5.4 How To Implement Secrets File Pattern?

1. Create a secrets file using the configuration format of choice, be it JSON, env, Config,YAML or even XML.
2. Add a secrets loader to manage the secrets in a cohesive,explicit manner.
3. Add the secrets file name to the project's .gitignore file.

Above points are explained with example given below.

First we make a json file.

secrets.json

```
{
"FILENAME": "secrets.json",
"SECRET_KEY": "I've got a secret!",
"DATABASES_HOST": "127.0.0.1",
"PORT": "5432"
}
```

To use the secrets.json file,add the following code to your base settings module.

```
# settings/base.py
import json

# Normally you should not import ANYTHING from Django directly
# into your settings, but ImproperlyConfigured is an exception.
from django.core.exceptions import ImproperlyConfigured

# JSON-based secrets module
with open('secrets.json') as f:
    secrets = json.load(f)

def get_secret(setting, secrets=secrets):
    '''Get the secret variable or return explicit exception.'''
    try:
        return secrets[setting]
    except KeyError:
        error_msg = 'Set the {0} environment
        ↪  variable'.format(setting)
        raise ImproperlyConfigured(error_msg)
```

SECRET_KEY = get_secret('SECRET_KEY')

5.5. **Using Multiple Requirements Files**

Finally, there’s one more thing you need to know about multiple settings files setup. It’s

good practice for each settings file to have its own corresponding requirements file. This

means we're only installing what is required on each server.

To follow this pattern, recommended to us by Jeff Triplett, first create a requirements/ di-

rectory in the <repository_root>. Then create '.txt' files that match the contents of your

settings directory. The results should look something like:

Example 5.21: Segmented Requirements

```
requirements/
├── base.txt
├── local.txt
├── staging.txt
├── production.txt
```

Example 5.22: requirements/base.txt

```
Django==3.2.0
psycopg2-binary==2.8.8
djangorestframework==3.11.0
```

Example 5.23: requirements/local.txt

```
-r base.txt # includes the base.txt requirements file

coverage==5.1
django-debug-toolbar==2.2
```

Example 5.24: requirements/ci.txt

```
-r base.txt # includes the base.txt requirements file

coverage==5.1
```

Example 5.25: requirements/production.txt

```
-r base.txt # includes the base.txt requirements file
```

Example 5.26: Installing Local Requirements

```
pip install -r requirements/local.txt
```

Example 5.27: Installing Production Requirements

```
pip install -r requirements/production.txt
```

5.6. Handling File Paths in Settings

If you switch to the multiple settings setup and get new file path errors to things like tem-

plates and media, don’t be alarmed. This section will help you resolve these errors.

We humbly beseech the reader to never hardcode file paths in Django settings files. This is

really bad:

Example 5.28: Never Hardcode File Paths

```
# settings/base.py

# Configuring MEDIA_ROOT
# DON'T DO THIS! Hardcoded to just one user's preferences
MEDIA_ROOT = '/Users/pydanny/twoscoops_project/media'

# Configuring STATIC_ROOT
# DON'T DO THIS! Hardcoded to just one user's preferences
STATIC_ROOT = '/Users/pydanny/twoscoops_project/collected_static'

# Configuring STATICFILES_DIRS
# DON'T DO THIS! Hardcoded to just one user's preferences
STATICFILES_DIRS = ['/Users/pydanny/twoscoops_project/static']

# Configuring TEMPLATES
# DON'T DO THIS! Hardcoded to just one user's preferences
TEMPLATES = [
    {
        'BACKEND':
            'django.template.backends.django.DjangoTemplates',
        DIRS: ['/Users/pydanny/twoscoops_project/templates',]
    },
]
```

The above code represents a common pitfall called hardcoding. The above code, called a

fixed path, is bad because as far as you know, pydanny (Daniel Feldroy) is the only person

who has set up their computer to match this path structure. Anyone else trying to use this

example will see their project break, forcing them to either change their directory structure

(unlikely) or change the settings module to match their preference (causing problems for

everyone else including pydanny).

Don’t hardcode your paths!

To fix the path issue, we dynamically set a project root variable intuitively named BASE_DIR

at the top of the base settings module. Since BASE_DIR is determined in relation to the

location of base.py, your project can be run from any location on any development computer

or server. We find the cleanest way to set a BASE_DIR - like setting is with Pathlib, part of Python.

Example 5.29: Using Pathlib to discover project root

```
# At the top of settings/base.py
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent.parent.parent
MEDIA_ROOT = BASE_DIR / 'media'
STATIC_ROOT = BASE_DIR / 'static_root'
```

```
STATICFILES_DIRS = [BASE_DIR / 'static']
TEMPLATES = [
    {
        'BACKEND':
        ↪  'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates']
    },
]
```

If you really want to set your BASE_DIR with the Python standard library's os.path library,

though, this is one way to do it in a way that will account for paths:

Example 5.30: Using os.path to discover project root

```
# At the top of settings/base.py
from os.path import abspath, dirname, join

def root(*dirs):
    base_dir = join(dirname(__file__), '..', '..')
    return abspath(join(base_dir, *dirs))

BASE_DIR = root()
MEDIA_ROOT = root('media')
STATIC_ROOT = root('static_root')
STATICFILES_DIRS = [root('static')]
TEMPLATES = [
    {
        'BACKEND':
        ↪  'django.template.backends.django.DjangoTemplates',
        'DIRS': [root('templates')],
    },
]
```

With your various path settings dependent on BASE_DIR , your file path settings should

work, which means your templates and media should be loading without error.

5.7. Summary

Any project that's destined for a real live production server is bound to need multiple set-

tings and requirements files. Even beginners to Django need this kind of settings/require-

ments file setup once their projects are ready to leave the original development machine.

We provide our solution, as well as an Apache-friendly solution since it works well for both

beginning and advanced developers.

Also, if you prefer a different shell than the ones provided, environment variables still work.

You'll just need to know the syntax for defining them.

The same thing applies to requirements files. Working with untracked dependency differ-

ences increases risk as much as untracked settings.