Configuration Handling

Applications need some kind of configuration. There are different settings you might want to change depending on the application environment like toggling the debug mode, setting the secret key, and other such environment-specific things.

The way Flask is designed usually requires the configuration to be available when the application starts up. You can hard code the configuration in the code, which for many small applications is not actually that bad, but there are better ways.

Independent of how you load your config, there is a config object available which holds the loaded configuration values: The **`config`** attribute of the **`Flask`** object. This is the place where Flask itself puts certain configuration values and also where extensions can put their configuration values. But this is also where you can have your own configuration.

Configuration Basics

The **`config`** is actually a subclass of a dictionary and can be modified just like any dictionary:

```
app = Flask(__name__)
app.config['TESTING'] = True
```

Certain configuration values are also forwarded to the **`Flask`** object so you can read and write them from there:

```
app.testing = True
```

To update multiple keys at once you can use the **`dict.update()`** method:

```
app.config.update(
    TESTING=True,
    SECRET_KEY=b'_5#y2L"F4Q8z\n\xec]/'
)
```

Environment and Debug Features

The **`ENV`** and **`DEBUG`** config values are special because they may behave inconsistently if changed after the app has begun setting up. In order to set the environment and debug mode reliably, Flask uses environment variables.

The environment is used to indicate to Flask, extensions, and other programs, like Sentry, what context Flask is running in. It is controlled with the **FLASK_ENV** environment variable and defaults to `production`.

Setting **FLASK_ENV** to `development` will enable debug mode. `flask run` will use the interactive debugger and reloader by default in debug mode. To control this separately from the environment, use the **FLASK_DEBUG** flag.

▶ *Changelog*

To switch Flask to the development environment and enable debug mode, set **FLASK_ENV**:

```
$ export FLASK_ENV=development
$ flask run
```

(On Windows, use `set` instead of `export`.)

Using the environment variables as described above is recommended. While it is possible to set **ENV** and **DEBUG** in your config or code, this is strongly discouraged. They can't be read early by the `flask` command, and some systems or extensions may have already configured themselves based on a previous value.

Builtin Configuration Values

The following configuration values are used internally by Flask:

ENV

What environment the app is running in. Flask and extensions may enable behaviors based on the environment, such as enabling debug mode. The **env** attribute maps to this config key. This is set by the **FLASK_ENV** environment variable and may not behave as expected if set in code.

Do not enable development when deploying in production.

Default: `'production'`

▶ *Changelog*

DEBUG

Whether debug mode is enabled. When using `flask run` to start the development server, an interactive debugger will be shown for unhandled exceptions, and the server will be reloaded when code changes. The **debug** attribute maps to this config key. This is enabled when **ENV** is `'development'` and is overridden by the `FLASK_DEBUG` environment variable. It may not behave as expected if set in code.

Do not enable debug mode when deploying in production.

Default: `True` if **`ENV`** is `'development'`, or `False` otherwise.

TESTING

Enable testing mode. Exceptions are propagated rather than handled by the the app's error handlers. Extensions may also change their behavior to facilitate easier testing. You should enable this in your own tests.

Default: `False`

PROPAGATE_EXCEPTIONS

Exceptions are re-raised rather than being handled by the app's error handlers. If not set, this is implicitly true if `TESTING` or `DEBUG` is enabled.

Default: `None`

PRESERVE_CONTEXT_ON_EXCEPTION

Don't pop the request context when an exception occurs. If not set, this is true if `DEBUG` is true. This allows debuggers to introspect the request data on errors, and should normally not need to be set directly.

Default: `None`

TRAP_HTTP_EXCEPTIONS

If there is no handler for an `HTTPException`-type exception, re-raise it to be handled by the interactive debugger instead of returning it as a simple error response.

Default: `False`

TRAP_BAD_REQUEST_ERRORS

Trying to access a key that doesn't exist from request dicts like `args` and `form` will return a 400 Bad Request error page. Enable this to treat the error as an unhandled exception instead so that you get the interactive debugger. This is a more specific version of `TRAP_HTTP_EXCEPTIONS`. If unset, it is enabled in debug mode.

Default: `None`

SECRET_KEY

A secret key that will be used for securely signing the session cookie and can be used for any other security related needs by extensions or your application. It should be a long random string of bytes, although unicode is accepted too. For example, copy the output of this to your config:

```
$ python -c 'import os; print(os.urandom(16))'
b'_5#y2L"F4Q8z\n\xec]/'
```

Do not reveal the secret key when posting questions or committing code.

