Learning Objectives

- **Define environment variable**
- **Reference, set, and override environment variables**
- **Use environment variables to configure your Django project**

Clone Blango Repo

Clone Blango Repo

Before we continue, you need to clone the `blango` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/blango.git
```

- You should see a `blango` directory appear in the file tree.

You are now ready for the next assignment.

Environment Variables

Django Configurations

What are environment variables?

We briefly discussed environment variables in the last section, in relation to configuration of 12 factor apps. But to give a better introduction to them and how they work with Python, we'll use an example script.

Environment variables are strings that are set before starting a process. Or to get more technical, a process inherits the environment variables of the process that launched it. It can then set or override its own environment variables, which are in turn used in processes it spawns.

For example:

- Your shell sets the environment variables `MY_VAR_1` to `foo` and `MY_VAR_2` to `bar`.
- You then start Python, and have access to both of these values.
- Inside Python, you set `MY_VAR_1` to `baz` and start a sub process. When you access this environment variable in Python in the future, it is now `baz`.
- That subprocess's `MY_VAR_1` environment variable would be `baz`, while its `MY_VAR_2` would still be `bar`.

Environment variables are available in Python in the `os.environ` variable, which is kind-of like a dictionary. But you can also set and override the values; or set default environment variables inside the process, which will be used if the parent process hasn't set a value for that environment variable.

This can be easier to understand with a simple script. Copy and paste the code below into the `environ_test.py` script to the left.

```
from os import environ

environ.setdefault("PYTHON_DEFAULT", "Python Default")

print(f"Value of 'MUST_BE_SET': '{environ['MUST_BE_SET']}'")
print(f"Value of 'PYTHON_DEFAULT':
        '{environ['PYTHON_DEFAULT']}'")

try:
    print(f"Value of 'ALWAYS_OVERRIDDEN' before override:
        '{environ['ALWAYS_OVERRIDDEN']}'")
except KeyError:
    print("'ALWAYS_OVERRIDDEN' was not set.")

environ["ALWAYS_OVERRIDDEN"] = "Always Overridden In Python"

print(f"Value of 'ALWAYS_OVERRIDDEN' after override:
        '{environ['ALWAYS_OVERRIDDEN']}'")
print(f"Value of 'OPTIONAL': '{environ.get('OPTIONAL')}'")
```

Here four concepts are demonstrated:

- `MUST_BE_SET` is an environment variable that must be set before the script runs, otherwise it will fail with a `KeyError`
- `PYTHON_DFEFAULT` is set with the `environ.setdefault()` method, however this value will only be used if the shell (or other parent process) doesn't specify a value for `PYTHON_DEFAULT`.
- `ALWAYS_OVERRIDDEN` is set inside the script, so any value that is set in the shell is not used as it's replaced by `"Python Default"`
- `OPTIONAL` is retrieved using the `environ.get()` method. Like the `dict.get()` method this returns `None` if the value is not set in the environment. Note that we could have retrieved any of the environment variables using `get()` if we wanted, for extra safety.

First, try running the script in the terminal without setting any environment variables:

```
python3 environ_test.py
```

You should see the following output:

```
Traceback (most recent call last):
 ...
KeyError: 'MUST_BE_SET'
```

As we expect, a `KeyError` is raised as `MUST_BE_SET` is not in the environment.

How do we set environment variables? There are a few methods, but here we'll cover the two most common. The first way is to set them just for the command you're going to run, by prepending `VAR=value` before the command.

For example, manually set the value in the terminal and then call the Python script:

```
MUST_BE_SET="is now set" python3 environ_test.py
```

You should see the following output:

```
Value of 'MUST_BE_SET': 'is now set'
Value of 'PYTHON_DEFAULT': 'Python Default'
'ALWAYS_OVERRIDDEN' was not set.
Value of 'ALWAYS_OVERRIDDEN' after override: 'Always Overridden
        In Python'
Value of 'OPTIONAL': 'None'
```

If you want to set multiple environment variables in this way, just add more, with a space between them.

