The Flask Mega-Tutorial Part XV: A Better Application Structure (/post/the-flask-mega-tutorial-part-xv-a-better-application-structure)

March 13 2018

Posted by **Miguel Grinberg (/author/Miguel Grinberg)** under **Python (/category/Python)**, **Flask (/category/Flask)**, **Programming (/category/Programming)**.

Tweet Like Share

This is the fifteenth installment of the Flask Mega-Tutorial series, in which I'm going to restructure the application using a style that is appropriate for larger applications.

For your reference, below is a list of the articles in this series.

Note 1: If you are looking for the legacy version of this tutorial, it's here (/post/the-flask-mega-tutorial-part-i-hello-world-legacy).

Note 2: If you would like to support my work on this blog, or just don't have patience to wait for weekly articles, I am offering the complete version of this tutorial packaged as an ebook or a set of videos. For more information, visit courses.miguelgrinberg.com (https://courses.miguelgrinberg.com).

Microblog is already an application of a decent size, so I thought this is a good opportunity to discuss how a Flask application can grow without becoming messy or too difficult to manage. Flask is a framework that is designed to give you the option to organize your project in any way you want, and as part of that philosophy, it makes it possible to change or adapt the structure of the application as it becomes larger, or as your needs or level of experience change.

In this chapter I'm going to discuss some patterns that apply to large applications, and to demonstrate them I'm going to make some changes to the way my Microblog project is structured, with the goal of making the code more maintainable and better organized. But of course, in true Flask spirit, I encourage you to take these changes just as a recommendation when trying to decide on a way to organize your own projects.

The GitHub links for this chapter are: Browse (https://github.com/miguelgrinberg/microblog/tree/v0.15), Zip (https://github.com/miguelgrinberg/microblog/archive/v0.15.zip), Diff (https://github.com/miguelgrinberg/microblog/compare/v0.14...v0.15).

Current Limitations

There are two basic problems with the application in its current state. If you look at how the application is structured, you are going to notice that there are a few different subsystems that can be identified, but the code that supports them is all intermixed, without any clear boundaries. Let's review what those subsystems are:

- The user authentication subsystem, which includes some view functions in *app/routes.py*, some forms in *app/forms.py*, some templates in *app/templates* and the email support in *app/email.py*.
- The error subsystem, which defines error handlers in *app/errors.py* and templates in *app/templates*.
- The core application functionality, which includes displaying and writing blog posts, user profiles and following, and live translations of blog posts, which is spread through most of the application modules and templates.

Thinking about these three subsystems that I have identified and how they are structured, you can probably notice a pattern. So far, the organization logic that I've been following is based on having modules dedicated to different application functions. There is a module for view functions, another one for web forms, one for errors, one for emails, a directory for HTML templates, and so on. While this is a structure that makes sense for small projects, once a project starts to grow, it tends to make some of these modules really large and messy.

One way to clearly see the problem is to consider how you would start a second project by reusing as much as you can from this one. For example, the user authentication portion should work well in other applications, but if you wanted to use that code as it is, you would have to go into several modules and copy/paste the pertinent sections into new files in the new project. See how inconvenient that is? Wouldn't it be better if this project had all the authentication related files separated from the rest of the application? The *blueprints* feature of Flask helps achieve a more practical organization that makes it easier to reuse code.

There is a second problem that is not that evident. The Flask application instance is created as a global variable in *app/__init__.py*, and then imported by a lot of application modules. While this in itself is not a problem, having the application as a global variable can complicate certain scenarios, in particular those related to testing. Imagine you want to test this application under different configurations. Because the application is defined as a global variable, there is really no way to instantiate two applications that use different configuration variables. Another situation that is not ideal is that all the tests use the same application, so a test could be making changes to the application that affect another test that runs later. Ideally you want all tests to run on a pristine application instance.

You can actually see in the *tests.py* module that I'm resorting to the trick of modifying the configuration after it was set in the application to direct the tests to use an in-memory database instead of the default SQLite database based on disk. I really have no other way to change the configured database, because by the time the tests start the application has been created and configured. For this particular situation, changing the configuration after it was applied to the application appears to work fine, but in other cases it may not, and in any case, it is a bad practice that can lead to obscure and difficult to find bugs.

