

Elanco Placement Program: Technical Task - Backend MVP Documentation

Index

1. Overview
2. End Points
 - 2.1. /sightings
 - 2.2. /filter
 - 2.3. /stats/locations
 - 2.4. /trends
 - 2.5. /stats/location-species
 - 2.6. /stats/all-locations-species
3. Data Handling
 - 3.1. Large datasets
 - 3.2. Caching
 - 3.3. Missing values
 - 3.4. Duplicate handling
4. AI/ML Insights (Extension Task)
 - 4.1. Machine Learning model to predict ticks for the upcoming weeks
 - 4.2. Use of MLFlow
5. Key Architectural Decisions
 - 5.1. Modular Choices
 - 5.2. Caching and logging
 - 5.3. Consistent error handling
 - 5.4. Handling missing values/duplicates
 - 5.5. Handling large datasets
6. Future Enhancements (if more time was available)
7. References
8. Related experience
9. System Architecture diagram

1. Overview

The backend for the MVP is built using **Flask**, leveraging my prior experience developing APIs and data-driven applications in this framework. The application exposes a set of RESTful endpoints designed to retrieve, filter, and aggregate tick-sighting data.

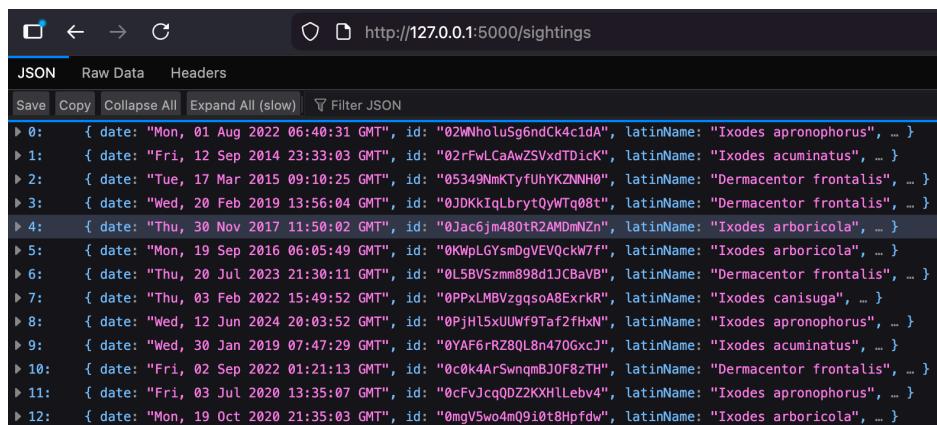
Each endpoint serves a specific analytical purpose and returns results in JSON format to support easy frontend integration.

2. End Points

/sightings

Method: GET

Purpose: Returns the complete tick-sightings dataset in JSON format. This endpoint is useful to load all sightings on the UI at once.



A screenshot of a browser's developer tools Network tab, specifically the 'JSON' tab, showing the response for the '/sightings' endpoint. The URL in the address bar is 'http://127.0.0.1:5000/sightings'. The response is a large array of 12 JSON objects, each representing a tick sighting with fields like date, id, and latinName. The JSON is displayed in a collapsible tree view where individual items can be expanded to show their internal structure.

```
[{"date": "Mon, 01 Aug 2022 06:40:31 GMT", "id": "02WnholuSg6ndCk4c1dA", "latinName": "Ixodes apronophorus", ...}, {"date": "Fri, 12 Sep 2014 23:33:03 GMT", "id": "02rfwlCeaWzSVx0TDick", "latinName": "Ixodes acuminatus", ...}, {"date": "Tue, 17 Mar 2015 09:10:25 GMT", "id": "05349NmKTyfUhYKZNNH0", "latinName": "Dermacentor frontalis", ...}, {"date": "Wed, 20 Feb 2019 13:56:04 GMT", "id": "0JDKKIqlbrytQyTq08t", "latinName": "Dermacentor frontalis", ...}, {"date": "Thu, 30 Nov 2017 11:50:02 GMT", "id": "0jac6jm480tR2AMdnZn", "latinName": "Ixodes arboricola", ...}, {"date": "Mon, 19 Sep 2016 06:05:49 GMT", "id": "0KwpLGysmDgVEVQckW7f", "latinName": "Ixodes arboricola", ...}, {"date": "Thu, 20 Jul 2023 21:30:11 GMT", "id": "0L5BVSzmm898d1JCbaVB", "latinName": "Dermacentor frontalis", ...}, {"date": "Thu, 03 Feb 2022 15:49:52 GMT", "id": "0PxLMBVzggsoABExrKR", "latinName": "Ixodes canisuga", ...}, {"date": "Wed, 12 Jun 2024 20:03:52 GMT", "id": "0PjHL5xUWF9taf2fHN", "latinName": "Ixodes apronophorus", ...}, {"date": "Wed, 30 Jan 2019 07:47:29 GMT", "id": "0YAF6rRZ8QL8n470GxcJ", "latinName": "Ixodes acuminatus", ...}, {"date": "Fri, 02 Sep 2022 01:21:13 GMT", "id": "0c0k4Ar5wngmBJ0F8zTH", "latinName": "Dermacentor frontalis", ...}, {"date": "Fri, 03 Jul 2020 13:35:07 GMT", "id": "0cFvJcqQDZ2KXH1Lebv4", "latinName": "Ixodes apronophorus", ...}, {"date": "Mon, 19 Oct 2020 21:35:03 GMT", "id": "0mgV5wo4mQ9i0t8Mpfdw", "latinName": "Ixodes arboricola", ...}]
```

