GDAPS

Christian González

Sep 28, 2019

TABLE OF CONTENTS

Welcome to the GDAPS documentation!

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

Warning: This software is in an early development state. It's not considered to be used in production yet. Use it at your own risk. **You have been warned.**

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 Use *The plugin AppConfig*, ExtensionPoints, and *Implementations* **without a plugin/module system**. It's not necessary to divide your application into GDAPS apps to use GDAPS. Just code your application as usual and have an easy-to-use "observer pattern" plugin system.

- Define an interface
- Create one or more implementations for it and
- put an extensionpoint anywhere in your code.

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

Use this if you just want to structure your Django software using an "observer pattern". This is used within GDAPS itself, for the Javascript frontend implementations (e.g. Vue.js).

Full Use GDAPS as **complete module/app system**.

- You'll have to add "gdaps" to your INSTALLED_APPS first.
- Create plugins using the `startplugin` managemant command, and install them via pip/pipenv.

You have a 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 mode enables you to create fully-fledged extensible applications with real plugins that can be written by different parties and distributed via PyPi.

See usage for further instructions.

CHAPTER

TWO

INSTALLATION

Install GDAPS in your Python virtual environment (pipenv is preferred):

Create a Django application as usual: `manage.py startproject myproject`.

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
    # if you also want frontend support, add:
    "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("myproject.plugins")
```

You can use whatever you want for your plugin path, but we recommend that you use "**<myproject>.plugins**" here to make things easier. See *Usage*.

For further frontend specific instructions, see *Admin site*.

Basically, this is all you really need so far, for a minimal working GDAPS-enabled Django application.

2.1 Frontend support

If you want to add frontend support too your project, you need to do as follows:

First, add *gdaps*, *gdaps.frontend*, and *webpack_loader* to Django.

```
from gdaps.pluginmanager import PluginManager

INSTALLED_APPS = [
    # ... standard Django apps and GDAPS
    "gdaps.frontend"
    "gdaps",
    "webpack_loader",  # you'll need that too
]
INSTALLED_APPS += PluginManager.find_plugins("myproject.plugins")
```

Now, to satisfy webpack-loader, add a section to settings.py:

```
WEBPACK_LOADER = {}
```

You can leave that empty by now, it's just that it has to exist. Another section is needed for GDAPS:

```
GDAPS = {
    "FRONTEND_ENGINE": "vue",
}
```

The `FRONTEND_ENGINE` is used for the following command to setup the right frontend. ATM it can only be "vue". Now you can initialize the frontend with

This creates a basic boilerplate (previously created with 'vue create' and calls *yarn install* to install the needed javascript packages. .. _Usage: usage

CHAPTER

THREE

USAGE

3.1 Creating plugins

Create plugins using a Django management command:

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.

If you use git in your project, install the `gitpython` module (`pip/pipenv install gitpython --dev`). `startplugin` will determine your git user/email automatically and use at the right places.

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.apps.FooConfig",
]
```

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/pipenv, you can make your application aware of that plugin too:

```
pipenv 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, they don't have to be in the project's `plugins` folder. You can conveniently start developing plugins in there, and later move them to the PyPi repository.

3.2 The plugin AppConfig

Django recommends to point ot the app's AppConfig directly in INSTALLED_APPS. You should do that too with GDAPS plugins. Plugins that are installed via pip(env) are found automatically, as their AppConfig class must be named after the Plugin.

Plugins' AppConfigs must inherit from `gdaps.apps.PluginConfig`, and provide an inner class, or a pointer to an external `PluginMeta` class. For more information see *gdaps.apps.PluginConfig*.

3.3 Interfaces

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

```
from gdaps import Interface

class IFooInterface(Interface):
    """Documentation of the interface"""

    class Meta:
        service = True

    def do_something(self):
        pass
```

Interfaces can have a default Meta class that defines Interface options. Available options:

service If `service=True` (which is the default), then all implementations are instantiated instantly at definition time, having a full class instance availably at any time. Iterations over ExtensionPoints return the instances directly.

If you use `service=False`, the plugin is not instantiated, and iterations over ExtensionPoints will return **classes**, not instances. This sometimes may be the desired functionality, e.g. for data classes, or classes that just return staticmethods.

3.4 ExtensionPoints

An ExtensionPoint (EP) is a plugin hook that refers to an Interface. An EP can be defined anywhere in code. You can then get all the plugins that implement that interface by just iterating over that ExtensionPoint:

```
from gdaps import ExtensionPoint from
myproject.plugins.fooplugin.api.interfaces import IFooInterface

class MyPlugin:
```

(continues on next page)

(continued from previous page)

