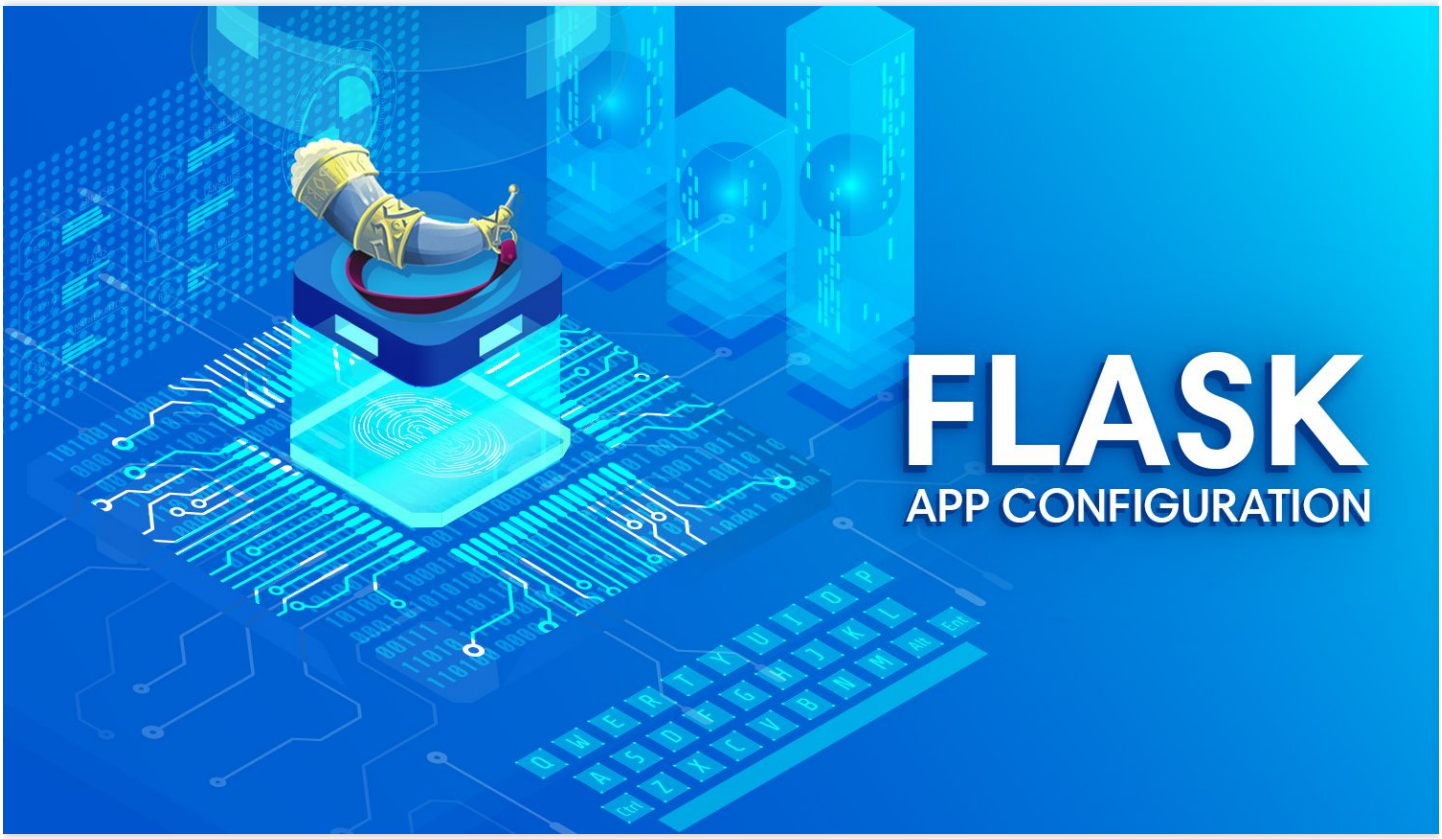
undefined Logo

Configuring Your Flask App

Todd Flask 7 Min Read



The software industry has seen no shortage of new web frameworks over the years, but I certainly wouldn't be alone in saying that none. Discovering Flask is to rediscover instant gratification itself: all it takes is a mere few seconds for anybody to deploy a Python

application, alive and well on a development server. I believe the cliche that the hardest part of anything is getting started, so it's no surprise that Flaks can get the imagination brewing.

Configuring apps isn't the sexiest topic to cover in the realm of hot new frameworks, which is precisely why *most* newcomers skip over this aspect of Flask development *entirely*. This level of short-term procrastination seems harmless to the untrained eye. From my perspective, it's like watching a generation of grown professionals postponing their child's potty-training by one more day: both cases ultimately end with grown adults shitting their pants.

Every web app needs to be configured with things like database URIs and secret keys to enable critical functionality. Unlike Django, there's no monolithic **settings.py** file in Flask *(thank god)*. Instead, we can choose to configure Flask via several methods. Today we're covering two topics in Flask configuration: best practices in configuration structure and configuration settings themselves.

