Flask

Contents

Flask

Basics and Routing

```python
from flask import Flask
app = Flask(__name__) # name of the module

@app.route("/") # multiple routes
@app.route("/home")
def home():
    return "<h1>Hello World!</h1>"

@app.route("/about")
def about():
    return "<h1>About page.</h1>"

@app.route("/posts/<postid>")     # variables routes
@app.route("/posts/<int:postid>") # converters: int, float, string, path
def posts(postid):
    return "Post " + postid

if __name__ == '__main__': # run in python directly
    app.run(debug=True)
```

- export the env variable `FLASK_APP=webpage.py`
  - hotsave flask app with env variable `FLASK_DEBUG=1`
  - run with `flask run`

Requests and Responses

Using query strings:

```python
from flask import request

@app.route('/hello')
```

```
def hello():
    if 'name' in request.args:
        return 'Hello ' + request.args['name']
    else:
        return 'Hello John Doe'
```

- GET request to '/hello?name=Tony' returns 'Hello Tony'

Request data and headers:

```
from flask import json

@app.route('/messages', methods=['POST'])
def message():
    if request.headers['Content-Type'] == 'text/plain':
        return 'Text Message: ' + request.data
    elif request.headers['Content-Type'] == 'application/json':
        # alternatively, request.is_json and request.get_json()
        return 'JSON Message: '+ json.dumps(request.json)
    else:
        return '415 Unsupported Media Type'
```

Handling responses:

```
from flask import Responses

@app.route('/hello', methods=['GET'])
def hello():
    data = {
        'content' : '...',
        'id' : 5
    }
    js = json.dumps(data)

    # mimetype is content-type
    resp = Response(js, status=200, mimetype='application/json')
```

```
    resp.headers['Link'] = '...'

    return resp

from flask import jsonify, make_response # alternative json parsing and response
...
if request.is_json:
    req = request.get_json()
    res_body = {
        'message': 'JSON received',
        'sender': req.get('name')
    }
    res = make_response(jsonify(res_body), 200)
...
```

Templating

```
from flask import Flask, render_template, url_for

posts = [
    {
        'author': ...,
        'title': ...,
        'content': ...,
        'date': ...,
    }
]

@app.route("/")
def home():
    return render_template('home.html', title='Home', posts=posts) # templates/home.
   html
```

Using Jinja2 templating engine in html:

```
{% if title %} # if statement
  <title>{{ title }}</title>
{% else %}
  ...
{% endif %}

{% for post in posts %} # for loop
  <h1>{{ post.title }}</h1>
  <p>By {{ post.author }} on {{ post.date }}</p>
{% endfor %}

{% extends "layout.html" %} # inheritance
{% block content %}
  ...
{% endblock content %}

{% with messages = get_flashed_messages(with_categories=true) %} # grabbing flashed
   messages
{% for category, message in messages %}
  ...
{% endfor %}

...href="{{ url_for('static', filename='main.css') }}" # static files, static/main.
   css
```

Forms

```
#forms.py

from flask_wtf import FlaskForm # wt forms flask plugin
from wtforms import StringField, PasswordField, SubmitField, BooleanField
```

```
from wtforms.validators import DataRequired, Length, Email, EqualTo

class RegistrationForm(FlaskForm):
    username = StringField('Username',
                           validators=[DataRequired(), Length(min=2, max=20)])
    email = StringField('Email',
                           validators=[DataRequired(), Email()])
    password = PasswordField('Password',
                           validators=[DataRequired()])
    confirm_password = PasswordField('Confirm Password',
                           validators=[DataRequired(), EqualTo('password')])
    remember = BooleanField('Remember me')
    submit = SubmitField('Sign Up')

    # validating username for uniqueness
    def validate_username(self, username):
        # querying our database by the user model
        user = User.query.filter_by(username=username.data)
        if user:
          raise ValidationError('That username is taken. Please choose another.')
```

Using forms in app:

```
from forms import RegistrationForm

app.config['SECRET_KEY'] = '...'

@app.route("/register", methods='GET', 'POST') # allowing for POST requests
    form = RegistrationForm()
    if form.validate_on_submit():
        # flask flash alert, second arg is custom category
        flash(f'Account created for {form.username.data}', 'success')
        # redirect to url of a route function
        return redirect(url_for('home'))
    return render_template('register.html', form=form)
```

Uploading files in forms:

```
from flask_wtf.file import FileField, FileAllowed

class UpdateAccountForm(FlaskForm):
    username = ...
    email = ...
    picture = FileField('Update Profile Picture', validators=[FileAllowed('jpg', 'png
    ')])
    ...
```

Databases

- using SQLAlchemy:
  - uses object-relational mapping, object-oriented paradigm
  - in cli:
    - `db.createAll()`
    - `db.session.add(some_user)`
    - `db.session.commit()`
    - `db.drop_all()`
    - can query all, first, filter by search, get by id
      - eg. `User.query.filter_by(username='Bob')` to query all user database models by username
      - eg. `Post.query.first()` to query first post database model “‘py #**init**.py

from flask_sqlalchemy import SQLAlchemy

#set up local sqlite database app.config[‘SQLALCHEMY_DATABASE_URI’] = ‘sqlite:///site.db’ # /// indicates relative path db = SQLAlchemy(app)

#models.py

from flaskapp import db

class User(db.Model): # database models are classes # instance of class is a table with columns

```
# primary key asigned automatically
id = db.Column(db.Integer, primary_key=True)
username = db.Column(db.String(20), unique=True, nullable=False)
email = db.Column(db.String(120), unique=True, nullable=False)
image_file = db.Column(db.String(20), nullable=False, default='default.jpg')
password = db.Column(db.String(60), nullable=False)

# User makes Posts (posts is an attribute in each user), backref is an attribute in
    posts,
# specify lazy loading
posts = db.relationship('Post', backref='author', lazy=True)
```

class Post(db.Model): ... # table and columns are lowercase user_id = db.Column(db.Integer, db.ForeignKey(‘user.id’), nullable=False)

```
- encrypting passwords:
  - `from flask_bcrypt import Bcrypt`
  - `bcrypt.generate_password_hash('passwd').decode('utf-8')`
  - `bcrypt.check_password_hash(hashed, 'passwd')`

Implementing authentication logic on registration page:
```py
@app.route("/register", methods='GET', 'POST') # allowing for POST requests
    form = RegistrationForm()
    if form.validate_on_submit():
        hashed_pw = bcrypt.generate_password_hash(form.password.data).decode('utf-8')
        user = User(username=form.username.data, email=form.email.data, password=
    hashed_pw)
        db.session.add(user)
        db.session.commit()
        flash('You are now able to log in.', 'success')
        return redirect(url_for('login'))
    return render_template('register.html', form=form)
```

- implementing a login system using flask_login:
```

```python
#models.py

from flaskapp import db, login_manager
from flask_login import UserMixin

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

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

#routes.py

from flask_login import login_user, logout_user, current_user

@app.route("/login", methods=['GET', 'POST'])
def login():
    # if user already logged in
    if current_user.is_authenticated:
        return redirect(url_for('home'))
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user and bcrypt.check_password_hash(user.password, form.password.data):
            login_user(user, remember=form.remember.data)
            next_page = request.args.get('next') # next page in url query
            return redirect(next_page) if next_page else redirect(url_for('home'))
        else:
            flash('Login unsuccesfful.', 'danger')
    return render_template('login.html', title='Login', form=form)

@app.route("/logout")
def logout():
    logout_user()
```

```
    return redirect(url_for('home'))
```

CRUD Example

```
#routes.py

@app.route("/post/new", methods=['GET', 'POST'])
@login_required
def new_post():
    form = PostForm()
    if form.validate_on_submit():
        post = Post(title=form.title.data, content=form.content.data, author=
    current_user)
        db.session.add(post)
        db.session.commit()
        flash('Your post has been created.', 'success')
        return redirect(url_for('home'))
    return render_template('create_post.html', title='New Post', form=form)

@app.route("/home")
def home():
    posts = Post.query.all()
    return render_template('home.html', posts=posts)

@app.route("/post/<int:post_id>") # route variables
def post(post_id):
    post = Post.query.get_or_404(post_id) # 404 missing error
    return render_template('post.html', title=post.title, post=post)

@app.route("/post/<int:post_id>/update", methods=['GET', 'POST'])
@login_required
def update_post(post_id):
    post = Post.query.get_or_404(post_id)
```

```
    if post.author != current_user:
        abort(403) # 403 forbidden error
    form = PostForm()
    if form.validate_on_submit():
        post.title = form.title.data
        post.content = form.title.content
        db.session.commit()
        flash('Your post has been updated.', 'success')
        return redirect(url_for('post', post_id=post.id))
    elif request.method == 'GET':
        form.title.data = post.title
        form.content.data = post.content
    return render_template('create_post.html', title='Update Post', form=form)

@app.route("/post/<int:post_id>/delete", methods=['POST'])
@login_required
def delete_post(post_id):
    post = Post.query.get_or_404(post_id)
    if post.author != current_user:
        abort(403)
    db.session.delete(post)
    db.session.commit()
    flash('Your post has been deleted.', 'success')
    return redirect(url_for('home'))
```

Package Structure

- common importing errors:
  - running script will overwrite its `__name__` as `__name__`
  - cyclic import dependencies
- turn the entire application into a package using an `__init__.py` file:
  - parent folder of init py is now a package
  - other modules inside package can import app from flaskapp
  - other modules inside package can import from other modules from flaskblog.modulename

Simple package `__init__.py`:

```
#run.py
from flaskapp import app

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

#__init__.py
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_bcrypt import Bcrypt
from flask_login import LoginManager

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db' # triple slash indicates
   relative path
db = SQLAlchemy(app)
bcrypt = Bcrypt(app)
login_manager = LoginManager(app)
login_manager.login_view = 'login' # default login page
login_manager.login_message_category = 'info'

from flaskblog import routes # still avoiding circular import issues
```

Pagination

- *paginate* data to load chunks of data at a time
- `posts = Post.query.paginate()`
  - `posts.per_page` defaults to 20
    - `Post.query.paginate(per_page=5)` specifies page limit
  - `posts.page` defaults at 1
    - `Post.query.paginate(page=2)` gives desired page
  - `for post in posts.items` to iterate through them
  - `posts.total` gives total number of posts

  - `posts.iter_pages(left_edge=1, right_edge=1, left_current=1, right_current=2)`
    * divides up page numbers
    * None values can be shown with ellipses

Pagination example:

```
@app.route("/home")
def home():
    page = request.args.get('page', 1, type=int)         # setting a query parameters
    posts = Post.query.order_by(post.date_posted.desc()) # ordering by latest posts
                .paginate(page=page, per_page=5)         # rather than querying all
   posts
    return render_template('home.html', posts=posts)

@app.route("/user/<string:username>") # only showing specific user's posts
def user_posts(username):
    page = request.args.get('page', 1, type=int)
    user = User.query.filter_by(username=username).first_or_404()
    posts = Post.query.filter_by(author=user)\
                .order_by(post.date_posted.desc())\
                .paginate(page=page, per_page=5)
    return render_template('user_posts.html', posts=posts, user=user)
```

Email and Password Reset

- creating a time sensitive token:
  - `from itsdangerous import TimedJSONWebSignatureSerializer as Serializer`
  - `s = Serializer('secret', 30)` with secret key, expires after 30 seconds
  - `token = s.dumps({'user_id': 1}).decode('utf-8')`
  - `s.loads(token)` gives back desired payload
  - after time limit, attempting to load token gives a TimeExpired error

User sign up model:

```
class User(db.model, UserMixin):
```

```
    ...
    def get_reset_token(self, expires_sec=1800):
        s = Serializer(app.config['SECRET_KEY'], expires_sec)
        return s.dumps({'user_id': self.id}).decode('utf-8')

    @staticmethod # no need to access class or instance attributes
    def verify_reset_token(token):
        s = Serializer(app.config['SECRET_KEY'])
        try:
            user_id = s.loads(token)['user_id']
        except:
            return None
        return User.query.get(user_id)
    ...
```

User administration forms:

```
class RequestResetForm(FlaskForm):
    email = ...
    submit = ...

    def validate_email(self, email):
        user = User.query.filter_by(email=email.data).first()
        if user is None:
            raise ValidationError('There is no account with that email.')

class ResetPasswordForm(FlaskForm):
    password = ...
    comfirm_password = ...
    submit = ...
```

User administration routes:

```
def send_reset_email(user): # from flask_mail import Mail, Message
    token = user.get_reset_token()
```

```
    msg = Message('Password Reset Request',
                  sender='noreply@demo.com',
                  recipients=[user.email])
    msg.body = 'To reset your password, visit the following link: {}'
               .format(url_for('reset_token', token=token, _external=True))
    mail.send(msg)

@app.route("/reset_password", methods=['GET', 'POST'])
def reset_request():
    if current_user.is_authenticated:
        return redirect(url_for('home'))
    form = RequestResetForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        send_reset_email(user)
        flash('An email has been sent.', 'info')
        return redirect(url_for('login'))
    return render_template('reset_request.html', title='Reset Password', form=form)

@app.route("/reset_password/<token>", methods=['GET', 'POST'])
def reset_token(token):
    if current_user.is_authenticated:
        return redirect (url_for('home'))
    user = User.verify_reset_token(token)
    if not user:
        flash('That is an invalid or expired token', 'warning')
        return redirect(url_for('reset_request'))
    form = ResetPasswordForm()
    if form.validate_on_submit():
        hashed_password = bcrypt.generate_password_hash(form.password.data)
        user.password = hashed_password
        db.session.commit()
        flash('Your password has been updated.')
        return redirect(url_for('login'))
    return render_template('rest_token.html', title="Reset Password", form=form)
```

Application Configuration

Flask Blueprints

Use flask blueprints to split app into modular packages:

```
#main/
#main/__init__.py
#main/routes.py

from flask import Blueprint, render_template, request
from flaskblog.models import Post

main = Blueprint('main', __name__)

@main.route("/home")
...
```

Posts blueprint:

```
#posts/
#posts/__init__.py
#posts/routes.py

from flask import (Blueprint, render_template, url_for, flash,
                   redirect, request, abort)
from flask_login import current_user, login_required
from flaskblog import db
from flaskblog.models import Post
from flaskblog.posts.forms import PostForm

posts = Blueprint('posts', __name__)
```

```
@posts.route("/post/new")
...

#posts/forms.py

from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField, TextAreaField
from wtforms.validators import DataRequired

class PostForm(FlaskForm):
    ...
...
```

Users blueprint:

```
#users/
#users/__init__.py
#users/routes.py

from flask import Blueprint, render_template, url_for, flash, redirect, request
from flask_login import login_user, current_user, logout_user, login_required
from flaskblog import db, bcrypt
from flaskblog.models import User, Post
from flaskblog.users.forms import (...)
from flaskblog.users.utils import save_picture, send_reset_email

users = Blueprint('users', __name__)

@users.route("/register")
...
@users.route("/login")
...

#users/forms.py
```

```
from flask_wtf import FlaskForm
from flask_wtf.file import FileField, FileAllowed
from wtforms import StringField, PasswordField, SubmitField, BooleanField
from wtforms.validators import DataRequired, Length, Email, EqualTo, ValidationError
from flask_login import current_user
from flasblog.models import User

class RegistrationForm(FlaskForm):
    ...
...

#users/utils.py

import os
import secrets
from PIL import Image
from flask import url_for
from flask_mail import Message
from flaskblog import app, mail

def save_picture ...
def send_reset_email ...

#must also update all url_for('link') calls as url_for('blueprint.link') !
```

Registering Blueprints

```
#config.py

import os

class Config:
    SECRET_KEY = ...
    SQLALCHEMY_DATABASE_URI = ...
```

```
    MAIL_SERVER = ...
    MAIL_PORT = ...
    ...

#__init__.py

from flasblog.config import Config

#initializing extensions here
db = SQLAlchemy() # instead of SQLAlchemy(app)
...

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

    # passing in app to extensions
    db.init_app(app)
    ...

    from flaskblog.users.routes import users
    from flaskblog.posts.routes import posts
    from flaskblog.main.routes import main

    app.register_blueprint(users)
    app.register_blueprint(posts)
    app.register_blueprint(main)

    return app

#must also update all imports / references to app:
#from flask import current_app

#run.py
from flasblog import create_app
```

```
app = create_app()

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

Modularization

Blueprints

- **blueprints** allow an application to be modularized
  - set of operations that can be *registered*
  - blueprints can be registered at a URL prefix
    - can be done multiple times with different URL rules
  - allows for blueprint specific templates, static files, etc.
  - blueprints can have specifc error handlers
    - 404 error handlers should still be defined at the application level

Basic blueprint example:

```
from flask import Blueprint, render_template, abort

simple_page = Blueprint('simple_page', __name__,
                        template_folder='templates')

@simple_page.route('/', defaults={'page': 'index'})
@simple_page.route('/<page>')
def show(page):
    try:
        return render_template('pages/%s.html' % page)
    except TemplateNotFound:
        abort(404)
```

Registering a blueprint:

```
from flask import Flask
from application.simple_page import simple_page

app = Flask(__name__)
app.register_blueprint(simple_page, url_prefix='/pages')
```

Custom error pages with blueprints:

```
from flask import Blueprint, render_template

errors = Blueprint('errors', __name__)

#errorhandler would be blueprint specific
@errors.errorhandler(404)
#app_errorhandler registered for the entire application
@errors.app_errorhandler(404)
def error_404(err):
    print(err)
    return render_template('errors/404.html'), 404
```

Pluggable Views

- **pluggable views**: *class* based definition for views/routes instead of functions:
  - allows for *inheritance*:
    - generic classes can be adapted to other models and templates
  - quickly create CRUD API's with default views

Converting a simple view to a class:

```
#as a function view:
@app.route('/users/')
def show_users():
    users = User.query.all()
    return render_template('users.html', users=users)
```

```
#as a class based view:
from flask.views import View

class ShowUsers(View):
    # specify methods view supports
    methods = ['GET']

    # all class views implement dispatch_request
    def dispatch_request(self):
        users = User.query.all()
        return render_template('users.html', users=users)

#convert class to view function using as_view
app.add_url_rule('/users/', view_func=ShowUsers.as_view('show_users'))
```

Using inheritance:

```
class ListView(View):

    def get_template_name(self):
        raise NotImplementedError()

    def render_template(self, context):
        return render_template(self.get_template_name(), **context)

    def dispatch_request(self):
        context = {'objects': self.get_objects()}
        return self.render_template(context)

class UserView(ListView):

    def get_template_name(self):
        return 'users.html'
```

```
    def get_objects(self):
        return User.query.all()
```

Using method based dispatching for RESTful APIs:

```
from flask.views import MethodView

class UserAPI(MethodView):
      # methods attribute automatically set

      # handle GET requests
      def get(self, user_id):
          if user_id is None:
              # return list of users
          else:
              # expose single user

      # handle POST requests
      def post(self):
          # create a new user

      def delete(self, user_id):
          # delete a single user

      def put(self, user_id):
          # update a single user

user_view = UserAPI.as_view('user_api')
#specifying URL rules for API
#could use a blueprint instead of app
app.add_url_rule('/users/', defaults={'user_id': None},
                 view_func=user_view, methods=['GET'])
app.add_url_rule('/users/', view_func=user_view, methods=['POST'])
app.add_url_rule('/users/<int:user_id>', view_func=user_view,
                 methods=['GET', 'PUT', 'DELETE'])
```

Application as an Instance

- running the application as an *instance* allows for:
  - mulitple instances of the application running at once
  - each can have unique configurations

Example `__init__.py` file:

```
from flask import Flask
from flask_cors import CORS
from flask_sqlalchemy import SQLAlchemy
from phonebook.config import Config

#initializing extensions
cors = CORS()
db = SQLAlchemy()

#can specify different config objects
def create_app(config=Config):

    app = Flask(__name__)
    app.config.from_object(Config)

    # initializing extensions with app instance
    cors.init_app(app)
    db.init_app(app)

    from phonebook.routes.person_list import person_list
    from phonebook.routes.person import person

    app.register_blueprint(person_list, url_prefix='/api')
    app.register_blueprint(person, url_prefix='/api')

    return app
```

User Administration with Tokens

- use *pyjwt* to encode and decode tokens

Example User model:

```
import datetime
import jwt
from project.server import app, db, bcrypt

class BlacklistToken(db.Model):
    __tablename__ = 'blacklist_tokens'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    token = db.Column(db.String(500), unique=True, nullable=False)
    blacklisted_on = db.Column(db.DateTime, nullable=False)

    def __init__(self, token):
        self.token = token
        self.blacklisted_on = datetime.datetime.now()

    def __repr__(self):
        return '<id: token: {}'.format(self.token)

    @staticmethod
    def check_blacklist(auth_token):
        res = BlacklistToken.query.filter_by(token=str(auth_token)).first()
        return True if res else False

class User(db.Model):
    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    email = db.Column(db.String(255), unique=True, nullable=False)
    password = db.Column(db.String(255), nullable=False)
```

```
    registered_on = db.Column(db.DateTime, nullable=False)
    admin = db.Column(db.Boolean, nullable=False, default=False)

    def __init__(self, email, password, admin=False):
        self.email = email
        self.password = bcrypt.generate_password_hash(
          password, app.config.get('BCRYPT_LOG_ROUNDS')
        ).decode()
        self.registered_on = datetime.datetime.now()
        self.admin = admin

    def encode_auth_token(self, user_id):
        try:
            # expiration, current time, subject (id)
            payload = {
                'exp': datetime.datetime.utcnow() + datetime.timedelta(days=0,
seconds=5)
                'iat': datetime.datetime.utcnow(),
                'sub': user_id
            }
            return jwt.encode(
              payload,
              app.config.get('SECRET_KEY'),
              algorithm='HS256'
            )
        except Exception as e:
              return e

    # static decode method, unrelated to class instance
    @staticmethod
    def decode_auth_token(auth_token):
        try:
            payload = jwt.decode(auth_token, app.config.get('SECRET_KEY'))
            is_blacklisted = BlacklistToken.check_blacklist(auth_token)
            if is_blacklisted:
                return 'Token blacklisted. Please log in again.'
```

```
            return payload['sub']
        except jwt.ExpiredSignatureError:
            return 'Signature expired. Please log in again.'
        except jwt.InvalidTokenError:
            return 'Invalid token. Please log in again.'
```

Example auth blueprint and register route:

```
from flask import Blueprint, request, make_reponse, jsonify
from flask.views import MethodView

from project.server import bcrypt, db
from project.server.models import User

auth_blueprint = Blueprint('auth', __name__)

class RegisterAPI(MethodView):

    def post(self):
        post_data = request.get_json()
        user = User.query.filter_by(email=post_data.get('email')).first()
        if not user:
            try:
                user = User(
                    email=post_data.get('email')
                    password=post_data.get('password')
                )
                db.session.add(user)
                db.session.commit()
                auth_token = user.encode_auth_token(user.id)
                resObj = {
                    'status': 'success',
                    'message': 'Successfully registered.',
                    'auth_token': auth_token.decode()
                }
```

```
                return make_reponse(jsonify(resObj)), 201
            except Exception as e:
                resObj = {
                    'status': 'fail',
                    'message': 'Some error occured. Please try again.'
                }
                return make_reponse(jsonify(resObj)), 401
        else:
            resObj = {
                'status': 'fail',
                'message': 'User already exists. Please log in.'
            }
            return make_reponse(jsonify(resObj)), 202

register_view = RegisterAPI.as_view('register_api')
auth_blueprint.add_url_rule('/auth/register', view_func=register_view. methods=['POST
    '])
```

Login route:

```
class LoginAPI(MethodView):

    def post(self):
        post_data = request.get_json()
        try:
            user = User.query.filter_by(
                email=post_data.get('email')
            ).first()
            if user and bcrypt.check_password_hash(
                user.password, post_data.get('password')
            ):
                auth_token = user.encode_auth_token(user.id)
                if auth_token:
                    resObj = {
                        'status': 'success',
```

```
                    'message': 'Successfully logged in.',
                    'auth_token': auth_token.decode()
                }
                return make_reponse(jsonify(resObj)), 200
            else:
                resObj = {
                    'status': 'fail',
                    'message': 'User deoes not exist.'
                }
                return make_reponse(jsonify(resObj)), 404
        except Exception as e:
            print(e)
            resObj = {
                'status': 'fail',
                'message': 'Try again.'
            }
            return make_reponse(jsonify(resObj)), 5004

login_view = LoginAPI.as_view('login_api')
auth_blueprint.add_url_rule('/auth/login', view_func=login_view. methods=['POST'])
```

User route requiring token authentication:

```
class UserAPI(MethodView):

    def get(self):
        auth_header = request.headers.get('Authorization')
        if auth_header:
            try:
                auth_token = auth_header.split(" ")[1]
            except IndexError:
                resObj = {
                    'status': 'fail',
                    'message': 'Bearer token malformatted.'
                }
```

```
                return make_reponse(jsonify(resObj)), 401
        else:
            auth_token = ''
        if auth_token:
            resp = User.decode_auth_token(auth_token)
            if not isinstance(resp, str):
                user = User.query.filter_by(id=resp).first()
                resObj = {
                    'status': 'success',
                    'data': {
                        'user_id': user.id,
                        'email': user.email,
                        'admin': user.admin,
                        'registered_on': user.registered_on
                    }
                }
                return make_reponse(jsonify(resObj)), 200
            resObj = {
                'status': 'fail',
                'message': resp
            }
            return make_reponse(jsonify(resObj)), 401
        else:
            resObj = {
                'status': 'fail',
                'message': 'Provide a valid auth token.'
            }
            return make_reponse(jsonify(resObj)), 401

user_view = UserAPI.as_view('user_api')
auth_blueprint.add_url_rule('/auth/status', view_func=user_view, methods=['GET'])
```

Logout route using blacklisted tokens:

```
class LogoutAPI(MethodView):
```

```
    def post(self):
        auth_header = request.headers.get('Authorization')
        if auth_header:
            auth_token = auth_header.split(" ")[1]
        else:
            auth_token = ''
        if auth_token:
            resp = User.decode_auth_token(auth_token)
            if not isinstance(resp, str):
                blacklist_token = BlacklistToken(token=auth_token)
                try:
                    db.session.add(blacklist_token)
                    db.session.commit()
                    resObj = {
                        'status': 'success',
                        'message': 'Successfully logged out.'
                    }
                    return make_reponse(jsonify(resObj)), 200
                except Exception as e:
                    resObj = {
                        'status': 'fail',
                        'message': e
                    }
                    return make_reponse(jsonify(resObj)), 200
            else:
                resObj = {
                    'status': 'fail',
                    'message': resp
                return make_reponse(jsonify(resObj)), 401
        else:
            resObj = {
                'status': 'fail',
                'message': 'Provide a valid auth token.'
            }
            return make_reponse(jsonify(resObj)), 401
```

```
logout_view = LogoutAPI.as_view('logout_api')
auth_blueprint.add_url_rule('/auth/logout', view_func=logout_view, methods=['POST'])
```

Database and Database Migrations

- easy to switch between SQL databases for development and production
  - **postgreSQL** for production, **sqlite** for testing/development
  - corresponding python packages: psycopg2, Flask-SQLAlchemy, Flask-Migrate
- *configuration* setup:
  - add `SQLALCHEMY_DATABASE_URI` field to config
- *Alembic* in Flask-Migrate manages database migrations:
  - updates a database's schema with SQLAlchemy's schema
  - using a `manage.py` script for migrations:
    - `python manage.py db init`
    - `python manage.py db migrate`
    - `python manage.py db upgrade`

Example `manage.py` for migrations:

```
import os
from flask_script import Manager
from flask_migrate import Migrate, MigrateCommand

from app import app, db

app.config.from_object(os.environ['APP_SETTINGS'])

migrate = Migrate(app, db)
manager = Manager(app)

manager.add_command('db', MigrateCommand)

if __name__ == '__main__':
    manager.run()
```

Deploying to a Linux Server

Linux Deployment

- many options for *deployment*:
  - eg. Heroku, Linux virtual machines such as Linode, AWS
  - custom url addresses usually requires a paid service
- Linode server setup:
  - create a Linode, different image and machine types
  - access server via *ssh*
  - basic setup:
    - `apt update && apt upgrade`
    - `hostnamectl set-hostname flask-server`
    - `vi /etc/hosts`, enter ip address and hostname
    - `adduser <userName>`, `adduser <userName> sudo`
  - setting up *key-based* authentication:
    - `mkdir .ssh`
    - on local machine:
      - `ssh-keygen -b 4096`
      - `scp ~/.ssh/id_rsa.pub <userName>@<ip>:~/.ssh/authorized_keys`
    - `sudo chmod 700 ~/.ssh/`, `sudo chmod 600 ~/.ssh/*`
  - disabling root login:
    - `sudo vi /etc/ssh/sshd_config`
      - disable PermitRootLogin, disable PasswordAuthentication
    - `sudo systemctl restart sshd`
  - firewall setup:
    - `sudo apt install ufw` (Uncomplicated Firewall)
    - `sudo ufw default allow outgoing`
    - `sudo ufw default deny incoming`
    - `sudo ufw allow ssh`
    - `sudo ufw allow 5000`
    - after development:
      - `sudo ufw allow http/tcp`
      - `sudo ufw delete allow 5000`

        * `sudo ufw enable`
- deploying flask app on a development server:
  - `pip freeze > requirements.txt` for python dependencies
  - `scp -r /path/to/app <userName>@<ip>:~/`
  - using a virtual environment on the server:
    * `sudo apt install python3-pip`
    * `sudo apt install python3-venv`
    * `python3 -m venv <appName>/venv`
    * `source venv/bin/activate`
    * `pip install -r requirements.txt`
  - setting up config file:
    * `sudo vi /etc/config.json`
    * `with open('/etc/config.json') as config_file:, config = json.load(config_file)`
  - running flask app:
    * `export FLASK_APP=run.py`
    * `flask run --host=0.0.0.0`
    * can access running app at :5000
- using *nginx* and *Gunicorn* to optimize app for production:
  - handles high traffic, etc.
  - nginx handles static files, gunicorn handles python code
  - `sudo apt install nginx`
  - `pip install gunicorn` in venv
  - reset nginx config:
    * `sudo rm /etc/nginx/sites-enabled/default`
    * `sudo vi /etc/nginx/sites-enabled/flaskapp`
    * `sudo systemctl restart nginx`
  - `gunicorn -w 3 run:app` (3 workers, recommended: 2xNumberOfCores+1)
- using *supervisor* to run gunicorn in the background:
  - `sudo apt install supervisor`
  - `sudo vi /etc/supervisor/conf.d/flaskapp.conf`
  - `sudo supervisorctl restart`

Nginx config:

```
server {
  listen 80;
  serer_name  <ip>;
```

```
  location /static {
    alias /home/<userName>/flaskapp/static;
  }

  location / {
    proxy_pass http://localhost:8000;
    include /etc/nginx/proxy_params;
    proxy_redirect off;
  }
}
```

Supervisor config:

```
[program:flaskapp]
directory=/home/<userName>/flaskapp
command=/home/<userName>flaskapp/venv/bin/gunicorn -w 3 run:app
user=<userName>
autostart=true
autorestart=true
stopasgroup=true
killasgroup=true
stderr_logfile=/var/log/flaskapp/flaskapp.err.log
stdout_logfile=/var/log/flaskapp/flaskapp.out.log
```

Custom Domain Name

- Namecheap, GoDaddy are domain *providers*:
  - can buy a domain name for a yearly fee
  - add Linode nameservers
- Linode has a DNS manager:
  - add a domain name and DNS records
  - set up reverse DNS

Enable HTTPS

- **HTTPS**: secure HTTP using SSL/TLS certificates
  - free *Let's Encrypt* and *Certbot* service for certificates
  - Certbot commands to set up certificates
  - `sudo ufw allow https`
- certificate must be renewed every 90 days
  - use `crontab` to automatically run scripts
  - `sudo crontab -e`
  - `30 4 1 * * sudo certbot renew --quiet`, runs 4:30 AM 1st of every month