A better solution would be to not use a global variable for the application, and instead use an *application factory* function to create the function at runtime. This would be a function that accepts a configuration object as an argument, and returns a Flask application instance, configured with those settings. If I could modify the application to work with an application factory function, then writing tests that require special configuration would become easy, because each test can create its own application.

In this chapter I'm going to refactor the application to introduce blueprints for the three subsystems I have identified above, and an application factory function. Showing you the detailed list of changes is going to be impractical, because there are little changes in pretty much every file that is part of the application, so I'm going to discuss the steps that I took to do the refactoring, and you can then download (https://github.com/miguelgrinberg/microblog/archive/v0.15.zip) the application with these changes made.

Blueprints

In Flask, a blueprint is a logical structure that represents a subset of the application. A blueprint can include elements such as routes, view functions, forms, templates and static files. If you write your blueprint in a separate Python package, then you have a component that encapsulates the elements related to specific feature of the application.

The contents of a blueprint are initially in a dormant state. To associate these elements, the blueprint needs to be registered with the application. During the registration, all the elements that were added to the blueprint are passed on to the application. So you can think of a blueprint as a temporary storage for application functionality that helps in organizing your code.

Error Handling Blueprint

The first blueprint that I created was one that encapsulates the support for error handlers. The structure of this blueprint is as follows:

```
app/
    errors/                             <-- blueprint package
        __init__.py                     <-- blueprint creation
        handlers.py                     <-- error handlers
    templates/
        errors/                         <-- error templates
            404.html
            500.html
    __init__.py                         <-- blueprint registration
```

In essence, what I did is move the *app/errors.py* module into *app/errors/handlers.py* and the two error templates into *app/templates/errors*, so that they are separated from the other templates. I also had to change the `render_template()` calls in both error handlers to use the new *errors* template sub-directory. After that I added the blueprint creation to the *app/errors/__init__.py* module, and the blueprint registration to *app/__init__.py*, after the application instance is created.

I should note that Flask blueprints can be configured to have a separate directory for templates or static files. I have decided to move the templates into a sub-directory of the application's template directory so that all templates are in a single hierarchy, but if you prefer to have the templates that belong to a blueprint inside the blueprint package, that is supported. For example, if you add a `template_folder='templates'` argument to the `Blueprint()` constructor, you can then store the blueprint's templates in *app/errors/templates*.

The creation of a blueprint is fairly similar to the creation of an application. This is done in the ___init__.py module of the blueprint package:

app/errors/__init__.py: Errors blueprint.

```
from flask import Blueprint

bp = Blueprint('errors', __name__)

from app.errors import handlers
```

The `Blueprint` class takes the name of the blueprint, the name of the base module (typically set to `__name__` like in the Flask application instance), and a few optional arguments, which in this case I do not need. After the blueprint object is created, I import the *handlers.py* module, so that the error handlers in it are registered with the blueprint. This import is at the bottom to avoid circular dependencies.

In the *handlers.py* module, instead of attaching the error handlers to the application with the `@app.errorhandler` decorator, I use the blueprint's `@bp.app_errorhandler` decorator. While both decorators achieve the same end result, the idea is to try to make the blueprint independent of the application so that it is more portable. I also need to modify the path to the two error templates to account for the new *errors* sub-directory where they were moved.

The final step to complete the refactoring of the error handlers is to register the blueprint with the application:

app/__init__.py: Register the errors blueprint with the application.

```
app = Flask(__name__)

# ...

from app.errors import bp as errors_bp
app.register_blueprint(errors_bp)

# ...

from app import routes, models  # <-- remove errors from this import!
```

To register a blueprint, the `register_blueprint()` method of the Flask application instance is used. When a blueprint is registered, any view functions, templates, static files, error handlers, etc. are connected to the application. I put the import of the blueprint right above the `app.register_blueprint()` to avoid circular dependencies.

Authentication Blueprint

The process to refactor the authentication functions of the application into a blueprint is fairly similar to that of the error handlers. Here is a diagram of the refactored blueprint:

```
app/
    auth/                               <-- blueprint package
        __init__.py                     <-- blueprint creation
        email.py                        <-- authentication emails
        forms.py                        <-- authentication forms
        routes.py                       <-- authentication routes
    templates/
        auth/                           <-- blueprint templates
            login.html
            register.html
            reset_password_request.html
            reset_password.html
    __init__.py                         <-- blueprint registration
```

To create this blueprint I had to move all the authentication related functionality to new modules I created in the blueprint. This includes a few view functions, web forms, and support functions such as the one that sends password reset tokens by email. I also moved the templates into a sub-directory to separate them from the rest of the application, like I did with the error pages.