```
    ep = ExtensionPoint(IFooInterface)

    def foo_method(self):
        for plugin in ep:
            print plugin().do_domething()
```

Depending on the *service* Meta flag, iterating over an ExtensionPoint returns either a **class** (`service = False`) or an already instantiated **object** (`service = True`). Depending on your needs, just set *service* to the correct value. The default is *True*.

3.5 Implementations

You can then easily implement this interface in any other file (in this plugin or in another plugin) using the `@implements` decorator syntax:

```
from gdaps import implements
from myproject.plugins.fooplugin.api.interfaces import IFooInterface

@implements(IFooInterface)
class OtherPluginClass:

    def do_something(self):
        print('I did something!')
```

I didn't want to force implementations to inherit a `Plugin` base class, like some other plugin systems do. This would mean that implementations won't be as flexible as I wanted them. When just using a decorator, you can easily use ANY, even your already existing, class and just ducktype-implement the methods the Interface demands.

3.6 Extending Django's URL patterns

To let your plugin define some URLs that are automatically detected by your Django application, you have to add some code to your global urls.py file:

```
from gdaps.pluginmanager import PluginManager

urlpatterns =  [
    # add your fixed, non-plugin paths here.
]

# just add this line after the urlpatterns definition:
urlpatterns += PluginManager.urlpatterns()
```

GDAPS then loads and imports all available plugins' *urls.py* files, collects their `urlpatterns` variables and merges them into the global one.

A typical `fooplugin/urls.py` would look like this:

```
from . import views

app_name = fooplugin

urlpatterns =  [
```

(continues on next page)

(continued from previous page)

```
    path("/fooplugin/myurl", views.MyUrlView.as_view()),
]
```

GDAPS lets your plugin create global, root URLs, they are not namespaced. This is because soms plugins need to create URLS for frameworks like DRF, etc. Plugins are responsible for their URLs, and that they don't collide with others.

3.7 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"

# 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.

3.8 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
}
```

3.9 Frontend support

GDAPS supports Javascript frontends for building e.g. SPA applications. ATM only Vue.js ist supported, but PRs are welcome to add more (Angular, React?).

Just add `gdaps.frontend` to `INSTALLED_APPS`, **before** `gdaps`. Afterwords, there is a new management command available: `manage.py initfrontend`. It has one mandatory parameter, the frontend engine:

This creates a /frontend/ directory in the project root. Change into that directory and run `yarn install` once to install all the dependencies of Vue.js needed.

It is recommended to install vue globally, you can do that with `yarn global add @vue/cli @vue/cli-service-global`.

Now you can start `yarn serve` in the frontend directory. This starts a development web server that bundles the frontend app using webpack automatically. You then need to start Django using `./manage.py runserver` to enable the Django backend. GDAPS manages all the needed background tasks to transparently enable hot-reloading when you change anything in the frontend source code now.

3.9.1 Frontend plugins

Django itself provides a template engine, so you could use templates in your GDAPS apps to build the frontend parts too. But templates are not always the desired way to go. Since a few years, Javascript SPAs (Single Page Applications) have come up and promise fast, responsive software.

But: a SPA mostly is written as monolithic block. All tutorials that describe Django as backend recommend building the Django server modular, but it should serve only as API, namely REST or GraphQL. This API then should be consumed by a monolithic Javascript frontend, built by webpack etc. At least I didn't find anything else on the internet. So I created my own solution:

GDAPS is a plugin system. It provides backend plugins (Django apps). But using `gdaps.frontend`, each GDAPS app can use a *frontend* directory which contains an installable npm module, that is automatically installed when the app is added to the system.

When the `gdaps.frontend` app is activated in `INSTALLED_APPS`, the `startplugin` management command is extended by a frontend part: When a new plugin is created, a *frontend* directory in that plugin is initialized with

a boilerplate javascript file `index.js`, which is the plugin entry point in the frontend. This is accomplished by webpack and django-webpack-loader.

So all you have to do is:

1. Add `gdaps.frontend` to `INSTALLED_APPS` (before `gdaps`)
2. Call `./manage.py initfrontend vue`, if you haven't already
3. Call `./manage.py startplugin fooplugin` and fill out the questions
4. start `yarn serve` in the *frontend* directory
5. start Django server using `./manage.py runserver`

Webpack aggregates all you need into a package, using the `frontend/plugins.js` file as index where to find plugin entry points. You shouldn't manually edit that file, but just install GDAPS plugins as usual (pip, pipenv, or by adding them to INSTALLED_APPS) and call `manage.py syncplugins` then.

This command scans your app for plugins, updates the database with plugin data, and recreates the plugins.js file.

CHAPTER

FOUR

API

4.1 Interfaces/ExtensionPoints

class gdaps.**Interface**
Base class for interface definitions.

