Django

Documentation

Applications

New in Django 1.7.

Django contains a registry of installed applications that stores configuration and provides introspection. It also maintains a list of available models.

This registry is simply called **apps** and it's available in **django.apps**:

```
>>> from django.apps import apps
>>> apps.get_app_config('admin').verbose_name
'Admin'
```

Projects and applications

Django has historically used the term **project** to describe an installation of Django. A project is defined primarily by a settings module.

The term **application** describes a Python package that provides some set of features. Applications may be reused in various projects.

Note

This terminology is somewhat confusing these days as it became common to use the phrase "web app" to describe what equates to a Django project.

Applications include some combination of models, views, templates, template tags, static files, URLs, middleware, etc. They're generally wired into projects with the **INSTALLED_APPS** setting and optionally with other mechanisms such as URLconfs, the **MIDDLEWARE_CLASSES** setting, or template inheritance.

It is important to understand that a Django application is just a set of code that interacts with various parts of the framework. There's no such thing as an **Application** object. However, there's a few places where Django needs to interact with installed applications, mainly for configuration and also for introspection. That's why the application registry maintains metadata in an **AppConfig** instance for each installed application.

Configuring applications

To configure an application, subclass **AppConfig** and put the dotted path to that subclass in **INSTALLED_APPS**.

When **INSTALLED_APPS** simply contains the dotted path to an application module, Django checks for a **default_app_config** variable in that module.

If it's defined, it's the dotted path to the **AppConfig** subclass for that application.

If there is no **default_app_config**, Django uses the base **AppConfig** class.

default_app_config allows applications that predate Django 1.7 such as **django.contrib.admin** to opt-in to **AppConfig** features without requiring users to update their **INSTALLED_APPS**.

New applications should avoid **default_app_config**. Instead they should require the dotted path to the appropriate **AppConfig** subclass to be configured explicitly in **INSTALLED_APPS**.

For application authors

If you're creating a pluggable app called "Rock 'n' roll", here's how you would provide a proper name for the admin:

```
# rock_n_roll/apps.py

from django.apps import AppConfig

class RockNRollConfig(AppConfig):
    name = 'rock_n_roll'
    verbose_name = "Rock 'n' roll"
```

You can make your application load this **AppConfig** subclass by default as follows:

```
# rock_n_roll/__init__.py

default_app_config = 'rock_n_roll.apps.RockNRollConfig'
```

That will cause **RockNRollConfig** to be used when **INSTALLED_APPS** just contains **'rock_n_roll'**. This allows you to make use of **AppConfig** features without requiring your users to update their **INSTALLED_APPS** setting.

Of course, you can also tell your users to put **'rock_n_roll.apps.RockNRollConfig'** in their **INSTALLED_APPS** setting. You can even provide several different **AppConfig** subclasses with different behaviors and allow your users to choose one via their **INSTALLED_APPS** setting.

The recommended convention is to put the configuration class in a submodule of the application called **apps**. However, this isn't enforced by Django.

You must include the **name** attribute for Django to determine which application this configuration applies to. You can define any attributes documented in the **AppConfig** API reference.

Note

If your code imports the application registry in an application's **__init__.py**, the name **apps** will clash with the **apps** submodule. The best practice is to move that code to a submodule and import it. A workaround is to import the registry under a different name:

```
from django.apps import apps as django_apps
```

For application users

If you're using "Rock 'n' roll" in a project called **anthology**, but you want it to show up as "Gypsy jazz" instead, you can provide your own configuration:

```
# anthology/apps.py

from rock_n_roll.apps import RockNRollConfig

class GypsyJazzConfig(RockNRollConfig):
    verbose_name = "Gypsy jazz"

# anthology/settings.py

INSTALLED_APPS = [
    'anthology.apps.GypsyJazzConfig',
    # ...
]
```

Again, defining project-specific configuration classes in a submodule called **apps** is a convention, not a requirement.

Application configuration

***class* AppConfig[source]**

Application configuration objects store metadata for an application. Some attributes can be configured in **AppConfig** subclasses. Others are set by Django and read-only.

Configurable attributes

AppConfig.name

Full Python path to the application, e.g. **'django.contrib.admin'**.

This attribute defines which application the configuration applies to. It must be set in all **AppConfig** subclasses.

It must be unique across a Django project.

AppConfig.label

Short name for the application, e.g. **'admin'**

This attribute allows relabeling an application when two applications have conflicting labels. It defaults to the last component of **name**. It should be a valid Python identifier.

It must be unique across a Django project.

AppConfig.verbose_name

Human-readable name for the application, e.g. “Administration”.

This attribute defaults to **`label.title()`**.

`AppConfig.path`

Filesystem path to the application directory, e.g. **`'/usr/lib/python3.4/dist-packages/django/contrib/admin'`**.

In most cases, Django can automatically detect and set this, but you can also provide an explicit override as a class attribute on your **`AppConfig`** subclass. In a few situations this is required; for instance if the app package is a namespace package with multiple paths.

Read-only attributes

`AppConfig.module`

Root module for the application, e.g. **`<module 'django.contrib.admin' from 'django/contrib/admin/__init__.pyc'>`**.

`AppConfig.models_module`

Module containing the models, e.g. **`<module 'django.contrib.admin.models' from 'django/contrib/admin/models.pyc'>`**.

It may be **`None`** if the application doesn’t contain a **`models`** module. Note that the database related signals such as **`pre_migrate`** and **`post_migrate`** are only emitted for applications that have a **`models`** module.

Methods

`AppConfig.get_models()`[source]

Returns an iterable of **`Model`** classes for this application.

`AppConfig.get_model(`*model_name*`)`[source]

Returns the **`Model`** with the given **`model_name`**. Raises **`LookupError`** if no such model exists in this application. **`model_name`** is case-insensitive.

