Overview

I built a simple backend with Python + FastAPI using Python 3.10 and a small Nuxt 3 frontend. I have very little experience in frontend development with javascript (I usually only use dart + flutter for frontend development), so I had to study Nuxt 3 and javascript frontend development from scratch and I created only a very simple frontend, which I'm sure is not written as well as it can and should be.

I tried to write everything the way I would write production code, however of course I had to make some assumptions on coding guidelines and whatnot: for example, different companies could have different conventions on how docstrings should be written, whether and when variables should be protected (via double leading underscores or a single leading underscore), what packaging tool to use, where to put licenses, if and when type hints are mandatory and so on.
Also, unluckily I worked on this project entirely on the weekend, so I couldn't ask any questions and had to make some assumptions.

Backend

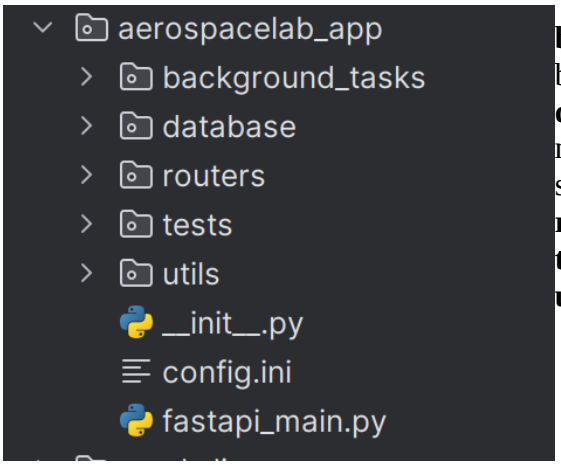
Project Structure and packaging tool

The project contains one python package that I called challenge and at the same level there's a a README.md file as well as a requirements.txt file and a puml directory with UML class and flowchart diagrams. The README specifies that the python version to use to run the project is 3.10: other python versions might require different versions of the dependencies. There are just a few dependencies:

- **aiohttp**: for making asynchronous HTTP requests
- **aioschedule**: to schedule asynchronous jobs
- **fastapi**, **unicorn** and **slowapi**: for FastAPI and throttling
- **SQLAlchemy** and **pydantic**: for DB
- **pytest, pytest-asyncio, pytest-mock** and **httpx**: for tests

Running pip install -r requirements.txt will install all needed packages.

The challenge package contains several packages as shown in the following pic:



background_tasks: includes tasks that must be run in background
database: includes what's needed for the database, including models, and it's also the folder where the locations.db file is stored
router: includes all routers for FastAPI
tests: includes all unit tests
utils: includes several utils modules

fastapi_main is the entrypoint for the app: running it spawns a webserver and exposes several endpoints thanks to FastAPI. It also schedules a background task to run and sets up the DB.
config.ini: contains configuration data

Initial Development Approach

The test required to retrieve the status of the ISS at most once every 20 seconds via this endpoint: https://api.wheretheiss.at/v1/satellites/25544
It then required to expose 2 endpoints for getting the latest iss position and its daylight time windows. That means the software needed to have some kind of schedule to retrieve the latest ISS status every 20 seconds. Moreover, with this 20 seconds betweem requests limitation and no useful enpoints to get multiple ISS positions (we can at most get positions at 10 different timestamps, which is a very little amount), I thought that a good solution would be to query the endpoint every 20 seconds and persist the provided location of the ISS, which will then be used by our API to calculate the time windows when requested.

Hence, the software also needed some kind of database to store the status and then of course a way to expose some endpoints.

Since this is just a challenge and not real code that will be deployed to production, I decided not to implement a horizontal-scaling solution overcome the limitation of being able to query the external endpoint only once every 20 seconds. More on this in the last chapter.
For the same reasons as above, I decided not to separate the code that retrieves the ISS position from the FastAPI code that exposes the endpoints.

For the schedule, I decided to use the aioschedule library.
For the database, since this is just a simple app, I decided to use SQLite with SQLAlchemy as its implementation is incredibly easy and straightforward, but in a real production environment I would analyze the current and possible future needs of a database before selecting one.
For exposing endpoints, I decided to use FastAPI with Python, as I mainly work with Java rather than Python in my current full time job (at the moment around 80% Java and 20% Python) and I wanted to spice things up a bit.

So the first think I did was using the TTD approach to create the IssPositionUpdater class: I first created some barebone tests in test_iss_position_updater, and then I went on to create IssPositionUpdater and all database related packages and classes, such as database.py and IssPosition. After creating all of them, I went on to expand the tests and make them run successfully.

After that, I kept using a TTD approach by creating test_fastapi_main and test_iss_router to write some barebone tests for fastAPI: even though fastAPI docks suggest to use a real test DB, I decided not to go that route because I believe unit tests should be completely independent from the DB, and one should use a mocked DB implementation, hence I decided to use the pytest-mock library to mock it.
After creating fastapi_main and iss_router, as usual I expanded the tests and made them work.
I decided to include the ISS router in a different package as I think it’s a good idea to do that in case the project grows to have multiple endpoints.

After all that, I decided to also implement a few bonus features:

- An IP-based API throttling that only allows the same IP to query certain endpoints a configurable amount of times in a certain time range
- A config.ini file and a ConfigUtils class to be able to configure without touching code the following variables: the URL to use to get the current IssPosition from the external API, the user agent to use for the request, the wait time between consecutive requests, the rate limit for the iss_router and a flag to enable or disable API throttling.
- A small easter-egg that is shown when querying the root endpoint
- The /iss/position and /iss/sun endpoints allow some query parameters to customize the response

What's a daylight time window?

One thing that came to mind during development was: what's the definition of a daylight time window? Let's say we have the following data set with random times:

18:30 Daylight, 18:40 Eclipsed, 18:50 Eclipsed, 19:00 Daylight, 19:10 Daylight, 19:20 Eclipsed, 19:30 Daylight

The times are all 10 minutes apart, which is a similar situation to the real one we'd have with the software, as we'll have all times at least 20 seconds apart. What are the daylight windows here?

I thought about different possibilities:

1. *Undefined – 18:40 | 19:00 – 19:20 | 19:30 – Undefined*
2. *Undefined – 18:30 | 18:50 – 19:10 | 19:20 – Undefined*
3. *Undefined – 18:30 | 19:00 – 19:10 | 19:30 – Undefined*

The first approach: the daylight time window starts at the first daylight time after eclipsed and ends at the first eclipsed time after daylight.
The second approach: the daylight time window starts at the last eclipsed time before daylight and ends at the last daylight time before eclipsed.
The third approach: the daylight time window starts at the first daylight after eclipsed and ends at the last daylight before eclipsed

I believe all approaches can make sense. The first two can be incorrect for a few seconds, but no time is "lost" if we look at eclipsed time windows, which will also be incorrect for a few seconds. This approach can also be improved by taking some assumptions or calculate a possible "real" time window: for example by not using the first or last eclipsed time but those +/- 5 minutes in this case.

The third is never incorrect but some time will be "lost" if we calculate the eclipsed time windows in the same way.

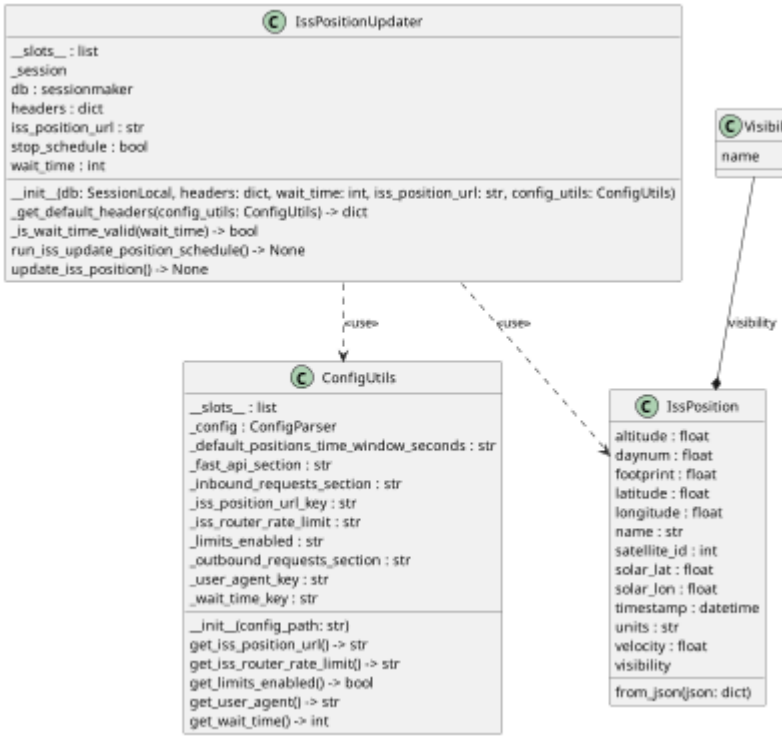
In the end, I decided to write code with a possibile future refactoring in mind, hence I wrote time_windows_utils and its function get_daylight_time_windows that follows the first approach but that can be easily modified to follow any of the other approaches, as seen by some commented out code at the end of the module, which I left just to show how easy it is to refactor it, but that I would definitely not keep in real production code.

Sofware Architecture

The software is a bit of a mix between procedural and object-oriented types of programming.

The part related to FastAPI (exposing endpoints) is done in the classic procedural style, which I think is very pythonic, as it emphasizes readability, simplicity, and explicitness.
The part related to the retrieval of the IssPosition and the update of the local DB is done in an object-oriented style, as IssPositionUpdater must be initialized with several arguments and OOP is a great choice for encapsulation and Complexity Management.

The following is the UML class diagram of IssPositionUpdater and the related classes:



IssPositionUpdater: It's the class responsible of updating the iss_positions table in the DB with new positions obtained by querying the given endpoint iss_position_url. It can be used by calling update_iss_position() directly or by calling run_iss_update_position_schedule(), which will call update_iss_position()

ConfigUtils: utils class that is used by IssPositionUpdater to get manually configured arguments that are required by IssPositionUpdater in case they are not provided when creating the instance

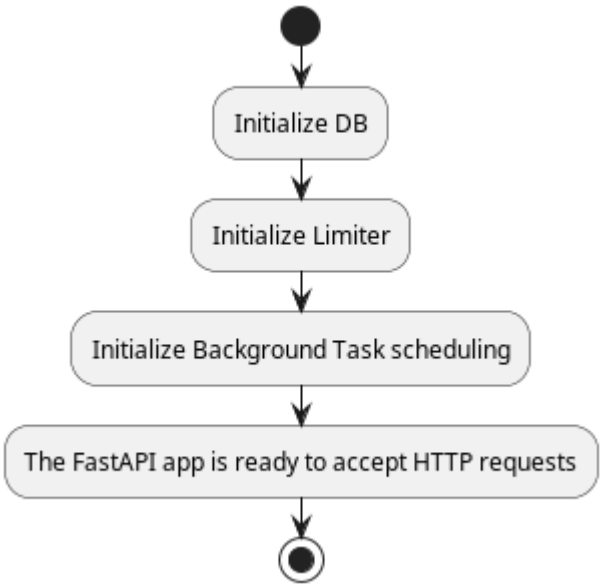
IssPosition: the IssPosition schema with all the fields that are obtained when querying the iss_position_url. The from_json method is used to create an IssPosition instance given the json obtained as the http response.

Visibility: an enum of the visibility used by IssPosition

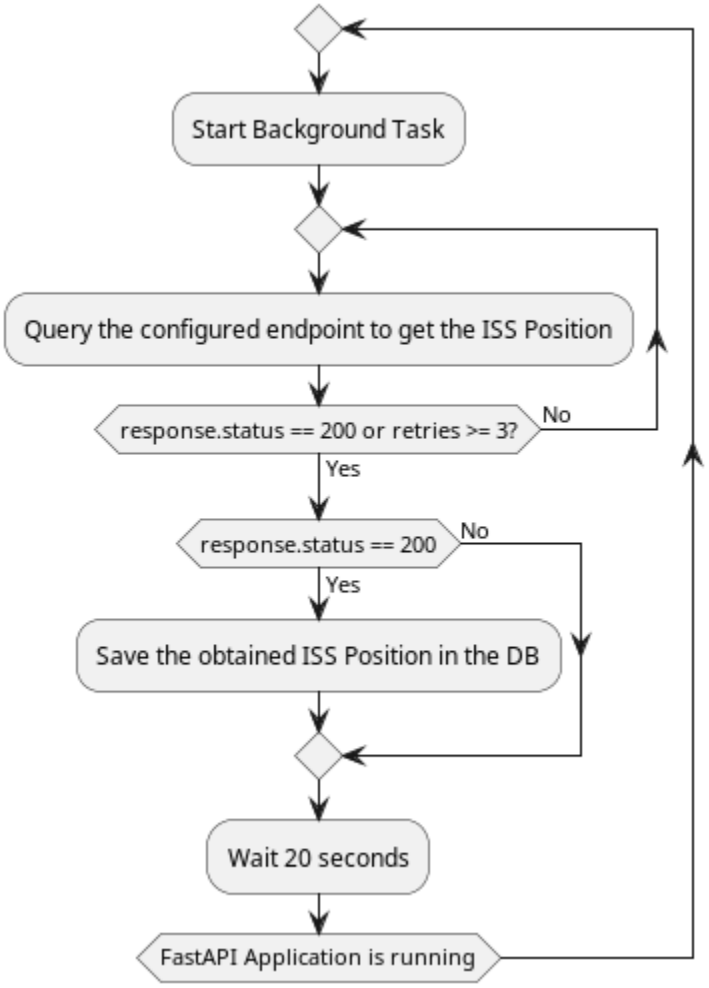
Below are self-explanatory flowcharts that show:

- How the FastAPI app is started
- How the background task for getting the IssPosition works
- How the FastAPI app handles HTTP requests

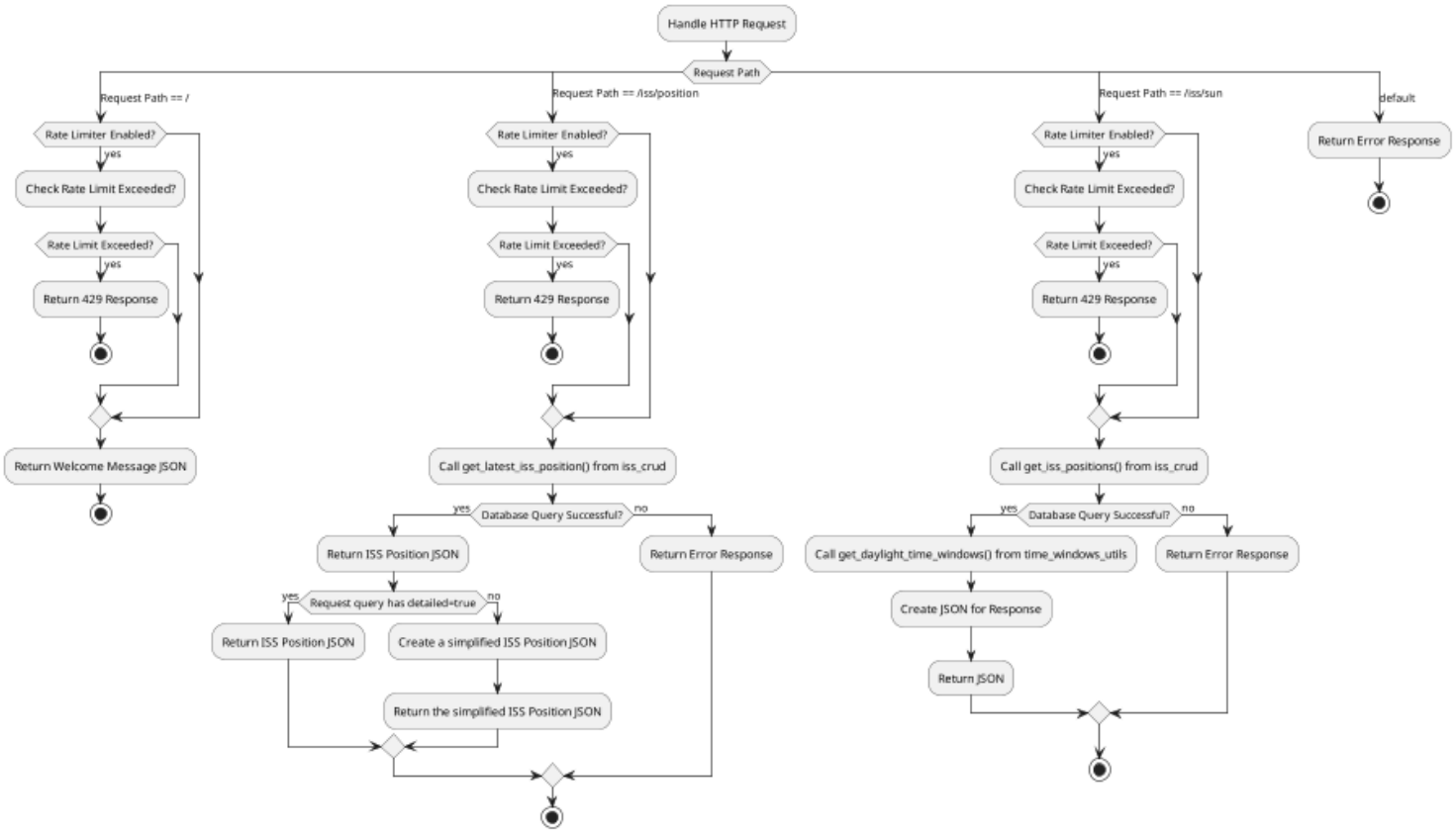
Startup diagram:



Background Task Diagram



HTTP Request Handling Diagram



Tests and coverage

The unit tests are located in the /challenge/tests python package. They include unit tests for:

fastapi_main.py
iss_router.py
time_windows_utils.py
iss_position_updater.py

They make use of pytest-mock a lot: for example, in order to test iss_router's method read_position, without mocks one would need to have a proper DB and let get_latest_iss_position() run to get the latest iss position from the local DB. However, that would defeat the unit test's key aspect: isolation.

Hence, even though FastAPI docs suggest to use a test DB to test the endpoints, I decided to mock the

calls that would need to make use of the DB. I used the pytest-mock library as that's the one that is mainly used in Python, but in Java I would use Mockito as I'm very experienced in using it and I love how powerful it is.

I did not write some tests on purpose: I did not write tests for database.py or iss_crud.py, as I think they should be written as integration tests and not unit tests, and integration tests were out of the scope of this challenge (and moreover, a real database was actually not a requirement for the test).

In a real production environment, I would aim for > 80% coverage between unit and integration tests. If this was a real app to deploy in production, the next step I would take to improve tests is running a test coverage analysis and then write more unit and integration tests to cover all uncovered lines.

Frontend

As I previously said, I had no real experience in Vue/Nuxt and Javascript programming, hence I had to spend some hours studying them from scratch. I ended up creating a very simple and functional (albeit ugly) frontend, as shown below:

How it works

As explained in the README.md file, an .env file must be created at the root of the project with a few options:

```
MAPBOX_ACCESS_TOKEN=INSERT_ACCESS_TOKEN
ISS_POSITION_URL=http://localhost:8000/iss/position
ISS_SUN_URL=http://localhost:8000/iss/sun
UPDATE_WAIT=20000
```

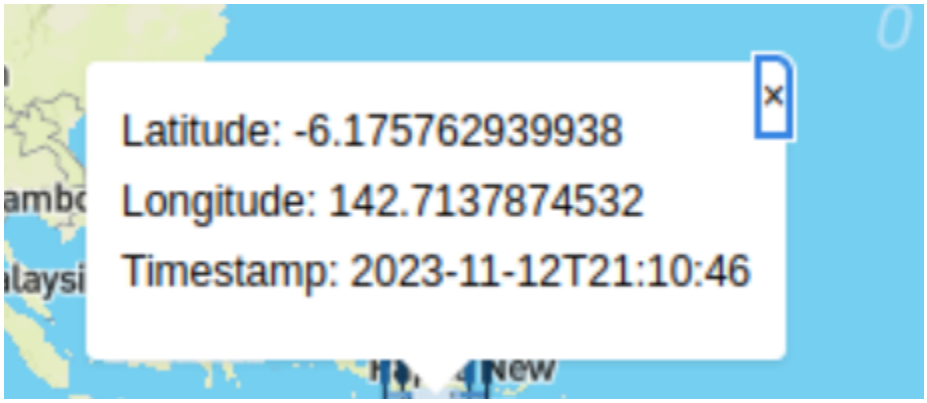
In particular, we need the mapbox access token, the url where to query the ISS position from, the url where to gen the daylight time windows from and the time in milliseconds to wait between requests to the same endpoint.

I created an IssMap component written as a Vue Single-File Component with the composition API style.
The component retrieves the ISS position and its daylight time windows in the last 24 hours by using the axios library to perform asynchronous HTTP requests to the configured endpoints.

The requests are performed every configurable amount of milliseconds, which in my case I set to 20 seconds (doing it more often would not be very beneficial, as a limitation of the backend is that it can only retrieve the ISS position every 20 seconds) via the setInterval function, which according to the docs “Schedules repeated execution of `callback` every `delay` milliseconds.”

So every 20 seconds the web page is updated with the new ISS Position and eventually new Daylight time windows.

Also, clicking on the ISS marker opens a pop like this:



Limitations and improvements

- Frontend can be made much better. If I had time to learn more about Nuxt/JS, I would probably build something like this:

On the left I would keep a scrollable view with the daylight time windows, but with a nicer style and graphics. On the map I would put also past and expected future orbit of the ISS and general information on it, such as velocity and orbital elements.

- The icons used by the frontend (the sun and ISS) must be selected to be copyright free, or their copyright should be included in the project or new ones should be created
- Define how to calculate the daylight time window given a discrete representation of the ISS position and modify the code accordingly (or verify that the current one is ok), or even better, provide the choice of what kind of approach to use to calculate the time window via the config.ini file.
- Better backend and frontent error handling
- Choose a better suited DB
- Use a better solution for API throttling (e.g. using a redis DB, implement DDOS protections, implement user registration/login and allow registered user to have more lenient API throttling)
- Write frontend tests
- Improve backend tests coverage and write integration tests
- Write integration tests between frontend and backend
- General optimization and clean up
- Use some kind of interpolation tecnique to better estimate the daylight window

Specific improvement: Horizontal Scaling for Rate-Limited API Access

I thought about a way to overcome the limit of 1 request every 20 seconds of the [whereitheiss] API: horizontal scaling.

The background task code could be separated from the FastAPI code and use the same remote DB. The background task job could be deployed on multiple servers with unique IP addresses, so that we can

increase the query rate capacity by distributing the API queries across different servers with different IPs.

This architecture involves:

1. **Multiple Servers:**
    - Deploying the background job on multiple servers, each with a distinct IP address.
2. **Orchestrator:**
    - Implementing a central coordinator that manages the distribution of tasks among the deployed servers. The coordinator ensures that API queries are evenly distributed among the servers, optimizing the usage of available IP addresses.
3. **Synchronization:**
    - Implementing a synchronization mechanism to ensure that each server queries the API at the specified intervals, adhering to the rate limit policy.
4. **Scalability:**
    - Allowing for easy addition or removal of servers based on demand.
5. **Monitoring and Error Handling:**
    - Implementing monitoring and error handling mechanisms to identify and address any issues that may arise during the API querying process.