in this manner, without any application state or use of live updating, is equivalent to inserting an `iframe` containing the URL of a `Dash` application.

Note: The registration code needs to be in a location that will be imported into the Django process before any model or template tag attempts to use it. The example Django application in the demo subdirectory achieves this through an import in the main `urls.py` file, but any `views.py` would also be sufficient.

1.4 Django models and application state

The `django_plotly_dash` application defines `DashApp` and `StatelessApp` models.

1.4.1 The `StatelessApp` model

An instance of the `StatelessApp` model represents a single dash application. Every instantiation of a `DjangoDash` object is registered, and any object that is referenced through the `DashApp` model - this includes all template access as well as model instances themselves - causes a `StatelessApp` model instance to be created if one does not already exist.

```
class StatelessApp(models.Model):
    '''
    A stateless Dash app.

    An instance of this model represents a dash app without any specific state
    '''

    app_name = models.CharField(max_length=100, blank=False, null=False, unique=True)
    slug = models.SlugField(max_length=110, unique=True, blank=True)

    def as_dash_app(self):
        '''
        Return a DjangoDash instance of the dash application
        '''
```

The main role of a `StatelessApp` instance is to manage access to the associated `DjangoDash` object, as exposed through the `as_dash_app` member function.

In the Django admin, an action is provided to check all of the known stateless instances. Those that cannot be instantiated are logged; this is a useful quick check to see what apps are avalilable. Also, in the same admin an additional button is provided to create `StatelessApp` instances for any known instance that does not have an ORM entry.

1.4.2 The `DashApp` model

An instance of the `DashApp` model represents an instance of application state.

```
class DashApp(models.Model):
    '''
    An instance of this model represents a Dash application and its internal state
    '''
    stateless_app = models.ForeignKey(StatelessApp, on_delete=models.PROTECT,
                                      unique=False, null=False, blank=False)
    instance_name = models.CharField(max_length=100, unique=True, blank=True,␣
↪null=False)
    slug = models.SlugField(max_length=110, unique=True, blank=True)
    base_state = models.TextField(null=False, default="{}")
    creation = models.DateTimeField(auto_now_add=True)
    update = models.DateTimeField(auto_now=True)
    save_on_change = models.BooleanField(null=False,default=False)

    ... methods, mainly for managing the Dash application state ...

    def current_state(self):
        '''
        Return the current internal state of the model instance
        '''

    def update_current_state(self, wid, key, value):
        '''
        Update the current internal state, ignorning non-tracked objects
        '''

    def populate_values(self):
        '''
        Add values from the underlying dash layout configuration
        '''
```

The `stateless_app` references an instance of the `StatelessApp` model described above. The `slug` field provides a unique identifier that is used in URLs to identify the instance of an application, and also its associated server-side state.

The persisted state of the instance is contained, serialised as JSON, in the `base_state` variable. This is an arbitrary subset of the internal state of the object. Whenever a `Dash` application requests its state (through the `<app slug>_dash-layout` url), any values from the underlying application that are present in `base_state` are overwritten with the persisted values.

The `populate_values` member function can be used to insert all possible initial values into `base_state`. This functionality is also exposed in the Django admin for these model instances, as a `Populate app` action.

From callback code, the `update_current_state` method can be called to change the initial value of any variable tracked within the `base_state`. Variables not tracked will be ignored. This function is automatically called for any callback argument and return value.

Finally, after any callback has finished, and after any result stored through `update_current_state`, then the application model instance will be persisted by means of a call to its `save` method, if any changes have been detected and the `save_on_change` flag is `True`.

1.5 Extended callback syntax

The `DjangoDash` class allows callbacks to request extra arguments when registered.

To do this, simply replace `callback` with `expanded_callback` when registering any callback. This will cause **all** of the callbacks registered with this application to receive extra `kwargs` in addition to the callback parameters.

For example, the `plotly_apps.py` example contains this dash application:

```
import dash
import dash_core_components as dcc
import dash_html_components as html

from django_plotly_dash import DjangoDash

a2 = DjangoDash("Ex2")

a2.layout = html.Div([
    dcc.RadioItems(id="dropdown-one",options=[{'label':i,'value':j} for i,j in [
    ("O2","Oxygen"),("N2","Nitrogen"),("CO2","Carbon Dioxide")]
    ],value="Oxygen"),
    html.Div(id="output-one")
    ])

@a2.expanded_callback(
    dash.dependencies.Output('output-one','children'),
    [dash.dependencies.Input('dropdown-one','value')]
    )

def callback_c(*args,**kwargs):
    da = kwargs['dash_app']
    return "Args are [%s] and kwargs are %s" %(",".join(args),str(kwargs))
```

The additional arguments, which are reported as the `kwargs` content in this example, include

dash_app For stateful applications, the `DashApp` model instance

dash_app_id The application identifier. For stateless applications, this is the (slugified) name given to the `DjangoDash` constructor. For stateful applications, it is the (slugified) unique identifier for the associated model instance.

session_state A dictionary of information, unique to this user session. Any changes made to its content during the callback are persisted as part of the Django session framework.

user The Django User instance.

