Ollscoil
Teicneolaíochta
an Atlantaigh

Atlantic
Technological
University

Satellite Follower App

Project Engineering

Year 4

Dan Koskiranta

Bachelor of Engineering (Honours) in Software and

Electronic Engineering

Atlantic Technological University

2024/2025

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Atlantic Technological University.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

_____Dan Koskiranta_____

Acknowledgements

I would like to thank my project supervisors, Pat Hurney and Paul Lennon, for their assistance and guidance throughout the duration of my final year project.

Additionally, I wish to thank my Project Engineering coordinators, Ben Kinsella, Niall O'Keeffe, Michelle Lynch, and Brian O'Shea for their feedback and support on my project work throughout the year.

Table of Contents

1 Summary

The goal of this project was to develop a web-based application that tracks and displays real-time satellite information over Galway, Ireland. With increasing interest in satellite navigation, space technology, and real-time geospatial data, the project aims to make satellite tracking more accessible and informative for educational or research purposes. It integrates data from two primary sources: GNSS (Global Navigation Satellite System) data received via UART from an L86-M33 GNSS receiver to an ESP32 microcontroller, and public satellite data from CelesTrak. GNSS data includes satellite ID, type, and country of origin, while CelesTrak data provides satellite name, ID, latitude, and longitude. The combined data is visualized on a user-friendly React.js frontend.

The project focuses on localized satellite tracking and includes backend data processing but excludes manual implementation of complex orbital mechanics or global tracking, with Satellite.js handling orbital calculations. Key features include real-time data visualization, efficient backend processing, and an accessible user interface.

In the first semester, the focus was on hardware setup, GNSS signal reception, and backend development. The second semester involved parsing NMEA sentences, extracting satellite metadata, integrating CelesTrak TLE data using Satellite.js, and displaying the combined data on the frontend.

Technologies used include hardware (ESP32 and L86-M33), embedded code (C/C++ via Arduino IDE), communication protocol (UART), backend (Node.js), orbital calculations (Satellite.js), and frontend (React.js).

The project successfully meets its core objectives, providing real-time satellite tracking over Galway. While the application is functionally complete, a more interactive layout—such as a map-based interface—could further improve the user experience.