/filter

Method: GET

Parameters:

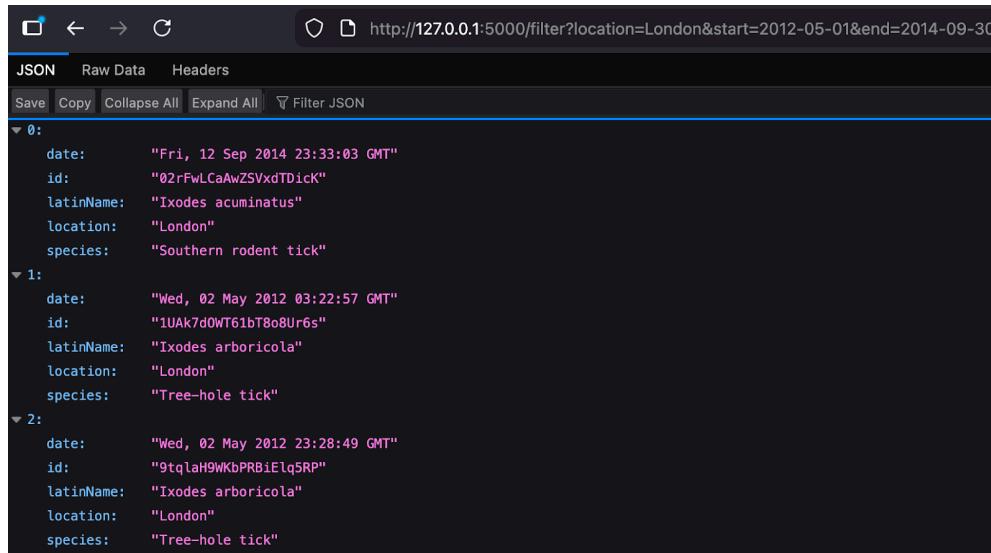
- start - optional start date
- end - optional end date
- location - optional location

Purpose: Returns a filtered subset of the dataset based on user-provided date range and/or location. Invalid results are displayed in JSON with error response with appropriate status code.

Error Handling:

Robust validation is implemented to handle

- a. Invalid date formats
- b. Missing or empty location strings
- c. Location values that do not exist in the dataset