What Not To Do: Inline Configuration

The fastest, dirtiest, and most dangerous method for configuring Flask is by directly setting config vars in source code using plain text. Amongst tutorials, it also seems to be the most common. Stop me if you've seen something like this before:

app.py

```
from flask import Flask

app = Flask(__name__)
app.config['FLASK_ENV'] = 'development'
```

From the perspective of an author writing tutorials for strangers, it's convenient to pretend that setting config values this way is acceptable. Surely nobody who is clicking into *"How To Do This Sensational Thing in Flask"* tutorials care to learn best software development practices, which encourages a lot of bad behavior and leaves many questions unanswered. For instance, why would we wait until *after* we create the Flask **app** object to set something as important as FLASK_ENV? Isn't the point of configuration to inform our app on how to function before it starts... *functioning*?

As the Flask docs promise, your app's config can be modified as we did above and can be modified/changed at any time. Let's set aside the horrendous readability this creates down the line. We'll even turn a blind eye to the unimaginable shit show we create by

mutating configs *while* our app is running. To make matters worse, it encourages committing sensitive values such as secret keys in the middle of your codebase:

app.py

```
from flask import Flask

# Flask app creation
app = Flask(__name__)

# Ugly and confusing tangent of in-line config stuff
app.config['TESTING'] = True
app.config['DEBUG'] = True
app.config['FLASK_ENV'] = 'development'
app.config['SECRET_KEY'] = 'GDtfDCFYjD'
app.config['DEBUG'] = False  # actually I want debug to be off now

# Actual app logic
@app.route('/')
def home():
    return render_template('/index.html')
```

We could update these settings in bulk instead, which is a slightly prettier way of executing the same horrible configuration pattern:

app.py

```
...
# Ugly and confusing tangent of in-line config stuff
app.config.update(
    TESTING=True,
    DEBUG= True,
    FLASK_ENV='development'
    SECRET_KEY='GDtfDCFYjD',
    DEBUG=False,
)
...
```

Config variables belong in *config files* for a reason. They're an organized set of instructions that are defined *before* our app is live. Moreover, it's much easier to avoid compromising sensitive secrets such as AWS or database credentials when we manage these things in one place. Let's explore our better options.

Configuration from a .py File

The simplest way to configure a Flask app is by setting configuration variables directly in a config file such as **config.py**.

app.py

```
from flask import Flask

app = Flask(__name__)
app.config.from_pyfile('config.py')
```

This allows us to avoid the mess in the previous example by isolating our configuration to a file separate from our app logic:

config.py

```
TESTING = True
DEBUG = True
FLASK_ENV = 'development'
SECRET_KEY = 'GDtfDCFYjD'
```

Flask immediately recognizes these variables as being Flask-specific and implements them without any further effort. This is a big win in terms of organization, but we still haven't solved the issue of exposing stuff like **SECRET_KEY** in a file that might make its way onto a public Github repo by accident someday. A standard solution is to pull these values from **environment variables.**

An "environment variable" is a value stored in the system memory of the device running your app. Environment variables can be temporarily created via the terminal like so:

Setting and environment variable.

```
$ export SECRET_KEY='GDtfDCFYjD'
```

`SECRET_KEY` will exist as long as you keep your terminal open. You can retrieve variables like these on your local machine like so:

Setting and environment variable.

```
$ echo $SECRET_KEY
>> 'GDtfDCFYjD'
```

It would be annoying to set these variables every time we open our terminal, so we can set environment variables in a local file called **.env** instead and grab those variables using a Python library like python-dotenv:

config.py

```
"""Flask config."""
from os import environ, path
from dotenv import load_dotenv

basedir = path.abspath(path.dirname(__file__))
load_dotenv(path.join(basedir, '.env'))

TESTING = True
DEBUG = True
FLASK_ENV = 'development'
SECRET_KEY = environ.get('SECRET_KEY')
```

Our app is now both significantly cleaner and *more secure* as long as sensitive values live in **.env**:

.env

```
SECRET_KEY='GDtfDCFYjD'
```

Configuring Flask From Class Objects

We can take our configuration a step further by setting different configuration values depending on which environment we're running in (production, development, etc). We accomplish this by splitting configurations into Python classes in **config.py**:

app.py

```
from flask import Flask

app = Flask(__name__)
app.config.from_object('config.Config')
```

Here's what **config.py** looks like now:

config.py

```
"""Flask config."""
from os import environ, path
from dotenv import load_dotenv

basedir = path.abspath(path.dirname(__file__))
load_dotenv(path.join(basedir, '.env'))


class Config:
    """Set Flask config variables."""

    FLASK_ENV = 'development'
    TESTING = True
    SECRET_KEY = environ.get('SECRET_KEY')
    STATIC_FOLDER = 'static'
    TEMPLATES_FOLDER = 'templates'

    # Database
    SQLALCHEMY_DATABASE_URI = environ.get('SQLALCHEMY_DATABASE_URI')
    SQLALCHEMY_TRACK_MODIFICATIONS = False

    # AWS Secrets
    AWS_SECRET_KEY = environ.get('AWS_SECRET_KEY')
    AWS_KEY_ID = environ.get('AWS_KEY_ID')
```

