WildWatch RESTful API Specification

Student: *Aly Shmahell* **Matricula:** *258912* **E-mail:** *aly.shmahell@student.univaq.it*

Preamble:

This project is a part of a large multi-course project:

- WildWatch: a mobile application implemented within the **Mobile Application course**.
- WildWatch Web: a web interface implemented within the **Web Engineering course**, consists of two web interfaces:
  - A public web app which can be accessed by anonymous guests.
  - A curator web app which can be accessed by the wildlife table curators.
- WildWatch RESTful API: an API implemented within the **Advanced Web Development course**, which would handle the databases; therefore will manage the data generated by the mobile users, and provide the data to the mobile users, the guests and the curators.

The Service:

URL: auth

VERB: POST

Input:

{“firstname”: str, “lastname”: str, “username”: str, “password”: str}

Semantics: If **(firstname & lastname)** are **not empty** then it creates a new user account based on the provided credentials. Otherwise it establishes a session based on the provided credentials and grants an {SID}.

URL: auth/{SID}?remove=bool

VERB: DELETE

Semantics: closes the session and invalidates the {SID}. If **remove** is **true** and the {SID} belongs to a user then it deletes the user account and anonymizes their personal info. However, if **remove** is **true** and the {SID} belongs to a curator, the return status would be **403 Forbidden**.

URL: auth/{SID}/profile

VERB: GET

Output:

{“fullname”: str, “website”: str, “bio”: str, “photo”: image/jpeg}

Semantics: returns the user profile information associated with the {SID}. This resource is only available to users; if the {SID} belongs to a curator, the return status would be **403 Forbidden**.

URL: auth/{SID}/profile/{category}

VERB: PUT

Input:

{ **{category}**: str }

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 {SID} belongs to a curator, the return status would be **403 Forbidden**.

URL: auth/{SID}/wildlife

VERB: POST

Input:

{“id”: int, “type”: str, “species”: str, “username”: str,
“photo”: image/jpeg, “notes”: str, “location”: int, “date”: int}

Semantics: submits a new wildlife entry to the database. This resource is only available to users; if the {SID} belongs to a curator, the return status would be **403 Forbidden**.

URL: auth/{SID}/wildlife?text=str&filters=dict&location=tuple&area=tuple

VERB: GET

Output:

if {SID} belongs to a user account:

[
{“id”: int, “type”: str, “species”: str, “photo”: image/jpeg, “notes”: str, “location”: int},
]

if {SID} belongs to a curator account:

[
{“id”: int, “type”: str, “species”: str, “photo”: image/jpeg, “notes”: str, “location”: int,
report: {“code”: int, “text”: str}
},
]

Semantics: fetches all the wildlife entries in the wildlife table based on the user’s or curator’s **location** and the size of their map **area**, the results are filtered by matching **filters** to the wildlife table columns using SQL where clause, and then further filtered down by matching the **text** part of the query to the notes column in the wildlife table using an off-the-shelf TFIDF algorithm. If {SID} belongs to a curator account, **filters** are also matched to the report table columns, then for every wildlife entry, if there is a report about the entry, a copy of the report will be added to the response. This resource only accepts ‘text/html’ mediatype.

URL: guest/wildlife?text=str&filters=dict&location=tuple&area=tuple&download=bool

VERB: GET

Output:

[
{“id”: int, “type”: str, “species”: str, “photo”: image/jpeg, “notes”: str, “location”: int},
]

Semantics: fetches all the wildlife entries in the wildlife table based on the guest’s chosen **location** and the size of their map **area**, the results are filtered by matching **filters** to the wildlife table columns using SQL where clause, and then further filtered down by matching the **text** part of the query to the notes column in the wildlife table using an off-the-shelf TFIDF algorithm. If mediatype is set to ‘application/zip’, then the response ‘Content-Disposition’ is set to ‘attachment’ and the return value will be a zip archive of the json objects in the output array.

URL: guest/wildlife/{id}

VERB: GET

Output:

{“id”: int, “type”: str, “species”: str, “photo”: image/jpeg, “notes”: str, “location”: int}

Semantics: downloads a single wildlife entry from the database according to its id. This resource only accepts ‘application/json’ mediatype, and the response ‘Content-Disposition’ is set to ‘attachment’.

URL: guest/report

VERB: POST

Input:

{“id”: int, “report”: {“code”: int, “text”: str} }

Semantics: submits a new report about a wildlife entry to the database, the report could be about animal abuse, improper fire, fake entries ...etc.

