

Documentation and setup guide n2yo tracker

by Luke Redmore

Submission date: 03-Jan-2026 04:39PM (UTC+0000)

Submission ID: 271953574

File name: 41367_Luke_Redmore_Documentation_and_setup_guide_n2yo_tracker_708810_492932913.docx
(1.44M)

Word count: 2841

Character count: 16954



DOCUMENTATION

N2YO satellite tracker, setup and overview

Abstract

This project implements a satellite observation tracker using Node.js and Express, integrated with the N2YO API for real-time satellite data. It features a modular architecture with secure middleware, CORS support, and SSRF mitigation. The front end is a Single Page Application (SPA) that dynamically updates satellite and observation data without page reloads. The solution demonstrates best practices in API integration, error handling, and responsive UI design for efficient satellite tracking and observation management.

Luke Redmore

2219318@student.uwtsd.ac.uk

Contents

Getting started, setup and runtime	2
Prerequisites:	2
Documentation (code overview).....	4
Project pipeline / workflow	4
Dependency tracking and management	5
Zoning into the dependencies	6
Express	6
Express handling parsing.....	6
CORS cross origin resource sharing.....	8
Getting data from N2YO	8
Data processing and endpoints.....	11
Helper functions:.....	14
In memory storage	15
Front end.....	16
app.js.....	16
DOM manipulation	16
Overview of DOM manipulation.....	17
Index.html.....	17
style.css.....	18
Data flow:.....	18
Key Integration Points:	18
Observation pipeline.....	23
Initial loading:	23
Creating a new observation.....	23
Update observation status.....	25
Deleting an observation	27
Technical overview data flow	28
Testing	30

Getting started, setup and runtime

Prerequisites:

- *Install the project via the zip file provided to the Moodle classroom and unpack it into a directory*
- *Have a stable LTS or latest version of NodeJS and NPM installed (i.e. node 24.12.0 NPM 11.7.0)*
- *Ensure the .env file contains valid, API key, base URI, app and port values as well as a TLS reject for bypass of self-signed certificates in trust store NOTE: this may not be needed for DOCCUMENTATION*
- *machine however due to recent tech refresh the dev device required this.*
- *Vscode PowerShell or cmd instances to use in the workspace or a remote terminal with node and npm installed*

This project uses NodeJS as a runtime environment meaning project metadata and its dependencies as well as associated scripts are all contained within the package JSON file, to install the project and lock specific versions into the package lock file please run

➤ `npm install`

Starting the project is quite simple as inside the package file:

```
"scripts": {  
  "start": "node server.js"
```

This allows the entire stack to be spun up with:

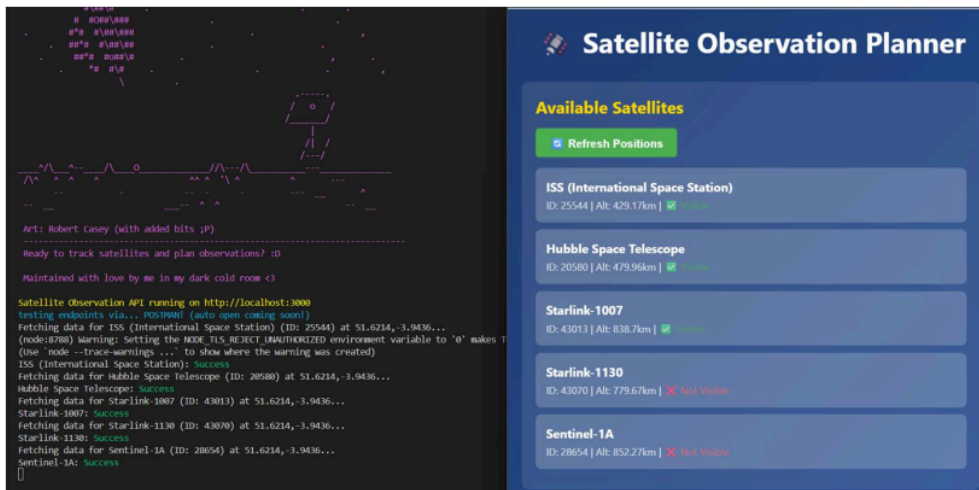
➤ `npm start`

If the project has installed correctly, you will see the following output:

(For the sake of convenience feel free to run `npm install && npm start`)

[illegible]

Upon seeing this the stack is running and listening on port 3000. (Feel free to adjust this should any native applications be using it, this can be adjusted in the .env file)



Note if using TLS reject unauthorised variable, warnings will appear, as this is only a local instance it's okay. However, this is not at all best practice and is merely a quick workaround to avoid changing the code so close to the deadline.

That concludes the quick overview of getting the satellite observation tracker up and running and so begins the documentation.

Documentation (code overview)

Project pipeline / workflow

Workflow

1. **Planning Stage**: User selects satellite and plans observation time
2. **Observation Stage**: User marks observation as "observed" or "missed"
3. **Updating stage**: User can update observation with new status

Dependency tracking and management

Package file JSON

```
{
  "name": "satellite-observation-planner",
  "version": "1.0.0",
  "description": "API for planning satellite observations",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "node server.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "cors": "^2.8.5",
    "node-fetch": "^2.6.7",
    "dotenv": "^16.0.3",
    "csurf": "^1.11.0"
  }
}
```

This contains all the dependencies and metadata for the project.

Metadata:

- name: Project identifier for npm registry and imports
- version: Tracks releases using semantic versioning (major.minor.patch)
- description: Brief explanation for npm search and documentation
- main: The entry point for the project, when another person downloads or imports the project
- scripts: custom scripts run via npm run <script-name>
- dependencies: External packages that the project needs to run and function
- author/licence: legal attribution information for published packages
- Keywords: optional but can help others find the project easier
- Repository: Used to link source code location (i.e. where it came from if using source control)

Zoning into the dependencies

```
"dependencies": {  
  "express": "^4.18.2",  
  "cors": "^2.8.5",  
  "node-fetch": "^2.6.7",  
  "dotenv": "^16.0.3",  
  "csurf": "^1.11.0"
```

Express

A minimal web framework for NodeJS that simplifies building web servers and APIs, it handles and facilitates standard HTTP requests such as GET, PUT and DELETE. Routes specific URI's to specified functions for precise handling of screen flow via backend logic. Express also manages middleware this being the logging, authentication anything that facilitates front and backend interaction.

Express also handles static files such as HTML, JS and even CSS allowing for separation of these files for better maintainability and convention, while also facilitating the parsing request data.

Handling HTTP requests via express:

```
const express = require('express');  
const app = express();  
  
// Route handling  
app.get('/api/satellites', (req, res) => {  
  // Handle GET request  
});  
  
app.post('/api/observations', (req, res) => {  
  // Handle POST request  
});
```

The above example is a streamlined version of how it is implemented in the satellite app, this implementation offers effective handling with minimal code wherein without express this would be achieved via NodeJS's built-in http module which would require a far more verbose implementation.

Express handling parsing

Raw HTTP request body (string):

```
'{"satelliteId": 25544, "plannedTime": "2024-01-15T20:30:00"}'
```

Put simply the received payload is one long string, this isn't very useful as it will be undefined or unlabelled meaning doing things such as this:

```
// Raw string from HTTP request
```

```
const rawData = '{"satelliteId": 25544, "plannedTime": "2024-01-15T20:30:00"}';
```

```
console.log(rawData.satelliteId); // undefined
```

```
if (rawData.satelliteId === 25544) // false (comparing string to number)
```

Impossible, however thanks to express middleware:

```
// Parsed JavaScript object
```

```
const parsedData = { satelliteId: 25544, plannedTime: "2024-01-15T20:30:00" };
```

```
console.log(parsedData.satelliteId); // 25544
```

```
if (parsedData.satelliteId === 25544) // true
```

```
const { satelliteId, plannedTime } = parsedData; // destructuring works
```

In this codebase this is achieved via:

```
const { satelliteId, satelliteName, plannedTime, userLocation } = req.body; (converting unusable text into workable data structures for server logic)
```

Complete example from code:

```
app.post("/api/observations", (req, res) => {
```

```
  const { satelliteId, satelliteName, plannedTime, userLocation } = req.body;
```

```
  const newObservation = {
```

```
    id: Date.now(),
```

```
    satelliteId,
```

```
    satelliteName,
```

```
    plannedTime,
```

```
    userLocation: userLocation || "Unknown",
```

```
    status: "planned",
```

```
    createdAt: new Date().toISOString(),
```

```
  };
```

```
data.observations.push(newObservation);  
writeData(data);  
});
```

CORS cross origin resource sharing

Getting data from N2YO

```
const cors = require("cors");  
app.use(cors());
```

Cors allows an API to accept requests from different origins such as domains, ports or via different protocols. Without a cors policy browsers block these requests for security.

While this entire stack listens on port 3000 meaning technically it's not needed as all the resource sharing occurs down the same port via in this case the same protocols implementing cors is still best practice as it allows the flexibility of running the front end on a different platform or separately via a build tool.

Regarding external calls to N2YO however cors is required as the payload or incoming request is of a different domain as such without a cors implementation no data could be accessed.

Setup node-fetch middleware

```
const fetch = require("node-fetch");
```

Use node-fetch to get satellite data from n2yo.com which makes a fetch or get request via a uri that gets built up earlier in the code.

```
const response = await fetch(apiUrl);
```

Here is the apiURI

```

// Popular satellites to track (verified IDs, find cooler way to do this later, maybe iterate through the catalogue? Find a db full of lots of sat's!)
const SATELLITES = [
  { id: 25544, name: "ISS (International Space Station)" },
  { id: 20580, name: "Hubble Space Telescope" },
  { id: 43013, name: "Starlink-1007" },
  { id: 43070, name: "Starlink-1130" },
  { id: 28654, name: "Sentinel-1A" },
];

// help prevent SSRF attacks ensure ONLY https:// can be parsed (fix this later)
function isValidProtocol(url) {
  const protocols = ["https://"];
  return protocols.some((protocol) => url.startsWith(protocol));
}

// Fetch satellite position from NZYO API (default: Swansea, Wales)
async function getSatellitePosition(satelliteId, lat = 51.6214, lng = -3.9436) {
  try {
    // Input validation to prevent SSRF
    if (
      !Number.isInteger(satelliteId) ||
      satelliteId < 1 ||
      satelliteId > 90000
    ) {
      console.error(`Invalid satellite ID: ${satelliteId}`);
      return null;
    }
    // These COORDS would be nigh on impossible / silly
    if (lat < -90 || lat > 90 || lng < -180 || lng > 180) {
      console.error(`Invalid coordinates: ${lat}, ${lng}`);
      return null;
    }
    // Construct and validate the API URL (passes in lat 'n lng)
    // node-fetch uses this URL to make the request
    const apiUrl = `${NZYO_BASE_URL}/satellite/positions/${satelliteId}/${lat}/${lng}/0/2/?apiKey=${NZYO_API_KEY}`;
  }
}

```

This is how the URI to get the relevant satellite data is built up, this occurs in several stages across a series of functions.

```

const SATELLITES = [
  { id: 25544, name: "ISS (International Space Station)" },
  { id: 20580, name: "Hubble Space Telescope" },
  { id: 43013, name: "Starlink-1007" },
  { id: 43070, name: "Starlink-1130" },
  { id: 28654, name: "Sentinel-1A" },
];

```

First a variety of satellites are hardcoded in, eventually these will be used to populate the fetch request sent to n2yo.com.

The following logic was introduced to mitigate a vulnerability in the code to server-side request forgery (SSRF), SSRF being where an attacker uses the server to make a connection with devices on the internal facing network.

```

// help prevent SSRF attacks ensure ONLY https:// can be parsed
function isValidProtocol(url) {
  const protocols = ["https://"];
  return protocols.some((protocol) => url.startsWith(protocol));
}

```

This works by ensuring that the protocol used is https only, ensuring that only secure connections can be made to the server via n2yo.com.

```

const data = await response.json();
// handle downtime / issues with incoming response avoid pulling my own hair out
if (data.error) {
  console.error(`N2YO Error for satellite ${satelliteId}:`, data.error);
  return null;
}

if (data.positions && data.positions.length > 0) {
  const pos = data.positions[0];
  return {
    latitude: pos.satlatitude,
    longitude: pos.satlongitude,
    altitude: pos.sataltitude,
    visible: pos.eclipsed === false,
  };
}
} catch (error) {
  console.error(`Error fetching satellite ${satelliteId}:`, error);
}
return null;
}

```

```

if (data.positions && data.positions.length > 0) {

```

```

  // Only executes if positions array has data

```

```

  const pos = data.positions[0];

```

```

  return { latitude: pos.satlatitude, ... };

```

```

}

```

```

// If no positions data, falls through to:

```

```

return null;

```

The logic above handles n2yo.com downtime by returning an error which makes debugging more intuitive. While also gracefully handling API errors by performing a safety check against the position data received, if it is null an error is thrown alongside the satellite ID, `const pos = data.positions[0]`; extracts the first position object from the N2YO API response array.

```

// N2YO API configuration (DO NOT PUSH ENV TO GIT)
const N2YO_API_KEY = process.env.N2YO_API_KEY;
const N2YO_BASE_URL = process.env.N2YO_BASE_URL;

```

Another critical step to getting the data from n2yo is a valid base URI (the start of the URI) and an API key (a unique identifier that is used to authenticate my requests, handling the whom what and why).

Data processing and endpoints.

Get endpoints:

// GET - Get all satellites with real-time positions

```
app.get("/api/satellites", async (req, res) => {
```

```
  try{
```

```
    // Get user location from query parameters
```

```
    const { lat, lng, location } = req.query;
```

```
    const userLat = lat ? parseFloat(lat) : 51.6214;
```

```
    const userLng = lng ? parseFloat(lng) : -3.9436;
```

```
    const satellitesWithPositions = [];
```

```
    // Iterate through all satellites via name and id from SATELLITES object
```

```
    for (const sat of SATELLITES) {
```

```
      console.log(
```

```
        `Fetching data for ${sat.name} (ID: ${sat.id}) at ${userLat},${userLng}...`,
```

```
      );
```

```
      const position = await getSatellitePosition(sat.id, userLat, userLng);
```

```
      // Create a standard object to store incoming satellite data with
```

```
      // Optional chaining '?' to safe access lat and long and provide defaults if not set
```

```
      // Processed data is then pushed to satellitesWithPositions array
```

```
      const satelliteData = {
```

```
        id: sat.id,
```

```
        name: sat.name,
```

```
        latitude: position?.latitude || 0,
```

```
        longitude: position?.longitude || 0,
```

```
        altitude: position?.altitude || 0,
```

```
        visible: position?.visible || false,
```

```
      // If position is not null Api call succesful else failed
```

```

        dataAvailable: position !== null,
    };

    // Push the standard object onto satellitesWithPositions array
    satellitesWithPositions.push(satelliteData);

    console.log(` ${sat.name}: ${position ? colors.green + "Success" + colors.reset :
    colors.red + "Failed" + colors.reset}`);
}

res.json(satellitesWithPositions);
} catch (error) {
    console.error(colors.red + "Error fetching satellites:", error + colors.res).
    res.status(500).json({ error: "Failed to fetch satellite data" });
}
});

```

```

// GET - Get all observations
app.get("/api/observations", (req, res) => {
    // access helper function
    const data = readData();
    res.json(data.observations);
});

```

These endpoints retrieve all the satellite data available from n2yo.com and all the planned observations made from the application respectively.

Getting all the satellite data creates a fresh satellite data object that is then pushed onto an array:

```

const position = await getSatellitePosition(sat.id, userLat, userLng);

// Create a standard object to store incoming satellite data with

// Optional chaining '?' to safe access lat and long and provide defaults if not set

// Processed data is then pushed to satellitesWithPositions array

const satelliteData = {

```

```
id: sat.id,  
name: sat.name,  
latitude: position?.latitude || 0,  
longitude: position?.longitude || 0,  
altitude: position?.altitude || 0,  
visible: position?.visible || false,  
// If position is not null Api call succesful else failed  
dataAvailable: position !== null,  
};  
res.json(satellitesWithPositions);
```

A response is then returned via json format.

Note default values are available should any part of the call fail allowing for graceful handling.

Helper functions:

```
// Helper functions (read from or write to files)
const readData = () => {
  const data = fs.readFileSync(DATA_FILE, "utf8");
  return JSON.parse(data);
};
```

Simple file to javascript converter that loads stored observations from 'disk' into memory.

return JSON.parse(data);

Transforms the string into a usable JavaScript object

Which returns;

```
{
  observations: [
    {
      id: 1234567890,
      satelliteId: 25544,
      satelliteName: "ISS (International Space Station)",
      status: "planned",
      createdAt: "2024-01-15T10:30:00.000Z"
    }
  ]
}
```

Used by:

- GET /api/observations
- to list all observations
- POST /api/observations
- to add new observations
- PATCH /api/observations/:id
- to update observations

- DELETE /api/observations/:id

- to delete observations

```
const writeData = (data) => {  
  fs.writeFileSync(DATA_FILE, JSON.stringify(data, null, 2));  
};
```

Converts JavaScript object to JSON string

data - JavaScript object (like {observations: [...]})

null - Replacer function

2 - Indentation spaces for pretty formatting

Without helper function:

```
{"observations":[{"id":123,"name":"ISS","status":"planned"}]}
```

With helper function:

```
{  
  "observations": [  
    {  
      "id": 123,  
      "name": "ISS",  
      "status": "planned"  
    }  
  ]  
}
```

In memory storage

```
// file system makes creation and writing to my in memory file easier  
const fs = require("fs");
```

```
// Initialize data file if it doesn't exist (stops the app crashing)  
// Json to str array!  
// Yes this is a synchronous file operation so no manual closing as .writeFileSync does that for me (Stop adding manual closing me)  
if (!fs.existsSync(DATA_FILE)) {  
  fs.writeFileSync(DATA_FILE, JSON.stringify({ observations: [] }));  
}
```

Front end

app.js

Handles the front-end behaviour. Such DOM manipulation referencing the elements defined in index.html to control content displayed and perform operations to control backend data. Also handles data flow by fetching data from the API endpoints and then populates HTML content with said data.

Event handling is also facilitated here by handling event listening for clicks and form submission while handling dynamic elements generating html strings and injecting them into containers.

DOM manipulation

Elements by ID

```
const satellitesList = document.getElementById("satellites-list");
const observationsList = document.getElementById("observations-list");
const observationForm = document.getElementById("observation-form");
const satelliteSelect = document.getElementById("satellite-select");
const refreshButton = document.getElementById("refresh-satellites");
```

Event listeners

```
document.addEventListener("DOMContentLoaded", () => {
  loadSatellites();
  loadObservations();

  refreshButton.addEventListener("click", () => {
    loadSatellites();
  });
});
```

Button state manipulation

```
refreshButton.disabled = true;
refreshButton.textContent = "🔄 Loading...";
satellitesList.classList.add("loading");

// Later...
refreshButton.disabled = false;
refreshButton.textContent = "🔄 Refresh Positions";
satellitesList.classList.remove("loading");
```

Page content updates via innerHTML

```
// Satellites display
satellitesList.innerHTML = satellites.map(...).join("");

// Satellite dropdown
satelliteSelect.innerHTML = '<option value="">Select a satellite...</option>' + satellites.map(...).join("");

// Observations display
observationsList.innerHTML = "<p>No observations planned yet.</p>";
observationsList.innerHTML = observations.map(...).join("");

// Error messages
satellitesList.innerHTML = "<p>Error loading satellites</p>";
observationsList.innerHTML = "<p>Error loading observations</p>";
```

Form data access

```
const satelliteId = satelliteSelect.value;
const satelliteName = satelliteSelect.options[satelliteSelect.selectedIndex].dataset.name;
const plannedTime = document.getElementById("planned-time").value;
const userLocation = document.getElementById("user-location").value;
```

Form reset for new data

```
observationForm.reset(); // Clear the form fields
```

Overview of DOM manipulation

- Getting elements: `document.getElementById()`
- Setting content: `.innerHTML` = Changing properties: `.disabled`, `.textContent`, `.value`
- CSS classes: `.classList.add()`, `.classList.remove()`
- Event handling: `.addEventListener()`
- Form operations: `.reset()`

Index.html

Acts as the foundation for the html layout and structure. Creates DOM elements with specific ID's that app.js can target.

- satellites-list - Container for satellite data
- observations-list - Container for observation data
- observation-form - Form for creating observations
- satellite-select - Dropdown for satellite selection
- refresh-satellites - Button to refresh data

Links: Connects to both CSS (<link rel="stylesheet" href="style.css">) and JavaScript (<script src="app.js">)

style.css

Handles the appearance of the front end, such as the visual design styling all HTML elements with colours spacing and the overall layout queuing in state styling to provide visual feedback to make the website more intuitive via visible, not visible and loading.

Individual styling of components such as buttons, forms and status indicators.

Data flow:

1. **HTML provides the structure and elements ID's**
2. **JavaScript finds elements by ID and populates them with API data**
3. **CSS styles the dynamically generated content**

Key Integration Points:

- JavaScript uses

`document.getElementById()`

to target HTML elements

- CSS classes are applied dynamically by JavaScript (e.g.,

`classList.add("loading")`

)

- Form elements in HTML trigger JavaScript event handlers
- JavaScript generates HTML strings that inherit CSS styling

// Instead of this every time:

```
document.getElementById("satellites-list").innerHTML = "...";
```

// store the reference once:

```
const satellitesList = document.getElementById("satellites-list");
```

// use it multiple times:

```
satellitesList.innerHTML = "...";
```

```

// seems redundant but if I change where the API is I will be grateful (i.e. v1/api/.....) from api/.....
const API_BASE = "/api";

// DOM elements
// Creates references to html elements on the DOM for easier manipulation
// Lets me use 'satellitesList.innerHTML = ...' as opposed to 'document.getElementById("satellites-list").innerHTML = ...'
const satellitesList = document.getElementById("satellites-list");
const observationsList = document.getElementById("observations-list");
const observationForm = document.getElementById("observation-form");
const satelliteSelect = document.getElementById("satellite-select");
const refreshButton = document.getElementById("refresh-satellites");

// Load data on page load
document.addEventListener("DOMContentLoaded", () => {
  loadSatellites();
  loadObservations();

  // Add refresh button event listener
  refreshButton.addEventListener("click", () => {
    loadSatellites();
  });
});

```

```

// Load data on page load
document.addEventListener("DOMContentLoaded", () => {
  loadSatellites();
  loadObservations();

  // Add refresh button event listener
  refreshButton.addEventListener("click", () => {
    loadSatellites();
  });
});

```

```

document.addEventListener("DOMContentLoaded", () => {

  loadSatellites(); // Populates satellite data from /api/satellites

  loadObservations(); // Populates observation data from /api/observations

  refreshButton.addEventListener("click", () => {

    loadSatellites(); // Only reloads satellite data (positions update)

  });

```

Allowing critical position and altitude data from /api/satellites to be updated without refreshing the entire DOM.

Key Points:

- No page refresh - Uses AJAX/fetch to update content dynamically
- Initial load - Gets both satellites and observations when page first loads

- Selective refresh - Button only refreshes satellites (since their positions change), not observations
- Stays on same page - User doesn't lose their place or form data

This is a Single Page Application (SPA) pattern.

Fetch and map satellite data into a Json array and pass that to the respective display and populate functions. While also handling UI updates and then resetting the UI to avoid confusion, while also handling errors thrown:

```
// Fetch and display satellites
async function loadSatellites() {
  try {
    refreshButton.disabled = true;
    refreshButton.textContent = "🔄 Loading...";
    satellitesList.classList.add("loading");
    // fetch and then populate satellites via http from endpoint /api/satellites
    const response = await fetch(`${API_BASE}/satellites`);
    // convert that response to json
    const satellites = await response.json();

    displaySatellites(satellites);
    populateSatelliteSelect(satellites);

    refreshButton.disabled = false;
    refreshButton.textContent = "🔄 Refresh Positions";
    satellitesList.classList.remove("loading");
  } catch (error) {
    console.error("Error loading satellites:", error);
    satellitesList.innerHTML = "<p>Error loading satellites</p>";

    refreshButton.disabled = false;
    refreshButton.textContent = "🔄 Refresh Positions";
    satellitesList.classList.remove("loading");
  }
}
```

Error handling;

```
catch (error) {

  console.error("Error loading satellites:", error);

  satellitesList.innerHTML = "<p>Error loading satellites</p>";
```

if any piece of code throws an error, replace the list of satellites (from the satellite-list referenced by its element id).



Display and populate satellite data:

```
// Display satellites
/*
- Takes my satellite array and maps it to a html string
- then sets the innerHTML of the satellitesList to that string
- this is done by mapping each satellite to a html string
- then joining all the strings together
- then setting the innerHTML of the satellitesList to that string

(Takes satellite array → satellites

Maps to HTML strings → .map((sat) =>

Joins strings together → .join("")

Sets innerHTML → satellitesList.innerHTML = ...)

*/
function displaySatellites(satellites) {
  satellitesList.innerHTML = satellites
    .map(
      (sat) => `
        <div class="satellite-card">
          <div class="satellite-info">
            <div class="satellite-name">${sat.name}</div>
            <div class="satellite-details">
              ID: ${sat.id} | Alt: ${sat.altitude}km |
              ${
                sat.dataAvailable
                  ? `<span class="${sat.visible ? "visible" : "not-visible"}">
                    | ${sat.visible ? "🟢 Visible" : "🔴 Not Visible"}
                  </span>`
                  : `<span class="not-visible">⚠️ Data Unavailable</span>`
              }
            </div>
          </div>
        </div>
      `
    )
    .join("");
  // turns an array of strings into one long string which is easier to handle
  // .join("");
}
```

Satellites.map() creates an array where each satellite becomes a HTML string, where .join() can combine all those HTML strings into one single string with no separators, assigning this string to satellitesList.innerHTML as this is used to update what the user sees on the page.

```
// Populate satellite select dropdown
/*
- This function produces a default option to ensure users can't provide a null answer
- maps that sat array to a html string much like the display function :D
*/
function populateSatelliteSelect(satellites) {
  satelliteSelect.innerHTML =
    '<option value="">Select a satellite...</option>' +
    satellites
      .map(
        (sat) => `
        <option value="${sat.id}" data-name="${sat.name}">${sat.name}</option>
        `,
      )
      .join("");
}

```

Observation pipeline

Initial loading:

loadObservations() → GET /api/observations → displayObservations()

- Fetches existing observations on page load
- Displays them as cards

```
const observationsList = document.getElementById("observations-list");
const observationForm = document.getElementById("observation-form");

```

```
// Fetch and display observations
async function loadObservations() {
  try {
    // Similar to the satellite fetch, gets a http response and converts that on fetch to a json payload
    const response = await fetch(`${API_BASE}/observations`);
    const observations = await response.json();

    displayObservations(observations);
  } catch (error) {
    console.error("Error loading observations:", error);
    observationsList.innerHTML = "<p>Error loading observations</p>";
  }
}

```

Creating a new observation

Form Submit → POST /api/observations → loadObservations() → Refresh Display

- User fills form (satellite, time, location)
- Validates required fields
- Sends POST request
- Refreshes observation list

```

// Handle form submission
observationForm.addEventListener("submit", async (e) => {
  e.preventDefault();

  const satelliteId = satelliteSelect.value;
  const satelliteName =
    satelliteSelect.options[satelliteSelect.selectedIndex].dataset.name;
  const plannedTime = document.getElementById("planned-time").value;
  const userLocation = document.getElementById("user-location").value;

  if (!satelliteId || !plannedTime) {
    alert("Please select a satellite and planned time");
    return;
  }

  try {
    const response = await fetch(`${API_BASE}/observations`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        satelliteId: parseInt(satelliteId),
        satelliteName,
        plannedTime,
        userLocation: userLocation || "Not specified",
      }),
    });

    if (response.ok) {
      observationForm.reset();
      loadObservations();
      alert("Observation planned successfully!");
    } else {
      const error = await response.json();
      alert(`Error: ${error.error}`);
    }
  } catch (error) {
    console.error("Error creating observation:", error);
    alert("Error creating observation");
  }
});

```

```

const satelliteId = satelliteSelect.value;
const satelliteName =
  satelliteSelect.options[satelliteSelect.selectedIndex].dataset.name;
const plannedTime = document.getElementById("planned-time").value;
const userLocation = document.getElementById("user-location").value;

if (!satelliteId || !plannedTime) {
  alert("Please select a satellite and planned time");
  return;
}

```

The above logic handles user input and validates it ensuring that the satelliteID from the dropdown menu, as well as the planned time fields are populated i.e. if not a satelliteID or not a planned throw an alert.

Update observation status

Button Click → Prompt for Notes → PATCH /api/observations/{id} → loadObservations()

- "Observed" or "Missed" buttons
- Prompts user for notes/reason
- Updates status via PATCH
- Refreshes list

```

observationsList.innerHTML = observations
    .map(
      (obs) => `
        <div class="observation-card">
          <div class="observation-info">
            <div class="observation-satellite">${obs.satelliteName}</div>
            <div class="observation-details">
              Planned: ${new Date(obs.plannedTime).toLocaleString()} |
              Location: ${obs.userLocation} |
              <span class="status-${obs.status}">Status: ${obs.status.toUpperCase()}</span>
              ${obs.notes ? `<br>Notes: ${obs.notes}` : ""}
            </div>
          </div>
          <div class="observation-actions">
            ${
              obs.status === "planned"
                ? `
                  <button class="btn-small btn-success" onclick="updateObservationStatus(${obs.id}, 'observed')">
                    Observed
                  </button>
                  <button class="btn-small btn-danger" onclick="updateObservationStatus(${obs.id}, 'missed')">
                    Missed
                  </button>
                `
                : ""
            }
            <button class="btn-small btn-delete" onclick="deleteObservation(${obs.id})">
              Delete
            </button>
          </div>
        </div>
      `
    )
    .join("");

```

User prompted

```

try {
  const response = await fetch(`${API_BASE}/observations/${id}`, {
    method: "PATCH",
    headers: {
      "Content-Type": "application/json",
    },

```

Patch to update the status,

```

    // turn a javascript object INTO a json string for http transmission (allows the payload or update to a status to be sent via http to my patch endpoint)
    // cast that json type to a http string
    body: JSON.stringify({ status, notes }),
  });

  // step three: actually process the patch request
  // This is done via TWO levels of error handling

  if (response.ok) { // (Success path http 200)
    loadObservations(); // call the helper function to refresh the data in observations
    alert('Observation marked as ${status}'); // Show the success message
  } else {
    const error = await response.json(); // parse the server error message
    alert('Error: ${error.error}'); // show the error
  }
} catch (error) {
  console.error("Error updating observation:", error);
  alert("Error updating observation");
}

```

Handle and implement the patch request as well as error handling (await response and parse an error to the server if there is one then display it).

As well as refreshing the data in the observations list via the helper function:

```
// Fetch and display observations
async function loadObservations() {
  try {
    // Similar to the satellite fetch, gets a http response and converts that on fetch to a json payload
    const response = await fetch(`${API_BASE}/observations`);
    const observations = await response.json();

    displayObservations(observations);
  } catch (error) {
    console.error("Error loading observations:", error);
    observationsList.innerHTML = "<p>Error loading observations</p>";
  }
}

// Display observations in the UI
function displayObservations(observations) {
  if (observations.length === 0) {
    observationsList.innerHTML = "<p>No observations planned yet.</p>";
    return;
  }
}
```

The

updateObservationStatus()

function handles:

- Button click triggers function call
- Conditional prompt based on status
- PATCH request to API
- UI refresh on success
- Error handling

Deleting an observation

Delete Button → Confirm Dialog → DELETE /api/observations/{id} → loadObservations()

- Confirmation prompt
- DELETE request
- Refreshes list

```
// Delete observation
async function deleteObservation(id) {
  if (!confirm("Are you sure you want to delete this observation?")) {
    return;
  }

  try {
    const response = await fetch(`${API_BASE}/observations/${id}`, {
      method: "DELETE",
    });

    if (response.ok) {
      loadObservations();
      alert("Observation deleted successfully");
    } else {
      const error = await response.json();
      alert(`Error: ${error.error}`);
    }
  } catch (error) {
    console.error("Error deleting observation:", error);
    alert("Error deleting observation");
  }
}
}
```

Technical overview data flow

Data Flow Pattern:

Action → API Call → Refresh Display

Every operation follows the same pattern:

1. User interaction
2. API request
3. loadObservations()

to refresh the UI

Key Functions:

- loadObservations()
- Fetches and displays all observations
 - displayObservations()
- Renders observation cards with action buttons
 - Form handler - Creates new observations

- `updateObservationStatus()`
- Changes status (planned → observed/missed)

- `deleteObservation()`
- Removes observations

`const { id } = req.params;`

Is used where ID is passed as an argument to dynamically affect the observation targeted, this as the frontend takes

`DELETE/api/observations/:123456`

Express matches the route pattern

`/api/observations/:id`

Express then uses `req.params;` to capture the id.

Allowing one route for delete and patch to handle ALL observations.

Testing

Case 1: GET all satellites

 GET Satellites Test	
Endpoint:	GET /api/satellites
Query:	?lat=51.6214&lng=-3.9436
Expected:	200 OK
Returns:	Array of 5 satellites with positions and visibility

Case 2: POST new observation to in memory file system

 Create Observation Test	
Endpoint:	POST /api/observations
Body:	{
	"satelliteld": 25544,
	"satelliteName": "ISS",
	"plannedTime": "2024-01-15T20:00:00Z"
	}
Expected:	201 Created

Case 3: PATCH (partial update) existing observation inside file system storage

✏ Update Status Test	
Endpoint:	PATCH /api/observations/123
Body:	{
	"status": "observed",
	"notes": "Clear sky, great view!"
	}
Expected:	200 OK

Case 4: DELETE an observation

🗑 Delete Observation Test	
Endpoint:	DELETE /api/observations/123
Expected:	200 OK
Returns:	Success message

Case 5: Error handling

✖ Invalid Status Test	
Endpoint:	PATCH /api/observations/123
Body:	{ "status": "invalid" }
Expected:	400 Bad Request
Returns:	"Invalid status" error

Documentation and setup guide n2yo tracker

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

96/100

Generic Assessment Criteria	Marks available	Marks awarded
1. Engagement with Literature Skills		
2. Knowledge and Understanding Skills		
3. Cognitive and Intellectual Skills	Assignment 2: 40	
Design and documentation		37
Test documentation and testing		
Nicely written and presented, comprehensive documentation of endpoints, frontend and instructions.		

Very good understanding demonstrated of the functionality of the SPA and API. Good use of code snippets. Good evidence of testing serverside, some testing UI side would have been advantageous.		
4. Practical Application Skills	Assignment 2: 50	
AJAX implementation REST API Good use truthies, async, helpers and fetch. Good clean and organised code and folder structure.		48
5. Transferable Skills for Life and Professional Practice	Assignment 2: 10	
Presentation Good clear understanding of the code and what it does. Good demonstration of truthies.		10

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32
