

# Build a CRUD Web App With Python and Flask - Part Three

*Mbithe Nzomo*

This is the last part of a three-part tutorial to build an employee management web app, named Project Dream Team. In [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:

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:

```
from flask import Flask, render_template
```

```
def create_app(config_name):

    @app.errorhandler(403)
    def forbidden(error):
        return render_template('errors/403.html', title='Forbidden'), 403

    @app.errorhandler(404)
    def page_not_found(error):
        return render_template('errors/404.html', title='Page Not Found'), 404

    @app.errorhandler(500)
    def internal_server_error(error):
        return render_template('errors/500.html', title='Server Error'), 500

    return app
```

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`.

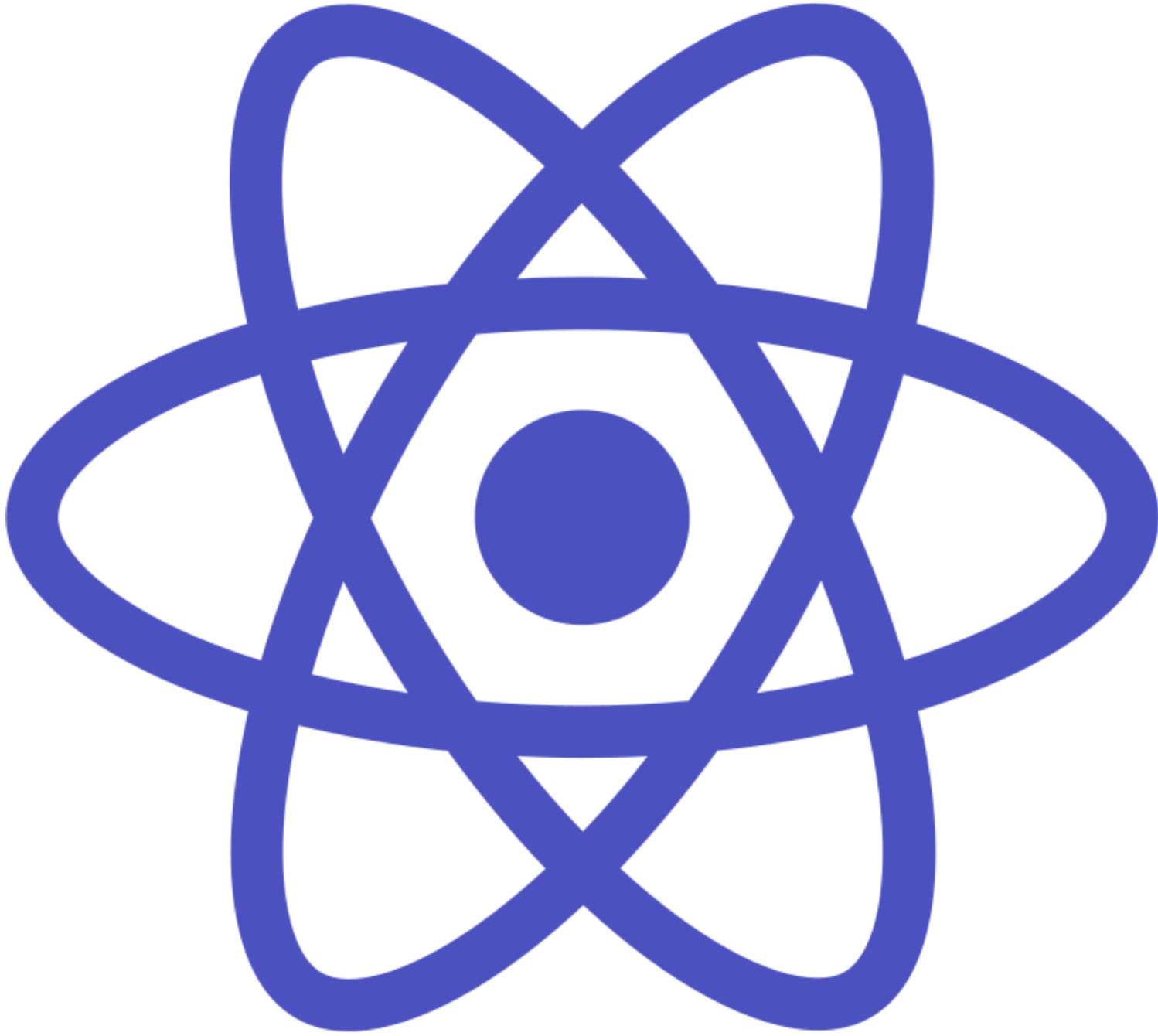
```
{% extends "base.html" %}
{% block title %}Forbidden{% endblock %}
{% block body %}
<div class="content-section">
  <div class="outer">
    <div class="middle">
      <div class="inner">
        <div style="text-align: center">
          <h1> 403 Error </h1>
          <h3> You do not have sufficient permissions to access this page. </h3>
          <hr class="intro-divider">
          <a href="{{ url_for('home.homepage') }}" class="btn btn-default btn-lg">
            <i class="fa fa-home"></i>
            Home
          </a>
        </div>
      </div>
    </div>
  </div>
</div>
{% endblock %}
```

