GDAPS

Christian González

Jul 11, 2023

TABLE OF CONTENTS

Welcome to the GDAPS documentation!

GDAPS is a plugin system that can be added to Django, **to create applications that can be extended via plugins later**.

Note: As of v0.7.0, Vue.js support was dropped, in favour of the *new template rendering plugin system*.

CHAPTER

ONE

INTRODUCTION

This library allows Django to make real "pluggable" apps.

A standard Django "app" is *reusable* (if done correctly), but is not *pluggable*, like being distributed and "plugged" into a Django main application without modifications. GDAPS is filling this gap.

The reason you want to use GDAPS is: **you want to create an application that should be extended via plugins**. GDAPS consists of a few bells and twistles where Django lacks "automagic":

GDAPS apps...

- are automatically found using setuptools' entry points
- can provide their own URLs which are included and merged into urlpatterns automatically
- can define `Interfaces`, that other GDAPS apps then can implement
- can provide Javascript frontends that are found and compiled automatically (WorkInProgress)

1.1 GDAPS working modes

The "observer pattern" plugin system is completely decoupled from the PluginManager (which manages GDAPS pluggable Django apps), so basically you have two choices to use GDAPS:

Simple mode Use *Interfaces*, and *Implementations* **without a complete plugin/module system**. It's not necessary to divide your application into GDAPS apps or create separate packages to use GDAPS. Just code your application as a monolithic django application and have an easy-to-use "observer pattern" plugin system.

- import gdaps
- Define an `gdaps.api.Interface` anywhere
- Create one or more implementations of it and
- iterate over the interface to get all the implementations, no matter where they are in the code.

Just *importing* the python files with your implementations will make them work automatically.

Use this if you just want to structure your Django software using an "observer pattern". This is used within GDAPS itself.

Full mode Use GDAPS as a **full-featured framework to create modular applications**.

- Add "gdaps" to your INSTALLED_APPS.
- Create plugins using `cookiecutter gl:nerdocs/gdaps-plugin-cookiecutter` (or `./manage.py startplugin`, which basically a wrapper), and install them via pip, (or add them to your INSTALLED_APPS).

You have a `gdaps.PluginManager` available then, and after a `manage.py migrate` and `manage.py syncplugins`, Django will have all GDAPS plugins recognized as models too, so you can easily administer them in your Django admin.

This "full" usage enables you to create fully-fledged extensible applications enabling third party plugins that can be distributed via PyPi.

See *Usage* for further instructions.

CHAPTER

TWO

INSTALLATION

Install GDAPS in your Python virtual environment. If you want to create plugins, install cookiecutter too.

```
pip install gdaps cookiecutter
```

Create a Django application as usual: `django-admin startproject myproject`.

First, set a variable named `PROJECT_NAME`.

```
PROJECT_NAME = "myproject"
```

This is a (machine) name for your project. Django itself does not provide such a name. It will be used in various places. Must be a valid python identifier.

Note: `PROJECT_NAME` is roughly what Django means with `ROOT_URLCONF[0]`, but GDAPS requires it to be set explicitly.

Now add "gdaps" to the `INSTALLED_APPS` section, and add a special line below it:

```
from gdaps.pluginmanager import PluginManager

INSTALLED_APPS = [
    # ... standard Django apps and GDAPS
    "gdaps",
]
# The following line is important: It loads all plugins from setuptools
# entry points and from the directory named 'myproject.plugins':
INSTALLED_APPS += PluginManager.find_plugins(PROJECT_NAME + ".plugins")
```

You can use whatever you want for your plugin path, but we recommend that you use "**<PROJECT_NAME>.plugins**" here to make things easier. Basically, this is all you really need so far, for a minimal working GDAPS-enabled Django application. See *Usage* for how to use GDAPS.

