This is the last part of a three-part tutorial to build an employee management web app, named Project Dream Team. In Part Two (https://scotch.io/tutorials/build-a-crud-web-app-with-python-and-flask-part-two) of the tutorial, we built out the CRUD functionality of the app.

We created forms, views, and templates to list, add, edit and delete departments and roles. By the end of Part Two, we could assign (and re-assign) departments and roles to employees.

In Part Three, we will cover:

Table of Contents

1. » Custom error pages
2. » Unit tests
3. » Deployment on PythonAnywhere

Custom Error Pages

Web applications make use of HTTP errors to let users know that something has gone wrong. Default error pages are usually quite plain, so we will create our own custom ones for the following common HTTP errors:

1. » 403 Forbidden: this occurs when a user is logged in (authenticated), but does not have sufficient permissions to access the resource. This is the error we have been throwing when non-admins attempt to access an admin view.
2. » 404 Not Found: this occurs when a user attempts to access a non-existent resource

such as an invalid URL, e.g `http://127.0.0.1:5000/nothinghere`.

3. » 500 Internal Server Error: this is a general error thrown when a more specific error cannot be determined. It means that for some reason, the server cannot process the request.

We'll start by writing the views for the custom error pages. In your `app/__init__.py` file, add the following code:

PYTHON

```
# app/__init__

# update impo
from flask imp

# existing coo

def create_app

    # existing

    @app.erro
    def forbi
        retur

    @app.erro
    def page_
        retur

    @app.erro
    def inter
```

```
            returr

        return apr
```

We make use of Flask's `@app.errorhandler` decorator to define the error page views, where we pass in the status code as a parameter.

Next, we'll create the template files. Create a `app/templates/errors` directory, and in it, create `403.html`, `404.html`, and `500.html`.

```
HTML

<!-- app/templ
```

```
{% extends "ba
{% block title
{% block body
<div class="co
  <div class="
    <div class
      <div cla
        <div s
            <h
            <h
            <h
            <a



            </
        </div>
      </div>
    </div>
  </div>
</div>
{% endblock %}
```

HTML

```
<!-- app/temp

{% extends "ba
{% block title
{% block body
<div class="co
  <div class="
    <div class
      <div cla
        <div s
            <h
            <h
            <h
            <a



            </
        </div>
      </div>
    </div>
  </div>
```

```
</div>
{% endblock %]
```

HTML

```
<!-- app/templ

{% extends "ba
{% block title
{% block body
<div class="co
  <div class="
    <div class
      <div cla
        <div s
            <h
            <h
            <h
            <a
```

```
                    </
                </div
            </div>
        </div>
    </div>
</div>
{% endblock %
```

All the templates give a brief description of the error, and a button that links to the homepage.

Run the app and log in as a non-admin user, then attempt to access `http://127.0.0.1:5000/admin/departments`. You should get the following page:

vanilla

JavaScript

👉 React

Now attempt to access this non-existent page: `http://127.0.0.1:5000/nothinghere`. You should see:

To view the internal server error page, we'll create a temporary route where we'll use Flask's `abort()` function to raise a 500 error. In the `app/__init__.py` file, add the following:

PYTHON

```
# app/__init__
# update impo
from flask imp

# existing coc

def create_app
    # existing

    @app.route
    def error(
        abort(

    return app
```

Go to

`http://127.0.0.1:5000/500`

; you should see the following page:

Now you can remove the temporary route we just created for the internal server error.

Tests

Now, let's write some tests for the app. The importance of testing software can't be overstated. Tests help ensure that your app is working as expected, without the need for you to manually test all of your app's functionality.

We'll begin by creating a test database, and give the database user we created in Part One (https://scotch.io/tutorials/build-a-crud-web-app-with-python-and-flask-part-one) all privileges on it:

BASH

```
$ mysql -u roc

mysql> CREATE
Query OK, 1 rc
```

```
mysql> GRANT A
Query OK, 0 ro
```

Now we need to edit the `config.py` file to add configurations for testing. Delete the current contents and replace them with the following code:

PYTHON

```python
# config.py

class Config(o
    """
    Common con
    """
```

```
        DEBUG = Tr


class Developm
    """
    Developmen
    """

    SQLALCHEMY


class Producti
    """
    Productio
    """

    DEBUG = Fa


class Testing(
    """
    Testing co
    """
```

```
        TESTING =

    app_config =
        'developme
        'productic
        'testing'
    }
```

We have put DEBUG = True in the base class, Config, so that it is the default setting. We override this in the ProductionConfig class. In the TestingConfig class, we set the TESTING configuration variable to True.

