Deploying with Setuptools

Setuptools, is an extension library that is commonly used to distribute Python libraries and extensions. It extends distutils, a basic module installation system shipped with Python to also support various more complex constructs that make larger applications easier to distribute:

- **support for dependencies**: a library or application can declare a list of other libraries it depends on which will be installed automatically for you.
- **package registry**: setuptools registers your package with your Python installation. This makes it possible to query information provided by one package from another package. The best known feature of this system is the entry point support which allows one package to declare an "entry point" that another package can hook into to extend the other package.
- **installation manager**: **pip** can install other libraries for you.

If you have Python 2 (>=2.7.9) or Python 3 (>=3.4) installed from python.org, you will already have pip and setuptools on your system. Otherwise, you will need to install them yourself.

Flask itself, and all the libraries you can find on PyPI are distributed with either setuptools or distutils.

In this case we assume your application is called `yourapplication.py` and you are not using a module, but a package. If you have not yet converted your application into a package, head over to the Larger Applications pattern to see how this can be done.

A working deployment with setuptools is the first step into more complex and more automated deployment scenarios. If you want to fully automate the process, also read the Deploying with Fabric chapter.

Basic Setup Script

Because you have Flask installed, you have setuptools available on your system. Flask already depends upon setuptools.

Standard disclaimer applies: you better use a virtualenv.

Your setup code always goes into a file named `setup.py` next to your application. The name of the file is only convention, but because everybody will look for a file with that name, you better not change it.

A basic `setup.py` file for a Flask application looks like this:

```
from setuptools import setup

setup(
    name='Your Application',
    version='1.0',
    long_description=__doc__,
    packages=['yourapplication'],
    include_package_data=True,
    zip_safe=False,
    install_requires=['Flask']
)
```

Please keep in mind that you have to list subpackages explicitly. If you want setuptools to lookup the packages for you automatically, you can use the `find_packages` function:

```
from setuptools import setup, find_packages

setup(
    ...
    packages=find_packages()
)
```

Most parameters to the `setup` function should be self explanatory, `include_package_data` and `zip_safe` might not be. `include_package_data` tells setuptools to look for a `MANIFEST.in` file and install all the entries that match as package data. We will use this to distribute the static files and templates along with the Python module (see Distributing Resources). The `zip_safe` flag can be used to force or prevent zip Archive creation. In general you probably don't want your packages to be installed as zip files because some tools do not support them and they make debugging a lot harder.

Tagging Builds

It is useful to distinguish between release and development builds. Add a `setup.cfg` file to configure these options.

```
[egg_info]
tag_build = .dev
tag_date = 1

[aliases]
release = egg_info -Db ''
```

Running `python setup.py sdist` will create a development package with ".dev" and the current date appended: `flaskr-1.0.dev20160314.tar.gz`. Running `python setup.py release sdist` will create a release package with only the version: `flaskr-1.0.tar.gz`.

Distributing Resources

If you try to install the package you just created, you will notice that folders like `static` or `templates` are not installed for you. The reason for this is that setuptools does not know which files to add for you. What you should do, is to create a `MANIFEST.in` file next to your `setup.py` file. This file lists all the files that should be added to your tarball:

```
recursive-include yourapplication/templates *
recursive-include yourapplication/static *
```

Don't forget that even if you enlist them in your `MANIFEST.in` file, they won't be installed for you unless you set the *include_package_data* parameter of the `setup` function to `True`!

Declaring Dependencies

Dependencies are declared in the `install_requires` parameter as a list. Each item in that list is the name of a package that should be pulled from PyPI on installation. By default it will always use the most recent version, but you can also provide minimum and maximum version requirements. Here some examples:

```
install_requires=[
    'Flask>=0.2',
    'SQLAlchemy>=0.6',
    'BrokenPackage>=0.7,<=1.0'
]
```

As mentioned earlier, dependencies are pulled from PyPI. What if you want to depend on a package that cannot be found on PyPI and won't be because it is an internal package you don't want to share with anyone? Just do it as if there was a PyPI entry and provide a list of alternative locations where setuptools should look for tarballs:

```
dependency_links=['http://example.com/yourfiles']
```

Make sure that page has a directory listing and the links on the page are pointing to the actual tarballs with their correct filenames as this is how setuptools will find the files. If you

have an internal company server that contains the packages, provide the URL to that server.

Installing / Developing

To install your application (ideally into a virtualenv) just run the `setup.py` script with the `install` parameter. It will install your application into the virtualenv's site-packages folder and also download and install all dependencies:

```
$ python setup.py install
```

If you are developing on the package and also want the requirements to be installed, you can use the `develop` command instead:

```
$ python setup.py develop
```

This has the advantage of just installing a link to the site-packages folder instead of copying the data over. You can then continue to work on the code without having to run `install` again after each change.