```
MUST_BE_SET="is now set" OPTIONAL="an optional value" python3
        environ_test.py
```

You should see the following output:

```
Value of 'MUST_BE_SET': 'is now set'
Value of 'PYTHON_DEFAULT': 'Python Default'
'ALWAYS_OVERRIDDEN' was not set.
Value of 'ALWAYS_OVERRIDDEN' after override: 'Always Overridden
        In Python'
Value of 'OPTIONAL': 'an optional value'
```

Environment variables can also be set with the `export` command, and they will be set for all subsequent commands in the shell. After you close the session (by logging out or closing the terminal window) the exported environment variables will be removed. Use the `export` command to set the value of `MUST_BE_SET`, then call the Python script in the terminal.

```
export MUST_BE_SET="set in export"
python3 environ_test.py
```

You should see the following output:

```
Value of 'MUST_BE_SET': 'set in export'
Value of 'PYTHON_DEFAULT': 'Python Default'
'ALWAYS_OVERRIDDEN' was not set.
Value of 'ALWAYS_OVERRIDDEN' after override: 'Always Overridden
        In Python'
Value of 'OPTIONAL': 'None'
```

Environment variables can also be unset with the `unset` command. First set the value for `PYTHON_DEFAULT` and run the script in the terminal.

```
export PYTHON_DEFAULT="also exported"
python3 environ_test.py
```

You should see the following output:

```
Value of 'MUST_BE_SET': 'set in export'
Value of 'PYTHON_DEFAULT': 'also exported'
'ALWAYS_OVERRIDDEN' was not set.
Value of 'ALWAYS_OVERRIDDEN' after override: 'Always Overridden
        In Python'
Value of 'OPTIONAL': 'None'
```

Now use the `unset` command to return `PYTHON_DEFAULT` to its default value. Run the script again.

```
unset PYTHON_DEFAULT
python3 environ_test.py
```

`PYTHON_DEFAULT` should now be its original value:

```
Value of 'MUST_BE_SET': 'set in export'
Value of 'PYTHON_DEFAULT': 'Python Default'
'ALWAYS_OVERRIDDEN' was not set.
Value of 'ALWAYS_OVERRIDDEN' after override: 'Always Overridden
        In Python'
Value of 'OPTIONAL': 'None'
```

challenge

Try it out:

Using `export`, set the value of the following environment variables:

- Set `ALWAYS_OVERRIDDEN` to `"Environment variables are useful"`.

▼ **Solution**

```
export ALWAYS_OVERRIDDEN="Environment variables are useful"
python3 environ_test.py
```

- Set `OPTIONAL` to `"The OPTIONAL variable now has a value"`.

▼ **Solution**

```
export OPTIONAL="The OPTIONAL variable now has a value"
python3 environ_test.py
```

Apart from setting environment variables in a shell, there are other methods. When deploying a Django application to production, it's not run through a shell (i.e. we don't have someone run `python3 manage.py runserver` to start the application). Instead it's executed through an app server (like Gunicorn or uWSGI) that's started by the operating system. These have environment variables in a configuration file that are read and then fed through to Python/Django.

Alternatively, if you're deploying using a cloud app server or container service, you might have a web GUI to set the environment variables which are passed to the process.

To summarize though, environment variables are a flexible way of injecting configuration into your application, with many different ways of doing so.

But now you might be wondering how to actually use environment variables for Django settings. Let's look at that next.

Django Configurations Part 1

Django Configurations Part 1

Now that we know how to read environment variables in Python, you might have an idea of how we could use them in Django. A naïve method could be to import `environ` in `settings.py` and then read the values you need.

For example, to set the `TIME_ZONE`

```
TIME_ZONE = environ.get("TIME_ZONE", "UTC")
```

But environment variables are strings, and so you would have to parse them to extract some settings. For example, `DEBUG` must be a boolean, `ALLOWED_HOSTS` a list, and `DATABASES` a dictionary. Of course you could write all this code yourself, but Django Configurations has done it for you, along with some handy extra features.

Installing Django Configurations

Django Configurations is installable using `pip`, its package name is `django-configurations`. We also want a helper package called `dj-database-url` which will let us set up the database configuration using a URL string instead of a dictionary. Install both of these packages in the terminal.