2.1 URL handling

Now add the URL path for GDAPS, so it can add plugins' URLs automatically to the global urlpattern.

```
# urls.py
from gdaps.pluginmanager import PluginManager

urlpatterns = PluginManager.urlpatterns() + [
```

(continues on next page)

(continued from previous page)

```
    # ... add your fixed URL patterns here, like "admin/", etc.
]
```

This way each plugin can have an *urlpatterns* variable in *urls.py*, and all are merged together. However, by now, the plugin order is not determined, so urlpatterns too are not in a deterministically determined order. This could lead to problems, depending on your application design, so keep that in mind when designing plugins.

2.2 Logging

Django does not write loggings to the command line automatically. GDAPS uses various levels of logging. It is recommended that you create a LOGGING section in settings.py for GDAPS:

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {"console": {"class": "logging.StreamHandler"}},
    "loggers": {
        "gdaps": {"handlers": ["console"], "level": "INFO", "propagate": True},
    },
}
```

This will output all GDAPS log messages to the console.

CHAPTER

THREE

USAGE

3.1 Creating plugins

If you use git in your project, install the `gitpython` module (`pip install gitpython`). `startplugin` will determine your git user/email automatically and use it.

Create a plugin using a Django management command:

```
./manage.py startplugin fooplugin
```

This command asks a few questions, creates a basic Django app in the plugin path chosen in `PluginManager.find_plugins()`. It provides useful defaults as well as a setup.py/setup.cfg file.

You now have two choices for this plugin:

- add it statically to `INSTALLED_APPS`: see *Static plugins*.
- make use of the dynamic loading feature: see *Dynamic plugins*.

3.1.1 Static plugins

In most of the cases, you will ship your application with a few "standard" plugins that are statically installed. These plugins must be loaded *after* the `gdaps` app.

```
# ...

INSTALLED_APPS = [
    # ... standard Django apps and GDAPS
    "gdaps",

    # put "static" plugins here too:
    "myproject.plugins.fooplugin",
]
```

This plugin app is loaded as usual, but your GDAPS enhanced Django application can make use of it's GDAPS features.

3.1.2 Dynamic plugins

By installing a plugin with pip, you can make your application aware of that plugin too:

```
pip install -e myproject/plugins/fooplugin
```

This installs the plugin as python module into the site-packages and makes it discoverable using setuptools. From this moment on it should be already registered and loaded after a Django server restart.

Of course this also works when plugins are installed from PyPi or from other directories, they don't have to be in the project's `plugins` folder. You can conveniently start developing plugins in there, and later move them into their own repository.

3.1.3 The plugin AppConfig

Plugins' AppConfigs must provide an inner class named `PluginMeta`, or a so named attribute pointing to an external class. For more information see `gdaps.apps.PluginMeta`.

3.1.4 Interfaces

Plugins can define interfaces, which can then be implemented by other plugins. The cookiecutter template contains an `<app_name>/api/interfaces.py` file automatically. It's not obligatory to put all Interface definitions in `api.interfaces`, but it is a recommended coding style for GDAPS plugins:

```
from gdaps import Interface

@Interface
class ITextRenderer:
    """Documentation of the interface"""

    __service__ = True  # is the default
    text_type = None

    def render(self):
        pass
```

Predefined attributes are:

__service__ If `__service__ = True` is set (which is the default), implementations are **instantiated when registered**. Iterating over the interface directly returns **instances** of the plugin.

```
for plugin in ITextRenderer:
    compiled_text = plugin.render()
```

If you use `__service__ = False`, the plugins are not instantiated at registration, and iterations over instances will return **classes**, not instances. This may be desired for reducing memory footprint, for data classes, or plugin classes that just contain static or class methods.