We will be writing unit tests. Unit tests are written to test small, individual, and fairly isolated units of code, such as functions. We will make use of Flask-Testing (https://pythonhosted.org/Flask-Testing/), an extension that provides unit testing utilities for Flask.

BASH

```
$ pip install
```

Next, create a `tests.py` file in the root directory of your app. In it, add the following code:

PYTHON

```
# tests.py

import unittes

from flask_tes

from app impor
from app.model


class TestBase

    def create

        # pass
        confi
        app =
        app.co
            SQ
        )
        retur
    def setUp(
        """
        Will b
        """

        db.cre

        # crea
        admin 

        # crea
        employ

        # save
        db.ses
        db.ses
        db.ses

    def tearDo
        """
        Will b
        """
```

```
                db.ses
                db.dro

        if __name__ ==
            unittest.r
```

In the base class above, `TestBase` , we have a `create_app` method, where we pass in the configurations for testing.

We also have two other methods: `setUp` and `tearDown` . The `setUp` method will be called automatically before every test we run. In it, we create two test users, one admin and one non-admin, and save them to the database. The `tearDown` method will be called automatically after every test. In it, we remove the database session and drop all database tables.

To run the tests, we will run the `tests.py` file:

BASH

```
$ python tests

--------------
Ran 0 tests i

OK
```

The output above lets us know that our test setup is OK. Now let's write some tests.

PYTHON

```
# tests.py

# update impo

import os

from flask imp

from app.mode

# add the fol

class TestMode

    def test_e
        """
        Test r
        """
```

```
            self.a

    def test_c
            """
            Test r
            """

            # crea
            depart

            # save
            db.ses
            db.ses

            self.a

    def test_r
            """
            Test r
            """

            # crea
            role =
        # save
        db.ses
        db.ses

        self.a


class TestView

    def test_h
        """
        Test t
        """
        respon
        self.a

    def test_l
        """
        Test t
        """
        respon
        self.a

    def test_l
```

```
        """
        Test t
        and re
        """
        target
        redire
        respon
        self.a
        self.a

    def test_d
        """
        Test t
        and re
        """
        target
        redire
        respon
        self.a
        self.a

    def test_a
        """
        Test t
```

```
            and re
            """
            target
            redire
            respor
            self.a
            self.a

        def test_c
            """
            Test t
            and re
            """
            target
            redire
            respor
            self.a
            self.a

        def test_r
            """
            Test t
            and re
            """
        targe
        redire
        respo
        self.
        self.

    def test_
        """
        Test 
        and re
        """
        targe
        redire
        respo
        self.
        self.


class TestErr

    def test_
        # cre
        @self
        def f
```

```
            ab

        respo
        self.a
        self.a

    def test_4
        respo
        self.a
        self.a

    def test_5
        # crea
        @self.
        def in
            ab

        respo
        self.a
        self.a


if __name__ ==
    unittest.r
```

We've added three classes: `TestModels`, `TestViews` and `TestErrorPages`.

The first class has methods to test that each of the models in the app are working as expected. This is done by querying the database to check that the correct number of records exist in each table.

The second class has methods that test the views in the app to ensure the expected status code is returned. For non-restricted views, such as the homepage and the login page, the `200 OK` code should be returned; this means that everything is OK and the request has succeeded. For restricted views that require authenticated access, a `302 Found` code is returned. This means that the page is redirected to an existing resource, in this case, the login page. We test both that the `302 Found` code is returned and that the page redirects to the login page.

The third class has methods to ensure that the error pages we created earlier are shown when the respective error occurs.

Note that each test

method begins with `test`. This is deliberate, because `unittest`, the Python unit testing framework, uses the `test` prefix to automatically identify test methods. Also note that we have not written tests for the front-end to ensure users can register and login, and to ensure administrators can create departments and roles and assign them to employees. This can be done using a tool like Selenium Webdriver (http://www.seleniumhq.org/projects/webdriver/); however this is outside the scope of this tutorial.

Run the tests again:

BASH

```
$ python tests
..............
--------------
Ran 14 tests

OK
```

Success! The tests are passing.

Deploy!

Now for the final part of the tutorial: deployment. So far, we've been running the app locally. In this stage, we will publish the application on the internet so that other people can use it. We will use PythonAnywhere (https://www.pythonanywhere.com), a Platform as a Service (PaaS) that is easy to set up, secure, and scalable, not to mention free for basic accounts!

PythonAnywhere Set-Up

Create a free PythonAnywhere account here (https://www.pythonanywhere.com/registration/register/beginner/) if you don't already have one. Be sure to select your username carefully since the app will be accessible at `your-username.pythonanywhere.com`.