Default: `None`

SESSION_COOKIE_NAME

The name of the session cookie. Can be changed in case you already have a cookie with the same name.

Default: `'session'`

SESSION_COOKIE_DOMAIN

The domain match rule that the session cookie will be valid for. If not set, the cookie will be valid for all subdomains of **SERVER_NAME**. If `False`, the cookie's domain will not be set.

Default: `None`

SESSION_COOKIE_PATH

The path that the session cookie will be valid for. If not set, the cookie will be valid underneath `APPLICATION_ROOT` or `/` if that is not set.

Default: `None`

SESSION_COOKIE_HTTPONLY

Browsers will not allow JavaScript access to cookies marked as "HTTP only" for security.

Default: `True`

SESSION_COOKIE_SECURE

Browsers will only send cookies with requests over HTTPS if the cookie is marked "secure". The application must be served over HTTPS for this to make sense.

Default: `False`

SESSION_COOKIE_SAMESITE

Restrict how cookies are sent with requests from external sites. Can be set to `'Lax'` (recommended) or `'Strict'`. See Set-Cookie options.

Default: `None`

▶ *Changelog*

PERMANENT_SESSION_LIFETIME

If `session.permanent` is true, the cookie's expiration will be set this number of seconds in the future. Can either be a **`datetime.timedelta`** or an `int`.

Flask's default cookie implementation validates that the cryptographic signature is not older than this value.

Default: `timedelta(days=31)` (`2678400` seconds)

SESSION_REFRESH_EACH_REQUEST

Control whether the cookie is sent with every response when `session.permanent` is true. Sending the cookie every time (the default) can more reliably keep the session from expiring, but uses more bandwidth. Non-permanent sessions are not affected.

Default: `True`

USE_X_SENDFILE

When serving files, set the `X-Sendfile` header instead of serving the data with Flask. Some web servers, such as Apache, recognize this and serve the data more efficiently. This only makes sense when using such a server.

Default: `False`

SEND_FILE_MAX_AGE_DEFAULT

When serving files, set the cache control max age to this number of seconds. Can either be a **`datetime.timedelta`** or an `int`. Override this value on a per-file basis using **`get_send_file_max_age()`** on the application or blueprint.

Default: `timedelta(hours=12)` (`43200` seconds)

SERVER_NAME

Inform the application what host and port it is bound to. Required for subdomain route matching support.

If set, will be used for the session cookie domain if **`SESSION_COOKIE_DOMAIN`** is not set. Modern web browsers will not allow setting cookies for domains without a dot. To use a domain locally, add any names that should route to the app to your `hosts` file.

```
127.0.0.1 localhost.dev
```

If set, `url_for` can generate external URLs with only an application context instead of a request context.

Default: `None`

APPLICATION_ROOT

Inform the application what path it is mounted under by the application / web server. This is used for generating URLs outside the context of a request (inside a request, the dispatcher is responsible for setting `SCRIPT_NAME` instead; see Application Dispatching for examples of dispatch configuration).

Will be used for the session cookie path if `SESSION_COOKIE_PATH` is not set.

Default: `'/'`

PREFERRED_URL_SCHEME

Use this scheme for generating external URLs when not in a request context.

Default: `'http'`

MAX_CONTENT_LENGTH

Don't read more than this many bytes from the incoming request data. If not set and the request does not specify a `CONTENT_LENGTH`, no data will be read for security.

Default: `None`

JSON_AS_ASCII

Serialize objects to ASCII-encoded JSON. If this is disabled, the JSON will be returned as a Unicode string, or encoded as `UTF-8` by `jsonify`. This has security implications when rendering the JSON into JavaScript in templates, and should typically remain enabled.

Default: `True`

JSON_SORT_KEYS

Sort the keys of JSON objects alphabetically. This is useful for caching because it ensures the data is serialized the same way no matter what Python's hash seed is. While not recommended, you can disable this for a possible performance improvement at the cost of caching.

Default: `True`

JSONIFY_PRETTYPRINT_REGULAR

`jsonify` responses will be output with newlines, spaces, and indentation for easier reading by humans. Always enabled in debug mode.

Default: `False`

JSONIFY_MIMETYPE

The mimetype of `jsonify` responses.

Default: `'application/json'`

TEMPLATES_AUTO_RELOAD

Reload templates when they are changed. If not set, it will be enabled in debug mode.

