sbMACRO complete re-do log

Credit: much of the changes were inspired by this fantastic blog series (the Flask Mega Tutorial).

Step 1: Basic Configuration

We'll start by creating a virtual environment with `virtualenv` using `python3`.

In the working directory for the project...

```
    $ virtualenv -p python3 venv
```

Then enter the virtual environemnt with `source env/bin/activate` on Unix.

Now, install flask using `pip`: `(env) $ python -m pip install flask`

Now, after creating an *app* directory, we add an `__init__.py` script to make the directory *app* into a python module.

This script just contains the basics:

```
from flask import Flask

app = Flask(__name__)

from app import routes
```

Now we need a place for all our URL routes and their respective functions to live. So, within the *app* directory, we create a `routes.py` file.

Here's an example of a simple URL route and route handler function that will live there:

```
from app import app

@app.route('/')
@app.route('/index')
def index():
    return "Hello, World!"
```

This is the basics of how we handle each URL: we define the URL/route and a function to call that defines how we handle it.

Now we need a top-level script that defines the Flask application instance. This will not go withing *app*, but within our projects working directory. We'll call it `sbmacro.py`.

```
from app import app
```

It only takes one line. That means that the Flask application instance, called "app" is call and is a member of the "app" package.

Now we need to set te FLASK_APP environmental variable in our virtual environment. Make sure you're in the virtual environment and run the `export FLASK_APP=sbmacro.py` command.

```
    $ export FLASK_APP=sbmacro.py
```

Now, to run the application, just type `flask run` while in the virtual environment and it will start running (this only works if you can use python modules without the `python -m` prefix. Otherwise, use `python -m flask run`). Since we're on a development environment, Flask will use port 5000 and localhost (which is IP address 127.0.0.1). On a server, it will typically liston on port 443 or maybe 80 (if it doesn't use encryption).

Check the site out in a browser by going to `http://localhost:5000/<route>`

Step 2: Setup Templates

You need to have templates set up to display our different routes.

For this we need a *templates* directory in our *app* directory. This will hold a bunch of HTML templates that we will display and adjust for each given route and user.

We can also use the built-in Jinja2 template engine to dynamically fill in parts of your templates that you block off with double braces ("{{...}}"). YOu can also use conditional statements with Jinja within {%...%}.

Be sure to set up your base-template as well, from which all other templates will inherit their look. This makes it easy to keep the look consistent throughout the site.

This base template will include {% block content %} {% endblock %} where you want all your other templates' HTML to go. Then, in those templates, you will have {% extends "base.html %} at the top and {% endblock %} at the bottom. This will cause the base.html page to surround the template's HTML.

Here's an example of using Jinja2 to dynamically display content.

First, the Python script, including passing variables to the HTML and Jinja2: app/routes.py:

```
from flask import render_template
from app import app

@app.route('/')
@app.route('/index')
def index():
```

```
    user = {'username': 'Miguel'}
    posts = [
        {
            'author': {'username': 'John'},
            'body': 'Beautiful day in Portland!'
        },
        {
            'author': {'username': 'Susan'},
            'body': 'The Avengers movie was so cool!'
        }
    ]
    return render_template('index.html', title='Home', user=user,
posts=posts)
```

Notice both variables being passed to the html template through render_template().

Now here's the HTML with Jinja2 implementation:

```
<html>
    <head>
        {% if title %}
        <title>{{ title }} - Microblog</title>
        {% else %}
        <title>Welcome to Microblog</title>
        {% endif %}
    </head>
    <body>
        <h1>Hi, {{ user.username }}!</h1>
        {% for post in posts %}
        <div><p>{{ post.author.username }} says: <b>{{ post.body }}</b>
</p></div>
        {% endfor %}
    </body>
</html>
```

Step 3: Setup Configuration (WTF Forms example)

We're going to use the extention Flask-WTF which is just a wrapper around the WTForms package. It is an easy and more secure way to do web forms.

```
    (venv) $ python -m pip install flask-wtf
```

Now we need to create an easy way to centralize configuration of the various aspects of our Flask app. So we create a `config.py` file in the main directory of the application. In it, we store a Config class object, the start of a configuration class object can be seen here:

`config.py`

```
import os

class Config(object):
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'self-defined-key'
```

This config class can be used to store all our configuration items, and more can be added as needed. We could even have subclasses of it if different configuration schemes are needed for different parts of the site.

The SECRET_KEY above is an important configuration item for Flask apps. It is often used as a cryptographic key, which is used to generate signatures or tokens. WTF uses it to prevent Cross-Site Request Forgery attacks. The self-defined key used above is fine in development, but later we'll define the SECRET_KEY environmental variable to make it more secure.

To make flask use the config object, go back to `app/__init__.py` and add to lines: `from config import Config` and `app.config.from_object(Config)`. The first imports our config object, the second sets the app's configuration using that imported object.

Now, using the configuration and WTForms, we can create a login in a new file `app/forms.py`:

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Remember Me')
    submit = SubmitField('Sign In')
```

Using Jinja2, we can add the form we just created to an HTML page (`app/templates/login.html`):

```
{% extends "base.html" %}

{% block content %}
    <h1>Sign In</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}
        </p>
        <p>{{ form.remember_me() }} {{ form.remember_me.label }}</p>
        <p>{{ form.submit() }}</p>
```

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

In `<form>`, "action" gives the URL the form is to be submitted to (if blank, it is submitted to the current URL), and "method" is used to specify the HTTP request method. "post" is used for user experience, as the default, "get", adds all form fields to the URL, which clutters things up, while "post" can submit the form data in the body of the request.

The form.hidden_tag() template argument generates a hidden field that includes a token that is used to protect the form against CSRF attacks. All you need to do to have the form protected is include this hidden field and have the SECRET_KEY variable defined in the Flask configuration. If you take care of these two things, Flask-WTF does the rest for you.

HTML is generated with the `{{ form.<field_name>.label }}` for labels and `{{ form.<field_name> }}` where you want the field. Additional HTML attributes are passed in as arguments (see `password(size=32` above). This is how you attach CSS classes or IDs to form fields.

Now, if you created a route for the new form and tried to submit it, it wouldn't work, you must override the default of the route to accept both 'GET' and 'POST' requests. The form processing work can also be done with for.validate_on_submit(), which returns True or False if it passed validation

Step 4: Adding Database support and the Database

We want a database, so we need to use the Flask-SQLAlchemy extention, which is a wrapper for the SQLAlchemy package. The package is an Object Relational Mapper or ORM, which allows the app to manage the database using high-level entities such as classes, objects, and methods, instead of tables and SQL.

```
(venv) $ python -m pip install flask-sqlalchemy
```

The database may also need to change and grow as sbMACRO grows, so we should implement an easy workflow for database migrations. For that, we use the extention Flask-Migrate.

```
(venv) $ python -m pip install flask-migrate
```

To use a database, you must add the configuration to the Config object:

```
import os

basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'self-defined-key'
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
```

```
        'sqlite:///' + os.path.join(basedir, 'sbmacro.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Then add some of those config handlers to `app/__init__.py`:

```
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)

from app import routes, models
```

Then, for each model you will create a class object in `app/models.py`. For example, for a user model:

```
from app import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))

    def __repr__(self):
        return '<User {}>'.format(self.username)
```

But this model may change in the future, so we create a migration repository and strategy using Alembic (which is under the hood of Flask-Migrate). Using this, the database won't have to be created from scratch in order to update it.

For Flask-Migrate to do this, it must maintain a migration repository that stores migration scripts. Whenever a change is made, a script is added to the repository. To apply the migration to the db, the scripts are executed in the order in which they were created.

To create the migration repository:

```
(venv) $ python -m flask db init
```

This should create a `migrations` directory with a few files. This is important to backup with the rest of the app. Now let's do our first Database Migration. To create an automatic migration (that compares models to current schema), use `python -m flask db migrate -m <table>`.

```
(venv) $ python -m flask db migrate -m "users table"
```

This generated a migration script, but did not change the database. To change the database, use the `upgrade()` and `downgrade()` functions. `upgrade()` applies the migration, `downgrade()` removes it. Using these you can migrate the db to any point in its history.

We want to apply the changes:

```
(venv) $ python -m flask db upgrade
```

Because we're using SQLite, the `upgrade` command detects if the db exists and creates one if it does not.

Note: Flask-SQLAlchemy uses snake case, so it translates class names to snake case. For example, for a AddressAndPhone model class, the table would be named address_and_phone. To chose a table name yourself, add an attribute named tablename to the model class.

Now we need to know our Database Upgrade and Downgrade workflow. Miguel Grinberg describes it well:

```
Database Upgrade and Downgrade Workflow
The application is in its infancy at this point, but it does not hurt to
discuss what is going to be the database migration strategy going forward.
Imagine that you have your application on your development machine, and
also have a copy deployed to a production server that is online and in
use.

Let's say that for the next release of your app you have to introduce a
change to your models, for example a new table needs to be added. Without
migrations you would need to figure out how to change the schema of your
database, both in your development machine and then again in your server,
and this could be a lot of work.

But with database migration support, after you modify the models in your
application you generate a new migration script (flask db migrate), you
probably review it to make sure the automatic generation did the right
thing, and then apply the changes to your development database (flask db
upgrade). You will add the migration script to source control and commit
it.

When you are ready to release the new version of the application to your
production server, all you need to do is grab the updated version of your
application, which will include the new migration script, and run flask db
upgrade. Alembic will detect that the production database is not updated
to the latest revision of the schema, and run all the new migration
scripts that were created after the previous release.

As I mentioned earlier, you also have a flask db downgrade command, which
```

```
undoes the last migration. While you will be unlikely to need this option
on a production system, you may find it very useful during development.
You may have generated a migration script and applied it, only to find
that the changes that you made are not exactly what you need. In this
case, you can downgrade the database, delete the migration script, and
then generate a new one to replace it.
```

Now let's add relation-ality to our database. You can have 'foreign keys' which point to the id of a specific User or something that will be related to the new thing being described. Here's an example of a User-Post relation in `models.py`:

```
from datetime import datetime
from app import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))
    posts = db.relationship('Post', backref='author', lazy='dynamic')

    def __repr__(self):
        return '<User {}>'.format(self.username)

class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.String(140))
    timestamp = db.Column(db.DateTime, index=True,
default=datetime.utcnow)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))

    def __repr__(self): #tells python how to print a post
        return '<Post {}>'.format(self.body)
```

In this example, the Post class represents blog posdts by users. The `timestamp` field is indexed, so we can retrieve posts in chronological order, with a default that is set to the function utcnow (not the result of the function, which would be 'utcnow()'), which means the default is the result of that function whenever the post is created in the database.

The `user_id` field is a foreign key to `user.id` which is the 'id' key in the User model.

NOTE: db.relationship() uses the model class name, which db.ForeignKey() uses the database table name, which are often different.

For the 'one-to-many' relationship of a user to posts, notice that the User model uses `db.relationship()` in the new `posts` field. So, if you had a user stored in the variable 'karen', if you called `karen.posts`, SQlAlchemy will run a db query that returns all posts written by that user. The first arg is the Model class for the 'many' side of the 'one-to-many' relationship. The `backref` arg is

the name of the field that will be added to the objects of the "many" class that points back at the "one" object. In this way, `post.author` will return the user who created the post.

Since we updated the application models, a new database migration needs to be generated, then applied:

```
(venv) $ python -m flask db migrate -m "posts table"
...
...
...
(venv) $ python -m flask db upgrade
```

~~**The key here is to migrate each model one at a time in highest one-to-many fashion. It will not work if you try to create migration scripts for a model that doesn't have all dependencies: For example, `User` needs done first, THEN you can migrate `Post`, as `Post` has a foreign key dependency on `User`. So, go in order. You also can't migrate twice in a row without updating. So you must migrate AND update for each model individually if they rely on each other for Foreign Keys.**~~ **Actually, just don't use all CAPs for model names.**

Optional: Experiment with the Database

If you want to test the database (for example, the user-post instance above), start the python interpreter from within the vm:

```
(venv) $ python
```

Once inside, import the database instance and the models:

```
>>> from app import db
>>> from app.models import User, Post
```

Then create a new user:

```
>>> u = User(username='john', email='john@example.com')
>>> db.session.add(u)
>>> db.session.commit()
```

Remember that changes to the database are done within sessions, so you must commit() a session for the changes that have happened since to take effect. If there are problems, you can abort the session and reove the changes stored in it with `db.session.rollback()`. You need sessions so that the db is never left in an inconsistent state.

Add another user, then query all users:

```
>>> u = User(username='susan', email='susan@example.com')
>>> db.session.add(u)
>>> db.session.commit()
>>> users = User.query.all()
>>>users
[ User: john, User: susan]
>>>for u in users:
...    print(u.id, u.username)
...
1. john
2. susan
```

So all the models have a `query` attribute that is the entry point to run db queries. The most basic query is the one we just used to get all elements fo that class (`all()`).

If you know the id of a user, then you can do something like:

```
>>> u = User.query.get(1)
>>> u
<User john>
```

Here you can add a post for a user:

```
>>> u = User.query.get(1)
>>> p = Post(body='my first post!', author=u)
>>> db.session.add(p)
>>> db.session.commit()
```

`timestamp` **did not need set because of the default. The** `user_id` **field is populated wiht the** `author` **argument automatically, instead of dealing with user IDs.**

Here are another couple examples:

```
>>> # get all posts written by a user
>>> u = User.query.get(1)
>>> u
<User john>
>>> posts = u.posts.all()
>>> posts
[<Post my first post!>]

>>> # same, but with a user that has no posts
>>> u = User.query.get(2)
>>> u
```

```
<User susan>
>>> u.posts.all()
[]

>>> # print post author and body for all posts
>>> posts = Post.query.all()
>>> for p in posts:
...     print(p.id, p.author.username, p.body)
...
1 john my first post!

# get all users in reverse alphabetical order
>>> User.query.order_by(User.username.desc()).all()
[<User susan>, <User john>]
```

For help with database queries and functions, check out the Flask-SQLAlchemy documentation.

But let's undo what we did before continuing to give ourselves a blank slate.

```
>>> users = User.query.all()
>>> for u in users:
...     db.session.delete(u)
...
>>> posts = Post.query.all()
>>> for p in posts:
...     db.session.delete(p)
...
>>> db.session.commit()
```

Setting up the Flask Shell for Debugging

You may need to use the python interpreter to test things out, for instance, with your database. But having to import everything explicitly is a bummer, so you can use the `python -m flask shell` command to open up an interpreter that pre-imports a lot of things and can be customized to do things like add the database instance and models for the shell session. To do that, add these to the sbmacro.py file:

```
from app import app, db
from app.models import User, Post

@app.shell_context_processor
def make_shell_context():
    return {'db': db, 'User': User, 'Post': Post}
```

This will add the function as a shell context function, so it will run when you invoke `python -m flask shell`.

Now you can use the shell to work with things like the database without having to import anything explicitly.

Creating a User Login Functionality

Password Hashing

Password hashing and security is implemented with Werkzeug, a core Flask dependency. Here's an example:

```
>>> from werkzeug.security import generate_password_hash
>>> hash = generate_password_hash('foobar')
>>> hash
'pbkdf2:sha256:50000$vT9fkZM8$04dfa35c6476acf7e788a1b5b3c35e217c78dc04539d
295f011f01f18cd2175f'
```

This works in such a way as to not be reversable and it can hash the same password and return different results. You can verify a password like this:

```
>>> from werkzeug.security import check_password_hash
>>> check_password_hash(hash, 'foobar')
True
>>> check_password_hash(hash, 'barfoo')
False
```

All this password verification and hashing logic can be implemented as two new methods in the User model:

```
from werkzeug.security import generate_password_hash, check_password_hash

# ...

class User(db.Model):
    # ...

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)
```

Here's what that would look like:

```
>>> u = User(username='susan', email='susan@example.com')
>>> u.set_password('mypassword')
>>> u.check_password('anotherpassword')
False
>>> u.check_password('mypassword')
True
```

Flask-Login

Flask-Login is a login extention that tracks if a user is logged in or not and gives you a 'remember me' functionality. We will use it, so install it:

```
(venv) $ python -m pip install flask-login
```

As with the other extentions, Flask-Login needs to be created and initialized right after the application instance in `app/__init__.py`:

```
# ...
from flask_login import LoginManager

app = Flask(__name__)
# ...
login = LoginManager(app)

# ...
```

Flask-Login requires that some properties and methods are implemented on any model that is being used as the login basis. So, for our User model, we need to implement these. The 4 required items are:

1. **is_authenticated: a property that is True if user has valid credentials, and False otherwise.**
2. **is_active: True if the user's account is active and False otherwise.**
3. **is_anonymous: False for regular users, True for special, anonymous users.**
4. **get_id(): returns a unique identifier for the user as a string.**

You can either add these yourself fairly easily, or use the provided `mixin` class called `UserMixin` that includes generic implementations that are appropriate for most user model classes. Add it like this:

```
# ...
from flask_login import UserMixin

class User(UserMixin, db.Model):
    # ...
```

Flask-Login tracks the logged in user as they go from page to page, but doesn't know about databases, so we need to add a function that loaded the user whenever they visit a page. The function needs to be able to load a user given their ID. Add this to the `app/models.py` module:

```
from app import login
# ...

@login.user_loader
def load_user(id):
    return User.query.get(int(id))
```

Implement logging in

Now that we have all the pieces, here's an example of what we need to add to the login route handler so that it actually logs users in:

```
# ...
from flask_login import current_user, login_user
from app.models import User

# ...

@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        return redirect(url_for('index'))
    return render_template('login.html', title='Sign In', form=form)
```

Logout is even more simple:

```
# ...
from flask_login import logout_user

# ...

@app.route('/logout')
def logout():
    logout_user()
    return redirect(url_for('index'))
```

Here's a handy thing: Make the Login and Logout buttons visible depending onthe login status. if user.is_anonymous. Add this to base.html:

```
    <div>
        Microblog:
        <a href="{{ url_for('index') }}">Home</a>
        {% if current_user.is_anonymous %}
        <a href="{{ url_for('login') }}">Login</a>
        {% else %}
        <a href="{{ url_for('logout') }}">Logout</a>
        {% endif %}
    </div>
```

You can also do fancy things like require a login to view certain pages and redirect back to that page once logged in. To see how to do that, visit Miguel Grinberg's page on the subject.

But Users cannot yet register themselves, so you cannot add users other than through the flask shell right now... Let's solve that.

Adding User Registration

First you need a form to add that implements all our new features. Here's an example. It should live in `app/forms.py`:

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import ValidationError, DataRequired, Email,
EqualTo
from app.models import User

# ...

class RegistrationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    email = StringField('Email', validators=[DataRequired(), Email()])
#Email() makes sure it is in the form of an email address.
    password = PasswordField('Password', validators=[DataRequired()])
    password2 = PasswordField(
        'Repeat Password', validators=[DataRequired(),
EqualTo('password')])  # EqualTo is another stock validator that makes
sure one field is equal to another.
    submit = SubmitField('Register')

    def validate_username(self, username):
        user = User.query.filter_by(username=username.data).first()
        if user is not None:
            raise ValidationError('Please use a different username.')

    def validate_email(self, email):
```

```
        user = User.query.filter_by(email=email.data).first()
        if user is not None:
            raise ValidationError('Please use a different email address.')
```

The methods are interesting here because if they are of the form `validate_<fieldname>`, then WTForms asumes they are custom validators for that field and invokes them in addition to the stock validators. Here, they make sure that the username and email aren't already in the database.

Now you need a template to display all this:

```
{% extends "base.html" %}

{% block content %}
    <h1>Register</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}<br>
            {% for error in form.username.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.email.label }}<br>
            {{ form.email(size=64) }}<br>
            {% for error in form.email.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}<br>
            {% for error in form.password.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password2.label }}<br>
            {{ form.password2(size=32) }}<br>
            {% for error in form.password2.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

And this needs linked to near the login form for new users.

Something like this:

```
<p>New User? <a href="{{ url_for('register') }}">Click to Register!</a>
</p>
```

And you need to handle the new route in `app/routes.py`:

```
from app import db
from app.forms import RegistrationForm

# ...

@app.route('/register', methods=['GET', 'POST'])
def register():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = RegistrationForm()
    if form.validate_on_submit():
        user = User(username=form.username.data, email=form.email.data)
        user.set_password(form.password.data)
        db.session.add(user)
        db.session.commit()
        flash('Congratulations, you are now a registered user!')
        return redirect(url_for('login'))
    return render_template('register.html', title='Register', form=form)
```

Allowing users to edit their profiles

If you have users, they should be able to change their username or email or password should they need to. So we need to create a profile page and a way to edit their info.

First we add a new route that is dynamic and takes the user's username:

```
@app.route('/user/<username>')
@login_required
def user(username):
    user = User.query.filter_by(username=username).first_or_404()
    posts = [
        {'author': user, 'body': 'Test post #1'},
        {'author': user, 'body': 'Test post #2'}
    ]
    return render_template('user.html', user=user, posts=posts)
```

Notice the `<username>` in the route. This indicates a dynamic component. That means that flask will accept any text in the part of the URL. This is ok for us because we made it only accessible by logged in

users with `@login_required`. Notice also the user of `first_or_404()` which get's the first result from the database for a certain query, or redirects to a 404 error if there are no results.

We'll need a `user.html` template as well to render:

```
{% extends "base.html" %}

{% block content %}
    <h1>User: {{ user.username }}</h1>
    <hr>
    {% for post in posts %}
    <p>
    {{ post.author.username }} says: <b>{{ post.body }}</b>
    </p>
    {% endfor %}
{% endblock %}
```

Now it will display the user and their 'posts' which we hardcoded in as an example. So we now have a user profile page, but no links to it anywhere, so you'll want to add that to `base.html`. Something like this:

```
    <div>
      Microblog:
      <a href="{{ url_for('index') }}">Home</a>
      {% if current_user.is_anonymous %}
      <a href="{{ url_for('login') }}">Login</a>
      {% else %}
      <a href="{{ url_for('user', username=current_user.username)
}}">Profile</a>
      <a href="{{ url_for('logout') }}">Logout</a>
      {% endif %}
    </div>
```

While we are not actually implementing 'posts', it is a good reason to show an example of another useful thing: Jinja2 sub-templates.

The posts are ugly as is, and it would be great to be able to format them uniquely from the rest of the template. So let's create a subtemplate `app/templates/_posts.html`:

```
    <table>
        <tr valign="top">
            <td><img src="{{ post.author.avatar(36) }}"></td>
            <td>{{ post.author.username }} says:<br>{{ post.body }}</td>
        </tr>
    </table>
```

Then, to include this in `user.html`, use Jinja2's `include` statement:

```
{% extends "base.html" %}

{% block content %}
    <table>
        <tr valign="top">
            <td><img src="{{ user.avatar(128) }}"></td>
            <td><h1>User: {{ user.username }}</h1></td>
        </tr>
    </table>
    <hr>
    {% for post in posts %}
        {% include '_post.html' %}
    {% endfor %}
{% endblock %}
```

It would also be nice to track when the user was last seen. First, add something like this to the User class model:

```
class User(UserMixin, db.Model):
    #...
    last_seen = db.Column(db.DateTime, default=datetime.utcnow)
```

This requires that you migrate and upgrade the database:

```
(venv) $ python -m flask db migrate -m "Added 'last_seen' to User model"
(venv) $ python -m flask db upgrade
```

You can add the new "last_seen" field to the Profile page if you want...

Now, to record the Last Visit Time of a User, you want to record the time whenever that user makes a request from the server. To do that, Flask has a native feature for executing something vefore each request. Add this to `app/routes.py`:

```
from datetime import datetime

@app.before_request
def before_request():
    if current_user.is_authenticated:
        current_user.last_seen = datetime.utcnow()
        db.session.commit()
```

We can reuse this function for any logic that we want executed before each request.

This is all great, but now we need to make it so that the user can edit their profile to change things like email, password, etc.

First, we need a new WTForm for the profile editing... So, in `app/forms.py` insert something like:

```
from wtforms import StringField, TextAreaField, SubmitField
from wtforms.validators import DataRequired, Length

# ...

class EditProfileForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    about_me = TextAreaField('About me', validators=[Length(min=0,
max=140)])
    submit = SubmitField('Submit')
```

We must also be able to display it, so create a new file `app/templates/edit_profile.html`:

```
{% extends "base.html" %}

{% block content %}
    <h1>Edit Profile</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.username.label }}<br>
            {{ form.username(size=32) }}<br>
            {% for error in form.username.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.about_me.label }}<br>
            {{ form.about_me(cols=50, rows=4) }}<br>
            {% for error in form.about_me.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

Now you need a route for this `edit_profile.html`, so add something like this to `app/routes.py`:

```
from app.forms import EditProfileForm

@app.route('/edit_profile', methods=['GET', 'POST'])
@login_required
```

```
def edit_profile():
    form = EditProfileForm()
    if form.validate_on_submit():
        current_user.username = form.username.data
        current_user.about_me = form.about_me.data
        db.session.commit()
        flash('Your changes have been saved.')
        return redirect(url_for('edit_profile'))
    elif request.method == 'GET':
        form.username.data = current_user.username
        form.about_me.data = current_user.about_me
    return render_template('edit_profile.html', title='Edit Profile',
                           form=form)
```

Now we must make sure that the user (and only when they are logged in) can edit their profile by providing them a link in `app/templates/user.html`**:**

```
{% if user == current_user %}
<p><a href="{{ url_for('edit_profile') }}">Edit your profile</a></p>
{% endif %}
```

Setting Debug Mode: On

There are two main ways you can implement this:

1. **Set the FLASK_DEBUG environmental variable to one:**

```
(venv) $ export FLASK_DEBUG=1
```

2. **Add a run script to sbmicro.py at the end like this:**

```
if __name__ == "__main__":
    app.run(debug=True)
```

If you choose option 1, just run the server like normal (`python -m flask run`**) and it should be in debug mode which gives you Stack Traces when it crashes and automatically restarts when you change things. If option 2, just run** `python sbmacro.py` **and it shoot boot right up.**

Custom Error Handling

HTTP Errors

Errors must be handled and logged in such a way that the user is not exactly privy to what went wrong (security), they are minimally affected (user experience) and the administrators are aware so as to fix any error or bugs (via alerts and/or logging).

Let's start by handling the common HTTP errors 404 and 500. Create a new file: `app/errors.py`. Note the second return value, which we need to include because the default status code (200) is what we wanted before.

500 errors occur when there is a database error. Notice the `db.session.rollback()` call. This is becuase a 500 error is generated when a db session had a failure (such as a duplicate username or something), so you want to roll back to a clean slate.

Now we need to create the corresponding HTML pages that will be called. Here's where you can customize what your user sees when these errors occur.

We could do something simple or more complex. Here are some simple examples for `app/templates/404.html` and `app/templates/500.html`: 404:

```
{% extends "base.html" %}

{% block content %}
    <h1>File Not Found</h1>
    <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

500:

```
{% extends "base.html" %}

{% block content %}
    <h1>An unexpected error has occurred</h1>
    <p>The administrator has been notified. Sorry for the inconvenience!
</p>
    <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

Now you need to import `errors.py` as a module in the application instance (so in `__init__.py`).

```
# ...

from app import routes, models, errors
```

We can test these by turning off debugging if it's on (`FLASK_DEBUG=0`), and trying to change a username to one that already exists via a "edit profile" page or something. This is still not an elogant way to handle the error, but it's much better than the default.

Emailing Errors

As the app currently stands, the stack trace is printed as it goes to the terminal, meaning that errors would only be found if you were constantly monitoring it. That's fine for now, but certainly not in

production.

One nice way is to have stack traces sent via email to an administrator's email address.

First you add the email server details to the `config.py` file:

```
class Config(object):
    # ...
    MAIL_SERVER = os.environ.get('MAIL_SERVER')
    MAIL_PORT = int(os.environ.get('MAIL_PORT') or 25)
    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS') is not None
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
    ADMINS = ['your-email@example.com']
```

Because Flask is awesome, it already has a `logging` package to write it's logs and send them via email.

So, we need to add a SMTPHandler instance to the new Flask logger object, which is `app.logger`:
`app/__init__.py`:

```
import logging
from logging.handlers import SMTPHandler

# ...

if not app.debug:
    if app.config['MAIL_SERVER']:
        auth = None
        if app.config['MAIL_USERNAME'] or app.config['MAIL_PASSWORD']:
            auth = (app.config['MAIL_USERNAME'],
app.config['MAIL_PASSWORD'])
        secure = None
        if app.config['MAIL_USE_TLS']:
            secure = ()
        mail_handler = SMTPHandler(
            mailhost=(app.config['MAIL_SERVER'], app.config['MAIL_PORT']),
            fromaddr='no-reply@' + app.config['MAIL_SERVER'],
            toaddrs=app.config['ADMINS'], subject='Microblog Failure',
            credentials=auth, secure=secure)
        mail_handler.setLevel(logging.ERROR)
        app.logger.addHandler(mail_handler)
```

This only works when debugging is not enabled. It creates an SMTPHandler instance, sets the sensitivity level to only log errors instead of warnings, info, or debugging messages, and attaches it to the `app.logger` object from Flask.

Now, we created a temporary gmail for admin logging purposes (ad.sbmacro@gmail.com), and can practice sending emails to there. Here's what needs done to set up a gmail:

```
(venv) export MAIL_SERVER=smtp.googlemail.com
(venv) export MAIL_PORT=587
(venv) export MAIL_USE_TLS=1
(venv) export MAIL_USERNAME=<your-gmail-username>
(venv) export MAIL_PASSWORD=<your-gmail-password>
```

File Logging Errors

Keeping track of more types of errors and problems in a rotating file log is also useful. Here's how you add the handler `RotatingFileHandler` to the application logger. `app/__init__.py`:

```
# ...
from logging.handlers import RotatingFileHandler
import os

# ...

if not app.debug:
    # ...

    if not os.path.exists('logs'):
        os.mkdir('logs')
    file_handler = RotatingFileHandler('logs/microblog.log',
maxBytes=10240,
                                       backupCount=10)
    file_handler.setFormatter(logging.Formatter(
        '%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%
(lineno)d]'))
    file_handler.setLevel(logging.INFO)
    app.logger.addHandler(file_handler)

    app.logger.setLevel(logging.INFO)
    app.logger.info('Microblog startup')
```

Unit Testing the User Model

We should get in the habit of created automated tests to make sure the methods we've written work as desired as things continue to change.

Python includes a very useful `unittest` package that makes it easy to write and execute unit tests. Here is how you would use it to write unit tests for the User class in a `tests.py` module (particularly if they the User was more complex and we implemented a follower/followed schema for our database): `test.py`:

```
from datetime import datetime, timedelta
import unittest
from app import app, db
```

```
from app.models import User, Post

class UserModelCase(unittest.TestCase):
    # setUp and tearDown are special methods that the unit testing
framework executes before and after each test respectively.
    def setUp(self):
        app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite://'  # Uses an in-
memory SQLite database during the tests.
        db.create_all()  # creates all the database tables

    def tearDown(self):
        db.session.remove()
        db.drop_all()

    def test_password_hashing(self):
        u = User(username='susan')
        u.set_password('cat')
        self.assertFalse(u.check_password('dog'))
        self.assertTrue(u.check_password('cat'))

    def test_avatar(self):
        u = User(username='john', email='john@example.com')
        self.assertEqual(u.avatar(128),
('https://www.gravatar.com/avatar/'

'd4c74594d841139328695756648b6bd6'
                                           '?d=identicon&s=128'))

    def test_follow(self):
        u1 = User(username='john', email='john@example.com')
        u2 = User(username='susan', email='susan@example.com')
        db.session.add(u1)
        db.session.add(u2)
        db.session.commit()
        self.assertEqual(u1.followed.all(), [])
        self.assertEqual(u1.followers.all(), [])

        u1.follow(u2)
        db.session.commit()
        self.assertTrue(u1.is_following(u2))
        self.assertEqual(u1.followed.count(), 1)
        self.assertEqual(u1.followed.first().username, 'susan')
        self.assertEqual(u2.followers.count(), 1)
        self.assertEqual(u2.followers.first().username, 'john')

        u1.unfollow(u2)
        db.session.commit()
        self.assertFalse(u1.is_following(u2))
        self.assertEqual(u1.followed.count(), 0)
        self.assertEqual(u2.followers.count(), 0)

    def test_follow_posts(self):
        # create four users
        u1 = User(username='john', email='john@example.com')
```

```
        u2 = User(username='susan', email='susan@example.com')
        u3 = User(username='mary', email='mary@example.com')
        u4 = User(username='david', email='david@example.com')
        db.session.add_all([u1, u2, u3, u4])

        # create four posts
        now = datetime.utcnow()
        p1 = Post(body="post from john", author=u1,
                  timestamp=now + timedelta(seconds=1))
        p2 = Post(body="post from susan", author=u2,
                  timestamp=now + timedelta(seconds=4))
        p3 = Post(body="post from mary", author=u3,
                  timestamp=now + timedelta(seconds=3))
        p4 = Post(body="post from david", author=u4,
                  timestamp=now + timedelta(seconds=2))
        db.session.add_all([p1, p2, p3, p4])
        db.session.commit()

        # setup the followers
        u1.follow(u2)  # john follows susan
        u1.follow(u4)  # john follows david
        u2.follow(u3)  # susan follows mary
        u3.follow(u4)  # mary follows david
        db.session.commit()

        # check the followed posts of each user
        f1 = u1.followed_posts().all()
        f2 = u2.followed_posts().all()
        f3 = u3.followed_posts().all()
        f4 = u4.followed_posts().all()
        self.assertEqual(f1, [p2, p4, p1])
        self.assertEqual(f2, [p2, p3])
        self.assertEqual(f3, [p3, p4])
        self.assertEqual(f4, [p4])

if __name__ == '__main__':
    unittest.main(verbosity=2)
```

Each time a feature is added to the application, a unit test should be added for it. For example, we need to test our ScienceBase algorithms and data tables once added...

The other models only need unit tests created for them when they have methods created within them to test.

To run the entire test suite, use:

```
(venv) $ python tests.py
```

Email-based Password Resetting

We now implement the ability for a user to reset their password via email if they forget it.

Do do this, we install Flask-Mail, which allows us to send emails.

```
(venv) $ python -m pip install flask-mail
```

The password reset links will have a secure token in them. To generate these tokens, I'm going to use JSON Web Tokens, which also have a popular Python package:

```
(venv) $ python -m pip install pyjwt
```

The Flask-Mail extension is configured from the app.config object. Remember when we added the email configuration for sending ourself an email whenever an error occurred in production? The choice of configuration variables was modeled after Flask-Mail's requirements, so there isn't really any additional work that is needed, the configuration variables are already in the application.

Like most Flask extensions, you need to create an instance right after the Flask application is created. In this case this is an object of class Mail:

`app/__init__.py`**: Flask-Mail instance.**

```
# ...
from flask_mail import Mail

app = Flask(__name__)
# ...
mail = Mail(app)
```

Then you can either use a real email address and server, or use the Python one. We will use our gmail account we made for sbmacro admin: ad.sbmacro@gmail.com. Don't forget to make sure these environmental variables are set:

```
(venv) $ export MAIL_SERVER=smtp.googlemail.com
(venv) $ export MAIL_PORT=587
(venv) $ export MAIL_USE_TLS=1
(venv) $ export MAIL_USERNAME=ad.sbmacro@gmail.com
(venv) $ export MAIL_PASSWORD=sbMACRO_@dmin1
```

Remember that the security features in your Gmail account may prevent the application from sending emails through it unless you explicitly allow "less secure apps" access to your Gmail account. You can read about this here.

From here, we want to set up an email framework that we can use to send emails to users. We'll start by creating a new module `app/email.py` and adding the following:

```
from flask_mail import Message
from app import mail

def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    mail.send(msg)
```

Now, there's more options. If we want to implement those (such as CC and BCC) check out the documentation.

After creating the email framework, we need to add the ability to request a password reset from the login page. First, we add a link for the user to click: `app/templates/login.html`

```
<!-- ... -->
<p>
    <a href="{{ url_for('reset_password_request') }}">Forgot Your
Password?</a>
</p>
```

This link will bring the user to a new form that asks for the email related to the account who's password is to be reset. `app/forms.py`**:**

```
class ResetPasswordRequestForm(FlaskForm):
    email = StringField('Email', validators=[DataRequired(), Email()])
    submit = SubmitField('Request Password Reset')
```

Then we need to write an html template for the form: `app/templates/password_reset.html`**:**

```
{% extends "base.html" %}

{% block content %}
    <h1>Reset Password</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.email.label }}<br>
            {{ form.email(size=64) }}<br>
            {% for error in form.email.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

This, of course, means that we need a new view function within the `app/routes.py` module:

```
from app.forms import ResetPasswordRequestForm
from app.email import send_password_reset_email

#...

@app.route('/reset_password_request', methods=['GET', 'POST'])
def reset_password_request():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = ResetPasswordRequestForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user:
            send_password_reset_email(user)
        flash('Check your email for the instructions to reset your
password')
        return redirect(url_for('login'))
    return render_template('reset_password_request.html',
                           title='Reset Password', form=form)
```

The `send_password_reset_email()` function doesn't exist yet, but it will. After the email is sent, a flash message is used. In our site, since we don't use flash messages, we redirect to a new page with the information.

Now we need to have a way to create a password reset link. This is the link sent to the user via email. When clicked, a page where the new password can be set is presented to the user. We must make sure that only valid reset links can be used to reset an account's password.

The links will have a *token*. This token will be validated before allowing the password to change, as proof that the user that requested the email has access to the email address on that account. A JSON Web Token (JWT) is a popular token standard that is self-contained. You can send a token to a user in an email, and when the user clicks the link that feeds the token back into the application, it can be verified on its own.

Here's an example for how JWTs work:

```
>>> import jwt
>>> token = jwt.encode({'a': 'b'}, 'my-secret', algorithm='HS256')
>>> token
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhIjoiYiJ9.dvOo58OBDHiuSHD4uW88nf
JikhYAXc_sfUHq1mDi4G0'
>>> jwt.decode(token, 'my-secret', algorithms=['HS256'])
{'a': 'b'}
```

The `{'a': 'b'}` dictionary is an example payload that is going to be written into the token. To make the token secure, a secret key needs to be provided to be used in creating a cryptographic signature. For this example we have used the string `'my-secret'`, but with the application we'll use the `SECRET_KEY` from the configuration. The `algorithm` argument specifies how the token is to be generated. The `HS256` algorithm is the most widely used.

As you can see the resulting token is a long sequence of characters. But do not think that this is an encrypted token. The contents of the token, including the payload, can be decoded easily by anyone (don't believe it? Copy the above token and then enter it in the JWT debugger to see its contents). What makes the token secure is that the payload is signed. If somebody tried to forge or tamper with the payload in a token, then the signature would be invalidated, and to generate a new signature the secret key is needed. When a token is verified, the contents of the payload are decoded and returned back to the caller. If the token's signature was validated, then the payload can be trusted as authentic.

The payload that we're going to use for the password reset tokens is going to have the format `{'reset_password': user_id, 'exp': token_expiration}`. The `exp` field is standard for JWTs and if present it indicates an expiration time for the token. If a token has a valid signature, but it is past its expiration timestamp, then it will also be considered invalid. For the password reset feature, we're going to give these tokens 10 minutes of life.

When the user clicks on the emailed link, the token is going to be sent back to the application as part of the URL, and the first thing the view function that handles this URL will do is to verify it. If the signature is valid, then the user can be identified by the ID stored in the payload. Once the user's identity is known, the application can ask for a new password and set it on the user's account.

Since these tokens belong to users, we're going to write the token generation and verification functions as methods in the User model: `app/models.py`:

```
from time import time
import jwt
from app import app

class User(UserMixin, db.Model):
    # ...

    def get_reset_password_token(self, expires_in=600):
        return jwt.encode(
            {'reset_password': self.id, 'exp': time() + expires_in},
            app.config['SECRET_KEY'], algorithm='HS256').decode('utf-8')

    @staticmethod
    def verify_reset_password_token(token):
        try:
            id = jwt.decode(token, app.config['SECRET_KEY'],
                            algorithms=['HS256'])['reset_password']
        except:
            return
        return User.query.get(id)
```

The `get_reset_password_token()` function generates the token as a string, and the `decode(utf-8)` is needed because the `jwt.encode()` function returns the token as a byte sequence, which is not as convenient as a string.

The `verify_reset_password_token()` is a static method, which menas that it can be invoked directoy from teh class. A static method is similar to a class method, but it doesn't receive the class as a first argument. This method takes a token and attempts to decode is by invoking PyJWT's `jwt.decode()` function. If it fails (invalid or expired), an exception is raised, and in that case we catch it to prevent the error and then return None to the caller. If the token is valid, then the value of the `reset_password` key from the token's payload is the ID of the user, so we can load the user and return it.

Now that we have tokens, we must include the ability to send the password reset email. The `send_password_reset_email()` function relies on the `send_email()` function we wrote above. `app/email.py`:

```
from flask import render_template
from app import app

#...

def send_password_reset_email(user):
    token = user.get_reset_password_token()
    send_email('[sbMACRO] Reset Your Password',
               sender= app.config['ADMINS'][0],
               recipients=[user.email],
               text_body=render_template('email/reset_password.txt',
                                         user=user, token=token),
               html_body=render_template('email/reset_password.html',
                                         user=user, token=token))
```

The cool part here is that the text and HTML content for the emails is generated from templates, just like out web routes, using `render_template()`. The templaces receive the user and the token as argumets, so that a personalized email message can be generated. Here is the text template for the reset password email: `app/templates/email/reset_password.txt`:

```
Hello {{ user.username }},

To reset your password click the following link:

{{ url_for('reset_password', token=token, _external=True) }}

If you have not requested a password reset, you may ignore this email.

Yours,

sbMACRO Devs
```

And the nicer HTML version of the same email:

`app/templates/email/reset_password.html`:

```
<p>Hello {{ user.username }},</p>
<p>
    To reset your password
    <a href="{{ url_for('reset_password', token=token, _external=True)
}}">
        click here
    </a>.
</p>
<p>Alternatively, you can paste the following link in your browser's
address bar:</p>
<p>{{ url_for('reset_password', token=token, _external=True) }}</p>
<p>If you have not requested a password reset simply ignore this message.
</p>
<p>Yours,</p>
<p>sbMACRO Devs</p>
```

The reset_password route that is referenced in the `url_for()` call in these two email templates does not exist yet, this will be added in a bit. The `_external=True` argument that we included in the `url_for()` calls in both templates is also new. The URLs that are generated by `url_for()` by default are relative URLs, so for example, the `url_for('user', username='susan')` call would return *`/user/susan`*. This is normally sufficient for links that are generated in web pages, because the web browser takes the remaining parts of the URL from the current page. When sending a URL by email however, that context does not exist, so fully qualified URLs need to be used. When `_external=True` is passed as an argument, complete URLs are generated, so the previous example would return *`http://localhost:5000/user/susan`*, or the appropriate URL when the application is deployed on a domain name.

Now we need to create a route to reset the user password.

`app/routes.py`**: Password reset view function.**

```
from app.forms import ResetPasswordForm

@app.route('/reset_password/<token>', methods=['GET', 'POST'])
def reset_password(token):
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    user = User.verify_reset_password_token(token)
    if not user:
        return redirect(url_for('index'))
    form = ResetPasswordForm()
    if form.validate_on_submit():
        user.set_password(form.password.data)
        db.session.commit()
        flash('Your password has been reset.')
```

```
            return redirect(url_for('login'))
        return render_template('reset_password.html', form=form)
```

In this view function we first make sure the user is not logged in, and then we determine who the user is by invoking the token verification method in the User class. This method returns the user if the token is valid, or None if not. If the token is invalid I redirect to the home page.

If the token is valid, then we present the user with a second form, in which the new password is requested. This form is processed in a way similar to previous forms, and as a result of a valid form submission, we invoke the `set_password()` method of User to change the password, and then redirect to the login page, where the user can now login.

Here is the `ResetPasswordForm` class: `app/forms.py`:

```
class ResetPasswordForm(FlaskForm):
    password = PasswordField('Password', validators=[DataRequired()])
    password2 = PasswordField('Repeat Password',
                              validators=[DataRequired(),
                              EqualTo('password')])
    submit = SubmitField('Request Password Reset')
```

And here is the corresponding HTML template... `app/templates/reset_password.html`:

```
{% extends "base.html" %}

{% block content %}
    <h1>Reset Your Password</h1>
    <form action="" method="post">
        {{ form.hidden_tag() }}
        <p>
            {{ form.password.label }}<br>
            {{ form.password(size=32) }}<br>
            {% for error in form.password.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>
            {{ form.password2.label }}<br>
            {{ form.password2(size=32) }}<br>
            {% for error in form.password2.errors %}
            <span style="color: red;">[{{ error }}]</span>
            {% endfor %}
        </p>
        <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

This means that the password reset feature is basically done. We still need to deal with the slowdown from Emails. We need Asynchronous Emails. All the interactions that need to happen when sending an email make the task slow, it usually takes a few seconds to get an email out, and maybe more if the email server of the addressee is slow, or if there are multiple addressees.

So, what we want is for `send_email()` to be asynchronous, meaning that when this function is called, the task fo sending the email is scheduled to happen in the backgorund, freeing the `send_email()` to return immediately so that the application can continue running concurrently with the email being sent.

Python has support for running asynchronous tasks. The `threading` and `multiprocessing` modules can both do this. Starting a background thread for email being sent is much less resource intensive than starting a brand new process, so we're going to go with that approach:

`app/email.py`:

```
from threading import Thread
# ...

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)


def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    Thread(target=send_async_email, args=(app, msg)).start()
```

The `send_async_email` function now runs in a background thread, invoked via the `Thread()` class in the last line of `send_email()`. With this change, the sending of the email will run in the thread, and when the process completes the thread will end and clean itself up. If you have configured a real email server, you will definitely notice a speed improvement when you press the submit button on the password reset request form.

You probably expected that only the msg argument would be sent to the thread, but as you can see in the code, I'm also sending the application instance. When working with threads there is an important design aspect of Flask that needs to be kept in mind. Flask uses contexts to avoid having to pass arguments across functions. It can be complicated, but know that there are two types of contexts, the application context and the request context. In most cases, these contexts are automatically managed by the framework, but when the application starts custom threads, contexts for those threads may need to be manually created.

There are many extensions that require an application context to be in place to work, because that allows them to find the Flask application instance without it being passed as an argument. The reason many extensions need to know the application instance is because they have their configuration stored in the `app.config` object. This is exactly the situation with Flask-Mail. The `mail.send()` method

needs to access the configuration values for the email server, and that can only be done by knowing what the application is. The application context that is created with the with `app.app_context()` call makes the application instance accessible via the `current_app` variable from Flask.

Universal Time Displayed As Local

Now, we chose to use UTC time (Coordinated Universal Time), as it is the same around the world and can be changed to the user's local time when displayed, saving trouble with managing a bunch of different time zones on the server side. The easiest way to do this is by converting those UTC times using `Moment.js` and `Flask-Moment`.

`Moment.js` (found here) is a small open-source JavaScript library that takes date and time rendering to another level, as it provides every imaginable formatting option, and then some. And a while ago Miguel Grinberg (author of the Flask Mega Tutorial) created Flask-Moment, a small Flask extension that makes it very easy to incorporate moment.js into our application.

First, we install Flask-Moment:

```
(venv) python -m pip install flask-moment
```

Then add the extention to the application, like usual: `app/__init__.py`:

```
#...
from flask_moment import Moment

app = Flask(__name__)
#...
moment = Moment(app)
```

Because Flask-Moment needs moment.js to function, it must be included in every page. To do that, it needs declared in a `<script>` tag explicitly on every page (or in the `base.html` page), or we can do a fancy super block. ~~Flask-Moment gives us an easy way to do this by exposing a `moment.include_moment()` function that generates the `<script>` tag.~~

~~`app/templates/base.html`~~

```
...

{% block scripts %}
    {{ super() }}
    {{ moment.include_moment() }}
{% endblock %}
```

~~The `scripts` block here is another block exported by by Flask-Bootstrap's base template.~~ This requires Flask-Bootstrap, which we do not use. So we will use the first technique by installing moment.js with

npm:

```
(venv) npm i moment
```

Then we will add it to `base.html`. `app/templates/base.html`:

```
<!-- ... -->
<body>
    <script src="../../node_modules/moment/moment.js"></script>
    <script>
        moment().format();
    </script>
    <!-- ... -->
</body>
```

Moment.js makes a `moment` **class available to the browser. The first step to render a timestamp is to create an object of this class, passing the desired timestamp in ISO 8601 format.**

The ISO 8601 standard format for dates and times is as follows: {{ year }}-{{ month }}-{{ day }}T{{ hour }}:{{ minute }}:{{ second }}{{ timezone }}. We already decided that we were only going to work with UTC timezones, so the last part is always going to be Z, which represents UTC in the ISO 8601 standard.

The `moment` **object provides several methods for different rendering options. Below are some of the most common options:**

```
moment('2017-09-28T21:45:23Z').format('L')
"09/28/2017"
moment('2017-09-28T21:45:23Z').format('LL')
"September 28, 2017"
moment('2017-09-28T21:45:23Z').format('LLL')
"September 28, 2017 2:45 PM"
moment('2017-09-28T21:45:23Z').format('LLLL')
"Thursday, September 28, 2017 2:45 PM"
moment('2017-09-28T21:45:23Z').format('dddd')
"Thursday"
moment('2017-09-28T21:45:23Z').fromNow()
"7 hours ago"
moment('2017-09-28T21:45:23Z').calendar()
"Today at 2:45 PM"
```

This example creates a moment object initialized to September 28th 2017 at 9:45pm UTC. You can see that all the options we tried above are rendered in UTC-7, which is the timezone configured on the computer when the commands were entered.

Note how the different methods create different representations. With `format()` **you control the format of the output with a format string, similar to the strftime function from Python. The**

`fromNow()` and `calendar()` methods are interesting because they render the timestamp in relation to the current time, so you get output such as "a minute ago" or "in two hours", etc.

If we were working directly in JavaScript, the above calls return a string that has the rendered timestamp. Then it is up to you to insert this text in the proper place on the page, which unfortunately requires some JavaScript to work with the DOM. The Flask-Moment extension greatly simplifies the use of moment.js by enabling a moment object similar to the JavaScript one in your templates.

Let's look at the timestamp that appears in the profile page. The current `user.html` template lets Python generate a string representation of the time:

`app/templates/user.html`:

```
{% if user.last_seen %}
    <p>Last seen on: {{ user.last_seen }} </p>
{% endif %}
```

Now, we can now render this timestamp using Flask-Moment as follows:

`app/templates/user.html`:

```
{% if user.last_seen %}
    <p>Last seen on: {{ moment(user.last_seen).format('LLL') }} </p>
{% endif %}
```

Here, the argument passed to `moment()` is the python `datetime` object that we store in our database. The `moment()` call issued from a template also automatically generates the required JavaScript code to insert the rendered timestamp in the proper place of the DOM.

Implementing a Better Structure for the Application

When looking at our application as it stands, we can see that there are several subsystems (eg. user authentication, error subsystem, core functionality), but these are so interweaved throughout the application, that the code for the subsystems cannot really be easily isolated to work with or reuse. This isn't ideal. For instance, debugging a particular feature or subsystem is harder if that code is spread throughout several different general files. To do this sort of centralization of code, we can use the *blueprints* feature of Flask.

Another issue is that the Flask application instance is created as a global variable in `app/__init__.py`, then imported by a lot of modules. This can be a problem if, for example, you are testing the application under different configurations. Because the application is global, there is no way to have more than one instance of it. Instead, it is better to have an *application factory function* that is called at runtime, accepts a configuration object, and returns a new, pristine application instance.

However, this requires changes to almost every file in the application. Regardless, we will

- **Refactor application to introduce blueprints for the three subsystems above**
- **Create and implement an application factory function**

Blueprints

In Flask, a 'blueprint' is a logical structure that represents a subset of the application. A blueprint can include elements such as routes, view functions, forms, templates, and static files. If written in a separate Python package, then you have a component that encasulates the elements related to a specific feature of the application.

From the tutorial:

```
The contents of a blueprint are initially in a dormant state. To associate
these elements, the blueprint needs to be registered with the application.
During the registration, all the elements that were added to the blueprint
are passed on to the application. So you can think of a blueprint as a
temporary storage for application functionality that helps in organizing
your code.
```

Good stuff.

Error Handling Blueprint

First, we can start with the Error Handling subsystem. The structure of the blueprint is:

```
app/
    errors/                             <-- blueprint package
        __init__.py                     <-- blueprint creation
        handlers.py                     <-- error handlers
    templates/
        errors/                         <-- error templates
            404.html
            500.html
    __init__.py                         <-- blueprint registration
```

What is done then, is that `app/errors.py` is moved into `app/errors/handlers.py` and the error templates are moved to `app/templates/errors` to separate them from other templates. The `render_template()` calls for each of the errors also needs changed to reflect the new template location. After doing that, the blueprint creation needs to be added to `app/__init__.py`, after the application instance is created.

Another option that Flask blueprints provides is the ability to have a separate directory for templates or static files. In the example above, we have an `errors` subdirectory within the `templates` directory. We could also have the templates that belong to a particular blueprint within that blueprint package. For example, to have a `templates` directory within the blueprint package, you would just add

templade_folder='templates as an argument to the Blueprint() constructor. Then they would all be stored in app/errors/templates like so:

```
app/
    errors/                             <-- blueprint package
        __init__.py                     <-- blueprint creation
        handlers.py                     <-- error handlers
        templates/                      <-- error templates
            404.html
            500.html
    __init__.py                         <-- blueprint registration
```

This is the structure that we will use as it keeps the packages more insulated and centralized.

The creation of a blueprint is similar to creating an application. It is done in the __init__.py module of the blueprint package: app/errors/__init__.py:

```
from flask import Blueprint

bp = Blueprint('errors', __name__, template_folder='templates')

from app.errors import handlers
```

The Blueprint class takes the name of the blueprint, the name of the base module (typically set to __name__ like the flash application instance to signify that it is the file that it is currently in), and a few optional arguments (including the template_folder argument we provided). After the blueprint object is created, we import the handlers.py module, so the error handlers in it are registered with the blueprint. The import is at the bottom to avoid circular dependencies.

In the handlers.py module, intead of attaching the error handlers to the application with the old @app.errorhandler decorator, we instead use the blueprint's @bp.app_errorhandler decorator. We also need to modify the path to the two error templates, since we moved them into the new 'errors' package.

The final step to complete the refactoring of the error handlers is to register the blueprint with the application: app/__init__.py:

```
app = Flask(__name__)

#...

from app.errors import bp as errors_bp
app.register_blueprint(errors_bp)

# ...

from app import routes, models  # <-- remove errors from this import!
```

To register a blueprint, the `register_blueprint()` method of the Flask application instance is used. When a blueprint is registered, any view functions, templates, static files, error handlers, etc. are connected to the application. We put the import of the blueprint right above the app.register_blueprint() to avoid circular dependencies.

Authentication Blueprint

The layout is similar to the Error Handling blueprint. Here is a layout similar to one that we will use:

```
app/
    auth/                               <-- blueprint package
        __init__.py                     <-- blueprint creation
        email.py                        <-- authentication emails
        forms.py                        <-- authentication forms
        routes.py                       <-- authentication routes
        templates/                      <-- blueprint templates
            login.html
            register.html
            reset_password_request.html
            reset_password.html
    __init__.py                         <-- blueprint registration
```

From the tutorial:

```
To create this blueprint I had to move all the authentication related
functionality to new modules I created in the blueprint. This includes a
few view functions, web forms, and support functions such as the one that
sends password reset tokens by email. I also moved the templates into a
[...]directory [within the package] to separate them from the rest of the
application, like I did with the error pages.
```

When defining the different routes for the blueprint, the `@bp.route` decorater was used instead of the previously used `@app.route`. We also have to change the syntax used for the `url_for()` function to build URLs. Normally, the first argument is the view function name, but when a route is defined in a blueprint, this argument must include the blueprint name and the view function name, seperated by a period. For example: `url_for('login')` becomes `url_for('auth.login')`. This must be done for all view functions in the blueprint.

Then register the `auth` blueprint with the application"

```
#...
from app.auth import bp as auth_bp
app.register_blueprint(auth_bp, url_prefix='/auth')
# ...
```

Notice here is an example of the optional `url_prefix` argument. This can be provided to add a prefix to any of the view/route URLs. For example, for the /login route, rather than http://localhost:5000/login, it would be http://localhost:5000/auth/login. This is optional. It helps seperate different parts of the application and keep the namespaces clean.

Main Application Blueprint

This blueprint is the one for the core application logic. The refactoring is basically the same process as the previous blueprints. The name `main` is a good choice for the blueprint, which means that `main.` needs added to each of the view functions as above.

The Application Factory Pattern

Now let's deal with the global application variable issue. To do this, we will create an application factory function called `create_app()` that constructions a Flask application instance and excepts a configuration object as a parameter.

This is what the transformation of `app/__init__.py` looks like:

```
# ...
db = SQLAlchemy()
migrate = Migrate()
login = LoginManager()
login.login_view = 'auth.login'
login.login_message = _l('Please log in to access this page.')
mail = Mail()
moment = Moment()


def create_app(config_class=Config):
    app = Flask(__name__)
    app.config.from_object(config_class)

    db.init_app(app)
    migrate.init_app(app, db)
    login.init_app(app)
    mail.init_app(app)
    moment.init_app(app)

    # ... no changes to blueprint registration

    if not app.debug and not app.testing:
        # ... no changes to logging setup

    return app
```

Most all of our Flask extentions were initialized by creating an instance of the extension and passing the application as an argument. However, that will no longer be possible when it isn't a global variable any longer. So we initialize the extentions in two phases:

1. **We extention instance is created in the global scope, but given no arguments. The instance is created, but not attached to the applications.**
2. **Once the *application* instance is created, the extension instances are bound to the new application using the `init_app()` method.**

Most everything else about initialization remains the same, but have been moved into the factory function instead of the global scope. We also includeda new `not app.testing` clause to the conditional that decides if email and file logging should be enabled or not so that we can skip those things during unit tests. The `app.testing` flag is going to be `True` when running unit tests when we set the `TESTING` variable to `True` in the configuration.

So, where do we call the application factory function? The top-level `microblog.py` script is the only module in which the applicaton now exists in the global scope. The other place is `tests.py`, but that will be a different section on Unit Testing.

So, most references to `app` went away with the introductoin of blueprints, but there is still remnants of code that refer to the global `app` variable. Examples include `app/models.py`, and `app/main/routes.py` modules that all reference `app.config`. Luckily, Flask developers tried to make it easy for view functions to access the application instance without having to import it like we have been doing. The `current_app` variable that Flask provides is a special "context" variable that Flask initializes with the application before it dispatches a request. Another such context variable is `g` that is used in the tutorial to store the current locale. These two, along with Flask-Login's `current_user` and a few others are pretty special, because they work like global variables, but are only accessible during the handling of a request, and only in the thread that is handling it.

Therefore, for the above mentioned modules, we just replace `app` with `current_app` and make sure to import `current_app` instead of `app` as well. Things like `app.config` turn in to `current_app.config`. Just find-replace.

`app/email.py` is a bit more complicated:

```
from flask import current_app

def send_async_email(app, msg):
    with app.app_context():
        mail.send(msg)

def send_email(subject, sender, recipients, text_body, html_body):
    msg = Message(subject, sender=sender, recipients=recipients)
    msg.body = text_body
    msg.html = html_body
    Thread(target=send_async_email,
           args=(current_app._get_current_object(), msg)).start()
```

The real difference here is the passing of `current_app._get_current_object()` as the application instance to `send_async_email()`. The reason we did this is because, if we passed `current_app` to `send_async_email()`, the actual application wouldn't be accesible. That's because `current_app` is a *proxy object* that dynamically maps to the current application instance, so it would change to nothing

once passed to a new Thread. So the we need to access the *real* application instance that is stored in the proxy object and pass that as the `app` argument. `current_app._get_current_object()` is the way to extract the actual application object.

Unit Testing Improvements

From the tutorial:

> As I hinted in the beginning of this chapter, a lot of the work that I did so far had the goal of improving the unit testing workflow. When you are running unit tests you want to make sure the application is configured in a way that it does not interfere with your development resources, such as your database.
>
> The current version of tests.py resorts to the trick of modifying the configuration after it was applied to the application instance, which is a dangerous practice as not all types of changes will work when done that late. What I want is to have a chance to specify my testing configuration before it gets added to the application.

As of now, the `create_app()` function now accepts a configuration class as an argument. That class is defined in `config.py`. However, the idea is to be able to create an application instance that uses a different configuration simply by passing a new class to the factory function. Here is an example of a test configuration class that would be suitable for unit tests: `tests.py`:

```
from config import Config

class TestConfig(Config):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = 'sqlite://'
```

What this is, is a subclass of the Config class that tests `TESTING` to `True` (unnecessary, but possibly useful) and overrides the `SQLALCHEMY_DATABASE_URI` key to force the application to use an in-memory SQLite database.

Remember the `setUp()` and `tearDown()` methods that we used for creating and destroying the appropriate environment for each test to run? We can now use those to create and destroy a brand new application for each test: `tests.py`:

```
class UserModelCase(unittest.TestCase):
    def setUp(self):
        self.app = create_app(TestConfig)
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

    def tearDown(self):
```

```
        db.session.remove()
        db.drop_all()
        self.app_context.pop()
```

The new application is stored as `self.app`, but some things won't know that that's the application, such as `db.create_all()`. Do make sure that your app is findable as `current_app`, which is found dynamically, you `.push()` the app context, then `.pop()` it when done (in `tearDown()`) to wipe the slate clean.

FYI (from the tutorial):

```
You should also know that the application context is one of two contexts
that Flask uses. There is also a request context, which is more specific,
as it applies to a request. When a request context is activated right
before a request is handled, Flask's request and session variables become
available, as well as Flask-Login's current_user.
```

Environmental Variables

Our application relies on a lot of environmental variables (including your secret key, email server information, database URL, etc). This can be inconvenient when trying to run it in a new terminal window, or several other situations.

Commonly, these sort of environmental variables are often stored in a `.env` file in the root directory. The variables are imported when the application starts, meaning they don't need manually set.

Let's install the python package that supports `.env` files:

```
(venv) $ python -m pip install python-dotenv
```

Because we use all of the env variables in the `config.py` file, we should import the `.env` file immediately before needing them when the `Config` class is created: `config.py`:

```
import os
from dotenv import load_dotenv

basedir = os.path.abspath(os.path.dirname(__file__))
load_dotenv(os.path.join(basedir, '.env'))

class Config(object):
    # ...
```

Below is an example `.env` file (not set up for our email preferences). Notice that you don't want to add such a file to source control, or git, or whatever, as it has passwords that are integral to the security of the app:

```
SECRET_KEY=a-really-long-and-unique-key-that-nobody-knows
MAIL_SERVER=localhost
MAIL_PORT=25
```

Requirements File

Because our Python in our environment has been extensively modified, it can be hard to remember everything that needs installed to run the app from a clean slate. So, we create a `requirements.txt` file using `pip freeze` to track all of the packages that python has installed and to be able to install them all in one easy command.

Create the file (needs done whenever Python has anything new installed):

```
(venv) $ python -m pip freeze > requirements.txt
```

The `pip freeze` command will dump all the packages that are installed on your virtual environment in the correct format for the `requirements.txt` file. Now, if you need to create the same virtual environment on another machine, instead of installing packages one by one, you can run:

```
(venv) $ python -m pip install -r requirements.txt
```

Migrating New Code to New Structure

From here, all of the code that we've written throughout this tutorial was slightly different than the actual tutorial. So, we will use the .zip file of the code from after the "Application Structure" chapter, and we will migrate our code into the files and structure they have already created (updating some things along the way).

Moving and Deleting

We start by moving `sbMACRO.db` and `requirements.txt` to the new folder for v1.5. These should replace the current `.db` file and `requirements.txt` file.

We then create a new virtual environment using the new `requirements.txt`:

```
$ virtualenv -p python3 venv
$ source venv/bin/activate
(venv) $ python -m pip install -r requirements.txt
```

Let's delete the files we don't need:

- `cli.py`
- `translate.py`

Moving/Editing Blueprints (templates)

We also should move the specific templates for each blueprint into those folders, since we wanted to use that schema:

- **templates/emails/ -> emails/templates/**
- **templates/auth/ -> auth/templates/**
- **templates/errors/ -> errors/templates/**

This, of course, requires that we change the reference to where they are in each `__init__.py` file in each blueprint. In addition, we need to make sure it lives up to our coding standards. Example: `auth` blueprint.

Before:

```
from flask import Blueprint

bp = Blueprint('auth', __name__)

from app.auth import routes
```

After:

```
"""Initialization of the authentication blueprint."""
from flask import Blueprint

# Create a blueprint for the authentication subsystem. Define template
folder.
bp = Blueprint('auth', __name__, template_folder='templates')

from app.auth import routes
```

Then we copy the `migrations` directory (and contents) over to the new folder to replace the old one which is incorrect for the new database.

config.py

- **Create `dotenv` support (had neglected this earlier)**
  - **Create `.env` file**
  - **Add `.env` to `.gitignore`**
  - **Add self-defined secret key**
- **Delete unused Config keys (eg translation-related, or post-related)**
- **Add helpful comments from our file**
- **Add appropriate coding styling to fit protocol (Docstrings, variable names, etc)**

Result:

```
"""Module containing the master configuration class for entire
application."""
import os
from dotenv import load_dotenv

BASEDIR = os.path.abspath(os.path.dirname(__file__))
load_dotenv(os.path.join(BASEDIR, '.env'))


class Config(object):
    """Master configuration class for entire application."""

    SECRET_KEY = os.environ.get(
        'SECRET_KEY') or 'odm93hj0pGHG[p03i{()UGHH=AKHHAS0D3THTHE900ANN'
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(BASEDIR, 'sbmacro.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False

    # Email server details:
    MAIL_SERVER = os.environ.get('MAIL_SERVER')
    MAIL_PORT = int(os.environ.get('MAIL_PORT') or 25)
    # Bolean flag to enable encrypted connections:
    MAIL_USE_TLS = os.environ.get('MAIL_USE_TLS') is not None
    MAIL_USERNAME = os.environ.get('MAIL_USERNAME')
    MAIL_PASSWORD = os.environ.get('MAIL_PASSWORD')
    ADMINS = ['ad.sbmacro@gmail.com'
             ]  # Must be changed once hosted. Is a list.
```

sbmacro.py

- **Add new db items to return statement**
- **Delete** `cli.register(app)`
- **Delete** `import`**s for cli related stuff.**
- **Add appropriate code styling to fit protocol (Docstrings, variable names, etc)**
- **Add end lines to start app if script is run (**`if __name__== "__main__": app.run(debug=True)`**)**

Result

```
"""Module for app instantiation and shell context creation."""
from app import create_app, db
from app.models import User, casc, FiscalYear, Project, Item, SbFile
from app.models import ProblemItem

app = create_app() # pylint: disable=C0103


@app.shell_context_processor
def make_shell_context():
```

```
    """Define shell context for FLASK_SHELL and import model classes.

    Returns:
        db -- SQLite database instance.
        User -- User database model class
        casc -- casc database model class
        FiscalYear -- FiscalYear database model class
        Project -- Project database model class
        Item -- Item database model class
        SbFile -- SbFile database model class
        ProblemItem -- ProblemItem database model class

    """
    return {
        'db': db,
        'User': User,
        'casc': casc,
        'FiscalYear': FiscalYear,
        'Project': Project,
        'Item': Item,
        'SbFile': SbFile,
        'ProblemItem': ProblemItem
    }

if __name__ == "__main__":
    app.run(debug=True)
```

tests.py

- **Add `TestConfig` class**
- **Add third test to `test_password_hashing(self)`**
- **Delete remainder of useless tests (following, posting, avatar, etc.)**
- **Add (useless) CascModelCase test.**

Result:

```
#!/usr/bin/env python
"""Module containing application unit tests."""
from datetime import datetime, timedelta
import unittest
from app import create_app, db
from app.models import User, casc, FiscalYear, Project, Item
from app.models import SbFile, ProblemItem
from config import Config


class TestConfig(Config):
    """Master testing configuration, creates in-memory db and sets
TESTING."""
```

```
    TESTING = True
    SQLALCHEMY_DATABASE_URI = 'sqlite://'


class UserModelCase(unittest.TestCase):
    """Test suite for User DB Model."""

    def setUp(self):
        """Create new app initialization with in memory database."""
        self.app = create_app(TestConfig)
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

    def tearDown(self):
        """Clear in-memory DB and pops the app context off the stack."""
        db.session.remove()
        db.drop_all()
        self.app_context.pop()

    def test_password_hashing(self):
        """Test suite for password hashing."""
        u = User(username='susan') # pylint: disable=C0103
        u.set_password('cat')
        self.assertFalse(u.check_password('car'))
        self.assertFalse(u.check_password('caT'))
        self.assertTrue(u.check_password('cat'))

class CascModelCase(unittest.TestCase):
    """Test suite for CASC DB Model."""

    def setUp(self):
        """Create new app initialization with in memory database."""
        self.app = create_app(TestConfig)
        self.app_context = self.app.app_context()
        self.app_context.push()
        db.create_all()

    def tearDown(self):
        """Clear in-memory DB and pops the app context off the stack."""
        db.session.remove()
        db.drop_all()
        self.app_context.pop()



if __name__ == '__main__':
    unittest.main(verbosity=2)
```

---

node_modules

`node_modules/` **needed copied over to the new code folder.**

---

app/init.py

- **Delete babel and bootstrap** `import`**s**
- **Delete babel and bootstrap initializations**
- **Delete** `@babel.localeselector` **decorator and** `get_locale()` **function**
- **Add appropriate code syling and comments**
- **"microblog.log" -> "sbmacro.log"**
- **"Microblog startup" -> "sbMACRO startup"**

app/email.py

- **Add appropriate code syling**
- **Make sure** `args=(app, msg)` **is now** `args=(current_app._get_current_object(), msg)`

app/models.py

- **Add appropriate code syling and comments**
- **Copy over old model fields.**
- **Remove Post class**
- **Add classes: CASC, FiscalYear, Project, Item, SbFile, ProblemItem**

Shuffling Templates

We hae a lot of different templates for a lot of different systems that need to go to a lot of different places. The new system we are integrating into has a lot that we don't use, and is missing several that we do.

- **Delete all templates in new system.**
- **Move all old templates to appropriate subsystem** `template/` **folders.**

Auth Subsystem

`__init__.py` **was already done, so we will move down the list of files:**

- `email.py`
  - **Remove flask_babel** `import`
  - **Change "Microblog" -> "sbMACRO"**
  - **Add appropriate code syling and comments**
  - **Remove** `_()` **which we don't use (babel-related)**
  - **Update paths to** `.txt` **and** `.html` **templates.**
- `forms.py`
  - **Remove babel-related imports**
  - **Replace** `LoginForm`
  - **Replace** `RegistrationForm`
  - **Replace** `ResetPasswordRequestForm`
  - **Replace** `ResetPasswordForm`
  - **Add appropriate code syling and comments**

- routes.py
  - **Remove babel-related imports**
  - **Add appropriate code syling and comments**
  - **Remove any** `_()` **and some flash messages (as we don't use flash)**
  - **Replace** `reset_password_request()`
  - **change validated form** `render_template()` **to the correct one.**

Errors Subsystem

Again, `__init__.py` **was already done, so we will move down the list of files:**

- handlers.py
  - **Copy over all handler functions**
  - **change** `@app.errorhandler()` **decorator to** `@bp.errorhandler()`
  - **Add appropriate code syling and comments**

Main Subsystem

Finally, as we know, `__init__.py` **was already done, so we will move down the list of files:**

- forms.py
  - **Remove babel-related imports**
  - **Replace** `EditProfileForm`
  - `import`**, from** `wtforms.validators` `Email`**,** `PasswordField`**, and** `Optional`
  - **Add appropriate code syling and comments**
- routes.py
  - **Remove babel-related imports (incl. guess_language)**
  - **Add appropriate code syling and comments**
  - **Remove any** `_()` **and some flash messages (as we don't use flash)**
  - **Delete** `g.locale = str(get_locale())`
  - **Delete** `explore()`
  - **Replace** `index()`
  - **Add** `fiscalyear()`**,** `project()`**, and** `report()` **routes.**
  - **Add** `fiscalyear()`**,** `project()`**, and** `report()` **route decorators to blueprints**
  - **Replace** `user()`
  - **Replace** `edit_profile()` **and change url_for('user') to url_for('main.user')**
  - **Delete** `follow()`**,** `unfollow()` **and** `translate_text` **routes.**
  - **Remove PostForm from** `app.main.forms` **import**
  - **Remove** `app.translate` **import**
  - **Remove 'Post' import from** `app.models`

Debugging After Merging

`dotenv` not installed

`python-dotenv` **was not installed. We had to install it and add it to** `requirements.txt`**.**

```
python -m pip install python-dotenv
python -m pip freeze > requirements.txt
```

Could not import sbmacro

After that, cli could not import 'sbmacro'. Changed `sbMACRO.py` **to** `sbmacro.py`**.**

`ImportError: cannot import name 'PasswordField'`

`PasswordField` **import in** `app/main/forms.py` **was moved from** `wtforms.validators` **to** `wtforms`**.**

jinja2.exceptions.TemplateNotFound: auth/login.html

Needed to remove `auth/` **from all routes in** `app/auth/routes.py`**, then move** `password_reset_request.html`**,** `post_pass_reset_request.html`**,** `register.html`**,** `reset_password.html`**, and** `successful_pass_reset.html` **from** `main/templates/` **to** `auth/templates/`**. This required moving both** `reset_password.html` **and** `reset_password.txt` **(the templates for the reset password email), to be moved into a new folder** `app/auth/templates/email/`**, which required** `app/auth/email.py` **to change all references to prepend 'email/' to both** `reset_password.html` **and** `reset_password.txt`**.**

werkzeug.routing.BuildError: Could not build url for endpoint 'register'. Did you mean 'auth.register' instead?

I certainly did mean that. The issue was in the templates. The `url_for()` **calls had not been updated to reflect the new blueprints.**

- `main/templates/login.html`**:**
  - **Change** `register` **to** `auth.register`
  - **Change** `reset_password_request` **to** `auth.reset_password_request`
- `main/user.html`**:**
  - `edit_profile` **->** `main.edit_profile`
- **All** `errors` **templates:**
  - **Change** `index` **to** `main.index`
- `app/auth/templates/email/reset_password.html` **and** `app/auth/templates/email/reset_password.txt`**:**
  - `reset_password` **->** `auth.reset_password`
- `/app/auth/templates/post_pass_reset_request.html`**:**
  - `reset_password_request` **->** `auth.reset_password_request`
- `/app/auth/templates/successful_pass_reset.html`**:**
  - `login` **->** `main.login`

werkzeug.routing.BuildError: Could not build url for endpoint 'reset_password' with values ['token']. Did you mean 'auth.reset_password' instead?

Same issue as above. I searched for 'url_for' in `app/auth/routes.py` **and replaced one missing** `main.`**. That obviously isn't the problem, so I moved on.**

Found that `app/auth/templates/email/reset_password.html` was missing an 'auth.' before 'reset_password'. Maybe that was it?

It appears to have worked. But, when I tried to reset my password:

werkzeug.routing.BuildError: Could not build url for endpoint 'main.login'. Did you mean 'auth.login' instead?

Changed `app/auth/templates/email/successful_pass_reset.html` reference to 'main.login' to 'auth.login'. I also moved the `login.html` template into the `app/auth/templates` folder.

'Invalid username or password' is flashed, not shown on form

The flash is fine, but it was flashed twice each time, so I had to go to `base.html` and edit the `<div>` that had the flash message. Then I added a `flash-div` class to that div and made a note to give it some better css.

Url for login has '/auth/' prepended to 'login'

Decided I'm ok with this. I added the prepending `/error/` to the errors subsystem as well (you do this when importing the blueprints in `app/__init__.py`).

Created New Unit Test for Password Resetting

Just as the title here says. I added this test to the `User` model:

```
#...

    def test_password_reset(self):
        """Test suite for token creation, verifying, and pass reset."""
        # u = User(username='susan_belinda7456789142') # pylint:
disable=C0103
        # u.set_password('cat')
        # db.session.add(u)
        # db.session.commit()
        u =
User.query.filter_by(username='susan_belinda7456789142').first()
        self.assertIsNotNone(u, msg="Could not find test user.")
        token = u.get_reset_password_token()
        self.assertIsNotNone(token, msg="Token is 'None'")
        self.assertTrue(User.verify_reset_password_token(token),
                        msg="Failed to verify pass reset token.")
        u.set_password('dog')
        self.assertFalse(u.check_password('dot'))
        self.assertFalse(u.check_password('doG'))
        self.assertTrue(u.check_password('dog'))
#...
```

Error Handling Subsystem was not functioning

404 errors were not being handled, nor were other ones. I searched long and hard to find that I hadn't used the right decorator. Instead of the correct `@bp.app_errorhandler(404)`, I had used `@bp.errorhandler(404)` from before the blueprinting. It now works fine.

Usernames are case sensitive

This shouldn't be the case. They should all be lower case, and when typed into the address bar, it should be converted to lowercase.

This requires that

1. **Any username typed is converted to lower case and the user is told it is not case sensitive.**
2. **Any username in URLs is converted to lowercase.**

The only username entered in a URL is the one for seeing a user profile in the `main` subsystem. I simply made sure to search for the username in the database as `User.query.filter_by(username=username.lower()).first_or_404()`, using `.lower()` to transform the provided username to lowercase.

I then needed to go to the forms and make sure that the usernames were always converted.

- `main` **subsystem**
  - **Edit Profile --** `validate_username` **Form validation**
    - **changed query from** `filter_by(username=username.data)` **to** `filter_by(username=username.data.lower())`
    - `@bp.route('/edit_profile')` **in** `main/routes.py`
      - **Changed the form submit action to change the current user's username to a lowercase verson of the string: from** `current_user.username = str(form.username.data)` **to** `current_user.username = str(form.username.data).lower()`
- `auth` **subsystem**
  - **Login**
    - **Did not edit the form, but processed the data in the** `/login` **route using** `.lower()`**: from** `user = User.query.filter_by(username=form.username.data).first()` **to** `user = User.query.filter_by(username=form.username.data.lower()).first()`
  - **Register**
    - **Added** `.lower()` **to the post-submit processing for the username.**

v2: Merging v1 and v1.5

Now is where the rubber meets the road, and where we combine all of the new code we've created with the old code to create our working web app that should, theoretically, be ready to host.

This means that we start with creating a new folder for v2, and merge all of the code together, debugging as we go. After it is all successfully merged, we will stop tracking any other folder except this one. It should be huge changes. At that point, we will add all of our changes to the master branch and v2 will be completed.

Steps:

1. **Create folder `sbMACROv2`**
2. **Move new code to new folder.**
3. **Move old algorithm to new code**

   3a. Create `sb_data_gather` package

   3b. Change from json -> database

   3c. Run algorithm to populate database

   3d. Make sure database is correctly populated

   3e. Add new unit tests
4. **Merge old `app.py` with new `sbmacro.py` 4a. Merge old routes to new code**
5. **Move old templates to new code**
6. **test that app is working (aside from data)**
7. **Make sure app routes now access database, and any other db access is changed from `jsonCache2` to the new db.**
8. **Bring old scripts to new code (google sheets, test scripts, etc.)**
9. **Test all additions, creating unit tests where necessary.**
10. **Update front end javascript to be clean and use more Flask features.**

1-2. Create folder `sbMACROv2` and move new code there

This was done easily.

3. Move old algorithm to new code

This is a significant process. It will be broken into steps.

3a. Create `sb_data_gather` package

We will start by creating a package in a new `bin/` directory that will contain the references to and code for the "sb algorithm".

To create our package, we create a new folder, `sb_data_gather/` within the `bin/` directory. Inside that `sb_data_gather` directory, we create an `__init__.py` file, which we can use to set paths, import modules as our package API, etc. It will let Python know that this directory is a Python package. We then move the files in `DataCounting/` to `sb_data_gather/`. Our `__init__.py` file will look like this to allow a master `start()` function to control the algorithm:

```
"""Initialization file for sb_data_gather package."""
from main import full_hard_search, defined_hard_search
```

```
# To run this function from command line (add any args to 'start()'):
# python -c 'from __init__ import start; start()'
def start(defined=None):
    """Start new sb_data_gather instance.

    Depending on the provided argument, this function calls either
    defined_hard_search() or full_hard_search(). Default is
full_hard_search()
    if no arg is provided.
    Args:
        defined -- (string, optional) if provided as "defined", calls
                   defined_hard_search(), otherwise, calls
full_hard_search().
    """
    if not defined or defined != "defined":
        full_hard_search()
    else:
        defined_hard_search()
```

`data_main.py` **was renamed** `main.py`**. This meant finding all references to** `data_main` **and replacing it with** `main`**.**

`pysb` **package is not installed in the new code. We must install it for** `sb_data_gather` **package to run.**

This requires downloading the zip file from here and running `python setup.py install`

`pysb` **also requires the** `requests` **package, so that needed installed as well as the requirements frozen into requirements.txt:**

```
(venv) python -m pip install requests
(venv) python -m pip freeze > requirements.txt
```

3b. Change from json -> database

An error then occurred because `jsonpickle` **was required, but not installed. However, we are no longer creating JSONs, so we can do away with that** `import` **as well as any references to it.**

- `import jsonpickle` **was removed**
- **Docstrings mentioning jsons were changed to mention the database.**
- **The** `JsonTransformer` **class and methods were deleted.**
- `save_json()` **was deleted in** `main`**.**

Before starting anything else, we need to establish a connection to our database and be able to add Users, cscs, Fiscal Years, etc. We will be using `flask_sqlalchemy`**, as we did before because we want all of our models kept in one place (and we already defined them).**

Note: The relative path to the db from the modules in the package is `../../sbmacro.db`***.***

First, we must add the basics for creating an application instance. We start with defining a new `Config class` in our package's `__init__.py` that is an extention of the original `Config`, but with a new relative path to the db. Then we create the app instance:

```
class DataGatherConfig(Config):
    """Master Data Gather Algorithm config, connects to relative db."""

    SQLALCHEMY_DATABASE_URI = 'sqlite://../../sbmacro.db'


app_instance = create_app(DataGatherConfig)  # pylint: disable=C0103
```

We want our algorithm to have access to the models and the db, so we can create a dictionary with all of these things and pass it into `full_hard_search()` and `defined_hard_search()`.

```
app = {
    'app': app_instance,
    'db': db,
    'User': User,
    'casc': casc,
    'FiscalYear': FiscalYear,
    'Project': Project,
    'Item': Item,
    'SbFile': SbFile,
    'ProblemItem': ProblemItem
}
```

We must make sure that `app` is passed into `full_hard_search()` and `defined_hard_search()` within `start()`.

This is actually wrong, as I found out after trying to run the program. Instead, we crate another class, initialize it, and send it through. Here's the class and initialization:

```
class App(object):
    """Object containing important application references."""

    app = app_instance
    db = db
    User = User
    casc = casc
    FiscalYear = FiscalYear
    Project = Project
    Item = Item
    SbFile = SbFile
    ProblemItem = ProblemItem

app = App()
```

After a bunch of debugging the final `__init__.py` that was able to pass the `App class` to the workhorse functions was as follows:

```
"""Initialization file for sb_data_gather package."""
import sys
import os
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
LOC = os.path.dirname(os.path.realpath(__file__))
# LOC == sb_data_gather
LOC = os.path.dirname(LOC)
# LOC == bin
LOC = os.path.dirname(LOC)
# LOC == sbMACRO
sys.path.insert(0, LOC)
from app.models import User, casc, FiscalYear, Project, Item, SbFile
from app.models import ProblemItem
from main import full_hard_search, defined_hard_search
from config import Config


class DataGatherConfig(Config):
    """Master Data Gather Algorithm config, connects to relative db."""

    SQLALCHEMY_DATABASE_URI = 'sqlite:///../../sbmacro.db'



app = Flask(__name__)
app.config.from_object(DataGatherConfig)
db = SQLAlchemy(app)

print("db in __init__.py in package: {}".format(db))
class App(object):
    """Object containing important application references."""

    def __init__(self):
        "Initializes App class object."
        print("creating App object...")
        print("Using app_instance: {}".format(app))
        print("Using db: {}".format(db))
        self.app = app
        self.db = db
        self.User = User
        self.casc = casc
        self.FiscalYear = FiscalYear
        self.Project = Project
        self.Item = Item
        self.SbFile = SbFile
        self.ProblemItem = ProblemItem
```

```
APP = App()

# To run this function from command line (add any args to 'start()'):
# python -c 'from __init__ import start; start()'
def start(defined=None):
    """Start new sb_data_gather instance.

    Depending on the provided argument, this function calls either
    defined_hard_search() or full_hard_search(). Default is
full_hard_search()
    if no arg is provided.
    Args:
        defined -- (string, optional) if provided as "defined", calls
                   defined_hard_search(), otherwise, calls
full_hard_search().
    """
    if not defined or defined != "defined":
        full_hard_search(APP)
    else:
        defined_hard_search(APP)
```

This new passing of `app` to `full_hard_search()` and `defined_hard_search()` required additions to both functions' docstrings as well.

In fact, many other functions and modules needed access to `app`. So, `parse_fiscal_years()` in `fiscal_years.py` was edited to accept `app` as an argument (the docstring was also edited appropriately).

A replacement for `save_json()` is now necessary. We create a `save_db()` function in `main.py` to replace `save_json()`. However, it would be best to modularize and have a function for each level of sb item (CASC, FY, Project, etc). Therefore, we create a new module called `db_save.py` and `import` it into `main.py` and call each function.

However, when looking at adding a `save_casc()` function, I noticed that we only took note of the casc name. This was resolved by loking at `get_csc_from_fy_id()` in `fiscal_years.py`. I added a new optional second boolean argument that, if set to `True`, returns more than just the CASC name, but also the URL, and the science base ID.

So, I worked through creating functions to save cascs, FYs, projects, etc to the database. I also realized that `SbItem`s should have a `total_data` field. Also, that all `total_data` should be tabulated at the end, after all `SbFiles` are accounted for.

~~**In addition, I noticed that the database did not allow for the `Item` to have a `size` field. This does not work. I need to add a size field as sometimes the item contains files and have a size itself. The original `sb` classes in `gl.py` have a `size` attribute.**~~ **This was not necessary. `check_for_files()` in `gl.py` finds the `total_data` of the `Item` by tallying up all the files and extentions.**

However, the `SbFile` model in `models.py` did not give the `sb_id` field enough characters, as there is no `sb_id` for a file. Instead, we should use "path_on_disk" from the file's json. Therefore, I made some changes to the `SbFile` model after looking at a file json.

- **deleted the `sb_id` field**
- **deleted both `start_date` and `end_date`, the parent item should have that, and the `date_uploaded` is more appropriate for a file.**
- **deleted `file_count`, as a file does not have numerous other files. It is the smallest item.**
- **added `content_type` field, as there is a json field for it, and it could be quite useful.**

This requires changes to the database, of course. First, we generate the migration script for the changes to our db schema(`python -m flask db migrate -m "message"`). Then we apply those changes to the database:

```
(venv) $ python -m flask db migrate -m "Update SbFiles table with
content_type, delete unused"
(venv) $ python -m flask db upgrade
```

Now `db_save.py` should, theoretically, be ready to use in place of the previous save operations. However, to replace `save_json()` that we deleted from `main.py`, we now need to complete `save_to_db()`, which will call all of the functions in the new `db_save.py` module.

NOTE: for `defined_hard_search`, we need to have a way to update the data of all 'parent' items by the difference we found between before and after the search. This would be solved by tabulating data at the end. This means NOT using previously calculated `total_data` from the algorithm and calculating it again. This means that the `total_data` code in the algorithm before `save_data()` can be deleted.

Now, there are still references to `save_json()` and a lot of effort put into making the jsons. All modules must be searched for any mention of jsons, and changed, when appropriate, to database references.

- `__init__.py`
  - **There were no references to jsons, `save_json()`, etc. as it was just created.**
- `exceptions_raised.py`
  - **No mention of `save_json()`. 'json' was mentioned, but in the context of sb jsons.**
- `fiscal_years.py`
  - **`save_json()` was called in `parse_fiscal_years()`, so it was replaced with `save_to_db.py`.**
- `gl.py`
  - **No mention of `save_json()`. 'json' was mentioned, but in the context of sb jsons.**
- `main.py`
  - **No mention of `save_json()`. 'json' was mentioned, but in the context of sb jsons.**
- `projects.py`
  - **No mention of `save_json()`. 'json' was mentioned, but in the context of sb jsons.**