InfrasCloudy Flask Base Documentation

Release 1.0

Allan Swanepoel

Feb 20, 2019

Contents:

CHAPTER 1

Home

1.1 Source Code

See the Github repo

1.2 Purpose

Getting a decent flask base / boilerplate up and running quickly with some sane defaults

1.3 Synopsis

A Flask application template with the boilerplate code already done for you.

1.4 What's Included?

- Blueprints
- User and permissions management
- Flask-SQLAlchemy for databases
- Flask-WTF for forms
- Flask-Assets for asset management and SCSS compilation
- Flask-Mail for sending emails
- gzip compression
- gulp autoreload for quick static page debugging

1.5 Formatting code

Before you submit changes to flask-base, you may want to auto format your code with `python manage.py format`.

1.6 Contributing

See the Github repository

1.7 License

MIT License

CHAPTER 2

Setting up

2.1 Clone the repo

```
$ git clone https://github.com/infrascloudy/flask-base.git
$ cd flask-base
```

2.2 Initialize a virtualenv

```
$ pip install virtualenv
$ virtualenv env
$ source env/bin/activate
```

2.3 (If you're on a mac) Make sure xcode tools are installed

```
$ xcode-select --install
```

2.4 Add Environment Variables

Create a file called `.env` that contains environment variables in the following syntax:
`ENVIRONMENT_VARIABLE=value`. For example,
the mailing environment variables can be set as the following

```
MAIL_NAME = 'My Visible Name'
MAIL_ADDRESS = 'no-reply@example.com'
SECRET_KEY=SuperRandomStringToBeUsedForEncryption
```

Note: do not include the ‘‘.env‘‘ file in any commits. This should remain private.

2.5 Install the dependencies

```
$ pip install -r requirements/common.txt
$ pip install -r requirements/dev.txt
```

2.6 Other dependencies for running locally

You need to install Foreman and Redis. Chances are, these commands will work:

```
$ gem install foreman
```

Mac (using homebrew):

```
$ brew install redis
```

Linux:

```
$ sudo apt-get install redis-server
```

2.7 Create the database

```
$ python manage.py recreate_db
```

2.8 Other setup (e.g. creating roles in database)

```
$ python manage.py setup_dev
```

Note that this will create an admin user with email and password specified by the `ADMIN_EMAIL` and `ADMIN_PASSWORD` config variables. If not specified, they are both `flask-base-admin@example.com` and `password` respectively.

2.9 [Optional] Add fake data to the database

```
$ python manage.py add_fake_data
```

2.10 [Optional. Only valid on `gulp-static-watcher` branch] Use gulp to live compile your files

- Install the Live Reload browser plugin from here
- Run `npm install`
- Run `gulp`

2.11 Running the app

```
$ source env/bin/activate
$ foreman start -f Local
```

CHAPTER 3

Manage.py and Commands

3.1 *python manage.py runserver*

A note about python manage.py runserver. Runserver is actually located in flask_script. Since we have not specified a runserver command, it defaults to flask_script's Server() method which calls the native flask method app.run(). You can pass in some arguemnts such as changing the port on which the server is run.

3.2 *.env*

The following code block will look for a '.env' file which contains environment variables for things like email address and any other env vars. The .env file will be parsed and santized. Each line contains some "NAME=VALUE" pair. Split this and then store var[0] = "NAME" and var[1] = "VALUE". Then formally set the environment variable in the last line of this block. Per our running example, os.environ["NAME"] = "VALUE" These environment variables can be accessed with "os.getenv('KEY')"

```
if os.path.exists('.env'):
    print('Importing environment from .env file')
    for line in open('.env'):
        var = line.strip().split('=')
        if len(var) == 2:
            os.environ[var[0]] = var[1]
```

3.3 Config and *create_app*

Refer to *manage.py* for more details

```
app = create_app(os.getenv('FLASK_CONFIG') or 'default')
manager = Manager(app)
migrate = Migrate(app, db)
```

Currently the application will look for an environment variable called *FLASK_CONFIG* or it will move to the 'default' configuration which is the DevelopmentConfig (again see *manage.py* for full details). Next it will call the *create_app* method found in *app/__init__.py*. This method takes in a name of a configuration and finds the configuration settings in config.py. In heroku this will be set to 'production' i.e. ProductionConfig.

Next a *Manager* instance is created. Manager is basically an extention that will allow us to get some useful feedback when we call *manage.py* from the command line. It also handles all the *manage.py* commands. The *@manager.command* and *@manager.option(...)* decorators are used to determine what the help output should be on the terminal. Migrate is used to make migration between db instances really easy. Additionally *@manager.command* creates an application context for use of plugins that are usually tied to the app.

3.4 Make Shell Context

```
def make_shell_contect():
    return dict(app=app, db=db, User=User, Role=Role)

manager.add_command('shell', Shell(make_context=make_shell_context))
manager.add_command('db', MigrateCommand)
```

Make shell context doesn't really serve a ton of purpose in most of our development at InfrasCloudy. However, it is entirely possible to explore the database from the command line with this as seen in the lines above.

It is possible to create a general app shell or database specific shell. For example doing 'python manage.py shell'

```
$ me = User()
$ db.sesion.add(me) && db.session.commit()
$ me.id
```

This basically creates a new user object, commits it to the database gives it a id. The db specific shell exposes the native MigrateCommands... honestly you won't have to worry about these and future info can be found the Flask-Migrate documentation.

