# GDAPS

Christian González

Dec 04, 2020

# 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 *Interfaces*, 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 a monolithic django application have an easy-to-use "observer pattern" plugin system.

- import gdaps
- Define an interface
- Create one or more implementations for it and
- iterate over the interface to get all the implementations.

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** Use GDAPS as a **full-featured system to create modular applications**.

- Add "gdaps" to your INSTALLED_APPS.
- Create plugins using the `startplugin` managemant command, and install them via pip, (or add them to your INSTALLED_APPS too).

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:

```
pip install gdaps
```

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 ill be used in various places, e.g. when creating frontends for plugins. Must be a valid python identifier.

**Note:** `PROJECT_NAME` is roughly what Django means with `ROOT_URLCONF[0]`, but it's better to set it 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() + [
    # ... 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.

## 2.3 Frontend support

If you want to add frontend support to your project, you need to add `gdaps.frontend` (before `gdaps`!), and `webpack_loader` to Django.

Frontend engines are packed in plugin eggs. You can easily install them using pip, e.g.

```
pip install gdaps-frontend-vue
```

Then you have to tell Django which engine to use:

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

Further configuration may be necessary depending on your frontend plugin. Available plugins ATM:

- Vue
- PySide (currently only stub)

There are some keys in this section to configure:

### 2.3.1 FRONTEND_DIR

This is the directory for the frontend, relative to DJANGO_ROOT.

Default: `frontend`

### 2.3.2 FRONTEND_ENGINE

The engine which is used for setting up a frontend. ATM it can only be "vue". In future, maybe other engines are supported (Angular, React, etc.). PRs welcome.

### 2.3.3 FRONTEND_PKG_MANAGER

This is the package manager used to init/install packages. It depends on your frontend which are available.

### 2.3.4 PROJECT_TITLE

A title for your project. If left empty, PROJECT_NAME is taken (without hyphens/underlines and capitalized).

Default: `PROJECT_NAME`, without underlines and capitalized.

### 2.3.5 ADMIN

True if your app should add the GDAPS specific parts to the Django admin panel. You can e.g. see the installed plugins there.

Default: `True`

## 2.4 Final step

Now you can initialize the frontend with

```
./manage.py initfrontend
```

This creates a basic boilerplate frontend, depending on which engine is installed.

CHAPTER

THREE

# USAGE

## 3.1 Creating plugins

If you use git in your project, install the `gitpython` module (`pip install gitpython --dev`). `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, 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

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`.

## 3.3 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

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

    __service__ = True  # is the default

    def do_something(self):
        pass
```

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

**__service__** If `__service__ = True` is set (which is the default), then all implementations are instantiated directly at loading time, having a full class instance availably at any time. Iterations over Interfaces return **instances**:

```
for plugin in IFooInterface:
    plugin.do_something()
```

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

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

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

## 3.4 Implementations

You can then easily implement this interface in any other file (in this plugin or in another plugin) by subclassing the interface:

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

class OtherPluginClass(IFooInterface):

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

## 3.5 Using Implementations

You can straight-forwardly use implementations that are bound to an interface by iterating over that interface, anywhere in your code.

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

class MyPlugin:

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

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.

## 3.6 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
}
```

### 3.8.1 Frontend plugins

The GDAPS frontend module can be extended via plugins, each providing a pluggable frontend for your Django application. See

### 3.8.2 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

# API

## 4.1 Interfaces/Implementations

gdaps.api.**Interface**(*cls*)
Decorator for classes that are interfaces.

Declare an interface using the `@Interface` decorator, optionally add add attributes/methods to that class:

```
@Interface
class IFooInterface:
    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 *Interfaces* section.

## 4.2 Plugin configuration and metadata

**class** gdaps.api.**PluginMeta**
Inner class of GDAPS plugins.

All GDAPS plugin AppConfig classes need to have an inner class named `PluginMeta`. This PluginMeta provides some basic attributes and methods that are needed when interacting with a plugin during its life cycle.

```
from django.utils.translation import gettext_lazy as _
from django.apps import AppConfig

class FooPluginConfig(AppConfig):