Default: `None`

EXPLAIN_TEMPLATE_LOADING

Log debugging information tracing how a template file was loaded. This can be useful to figure out why a template was not loaded or the wrong file appears to be loaded.

Default: `False`

MAX_COOKIE_SIZE

Warn if cookie headers are larger than this many bytes. Defaults to `4093`. Larger cookies may be silently ignored by browsers. Set to `0` to disable the warning.

▶ *Changelog*

Configuring from Files

Configuration becomes more useful if you can store it in a separate file, ideally located outside the actual application package. This makes packaging and distributing your application possible via various package handling tools (Deploying with Setuptools) and finally modifying the configuration file afterwards.

So a common pattern is this:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_settings')
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

This first loads the configuration from the *yourapplication.default_settings* module and then overrides the values with the contents of the file the **YOURAPPLICATION_SETTINGS** environment variable points to. This environment variable can be set on Linux or OS X with the export command in the shell before starting the server:

```
$ export YOURAPPLICATION_SETTINGS=/path/to/settings.cfg
$ python run-app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader...
```

On Windows systems use the *set* builtin instead:

```
> set YOURAPPLICATION_SETTINGS=\path\to\settings.cfg
```

The configuration files themselves are actual Python files. Only values in uppercase are actually stored in the config object later on. So make sure to use uppercase letters for your config keys.

Here is an example of a configuration file:

```
# Example configuration
DEBUG = False
SECRET_KEY = b'_5#y2L"F4Q8z\n\xec]/'
```

Make sure to load the configuration very early on, so that extensions have the ability to access the configuration when starting up. There are other methods on the config object as well to load from individual files. For a complete reference, read the **`Config`** object's documentation.

Configuring from Environment Variables

In addition to pointing to configuration files using environment variables, you may find it useful (or necessary) to control your configuration values directly from the environment.

Environment variables can be set on Linux or OS X with the export command in the shell before starting the server:

```
$ export SECRET_KEY='5f352379324c22463451387a0aec5d2f'
$ export MAIL_ENABLED=false
$ python run-app.py
 * Running on http://127.0.0.1:5000/
```

On Windows systems use the `set` builtin instead:

```
> set SECRET_KEY='5f352379324c22463451387a0aec5d2f'
```

While this approach is straightforward to use, it is important to remember that environment variables are strings – they are not automatically deserialized into Python types.

Here is an example of a configuration file that uses environment variables:

```
import os

_mail_enabled = os.environ.get("MAIL_ENABLED", default="true")
MAIL_ENABLED = _mail_enabled.lower() in {"1", "t", "true"}

SECRET_KEY = os.environ.get("SECRET_KEY")

if not SECRET_KEY:
    raise ValueError("No SECRET_KEY set for Flask application")
```

Notice that any value besides an empty string will be interpreted as a boolean `True` value in Python, which requires care if an environment explicitly sets values intended to be `False`.

Make sure to load the configuration very early on, so that extensions have the ability to access the configuration when starting up. There are other methods on the config object as well to load from individual files. For a complete reference, read the **`Config`** class documentation.

Configuration Best Practices

The downside with the approach mentioned earlier is that it makes testing a little harder. There is no single 100% solution for this problem in general, but there are a couple of things you can keep in mind to improve that experience:

1. Create your application in a function and register blueprints on it. That way you can create multiple instances of your application with different configurations attached which makes unit testing a lot easier. You can use this to pass in configuration as needed.
2. Do not write code that needs the configuration at import time. If you limit yourself to request-only accesses to the configuration you can reconfigure the object later on as needed.

Development / Production

Most applications need more than one configuration. There should be at least separate configurations for the production server and the one used during development. The easiest way to handle this is to use a default configuration that is always loaded and part of the version control, and a separate configuration that overrides the values as necessary as mentioned in the example above:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_settings')
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

Then you just have to add a separate `config.py` file and export `YOURAPPLICATION_SETTINGS=/path/to/config.py` and you are done. However there are alternative ways as well. For example you could use imports or subclassing.

What is very popular in the Django world is to make the import explicit in the config file by adding `from yourapplication.default_settings import *` to the top of the file and then overriding the changes by hand. You could also inspect an environment variable like `YOURAPPLICATION_MODE` and set that to *production*, *development* etc and import different hard-coded files based on that.

