Submitted by: John Jegard
Email: john.jegard@gmail.com
Phone: 403.771.2446

Created for: OpenHouse.AI

Backend Software Documentation

The code is written in Python / Flask framework utilizing SQLite database and SQLAlchemy for database access. As well Marshmallow is used to serialize the database objects. Also, the virtual environment is pipenv to keep all packages/dependencies in one place.

The code builds a RESTful API documented commencing page 2 below referred to as Log REST API. The JSON log data can be imported to the database weekly or as often as needed using the API POST method. Using the GET methods described in the Log REST API below, log data is retrievable by any combination of userid, time range, or log type, or if needed, as a list of all logs.

To build the REST API, the following is done:

1. Create database structure definition class
2. Create database schema class defining fields that can be displayed
3. Create endpoint route to import JSON log data to database
4. Create route to retrieve JSON log data to list all logs
5. Create route to retrieve JSON log data for a specific user
6. Create route to retrieve JSON log data for a specific user & log type
7. Create route to retrieve JSON log data for a specific user & time range
8. Create route to retrieve JSON log data for a specific user, type, & time range
9. Create route to retrieve JSON log data for a specific log type
10. Create route to retrieve JSON log data for a specific log type & time range
11. Create route to retrieve JSON log data for a specific time range
12. Create routes to delete JSON log data (several routes listed in API doc below)

Assumptions:

1. No PUT method required. Normally this would be used to change log info. In this case, log info is captured real-time by the frontend application and will not be required to be changed.
2. JSON data provided for the project reflects all the possible Parameters. The reason for this assumption is the Parameters affect the database structure design decision. If there are more varied parameters, a second linked table in the database may be required which, in turn, would require a change to the API methods coding. One draw back of my decision to use only one table is that the JSON output of the GET methods is not in the same format as the log file JSON.

Testing:

Testing was done during development using Postman as an API Client to send REST requests to the Log REST API.

Not implemented items:

oAuth authentication is not employed

Known Issues:

GET methods retrieve logs without time zone information

Log REST API

Schema:

API access is over HTTP and accessed http://localhost:5000
All data is sent and received as JSON.
Blank fields are included as NULL instead of being omitted.
All timestamps return ISO 8601 format YYYY-MM-DDTHH-MM-SS with time zone offset.
Log JSON data is stored in database fields:

id	userId	sessionId	time	log_type	locationX	locationY	viewedId	pageFrom	pageTo

Methods:
GET – used for retrieving log data from API
POST – create new log data entry in database
DELETE – used for deleting log data
Any colons (:) on a path denotes a variable. Replace these with values for the request. For instance replace :userid with ABC123XYZ

Post Logs:

POST /logs	Creates database records for logs in log file

List by userId:

GET /user/:userId	Lists all logs for a specific user

List by userId & log_type

GET /user/:userid/log_type/:type	Lists all logs for a specific user & log type

List by userId & time_range

GET /user/:userid/time_range/:start_time/:end_time	Lists all logs for a specific user & time range

List by userId & log_type & time_range

GET /user/:userid/log_type/:type/ time_range/:start_time/:end_time	Lists all logs for a specific user & log type & time range

Example Responses:

GET /user/ABC123XYZ


```
[
  {
    "id": 1,
    "locationX": 52,
    "locationY": 11,
    "log_type": "CLICK",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:37:28",
    "userId": "ABC123XYZ",
    "viewedId": null
  },
  {
    "id": 3,
```


GET /user/ABC123XYZ/log_type/CLICK


```
[
  {
    "id": 1,
    "locationX": 52,
    "locationY": 11,
    "log_type": "CLICK",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:37:28",
    "userId": "ABC123XYZ",
    "viewedId": null
  },
  {
    "id": 2,
```

```
    "locationX": null,
    "locationY": null,
    "log_type": "VIEW",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:37:30",
    "userId": "ABC123XYZ",
    "viewedId": "FDJKLHSLD"
  }
]
```

```
    "locationX": 52,
    "locationY": 11,
    "log_type": "CLICK",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:37:28",
    "userId": "ABC123XYZ",
    "viewedId": null
  }
]
```


GET /user/ABC123XYZ/time_range/2018-10-18T21:37:29-06:00/2018-10-18T21:37:35-06:00


```
[
  {
    "id": 3,
    "locationX": null,
    "locationY": null,
    "log_type": "VIEW",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:37:30",
    "userId": "ABC123XYZ",
    "viewedId": "FDJKLHSLD"
  },
  {
    "id": 4,
    "locationX": null,
    "locationY": null,
    "log_type": "NAVIGATE",
    "pageFrom": "communities",
    "pageTo": "inventory",
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:37:30",
    "userId": "ABC123XYZ",
    "viewedId": null
  }
]
```