The `DashApp` model instance can also be configured to persist itself on any change. This is discussed in the *Django models and application state* section.

1.5.1 Using session state

The *walkthrough* of the session state example details how the XXX demo interacts with a `Django` session.

Unless an explicit pipe is created, changes to the session state and other server-side objects are not automatically propagated to an application. Something in the front-end UI has to invoke a callback; at this point the latest version of these objects will be provided to the callback. The same considerations as in other Dash live updates apply.

The *live updating* section discusses how `django-plotly-dash` provides an explicit pipe that directly enables the updating of applications.

1.6 Live updating

Live updating is supported using additional `Dash` *components* and leveraging Django Channels to provide websocket endpoints.

Server-initiated messages are sent to all interested clients. The content of the message is then injected into the application from the client, and from that point it is handled like any other value passed to a callback function. The messages are constrained to be JSON serialisable, as that is how they are transmitted to and from the clients, and should also be as small as possible given that they travel from the server, to each interested client, and then back to the server again as an argument to one or more callback functions.

The round-trip of the message is a deliberate design choice, in order to enable the value within the message to be treated as much as possible like any other piece of data within a `Dash` application. This data is essentially stored on the client side of the client-server split, and passed to the server when each callback is invoked; note that this also encourages designs that keep the size of in-application data small. An alternative approach, such as directly invoking a callback in the server, would require the server to maintain its own copy of the application state.

Live updating requires a server setup that is considerably more complex than the alternative, namely use of the built-in Interval component. However, live updating can be used to reduce server load (as callbacks are only made when needed) and application latency (as callbacks are invoked as needed, not on the tempo of the Interval component).

1.6.1 Message channels

Messages are passed through named channels, and each message consists of a `label` and `value` pair. A *Pipe* component is provided that listens for messages and makes them available to `Dash` callbacks. Each message is sent through a message channel to all `Pipe` components that have registered their interest in that channel, and in turn the components will select messages by `label`.

A message channel exists as soon as a component signals that it is listening for messages on it. The message delivery requirement is 'hopefully at least once'. In other words, applications should be robust against both the failure of a message to be delivered, and also for a message to be delivered multiple times. A design approach that has messages of the form 'you should look at X and see if something should be done' is strongly encouraged. The accompanying demo has messages of the form 'button X at time T', for example.

1.6.2 Sending messages from within Django

Messages can be easily sent from within Django, provided that they are within the ASGI server.

```
from django_plotly_dash.consumers import send_to_pipe_channel

# Send a message
#
# This function may return *before* the message has been sent
# to the pipe channel.
#
send_to_pipe_channel(channel_name="live_button_counter",
                     label="named_counts",
                     value=value)

# Send a message asynchronously
#
await async_send_to_pipe_channel(channel_name="live_button_counter",
                                 label="named_counts",
                                 value=value)
```

In general, making assumptions about the ordering of code between message sending and receiving is unsafe. The `send_to_pipe` function uses the Django Channels `async_to_sync` wrapper around a call to `async_send_to_pipe` and therefore may return before the asynchronous call is made (perhaps on a different thread). Furthermore, the transit of the message through the channels backend introduces another indeterminacy.

1.6.3 HTTP Endpoint

There is an HTTP endpoint, *configured* with the `http_route` option, that allows direct insertion of messages into a message channel. It is a direct equivalent of calling the `send_to_pipe_channel` function, and expects the `channel_name`, `label` and `value` arguments to be provided in a JSON-encoded dictionary.

```
curl -d '{"channel_name":"live_button_counter",
          "label":"named_counts",
          "value":{"click_colour":"cyan"}}'
          http://localhost:8000/dpd/views/poke/
```

This will cause the (JSON-encoded) `value` argument to be sent on the `channel_name` channel with the given `label`.

The provided endpoint skips any CSRF checks and does not perform any security checks such as authentication or authorisation, and should be regarded as a starting point for a more complete implementation if exposing this functionality is desired. On the other hand, if this endpoint is restricted so that it is only available from trusted sources such as the server itself, it does provide a mechanism for Django code running outside of the ASGI server, such as in a WSGI process or Celery worker, to push a message out to running applications.