2 Poster

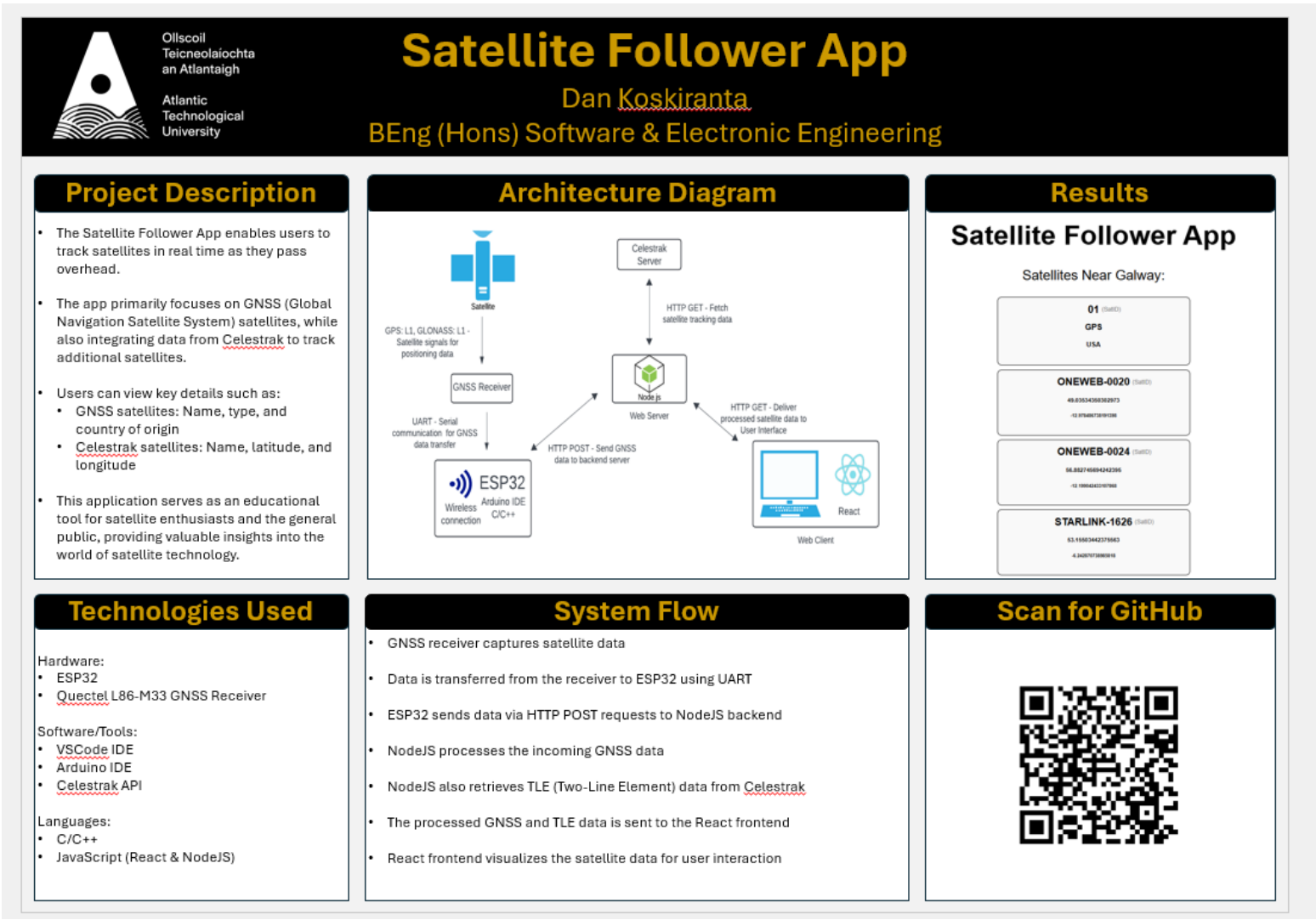

Figure 2.1 - Poster

3 Introduction

The Satellite Follower App is a web-based application designed to track and display real-time satellite information to users. The application integrates data from GNSS hardware and public satellite data from CelesTrak, brought together through full-stack development. The app aims to provide accessible satellite insights for satellite hobbyists and the general public with an interest in space technology.

The motivation behind this project stems from my personal interest in space technology, specifically the areas that combine software and electronic engineering. I have been inspired by the success of innovative startups such as MBRYONICS and ICEYE, which demonstrate how small teams can make significant advancements in the satellite industry. I wanted to gain a deeper understanding of how satellites operate, how they communicate with receivers on Earth, and how GNSS systems determine location using satellite signals. Taking this hands-on project allowed me to explore these technologies across hardware setup, data communication, and real-time data processing.

With satellite technology playing an increasingly important role in areas like navigation, climate monitoring, and telecommunications, the space sector is expanding rapidly. Gaining practical experience in this field is not only personally fulfilling but also valuable for future career development.

The project focuses on localized satellite tracking over Galway, Ireland. It covers GNSS signal reception, NMEA sentence parsing, retrieval of Two-Line Element (TLE) data from CelesTrak, backend processing, and frontend visualization. This project does not cover manual implementation of advanced orbital mechanics or global tracking; instead, orbital calculations for the CelesTrak data are handled using the Satellite.js library.

4 Background

4.1 Introduction to Satellite Technology

Satellite technology plays a crucial role in our society, powering everything from communication systems to navigation and weather forecasting. Satellites are used to monitor Earth, relay signals for communication, and provide precise positioning information via systems such as GNSS. GNSS enables devices to determine their location anywhere on Earth by receiving signals from multiple satellites.