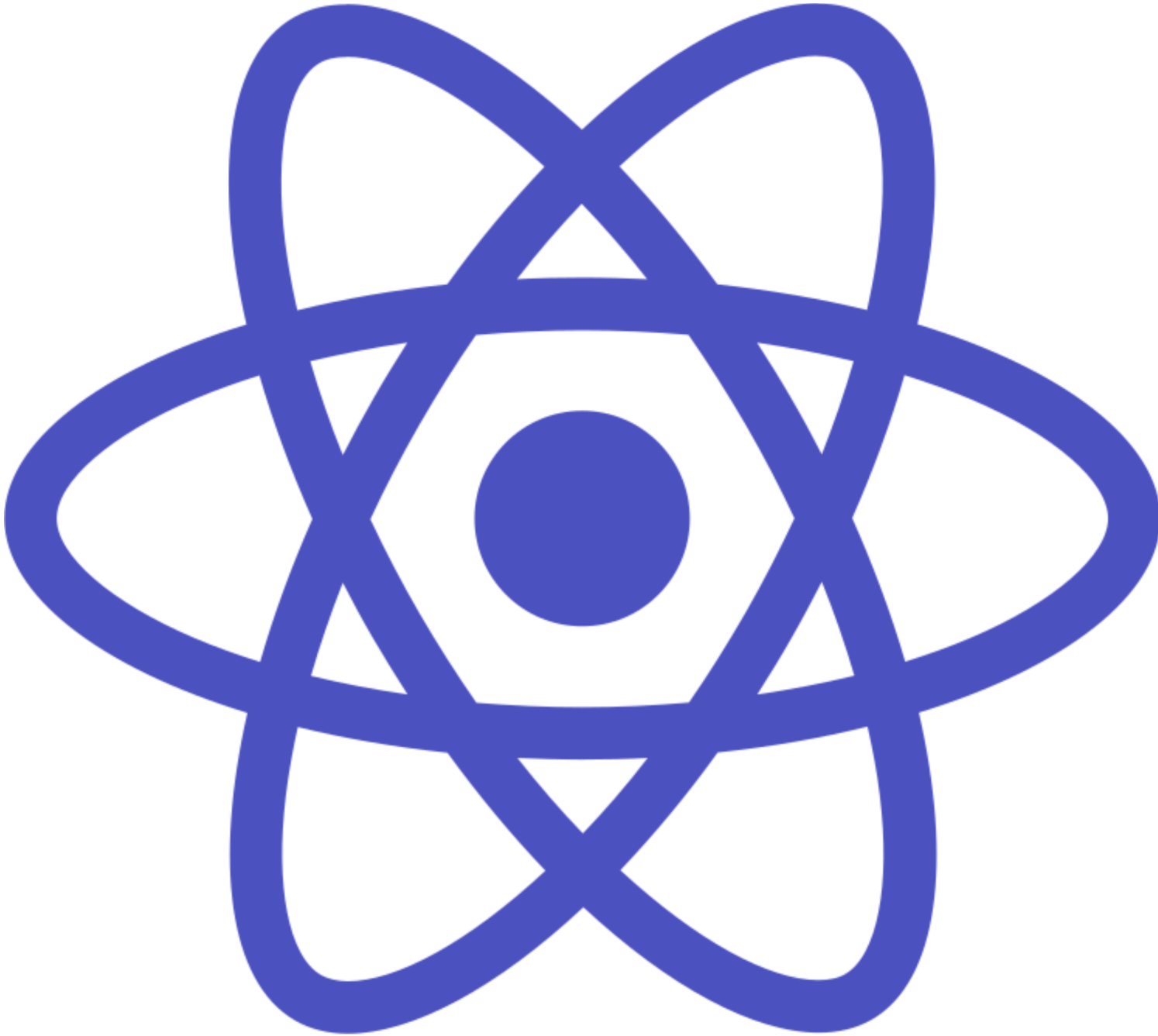
```
{% extends "base.html" %}
{% block title %}Page Not Found{% endblock %}
{% block body %}
<div class="content-section">
  <div class="outer">
    <div class="middle">
      <div class="inner">
        <div style="text-align: center">
          <h1> 404 Error </h1>
          <h3> The page you're looking for doesn't exist. </h3>
          <hr class="intro-divider">
          <a href="{{ url_for('home.homepage') }}" class="btn btn-default btn-lg">
            <i class="fa fa-home"></i>
            Home
          </a>
        </div>
      </div>
    </div>
  </div>
</div>
{% endblock %}
```

```
{% extends "base.html" %}
{% block title %}Internal Server Error{% endblock %}
{% block body %}
<div class="content-section">
  <div class="outer">
    <div class="middle">
      <div class="inner">
        <div style="text-align: center">
          <h1> 500 Error </h1>
          <h3> The server encountered an internal error. That's all we know. </h3>
          <hr class="intro-divider">
          <a href="{{ url_for('home.homepage') }}" class="btn btn-default btn-lg">
            <i class="fa fa-home"></i>
            Home
          </a>
        </div>
      </div>
    </div>
  </div>
</div>
{% endblock %}
```

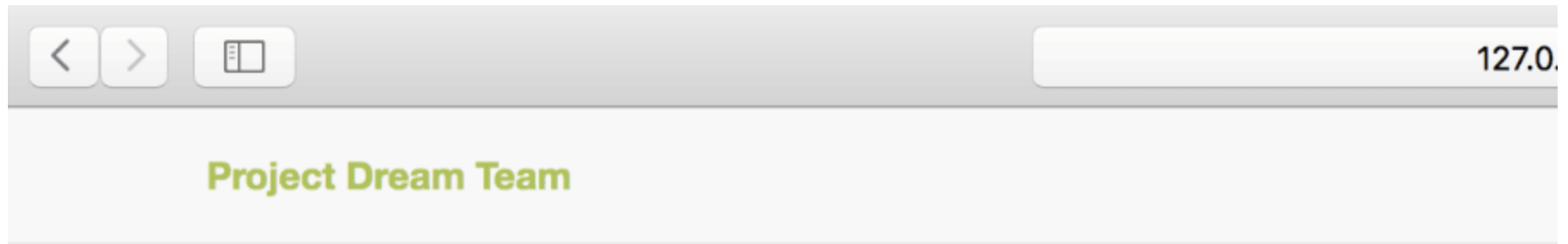
All the templates give a brief description of the error, and a button that links to the homepage.

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:





[Go from vanilla JavaScript](#) 📌 [React](#)



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





## Project Dream Team

4

The page you'r

---

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:

```
from flask import abort, Flask, render_template

def create_app(config_name):

    @app.route('/500')
    def error():
        abort(500)

    return app
```

Go to `http://127.0.0.1:5000/500`; you should see the following page:



5

# The server encountered

---

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](#) all privileges on it:

```
$ mysql -u root

mysql> CREATE DATABASE dreamteam_test;
Query OK, 1 row affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON dreamteam_test . * TO 'dt_admin'@'localhost';
Query OK, 0 rows affected (0.00 sec)
```

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:

```
class Config(object):
    """
    Common configurations
    """

    DEBUG = True

class DevelopmentConfig(Config):
    """
    Development configurations
    """

    SQLALCHEMY_ECHO = True

class ProductionConfig(Config):
    """
    Production configurations
    """

    DEBUG = False
```

```
class TestingConfig(Config):
    """
    Testing configurations
    """

    TESTING = True

app_config = {
    'development': DevelopmentConfig,
    'production': ProductionConfig,
    'testing': TestingConfig
}
```