The `http_poke_enabled` flag controls the availability of the endpoint. If false, then it is not registered at all and all requests will receive a 404 HTTP error code.

1.6.4 Deployment

The live updating feature needs both Redis, as it is the only supported backend at present for v2.0 and up of Channels, and Daphne or any other ASGI server for production use. It is also good practise to place the server(s) behind a reverse proxy such as Nginx; this can then also be configured to serve Django's static files.

A further consideration is the use of a WSGI server, such as Gunicorn, to serve the non-asynchronous subset of the http routes, albeit at the expense of having to separately manage ASGI and WSGI servers. This can be easily achieved through selective routing at the reverse proxy level, and is the driver behind the `ws_route` configuration option.

In passing, note that the demo also uses Redis as the caching backend for Django.

1.7 Template tags

Template tags are provided in the `plotly_dash` library:

```
{%load plotly_dash%}
```

1.7.1 The `plotly_app` template tag

Importing the `plotly_dash` library provides the `plotly_app` template tag:

```
{%load plotly_dash%}

{%plotly_app name="SimpleExample"%}
```

This tag inserts a `DjangoDash` app within a page as a responsive `iframe` element.

The tag arguments are:

name = None The name of the application, as passed to a `DjangoDash` constructor.

slug = None The slug of an existing `DashApp` instance.

da = None An existing `django_plotly_dash.models.DashApp` model instance.

ratio = 0.1 The ratio of height to width. The container will inherit its width as 100% of its parent, and then rely on this ratio to set its height.

use_frameborder = "0" HTML element property of the iframe containing the application.

initial_arguments = None Initial arguments overriding app defaults and saved state.

At least one of `da`, `slug` or `name` must be provided. An object identified by `slug` will always be used, otherwise any identified by `name` will be. If either of these arguments are provided, they must resolve to valid objects even if not used. If neither are provided, then the model instance in `da` will be used.

The `initial_arguments` are specified as a python dictionary. This can be the actual `dict` object, or a JSON-encoded string representation. Each entry in the dictionary has the `id` as key, and the corresponding value is a dictionary mapping property name keys to initial values.

1.7.2 The `plotly_app_bootstrap` template tag

This is a variant of the `plotly_app` template for use with responsive layouts using the Bootstrap library

```
{%load plotly_dash%}

{%plotly_app_bootstrap name="SimpleExample" aspect="16by9"%}
```

The tag arguments are similar to the `plotly_app` ones:

name = None The name of the application, as passed to a `DjangoDash` constructor.

slug = None The slug of an existing `DashApp` instance.

da = None An existing `django_plotly_dash.models.DashApp` model instance.

aspect= "4by3" The aspect ratio of the app. Should be one of 21by9, 16by9, 4by3 or 1by1.

initial_arguments = None Initial arguments overriding app defaults and saved state.

At least one of `da`, `slug` or `name` must be provided. An object identified by `slug` will always be used, otherwise any identified by `name` will be. If either of these arguments are provided, they must resolve to valid objects even if not used. If neither are provided, then the model instance in `da` will be used.

The aspect ratio has to be one of the available ones from the Bootstrap framework.

The `initial_arguments` are specified as a python dictionary. This can be the actual `dict` object, or a JSON-encoded string representation. Each entry in the dictionary has the `id` as key, and the corresponding value is a dictionary mapping property name keys to initial values.

1.7.3 The `plotly_direct` template tag

This template tag allows the direct insertion of html into a template, instead of embedding it in an iframe.

```
{%load plotly_dash%}

{%plotly_direct name="SimpleExample"%}
```

The tag arguments are:

name = None The name of the application, as passed to a `DjangoDash` constructor.

slug = None The slug of an existing `DashApp` instance.

da = None An existing `django_plotly_dash.models.DashApp` model instance.

These arguments are equivalent to the same ones for the `plotly_app` template tag. Note that `initial_arguments` are not currently supported, and as the app is directly injected into the page there are no arguments to control the size of the iframe.

This tag should not appear more than once on a page. This rule however is not enforced at present.

If this tag is used, then the *header and footer* tags should also be added to the template. Note that these tags in turn have middleware requirements.