`AppConfig.ready()`[source]

Subclasses can override this method to perform initialization tasks such as registering signals. It is called as soon as the registry is fully populated.

You cannot import models in modules that define application configuration classes, but you can use **`get_model()`** to access a model class by name, like this:

```
def ready(self):
    MyModel = self.get_model('MyModel')
```

Warning

Although you can access model classes as described above, avoid interacting with the database in your **`ready()`** implementation. This includes model methods that execute queries (**`save()`**, **`delete()`**, manager methods etc.), and also raw SQL queries via **`django.db.connection`**. Your **`ready()`** method will run during startup of every management command. For example, even though the test database configuration is separate from the production settings, **`manage.py test`** would still execute some queries against your **production** database!

Note

In the usual initialization process, the **`ready`** method is only called once by Django. But in some corner cases, particularly in tests which are fiddling with installed applications, **`ready`** might be called more than once. In that case, either write idempotent methods, or put a flag on your **`AppConfig`** classes to prevent re-running code which should be executed exactly one time.

Namespace packages as apps (Python 3.3+)

Python versions 3.3 and later support Python packages without an **`__init__.py`** file. These packages are known as "namespace packages" and may be spread across multiple directories at different locations on **`sys.path`** (see **PEP 420**).

Django applications require a single base filesystem path where Django (depending on configuration) will search for templates, static assets, etc. Thus, namespace packages may only be Django applications if one of the following is true:

1. The namespace package actually has only a single location (i.e. is not spread across more than one directory.)
2. The **`AppConfig`** class used to configure the application has a **`path`** class attribute, which is the absolute directory path Django will use as the single base path for the application.

If neither of these conditions is met, Django will raise **`ImproperlyConfigured`**.

Application registry

`apps`

The application registry provides the following public API. Methods that aren't listed below are considered private and may change without notice.

`apps.ready`

Boolean attribute that is set to **`True`** when the registry is fully populated.

`apps.get_app_configs`()

Returns an iterable of **`AppConfig`** instances.

`apps.get_app_config`(*app_label*)

Returns an **`AppConfig`** for the application with the given **`app_label`**. Raises **`LookupError`** if no such application exists.

`apps.is_installed`(*app_name*)

Checks whether an application with the given name exists in the registry. **app_name** is the full name of the app, e.g. **'django.contrib.admin'**.

apps.get_model(*app_label*, *model_name*)

Returns the **Model** with the given **app_label** and **model_name**. As a shortcut, this method also accepts a single argument in the form **app_label.model_name**. **model_name** is case- insensitive.

Raises **LookupError** if no such application or model exists. Raises **ValueError** when called with a single argument that doesn't contain exactly one dot.

Initialization process

How applications are loaded

When Django starts, **django.setup()** is responsible for populating the application registry.

setup()[source]

Configures Django by:

- Loading the settings.
- Setting up logging.
- Initializing the application registry.

This function is called automatically:

- When running an HTTP server via Django's WSGI support.
- When invoking a management command.

It must be called explicitly in other cases, for instance in plain Python scripts.

The application registry is initialized in three stages. At each stage, Django processes all applications in the order of **INSTALLED_APPS**.

1. First Django imports each item in **INSTALLED_APPS**.

   If it's an application configuration class, Django imports the root package of the application, defined by its **name** attribute. If it's a Python package, Django creates a default application configuration.

   At this stage, your code shouldn't import any models!

   In other words, your applications' root packages and the modules that define your application configuration classes shouldn't import any models, even indirectly.

   Strictly speaking, Django allows importing models once their application configuration is loaded. However, in order to avoid needless constraints on the order of **INSTALLED_APPS**, it's strongly recommended not import any models at this stage.

   Once this stage completes, APIs that operate on application configurations such as **get_app_config()** become usable.

2. Then Django attempts to import the **models** submodule of each application, if there is one.

   You must define or import all models in your application's **models.py** or **models/__init__.py**. Otherwise, the application registry may not be fully populated at this point, which could cause the ORM to malfunction.

   Once this stage completes, APIs that operate on models such as **get_model()** become usable.

3. Finally Django runs the **`ready()`** method of each application configuration.

Troubleshooting

Here are some common problems that you may encounter during initialization:

- **`AppRegistryNotReady`** This happens when importing an application configuration or a models module triggers code that depends on the app registry.

  For example, **`ugettext()`** uses the app registry to look up translation catalogs in applications. To translate at import time, you need **`ugettext_lazy()`** instead. (Using **`ugettext()`** would be a bug, because the translation would happen at import time, rather than at each request depending on the active language.)

  Executing database queries with the ORM at import time in models modules will also trigger this exception. The ORM cannot function properly until all models are available.

  Another common culprit is **`django.contrib.auth.get_user_model()`**. Use the **`AUTH_USER_MODEL`** setting to reference the User model at import time.

  This exception also happens if you forget to call **`django.setup()`** in a standalone Python script.

- **`ImportError: cannot import name ...`** This happens if the import sequence ends up in a loop.

  To eliminate such problems, you should minimize dependencies between your models modules and do as little work as possible at import time. To avoid executing code at import time, you can move it into a function and cache its results. The code will be executed when you first need its results. This concept is known as “lazy evaluation”.

- **`django.contrib.admin`** automatically performs autodiscovery of **`admin`** modules in installed applications. To prevent it, change your **`INSTALLED_APPS`** to contain **`'django.contrib.admin.apps.SimpleAdminConfig'`** instead of **`'django.contrib.admin'`**.

API Reference

System check framework

Learn More

About Django

Getting Started with Django

Team Organization

Django Software Foundation

Code of Conduct

Diversity Statement