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://pypi.org/project/Flask-Testing/), an extension that provides unit testing utilities for Flask.

```
$ pip install Flask-Testing
```

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

```
import unittest

from flask_testing import TestCase

from app import create_app, db
from app.models import Employee

class TestBase(TestCase):

    def create_app(self):

        config_name = 'testing'
        app = create_app(config_name)
        app.config.update(
            SQLALCHEMY_DATABASE_URI='mysql://dt_admin:dt2016@localhost/dreamteam_test'
        )
        return app

    def setUp(self):
```

```

    """
    Will be called before every test
    """

    db.create_all()

    admin = Employee(username="admin", password="admin2016", is_admin=True)

    employee = Employee(username="test_user", password="test2016")

    db.session.add(admin)
    db.session.add(employee)
    db.session.commit()

    def tearDown(self):
        """
        Will be called after every test
        """

        db.session.remove()
        db.drop_all()

if __name__ == '__main__':
    unittest.main()

```

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:

```
$ python tests.py
```

```
-----
Ran 0 tests in 0.000s
```

```
OK
```

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

```
import os

from flask import abort, url_for

from app.models import Department, Employee, Role


class TestModels(TestBase):

    def test_employee_model(self):
        """
        Test number of records in Employee table
        """
        self.assertEqual(Employee.query.count(), 2)

    def test_department_model(self):
        """
        Test number of records in Department table
        """

        department = Department(name="IT", description="The IT Department")

        db.session.add(department)
        db.session.commit()

        self.assertEqual(Department.query.count(), 1)

    def test_role_model(self):
        """
        Test number of records in Role table
        """

        role = Role(name="CEO", description="Run the whole company")

        db.session.add(role)
        db.session.commit()

        self.assertEqual(Role.query.count(), 1)
```

```
class TestViews(TestBase):

    def test_homepage_view(self):
        """
        Test that homepage is accessible without login
        """
        response = self.client.get(url_for('home.homepage'))
        self.assertEqual(response.status_code, 200)

    def test_login_view(self):
        """
        Test that login page is accessible without login
        """
        response = self.client.get(url_for('auth.login'))
        self.assertEqual(response.status_code, 200)

    def test_logout_view(self):
        """
        Test that logout link is inaccessible without login
        and redirects to login page then to logout
        """
        target_url = url_for('auth.logout')
        redirect_url = url_for('auth.login', next=target_url)
        response = self.client.get(target_url)
        self.assertEqual(response.status_code, 302)
        self.assertRedirects(response, redirect_url)

    def test_dashboard_view(self):
        """
        Test that dashboard is inaccessible without login
        and redirects to login page then to dashboard
        """
        target_url = url_for('home.dashboard')
        redirect_url = url_for('auth.login', next=target_url)
        response = self.client.get(target_url)
        self.assertEqual(response.status_code, 302)
        self.assertRedirects(response, redirect_url)

    def test_admin_dashboard_view(self):
        """
        Test that dashboard is inaccessible without login
        and redirects to login page then to dashboard
        """
        target_url = url_for('home.admin_dashboard')
        redirect_url = url_for('auth.login', next=target_url)
        response = self.client.get(target_url)
        self.assertEqual(response.status_code, 302)
        self.assertRedirects(response, redirect_url)
```



```
def test_departments_view(self):
    """
    Test that departments page is inaccessible without login
    and redirects to login page then to departments page
    """
    target_url = url_for('admin.list_departments')
    redirect_url = url_for('auth.login', next=target_url)
    response = self.client.get(target_url)
    self.assertEqual(response.status_code, 302)
    self.assertRedirects(response, redirect_url)

def test_roles_view(self):
    """
    Test that roles page is inaccessible without login
    and redirects to login page then to roles page
    """
    target_url = url_for('admin.list_roles')
    redirect_url = url_for('auth.login', next=target_url)
    response = self.client.get(target_url)
    self.assertEqual(response.status_code, 302)
    self.assertRedirects(response, redirect_url)

def test_employees_view(self):
    """
    Test that employees page is inaccessible without login
    and redirects to login page then to employees page
    """
    target_url = url_for('admin.list_employees')
    redirect_url = url_for('auth.login', next=target_url)
    response = self.client.get(target_url)
    self.assertEqual(response.status_code, 302)
    self.assertRedirects(response, redirect_url)

class TestErrorPages(TestBase):

    def test_403_forbidden(self):

        @self.app.route('/403')
        def forbidden_error():
            abort(403)

        response = self.client.get('/403')
        self.assertEqual(response.status_code, 403)
        self.assertTrue("403 Error" in response.data)

    def test_404_not_found(self):
        response = self.client.get('/nothinghere')
```

```

        self.assertEqual(response.status_code, 404)
        self.assertTrue("404 Error" in response.data)

    def test_500_internal_server_error(self):

        @self.app.route('/500')
        def internal_server_error():
            abort(500)

        response = self.client.get('/500')
        self.assertEqual(response.status_code, 500)
        self.assertTrue("500 Error" in response.data)

if __name__ == '__main__':
    unittest.main()

```

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](#); however this is outside the scope of this tutorial.

Run the tests again:

```

$ python tests.py
.....
-----
Ran 14 tests in 2.313s

```

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](#), 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](#) 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:



# Coming Soon

This is going to be another

PythonAnywhere lets you  
free plan gives you access  
for you. You can develop a  
directly from your browser  
your own server.

Need more power? Upgrade

[You can find out more about](#)

## Developer info

Hi! If this is your PythonAnywhere  
need to create a web app to have

We will use git to upload the app to PythonAnywhere. If you've been pushing your code to cloud repository management systems like [Bitbucket](#), [Gitlab](#) or [Github](#), 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:

```
*.pyc  
instance/
```

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

```
$ pip freeze > requirements.txt
```

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



pythonanywhere

Consoles

Files

Web

Schedule

Datab

## Start a new console:

Python: [3.5](#) / [3.4](#) / [3.3](#) / [2.7](#) IPython: [3.5](#) / [3.4](#) / [3.3](#) / [2.7](#) PyPy: [2.7](#)Other: [Bash](#) | [MySQL](#)Custom: [+](#)

## Your consoles:

*You have no consoles. Click a link above to start one.*

In the PythonAnywhere Bash console, clone your repository.