3.5 Recreate DB

```
@manager.command
def recreate_db():
    """
    Recreates the local database.
    YOU SHOULD NOT USE THIS IN PRODUCTION.
    """
    db.drop_all()
    db.create_all()
    db.session.commit()
```

So this will clear out all the user data (drop_all), will create a new database but with all the tables and columns set up per your models. create_all() and drop_all() rely upon the fact that you have imported ** ALL YOUR DATABASE MODELS **. If you are seeing some table not being created this is the most likely culprit.

3.6 Run Worker + Redis

The run_worker command will initialize a task queue. This is basically a list of operations stored in memory that the server will get around to doing eventually. This is great for doing asynchronous tasks. The memory store used for holding these tasks is called Redis. We set up a default redis password and then open a connection to the redis DB. We instantiate a worker and add a queue of items that needs to be processed on that worker.

```
@manager.command
def run_worker():
    """
    Initializes a slim rq task queue.
    """
    listen = ['default']
    conn = Redis(
        host=app.config['RQ_DEFAULT_HOST'],
        port=app.config['RQ_DEFAULT_PORT'],
        db=0,
        password=app.config['RQ_DEFAULT_PASSWORD']
    )

    with Connection(conn):
        worker = Worker(map(Queue, listen))
        worker.work()
```

3.7 Misc

You may/may not know this but the whole if __name__ == '__main__' check is to see if this file is being executed directly rather than indirectly (by being imported through another file). So when we execute this file directly (by running python manage.py SOMECMD) we get the option of instantiating the manager instance These methods should be accessible from other files though if imported. But you would have a tough time executing these commands from cmd line without the Manager init (otherwise you have to deal with argvs and stuff that is frankly tedious).

CHAPTER 4

Configuration Commands and *config.py*

So lets go through each of the configuring variables.

APP_NAME is the name of the app. This is used in templating to make sure that all the pages at least have the same html title

SECRET_KEY is a alpha-numeric string that is used for crypto related things in some parts of the application. Set it as an environment variable or default to our insecure one. This is used in password hashing see app/models/user.py for more info. YOU SHOULD SET THIS AS A CONFIG VAR IN PRODUCTION!!!!

SQLALCHEMY_COMMIT_ON_TEARDOWN is used to auto-commit any sessions that are open at the end of the 'app context' or basically the current request on the application. But it is best practice to go ahead and commit after any db.session is created

SSL_DISABLE is a boolean to used to enable adhoc ssl certificates (Self-signed) within the application. The next version of the flask base would accomodate specificying certificates as files

MAIL... is used for sending emails using MailGun. This is further described in email.py.

CHAPTER 5

__init__.py

5.1 CSRF Protection

Note about CSRF protection. This basically prevents hackers from being able to post to our POST routes without having actually loaded a form on our website. E.g. they could potentially create users if they found out the URL for our register routes and the params we expect (its fairly easy to do). But with CSRF protection, all forms have a hidden field that is verified on our end. This is a bit low level, but there is a SESSION object stored on the flask server in memory. Each user has their own session containing things like their username, user id, etc When a form created, a random string called a CSRF token is created and is sent along with the form in a hidden field. Simultaneously, this string is added to the user session stored on the server. When the user submits a form, then the server will check to see if the hidden form field with the CSRF token matches the CSRF token stored in the user's session on the server. If it does, then everything is fine and the POST request can proceed normally. If not, then the POST request is aborted as a 403 (i think) error is thrown... basically the user is not able to POST. This is great for forms, but if you want to create a public API that does not require a session, then you'll want to include a decorator on your route *@csrf.exempt*

5.2 Flask-Login

```
login_manager = LoginManager()
login_manager.session_protection = 'strong'
login_manager.login_view = 'account.login'
```

Flask-login provides us with a bunch of easy ways to do secure and simple login techniques. LoginManager() is the main class that will handle all of this. Session protection makes sure the user session is very secure and login_manager.login_view is the view that a non-authenticated user will get redirected to. Otherwise it is a 401 error.

5.3 *init_app(app)*

```
mail.init_app(app)
db.init_app(app)
login_manager.init_app(app)
csrf.init_app(app)
compress.init_app(app)
RQ(app)
```

init_app(app) are methods in each of these packages. It binds each instance of the respective application to the flask app. However, we do need to specify an application context while using things like db, mail, login_manager, and compress since they are not bound to our application _exclusively_.

5.4 Set up Asset Pipeline

This one is a bit complex. First an Environment instance is created that holds references to a single path to the 'static' folder. We don't really care about that since the url_for() method allows us to specify access to resources in the static/ directory. But we then append all the folders and files within the 'dirs' array to the environment. This action provides context for the subsequence set of register actions. Looking in app/assets.py there are some Bundle instances created with 3 parameters mainly: what type of file(s) to bundle, a type of filter/ transpiler to apply, and then a final output file. E.g. for the app_css bundle, it looks within assets/styles, assets/scripts for any *.scss files, converts them to css with the scss transpiler and then outputs it to the styles/app.css file. See the templates/partials/_head.html file for more information on how to actually include the file.