4.2 Global Navigation Satellite System (GNSS)

GNSS is a satellite-based navigation system that provides accurate location data globally. It is composed of several independent constellations operated by different countries:

- GPS (Global Positioning System) - United States
- GLONASS (GLObal NAvigation Satellite System) - Russia
- Galileo – Europe
- BeiDou – China
- QZSS (Quasi-Zenith Satellite System) – Japan
- IRNSS (Indian Regional Navigation Satellite System) – India

Among these, GPS, GLONASS, Galileo, and BeiDou offer global coverage. QZSS and IRNSS are both regional systems primarily covering Asia, with satellites placed in geosynchronous or geostationary orbits. All these systems use a constellation of satellites orbiting Earth to provide location information to receivers. GNSS works on trilateration, where a receiver calculates its position by measuring the distance from at least three or four satellites. The satellites broadcast signals that include timing information, allowing the receiver to compute the distance based on the time delay between transmission and reception:
*Distance = Speed of Light * Time Difference*
With distances from multiple satellites, the receiver can accurately pinpoint its location on Earth [1].

4.3 NMEA Sentences

GNSS receivers such as the L86-M33 transmit data in the form of NMEA sentences. NMEA (National Marine Electronics Association) is a standard for formatting data from marine electronics, including GNSS systems. The NMEA sentences contain information such as satellite ID, satellite type, and positioning data [2]. In this project, the focus was on parsing the NMEA sentences to extract satellite metadata like satellite PRN (Pseudo-Random Noise) code and type (e.g., GPS, GLONASS). This PRN code serves as the satellites ID number and helps to identify which satellites are currently passing overhead [3].

4.4 Two-Line Element Data

Tracking satellites in orbit requires knowledge of their positions and trajectories. Two-Line Element (TLE) data is used to describe the orbital parameters of a satellite at a specific time. This data is widely available from sources such as CelesTrak, which provides publicly accessible satellite information. Each TLE set contains values such as the satellite's inclination, orbital period, and position relative to Earth. These parameters are used to calculate and predict a satellite's location over time [4]. In this project, TLE data from CelesTrak was retrieved using an API and processed using the Satellite.js library to estimate satellite positions in real-time over Galway, Ireland.

4.5 Supporting Technologies and Tools

This project relied on several key technologies and libraries to handle satellite tracking, data communication, and visualization.
Satellite.js Library: Satellite.js is a JavaScript library used for satellite orbit calculations. It processes TLE data to determine a satellite's position and velocity at any given time [5]. This library simplifies orbital prediction by eliminating the need to manually implement complex orbital mechanics. In this project, Satellite.js was used on the backend to calculate satellite positions over Galway, Ireland, using real-time TLE data retrieved from CelesTrak.

Frontend and Backend Technologies: The application was developed using modern full-stack JavaScript technologies. The frontend was built using React.js which is a widely used library for building dynamic user interfaces. The backend was implemented using Node.js, which handles server-side operations such as communicating with the ESP32 microcontroller, processing data from the GNSS receiver and CelesTrak, and performing orbital calculations using Satellite.js.

UART Communication Protocol: The L86-M33 GNSS receiver and ESP32 microcontroller communicated using UART, a serial protocol that allows asynchronous data transfer between devices [6]. In this project, the GNSS receiver transmitted NMEA sentences via UART to the ESP32, which parsed the data and forwarded it to the backend for further processing and display on the frontend.

4.6 Challenges

Several challenges arose during the development of this project, particularly around hardware setup and system integration. One of the earliest obstacles was the difficulty in demonstrating the project indoors due to limited GNSS signal reception. GNSS receivers require a clear view of the sky to receive satellite signals, which made indoor testing of the system less reliable and required workarounds such as relocating the setup outdoors.

Another challenge involved configuring the L86-M33 GNSS receiver with the ESP32 microcontroller. Initially, no NMEA messages were being printed, which made it unclear whether the receiver was functioning. After troubleshooting, it was discovered that the V_BCKP

pin had not been connected to a valid power source. This pin is required to power the GNSS module's RTC (Real-Time Clock) domain, which is necessary for proper startup and for outputting NMEA messages [7].

Finally, parsing the NMEA sentences to extract relevant satellite metadata, particularly the satellite ID (PRN code), proved to be another key challenge. NMEA sentences are structured text strings that require careful parsing to isolate the necessary fields. Understanding the sentence format and implementing reliable string parsing logic was essential for identifying satellites and displaying accurate information.