```
pip3 install django-configurations dj-database-url
```

Updating `settings.py`

As well as having helpers for reading and parsing environment variables, Django Configurations moves settings into classes, to give you object-oriented features. A common pattern is to have a `Debug` class which stores most your settings, with a `Production` class that inherits from it. The `Production` class can override certain settings, like forcing `DEBUG` to `False` for security. The classes inherit from `configurations.Configuration`.

To demonstrate, before converting our settings file looks like this:

```
import os
from pathlib import Path

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent

# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/3.2/howto/deployment/checklist/


# SECURITY WARNING: keep the secret key used in production
        secret!
SECRET_KEY = "django-insecure-
        ym=d)ft4%)xiukqr&tgstl6i2091+x_#&o%*%n6g^epgy(bpd6"

# SECURITY WARNING: don't run with debug turned on in
        production!
DEBUG = True

ALLOWED_HOSTS = []
# other settings truncated for brevity
```

After converting to use Django Configurations, your settings should be a part of the `Dev` class. Doing this initial setup is actually quite easy, we can just add the import and define the class. Then, we can select all the settings and type **Tab** and your IDE should indent them correctly for you.

Make the changes as described above. Your file should now look something like this (**Important** the code sample below is incomplete; do not copy/paste it into your file.)

Open settings.py

```
import os
from pathlib import Path
from configurations import Configuration


class Dev(Configuration):
    # Build paths inside the project like this: BASE_DIR /
        'subdir'.
    BASE_DIR = Path(__file__).resolve().parent.parent

    # Quick-start development settings - unsuitable for
        production
    # See
        https://docs.djangoproject.com/en/3.2/howto/deployment/checklist/



    # SECURITY WARNING: keep the secret key used in production
        secret!
    SECRET_KEY = "django-insecure-
        ym=d)ft4%)xiukqr&tgstl6i2091+x_#&o%*%n6g^epgy(bpd6"
    # SECURITY WARNING: don't run with debug turned on in
        production!
    DEBUG = True

    ALLOWED_HOSTS = []
    # other settings truncated for brevity
```

Django Configurations Part 2

Django Configurations Part 2

Updating `manage.py`

Before Django can load the new type of settings, we also need to make some changes to the entry point scripts (`manage.py` and `wsgi.py`). Django Configurations provides a couple of custom functions that replace the built-in Django ones and know how to load the settings from the classes.

First in `manage.py`, an environment variable is already being set to define the default Django settings file to load. For example, in Blango it's set like this:

```
os.environ.setdefault("DJANGO_SETTINGS_MODULE",
        "blango.settings")
```

Django Configuration requires another environment variable to specify which of the `Configuration` classes it should load, so we can add a line to do so underneath:

```
os.environ.setdefault("DJANGO_CONFIGURATION", "Dev")
```

Next we replace the Django built-in `execute_from_command_line` with one that Django Configurations provides. The line:

```
from django.core.management import execute_from_command_line
```

is changed to:

```
from configurations.management import execute_from_command_line
```

Updating `wsgi.py`

We need to make similar changes to `wsgi.py` (which is in the same directory as `settings.py`).

▼ **What is wsgi.py?**
`wsgi.py` is used by an application server to load your web server, when you deploy to production. We don't actually make use of it during development

but it's good to make the changes to support Django Configurations now to make sure everything is compatible going forward.

Since `wsgi.py` is only used in production, we'll default our `DJANGO_CONFIGURATION` setting to `Prod` instead:

Open wsgi.py

```
os.environ.setdefault("DJANGO_CONFIGURATION", "Prod")
```

The built-in `get_wsgi_application` is replaced with one that Django Configurations provides. So this line:

```
from django.core.wsgi import get_wsgi_application
```

is changed to:

```
from configurations.wsgi import get_wsgi_application
```

We actually need to change the order of the imports as well, since trying to import Django Configuration's `get_wsgi_application` will fail if `DJANGO_CONFIGURATION` is not yet defined, so after all the updates `wsgi.py` will look like this (minus the introductory comments):

