OrbitIQ Project

Overview

Our code is located at: https://github.com/OrbitIQ/orbit-iq

Installation / Deployment

This section is about how to install and setup everything you need to run the app

Running Locally

First, install Docker Desktop (suggested). This should also install Docker compose. Navigate to the root directory of our project so you're in the same folder as docker-compose.yml then to start the app all you need to do is run docker compose up -d and it will run as a background process.

If you want to shut down the app you can run docker compose down in the same folder and it should stop the app for you.

If you're running into errors and need to see the logs you can run docker compose logs

Deployment

This will mostly depend on where you're deploying but here's some basic steps and advice.

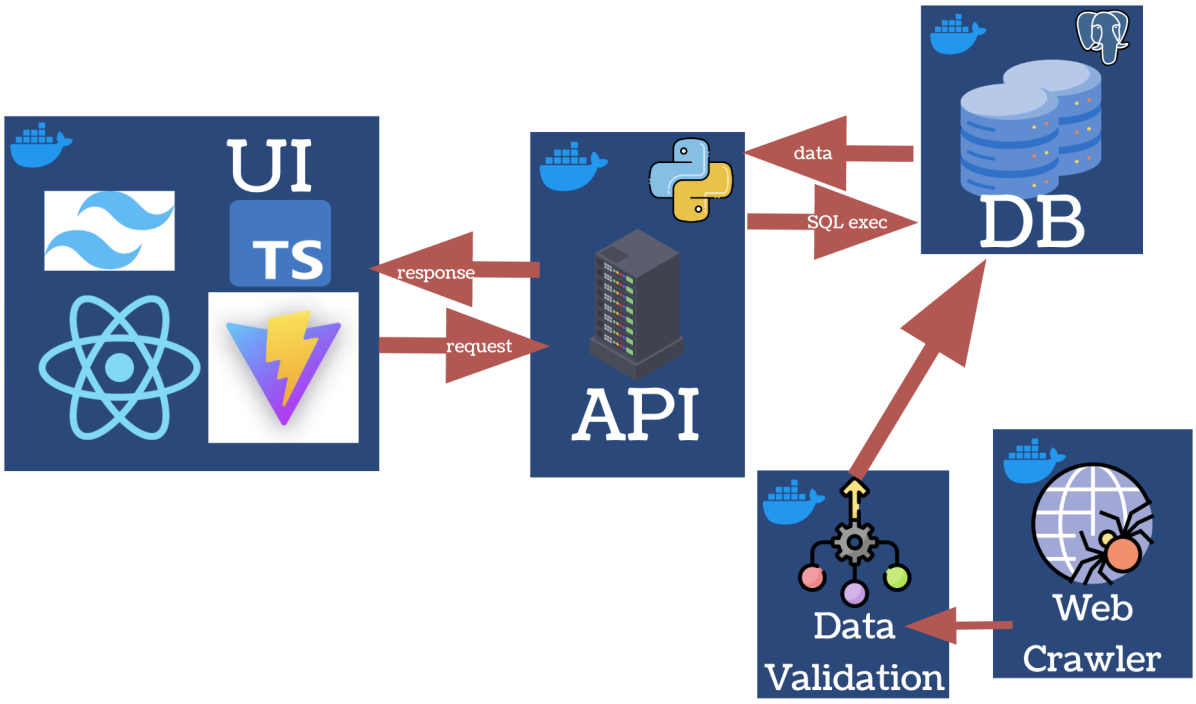
1. Change the database credentials to something more secure in the docker-compose.yml
2. You'll need to expose the website port (3000) to the public in addition to the API port (8080)
    a. When you know the IP address of where the API is deployed you'll need to update the API's address in the website/src/Constants/constants.ts file to reflect where it's actually being hosted
    b. **Note**: it's probably better to just manually build the react app, if you have nodejs and npm installed
        i. Run Commands
            1. cd website
            2. npm install
            3. npm run build
        ii. This will create a static build of the website (in the dist folder) which will be more efficient and likely more secure than just running it through the docker container which then you can just serve as a normal static website

Services

This project contains multiple services required to support this web app. We'll go into more detail in the sub heading below but they are all defined in docker-compose.yml and all interact with each other.

Connectivity Visualization

Here's a little visualization of how all of our different services shown by the blue boxes interact with each other.



Website (UI)

Our website for this is written in React.js with TypeScript and Tailwind and Vite.

Our website is structured in the normal website folder structure with configuration files at the top level and all of the project code within the /src folder. There are a ton of files that make up different parts of the website, so we will list the general structure

/components: All of the custom components we created and imported for this project
- **/Authentication**: All of our authentication components and pages
- **/Navbar**: Navbar component
- **/Table**: general and reusable table component
- **/ChangelogTable**: Changelog table based off table component

/SatelliteTable: Satellite table based off table component
/UpdateTable: Update table based off table component
/Toast: Creates the popup in the bottom right on an error
/ui: A list of simple styled React UI elements like a button, input field, table, etc
/Constants: Our data schema for formatting data from the database
/Layouts: Layouts are templates for reusable code, we use this for more Toast logic
/lib: basic tailwind helper functions
/pages: Satellite, Changelog, and Updates data view pages rendered by react-router
/requestLogic: Satellite, Changelog, and Updates Axios data fetching logic
/services: We have one service with is the AuthService.ts that handles requests and logic around authentication
/types: TypeScript types
/App.tsx: Top-level component describing general page behavior

API

The API's written in Python and uses Flask to make writing API routes more convenient.

Here's the structure of the main parts of the API.
app.py - Entry file
routes / - each subpath on the API has its own route.
- **authentication.py** - Handles account creation/login/deletion
- **confirmed.py** - Handles routes for the current satellites that have been **approved**
- **edit.py** - These routes are for when a user directly modifies the confirmed table
- **proposed.py** - These routes are for proposed changes from the web crawler

Database (DB)

Our database is a PostgreSQL instance currently just running on a Docker container. You can make changes to the root username, password, database name, and more using the environment variables passed to this service in the docker-compose.yml file.

You **should** change these and make sure the information is not exposed publicly otherwise people can access and modify the database.

init-db

This is the service that sets up the database. It executes the script **db/init_db.py** and executes and seeds the database at startup.

Note: because of the nature of this project, we have not implemented or used any existing database migration handling, so if you make a change directly to any tables created in **init_db.py** it will not be updated since this script is meant to be run once. If schema changes are needed, you'll need to access the running db service and execute your migration there.

Crawler

The crawler's main idea is to crawl our data sources and dump the **raw data** it finds into the **crawler_dump** table.

> The reason we are dumping the raw data here is that in web scraping, it's super common that the host of the data might change the way they are formatting data, or add new information, and other information like that. So we want to make sure we're capturing all that data regardless if it's the structure/format we're expecting.

Service File Structure
crawl.py - The main crawling logic that spins off the crawling for each source
sources / - These contain functions for extracting data for each source
- **aerospace.py** - Extracts data from aerospace's re-entry data
- **gcat.py** - Extracts data from GCAT
- **in_the_sky.py** - Extracts data from in the sky's satellite information

Validator

The validator's job is to take the **raw data** from the **crawler_dump** table and: extract, sanitize, combine, and export these changes to the **proposed_changes** table which is what the end user will see that the crawler has detected.

Service File Structure
validator.py - This is the entrypoint for the service and extracts all the rows from crawler_dump
proposed_change.py - This contains the class definition for a ProposedChange and logic that will insert a ProposedChange into the **proposed_changes** database table.
mappings / - These hold the corresponding functions to map a source from the crawler_dump to a proposed change
- **aerospace.py**
- **gcat.py**
- **in_the_sky.py**

Note: This is a likely service that could start causing problems, as if the format of the crawled data is changed it will cause problems here.

Troubleshooting

You can view all the logs this application is generating with docker compose logs
You can also view logs for a specific service ex: crawler with docker compose logs crawler

What Might Break

In our opinion the most likely thing to break is the crawler service because it's the only service that relies on an external site not to change anything. If like external sites change the way they're showing information it would make it more difficult

Future Improvements

- Currently the proposed changes only consider one source at a time, so that if we have two sources having conflicting or supporting information about the same satellite at the moment this will generate two proposed changes
- Better data formatting, parsing, and formatting into a consistent manner
- Map the export to excel function to have the exact same format and column names as UCS's release data
- Currently the database considers the official_name as the primary key, however this causes problems when there's alternative names or names that are different (ex: Starlink-34 vs Starlink 34)
    - This should probably be converted to NORAD ids would allow for easier merging of these sort of formatting issues because we are confident the NORAD ids should be unique and reference the same entity across any data source

Recommended Usage

Note: It's probably best for researchers to make regular backups/exports of the approved table in case the database is deleted.

Restoring To a UCS Release

Laura mentioned that she may want to reset the database to a previous UCS release. We didn't have time to fully implement this ourselves as a functionality. However, our current app initially seeds the database with UCS's Jan 1st 2022 release of satellite data. A maintainer of the deployment of this application could do the following to reset the database with a new version of UCS information.

Note: This will remove any other docker containers, images, and volumes on the machine. This also is a full reset of the app's data including loss of information about: changelog history, users, current proposed changes from the crawler, and more.

1. Stop the docker containers with **docker compose down**
2. Clean up docker containers, images, and volumes.

    a. Stop any containers: **docker stop $(docker ps -aq)**
    b. Remove any containers: **docker rm $(docker ps -aq)**
    c. Remove docker images: **docker rmi $(docker images -q)**
    d. Remove the database volume: **docker volume prune**
3. Replace **UCS-Satellite-Data.csv** in the **db** folder with the latest release or version you wish to restore to.
4. Restart the app with **docker compose up -d** using the steps at the top of the document
    a. It will automatically load in the new data into the approved/confirmed satellite data table automatically.