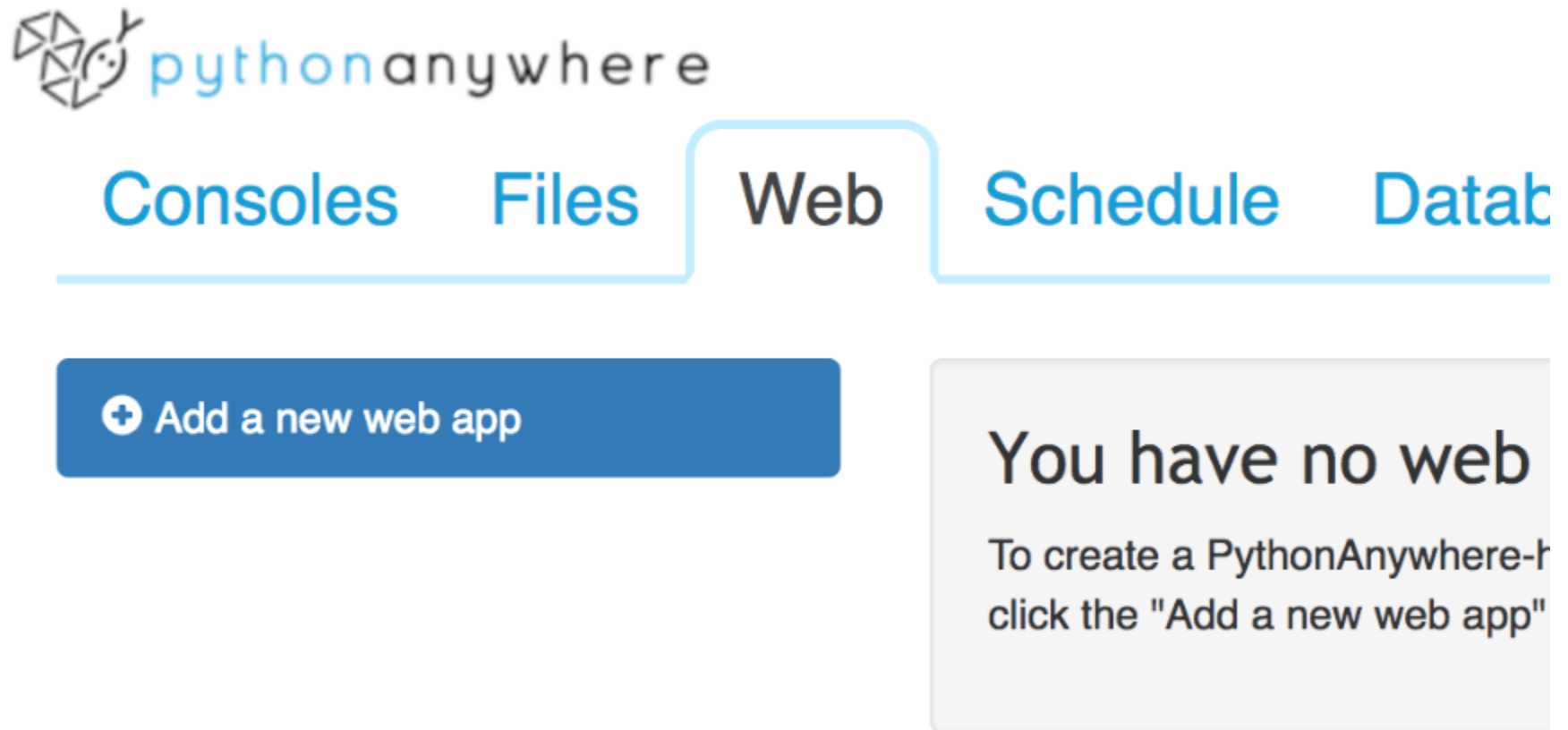
```
$ git clone https://github.com/andela-mnzomo/project-dream-team-three
```

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

```
$ mkvirtualenv dream-team
$ cd project-dream-team-three
$ pip install -r requirements.txt
```

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

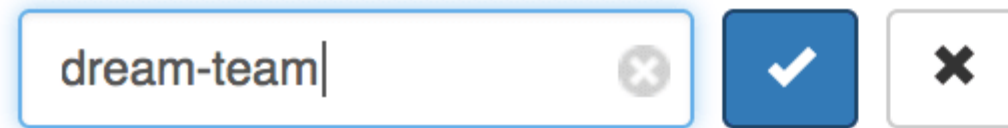
Now, in the Web tab on your dashboard, create a new web app.



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:

## Virtualenv:

Use a virtualenv to get different versions of flask, django etc from our [More info here](#). You need to **Reload your web app** to activate it; NB - virtualenv does not exist.



### 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:



[Consoles](#)[Files](#)[Web](#)[Schedule](#)[Databases](#)[MySQL](#)[Postgres](#)

## Initialize MySQL

Let's get started! The first thing

Enter a new password in the form below for the databases once you've created them.

**New password:**

.....

**Confirm password:**

.....

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.

## Create a database

Your database names always start with your username + "\$". There's a prefix in below, though: PythonAnywhere will automatically add it.

**Database name:**

dreamteam\_db

Create

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.

```
$ export FLASK_CONFIG=production
$ export FLASK_APP=run.py
$ export SQLALCHEMY_DATABASE_URI='mysql://your-username:your-password@your-host-address/your-database-name'
$ flask db upgrade
```

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:password@projectdreamteam.mysql.pythonanywhere-services.com
```

```
/projectdreamteam$dreamteam_db
```

[Consoles](#)[Files](#)[Web](#)[Schedule](#)[MySQL](#)[Postgres](#)

## MySQL set

### Connecting:

Use these settings in your code:

Database host:

1

### Your databases:

Click a database's name to view its details.

Start a new database

Start a new database

## 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.

## Code:

---

What your site is running.

Source code:	<u><i>Enter the path to your web app s</i></u>
Working directory:	<u>/home/projectdreamteam/</u>
WSGI configuration file:	<u>/var/www/projectdreamteam_pyth</u>
Python version:	2.7

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

```
import os
import sys

path = '/home/your-username/your-project-directory-name'
if path not in sys.path:
    sys.path.append(path)

os.environ['FLASK_CONFIG'] = 'production'
os.environ['SECRET_KEY'] = 'p9Bv<3Eid9%$i01'
os.environ['SQLALCHEMY_DATABASE_URI'] = 'mysql://your-username:your-password@your-host-address/your-database-name'

from run import app as application
```

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:

```
import os

def create_app(config_name):
    if os.getenv('FLASK_CONFIG') == "production":
        app = Flask(__name__)
        app.config.update(
            SECRET_KEY=os.getenv('SECRET_KEY'),
            SQLALCHEMY_DATABASE_URI=os.getenv('SQLALCHEMY_DATABASE_URI')
        )
    else:
        app = Flask(__name__, instance_relative_config=True)
        app.config.from_object(app_config[config_name])
        app.config.from_pyfile('config.py')
```

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

```
$ git pull origin master
```

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

[Consoles](#)[Files](#)[Web](#)[Schedule](#)[Datab](#)

projectdreamteam.pythonanywher...

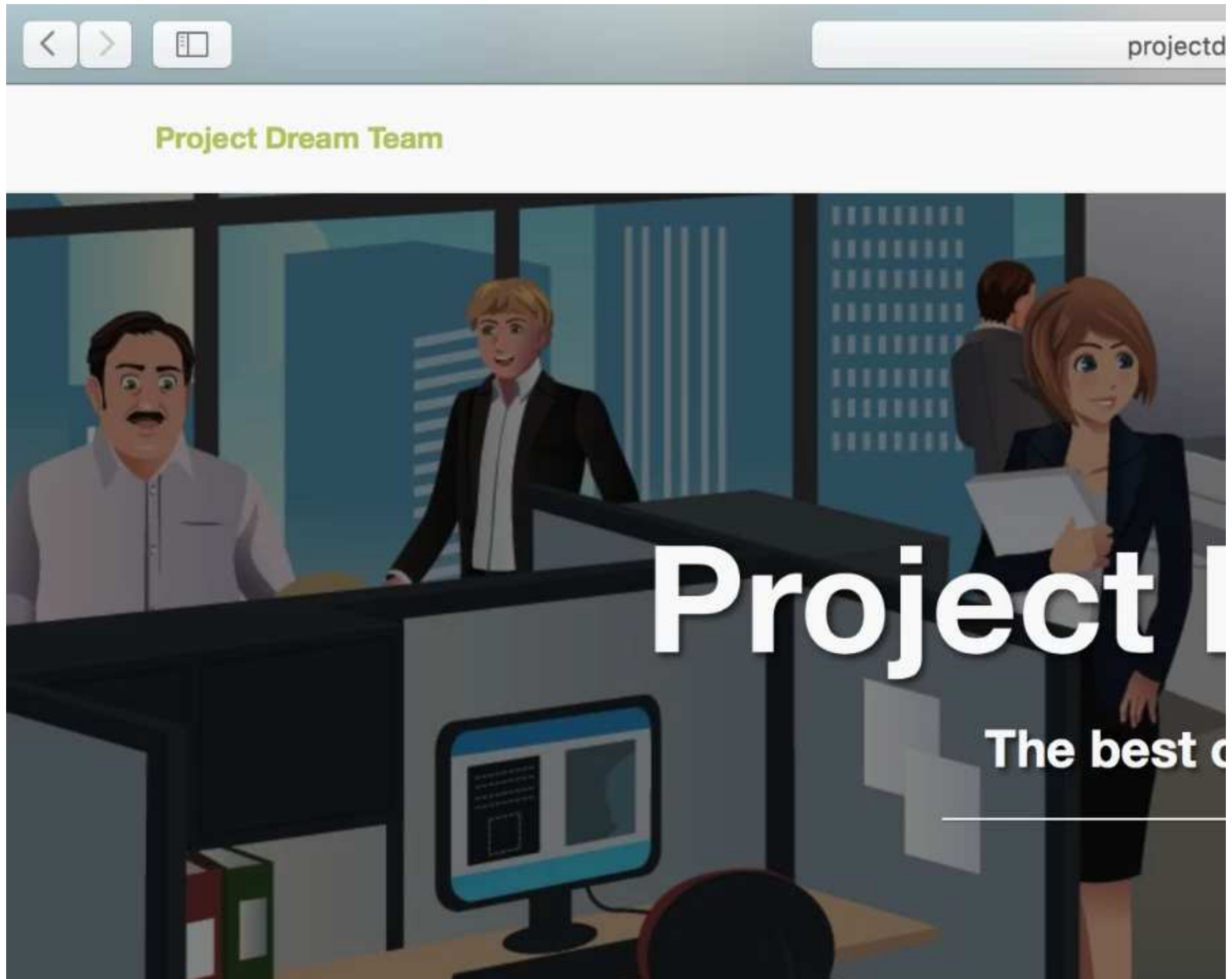
[+ Add a new web app](#)

## Configuration for projectdreamteam.

Reload:


[↻ Reload projectdreamteam.](#)

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.



projectdream

Project Dream Team

# Register for an account

**Email**

username@some-domain.com

**Username**


username

**First Name**

First

**Last Name**

Last



projectdream

**Project Dream Team**

You have successfully registered

# Login to your account

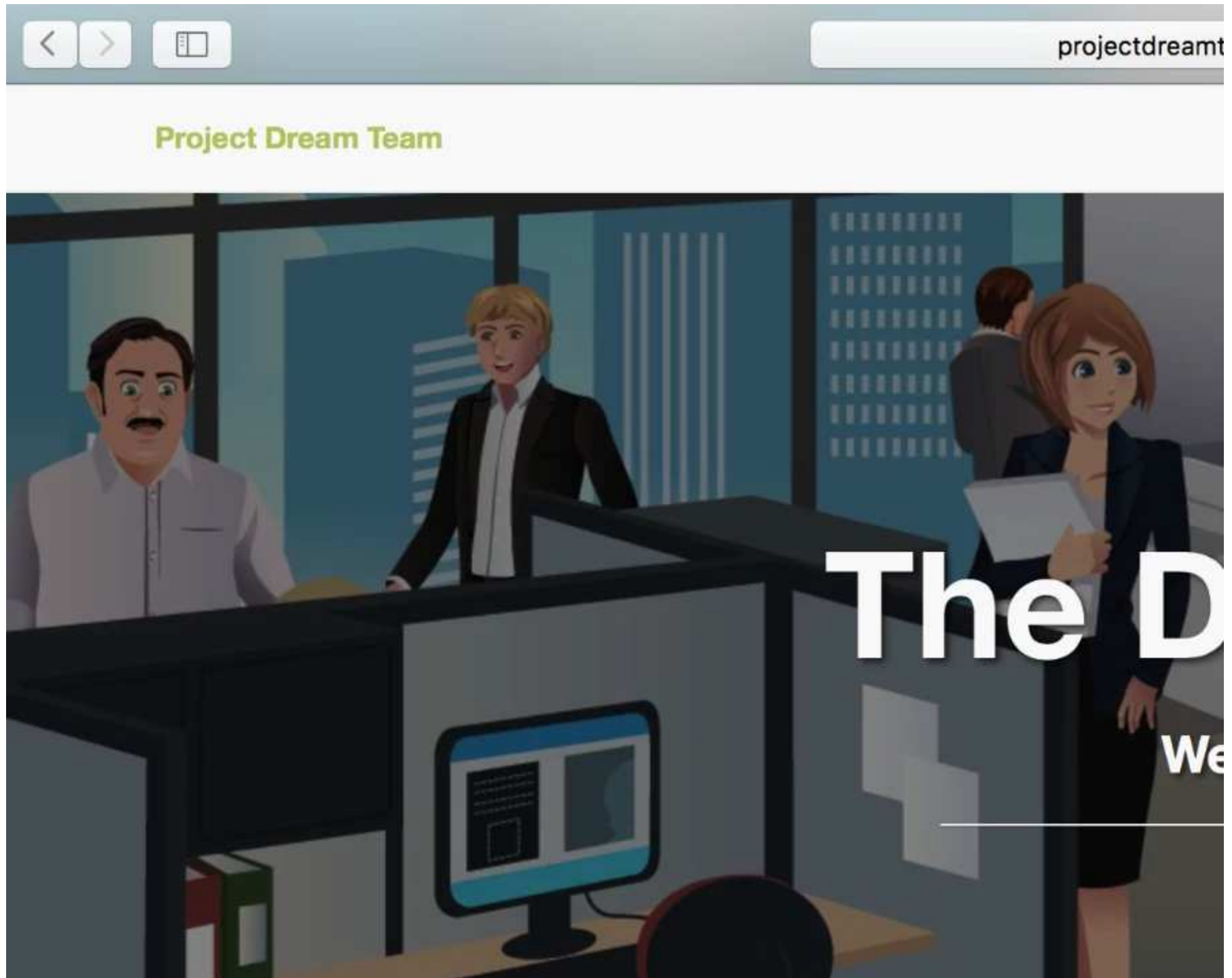
Email

username@some-domain.com

Password

....

Login




## Admin User

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