```
import os

os.environ.setdefault("DJANGO_SETTINGS_MODULE",
        "blango.settings")
os.environ.setdefault("DJANGO_CONFIGURATION", "Prod")

from configurations.wsgi import get_wsgi_application

application = get_wsgi_application()
```

After that's done, restart the Django dev server.

You shouldn't notice any change, except in the terminal you'll get a message like:

```
django-configurations version , using configuration Dev
```

(And yes it's normal to not have a version number displayed in the output sometimes).

Now Django Configurations is configured, but it won't read values from environment variables without being told which ones to read. Let's look at how to do that.

Django Configurations Values Part 1

Django Configurations Values Part 1

To have Django Configurations read configuration from environment variables, the `configurations.values.Value` (or one of its subclasses) must be specified, for those values that are to be read.

Let's start simply with the `Value` class itself. It reads a string from an environment variable, and takes one required argument: its default value.

You would use it for simple string values; `TIME_ZONE` is a good example. To update settings so `TIME_ZONE` is read from an environment variable, first bring in the `values` module with an import:

```
from configurations import values
```

Then update the setting, from:

```
TIME_ZONE = "UTC"
```

to:

```
TIME_ZONE = values.Value("UTC")
```

By default, the environment variable names are prefixed by `DJANGO_`, for example, `TIME_ZONE` is read from `DJANGO_TIME_ZONE`.

You can test how his works by making the change to your `settings.py` and then temporarily adding a little bit of debug code to `urls.py`. This file is chosen arbitrarily just because it's parsed every time the Django dev server starts and so will always give you some output.

Open settings.py

First, add the `from configurations import values` import at the top of `settings.py`. Then, change the `TIME_ZONE` setting to:

```
TIME_ZONE = values.Value("UTC")
```

Finally open `urls.py` and add these two lines at the bottom of the file:

Open urls.py

```
from django.conf import settings
print(f"Time zone: {settings.TIME_ZONE}")
```

Now, start the Django dev server with the `runserver` command.

```
python3 manage.py runserver 0.0.0.0:8000
```

You'll see output like:

```
django-configurations version , using configuration Dev
Watching for file changes with StatReloader
Performing system checks...

Time zone: UTC
System check identified no issues (0 silenced).
July 22, 2021 - 22:14:18
```

Stop the server by pressing **Control** and **C** on the keyboard.

challenge

Try it out:

- Set Auckland, New Zealand as the value for the `DJANGO_TIME_ZONE` environment variable.

```
DJANGO_TIME_ZONE="Pacific/Auckland" python3 manage.py
        runserver 0.0.0.0:8000
```

- Try setting a different `DJANGO_TIME_ZONE` value as an environment variable while you execute `runserver` again. Note that it must be a valid timezone. Note also that Django outputs the start time in the local time that we've specified.

Let's try one more thing, which is to change the prefix that the `Value` class expects for `TIME_ZONE`. We'll pass in the argument `environ_prefix="BLANGO"` when instantiating the `Value` class in `settings.py`:

```
TIME_ZONE = values.Value("UTC", environ_prefix="BLANGO")
```

Now Django Configurations will look for the environment variable `BLANGO_TIME_ZONE` for that setting:

```
BLANGO_TIME_ZONE="Pacific/Auckland" python3 manage.py runserver
        0.0.0.0:8000
```

Your output should be something similar to:

```
django-configurations version , using configuration Dev
Watching for file changes with StatReloader
Performing system checks...

Time zone: Pacific/Auckland
System check identified no issues (0 silenced).
July 23, 2021 - 10:20:10
```

Note that the `environ_prefix` applies only to that use of the `Value` class. Any other uses of the class would still look for the prefix `DJANGO` (or whatever prefix was specified for them).

Before moving on we'll clean up some of the testing changes. First remove the `environ_prefix="BLANGO"` argument from the `TIME_ZONE` setting, so it's back to this:

```
TIME_ZONE = values.Value("UTC")
```

Then open `urls.py` and remove the two testing lines you added (the `settings` import and `print` call). Restart the Django dev server to make sure it's all working.

▼ **Original Files**

If you made changes to your project and are concerned about getting it back to its original state, here is the original code for the files that were changed.

- Original code for `settings.py`:

```
"""
Django settings for blango project.