Let's spice things up:

config.py

```
"""Flask config."""
from os import environ, path
from dotenv import load_dotenv

basedir = path.abspath(path.dirname(__file__))
load_dotenv(path.join(basedir, '.env'))


class Config:
    """Base config."""
    SECRET_KEY = os.environ.get('SECRET_KEY')
    SESSION_COOKIE_NAME = os.environ.get('SESSION_COOKIE_NAME')
    STATIC_FOLDER = 'static'
    TEMPLATES_FOLDER = 'templates'


class ProdConfig(Config):
    FLASK_ENV = 'production'
    DEBUG = False
    TESTING = False
    DATABASE_URI = os.environ.get('PROD_DATABASE_URI')
```

```
class DevConfig(Config):
    FLASK_ENV = 'development'
    DEBUG = True
    TESTING = True
    DATABASE_URI = os.environ.get('DEV_DATABASE_URI')
```

Instead of a one-size-fits-all config, we can now swap configurations based on which environment Flask is running in. **ProdConfig** and **DevConfig** contain values specific to *production* and *development* respectively. Both of these classes extend a base class **Config** which contains values intended to be shared by both. Swapping between configs is now this easy:

app.py

```
# Using a production configuration
app.config.from_object('config.ProdConfig')

# Using a development configuration
app.config.from_object('config.DevConfig')
```

Configuration Variables

Now, what do all these variables mean anyway? We've polished our app to have a beautiful config setup, but what do **SECRET_KEY** or **SESSION_COOKIE_NAME** even mean anyway? We won't go down the rabbit hole of exploring *every* obscure setting in Flask, but I'll give you a crash course on the good parts:

- **FLASK_ENV**: The environment the app is running in, such as *development* or *production*. Setting the environment to *development* mode will automatically trigger other variables, such as setting **DEBUG** to `True`. Flask plugins similarly behave differently when this is true.
- **DEBUG**: Extremely useful when developing! *DEBUG* mode triggers several things. Exceptions thrown by the app will print to console automatically, app crashes will result in a helpful error screen, and your app will auto-reload when changes are detected.
- **TESTING**: This mode ensures exceptions are propagated rather than handled by the app's error handlers, which is useful when running automated tests.
- **SECRET_KEY**: Flask "secret keys" are random strings used to encrypt sensitive user data, such as passwords. Encrypting data in Flask depends on the randomness of this string, which means *decrypting* the same data is as simple as getting a hold of this string's value. Guard your secret key with your life; ideally, even you shouldn't know the value of this variable.

- SERVER_NAME: If you intend your app to be reachable on a custom domain, we specify the app's domain name here.

Static Asset Configuration

Configuration for serving static assets via the Flask-Assets library:

- ASSETS_DEBUG: Enables/disables a debugger specifically for static assets.
- COMPRESSOR_DEBUG: Enables/disables a debugger for asset compressors, such as LESS or SASS.
- FLASK_ASSETS_USE_S3: Boolean value specifying whether or not assets should be served from S3.
- FLASK_ASSETS_USE_CDN: Boolean value specifying whether or not assets should be served from a CDN.

Flask-SQLAlchemy

Flask-SQLAlchemy is pretty much *the* choice for handling database connections in Flask:

- SQLALCHEMY_DATABASE_URI: The connection string of your app's database.
- SQLALCHEMY_ECHO: Prints database-related actions to console for debugging purposes.
- SQLALCHEMY_ENGINE_OPTIONS: Additional options to be passed to the SQLAlchemy engine, which holds your app's database connection.

Flask-Session

Flask-Session is excellent for apps that store user information per session. While Flask supports storing data in cookies by default, Flask-Session builds on this by adding functionality and additional methods for where to store session information:

- SESSION_TYPE: Session information can be handled via Redis, Memcached, filesystem, MongoDB, or SQLAlchemy.
- SESSION_PERMANENT: A True/False value, which states whether or not user sessions should last forever.
- SESSION_KEY_PREFIX: Modifies the key names in session key/value pairs to always have a certain prefix.
- SESSION_REDIS: URI of a Redis instance to store session information.
- SESSION_MEMCACHED: URI of a Memcached client to store session information.

- SESSION_MONGODB: URI of a MongoDB database to store session information.
- SESSION_SQLALCHEMY: URI of a database to store session information.

Last Thoughts

We've spent way more time on the topic of configuring Flask than anybody could enjoy. Instead of making a dry technical post longer, why not learn from example by cruising our collection of countless Flask tutorials? We've pumped out at least a dozen Flask projects - feel free to find an example Flask project that meets your needs and take it from there.

Hackers and Slackers

Software Engineering, Data Science, & Data Engineering tutorials. - Hackers and Slackers

GitHub

Flask Python Software Development

Todd Birchard

118 Posts Website Twitter

Engineer with an ongoing identity crisis. Breaks everything before learning best practices. Completely normal and emotionally stable.