1.7.4 The `plotly_header` and `plotly_footer` template tags

`DjangoDash` allows you to inject directly the html generated by `Dash` in the DOM of the page without wrapping it in an iframe. To include the app CSS and JS, two tags should be included in the template, namely `plotly_header` and `plotly_footer`, as follows:

```
<!-- templates/base.html -->
<!DOCTYPE html>
<html>
    <head>
    ...
    {% load plotly_dash%}
    ...
    {% plotly_header %}
    ...
    </head>
    <body>
    ...
    {%plotly_direct name="SimpleExample"%}
    ...
    </body>
    ...
    {% plotly_footer %}
</html>
```

This part in mandatory if you want to use the *plotly_direct* tag, and these two tags can safely be included on any page that has loaded the `plotly_dash` template tag library with minimal overhead, making them suitable for inclusion in a base template. Neither tag has any arguments.

Note that if you are using any functionality that needs the use of these tags, then the associated middleware should be added in `settings.py`

```
MIDDLEWARE = [
                ...

                'django_plotly_dash.middleware.BaseMiddleware',

            ]
```

This middleware should appear low down the middleware list.

1.7.5 The `plotly_message_pipe` template tag

This template tag has to be inserted on every page that uses live updating:

```
{%load plotly_dash%}

{%plotly_app ... DjangoDash instances using live updating ... %}

{%plotly_message_pipe%}
```

The tag inserts javascript needed for the *Pipe* component to operate. It can be inserted anywhere on the page, and its ordering relative to the `Dash` instances using updating is not important, so placing it in the page footer - to avoid delaying the main page load - along with other scripts is generally advisable.

1.7.6 The `plotly_app_identifier` template tag

This tag provides an identifier for an app, in a form that is suitable for use as a classname or identifier in HTML:

```
{%load plotly_dash%}

{%plotly_app_identifier name="SimpleExample"%}

{%plotly_app_identifier slug="liveoutput-2" postfix="A"%}
```

The identifier, if the tag is not passed a `slug`, is the result of passing the identifier of the app through the `django.utils.text.slugify` function.

The tag arguments are:

name = None The name of the application, as passed to a `DjangoDash` constructor.

slug = None The slug of an existing `DashApp` instance.

da = None An existing `django_plotly_dash.models.DashApp` model instance.

postfix = None An optional string; if specified it is appended to the identifier with a hyphen.

The validity rules for these arguments are the same as those for the `plotly_app` template tag. If supplied, the `postfix` argument should already be in a slug-friendly form, as no processing is performed on it.

1.7.7 The `plotly_class` template tag

Generate a string of class names, suitable for a `div` or other element that wraps around `django-plotly-dash` template content.

```
{%load plotly_dash%}

<div class="{%plotly_class slug="liveoutput-2" postfix="A"%}">
  {%plotly_app slug="liveoutput-2" ratio="0.5" %}
</div>
```

The identifier, if the tag is not passed a `slug`, is the result of passing the identifier of the app through the `django.utils.text.slugify` function.

The tag arguments are:

name = None The name of the application, as passed to a `DjangoDash` constructor.

slug = None The slug of an existing `DashApp` instance.

da = None An existing `django_plotly_dash.models.DashApp` model instance.

prefix = None Optional prefix to use in place of the text `django-plotly-dash` in each class name

postfix = None An optional string; if specified it is appended to the app-specific identifier with a hyphen.

template_type = None Optional text to use in place of `iframe` in the template-specific class name

The tag inserts a string with three class names in it. One is just the `prefix` argument, one has the `template_type` appended, and the final one has the app identifier (as generated by the `plotly_app_identifier` tag) and any `postfix` appended.

The validity rules for these arguments are the same as those for the `plotly_app` and `plotly_app_identifier` template tags. Note that none of the `prefix`, `postfix` and `template_type` arguments are modified and they should already be in a slug-friendly form, or otherwise fit for their intended purpose.

1.8 Dash components

The `dpd-components` package contains `Dash` components. This package is installed as a dependency of `django-plotly-dash`.

1.8.1 The `Pipe` component

Each `Pipe` component instance listens for messages on a single channel. The `value` member of any message on that channel whose `label` matches that of the component will be used to update the `value` property of the component. This property can then be used in callbacks like any other `Dash` component property.

An example, from the demo application:

```
import dpd_components as dpd

app.layout = html.Div([
    ...
    dpd.Pipe(id="named_count_pipe",                # ID in callback
             value=None,                           # Initial value prior to any message
             label="named_counts",                 # Label used to identify relevant␣
↪messages
             channel_name="live_button_counter"), # Channel whose messages are to be␣
↪examined
    ...
    ])
```

The `value` of the message is sent from the server to all front ends with `Pipe` components listening on the given `channel_name`. This means that this part of the message should be small, and it must be JSON serialisable. Also, there is no guarantee that any callbacks will be executed in the same Python process as the one that initiated the initial message from server to front end.