When defining routes in a blueprint, the `@bp.route` decorate is used instead of `@app.route`. There is also a required change in the syntax used in the `url_for()` to build URLs. For regular view functions attached directly to the application, the first argument to `url_for()` is the view function name. When a route is defined in a blueprint, this argument must include the blueprint name and the view function name, separated by a period. So for example, I had to replace all occurrences of `url_for('login')` with `url_for('auth.login')`, and same for the remaining view functions.

To register the `auth` blueprint with the application, I used a slightly different format:

app/__init__.py: Register the authentication blueprint with the application.

```
# ...
from app.auth import bp as auth_bp
app.register_blueprint(auth_bp, url_prefix='/auth')
# ...
```

The `register_blueprint()` call in this case has an extra argument, `url_prefix`. This is entirely optional, but Flask gives you the option to attach a blueprint under a URL prefix, so any routes defined in the blueprint get this prefix in their URLs. In many cases this is useful as a sort of "namespacing" that keeps all the routes in the blueprint separated from other routes in the application or other blueprints. For authentication, I thought it was nice to have all the routes starting with */auth*, so I added the prefix. So now the login URL is going to be *http://localhost:5000/auth/login*. Because I'm using `url_for()` to generate the URLs, all URLs will automatically incorporate the prefix.

Main Application Blueprint

The third blueprint contains the core application logic. Refactoring this blueprint requires the same process that I used with the previous two blueprints. I gave this blueprint the name `main`, so all `url_for()` calls that referenced view functions had to get a `main.` prefix. Given that this is the core functionality of the application, I decided to leave the templates in the same locations. This is not a problem because I have moved the templates from the other two blueprints into sub-directories.

The Application Factory Pattern

As I mentioned in the introduction to this chapter, having the application as a global variable introduces some complications, mainly in the form of limitations for some testing scenarios. Before I introduced blueprints, the application had to be a global variable, because all the view functions and error handlers needed to be decorated with decorators that come from `app`, such as `@app.route`. But now that all routes and error handlers were moved to blueprints, there are a lot less reasons to keep the application global.

So what I'm going to do, is add a function called `create_app()` that constructs a Flask application instance, and eliminate the global variable. The transformation was not trivial, I had to sort out a few complications, but let's first look at the application factory function:

app/__init__.py: Application factory function.

```
# ...
db = SQLAlchemy()
migrate = Migrate()
login = LoginManager()
login.login_view = 'auth.login'
login.login_message = _l('Please log in to access this page.')
mail = Mail()
bootstrap = Bootstrap()
moment = Moment()
babel = Babel()

def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    db.init_app(app)
    migrate.init_app(app, db)
    login.init_app(app)
    mail.init_app(app)
    bootstrap.init_app(app)
    moment.init_app(app)
    babel.init_app(app)

    # ... no changes to blueprint registration

    if not app.debug and not app.testing:
        # ... no changes to logging setup

    return app
```

You have seen that most Flask extensions are initialized by creating an instance of the extension and passing the application as an argument. When the application does not exist as a global variable, there is an alternative mode in which extensions are initialized in two phases. The extension instance is first created in the global scope as before, but no arguments are passed to it. This creates an instance of the extension that is not attached to the application. At the time the application instance is created in the factory function, the `init_app()` method must be invoked on the extension instances to bind it to the now known application.

Other tasks performed during initialization remain the same, but are moved to the factory function instead of being in the global scope. This includes the registration of blueprints and logging configuration. Note that I have added a `not app.testing` clause to the conditional that decides if email and file logging should be enabled or not, so that all this logging is skipped during unit tests. The `app.testing` flag is going to be `True` when running unit tests, due to the `TESTING` variable being set to `True` in the configuration.

So who calls the application factory function? The obvious place to use this function is the top-level *microblog.py* script, which is the only module in which the application now exists in the global scope. The other place is in *tests.py*, and I will discuss unit testing in more detail in the next section.