Generated by 'django-admin startproject' using Django 3.2.5.

For more information on this file, see
https://docs.djangoproject.com/en/3.2/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/3.2/ref/settings/
"""

import os
from pathlib import Path
from configurations import Configuration
```

```
from configurations import values

class Dev(Configuration):

  # Build paths inside the project like this: BASE_DIR /
        'subdir'.
  BASE_DIR = Path(__file__).resolve().parent.parent



  # Quick-start development settings - unsuitable for production
  # See https://docs.djangoproject.com/en/3.2/howto/deployment/checklist/



  # SECURITY WARNING: keep the secret key used in production
        secret!
  SECRET_KEY = 'django-insecure-&!=9y436&^-bc$qia-
        mxngyf&xx)@ct)8lu@)=qxg_07-=z01w'

  # SECURITY WARNING: don't run with debug turned on in
        production!
  DEBUG = True

  ALLOWED_HOSTS = ['*']
  X_FRAME_OPTIONS = 'ALLOW-FROM ' +
        os.environ.get('CODIO_HOSTNAME') + '-8000.codio.io'
  CSRF_COOKIE_SAMESITE = None
  CSRF_TRUSTED_ORIGINS = [os.environ.get('CODIO_HOSTNAME') +
        '-8000.codio.io']
  CSRF_COOKIE_SECURE = True
  SESSION_COOKIE_SECURE = True
  CSRF_COOKIE_SAMESITE = 'None'
  SESSION_COOKIE_SAMESITE = 'None'

  # Application definition

  INSTALLED_APPS = [
      'django.contrib.admin',
      'django.contrib.auth',
      'django.contrib.contenttypes',
      'django.contrib.sessions',
      'django.contrib.messages',
      'django.contrib.staticfiles',
      'blog',
      'crispy_forms',
      'crispy_bootstrap5',
  ]

  MIDDLEWARE = [
      'django.middleware.security.SecurityMiddleware',
      'django.contrib.sessions.middleware.SessionMiddleware',
      'django.middleware.common.CommonMiddleware',
  #     'django.middleware.csrf.CsrfViewMiddleware',
      'django.contrib.auth.middleware.AuthenticationMiddleware',
      'django.contrib.messages.middleware.MessageMiddleware',
```

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

]

ROOT_URLCONF = 'blango.urls'

TEMPLATES = [
    {
        'BACKEND':
      'django.template.backends.django.DjangoTemplates',
        "DIRS": [BASE_DIR / "templates"],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',

      'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]

WSGI_APPLICATION = 'blango.wsgi.application'


# Database
# https://docs.djangoproject.com/en/3.2/ref/settings/#databases


DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}


# Password validation
# https://docs.djangoproject.com/en/3.2/ref/settings/#auth-
      password-validators


AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME':
      'django.contrib.auth.password_validation.UserAttributeSimilarityValida
      r',


    },
    {
```

```
            'NAME':
          'django.contrib.auth.password_validation.MinimumLengthValidato
          r',

        },
        {
            'NAME':
          'django.contrib.auth.password_validation.CommonPasswordValidato
          r',

        },
        {
            'NAME':
          'django.contrib.auth.password_validation.NumericPasswordValidato
          r',

        },
]



# Internationalization
# https://docs.djangoproject.com/en/3.2/topics/i18n/

LANGUAGE_CODE = 'en-us'

TIME_ZONE = values.Value("UTC")

USE_I18N = True

USE_L10N = True

USE_TZ = True



# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/3.2/howto/static-files/

STATIC_URL = '/static/'

# Default primary key field type
# https://docs.djangoproject.com/en/3.2/ref/settings/#default-
      auto-field



DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
CRISPY_ALLOWED_TEMPLATE_PACKS = "bootstrap5"
CRISPY_TEMPLATE_PACK = "bootstrap5"
```

