Application Factories

If you are already using packages and blueprints for your application (Modular Applications with Blueprints) there are a couple of really nice ways to further improve the experience. A common pattern is creating the application object when the blueprint is imported. But if you move the creation of this object into a function, you can then create multiple instances of this app later.

So why would you want to do this?

1. Testing. You can have instances of the application with different settings to test every case.
2. Multiple instances. Imagine you want to run different versions of the same application. Of course you could have multiple instances with different configs set up in your webserver, but if you use factories, you can have multiple instances of the same application running in the same application process which can be handy.

So how would you then actually implement that?

Basic Factories

The idea is to set up the application in a function. Like this:

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    from yourapplication.model import db
    db.init_app(app)

    from yourapplication.views.admin import admin
    from yourapplication.views.frontend import frontend
    app.register_blueprint(admin)
    app.register_blueprint(frontend)

    return app
```

The downside is that you cannot use the application object in the blueprints at import time. You can however use it from within a request. How do you get access to the application with the config? Use **current_app**:

```
from flask import current_app, Blueprint, render_template
admin = Blueprint('admin', __name__, url_prefix='/admin')
```

```
@admin.route('/')
def index():
    return render_template(current_app.config['INDEX_TEMPLATE'])
```

Here we look up the name of a template in the config.

Factories & Extensions

It's preferable to create your extensions and app factories so that the extension object does not initially get bound to the application.

Using Flask-SQLAlchemy, as an example, you should not do something along those lines:

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    db = SQLAlchemy(app)
```

But, rather, in model.py (or equivalent):

```
db = SQLAlchemy()
```

and in your application.py (or equivalent):

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    from yourapplication.model import db
    db.init_app(app)
```

Using this design pattern, no application-specific state is stored on the extension object, so one extension object can be used for multiple apps. For more information about the design of extensions refer to Flask Extension Development.

Using Applications

To run such an application, you can use the **flask** command:

```
$ export FLASK_APP=myapp
$ flask run
```

Flask will automatically detect the factory (`create_app` or `make_app`) in `myapp`. You can also pass arguments to the factory like this:

```
$ export FLASK_APP="myapp:create_app('dev')"
$ flask run
```

Then the `create_app` factory in `myapp` is called with the string `'dev'` as the argument. See Command Line Interface for more detail.

Factory Improvements

The factory function above is not very clever, but you can improve it. The following changes are straightforward to implement:

1. Make it possible to pass in configuration values for unit tests so that you don't have to create config files on the filesystem.
2. Call a function from a blueprint when the application is setting up so that you have a place to modify attributes of the application (like hooking in before/after request handlers etc.)
3. Add in WSGI middlewares when the application is being created if necessary.