Once you've signed up, `your-username.pythonanywhere.com` should show this page:

We will use git to upload the app to PythonAnywhere. If you've been pushing your code to cloud repository management systems like Bitbucket (https://bitbucket.org/) , Gitlab (https://about.gitlab.com/) or Github (github.com/), that's great! If not, now's the time to do it. Remember that we won't be pushing the `instance` directory, so be sure to include it in your `.gitignore` file, like so:

BASH

```bash
# .gitignore

*.pyc
instance/
```

Also, ensure that your `requirements.txt` file is up to date using the `pip freeze` command before pushing your code:

BASH

```
$ pip freeze
```

Now, log in to your PythonAnywhere account. In your dashboard, there's a `Consoles` tab; use it to start a new Bash console.

In the PythonAnywhere Bash console, clone your repository.

BASH

```
$ git clone ht
```

Next we will create a virtualenv, then install the dependencies from the `requirements.txt` file. Because PythonAnywhere installs virtualenvwrapper (https://virtualenvwrapper.readthedocs./en/latest/) for all users by default, we can use its commands:

BASH

```bash
$ mkvirtualenv
$ cd project-c
$ pip install
```

We've created a virtualenv called `dream-team`. The virtualenv is automatically activated. We then entered the project directory and installed the dependencies.

Now, in the Web tab on your dashboard, create a new web app.

Select the Manual Configuration option (**not** the Flask option), and choose Python 2.7 as your Python version. Once the web app is created, its configurations will be loaded. Scroll down to the Virtualenv section, and enter the name of the virtualenv you just created:

Database Configuration

Next, we will set up the MySQL production database. In the

Databases tab of your PythonAnywhere dashboard, set a new password and then initialize a MySQL server:

The password above will be your database user password. Next, create a new database if you wish. PythonAnywhere already has a default database which you can use.

By default, the database user is your username, and has all privileges granted on any databases created. Now, we need to migrate the database and populate it with the tables. In a Bash console on PythonAnywhere, we will run the

`flask db upgrade`

command, since we already have the migrations directory that we created locally. Before running the commands, ensure you are in your virtualenv as well as in the project directory.

BASH

```
$ export FLASK
$ export FLASK
$ export SQLAL
$ flask db upg
```

When setting the `SQLALCHEMY_DATABASE_URI` environment variable, remember to replace `your-username` , `your-password` , `your-host-address` and `your-database-name` with their correct values. The username, host address and database name can be found in the MySQL settings in the Databases tab on your dashboard. For example, using the information below, my database URI is: `mysql://projectdreamteam:passwor services.com/projectdreamteam$dr`

WSGI File

Now we will edit the WSGI file, which PythonAnywhere uses to serve the app. Remember that we are not pushing the `instance` directory to version control. We therefore need to configure the environment variables for production, which we will do in the WSGI file.

In the Code section of the Web tab on your dashboard, click on the link to the WSGI configuration file.

Delete all the current contents of the file, and replace them with the following:

PYTHON

```
import os
import sys

path = '/home,
    if path not i
        sys.path.a

    os.environ['FL
    os.environ['SE
    os.environ['SQ

    from run impo
```

In the file above, we tell PythonAnywhere to get the variable `app` from the `run.py` file, and serve it as the application. We also set the `FLASK_CONFIG`, `SECRET_KEY` and `SQLALCHEMY_DATABASE_URI` environment variables. Feel free to alter the secret key. Note that the `path` variable should contain your username and project directory name, so be sure to replace it with the correct values. The same applies for the database URI environment variable.

We also need to edit our local `app/__init__py` file to prevent it from loading the `instance/config.py` file in production, as well as to load the configuration variables we've set:

PYTHON

```
# app/__init_

# update impo
```

```
import os

# existing coc

def create_app
    if os.gete
        app =
        app.cc
            SE
            SC
        )
    else:
        app =
        app.cc
        app.cc

    # existing
```

Push your changes to version control, and pull them on the PythonAnywhere Bash console:

BASH

```
$ git pull or
```

Now let's try loading the app on PythonAnywhere. First, we need to reload the app on the Web tab in the dashboard:

Now go to your app URL:

Great, it works! Try registering a new user and logging in. This should work just as it did locally.

Admin User

We will now create an admin user the same way we did locally. Open the Bash console, and run the following commands:

BASH

```bash
$ flask shell
```

```
>>> from app.r
>>> from app
>>> admin = Er
>>> db.sessior
>>> db.sessior
```

Now you can login as an admin user and add departments and roles, and assign them to employees.