5 Project Architecture

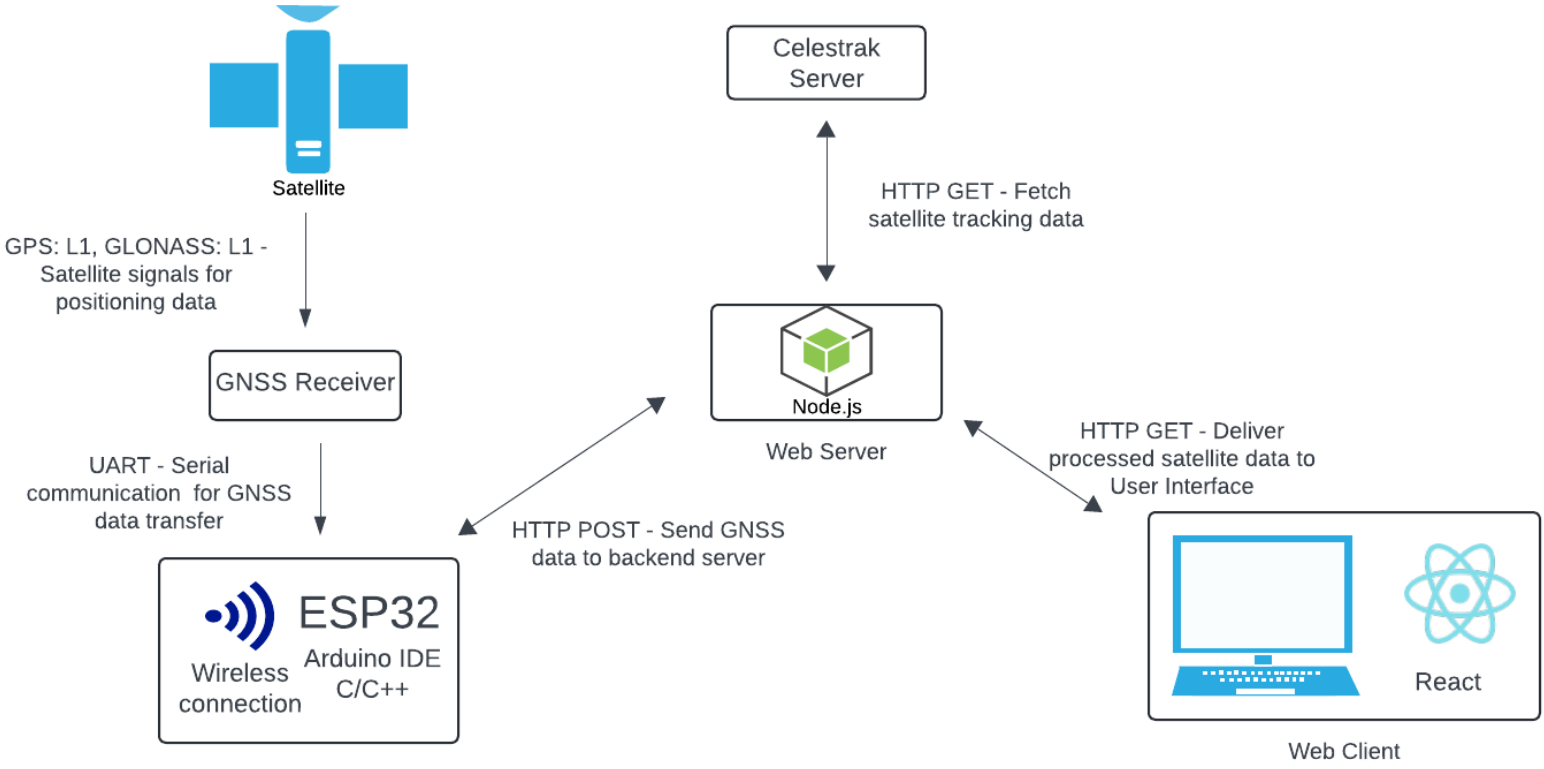

Figure 5.1 – Satellite Follower App Architecture Diagram

6 Project Plan

For planning and managing my project, I used a project management tool called Jira, where I followed the Scrum framework to structure and track my progress. I began by creating an initial timeline that outlined all the major tasks. These were then broken down into subtasks and organized into sprints. Each sprint typically lasted one week and consisted of around three tasks which allowed me to update my project progress on a weekly basis within the tool. At the end of each sprint, I generated a burndown chart to visualize task completion and track whether I was staying on schedule. These charts helped me to identify delays and adjust my planning accordingly.