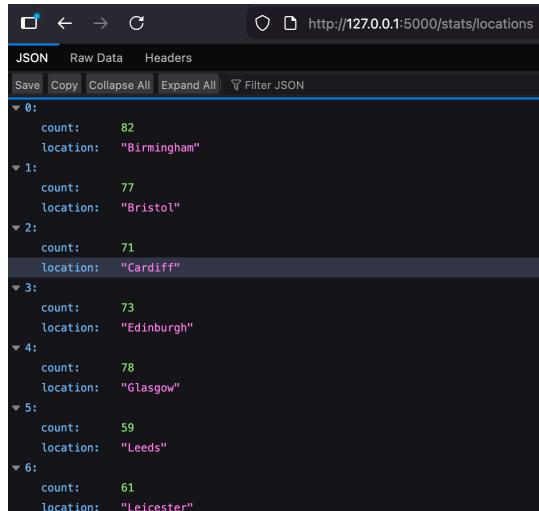
A screenshot of a web browser displaying a JSON response from a local server. The URL in the address bar is `http://127.0.0.1:5000/filter?location=London&start=2012-05-01&end=2014-09-30`. The browser interface includes tabs for 'JSON', 'Raw Data', and 'Headers'. Below these are buttons for 'Save', 'Copy', 'Collapse All', 'Expand All', and 'Filter JSON'. The JSON data is presented as an array of three objects (0, 1, 2), each representing a tick sighting. Each object contains the following fields: date, id, latinName, location, and species.

```
[{"date": "Fri, 12 Sep 2014 23:33:03 GMT", "id": "02rFwLCaAvZSVxdTBick", "latinName": "Ixodes acuminatus", "location": "London", "species": "Southern rodent tick"}, {"date": "Wed, 02 May 2012 03:22:57 GMT", "id": "1UAK7d0WT61bT8o8Ur6s", "latinName": "Ixodes arboricola", "location": "London", "species": "Tree-hole tick"}, {"date": "Wed, 02 May 2012 23:28:49 GMT", "id": "9tqlaH9WKbPRBiElq5RP", "latinName": "Ixodes arboricola", "location": "London", "species": "Tree-hole tick"}]
```

/stats/locations

Method: GET

Purpose: Returns aggregated sighting counts per location. It can be useful for identifying hotspots and regional trends.



A screenshot of a web browser displaying a JSON response from a local server. The URL in the address bar is `http://127.0.0.1:5000/stats/locations`. The browser interface includes tabs for 'JSON', 'Raw Data', and 'Headers'. Below these are buttons for 'Save', 'Copy', 'Collapse All', 'Expand All', and 'Filter JSON'. The JSON data is an array of seven objects (0 to 6), each representing a location with its corresponding sighting count.

```
[{"count": 82, "location": "Birmingham"}, {"count": 77, "location": "Bristol"}, {"count": 71, "location": "Cardiff"}, {"count": 73, "location": "Edinburgh"}, {"count": 78, "location": "Glasgow"}, {"count": 59, "location": "Leeds"}, {"count": 61, "location": "Leicester"}]
```

/trends

Method: GET

Purpose: Returns a JSON object containing weekly trends, monthly trends and yearly trends.

The screenshot shows a browser's developer tools JSON viewer. The URL is `http://127.0.0.1:5000/trends`. The JSON response contains three main objects: `monthly`, `weekly`, and `yearly`, each represented by a long array of objects.

```
▶ monthly: (12)[{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, ...]
▶ weekly: (52)[{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, ..., ...]
▶ yearly: (13)[{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, ..., ...]
```

/stats/location-species

Method: GET

Parameter: location - required

Purpose: Returns species-level counts for a specific location.

Error Handling: If an invalid location is entered, the response includes an error message and a list of valid locations

The screenshot shows a browser's developer tools JSON viewer. The URL is `http://127.0.0.1:5000/stats/location-species?location=London`. The JSON response lists species ticks for London:

```
Fox/badger tick: 19
Marsh tick: 15
Passerine tick: 15
Southern rodent tick: 12
Tree-hole tick: 15
```

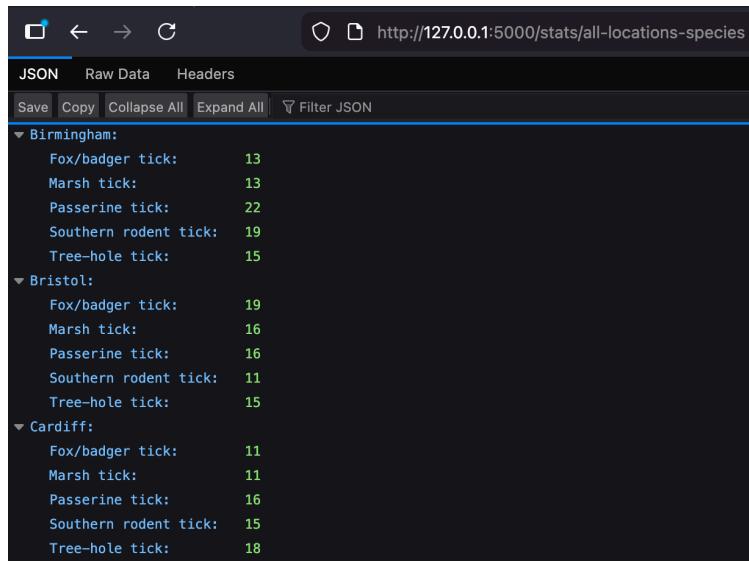
The screenshot shows a browser's developer tools JSON viewer. The URL is `http://127.0.0.1:5000/stats/location-species?location=Belfast`. The JSON response includes an error message and a list of valid cities:

```
error: "'belfast' is not a valid city."
valid_cities:
  0: "Manchester"
  1: "London"
  2: "Glasgow"
```