The `Pipe` properties can be persisted like any other `DashApp` instance, although it is unlikely that continued persistence of state on each update of this component is likely to be useful.

This component requires a bidirectional connection, such as a websocket, to the server. Inserting a `plotly_message_pipe` *template tag* is sufficient.

1.9 Configuration options

The `PLOTLY_DASH` settings variable is used for configuring `django-plotly-dash`. Default values are shown below.

```
PLOTLY_DASH = {

    # Route used for the message pipe websocket connection
    "ws_route" :   "dpd/ws/channel",

    # Route used for direct http insertion of pipe messages
    "http_route" : "dpd/views",

    # Flag controlling existince of http poke endpoint
    "http_poke_enabled" : True,

    # Insert data for the demo when migrating
    "insert_demo_migrations" : False,

    # Timeout for caching of initial arguments in seconds
    "cache_timeout_initial_arguments": 60,

    # Name of view wrapping function
    "view_decorator": None,

    # Flag to control location of initial argument storage
    "cache_arguments": True,

    # Flag controlling local serving of assets
    "serve_locally': False,
}
```

Defaults are inserted for missing values. It is also permissible to not have any `PLOTLY_DASH` entry in the Django settings file.

The Django staticfiles infrastructure is used to serve all local static files for the Dash apps. This requires adding a setting for the specification of additional static file finders

```
# Staticfiles finders for locating dash app assets and related files

STATICFILES_FINDERS = [

    'django.contrib.staticfiles.finders.FileSystemFinder',
    'django.contrib.staticfiles.finders.AppDirectoriesFinder',

    'django_plotly_dash.finders.DashAssetFinder',
    'django_plotly_dash.finders.DashComponentFinder',
    'django_plotly_dash.finders.DashAppDirectoryFinder',
]
```

and also providing a list of components used

```
# Plotly components containing static content that should
# be handled by the Django staticfiles infrastructure

PLOTLY_COMPONENTS = [

    # Common components
    'dash_core_components',
    'dash_html_components',
    'dash_renderer',
```

(continues on next page)

(continued from previous page)

```
    # django-plotly-dash components
    'dpd_components',
    # static support if serving local assets
    'dpd_static_support',

    # Other components, as needed
    'dash_bootstrap_components',
]
```

This list should be extended with any additional components that the applications use, where the components have files that have to be served locally.

Furthermore, middleware should be added for redirection of external assets from underlying packages, such as `dash-bootstrap-components`. With the standard Django middleware, along with `whitenoise`, the entry within the `settings.py` file will look something like

```
# Standard Django middleware with the addition of both
# whitenoise and django_plotly_dash items

MIDDLEWARE = [

      'django.middleware.security.SecurityMiddleware',

      'whitenoise.middleware.WhiteNoiseMiddleware',

      'django.contrib.sessions.middleware.SessionMiddleware',
      'django.middleware.common.CommonMiddleware',
      'django.middleware.csrf.CsrfViewMiddleware',
      'django.contrib.auth.middleware.AuthenticationMiddleware',
      'django.contrib.messages.middleware.MessageMiddleware',

      'django_plotly_dash.middleware.BaseMiddleware',
      'django_plotly_dash.middleware.ExternalRedirectionMiddleware',

      'django.middleware.clickjacking.XFrameOptionsMiddleware',
  ]
```

Individual apps can set their `serve_locally` flag. However, it is recommended to use the equivalent global `PLOTLY_DASH` setting to provide a common approach for all static assets. See *Local assets* for more information on how local assets are configured and served as part of the standard Django staticfiles approach, along with details on the integration of other components and some known issues.

1.9.1 Endpoints

The websocket and direct http message endpoints are separately configurable. The configuration options exist to satisfy two requirements

- Isolate paths that require serving with ASGI. This allows the asynchronous routes - essentially the websocket connections and any other ones from the rest of the application - to be served using `daphne` or similar, and the bulk of the (synchronous) routes to be served using a WSGI server such as `gunicorn`.
- Isolate direct http posting of messages to restrict their use. The motivation behind this http endpoint is to provide a private service that allows other parts of the overall application to send notifications to `Dash` applications, rather than expose this functionality as part of the public API.

A reverse proxy front end, such as `nginx`, can route appropriately according to URL.

1.9.2 View decoration

Each view delegated through to `plotly_dash` can be wrapped using a view decoration function. This enables access to be restricted to logged-in users, or using a desired conditions based on the user and session state.

To restrict all access to logged-in users, use the `login_required` wrapper:

```
PLOTLY_DASH = {

    ...
    # Name of view wrapping function
    "view_decorator": "django_plotly_dash.access.login_required",
    ...
}
```

More information can be found in the *view decoration* section.