```
for plugin in INonServiceInterface:
    print(plugin.name)  # class attribute
    plugin.some_classmethod()

    # if you need instances, you have to instantiate the plugin here.
    # this is not recommended.
    p = plugin()
    p.do_something()
```

Interfaces can not be inherited to create other interfaces. If you inherit from an interface, you create an *Implementation*.

If you want to create some similar interfaces, use Mixins:

```
class IQuackWalkMixin:
    def do_something(self):
        pass

    def walk(self):
        pass

@Interface
class IDuck(IQuackWalkMixin):
    name = "Duck"

@Interface
class IGoose(IQuackWalkMixin)
    name = "Goose"
```

This way you can create interfaces that inherit from one or more mixins.

3.1.5 Implementations

You can then easily implement this interface in any other file (in this plugin or in another plugin) by subclassing the interface. Let's imagine a simple interface for letting plugins modify persons after creating them in a view:

```
@Interface
class IModifyPersonAfterCreate
    """Modify persons after creating them in a view"""
    def modify(self, person: Person):
        """modify the person"""
```

You can straight-forwardly use implementations that are bound to an interface by iterating over that interface, anywhere in your code - here in the CreateView of the main app:

```
from django.views.generic import CreateView
from myproject.plugins.fooplugin.api.interfaces import IModifyPersonAfterCreate

class CreatePersonView(CreateView):
    ...

    def form_valid(self, form):
        for plugin in IModifyPersonAfterCreate:
            plugin.modify(form.instance)
```

After defining an interface, any plugin found by GDAPS can implement this interface, let's say we want to capitalize the first name of the person:

```
from myproject.plugins.fooplugin.api.interfaces import IModifyPersonAfterCreate

class PersonFirstnameCapitalizer(IModifyPersonAfterCreate):
    weight = 10

    def modify(self, person):
        person.first_name = person.first_name.capitalize()
```

Depending on the *__service__* Meta flag, iterating over an Interface returns either a **class** (`__service__ = False`) or an **instance** (`__service__ = True`), which is the default.

3.1.6 Template support

Plugins usually provide not only interfaces for the backend, but also for the frontend. GDAPS supports plugin rendering in Django templates, which have to follow a certain pattern. Define your interface in the providing app, e.g. as usually in `.api.interfaces`, and let it inherit *gdaps.api.interfaces.ITemplatePluginMixin*. Don't forget to document your interface, so that the implementor knows what to expect.

```
# main_app/api/interfaces.py

from gdaps.api import Interface
from gdaps.api.interfaces import ITemplatePluginMixin

@Interface
class AnyItem(ITemplatePluginMixin):
    """Any list item, must contain a <li> element as root."""
```

This defines the plugin hook your plugins can implement. You have to follow a certain pattern here, or let your interface inherit from `ITemplatePluginMixin`, which helps your IDE with auto-suggestions. The mixin defines a few methods and attributes you can make use of:

ITemplatePluginMixin

template A string that is rendered as Template. For simple & small templates, e.g. one-liners. If this attribute is present, it is used.

template_name The usual django-like template name, where to find the template file within the `templates` directory, like "my_app/any_item.html" This attribute is used, if no `template` attribute is provided.

context a dict that provides the context for template rendering. It updates the global context.

If you want to customize it further, see *gdaps.api.interfaces.ITemplatePluginMixin*

Now, in your other plugins, create the implementation:

```
# in plugin A

from main_app.api.interfaces import AnyItem

class SayFooItem(AnyItem):
    template = "<li>Foo!</li>"


# in plugin B

from main_app.api.interfaces import AnyItem

class SayBarItem(AnyItem):
    template = "<li>Bar!</li>"
```

render_plugin hook

Now in your main app's template, render the plugins using the `render_plugins` tag, with the interface name as parameter:

```
{% load gdaps %}

<h1>Plugin sandbox</h1>
<ul>
    {% render_plugins IAnyItem %}
</ul>
```

That's all. GDAPS finds any plugins implementing this interface and renders them, one after another, in place. In this example, the resulting HTML code would be:

```
<li>Foo</li><li>Bar!</li>
```

As said before, the plugin templates can contain anything you like, not only `<li>` elements. You can use it for select options, cards on a dashboard, or whole page contents - it's up to you.

3.1.7 Extending Django's URL patterns

App URLs

App URLs are automatically detected by GDAPS/Django and put into your app's namespace. First, you have to add a code fragment to your global urls.py file:

```
from gdaps.pluginmanager import PluginManager
urlpatterns = PluginManager.urlpatterns() + [
    # add your usual, fixed, non-plugin paths here.
]
```

GDAPS then loads and imports all available plugins' *urls.py* files, collects their `urlpatterns` variables and merges them into your application's global urlpattern, using your plugin's `app_name` as namespace:

```
from .views import MyUrlView, SomeViewSet
from django.views.generic import TemplateView
# fooplugin/urls.py

app_name = "foo"

# This will be included under the "foo/" namespace
urlpatterns = [
    path("", TemplateView("foo/index.html").as_view(), name="index"),
    path("detail/", MyUrlView.as_view(), name="detail"),

    # ...
]
```

Global URLs

Sometimes, plugins need to provide top level URLs like */about* GDAPS also lets your plugin create those global, not namespaced URLs easily by using the `root_urlpatterns` attribute in your plugin's urls.py.

```
app_name = "about"

# This will be merged into the global "/" urlpattern
root_urlpatterns = [
    path("about/", SomeViewSet.as_view(), name="api")
```

(continues on next page)

(continued from previous page)

```
]

# and the ones under "/about/..."
urlpatterns =  [...]
```

Note: Plugins are self-responsible for their URLs and namespaces, and that they don't collide with others.

URL hooks

A third option which is a common pattern is that a plugin provides a "hook" under which *other* plugins can create sub-URLs. This is needed when you e.g. create an API, or a dashboard, or administration sites that should be pluggable. This is easy too with GDAPS. In *your_app/api/interfaces*, create a plugin interface:

```
@Interface
class IDashboardURL:
    urlpatterns = []
```

This interface offers a urlpattern that is included dynamically into the dashboard.

In your *global urls.py* file, you can include the interface as *dashboard/*:

```
urlpatterns = [
    ...
]
for plugin in IDashboardURL:
    urlpatterns += plugin.urlpattern
```

Add an IDashBoardURL implementation to your plugin's urls.py, and its urlpatterns will show up in the dashboard automatically:

```
# in myplugin
from your_app.api.interfaces import IDashboardURL
from . import views

class MyPluginDashboardURL(IDashboardURL):  # class name doesn't matter.
    urlpatterns = [
        path("about/", views.DashboardIndexView.as_view(), name="index/")
    ]
```

All patterns that are listed here are merged into the global .. _Settings:

3.1.8 Per-plugin Settings

GDAPS allows your application to have own settings for each plugin easily, which provide defaults, and can be overridden in the global `settings.py` file. Look at the example conf.py file (created by `./manage.py startplugin fooplugin`), and adapt to your needs:

```
from django.test.signals import setting_changed
from gdaps.conf import PluginSettings

NAMESPACE = "FOOPLUGIN"
```

(continues on next page)

(continued from previous page)

```
# Optional defaults. Leave empty if not needed.
DEFAULTS = {
    "MY_SETTING": "somevalue",
    "FOO_PATH": "django.blah.foo",
    "BAR": [
        "baz",
        "buh",
    ],
}

# Optional list of settings that are allowed to be in "string import" notation. Leave␣
↪empty if not needed.
IMPORT_STRINGS = (
    "FOO_PATH"
)