As I mentioned above, most references to `app` went away with the introduction of blueprints, but there were some still in the code that I had to address. For example, the *app/models.py*, *app/translate.py*, and *app/main/routes.py* modules all had references to `app.config`. Fortunately, the Flask developers tried to make it easy for view functions to access the application instance without having to import it like I have been doing until now. The `current_app` variable that Flask provides is a special "context" variable that Flask initializes with the application before it dispatches a request. You have already seen another context variable before, the `g` variable in which I'm storing the current locale. These two, along with Flask-Login's `current_user` and a few others you haven't seen yet, are somewhat "magical" variables, in that they work like global variables, but are only accessible during the handling of a request, and only in the thread that is handling it.

Replacing `app` with Flask's `current_app` variable eliminates the need of importing the application instance as a global variable. I was able to change all references to `app.config` with `current_app.config` without any difficulty through simple search and replace.

The *app/email.py* module presented a slightly bigger challenge, so I had to use a small trick:

app/email.py: Pass application instance to another thread.

```
from flask import current_app

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    Thread(target=send_async_email,
           args=(current_app._get_current_object(), msg)).start()
```

In the `send_email()` function, the application instance is passed as an argument to a background thread that will then deliver the email without blocking the main application. Using `current_app` directly in the `send_async_email()` function that runs as a background thread would not have worked, because `current_app` is a context-aware variable that is tied to the thread that is handling the client request. In a different thread, `current_app` would not have a value assigned. Passing `current_app` directly as an argument to the thread object would not have worked either, because `current_app` is really a *proxy object* that is dynamically mapped to the application instance. So passing the proxy object would be the same as using `current_app` directly in the thread. What I needed to do is access the real application instance that is stored inside the proxy object, and pass that as the `app` argument. The `current_app._get_current_object()` expression extracts the actual application instance from inside the proxy object, so that is what I passed to the thread as an argument.

Another module that was tricky was *app/cli.py*, which implements a few shortcut commands for managing language translations. The `current_app` variable does not work in this case because these commands are registered at start up, not during the handling of a request, which is the only time when `current_app` can be used. To remove the reference to `app` in this module, I resorted to another trick, which is to move these custom commands inside a `register()` function that takes the `app` instance as an argument:

app/cli.py: Register custom application commands.

```
import os
import click

def register(app):
    @app.cli.group()
    def translate():
        """Translation and localization commands."""
        pass

    @translate.command()
    @click.argument('lang')
    def init(lang):
        """Initialize a new language."""
        # ...

    @translate.command()
    def update():
        """Update all languages."""
        # ...

    @translate.command()
    def compile():
        """Compile all languages."""
        # ...
```

Then I called this `register()` function from *microblog.py*. Here is the complete *microblog.py* after all the refactoring:

microblog.py: Main application module refactored.

```
from app import create_app, db, cli
from app.models import User, Post

app = create_app()
cli.register(app)

@app.shell_context_processor
def make_shell_context():
    return {'db': db, 'User': User, 'Post' :Post}
```

Unit Testing Improvements

As I hinted in the beginning of this chapter, a lot of the work that I did so far had the goal of improving the unit testing workflow. When you are running unit tests you want to make sure the application is configured in a way that it does not interfere with your development resources, such as your database.

The current version of *tests.py* resorts to the trick of modifying the configuration after it was applied to the application instance, which is a dangerous practice as not all types of changes will work when done that late. What I want is to have a chance to specify my testing configuration before it gets added to the application.

The `create_app()` function now accepts a configuration class as an argument. By default, the `Config` class defined in *config.py* is used, but I can now create an application instance that uses different configuration simply by passing a new class to the factory function. Here is an example configuration class that would be suitable to use for my unit tests:

tests.py: Testing configuration.

```
from config import Config

class TestConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = 'sqlite://'
```

What I'm doing here is creating a subclass of the application's `Config` class, and overriding the SQLAlchemy configuration to use an in-memory SQLite database. I also added a `TESTING` attribute set to `True`, which I currently do not need, but could be useful if the application needs to determine if it is running under unit tests or not.

If you recall, my unit tests relied on the `setUp()` and `tearDown()` methods, invoked automatically by the unit testing framework to create and destroy an environment that is appropriate for each test to run. I can now use these two methods to create and destroy a brand new application for each test:

tests.py: Create an application for each test.

```
class UserModelCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app(TestConfig)
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

    def tearDown(self):
        db.session.remove()
        db.drop_all()
        self.app_context.pop()
```

The new application will be stored in `self.app`, but creating an application isn't enough to make everything work. Consider the `db.create_all()` statement that creates the database tables. The `db` instance needs to know what the application instance is, because it needs to get the database URI from `app.config`, but when you are working with an application factory you are not really limited to a single application, there could be more than one created. So how does `db` know to use the `self.app` instance that I just created?