GET /user/ABC123XYZ/log_type/NAVIGATE/time_range/2018-10-18T21:37:29-06:00/2018-10-18T21:37:35-06:00


```
[
  {
    "id": 4,
    "locationX": null,
    "locationY": null,
    "log_type": "NAVIGATE",
    "pageFrom": "communities",
    "pageTo": "inventory",
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:37:30",
    "userId": "ABC123XYZ",
    "viewedId": null
  }
]
```

List by log_type:

GET /log_type/:type	Lists all logs for a specific log type

List by log_type & time_range:

GET /log_type/:type/time_range/:start_time/:end_time	Lists all logs for a specific time range

Example Response:

GET /log_type/CLICK


```
[
  {
    "id": 1,
    "locationX": 52,
    "locationY": 11,
    "log_type": "CLICK",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:37:28",
    "userId": "ABC123XYZ",
    "viewedId": null
  },
  {
    "id": 2,
    "locationX": 52,
    "locationY": 11,
    "log_type": "CLICK",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:37:28",
    "userId": "ABC123XYZ",
    "viewedId": null
  }
]
```


GET /log_type/CLICK/time_range/2018-10-18T21:01:00-06:00/2018-10-18T21:04:00-06:00


```
[
  {
    "id": 8,
    "locationX": 52,
    "locationY": 11,
    "log_type": "CLICK",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABQ",
    "time": "2018-10-18T21:02:23",
    "userId": "ABC123XYZ",
    "viewedId": null
  },
  {
    "id": 12,
    "locationX": 21,
    "locationY": 43,
    "log_type": "CLICK",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABZ",
    "time": "2018-10-18T21:03:39",
    "userId": "ABC123XYZ",
    "viewedId": null
  }
]
```


List by time_range:

GET /time_range/:start_time/:end_time	Lists all logs for a specific time range

Example Response:

GET /time_range2018-10-18T21:01:00-06:00/2018-10-18T21:01:10-06:00


```
[
  {
    "id": 16,
    "locationX": 50,
    "locationY": 19,
    "log_type": "CLICK",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:01:03",
```

```
    "userId": "ABC123XYZ",
    "viewedId": null
  },
  {
    "id": 29,
    "locationX": 50,
    "locationY": 12,
    "log_type": "CLICK",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:01:06",
    "userId": "ABC123XYZ",
    "viewedId": null
  },
  {
    "id": 33,
    "locationX": null,
    "locationY": null,
    "log_type": "VIEW",
    "pageFrom": null,
    "pageTo": null,
    "sessionId": "XYZ456ABC",
    "time": "2018-10-18T21:01:08",
    "userId": "ABC123XYZ",
    "viewedId": "FDJKLHSLD"
  }
]
```


List of Logs:

GET /logs	Lists all logs

Delete Logs:

DELETE /logs/id	Delete specific log by id
DELETE /user/:userid	Deletes all logs for a specific user
DELETE /time_range/:start_time/:end_time	Deletes all logs for a specific time range
DELETE /log_type/:log_type	Delete specific log by log_type

Client Errors:

All error objects have properties so the client can tell what the problem is. These are the possible validation error codes:

Error Name	**Description**
bad_request_400	Data incorrectly formatted
not_found_404	Requested log data not found
method_not_allowed_405	Formatting of the method routing is invalid or missing
internal_server_error_500	Server error

What needs to be done to make this cloud scalable?

To make this cloud scalable, the code needs to be:

1. Portable across execution environments
    a. Dependencies of the deployable services on each other, so common code needs to in packages/libraries/repositories.
    b. Cloud Backing services: The SQLite file is saved locally. Different cloud environments may have different requirements around this file, or the database used. As well cloud providers offer datastore as a service which perhaps could be used.
    c. Configurations: better to store configuration data in a JSON file or some external file rather than in the code itself
2. Scalable without significant change or effort
    a. Testing needs to be done for high number of logs and streaming. This code is written where REST API is used to POST the logs to the database. If logs are continuously streamed in real time the code would need to be modified to accommodate this. Your document states up to 10K logs could be streamed at once. Code has not been tested on volume logs.
    b. Need to determine whether the database can withstand high load. Does it need to be changed to a different relational database? Are sharing, partitioning, codebase optimizations needed?
    c. AWS offers scalability built in to handle different levels of traffic and auto scaling back end services.
3. Multi-node deployments in the Cloud
    a. Best not to rely on data being written to file systems or memory as you can't be sure it will be available. There would be issues if each instance of the app were accessing the same database
4. Ensure enough resources are available
    a. AWS, Amazon has CloudWatch to monitor resources
    b. GCP, Google has Stackdriver to monitor resources
    c. Cloud Agnostic has Crontab to monitor resources
    d. Heroku has Runtime to monitor resources