# Optional list of settings that have been removed. Leave empty if not needed.
REMOVED_SETTINGS = ( "FOO_SETTING" )


fooplugin_settings = PluginSettings("FOOPLUGIN", None, DEFAULTS, IMPORT_STRINGS)
```

Detailed explanation:

DEFAULTS The `DEFAULTS` are, as the name says, a default array of settings. If `fooplugin_setting.BLAH` is not set by the user in settings.py, this default value is used.

IMPORT_STRINGS Settings in a *dotted* notation are evaluated, they return not the string, but the object they point to. If it does not exist, an `ImportError` is raised.

REMOVED_SETTINGS A list of settings that are forbidden to use. If accessed, an `RuntimeError` is raised.

This allows very flexible settings - as dependant plugins can easily import the `fooplugin_settings` from your `conf.py`.

However, the created conf.py file is not needed, so if you don't use custom settings at all, just delete the file.

3.1.9 Admin site

GDAPS provides support for the Django admin site. The built-in `GdapsPlugin` model automatically are added to Django's admin site, and can be administered there.

Note: As GdapsPlugin database entries must not be edited directly, they are shown read-only in the admin. **Please use the 'syncplugins' management command to update the fields from the file system.** However, you can enable/disable or hide/show plugins via the admin interface.

If you want to disable the built-in admin site for GDAPS, or provide a custom GDAPS ModelAdmin, you can do this using:

```
GDAPS = {
    "ADMIN": False
}
```

Signals

If you are using Django signals in your plugin, we recommend to put them into a `signals` submodule. Import it then from the `AppConfig.ready()` method.

```
def ready(self):
    # Import signals if necessary:
    from . import signals  # NOQA
```

See also:

Don't overuse the `ready` method. Have a look at the Django documentation of ready().

CHAPTER

FOUR

ROUTERS DJANGO REST FRAMEWORK

DRF offers great router classes, but implementations always assume that your main urls.py knows about all of your apps. GDAPS lets you define one *SimpleRouter* for each of your apps, and automatically collects them into one global *DefaultRouter*.

In your global *urls.py* add:

```
router = PluginManager.router()
urlpatterns = [
    # ...
    path("api/", include(router.urls)),
]
```

In your apps' urls.py, similar to urlpatterns, create a *router* variable:

```
from rest_framework.routers import SimpleRouter

router = SimpleRouter()
router.register(r"app", AppListViewSet)
```

. . . where AppListViewSet is your DRF ViewSet. That's all, GDAPS takes care of the merging.

CHAPTER

FIVE

API

5.1 Interfaces/Implementations

5.2 Plugin configuration and metadata

5.3 PluginManager

5.4 Templates

class gdaps.api.interfaces.**ITemplatePluginMixin**

A mixin that can be inherited from to build renderable plugins.

Create an interface that inherits from ITemplatePluginMixin. Each implementation must either have a *template* (direct string HTML template, for short ones) or *template_name* (which points to the template to render).

You can add fixed *context*, or override *get_plugin_context* which receives the context of the view the plugin is rendered in.

In your template, use ` {% load gdaps %} <ul class="alerts"> {% render_plugin IMyListItem %} </ul> ` in your template anywhere, and all of your implementations are rendered, one after each other, in a line. Each plugin can come from another app. You can change the order of the rendering using the *weight* attribute.

context = {}

get_plugin_context(*context:* *django.template.context.Context*) → django.template.context.Context

Override this method to add custom context to the plugin.

Parameters **context** – the context where the plugin is rendered in. You can update it with own values, and return it. The return variable of this function will be the context of the rendered plugin. So if you don't update the passed context, but just return a new one, the plugin will not get access to the global context.

Per default, it merges the plugin's `context` attribute into the given global context.

template = ''

template_name = ''

weight = 0

5.5 Helper functions

CHAPTER

SIX

CONTRIBUTING

This is an Open Source project. Any help, ideas, and Code are welcome. If you want to contribute, feel free and write a PR, or contact me.

6.1 Code style

No compromises. Format your code using Black before committing.

CHAPTER

SEVEN

LICENSE

I'd like to give back what I received from many Open Source software packages, and keep this library as open as possible, and it should stay this way. GDAPS is licensed under the BSD License (BSD).

INDEX