    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"
```

**Note:** If `PluginMeta` is missing, the plugin is not recognized by GDAPS.

**author = 'Me, myself and Irene'**
The author of the plugin. Not translatable.

**author_email = 'me@example.com'**
The email address of the author

**category = 'GDAPS'**
A free-text category where your plugin belongs to. This can be used in your application to group plugins.

**compatibility = 'gdaps>=1.0.0'**
A string containing one or more other plugins that this plugin is known being compatible with, e.g. "myproject.core>=1.0.0<2.0.0" - meaning: This plugin is compatible with `myplugin.core` from version 1.0.0 to 1.x - v2.0 and above is incompatible.

> **Note:** Work In Progress.

**description = ''**
A longer text to describe the plugin.

**hidden = False**
A boolean value whether the plugin should be hidden, or visible. False by default.

**initialize**()
Callback to initialize the plugin.

This method is optional. It is called and run at Django start once. If your plugin needs to make some initial checks, do them here, but make them quick, as they slow down Django's start.

**install**()
Callback to setup the plugin for the first time.

This method is optional. If your plugin needs to install some data into the database at the first run, you can provide this method to `PluginMeta`. It will be called when `manage.py syncplugins` is called and the plugin is run, but only for the first time.

An example would be installing some fixtures, or providing a message to the user.

**verbose_name = 'My special plugin'**
The verbose name, as shown to the user

**version = '1.0.0'**
The version of the plugin, following Semantic Versioning. This is used for dependency checking as well, see `compatibility`.

**visible = True**
A boolean value whether the plugin should be visible, or hidden.

Deprecated since version 0.4.2: Use *hidden* instead.

**class** gdaps.api.**PluginConfig**(**args*, ***kwargs*)
Convenience class for GDAPS plugins to inherit from.

While it is not strictly necessary to inherit from this class - duck typing is ok - it simplifies the type suggestions and autocompletion of IDEs like PyCharm, as PluginMeta is already declared here.

## 4.3 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.

**static orphaned_plugins**() → django.db.models.QuerySet
Returns a list of GdapsPlugin models that have no disk representance any more.

---

**Note:** This method needs Django's ORM to be running.

---

**classmethod plugin_path**() → str
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[gdaps.api.PluginConfig]
Returns a list of AppConfig classes that are GDAPS plugins.

This method basically checks for the presence of a `PluginMeta` attribute within the AppConfig of all apps and returns a list of apps containing it. :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.

## 4.4 Helper functions

gdaps.api.**require_app**(*app_config: django.apps.AppConfig*, *required_app_name: str*) → None
Helper function for AppConfig.ready - checks if an app is installed.

An `ImproperlyConfigured` Exception is raised if the required app is not present.

**Parameters**

- **app_config** – the AppConfig which requires another app. usually use `self` here.
- **required_app_name** – the required app name.

CHAPTER

FIVE

# GDAPS FRONTEND SUPPORT

GDAPS supports frontends for building e.g. SPA applications. ATM only Vue.js ist supported well, but PRs are welcome to add more (Angular, React?). Even PySide or Qt5 would be possible.

## 5.1 Install

```
pip install gdaps-frontend
```

Add `gdaps.frontend` to `INSTALLED_APPS`, **before** `gdaps`.

The included *gdaps.frontend* package provides basic tools which then can be extended by other plugins, like *gdaps-frontend-vue*. You have to install at least one frontend plugin, e.g.

```
pip install gdaps-frontend-vue
```

GDAPS detects it automatically and makes the "vue" FRONTEND_ENGINE available.

```
# settings.py

GDAPS = {
    "FRONTEND_ENGINE": "vue",
    "FRONTEND_DIR": "frontend",
    "FRONTEND_PKG_MANAGER": "",
}
```

There are some keys here to configure:

**FRONTEND_ENGINE (mandatory)** The engine which is used for setting up a frontend. ATM it can only be "vue" or "pyside". See the [gdaps-frontend-vue package](gdaps-frontend-vue.readthedocs.org)

**FRONTEND_DIR (optional)** This is the directory for the frontend, relative to DJANGO_ROOT. **Default is "frontend".**

**FRONTEND_PKG_MANAGER (optional)** This is the package manager used to init/install packages. ATM you can use "yarn" or "npm". **Default is *npm*.**

## 5.2 Template overriding

`gdaps.frontend` renders a simple builtin index.html file as template.

If you need to override that template, e.g. your (Javascript?) frontend provides an own, you can do that: Just create an `index.html` file within your `<PROJECT_NAME>/templates` directory (e.g. *myproject/templates*). GDAPS searches for templates using Django's methods and will use any template that is found under that template name.

## 5.3 Management Commands

With `gdaps.frontend`, you have a new management command available. Set the `GDAPS["FRONTEND_ENGINE"]` to your desired engine ("vue", "pyside"), and call:

```
./manage.py initfrontend
```

This creates a /frontend/ directory in the project root, and installs a frontend application there. The type of frontend (and installation) depends on what you have selected in `GDAPS["FRONTEND_ENGINE"]`.

So all you have to do is:

1. Add `gdaps.frontend` to `INSTALLED_APPS` (before `gdaps`)
2. Install a frontend plugin, like *pip install gdaps-frontend-vue*.
3. Execute `./manage.py initfrontend`
4. Call `./manage.py startplugin fooplugin`
5. Call `./manage.py syncplugins`
6. start `yarn serve` in the *frontend* directory
7. start Django server using `./manage.py runserver`

To remove a plugin from the frontend, just remove the backend part (remove it from INSTALLED_APPS or uninstall it using pip) and call `manage.py syncplugins` again. It will take care of the database models, and the uninstallation of the frontend part.

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

# 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.

## 7.1 Code style

No compromises. Format your code using Black before committing.

CHAPTER

EIGHT

# 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.

# PYTHON MODULE INDEX

## g

# INDEX