1.9.3 Initial arguments

Initial arguments are stored within the server between the specification of an app in a template tag and the invocation of the view functions for the app. This storage is transient and can be efficiently performed using Django's caching framework. In some situations, however, a suitably configured cache is not available. For this use case, setting the `cache_arguments` flag to `False` will cause initial arguments to be placed inside the Django session.

1.10 Local assets

Local ploty dash assets are integrated into the standard Django staticfiles structure. This requires additional settings for both staticfiles finders and middleware, and also providing a list of the components used. The specific steps are listed in the *Configuration options* section.

Individual applications can set a `serve_locally` flag but the use of the global setting in the `PLOTLY_DASH` variable is recommended.

1.10.1 Additional components

Some components, such as `dash-bootstrap-components`, require external packages such as Bootstrap to be supplied. In turn this can be achieved using for example the `bootstrap4` Django application. As a consequence, dependencies on external URLs are introduced.

This can be avoided by use of the `dpd-static-support` package, which supplies mappings to locally served versions of these assets. Installation is through the standard `pip` approach

```
pip install dpd-static-support
```

and then the package should be added as both an installed app and to the `PLOTLY_COMPONENTS` list in `settings.py`, along with the associated middleware

```
INSTALLED_APPS = [
    ...
    'dpd_static_support',
]

MIDDLEWARE = [
```

(continues on next page)

(continued from previous page)

```
    ...
    'django_plotly_dash.middleware.ExternalRedirectionMiddleware',
]

PLOTLY_COMPONENTS = [
    ...
    'dpd_static_support'
]
```

Note that the middleware can be safely added even if the `serve_locally` functionality is not in use.

1.10.2 Known issues

Absolute paths to assets will not work correctly. For example:

```
app.layout = html.Div([html.Img(src=localState.get_asset_url('image_one.png')),
                       html.Img(src='assets/image_two.png'),
                       html.Img(src='/assets/image_three.png'),
                       ])
```

Of these three images, both `image_one.png` and `image_two.png` will be served up - through the static files infrastructure - from the `assets` subdirectory relative to the code defining the `app` object. However, when rendered the application will attempt to load `image_three.png` using an absolute path. This is unlikely to be the desired result, but does permit the use of absolute URLs within the server.

1.11 Demonstration application

There are a number of pages in the demo application in the source repository.

1. Direct insertion of one or more dash applications
2. Initial state storage within Django
3. Enhanced callbacks
4. Live updating
5. Injection without using an iframe
6. Simple html injection
7. Bootstrap components
8. Session state storage
9. Local serving of assets
10. Multiple callback values

The templates that drive each of these can be found in the github repository.

There is a more details walkthrough of the *session state storage* example. This example also shows the use of dash bootstrap components.

The demo application can also be viewed online.

1.11.1 Session state example walkthrough

The session state example has three separate components in the demo application

- A template to render the application
- The `django-plotly-dash` application itself
- A view to render the template having initialised the session state if needed

The first of these is a standard Django template, containing instructions to render the Dash application:

```
{%load plotly-dash%}

...

<div class="{%plotly_class name="DjangoSessionState"%}">
  {%plotly_app name="DjangoSessionState" ratio=0.3 %}
</div>
```

The view sets up the initial state of the application prior to rendering. For this example we have a simple variant of rendering a template view:

```
def session_state_view(request, template_name, **kwargs):

    # Set up a context dict here
    context = { ... values for template go here, see below ... }

    return render(request, template_name=template_name, context=context)
```

and it suffices to register this view at a convenient URL as it does not use any parameters:

```
...
url('^demo-eight',
    session_state_view,
    {'template_name':'demo_eight.html'},
    name="demo-eight"),
...
```

In passing, we note that accepting parameters as part of the URL and passing them as initial parameters to the app through the template is a straightforward extension of this example.

The session state can be accessed in the app as well as the view. The app is essentially formed from a layout function and a number of callbacks. In this particular example, dash-bootstrap-components are used to form the layout:

```
dis = DjangoDash("DjangoSessionState",
                 add_bootstrap_links=True)

dis.layout = html.Div(
    [
        dbc.Alert("This is an alert", id="base-alert", color="primary"),
        dbc.Alert(children="Danger", id="danger-alert", color="danger"),
        dbc.Button("Update session state", id="update-button", color="warning"),
    ]
)
```