Inherit from *Interface* and eventually add methods to that class:

```
class IMyInterface(Interface):

    def do_something(self):
        pass
```

You can choose whatever name you want for your interfaces, but we recommend you start the name with a capital "I". Read more about interfaces in the *The plugin AppConfig* section.

gdaps.**implements**
alias of *gdaps.Implements*

class gdaps.**ExtensionPoint**(*interface: Type[gdaps.Interface]*)
Marker class for Extension points in plugins.

You can iterate over 'Extensionpoint's via `for..in`:

```
ep = ExtensionPoint(IMyInterface)
for plugin in ep:
    plugin.do_something()
```

extensions() → set
Returns a set of plugin instances that match the interface of this extension point.

class gdaps.**Implements**(**interfaces*)
Decorator class for implementing interfaces.

Just decorate a class with @implements to make it an implementation of an *Interface*:

```
@implements(IMyInterface)
class PluginA:

    def do_something(self):
        print("Greetings from PluginA")
```

You can also implement more than one interface: *@implements(InterfaceA, InterfaceB)* and implement all their methods.

Read more about implementations in the *Implementations* section.

4.2 PluginManager

class gdaps.pluginmanager.**PluginManager**

A Generic Django Plugin Manager that finds Django app plugins in a plugins folder or setuptools entry points and loads them dynamically.

It provides a couple of methods to interaft with plugins, load submodules of all available plugins dynamically, or get a list of enabled plugins. Don't instantiate a `PluginManager` directly, just use its static and class methods directly.

classmethod find_plugins(*group: str*) → List[str]

Finds plugins from setuptools entry points.

This function is supposed to be called in settings.py after the INSTALLED_APPS variable. Therefore it can not use global variables from settings, to prevent circle imports.

Parameters **group** – a dotted path where to find plugin apps. This is used as 'group' for setuptools' entry points.

Returns A list of dotted app_names, which can be appended to INSTALLED_APPS.

classmethod load_plugin_submodule(*submodule: str*, *mandatory=False*) → list

Search plugin apps for specific submodules and load them.

Parameters

- **submodule** – the dotted name of the Django app's submodule to import. This package must be a submodule of the plugin's namespace, e.g. "schema" - then ["<main>.core.schema", "<main>.laboratory.schema"] etc. will be found and imported.
- **mandatory** – If set to True, each found plugin _must_ contain the given submodule. If any installed plugin doesn't have it, a PluginError is raised.

Returns a list of module objects that have been successfully imported.

classmethod plugin_path()

Returns the absolute path where application plugins live.

This is basically the Django root + the dotted entry point. CAVE: this is not callable from within the settings.py file.

static plugins(*skip_disabled: bool = False*) → List[django.apps.AppConfig]

Returns a list of AppConfig classes that are GDAPS plugins.

This method basically checks for the presence of a `PluginMeta` class within the AppConfig of all apps and returns a list of them. :param skip_disabled: If True, skips disabled plugins and only returns enabled ones. Defaults to `False`.

static urlpatterns() → list

Loads all plugins' urls.py and collects their urlpatterns.

This is maybe not the best approach, but it allows plugins to have "global" URLs, and not only namespaced, and it is flexible

Returns a list of urlpatterns that can be merged with the global urls.urlpattern.

4.3 Plugin configuration and metadata

Plugins need to have a special AppConfig class. GDAPS provides a convenience `PluginConfig` class to inherit from:

class gdaps.apps.**PluginConfig**(**args*, ***kwargs*)

Base config class for GDAPS plugins.

All GDAPS plugin apps files need to have an AppConfig class which inherits from `PluginConfig`. It is a convenience class that checks for the existence of the PluginMeta inner class, and provides some basic methods that are needed when interacting with a plugin during its life cycle.

```
from django.utils.translation import gettext_lazy as _
from gdaps.apps import PluginConfig

class FooPluginConfig(PluginConfig):

    class PluginMeta:
        # the plugin machine "name" is taken from the Appconfig, so no name here
        verbose_name = _('Foo Plugin')
        author = 'Me Personally'
        description = _('A foo plugin')
        visible = True
        version = '1.0.0'
        compatibility = "myproject.core>=2.3.0"
```

If you are using signals in your plugin, we recommend to put them into a `signals` submodule. Import them 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().

If your plugin needs to install some data into the database at the first run, you can provide a `initialize` method, which will be called using the `initializeplugins` management command:

Do all necessary things there that need to be done when the plugin is available the first time, e.g. after installing a plugin using pip/pipenv.

```
def initialize(self):
    # install some fixtures, etc.
    pass
```

CHAPTER

FIVE

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.

5.1 Code style

No compromises. Format your code using Black before committing.

CHAPTER

SIX

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 General Public License, version 3.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

g

INDEX