```
$ flask shell
>>> from app.models import Employee
>>> from app import db
>>> admin = Employee(email="admin@admin.com",username="admin",password="admin2016",is_admin=True)
>>> db.session.add(admin)
>>> db.session.commit()
```

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



projectdream

Project Dream Team

# Login to your account

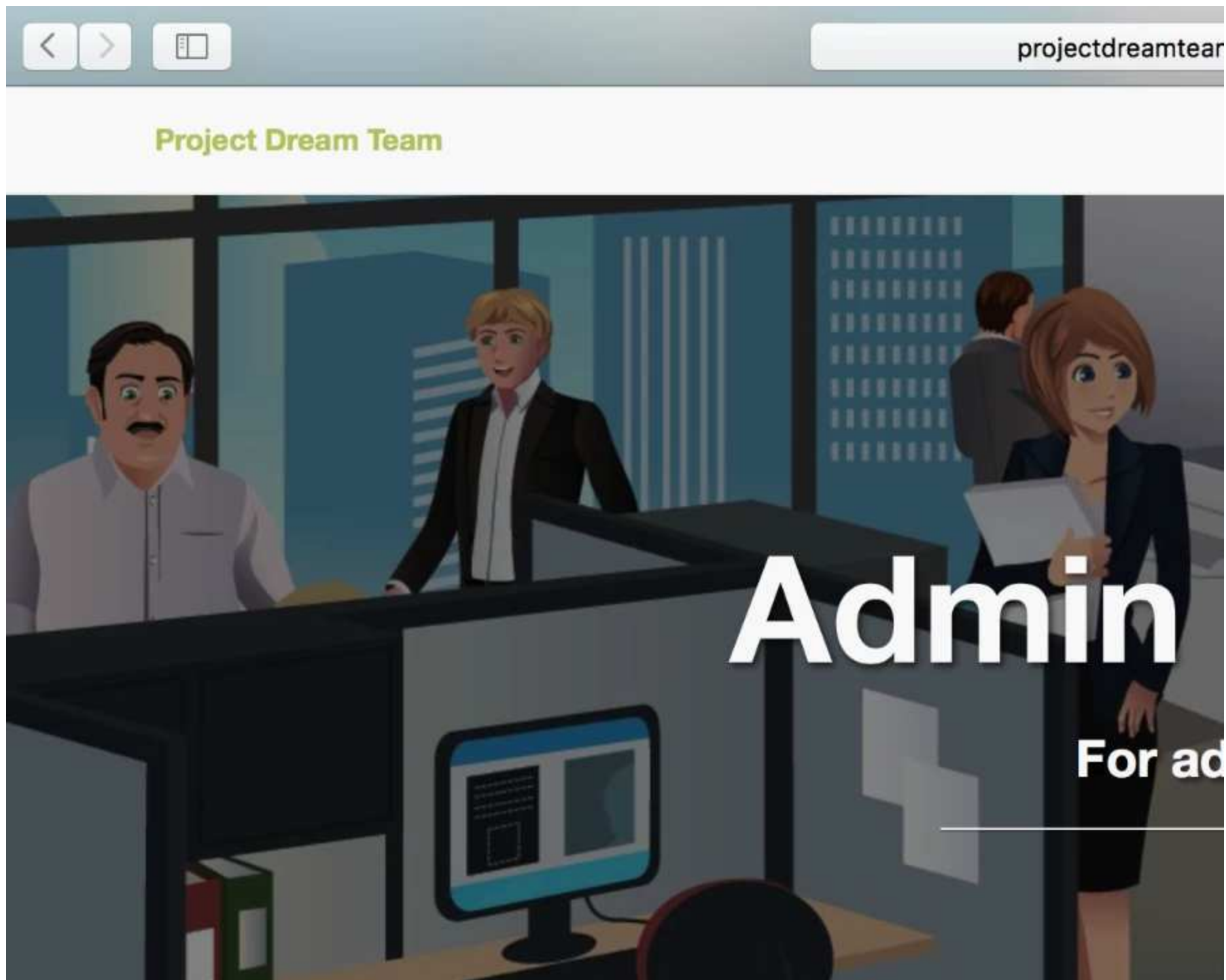
