ISS Orbit Tracking API

Ashton Cole
AVC687

March 10, 2023

Contents

Chapter 1

Introduction

This program is an *Application Programming Interface* (API) for position and velocity data for the International Space Station (ISS) made live and public by NASA [1]. It acts as an intermediary, enabling the user to make a variety of requests to retrieve and manipulate all or part of the data set. It exists as a Python-based Flask[2] web application. Clients can interact with the API by making HTTP requests, although currently, the application is only designed to be locally hosted.

This was constructed as a mid-term assignment for the course *COE 332: Software Engineering and Design*. As such, the project is not intended to be published to the Internet. Its design exists more to demonstrate proficiency in software engineering than to offer a novel, distributable product. Nevertheless, the concept behind it offers significant value to the hypothetical user. Instead of having to download the original XML-format file from NASA and sift through the entire data set, they can issue commands to navigate through it and only retrieve information that they need. If someone wants to know where the ISS is right now, all they have to do is ask, and the API will even tell them the closest ground location.

This section continues in introducing the project structure, underlying data structure and discussing and the value of the API as a demonstration of software engineering concepts. Chapter 2 provides instructions on how to set up and execute the program. Chapter 3 outlines all of the endpoints of the API. Appendix A explores important ethical considerations. Appendix B provides example outputs for each output. Finally, Appendix C discusses important considerations for the `/now` endpoint.

1.1 Project Structure

This project can be accessed through two means. The first is as an image on Docker Hub[3]. This is ideal for running the application as-is within a Docker container. The sec-

[1] `https://spotthestation.nasa.gov/trajectory_data.cfm`

[2] *Flask* is a Python module which facilitates setting up a web-based service with the execution of endpoints, i.e. requests to access or modify data, defined in Python.

[3] `https://hub.docker.com/r/ashtonvcole/iss_tracker`

Figure 1.1: Directory of API Source Code

iss_tracker

Dockerfile	This builds the Docker image to build and run containers.
README.md	This provides briefer programming-oriented descriptions and instructions for the project.
dockercompose.yaml	This is used with Docker Compose to streamline building an image and executing a container in one step.
iss_tracker.py	This is the contents of the actual program. It starts the Flask application, pulls data from the Internet, and processes requests for each of the endpoints.

ond is a repository of the source code on GitHub[4]. This allows individual components of the application to be inspected and modified. From there, the program can be run with or without a container. See Chapter 2 for more details on executing the application from either source. The repository organizes all of the source files as seen in Figure 1.1.

1.2 Underlying Data Structure

The API consists of data points representing the predicted position and velocity of the ISS. These are set at four-minute intervals spanning about fifteen days, periodically updated such that most points tend to be in the future.

Studying the structure of the source data reveals the value of creating an intermediary API. Data points are nested within several levels of a data structure which contains additional ancillary information aiding in interpretation. The rough structure is outlined in Figure 1.2. Note that the full data set is given in XML format. The structure is very complicated, probably to give it flexibility, but this also means that the user must specify five levels of structure before accessing individual state vectors. Since this data set is consistent, only including one Orbital Ephemeris Message (OEM) containing one segment for a single orbiting object, our API is valuable as a means to winnow away unnecessary complexity and obtain the most direct access to important information.

1.3 Software Engineering Concepts

This project is also important as a demonstration of several software engineering concepts. Firstly, as an API, it acts like a black box. The user provides inputs, in this case specifications of the data they want, and can expect reliable outputs. There is no need to understand the underlying functionality that makes this possible. So long as the user knows the format of requests and their corresponding results, they can leverage the service effectively.

[4]`https://github.com/ashtonvcole/iss_tracker`

Figure 1.2: Approximate Organization of the Source Data

- `ndm`: the entire contents of the data file
    - `oem`: the Orbital Ephemeris Message (OEM), specification of the position and velocity of an object at several epochs
        - `header`: the creation information of the OEM
        - `body`: the contents of the OEM
            - `segment`: a set of epochs
                - `metadata`: information about the segment, including the object name, center of orbit, and reference frames
                - `data`
                    - `comment`: commentary about the segment
                    - `stateVector`: a set of position, velocity, and time of an epoch

This focus on clear communication protocols, without having to understand underlying processes, facilitates the development of the API as a microservice. Microservices are smaller, loosely coupled pieces of software that function relatively independently. This means that certain programs can be modified or expanded under-the-hood without impacting other services. So long as the communication protocols are the same, other programs can rely on the same functionality. In a similar vein, it means that cohesive pieces of software can be reused by other programs and end users for different end goals. For example, an individual might peruse this API to know when the ISS passes over them, while another service could use it to create a visualization, and another might synthesize it with other API's to predict collisions. All of these clients can re-use the same API, focusing instead on expanding functionality to their unique needs.

Microservices, by design, are meant to be containerizable. This means that they are cohesive. External dependencies, which may be version-specific, are included with the service, while interactions outside of the software "container" are strictly limited. This cordoning-off enables the program to function reliably in a variety of environments. It also makes it more portable, since an executor can simply download the container with everything necessary for execution. There may be some associated memory cost. If multiple containerized programs rely on the same dependency, they each get a separate copy. However, the benefit of portability with consistent results seems to outweigh this.

Chapter 2

Setup and Execution

Ideally, this project is run as a Docker container, outlined in Section 2.1. As mentioned in Section 1.3, containerization simplifies replication and execution by including all dependencies within the product and limiting interactions with the outside environment to provide system-agnostic, consistent results. However, the user may also run the program fron the source files in a non-containerized environment, which is detailed in Section 2.2. It is important to emphasize again that the program is intended to be locally hosted, and the instructions below seek that end. However, it is not likely difficult to adjust the implementation to host the project on the web.

All methods of execution require a stable Internet connection. This is because the API accesses its source data online. They also require usage of a command line interpreter. A Linux-type shell is strongly recommended for commands to work as expected.

2.1 Running within a Docker Container

With Docker[1] containerization, only Docker software is necessary. All other dependencies are included within the container. The image can either be pulled or built, and then run with Docker. Alternatively, docker-compose streamlines this process into one step.

2.1.1 Using only Docker

The image can be pulled from the Docker Hub[2] with a simple command.

```
docker pull ashtonvcole/iss_tracker:hw05
```

[1]*Docker* is a program that facilitates distributing and running containerized software applications. This means that they include all dependent softawre, and are executed in an environment separate from the host operating system. This maximizes consistent operation for diverse users.

[2]*Docker Hub* is an online repository which holds images of software containers. It is to Docker what GitHub is to Git. An image is a set of instructions for setting up a container, i.e. an isolated environment, for executing software.

Alternatively, an image may be constructed from a copy of the source code using the following command.

```
docker build -t <your_image_name> .
```

Once the image is obtained, the program may be with Docker. Running with Docker requires "binding" the port of the container to a port of the environment in which the program is running. Since the API uses port 5000, the external port 5000 is linked to the container's port 5000 with the `-p` tag. If a live view of the API's console is desired, to view requests and debug errors, the `-it` tag may be added. The tag `--rm` removed the container after execution is stopped. Combined, the command may be executed as follows.

```
docker run -it --rm -p 5000:5000 ashtonvcole/iss_tracker:1.0
```

2.1.2 Using Docker with Docker Compose

The docker-compose command streamlines the building and startup process by permitting configurations to be defined in a YAML file. The service is a plugin to Docker that must be added separately.[3] Simply download the `docker-compose.yaml` from GitHub with the rest of the source code, navigate to their directory, and execute the following command.

```
docker-compose up
```

The container may need to be removed after execution.

2.2 Running Directly with Python

This project can also be run directly with Python 3.8.10. This permits the executor to inspect and modify the source code to suit their own needs. The only file required is `iss_tracker.py` from the GitHub repository. Several modules beyond the Python standard library are required to run the application.

- Flask 2.2.2
- requests 2.22.0
- xmltodict 0.13.0
- geopy 2.3.0

Each of these may be installed with the Pip package manager as follows.

```
pip install --user <module_name>==<version>
```

[3]Instructions for installation may be found here: `https://docs.docker.com/compose/install/`

There are a few ways to execute the Python script. It may be interpreted with the `python3` command. If the file permission is changed to executable, and the appropriate shebang is added at the beginning of the file, it may be executed directly. Finally, it may be run with Flask as follows. The `--debug` tag logs errors in the Python script.

```
flask --app iss_tracker --debug run
```

Chapter 3

Usage

Once the Flask application is started, ordinary HTTP requests may be made to the local host through port 5000. This means that the API can be accessed through a local web browser via `http://127.0.0.1:5000`. The term `localhost` is an equivalent term. Requests can also be made through the command line with the `curl` command, as seen below. By default, this makes a `GET` request. Alternative modes can be specified with the `-X` flag, followed by the HTTP request type. For a more detailed output, the `-v` tag will show the response code and other metadata.

```
curl "http://localhost:5000/"
```

3.1 Endpoints

The API has several functions that the user may call to retrieve and modify data, listed in Table 3.1. They appear similar to nested file paths. Like directories referring to a particular web page, they go after the main address. However, despite their appearance, they do not represent a nested structure. Within `iss_tracker.py`, all endpoints are defined at the same level. The styling simply serves to make the requests appear logically organized.

The sequence as a whole is a unique identifier for an endpoint. Some of the segments delineated by forward slashes are parameters. In our notation, these are marked with angled brackets. Optional arguments may be passed to some endpoints too. These follow a question mark at the end of the identifier, and are separated by ampersands.

It is important to note that, since the API uses HTTP for communication, certain characters are not allowed to be typed, and will result in an error. These can be represented with appropriate escape sequences. In this API, there are only two pertinent characters that need to be escaped.

While return types have been specified for each endpoint, if an error occurs, all will return an error message in `text/plain` format. This could happen, for example, if the user provides an ill-constructed input or if the internal data set is empty. They will also provide a different HTTP response code to notify the client.

Table 3.1: HTTP Escape Codes

Character	Code
:	%3A
.	%2E

Table 3.2: List of Endpoints

Endpoint	Request	200 Response Type	Description
/	GET	application/json	This returns the entire data set.
/epochs	GET	application/json	This returns a list of all epochs. They are in the form YYYY-DDDTHH:MM:SS.000Z. Two optional integer parameters are available: offset determines how many spaces the result is offset from the zeroth epoch, and limit determines the number of epochs returned.
/epochs/<>	GET	application/json	This returns the state vector associated with an epoch. The input is in the form YYYY-DDDTHH:MM:SS.000Z.
/epochs/<>/speed	GET	application/json	This returns the speed associated with a given epoch. The input is in the form YYYY-DDDTHH:MM:SS.000Z.
/epochs/<>/location	GET	application/json	This returns a variety of location parameters including the latitude, longitude, and altitude for a given epoch. It also shows the nearest ground location below the ISS at that time, or null if the point is over the ocean or otherwise remote. The input is in the form YYYY-DDDTHH:MM:SS.000Z.

Continued on next page

Table 3.2 – *Continued from previous page*

Endpoint	Request	200 Response Type	Description
`/now`	`GET`	`application/json`	Warning: This only functions circa the beginning of March. See Appendix C. This returns a variety of location parameters including the latitude, longitude, and altitude for the epoch closest to the present time. It also shows the nearest ground location below the ISS at that time, or `null` if the point is over the ocean or otherwise remote. The difference between the present time and epoch is recorded in seconds.
`/delete-data`	`DELETE`	`text/plain`	This clears all data from the application.
`/post-data`	`POST`	`text/plain`	This overwrites existing data in the application with the most recent posting of the source data.
`/help`	`GET`	`text/plain`	This provides a catalog of endpoints with brief descriptions.
`/comment`	`GET`	`application/json`	This provides the comments added to the segment. This includes information about the conventions used to represent the orbit and upcoming trajectory events.
`/header`	`GET`	`application/json`	This provides the header added to the OEM. This includes information about the creation of the data file.
`/metadata`	`GET`	`application/json`	This provides the metadata added to the segment. This includes identifiers for the orbital center and orbiting object, time and spatial references, and start and stop times.

Appendix A

Ethical and Professional Responsibilities in Engineering Situations

There are a few important responsibilities that were considered in developing this application. This application is provided freely to the public. Anyone can access it from GitHub or Docker Hub, and use it as they please. Thus, under an ethical responsibility not to commit *misfeasance* which could harm others, it is essential to ensure that the application is reliable. This contributes to a strong chain of trust that makes APIs practicable, benefitting society.

A.1 It functions as described

Firstly, it is important for the application to function as described. This means that it should provide the expected output in all reasonable situations. To ensure that the product functions, testing and validation is incorporated into the development process. For example, the speed endpoint should always return the correct Euclidean 2-norm from the state vector. However, it is reasonable to expect that the data set in memory will sometimes be empty, and that sometimes the user will ask for an epoch which is not recorded or ill-defined. In such a case it should return an error message. When the endpoint was written, its output was compared to hand calculations, and confirmed to provide an error when it should. Just as important as merely functioning as intended, the endpoint was ensured to function as described, or, equivalently, to be described as it functions. The user is well-informed of the application's behavior in the documentation. If everything functions correctly, they receive a 200 success code, meaning that they can expect a JSON output. If there is an error, they receive a 4XX error code, meaning that they can expect a plain text explanation of the error. This is not a perfectly fail-safe system. For example, if there is unexpected ill behavior from one of the dependencies, in spite of containerization and version control, or if the source data is ill-formed, then the application may fail. Just as the user trusts the application as a black box, the application itself trusts other black boxes. With validation steps and thorough documentation, the application contributes a strong link in this chain.

A.2 It sources responsibly

As mentioned before, while the developer has a responsibility to verify and document their own application, it is not possible to perfectly verify all dependent sources. In remedy, the developer should make good-faith efforts to draw from reliable sources and document them as necessary. For example, all of the Python dependencies are either from the Python standard library or are public, reasonably popular, established modules on the Pip online repository. Further, specific versions are packaged with the application to establish consistency. Similarly, the information is sourced from the NASA website. As an established government agency responsible for the ISS, with a strong interest in tracking it to high precision and accuracy, NASA may be accepted as a reliable source for ISS tracking data. In turn, the application documents its own sources to establish its legitimacy to the user. It mentions all modules not part of the Python standard library. More importantly, as a provider of ISS tracking data, when the developer is not an established source thereof, it cites this source in its documentation. In short, the developer should find reliable sources so that the application functions correctly, and document them to establish legitimacy.

Appendix B

Example Outputs for Each Endpoint

Table B.1: List of Example Outputs for Each Endpoints

Endpoint	200 Response Sample Output

`/`

```
{
  "ndm": {
    "oem": {
      "@id": "CCSDS_OEM_VERS",
      "@version": "2.0",
      "body": {
        "segment": {
          "data": {
            "stateVector": [
              {
                "EPOCH": "2023-048T12:00:00.000Z",
                "X": {
                  "#text": "-5097.51711371908",
                  "@units": "km"
                },
                "X_DOT": {
                  "#text": "-4.5815461024513304",
                  "@units": "km/s"
                },
                "Y": {
                  "#text": "1610.3574036042901",
                  "@units": "km"
                },
                "Y_DOT": {
                  "#text": "-4.8951801207083303",
                  "@units": "km/s"
                },
                "Z": {
                  "#text": "-4194.4848049601396",
                  "@units": "km"
                },
                "Z_DOT": {
                  "#text": "3.70067961081915",
                  "@units": "km/s"
                }
              },
              ...
            ]
          },
          "metadata": {
            "CENTER_NAME": "EARTH",
            "OBJECT_ID": "1998-067-A",
            "OBJECT_NAME": "ISS",
            "REF_FRAME": "EME2000",
            "START_TIME": "2023-048T12:00:00.000Z",
            "STOP_TIME": "2023-063T12:00:00.000Z",
            "TIME_SYSTEM": "UTC"
          }
        }
      },
      "header": {
        "CREATION_DATE": "2023-049T01:38:49.191Z",
        "ORIGINATOR": "JSC"
      }
    }
  }
}
```

`/epochs`

```
[
  "2023-048T12:00:00.000Z",
  "2023-048T12:04:00.000Z",
  "2023-048T12:08:00.000Z",
  "2023-048T12:12:00.000Z",
  "2023-048T12:16:00.000Z"
]
```

Continued on next page

Table B.1 – *Continued from previous page*

Endpoint	200 Response Sample Output

`/epochs/<>`

```
{
  "EPOCH": "2023-048T12:00:00.000Z",
  "X": {
    "#text": "-5097.51711371908",
    "@units": "km"
  },
  "X_DOT": {
    "#text": "-4.5815461024513304",
    "@units": "km/s"
  },
  "Y": {
    "#text": "1610.3574036042901",
    "@units": "km"
  },
  "Y_DOT": {
    "#text": "-4.8951801207083303",
    "@units": "km/s"
  },
  "Z": {
    "#text": "-4194.4848049601396",
    "@units": "km"
  },
  "Z_DOT": {
    "#text": "3.70067961081915",
    "@units": "km/s"
  }
}
```

`/epochs/<>/speed`

```
{
  "value": 7.658223206788738,
  "units": "km/s"
}
```

Continued on next page

Table B.1 – *Continued from previous page*

Endpoint	200 Response Sample Output

`/epochs/<>/location`

```
{
  "closest_epoch": "2023-077T15:47:35.995Z",
  "location": {
    "altitude": {
      "units": "km",
      "value": 428.6137193341565
    },
    "geo": {
      "address": {
        "ISO3166-2-lvl4": "AO-BGU",
        "country": "Angola",
        "country_code": "ao",
        "state": "Benguela Province"
      },
      "boundingbox": [
        "-13.874445",
        "-11.7589169",
        "12.3159609",
        "15.1108341"
      ],
      "display_name": "Benguela Province, Angola",
      "lat": "-12.9104657",
      "licence": "Data \u00a9 OpenStreetMap contributors,
        ODbL 1.0. https://osm.org/copyright",
      "lon": "14.0356608",
      "osm_id": 1802540,
      "osm_type": "relation",
      "place_id": 298335923
    },
    "latitude": -13.479282638990789,
    "longitude": 13.126560404682472
  },
  "seconds_from_now": 0,
  "speed": {
    "units": "km/s",
    "value": 7.6577354899224375
  }
}
```

`/now`

```
{
  "closest_epoch": "2023-065T22:59:30.000Z",
  "location": {
    "altitude": {
      "units": "km",
      "value": 430.322553722729
    },
    "geo": null,
    "latitude": -15.297069227120538,
    "longitude": -193.6978998196031
  },
  "seconds_from_now": -8.718001127243042,
  "speed": {
    "units": "km/s",
    "value": 7.6552459153446755
  }
}
```

`/delete-data`

```
Data deleted from instance
```

`/post-data`

```
Data refreshed
```

Continued on next page

Table B.1 – *Continued from previous page*

Endpoint	200 Response Sample Output
`/help`	

```
These are the endpoints of the iss_tracker API.

Note that if the data is empty, all GET
messages will return a string message with a 404
status.

Note that epochs are in the form
YYYY-DDDTHH:MM:SS.000Z, or in URL-friendly form
YYYY-DDDTHH%3AMM%3ASS%2E000Z

        / GET Return the entire data set in JSON form.
        /epochs GET Return a list of all epochs in JSON form.
                int:limit The maximum number of epochs to return.
                int:offset What epoch to start from, zero-indexed
        /epochs/<epoch> GET Return the state vector for
                an epoch in JSON form. Returns a string error
                message with a 404 status if no such record.
        /epochs/<epoch>/speed GET Return the instantaneous
                speed for an epoch in JSON form. Returns a string
                error message with a 404 status if no such record.
        /epochs/<epoch>/location GET Return a dictionary of
                the latitude, longitude, altitude, and geoposition
                for an epoch in JSON form. Returns a string error
                message with a 404 status if no such record.
        /now GET Return a dictionary of the latitude,
                longitude, altitude, and geoposition for the closest
                epoch to present in JSON form. It also indicates
                how far away in time the closest epoch is.
        /delete-data DETETE Clear all data in the instance.
        /post-data POST Update and overwrite all data.
        /comment GET Return comment list from data.
        /header GET Return header dictionary from data.
        /metadata GET Return metadata dictionary from data.
```

Continued on next page

Table B.1 – *Continued from previous page*

Endpoint	200 Response Sample Output

`/comment`

```
[
  "Units are in kg and m^2",
  "MASS=473413.00",
  "DRAG_AREA=1618.40",
  "DRAG_COEFF=2.20",
  "SOLAR_RAD_AREA=0.00",
  "SOLAR_RAD_COEFF=0.00",
  "Orbits start at the ascending node epoch",
  "ISS first asc. node: EPOCH = 2023-03-03T16:45:01.089
    $ ORBIT = 2542 $ LAN(DEG) = 78.61627",
  "ISS last asc. node : EPOCH = 2023-03-18T14:19:09.505
    $ ORBIT = 2773 $ LAN(DEG) = 26.64425",
  "Begin sequence of events",
  "TRAJECTORY EVENT SUMMARY:",
  null,
  "|       EVENT        |       TIG        | ORB |   DV    |   HA    |
    HP    |",
  "|                    |       GMT        |     |   M/S   |   KM    |
    KM    |",
  "|                    |                  |     |  (F/S)  |  (NM)   |
    (NM)   |",
  "====================================================================
    ==========",
  "GMT 067 ISS Reboost   067:20:02:00.000              1.0     427.0
    407.3",
  "(3.3)   (230.6)   (219.9)",
  null,
  "Crew05 Undock         068:08:00:00.000              0.0     427.0
    410.8",
  "(0.0)   (230.6)   (221.8)",
  null,
  "SpX27 Launch          074:00:30:00.000              0.0     426.7
    409.9",
  "(0.0)   (230.4)   (221.3)",
  null,
  "SpX27 Docking         075:12:00:00.000              0.0     426.7
    409.8",
  "(0.0)   (230.4)   (221.3)",
  null,
  "=====================================================================
    =========",
  "End sequence of events"
]
```

`/header`

```
{
  "CREATION_DATE": "2023-063T04:34:04.606Z",
  "ORIGINATOR": "JSC"
}
```

`/metadata`

```
{
  "CENTER_NAME": "EARTH",
  "OBJECT_ID": "1998-067-A",
  "OBJECT_NAME": "ISS",
  "REF_FRAME": "EME2000",
  "START_TIME": "2023-062T15:47:35.995Z",
  "STOP_TIME": "2023-077T15:47:35.995Z",
  "TIME_SYSTEM": "UTC"
}
```

Appendix C

Special Consideration for /now Endpoint

The longitude is calculated from the J2000 position vector, accounting for Earth's daily rotation relative to the sun. However, it does not account for Earth's orbit around the sun. Every time Earth completes a rotation relative to the sun, it completes slightly more than one rotation relative to the J2000 coordinate system. This unaccounted rotation means that a correction term needs to be added. Around the beginning of March, this term is 24.