/stats/all-locations-species

Method: GET

Purpose: Returns a nested JSON object showing the count of each species across all locations.



The screenshot shows a browser's developer tools JSON viewer with the URL `http://127.0.0.1:5000/stats/all-locations-species`. The JSON data is displayed under the "JSON" tab, showing counts for various tick species across three locations: Birmingham, Bristol, and Cardiff. The data is structured as follows:

```
{
  "Birmingham": {
    "Fox/badger tick": 13,
    "Marsh tick": 13,
    "Passerine tick": 22,
    "Southern rodent tick": 19,
    "Tree-hole tick": 15
  },
  "Bristol": {
    "Fox/badger tick": 19,
    "Marsh tick": 16,
    "Passerine tick": 16,
    "Southern rodent tick": 11,
    "Tree-hole tick": 15
  },
  "Cardiff": {
    "Fox/badger tick": 11,
    "Marsh tick": 11,
    "Passerine tick": 16,
    "Southern rodent tick": 15,
    "Tree-hole tick": 18
  }
}
```

3. Data Handling

3.1. Large datasets

For handling large datasets, a method is implemented to process the data in batches instead of processing the entire data together. This process is known as *chunking* [1] helps reduce memory usage and improves performance when working with millions of rows. For this specific dataset, chunking is intentionally disabled because: we just have 1000 rows, so chunking will slow down processing because of multiple for loop calls.

3.2. Caching

RLU Cache is implemented to serve frequent results more quickly.^[2]

3.3. Missing values

The dataset may contain several missing fields such as *species*, *Latin names*, *locations*, and some *dates*. Each type of missing value is handled using an appropriate strategy.

a. Missing Species / Latin Names

- i. A dictionary is created for mapping species with their corresponding Latin names.
- ii. Any missing value is filled based on the dictionary to ensure internal consistency.

b. Missing Locations

- i. Location values are imputed using a probability-based approach by calculating the frequency distribution of existing locations.
- ii. Each missing location is filled by random sampling from this distribution.

c. Missing date

Since the data is very sparse, it's challenging to estimate date values for missing values, removal is the safest option. Following common data quality guidance, if fewer than 5% of rows have missing values for a critical field, it is generally acceptable to drop them.

3.4. Duplicate handling

Duplicates are removed to prevent inflated counts in aggregation endpoints.

4. AI/ML Insights (Extension Task)

4.1. Machine Learning model to predict ticks for the upcoming weeks

A machine learning model is built to predict ticks for any location for upcoming n-weeks. Since the data is very sparse, ARIMA (Autoregressive Integrated Moving Average) is preferred over Deep Learning, which is the usual choice for time-series data.

Deep Learning methods are data hungry algorithms, requiring large volumes of data. ARIMA, on the other hand, works extremely well on small time series, capturing trends with minimal data.

For this task, ARIMA provides a practical and accurate solution, with very low MAE (Mean Absolute Error) on training data.

4.2. Use of MLFlow

MLflow is a state-of-the-art platform for tracking machine learning experiments. It allows logging model parameters, metrics, artifacts, and comparing multiple runs through an intuitive UI. This makes it easy to evaluate different model configurations and identify which version of the model performs best.

In this project, MLflow is used to compare how variations in ARIMA hyperparameters (p, d, q) influence forecasting accuracy. By logging metrics such as MAE and MSE across several runs, MLflow provides a clear, visual way to understand which configuration offers the strongest predictive performance.

In principle, MLflow plays a key role in enabling MLOps practices such as experiment tracking, reproducibility, and comparison across model versions.

Below are screenshots of the MLflow UI, which illustrate how metrics, parameters, and model runs can be tracked and compared. For demonstration purposes, the city of Manchester is used as an example.