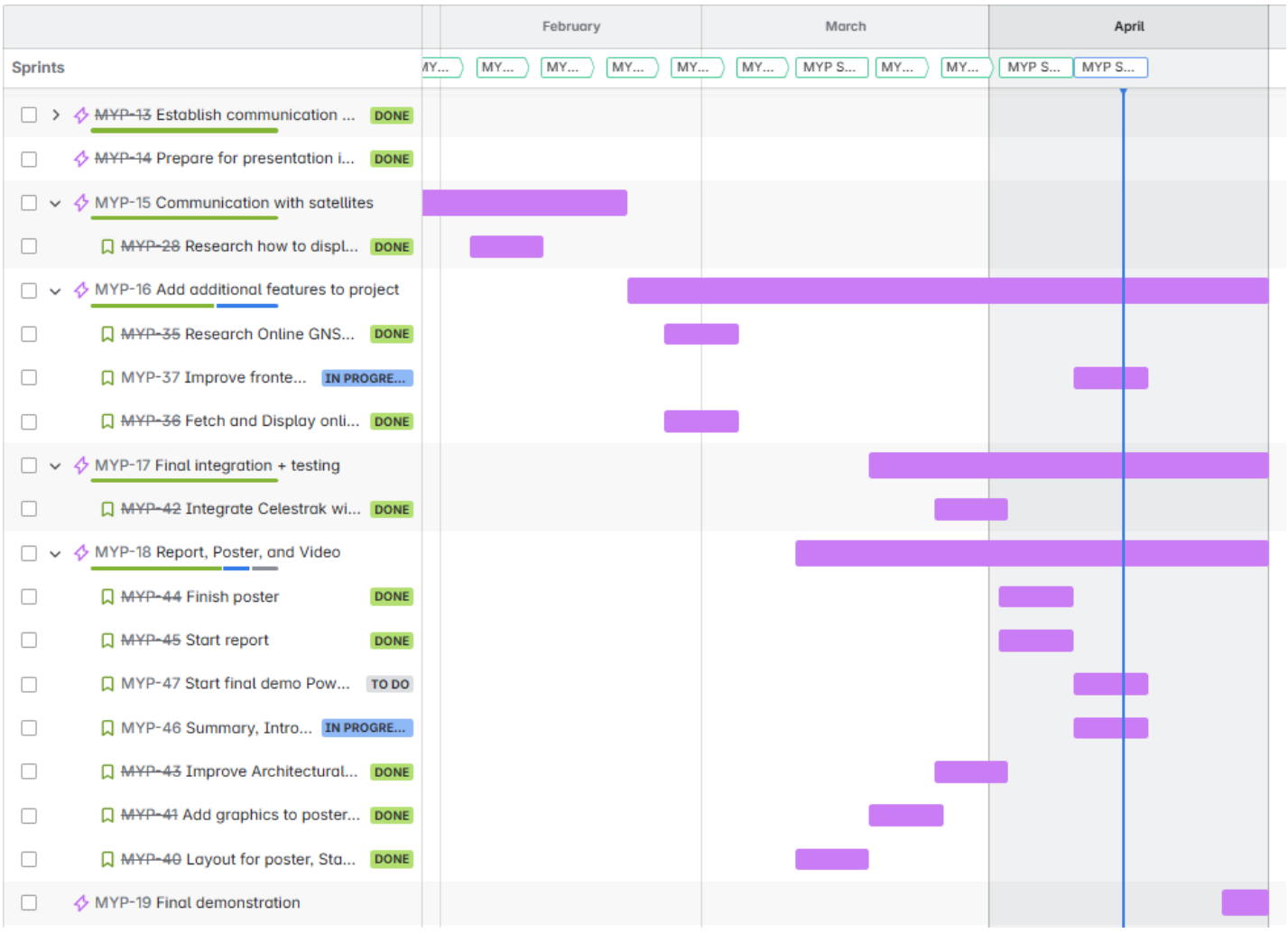

Figure 6.1 – Project Timeline showing task progress

In addition to Jira, I maintained weekly logs on OneNote to track detailed progress, challenges, and completed tasks. These logs helped me to reflect on each sprint and make necessary adjustments to the project plan.