Within the *expanded callback*, the session state is passed as an extra argument compared to the standard `Dash` callback:

```
@dis.expanded_callback(
    dash.dependencies.Output("danger-alert", 'children'),
    [dash.dependencies.Input('update-button', 'n_clicks'),]
    )
def session_demo_danger_callback(n_clicks, session_state=None, **kwargs):
    if session_state is None:
        raise NotImplementedError("Cannot handle a missing session state")
    csf = session_state.get('bootstrap_demo_state', None)
    if not csf:
        csf = dict(clicks=0)
        session_state['bootstrap_demo_state'] = csf
    else:
        csf['clicks'] = n_clicks
    return "Button has been clicked %s times since the page was rendered" %n_clicks
```

The session state is also set during the view:

```
def session_state_view(request, template_name, **kwargs):

    session = request.session

    demo_count = session.get('django_plotly_dash', {})

    ind_use = demo_count.get('ind_use', 0)
    ind_use += 1
    demo_count['ind_use'] = ind_use
    session['django_plotly_dash'] = demo_count

    # Use some of the information during template rendering
    context = {'ind_use' : ind_use}

    return render(request, template_name=template_name, context=context)
```

Reloading the demonstration page will cause the page render count to be incremented, and the button click count to be reset. Loading the page in a different session, for example by using a different browser or machine, will have an independent render count.

1.12 View decoration

The `django-plotly-dash` views, as served by Django, can be wrapped with an arbitrary decoration function. This allows the use of the Django login_required view decorator as well as enabling more specialised and fine-grained control.

1.12.1 The `login_required` decorator

The `login_required` decorator from the Django authentication system can be used as a view decorator. A wrapper function is provided in `django_plotly_dash.access`.

```
PLOTLY_DASH = {

    ...
    # Name of view wrapping function
    "view_decorator": "django_plotly_dash.access.login_required",
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

Note that the view wrapping is on all of the `django-plotly-dash` views.

1.12.2 Fine-grained control

The view decoration function is called for each variant exposed in the `django_plotly_dash.urls` file. As well as the underlying view function, each call to the decorator is given the name of the route, as used by `django.urls.reverse`, the specific url fragment for the view, and a name describing the type of view.

From this information, it is possible to implement view-specific wrapping of the view functions, and in turn the wrapper functions can then use the request content, along with other information, to control access to the underlying view function.

```
from django.views.decorators.csrf import csrf_exempt

def check_access_permitted(request, **kwargs):
    # See if access is allowed; if so return True
    # This function is called on each request

    ...

    return True

def user_app_control(view_function, name=None, **kwargs):
    # This function is called on the registration of each django-plotly-dash view
    # name is one of main component-suites routes layout dependencies update-component

    def wrapped_view(request, *args, **kwargs):
        is_permitted = check_access_permitted(request, **kwargs)
        if not is_permitted:
            # Access not permitted, so raise error or generate an appropriate response

            ...

        else:
            return view_function(request, *args, **kwargs)

    if getattr(view_function,"csrf_exempt",False):
        return csrf_exempt(wrapped_view)

    return wrapped_view
```

The above sketch highlights how access can be controlled based on each request. Note that the `csrf_exempt` property of any wrapped view is preserved by the decoration function and this approach needs to be extended to other properties if needed. Also, this sketch only passes `kwargs` to the permission function.

1.13 FAQ

- What environment versions are supported?

At least v3.5 of Python, and v2.0 of Django, are needed.

- Is a `virtualenv` mandatory?

No, but it is strongly recommended for any Python work.

- What about Windows?

The python package should work anywhere that Python does. Related applications, such as Redis, have their own requirements but are accessed using standard network protocols.

- How do I report a bug or other issue?

Create a github issue. See *bug reporting* for details on what makes a good bug report.

- Where should `Dash` layout and callback functions be placed?

In general, the only constraint on the files containing these functions is that they should be imported into the file containing the `DjangoDash` instantiation. This is discussed in the *Installation* section and also in this github issue.

- Can per-user or other fine-grained access control be used?

  Yes. See the *View decoration* configuration setting and *View decoration* section.

- What settings are needed to run the server in debug mode?

The `prepare_demo` script in the root of the git repository contains the full set of commands for running the server in debug mode. In particular, the debug server is launched with the `--nostatic` option. This will cause the staticfiles to be served from the collected files in the `STATIC_ROOT` location rather than the normal `runserver` behaviour of serving directly from the various locations in the `STATICFILES_DIRS` list.

- Is use of the `get_asset_url` function optional for including static assets?

No, it is needed. Consider this example (it is part of `demo-nine`):