The answer is in the *application context*. Remember the `current_app` variable, which somehow acts as a proxy for the application when there is no global application to import? This variable looks for an active application context in the current thread, and if it finds one, it gets the application from it. If there is no context, then there is no way to know what application is active, so `current_app` raises an exception. Below you can see how this works in a Python console. This needs to be a console started by running `python`, because the `flask shell` command automatically activates an application context for convenience.

```
>>> from flask import current_app
>>> current_app.config['SQLALCHEMY_DATABASE_URI']
Traceback (most recent call last):
    ...
RuntimeError: Working outside of application context.

>>> from app import create_app
>>> app = create_app()
>>> app.app_context().push()
>>> current_app.config['SQLALCHEMY_DATABASE_URI']
'sqlite:////home/miguel/microblog/app.db'
```

So that's the secret! Before invoking your view functions, Flask pushes an application context, which brings `current_app` and `g` to life. When the request is complete, the context is removed, along with these variables. For the `db.create_all()` call to work in the unit testing `setUp()` method, I pushed an application context for the application instance I just created, and in that way, `db.create_all()` can use `current_app.config` to know where is the database. Then in the `tearDown()` method I pop the context to reset everything to a clean state.

You should also know that the application context is one of two contexts that Flask uses. There is also a *request context*, which is more specific, as it applies to a request. When a request context is activated right before a request is handled, Flask's `request` and `session` variables become available, as well as Flask-Login's `current_user` .

Environment Variables

As you have seen as I built this application, there are a number of configuration options that depend on having variables set up in your environment before you start the server. This includes your secret key, email server information, database URL, and Microsoft Translator API key. You'll probably agree with me that this is inconvenient, because each time you open a new terminal session those variables need to be set again.

A common pattern for applications that depend on lots of environment variables is to store these in a *.env* file in the root application directory. The application imports the variables in this file when it starts, and that way, there is no need to have all these variables manually set by you.

There is a Python package that supports *.env* files, called `python-dotenv` . So let's install that package:

```
(venv) $ pip install python-dotenv
```

Since the *config.py* module is where I read all the environment variables, I'm going to import a *.env* file before the `Config` class is created, so that the variables are set when the class is constructed:

config.py: Import a .env file with environment variables.

```
import os
from dotenv import load_dotenv

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

class Config(object):
    # ...
```

So now you can create a *.env* file with all the environment variables that your application needs. It is important that you do not add your *.env* file to source control. You do not want to have a file that contains passwords and other sensitive information included in your source code repository.

The *.env* file can be used for all the configuration-time variables, but it cannot be used for Flask's `FLASK_APP` and `FLASK_DEBUG` environment variables, because these are needed very early in the application bootstrap process, before the application instance and its configuration object exist.

The following example shows a *.env* file that defines a secret key, configures email to go out on a locally running mail server on port 25 and no authentication, sets up the Microsoft Translator API key, and leaves the database configuration to use the defaults:

```
SECRET_KEY=a-really-long-and-unique-key-that-nobody-knows
MAIL_SERVER=localhost
MAIL_PORT=25
MS_TRANSLATOR_KEY=<your-translator-key-here>
```

Requirements File

At this point I have installed a fair number of packages in the Python virtual environment. If you ever need to regenerate your environment on another machine, you are going to have trouble remembering what packages you had to install, so the generally accepted practice is to write a *requirements.txt* file in the root folder of your project listing all the dependencies, along with their versions. Producing this list is actually easy:

```
(venv) $ pip freeze > requirements.txt
```

The `pip freeze` command will dump all the packages that are installed on your virtual environment in the correct format for the *requirements.txt* file. Now, if you need to create the same virtual environment on another machine, instead of installing packages one by one, you can run:

```
(venv) $ pip install -r requirements.txt
```

Hello, and thank you for visiting my blog! If you enjoyed this article, please consider supporting my work on this blog on Patreon (https://patreon.com/miguelgrinberg)!

BECOME A PATRON *(https://patreon.com/miguelgrinberg)*

Tweet Like Share