Currently, the ML model is kept outside the main API to keep the backend service lightweight and focused, and to support independent experimentation in line with good MLOps practices. A future enhancement would be to integrate the ML model into the backend.

The screenshot shows the MLflow UI interface. At the top, there's a navigation bar with the MLflow logo, version 3.6.0, GitHub, and Docs links. Below the navigation is a breadcrumb trail: Tick Forecasting > Runs > nebulous-stork-667. The main content area is divided into sections: Overview, Model metrics, System metrics, Traces, and Artifacts. The Overview section is active, showing a table of metrics with columns Metric and Value. The table includes rows for forecast_week_7, forecast_week_6, forecast_week_1, mae, mse, forecast_week_4, forecast_week_3, forecast_week_2, and forecast_week_5. To the right of the metrics table is a sidebar with experiment details: Experiment ID (00000000000000000000), Status (Finished), Run ID (1d4f1d1652124947b51591b8853d2004), Duration (2.8s), Source (ipykernel_launcher.py), Logged models (—), Registered prompts (—), Datasets (None), Tags (Add tags), and Registered models (None). At the bottom left is a Parameters section with a table for city (Value: Manchester).

Metric	Value
forecast_week_7	0.08903980154148575
forecast_week_6	0.1048578665954779
forecast_week_1	0.2904224998590817
mae	0.06617440155337866
mse	0.010092364630060451
forecast_week_4	0.1531281465616496
forecast_week_3	0.27169043095479667
forecast_week_2	0.23421250827245077
forecast_week_5	0.15959458682020666

Parameter	Value
city	Manchester

arima_order	[1, 0, 2]	(3, 0, 1)	(4, 0, 2)
city	Manchester	Manchester	Manchester
num_weekly_points	675	675	675
window_size	4	4	4
<hr/>			
▼ Metrics			
<input type="checkbox"/> Show diff only			
forecast_week_4	0.177	0.173	0.153
forecast_week_5	0.146	0.143	0.16
forecast_week_6	0.13	0.121	0.105
forecast_week_7	0.121	0.108	0.089
mae	0.071	0.066	0.066
mse	0.012	0.012	0.01

5. Key Architectural Decisions

5.1. Modular Choices

All data processing logic of filtering, trend calculations, species aggregation, and data cleaning is implemented inside the utils/ module.

Implementation of defining routes, passing request parameters to the utility functions, and returning JSON responses is done in main.py

This separation of concerns ensures a cleaner architecture where so that the application is easier to maintain, extend, or debug.

5.2. Caching and logging

Even working with a small website, logging is a good practice [2]. Basic logging in this application captures when certain endpoints are hit, invalid date or location values and successful operations.

Since total sightings may be requested frequently, caching [2] helps avoid recalculating the same results repeatedly. This offers significant performance benefits when working with large datasets. Although our current dataset is small which means the speed boost isn't huge, this practice ensures the system remains scalable and future-proof.

5.3. Consistent error handling

5.4. Handling missing values/duplicates

5.5. Handling large datasets

An architecture diagram is added in the last page for a better visualisation.

6. Future Enhancements (if more time was available)

1. Writing unit tests using pytest for end-to-end testing.
2. Integrating the ML model developed to predict the ticks in upcoming weeks in the backend. Currently the model runs independently of the service.
3. Persistence storage using databases such as PostgreSQL to support larger datasets.
4. Apply more MLOps principles by
 - a. Setting up a CI/CD pipeline using Github actions to automate workflows.
 - b. Package the entire application using Docker, and integrate it with MLFlow by running MLFlow tracking server in Docker.
 - c. Deploying the models in the cloud.

7. References

- [1] https://pandas.pydata.org/docs/user_guide/scale.html
- [2] System Design Interview - An Insider's Guide by Alex Xu

8. Related experience

I have previously developed backend and data-processing applications using Flask, Pandas, and REST APIs, which strengthened my skills in API design, data validation, and clean backend architecture. This experience directly contributed to how I structured the Elanco MVP backend.

Relevant Github projects

- Flask Movie App - <https://github.com/kamal0803/flask-top-movies-app>
- Flask Cafe App - <https://github.com/kamal0803/flask-cafe-application>
- Node.js Travel Tracker - <https://github.com/kamal0803/Travel-Tracker>

9. System Architecture Diagram (below)

System Architecture