- Original code for `urls.py`:

```
import blog.views

"""blango URL Configuration

The `urlpatterns` list routes URLs to views. For more
        information please see:
    https://docs.djangoproject.com/en/3.2/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home,
        name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(),
        name='home')
Including another URLconf
    1. Import the include() function: from django.urls import
        include, path
    2. Add a URL to urlpatterns:  path('blog/',
        include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
    path("", blog.views.index),
    path("post/<slug>/", blog.views.post_detail, name="blog-
        post-detail")
]
```

Next we'll look at some of the more advanced `Value` subclasses that are available, and implement them in a `Prod` settings class.

Django Configurations Values Part 2

Django Configurations Values Part 2

Parsing Other Configuration Types

The `Value` class is the simplest type of configuration, it just reads a string from an environment variable and sets it to a particular setting. But you've seen in `settings.py` that settings can be booleans, lists, dictionaries and more. Django Configurations provides a number of `Value` subclasses that can be used to both parse these values from strings, and validate their format.

Before we start using these new classes, we'll implement the class to store production settings: `Prod`. You'll recall we referred to this class in `wsgi.py` earlier. The reason we need this class is because some classes we're using (namely `SecretValue`) don't make sense in the development configuration context – more on `SecretValue` later.

The `Prod` class is defined like any other Python class. It comes after the `Dev` class and inherits from it.

```
class Prod(Dev):
```

BooleanValue

Now let's set some settings (class attributes), first, `DEBUG`. When we're in production we definitely want `DEBUG` to be off, for security reasons. We'll force it to `False` in the `Prod` class:

```
class Prod(Dev):
    DEBUG = False
```

However in development, we want it to be `True` by default, but have the option to set it false with an environment variable. We can't use `Value` here, because in Python non-empty strings are truthy. So we could try to set

DEBUG to the string `"false"` and it would evaluate to true! We need to use a `BooleanValue` class instead.

This will parse an environment variable string into a boolean:

- `True` values are `"yes"`, `"y"`, `"true"` and `"1"`
- `False` values are `"no"`, `"n"`, `"false"`, `"0"` and `""` (empty string)

In the `Dev` class we use it like this:

```
DEBUG = values.BooleanValue(True)
```

SecretValue

Next, `SECRET_KEY`. Your production secret key is not something that you want to be made public, and one way for secret data like this to leak might be if it were accidentally checked into a public Github repository. Here's a story about someone accidentally checking AWS access tokens into Github, it doesn't take long for attackers to find them and set up virtual machine instances.

While Django can't prevent you from committing private data like this, by using the `SecretValue` class it can at least prevent you from using a key that has been commmited in code, in production.

`SecretValue` does not take a default value, and if one is provided, an exception is raised.

We can leave our `Dev` `SECRET_KEY` value hard coded, but for the `Prod` class we will set it like this:

```
class Prod(Dev):
    DEBUG = False
    SECRET_KEY = values.SecretValue()
```

If we were to set it like this:

```
class Prod(Dev):
    DEBUG = False
    SECRET_KEY = values.SecretValue("any-hard-coded-value")
```

then, when starting Django in production, you'd get an exception, thus preventing you from using a potentially committed and compromised value.

ListValue

The `ALLOWED_HOSTS` setting is a list, and Django Configurations provides a way to work with this too: the `ListValue` class. This parses a comma-separated string into a list of values. We still set the default value as a list though.

```
ALLOWED_HOSTS = values.ListValue(["localhost", "0.0.0.0",
        ".codio.io"])
```

We can make this change just on the `Dev` class and since `Prod` inherits this, it will pick it up too.

The equivalent way of setting these values as an environment variable, when using the `runserver` command, would be like this:

```
ALLOWED_HOSTS=localhost,0.0.0.0,.codio.io python3 manage.py
        runserver 0.0.0.0:8000
```

challenge

Try it out:

Verify that your blog still runs as expected after these changes.