An interesting pattern is also to use classes and inheritance for configuration:

```
class Config(object):
    DEBUG = False
    TESTING = False
    DATABASE_URI = 'sqlite:///:memory:'

class ProductionConfig(Config):
    DATABASE_URI = 'mysql://user@localhost/foo'

class DevelopmentConfig(Config):
    DEBUG = True

class TestingConfig(Config):
    TESTING = True
```

To enable such a config you just have to call into **`from_object()`**:

```
app.config.from_object('configmodule.ProductionConfig')
```

Note that **`from_object()`** does not instantiate the class object. If you need to instantiate the class, such as to access a property, then you must do so before calling **`from_object()`**:

```
from configmodule import ProductionConfig
app.config.from_object(ProductionConfig())

# Alternatively, import via string:
from werkzeug.utils import import_string
cfg = import_string('configmodule.ProductionConfig')()
app.config.from_object(cfg)
```

Instantiating the configuration object allows you to use `@property` in your configuration classes:

```
class Config(object):
    """Base config, uses staging database server."""
    DEBUG = False
    TESTING = False
    DB_SERVER = '192.168.1.56'

    @property
    def DATABASE_URI(self):         # Note: all caps
        return 'mysql://user@{}/foo'.format(self.DB_SERVER)

class ProductionConfig(Config):
    """Uses production database server."""
    DB_SERVER = '192.168.19.32'

class DevelopmentConfig(Config):
    DB_SERVER = 'localhost'
    DEBUG = True

class TestingConfig(Config):
    DB_SERVER = 'localhost'
    DEBUG = True
    DATABASE_URI = 'sqlite:///:memory:'
```

There are many different ways and it's up to you how you want to manage your configuration files. However here a list of good recommendations:

- Keep a default configuration in version control. Either populate the config with this default configuration or import it in your own configuration files before overriding values.
- Use an environment variable to switch between the configurations. This can be done from outside the Python interpreter and makes development and deployment much easier because you can quickly and easily switch between different configs without having to touch the code at all. If you are working often on different projects you can even create your own script for sourcing that activates a virtualenv and exports the development configuration for you.
- Use a tool like fabric in production to push code and configurations separately to the production server(s). For some details about how to do that, head over to the Deploying with Fabric pattern.

Instance Folders

▶ *Changelog*

Flask 0.8 introduces instance folders. Flask for a long time made it possible to refer to paths relative to the application's folder directly (via **`Flask.root_path`**). This was also how many developers loaded configurations stored next to the application. Unfortunately however this only works well if applications are not packages in which case the root path refers to the contents of the package.

With Flask 0.8 a new attribute was introduced: **`Flask.instance_path`**. It refers to a new concept called the "instance folder". The instance folder is designed to not be under version control and be deployment specific. It's the perfect place to drop things that either change at runtime or configuration files.

You can either explicitly provide the path of the instance folder when creating the Flask application or you can let Flask autodetect the instance folder. For explicit configuration use the *instance_path* parameter:

```
app = Flask(__name__, instance_path='/path/to/instance/folder')
```

Please keep in mind that this path *must* be absolute when provided.

If the *instance_path* parameter is not provided the following default locations are used:

- Uninstalled module:

  ```
  /myapp.py
  /instance
  ```

- Uninstalled package:

  ```
  /myapp
      /__init__.py
  /instance
  ```

- Installed module or package:

  ```
  $PREFIX/lib/python2.X/site-packages/myapp
  $PREFIX/var/myapp-instance
  ```

  `$PREFIX` is the prefix of your Python installation. This can be `/usr` or the path to your virtualenv. You can print the value of `sys.prefix` to see what the prefix is set to.

Since the config object provided loading of configuration files from relative filenames we made it possible to change the loading via filenames to be relative to the instance path if wanted. The behavior of relative paths in config files can be flipped between "relative to the

application root" (the default) to "relative to instance folder" via the *instance_relative_config* switch to the application constructor:

```
app = Flask(__name__, instance_relative_config=True)
```

Here is a full example of how to configure Flask to preload the config from a module and then override the config from a file in the instance folder if it exists:

```
app = Flask(__name__, instance_relative_config=True)
app.config.from_object('yourapplication.default_settings')
app.config.from_pyfile('application.cfg', silent=True)
```

The path to the instance folder can be found via the **Flask.instance_path**. Flask also provides a shortcut to open a file from the instance folder with **Flask.open_instance_resource()**.

Example usage for both:

```
filename = os.path.join(app.instance_path, 'application.cfg')
with open(filename) as f:
    config = f.read()

# or via open_instance_resource:
with app.open_instance_resource('application.cfg') as f:
    config = f.read()
```