Email

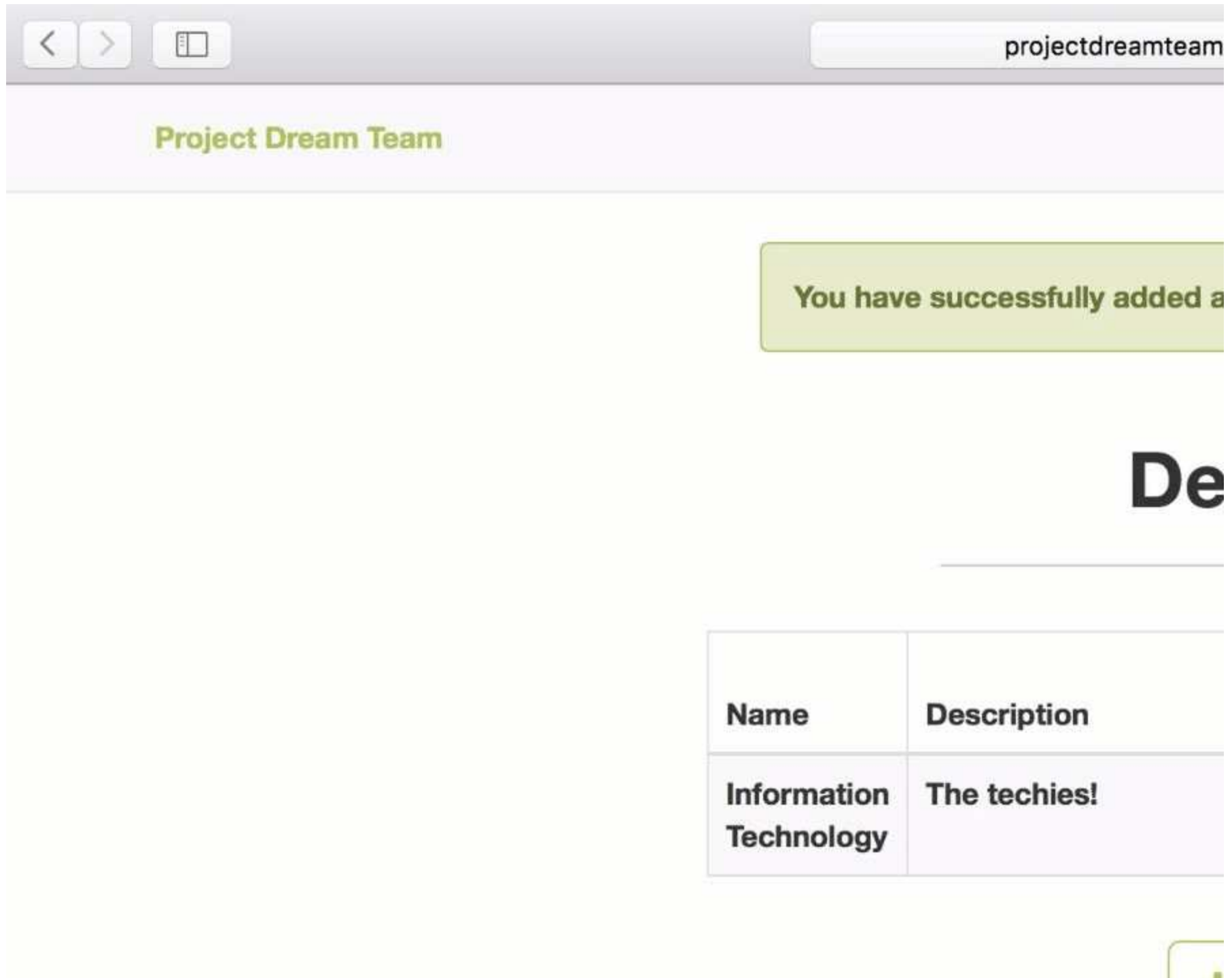
admin@admin.com

Password

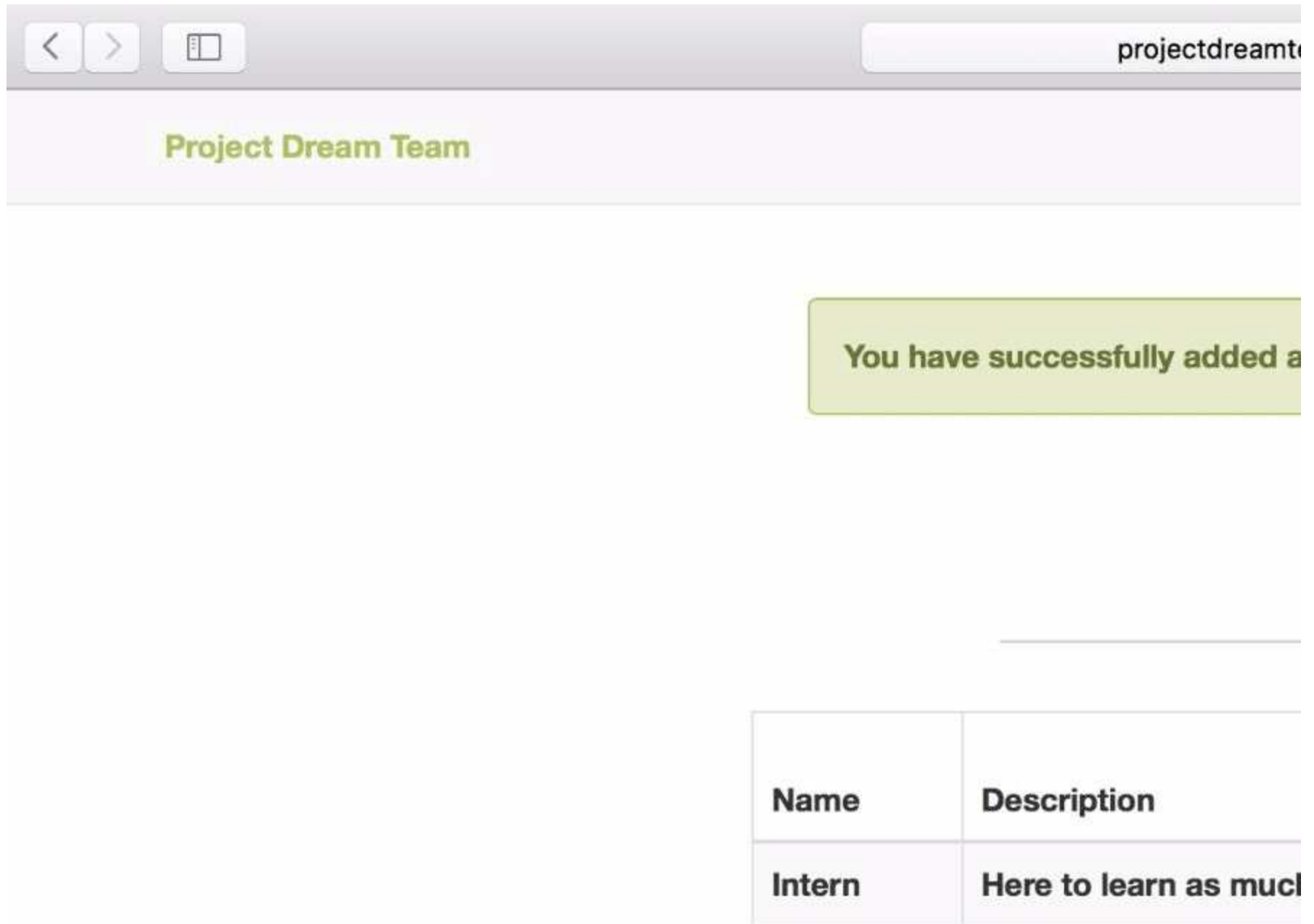
.....

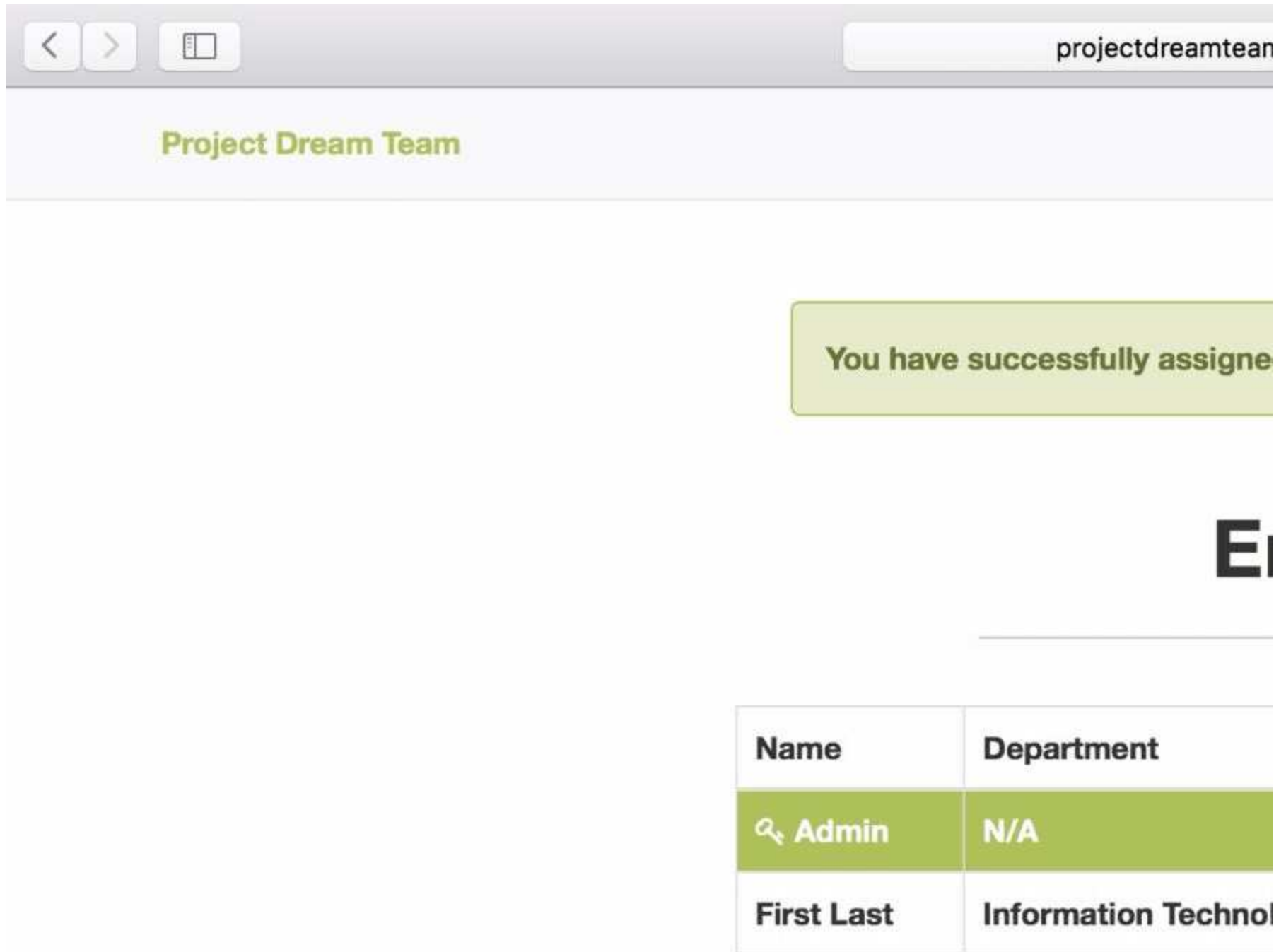
Login











## Conclusion

Congratulations on successfully deploying your first Flask CRUD web app! From setting up a MySQL database, to creating models, blueprints (with forms and views), templates, custom error pages, tests, and finally deploying the app on PythonAnywhere, you now have a strong foundation in web development with Flask. I hope this has been as fun and educational for you as it has for me! I'm looking forward to hearing about your experiences in the comments below.

Like this article? [Follow @mbithenzomo on Twitter](#)