View Blog

Django Configurations Values Part 3

Django Configurations Values Part 3

DatabaseURLValue

The last `Value` class that we're going to look at is `DatabaseURLValue`, which parses a database URL into a dictionary for Django's `DATABASES` setting. This requires the `dj_database_url` package which we installed earlier. Import `dj_database_url` before making any changes.

```
import dj_database_url
```

URL schemas for many different databases are supported, including:
* PostgreSQL - `postgres://`
* MSSQL - `mssql://`
* MySQL - `mysql://`

Username, password, address, port, database name, and sometime other options, can be included in the URL. For example, to connect to a MySQL database:

```
mysql://username:password@mysql-host.example.com:3306/db_name?
        option1=value1&option2=value2
```

We'll set the default to our existing SQLite database:

```
DATABASES =
        values.DatabaseURLValue(f"sqlite:///{BASE_DIR}/db.sqlite3")
```

▼ **Username, password, and host**
You'll notice that we don't need username, password, and host for SQLLite. Also notice that there are 3 slashes after the schema: this indicates the empty hostname.

A full list of schemata and formats can be found in the dj-database-url project homepage

There are a couple of things to be aware of when using `DatabaseURLValue`. It differs from the other `Value` classes in that it doesn't read the value from the environment variable `DJANGO_DATABASES`, as you would expect from the convention seen so far. Instead, it reads from the environment variable `DATABASE_URL`, which is "inspired by the Twelve-Factor methodology" that we learned about in the last chapter.

Django allows you to work with different databases for different models. It's the reason why `DATABASES` is a dictionary and the connection settings for the database are in the `default` key. Although we won't make use of different databases, let's look at how to handle them with Django Configurations. `DatabaseURLValue` parses the URL into a dictionary and populates the `default` key so it's not possible to handle multiple databases. We need to drop down into the lower level `dj_database_url` library and make use of its `config()` function. This is similar to `DatabaseURLValue` in that you can provide a default, but it returns the database config dictionary only rather than the enclosing dictionary. We would use it like this:

```
DATABASES = {
    "default":
        dj_database_url.config(default=f"sqlite:///{BASE_DIR}/db
        .sqlite3"),
    "alternative": dj_database_url.config(
        "ALTERNATIVE_DATABASE_URL",
        default=f"sqlite:///{BASE_DIR}/alternative_db.sqlite3",
    ),
}
```

The `default` database connection is read from the `DATABASE_URL` environment variable. We've passed the argument `ALTERNATIVE_DATABASE_URL`, which indicates that the `alternative` database should be read from the `ALTERNATIVE_DATABASE_URL` database variable. Since we're only using the `default` database in Blango, we'll stick to just the `DatabaseURLValue` class.

Other Value Subclasses

These are the only `Value` subclasses we'll be using. This background should give you all the information you need to start using others. The full list is available at the Value class documentation. Some other subclasses provide helpers for converting values to integers, floats or decimals, validating email address or arbitrarily against regular expressions, or parsing nested lists of lists. You can also create your own `Value` subclass with a custom `to_python()` method, to parse values any way you choose.

We've only updated a few settings to be read from environment variables. This is because most of the settings don't need to be changed between development and production environments. For example, `INSTALLED_APPS`, `MIDDLEWARE`, `ROOT_URLCONF`, `AUTH_PASSWORD_VALIDATORS`, and others, stay the same between all deployments (usually). Some projects choose to convert all the settings to use `Value` classes, and there are arguments to be made for this approach, it allows for more flexibility. At the same time, it does take time to go through the settings file and make these changes in the first place. It's really down to your personal preference.

Finally start the Django dev server, and everything should work as it did before.

However, if you see an error message like this:

```
You have 20 unapplied migration(s). Your project may not work
properly until you apply the migrations for app(s): admin, auth,
blog, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
```

it means that the default database URL value that you specified is not correct and Django has created a new file. Refer to the above examples to make sure you have it correct and can start the dev server without receiving the error before proceeding.

In the next section we'll look at the other of the 12 Factors that we need to consider during development: logging.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish django configurations"
```

- Push to GitHub:

```
git push
```