5.5 Blueprints

```
from account import account as account_blueprint
from admin import admin as admin_blueprint
from main import main as main_blueprint

app.register_blueprint(main_blueprint)
app.register_blueprint(account_blueprint, url_prefix='/account')
app.register_blueprint(admin_blueprint, url_prefix='/admin')
```

Blueprints allow us to set up url prefixes for routes contained within the views file of each of the divisions we specify to be registered with a blueprint. Blueprints are meant to distinguish between the variable different bodies within a large application. In the case of flask-base, we have 'main', 'account', and 'admin' sections. The 'main' section contains error handling and views. The other sections contain mainly just views. The folders for each of these sections also contain an __init__ file which actually creates the Blueprint itself with a name and a default __name__ param as well. After that, the views file and any other files that depend upon the blueprint are imported and can use the variable name assigned to the blueprint to reference things like decorators for routes. e.g. if my blueprint is name 'first_component', I would use the following as a decorator for my routes '@first_component.route'. By specifying the url_prefix, all of the functions and routes etc of the blueprint will be read with the base url_prefix specified. E.g. if I wanted to access the '/blah' route within the 'acount' blueprint, I need only specify @account.router('/blah') def ... as my method in views.py under the account/ directory. But I would be able to access it in the browser with yourdomain.com/accounts/blah

A note on why we are importing here: Because stuff will break... and for a good reason! The account import in turn imports the views.py file under the account/ directory. The views.py in turn references db db is the database instance which was created after the import statements If we had included these import statements at the very top, views.py under account would have refered to a db instance which was not created! hence errors... all the errors (at least in files relying upon a created db instance... and any instance created beyond that.

CHAPTER 6

Assets

(refer to *flask-base/app/assets.py*)

See app/__init__.py for details on this

context is set as the assets/styles and assetts/scripts folders

filter = scss -> convers .scss to css filter = jsmin -> converts to minified

javascript

Bundle is just the plugin that helps us do this task.

6.1 Decorators

```
def permission_required(permission):
    """Restrict a view to users with the given permission."""
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            if not current_user.can(permission):
                abort(403)
            return f(*args, **kwargs)
        return decorated_function
    return decorator
```

This is a rather complicated function, but the general idea is that it will allow is to create a decorator that will kick users to a 403 page if they dont have a certain permission or let them continue. First there is a permission_required method which takes in a permission e.g. Permission.ADMINISTER Then it create a function called 'decorator' which performs the check in a separate function itself decorates called 'decorated_function'. It returns the result from 'decorate_function' as well as the results from a specified parameter f that serves as an extra function call. The @wraps(f) decorator is itself used to give context for the decorated function and actually point that context towards the fully decorated function when the permission_required() decorator is invoked. Tl;dr it does some complicated stuff you don't really need to know about

6.2 *@admin_required*

```
def admin_required(f):
    return permission_required(Permission.ADMINISTER)(f)
```

This is a decorator created by the permission required decorator It checks if the current_user is an admin or not. It takes in a function f as the next action to occur after the check happens however in practice, we only use the decorator @admin_required on routes.

CHAPTER 7

Models

7.1 `Permission` class

```
GENERAL = 0x01
ADMINISTER = 0xff
```

Okay so here is a seemingly simple piece of code I really think is
really cool! First of all we are setting up two enums here.
But they are set to weird hexadecimal numbers 0x01 and 0xff.
If you stick these into a hexadecimal -> decimal converter
you'll find that they represent 1 and 255 respectively. But
in binary they come out to 00000001 and 11111111 (8 ones).
If we do a binary and (&) on these two numbers, we can
actually get some unique properties from these.
So if we do GENERAL & ADMINSTER, it will come out to the following

```
  00000001
& 11111111
----------
  00000001
```

We get back the exact same value as GENERAL! Similarly if we do
ADMINSTER & GENERAL we get back GENERAL. This is useful for
checking user roles and who is exactly who in this system.
So we can create a method 'check(input, checker)' that will
take an input hex to test and one to text against. We only need

to do '(input & checker) == checker'. But there are some more
interesting applications for this. Let us define, for example,
a set of enums CAN_LIKE = 0x01, CAN_POST = 0x02, CAN_EDIT = 0x04
and CAN_REMOVE = 0x08. These are respectively in binary 00000001,
00000010, 00000100, 00001000. We can use binary OR (|) to create
composite user permissions e.g. CAN_LIKE | CAN_POST | CAN_EDIT =
0x07 = 00000111 -> NEW_ROLE. We can run 'check(NEW_ROLE, CAN_LIKE)'
or 'check(NEW_ROLE, CAN_POST)' or 'check(NEW_ROLE, CAN_EDIT)' and
all of these will return True.
For example NEW_ROLE & CAN_EDIT

```
  00000111
& 00000001
----------
  00000001 <- equivalent to CAN_EDIT enum
```

A function similar to the check described above can be found in
as the 'can' method below in the User class. Moving on!

7.2 `Role` class

The Role class instatiates Role model. This is used for the
creation of users such as a general user and an administrator

7.2.1 COLUMN DEFINITIONS:

`id` serves as the primary key (expects int).

`name` is the name of the role itself (expects unique String len 64)

`index` is the name of the index route for the route

`default` is a T/F value that determines whether a new user created
has that permission or note (ref insert_roles()). This is indexed
meaning that a separate table has been created with default as the
first column and id as the second column. Default in this table
is sorted and a query for default performs a binary search rather
than a linear search (reduces search time complexity from `O(N)` to
`O(log n)`

`permissions` contains the permissions enum (see Permissions class)

`users` is not a column but it sets up a database relation. This case
is a one-to-many relationship in that for ONE Role record, there are

MANY associated User objects. The `backref` param specifies a
bi-directional relationship between the two tables in that there is
a new property on both a given Role and User object. E.g. Role.users
will refer to the User object (i.e. the user table). and User.role
(role being the string specified with backref) will refer to the
Role object. Lazy = dynamic specifies to return a Query object
instead of actually asking the relationship to load all of its child
elements upon creating the relationship. It is best practice to
include lazy=dynamic upon the establishment of a relationship.

7.2.2 Sub-note on lazy-dynamic and backref:

Currently, lazy-dynamic will
make the User collection to be loaded in as a Query object (so not
everything is loaded at once). Simiarly (as mentioned above), the
User object can reference the Role object by doing User.role however,
this uses the default collection loading behavior (i.e. load the entire
collection at once). It is fine in this case since the amount of
Roles in the Role collection will be *much* less than the amount of
entries in the User collection. However, we can specify that User.role
uses the lazy-dynamic loading scheme. Simply redefine users here to

```
users = db.relationship('User', backref=db.backref('role',
                                    lazy='dynamic'), lazy='dynamic')
```

7.2.3 insert_roles() and SQLAlchemy Sessions

The staticmethod decorator specifies that insert_roles() must be
be called with a instance of the Role class. E.g. role_obj.insert_roles()
This method is fairly self-explanatory. It specifies a 'roles' dict
This is then iterated through and foreach role in the 'roles' dict
we check to see if it already exists (by name) in the Role object
i.e. the Roles table. If not, then a new Role object is instantiated
After that, the perms, index, default props are set and the the
role object is now added to the db session and then committed.

A note about sqlalchemy if you haven't noticed already: All changes
are added to a Session object (handled by SQLAlchemy). Unless specified
otherwise, the session object has a merge operation that finds the difs
between the new object (that was created and added to the session object)
and the currently existing (corresponding) object existing in the table
right now. Then a commit() propegates these changes into the database
making as little changes as possible (i.e. every time we update a
record, the record's attribute is changed 'in place' rather than being
deleted and then replaced. Neat :)

7.2.4 __repr__

```
def __repr__(self):
    return '<Role \'%s\'>' % self.name
```

this **repr** method is pretty much optional, but it is helpful in that
it will allow the program to pretty print the user object when you come
across an error

7.3 User Model

The class User represents users... it extends db.Model and
UserMixin. Per the flask-login documentation, the User class
needs to implement is_authenticated (returns True if the user is
authenticated and in turn fulfill login_required), is_active
(returns True if the user has been activated i.e. confirmed by
email in our case), is_anonymous (returns if a user is Anonymous
i.e. is_active = is_authenticated = False, is_anonymous = True,
and get_id() = None), get_id() (returns a UNICODE that has the
id of the user NOT an int).

7.3.1 Column Descriptions:

`id` - primary key for the table. Id of the user. i.e. the
unique identifier for the collection

`confirmed` - boolean val (default value = False) that is
an indication of whether the user has confirmed their
account via email.

`first_name` - ... string self explanatory

`last_name` - ... string self explanatory

`email` - string self explanatory. But we impose the uniqueness
constraint on this column. It is necessary to check for this
on the backend before entering an email into the table,
else there will be some nasty errors produced when the user
tries to add an existing email into the table.

Note: first_name, last_name, email form an index table for easy lookup. See Role for more info

`password_hash` is a 128 char long string containing the hashed password. As always, it is best practice to never include the plaintext password on the server. This hashed password is checked against when authenticating users.

`role_id` is the id of the role the user is. It is a foreign key and relates to the id's in the Role collection. By default the general user is role.id = 1, and role.id = 2 is the admin. Also note that we refer to the Role collection with 'roles' rather than the assigned backref 'role' since we are referring to an individual column.

7.4 Other User Class Variables and Methods

Note that the following methods are actually available in your Jinja templates since they are attached to the user instance.

`full_name` provides the full name of the user given a first and last name

`can` provides a really cool way of determining whether a user has given permissions. See the Permissions class for more info. `is_admin` is an implementation of `can` to test a user against admin permissions.

`password` This does not give a password if a user just calls the method and throws an AttributeError. However if someone chooses to set a password e.g. u = User(password = `test`) the second definition of password method is run, taking the keyword arg (kwarg) as the password to then call the generate_password_hash method and set the password_hash property of the user to the generated password.

`verify_password` well... verifies a provided user plaintext password against the password_hash in the user record. Uses the check_password_hash method.

`generate_confirmation_token` returns a cryptographically signed string with encrypted user id under key `confirm`. This will expire in 7 days. Note that Serializer is actually TimedJSONWebSerializer when looking for documentation.

`generate_changed_email_token` also returns a cryptographically signed string with encrypted user id under key `change_email` and a encrypted new_email parameter password into the method containing the desired new email the user wants to replace the old email with.

`generate_password_reset_token` operates similarly to `generate_ confirmation_token`. Generates token for password reset

NOTE: For context, the generate_. . . _token methods are used to create a random string that will be later added to an email (usually) to the requesting user.

7.4.1 `confirm_account`

The confirm_account method will take in a token (which was presumably generated from the generate_confirmation_token method) and then return True if the provided token is valid (and can be decrypted with the SECRET_KEY and has not expired) AND the decrypted token has the key 'confirm' with the id of the requesting user. If so, it flips the 'confirmed' attribute of the requesting user to True.
Will throw BadSignature of the token is invalid, will throw SignatureExpired if the token is past the expiration time.

7.4.2 `change_email`

The change_email method will take in a token (which was presumably generated from the generate_email_token method) and then return True True if the token is valid (see above method for explanation of 'valid') and contains the key 'change_email' with value = user id in addition to the key 'new_email' with the new email address the user wants to change their email to. Before the new_email is committed to the session, a query is performed on the User collection on all the emails to maintain the unique constraint on the email columns. Then the user's 'email' attribute is set to the 'new_email' specified in the decrypted token.
will throw BadSignature if invalid token and SignatureExpired if the token is expired.

7.5 AnonymousUser

We define a custom AnonymousUser class that represents a non-logged user. It extends the AnonymousUserMixing provided by flask-loginmanager we deny all permissions and affirm that this user is not an admin

```
class AnonymousUser(AnonymousUserMixin):
    def can(self, _):
        return False

    def is_admin(self):
        return False
```

```
login_manager.anonymous_user = AnonymousUser
```

We then register our custom AnonymousUser class as the default login_manager
anonymous user class

```
@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))
```

This is the default user_loader method for login_manager. This method
defines how to query for a user given a user_id from the user SESSION object.
It is pretty straightforward, it will query the User table and find the user
with ID equal to the user_id provided in the user SESSION

CHAPTER 8

Routing (Account Routes)

This guide will be explaining the concept of routing by going through a file. We will be using *app/account/views.py*

8.1 Login

```
@account.route('/login', methods=['GET', 'POST'])
def login():
    """Log in an existing user."""
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user is not None and user.password_hash is not None and \
                user.verify_password(form.password.data):
            login_user(user, form.remember_me.data)
            flash('You are now logged in. Welcome back!', 'success')
            return redirect(request.args.get('next') or url_for('main.index'))
        else:
            flash('Invalid email or password.', 'form-error')
    return render_template('account/login.html', form=form)
```

All routes are decorated with the name of the associated Blueprint along with the .route prop with attributes of (name, methods=[]). For example *@account.route('/login', method=['GET', 'POST'])* creates a route accessible at *yourdomain.com/account/login*.

This route can accept either *POST* or *GET* requests which is appropriate since there is a form associated with the login process. This form is loaded from the forms.py file (in this case the *LoginForm()* is loaded) and we then check if the form is valid (*validate_on_submit*) in that it is a valid POST request. We grab the form field named 'email' and query the User database for the user that has that email. Then we call the *verify_password* method from the User class for this specific user instance and check the hashed password in the database against the password provided by the user which is hashed with the SECRET_KEY. If everything is fine, the Flask-login extendion performs a login_user action and sets the *SESSION['user_id']* equivalent to the user id provided from the user instance. If the form has remember_me set to True (ie checked) then that is passed along as a parameter in login_user.

If it was redirected to this /login page, their URL will have a parameter called *next* containing the URL they need to be directed to after they login. Otherwise, they will just be sent to the main.index route This is true for the admin as well. It is best to edit this functionality since index pages should differ by user type. There is a flash sent as well if the request is successful.

If there is an error in the user checking process, then the user is kicked back to the account/login page with a flashed form error.

If this is a GET request, only the account/login page is rendered

8.2 Logout

```
@account.route('/logout')
@login_required
def logout():
    logout_user()
    flash('You have been logged out.', 'info')
    return redirect(url_for('main.index'))
```

The Flask-login Manager has a built in logout_user function that removes the SESSION variables from the user's browser and logs out the user completely

CHAPTER 9

Templating

This will cover various methods used in our jinja templates.

9.1 Base.html

{% import 'macros/nav_macros.html' as nav %}

```
<!DOCTYPE html>
<html>
    <head>
        {% include 'partials/_head.html' %}
        {# Any templates that extend this template can set custom_head_tags to add␣
↪scripts to their page #}
        {% block custom_head_tags %}{% endblock %}
    </head>
    <body>
      {# Example dropdown menu setup. Uncomment lines to view
        {% set dropdown =
          [
            ('account stuff',
              [
                ('account.login', 'login', 'sign in'),
                ('account.logout', 'logout', 'sign out'),
                ('2nd drop', [
                  ('account.login', 'login 2', ''),
                  ('3rd drop', [
                    ('main.index', 'home 2', '')
                  ])
                ])
              ]
            ),
            ('main.index', 'home 1', 'home')
          ]
```

(continues on next page)

(continued from previous page)

```
        %}
      #}

        {% block nav %}
          {# add dropdown variable here to the render_nav method to render dropdowns
↪#}
          {{ nav.render_nav(current_user) }}
        {% endblock %}

        {% include 'partials/_flashes.html' %}
        {# When extended, the content block contains all the html of the webpage #}
        {% block content %}
        {% endblock %}

        {# Implement CSRF protection for site #}
        {% if csrf_token() %}
            <div style="visibility: hidden; display: none">
                <input type="hidden" name="csrf_token" value="{{ csrf_token() }}">
            </div>
        {% endif %}
    </body>
</html>
```

9.2 Macros: Password Strength (`check_password.html`)

Refer to `app/templates/macros/check_password.html`

This uses the zcvbn password checker to check the entropy of the password provided in the password field. Given a specified field, the password checker will check the entropy of the field and disable the submit button until the give 'level' is surpassed

9.3 Macros: Form rendering (`render_form`)

```
{% macro render_form(form, method='POST', extra_classes='', enctype=None) %}
    {% set flashes = {
        'error':   get_flashed_messages(category_filter=['form-error']),
        'warning': get_flashed_messages(category_filter=['form-check-email']),
        'info':    get_flashed_messages(category_filter=['form-info']),
        'success': get_flashed_messages(category_filter=['form-success'])
    } %}

    {{ begin_form(form, flashes, method=method, extra_classes=extra_classes,␣
↪enctype=enctype) }}
        {% for field in form if not (is_hidden_field(field) or field.type ==
↪'SubmitField') %}
            {{ render_form_field(field) }}
        {% endfor %}

        {{ form_message(flashes['error'], header='Something went wrong.', class='error
↪') }}
```

(continues on next page)

(continued from previous page)

```
        {{ form_message(flashes['warning'], header='Check your email.', class='warning
↪') }}
        {{ form_message(flashes['info'], header='Information', class='info') }}
        {{ form_message(flashes['success'], header='Success!', class='success') }}

        {% for field in form | selectattr('type', 'equalto', 'SubmitField') %}
            {{ render_form_field(field) }}
        {% endfor %}
    {{ end_form(form) }}
{% endmacro %}
```

Render a flask.ext.wtforms.Form object.

```
Parameters:
    form          - The form to output.
    method        - <form> method attribute (default 'POST')
    extra_classes - The classes to add to the <form>.
    enctype       - <form> enctype attribute. If None, will automatically be set to
                    multipart/form-data if a FileField is present in the form.
```

Render Form renders a form object. It calls the begin form macro. Initially a 'flashes' variable is set with 'error', 'warning', 'info', 'success' which have values gathered from the get_flashed_messages method from flask. Note that all flashes are stored in SESSIOn with a category type. For most of our purposes, we only have form-error and form-success as our flash types (the second parameter in the flash function call seen in the views.

Then the begin_form macro is called and for each form field in the provided form render_form_field macro is called with the field.
All hidden fields (i.e. the CSRF field) and all submit fields is not rendered at this fime in render_form_field. In the render_form_field method, render_form_input is called for each input in the form field.

After that, the form_message macro is called with each of the flash types.

Lastly, the submit field is rendered. And the form is closed with the end_form macro

9.4 Macros: Start Form (`begin_form`)

Set up the form, including hidden fields and error states.
begin_form is called from render_form. First a check is performed to check if there exists a field within the form with type equal to FileField. This check is performed via filter ("l") in Jinja. This initial check produces a filtered object, the 'list' filter creates a iterable list which we can then check the length of with 'length > 0'. So if this check passes, then the enctype

must be set to multipart/form-data to accomodate a file upload. Otherwise, there is no enctype.

Then the form tag is created with a method default of POST, enctype decided by the check explained above. If there are errors (by field specific validator errors or if the flashes.error, flashes.warning, flashes.info, flashes.success is not None, then that class is added to the overall class of the form (along with any specified extra_classes, default = ‘’).

Lastly the hidden_tags are rendered. WTForms includes in this method the rendering of the hidden CSRF field. We don’t have to worry about that.

Example output:

```
<form action="" method="POST" enctype="multipart/form-data" class="ui form">
  <div style="display:none;">
    <input id="csrf_token" name="csrf_token" type="hidden" value="SOME_CSRF_TOKEN_HERE
↪">
```

9.5 Macros: Flash message to Form (`form_message`)

Render a message for the form. This is called from the render_form macro.

Recall the get_flashed_messages method. It will get the flash message from the SESSION object with a given cateogory_filter. Within the render_form macro, the flashes variable is set with attributes ‘errors’, ‘success’, ‘info’, and ‘warning’. The messages parameter for form_message contains the flash messages for the respective attribute specified in flashes[‘some_attr’].

The form_message macro is called after all form fields have been rendered, except for the Submit field. A div is created with class= ‘ui CLASS message’ class being either error, success, info, or warning. This div is only created if there are messages for a given flashes type! For each of the messages in the flashes type, the message is filtered to only contain escaped HTML chars and appended within the div ul as a list element.

Example Output:

```
<div class="ui error message">
  <div class="header">Something went wrong.</div>
  <ul class="list">
    <li>Invalid email or password.</li>
  </ul>
</div>
```

9.6 Macros: Render a form field (`render_form_field`)

Render a field for the form. This is rather self explanatory.
If the field is
a radio field (RadioField WTForms object) extra_classes has an added class of
'grouped fields' since all the options of a Radio Field must be styled in this
way to display together.
If there is a validation error on the form field, a error class is added to the
field div (to make the field colored red). Then the render_form_input macro is
called with field object itself as a parameter. Any validation errors are then
added with a sub-dev with content field.errors (we only show the first validation
error for the given error for simplicity) and filter for HTML safe chars.

9.7 Partials: `_flashes`

See the macros/form_macros for extended explanation of the
get_flashed_messages(category_filter) method. This macro renders
general flash methods that appear at the top of the page. We render
by flash type and create a separate 'ui {{ class }} message' div
for each message within a specific flash type. Error = red,
warning = yellow, info = blue, success = green.

9.8 Partials: `_head`

This method contains all the assett imports (i.e. imports for scripts and styles for the app)
Note that the asssets will be contained in the static/webassets-external folder when the app
is in debug mode.

CHAPTER 10

Deployment

The aim of this guide is to walk you through launching our basic *flask-base* repository found [here](https://github.com/infrascloudy/flask-base) and will also cover some common situations and issues encountered from previous projects.

10.1 What is Heroku and Why are we using it?

To get started we are going to cover what heroku is and how to set it up.

Just a little bit of background. Currently, when you run your app with *python manage.py runserver* or *foreman start -f Local* you are running on your computer only (on something like *localhost:5000*). Of course this means that if anyone tries to access your application, they will be stuck with a *404 not found* error. Thus we must put your application onto a publicly accessible computer that is constantly running. This is exactly what a server does. When you type in something like *linaccess.za.net*, a request is first sent to a Domain Name Server or _DNS_ which then maps the domain name *linaccess.za.net* to an IP Address which points to the server which then renders pages and serves them over to you, the client. Seems simple. But how do you get a server?

Heroku is the answer. The heroku platform is a cloud platform that runs your apps in containers called **dynos** and hosts these apps for free (...ish, we'll get to pricing later). These dynos can host apps and allow you to scale the applications infinitely (at a cost of course) to handle more traffic. Additionally, the heroku dynos contain all the code you need to run a python app from the get go and will install any pip dependencies. Your app lives in a remote git repository on heroku's servers. When you push to the remote heroku repository, heroku will merge the changes, reset your server, and run the new version of your app. Heroku makes this entire process seamless, so its super easy to maintain your app well after it has been launched.

Now that we have a good understanding of what heroku is and why we want to use it. Let's get started with launching the application to heroku!

10.2 Basic Setup: Heroku Account and CLI Installation

Head over to https://signup.heroku.com to set up an account. Once you are set up, confirm your email and set up your password.

Next, install the heroku command line interface (CLI) for your operating system at https://devcenter.heroku.com/articles/heroku-cli.

10.3 Heroku Dyno Creation and Initial Setup

Go to the directory containing the application you wish to launch. For demo purposes, we will be using the *flask-base* repository which you can clone from https://github.com/infrascloudy/flask-base. This is a python application that has a SQLite database and a Redis Task Queue.

Go to your terminal and type in *heroku login*. If you have set up everything correctly with the CLI installation in the previous section, you should be prompted for your Heroku account credentials (from the previous section as well).

```
$ heroku login
Enter your Heroku credentials.
Email: admin@example.com
Password (typing will be hidden):
Authentication successful.
```

Before creating a heroku dyno, make sure you are at the root directory of your application. Next make sure your application is a git repository (you can do *git init* to make it one), and make sure the current git branch you are on is **master** since heroku only pushes changes from that branch. Also make sure that your *requirements.txt* file contains all the pip modules to work (you can do *pip freeze > requirements.txt* to place all your installed pip modules in *requirements.txt*).

To create the dyno, run in the terminal *heroku create <app-name>*.

Note that I use ''<variable>'' to indicate that the variable is optional and the carats should be excluded. E.g. a valid interpretation of the above would be ''heroku create'' or ''heroku create myappname'' but NOT ''heroku create <myappname>''.

Heroku will create an empty dyno with name you specified with `app-name` or a random name which it will output to the terminal.

```
$ heroku create flask-base-demo
Creating  flask-base-demo... done
https://flask-base-demo.herokuapp.com/ | https://git.heroku.com/flask-base-demo.git
```

Your application will be accessible at _https://**flask-base-demo.herokuapp.com**_ (per the example above) and the remote github repository you push your code to is at `https://git.heroku.com/flask-base-demo.git`.

Next we can run `git push heroku master`. This will push all your existing code to the heroku repository. Additionally, heroku will run commands found in your `Procfile` which has the following contents:

This specifies that there is will be a `web` dyno (a server that serves pages to clients) and a `worker` dyno (in the case of flask-base, a server that handles methods equeued to the Redis task queue).

If all goes well, you should see an output something similar to this:

```
Counting objects: 822, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (339/339), done.
Writing objects: 100% (822/822), 1.12 MiB | 914.00 KiB/s, done.
Total 822 (delta 457), reused 822 (delta 457)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Python app detected
```

(continues on next page)

(continued from previous page)

```
remote: -----> Installing python-2.7.13
remote:      $ pip install -r requirements.txt
remote:        Collecting Flask==0.10.1 (from -r /tmp/.../requirements.txt (line 1))
...
...
...
remote:        Successfully installed Faker-0.7.3 Flask-0.10.1 Flask-Assets-0.10␣
↪Flask-Compress-1.2.1 Flask-Login-0.2.11 Flask-Mail-0.9.1 Flask-Migrate-1.4.0 Flask-
↪RQ-0.2 Flask-SQLAlchemy-2.0 Flask-SSLify-0.1.5 Flask-Script-2.0.5 Flask-WTF-0.11␣
↪Jinja2-2.7.3 Mako-1.0.1 MarkupSafe-0.23 SQLAlchemy-1.0.6 WTForms-2.0.2 Werkzeug-0.
↪10.4 alembic-0.7.6 blinker-1.3 click-6.6 gunicorn-19.3.0 ipaddress-1.0.17␣
↪itsdangerous-0.24 jsmin-2.1.6 jsonpickle-0.9.2 psycopg2-2.6.1 python-dateutil-2.6.0␣
↪raygun4py-3.0.2 redis-2.10.5 rq-0.5.6 six-1.10.0 webassets-0.10.1
remote:
remote: -----> Discovering process types
remote:        Procfile declares types -> web, worker
remote:
remote: -----> Compressing...
remote:        Done: 43.7M
remote: -----> Launching...
remote:        Released v4
remote:        https://flask-base-demo.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/flask-base-demo.git
 * [new branch]      master -> master
```

10.4 Configuration

Next we have to set up some configuration variables to ensure that the application will be in production mode.

From the command line run

```
heroku config:set FLASK_CONFIG=production
```

Also set your Mailgun credentials as configuration variables as well (if you want the application to send email) The MAIL_DOMAIN is the last segment of the MailGun API Base URL Ex: If API Base URL is https://api.mailgun.net/v3/mg.example.com then MAIL_DOMAIN would be mg.example.com

MAIL_KEY is your MailGun Api Key

```
heroku config:set MAIL_NAME=yourVisableName MAIL_ADDRESS=no-reply@example.com MAIL_
↪DOMAIN=mg.example.com MAIL_KEY=key-d3adb33fd3adb33f
```

Next you should add a `SECRET_KEY`

```
heroku config:set SECRET_
↪KEY=SuperRandomLongStringToPreventDecryptionWithNumbers123456789
```

And also set, `SSL_DISABLE` to `False`

```
heroku config:set SSL_DISABLE=False
```

If you plan to use redis, go to https://elements.heroku.com/addons/redistogo?app=flask-base-demo and follow the onscreen steps to provision a redis instance.

Also if you have a Raygun API Key, add the config variable `RAYGUN_APIKEY` in a similar fashion to above. This will enable error reporting. See https://raygun.com for more details

10.5 Database Creation & Launching

First run `heroku ps:scale web=1 worker=1`. You may need to add a credit card for this to work (it will notify you on the command line to do that).

Next run `heroku run python manage.py recreate_db` to create your database.

Lastly, run the command to add an admin user for you app. In flask base it will be the following `heroku run python manage.py setup_dev`.

In general if you want to run a command on the app it will be in the format of `heroku run <full command here>`. Additionally you can access the file system with `heroku run bash`.

You can now access your app at the URL from earlier and log in with the default user.

10.6 Domain Name + HTTPS Setup

This guide encompasses all you need to get set up with SSL https://support.cloudflare.com/hc/en-us/articles/205893698-Configure-CloudFlare-and-Heroku-over-HTTPS.

10.7 Debugging

`heroku logs --tail` will open up a running log of anything that happens on your heroku dyno.

Additionally, if you have Raygun configured, you'll get error reports (otherwise, you can look at older versions of flask base where we sent errors to the main administrator email).

Lastly, you can use an application like **'Postico <https://eggerapps.at/postico/>'__** to actually look at your database in production. To get the credentials for the application to work with Postico, do the following:

- Run `heroku config` to print out all configuration variables.
- Find the `DATABASE_URL` variable, it should look something like `postgres://blahblahblah:morerandomstuff123456@ec2-12-345-678-9.compute-1.amazonaws.com:5432/foobar`
- In Postico, click "New Favorite".
- For the fields use the following reference to interpret the parts of the `DATABASE_URL` variable: `postgres://User:Password@Host:Port/Database`
- If you want to view your redis queue, use the following web interface https://www.redsmin.com/ or the command line.

10.8 Heroku considerations, scaling and pricing

If your application uses file uploads, **Heroku does not have a persistent file system**, thus you need to set up a Amazon S3 Bucket to upload your file to. This heroku guide has a nice way to upload files with AJAX on the frontend https://devcenter.heroku.com/articles/s3. You can also view the Reading Terminal Market Repo for an example of how to use file uploads

Heroku has a limit of 30 seconds on processing a request. This means that once a user submits a request to a URL Endpoint, a response must be sent back in 30 seconds, otherwise the request will abort and the user will get a timeout error. You should explore using a Redis queue to process requests in the background if they require more than a few seconds to run. Or you can issue AJAX requests on the frontend to a URL (at least this will just silently fail).

Heroku postgresQL has a limit of about 10k rows. If your application will use more than that, then you should follow this guide.

Also you should upgrade your heroku instance to the `hobby` tier to ensure that it will be working 24 hrs. The free tier will only work 18 hrs a day and will *sleep* the application after 5 minutes if inactive (meaning that it will take a while to start up again from a sleep state). You can change this on the heroku dashboard https://dashboard.heroku.com/apps/.