URL: auth/{SID}/report/{id}

VERB: PUT

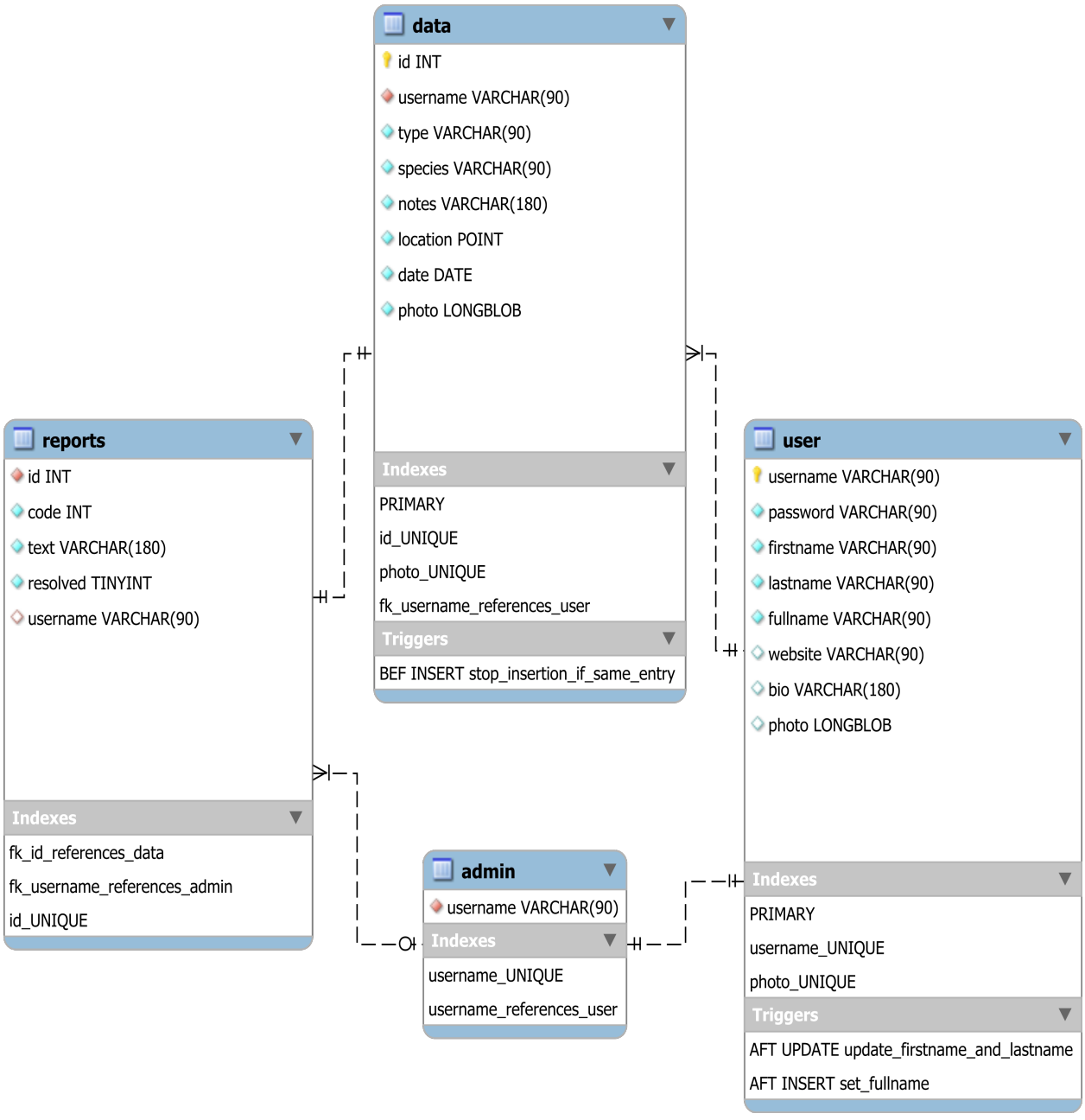
Semantics: submits a report resolution request to the API, which marks the report as solved, this kind of report concerns animal abuse or similar issues, if found genuine the curator would contact the authorities, then resolve the report, it will be updated as solved in the reports table but not deleted. This resource is only available to curators; if the {SID} belongs to a user, the return status would be **403 Forbidden**.

URL: auth/{SID}/report/{id}?deletentry=bool

VERB: DELETE

Semantics: submits a report deletion request to the database, if **deletentry** is **true** then the wildlife entry will be deleted as well. This resource is only available to curators; if the {SID} belongs to a user, the return status would be **403 Forbidden**.

Entity Relationship Model:



Technologies:

In this project I've chosen Flask to use in the design of the RESTful API server, for the following reasons:

- It supports Object Oriented Design Patterns.
- It supports RESTful development right out of the box, but more so with the flask-restful extension.
- It encourages compact and minimal code.
- It has as few layers of abstraction as possible.
- It's easy to write unit tests for it.
- It supports multitudes of database technologies (MySQL, Postgres, MongoDB …).
- It allows to interface database tables directly with python classes which inherit the database template.
- It has a good CORS support.
- It's thread-safe, allowing for running background services.
- It allows for the creation of loosely couples microservices, which makes it scalable.
- It treats the user as a consenting adult, allowing for easy modification of the core code, which makes it flexible.
- It's well integrated on heroku servers, which I will use to demo the API.
- Previous professional experience with flask and the desire to improve said experience.