```
localState = DjangoDash("LocalState",
                        serve_locally=True)

localState.layout = html.Div([html.Img(src=localState.get_asset_url('image_one.png')),
                              html.Img(src='/assets/image_two.png'),
                              ])
```

The first `Img` will have its source file correctly served up by Django as a standard static file. However, the second image will not be rendered as the path will be incorrect.

See the *Local assets* section for more information on configuration with local assets.

- Is there a live demo available?

Yes. It can be found here

1.14 Development

The application and demo are developed, built and tested in a virtualenv enviroment, supported by a number of `bash` shell scripts. The resultant package should work on any Python installation that meets the requirements.

Automatic builds have been set up on Travis-CI including running tests and reporting code coverage.

Current status:

1.14.1 Environment setup

To set up a development environment, first clone the repository, and then use the `make_env` script:

```
git clone https://github.com/GibbsConsulting/django-plotly-dash.git

cd django-plotly-dash

./make_env
```

The script creates a virtual environment and uses `pip` to install the package requirements from the `requirements.txt` file, and then also the extra packages for development listed in the `dev_requirements.txt` file. It also installs `django-plotly-dash` as a development package.

Redis is an optional dependency, and is used for live updates of Dash applications through channels endpoints. The `prepare_redis` script can be used to install Redis using Docker. It essentially pulls the container and launches it:

```
# prepare_redis content:
docker pull redis:4
docker run -p 6379:6379 -d redis
```

The use of Docker is not mandatory, and any method to install Redis can be used provided that the *configuration* of the host and port for channels is set correcty in the `settings.py` for the Django project.

During development, it can be convenient to serve the `Dash` components locally. Whilst passing `serve_locally=True` to a `DjangoDash` constructor will cause all of the css and javascript files for the components in that application from the local server, it is recommended to use the global `serve_locally` configuration setting.

Note that it is not good practice to serve static content in production through Django.

1.14.2 Coding and testing

The `pylint` and `pytest` packages are important tools in the development process. The global configuration used for `pylint` is in the `pylintrc` file in the root directory of the codebase.

Tests of the package are contained within the `django_plotly_dash/tests.py` file, and are invoked using the Django settings for the demo. Running the tests from the perspective of the demo also enables code coverage for both the application and the demo to be measured together, simplifying the bookkeeping.

Two helper scripts are provided for running the linter and test code:

```
# Run pylint on django-plotly-dash module
./check_code_dpd

# Run pylint on the demo code, and then execute the test suite
./check_code_demo
```

It is also possible to run all of these actions together:

```
# Run all of the checks
./check_code
```

The goal is for complete code coverage within the test suite and for maximal ('ten out of ten') marks from the linter. Perfection is however very hard and expensive to achieve, so the working requirement is for every release to keep the linter score above 9.5, and ideally improve it, and for the level of code coverage of the tests to increase.

1.14.3 Documentation

Documentation lives in the `docs` subdirectory as reStructuredText and is built using the `sphinx` toolchain.

Automatic local building of the documentation is possible with the development environment:

```
source env/bin/activate
cd docs && sphinx-autobuild . _build/html
```

In addition, the `grip` tool can be used to serve a rendered version of the `README` file:

```
source env/bin/activate
grip
```

The online documentation is automatically built by the `readthedocs` infrastructure when a release is formed in the main `github` repository.

1.14.4 Release builds

This section contains the recipe for building a release of the project.

First, update the version number appropriately in `django_plotly_dash/version.py`, and then ensure that the checks and tests have been run:

```
./check_code
```

Next, construct the `pip` packages and push them to pypi:

```
source env/bin/activate

python setup.py sdist
python setup.py bdist_wheel

twine upload dist/*
```

Committing a new release to the main github repository will invoke a build of the online documentation, but first a snapshot of the development environment used for the build should be generated:

```
pip freeze > frozen_dev.txt

git add frozen_dev.txt
git add django_plotly_dash/version.py

git commit -m" ... suitable commit message for this release ..."

# Create PR, merge into main repo, check content on PYPI and RTD
```

This preserves the state used for building and testing for future reference.

1.14.5 Bug reports and other issues

The ideal bug report is a pull request containing the addition of a failing test exhibiting the problem to the test suite. However, this rarely happens in practice!

The essential requirement of a bug report is that it contains enough information to characterise the issue, and ideally also provides some way of replicating it. Issues that cannot be replicated within a virtualenv are unlikely to get much attention, if any.

To report a bug, create a github issue.

1.15 License

The `django-plotly-dash` package is made available under the MIT license.

The license text can be found in the LICENSE file in the root directory of the source code, along with a CONTRIBUTIONS.md file that includes a list of the contributors to the codebase.

A copy of the license, correct at the time of writing of this documentation, follows: