WildWatch-RESTful

WildWatch is an app which helps people keep track of wildlife, it provides a map to add and view information on wildlife collected by both professionals and enthusiasts. It provides tools to narrow down a certain search pattern, and a way to provide data to a centralized database and get credit for the collected data, the collected data will be used in another open project.

Authors

First Name	Last Name	ID	Role in Project
Aly	Shmahell	258912	Lead/Solo Developer

Instructions

Download

```
git clone --depth 1 --single-branch --branch restful
https://github.com/AlyShmahell/WildWatch wildwatch-restful
```

Pre-Requirements

python 3.7
pip for python 3.7

Installation

open a terminal inside **wildwatch-restful**, then:

```
pip install -r requirements.txt
```

Operation

```
python server.py
```

WildWatch-RESTful Dependencies

`Flask`: a micro web framework for Python.
`Flask SQLAlchemy`: provides a database abstraction layer for flask.
`Flask RESTful`: provides a REST abstraction layer for flask.
`Flask CORS`: provides **Cross Origin Resource Sharing** support for flask.
`Flask Authorize`: provides **Access Control Lists & Role-Based Access Control** support for flask.
`Flask Login`: provides user session management for flask.
`Werkzeug`: implementation-agnostic interface between web servers and web applications, in our case it is used for header and password management.
`NLTK`: a natural language toolkit, used for word lemmatization support within our search engine.
`Scikit Learn`: a machine learning library, used for TFIDF support within our search engine.
`Pandas`: a data manipulation tool, used for data analysis within our search engine.
`OpenCV`: a computer vision library used in image conversion to base64.

WildWatch-RESTful Problems

MySQL was chosen as the default database backbone because of its compatibility with SQLAlchemy, however it was buggy on the development machine and the issues piled up, therefore SQLite was chosen as a proper replacement for rapid prototyping keeping in mind that in the future MySQL will replace it.

SQLite does not support math functions, some of which (sin, cos, arctan) are required for correct calculation of distances between longitude/latitude points on a globe, however mathematical extension functions for SQLite can be found in the `ext` directory, with the official documentation on how to compile them for the target operating system present in the main file `extension-functions.c`, these were compiled on the development machine and wrapped/loaded in `server.py`.

SQLite handles boolean operations as numerical values, while SQLalchemy supplies symbolic values, which resulted in a bug, therefore comparing distances between points has been disabled and replaced by a crude inaccurate method for the moment and once MySQL is up and running we will revert back to the correct method.

WildWatch-RESTful API

Models: `models.py`

provides database schema logic

Users

```
class Users(UserMixin,  db.Model)
```

attributes:
- id: integer
- username: string
- password: string
- fullname: string
- website: string
- bio: string
- photo: blob
- roles: foreign key

Roles

```
class Roles(db.Model,  AllowancesMixin)
```

attributes:
- id: integer
- name: string

WildLife

```
class WildLife(db.Model)
```

attributes:
- id: integer
- userid: integer
- species: string
- notes: string
- lon: float, longiture
- lat: float, latitude
- date: datetime
- photo: blob

Reports

```
class Reports(db.Model)
```

attributes:

id: integer
userid: integer
wildlifeid: integer
code: integer
text: string
resolved: boolean

ReportCodes

```
class ReportCodes(db.Model)
```

attributes:

id: integer
name: string

get_id

```
@login_manager.user_loader
def get_id(user_id)
```

loads a specific user from Users table

Routes: `routes.py`

provides resource logic allocation

has_role

```
def has_role(role=None)
```

checks whether or not a user has a specific role
parameters:
role: string

Auth

```
class Auth(Resource)
```

allocated logic for resource `/auth`

post

```
| def post()
```

url `/auth` verb `POST`
input:
username: string
password: string
fullname: string
`Semantics:` If fullname is not empty then it creates a new user account based on the provided credentials. Otherwise it establishes a Session based on the provided credentials.

delete

```
| @login_required
| def delete()
```

url `/auth` verb `DELETE`
`Semantics:` closes the Session.

AuthProfile

```
class AuthProfile(Resource)
```

allocated logic for resource `/auth/profile`

get

```
| @login_required
| @has_role('user')
| def get()
```

url `/auth/profile` verb `GET`
`output:` {'fullname': string, 'website': string, 'bio': string, 'photo': base64}
`Semantics:` returns the user profile information associated with the Session. This resource is only available to users; if the Session belongs to a curator, the return status would be 403 Forbidden.

delete

```
| @login_required
| @has_role('user')
| def delete()
```

url `/auth/profile` verb `DELETE`
`Semantics:` deletes the user account then closes the Session. This resource is only available to users; if the Session belongs to a curator, the return status would be 403 Forbidden.

AuthProfileCat

```
class AuthProfileCat(Resource)
```

allocated logic for resource `/auth/profile/{category}`

put

```
| @login_required
| @has_role('user')
| def put(category)
```

url `/auth/profile/{category}` verb `PUT`
`input:` { 'value': string | base64 }
`Semantics:` updates the user info according to the {category}, which could be: fullname, website, bio, password, or photo. This resource is only available to users; if the Session belongs to a curator, the return status would be 403 Forbidden.

AuthProfileDel

```
class AuthProfileDel(Resource)
```

allocated logic for resource `/auth/profile/{userid}`

delete

```
| @login_required
| @has_role('curator')
| def delete(userid)
```

url `/auth/profile/{userid}` verb `DELETE`
`Semantics:` deletes the user account associated with {userid}; if and only if the Session belongs to a curator and {userid} belongs to a user. Otherwise the return status would be 403 Forbidden.

AuthWildLife

```
class AuthWildLife(Resource)
```

allocated logic for resource `/auth/wildlife`

post

```
| @login_required
| @has_role('user')
| def post()
```

url `/auth/wildlife` verb `POST`
input:
type: string
species: string
notes: string
photo: base64 string
date: iso datetime string
lon: float, longitude
lat: float, latitude
`Semantics:` submits a new wildlife entry to the wildlife table. This resource is only available to users; if the Session belongs to a curator, the return status would be 403 Forbidden.

get

```
| @login_required
| def get()
```

url `/auth/wildlife` verb `GET`
arguments:
text: string
mind: iso datetime string
maxd: iso datetime string
by: string
type: array of strings
lon: float, longitude
lat: float, latitude
area: float
`output:`
if the Session belongs to a user account:
[{'wildlifeid': integer, 'type': string, 'species': string, 'photo': base64, 'notes': string, 'lon': float, 'lat': float, 'date': integer},]
if the Session belongs to a curator account:
[{'wildlifeid': integer, 'type': string, 'species': string, 'photo': base64, 'notes': string, 'lon': float, 'lat': float, 'date': integer, 'userid': integer, 'reports': [{'reportid': integer, 'code': integer, 'text': string},]},]
`Semantics:` fetches all the wildlife entries in the wildlife table based on the userâ€™s or curatorâ€™s longitude and latitude and the size of their map area, the results are filtered by first matching the filters (mind=datetime&maxd=datetime&by=string&type=[string]) part of the query to the wildlife table columns using an SQL WHERE clause, and then by matching the text part of the query to the notes column in the wildlife table using an off-the-shelf TFIDF algorithm. If the Session belongs to a curator account, the filtering process is also applied on the reports table, then only wildlife entries with unresolved reports matching their â€˜wildlifeidâ€™ will be returned in the response, with a copy of the reports added to the response.

GuestWildLifeMany

```
class GuestWildLifeMany(Resource)
```

allocated logic for resource `/guest/wildlife`

get

```
| def get()
```

url `/guest/wildlife` verb `GET`
arguments:
text: string
mind: iso datetime string
maxd: iso datetime string
by: string
type: array of strings
lon: float, longitude
lat: float, latitude
area: float
`output:` [{'wildlifeid': integer, 'type': string, 'species': string, 'photo': base64, 'notes': string, 'lon': float, 'lat': float,, 'date': integer},]
`Semantics:` fetches all the wildlife entries in the wildlife table based on the guestâ€™s chosen longitude and latitude and the size of their map area, the results are filtered first by matching the filters (mind=datetime&maxd=datetime&by=string&type=[string]) part of the query to the wildlife table columns using an SQL WHERE clause, and then by matching the text part of the query to the notes column in the wildlife table using an off-the-shelf TFIDF algorithm.

GuestWildLifeOne

```
class GuestWildLifeOne(Resource)
```

allocated logic for resource `/guest/wildlife/{wildlifeid}`

get

```
| def get(wildlifeid=None)
```

url `/guest/wildlife/{wildlifeid}` verb `GET`
`output:` {'wildlifeid': integer, 'type': string, 'species': string, 'photo': base64, 'notes': string, 'lon': float, 'lat': float, 'date': integer}
`Semantics:` downloads a single wildlife entry from the wildlife table according to its {wildlifeid}. The response â€˜Content-Dispositionâ€™ is set to â€˜attachmentâ€™ and the output value will be a json file containing the output json object.

GuestReport

```
class GuestReport(Resource)
```

allocated logic for resource `/guest/report`

post

```
| def post()
```

url `/guest/report` verb `POST`
input:
code: integer, report code
text: string, report body
wildlifeid: integer, wildlife entry being reported
`Semantics:` submits a new report about a wildlife entry to the reports table, the report could be about animal abuse, improper fire, fake entries ...etc.

AuthReport

```
class AuthReport(Resource)
```

allocated logic for resource `/auth/report/{reportid}`

put

```
| @login_required
| @has_role('curator')
| def put(reportid)
```

url `/auth/report/{reportid}` verb `PUT`
arguments:
cascade: boolean
`Semantics:` submits a report resolution request to the API, which marks the report as solved, this kind of report concerns animal abuse or similar issues, if found genuine the curator would contact the authorities, then resolve the report, it will be updated as solved in the reports table but not deleted. If cascade is true then all other reports about the same wildlife entry will be resolved as well. This resource is only available to curators; if the Session belongs to a user, the return status would be 403 Forbidden.

delete

```
| @login_required
| @has_role('curator')
| def delete(reportid)
```

url `/auth/report/{reportid}` verb `DELETE`
arguments:
cascade: boolean
`Semantics:` submits a report deletion request to the API, which deletes the report from the reports table. If cascade is true then the wildlife entry will be deleted from the wildlife table as well; in that case all other reports associated with the entry will be deleted as well. This resource is only available to curators; if the Session belongs to a user, the return status would be 403 Forbidden.

Search: `search.py`

WordLemmatizer

```
def WordLemmatizer(data)
```

provides Word Lemmatization for correct search functionality

SearchEngine

```
def SearchEngine(documents, query, theshold)
```

provides search capability via TFIDF

Server: `server.py`

sqlitext

```
def sqlitext(app)
```

loads sqlite extenstions into flask_sqlalchemy

Image: `image.py`

img2base64

```
def img2base64(path)
```

a function that encodes an image file into base64
parameters:
path: path to image file
returns:
base64 encoded image