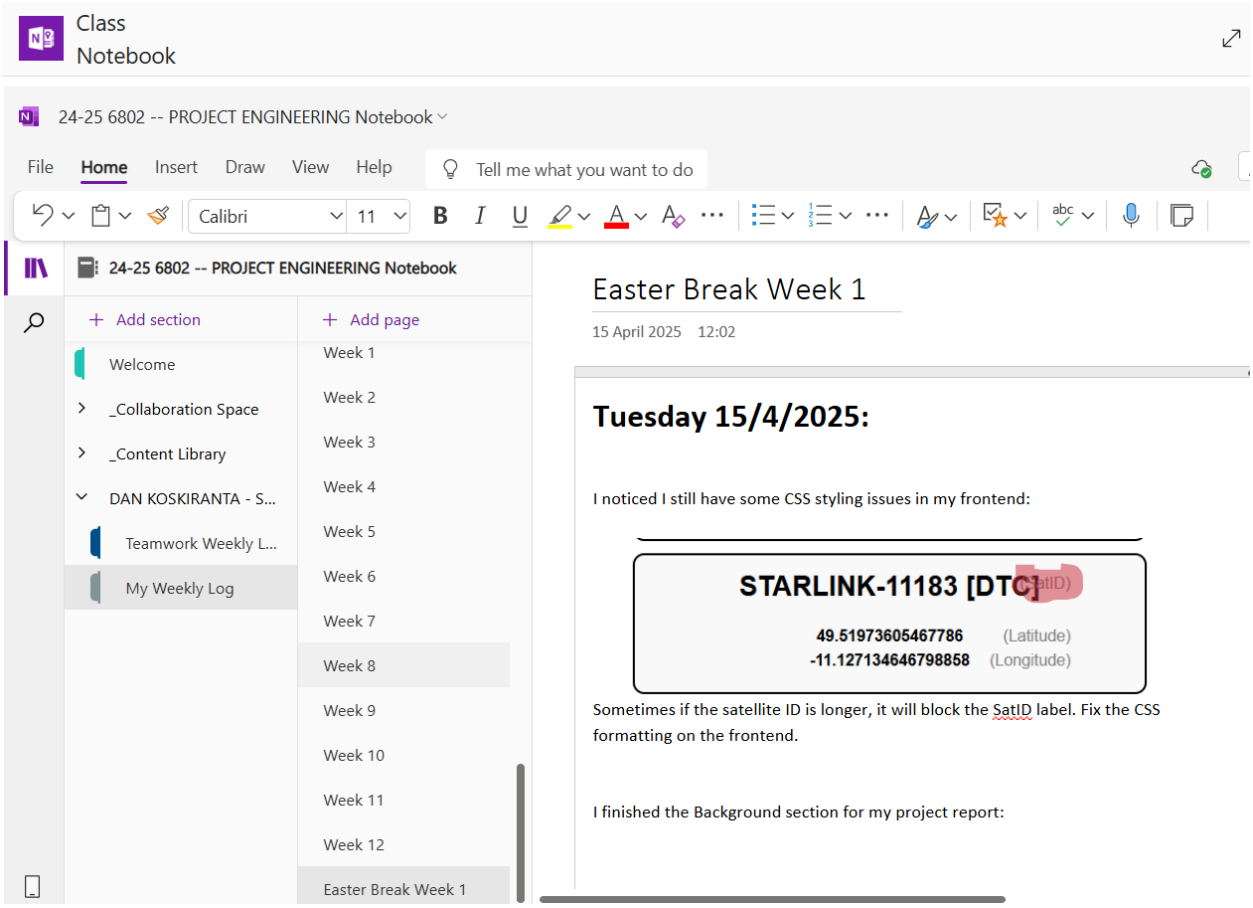

Figure 6.2 – Weekly project logs with task details and dates

7 System Design and Technologies

7.1 Hardware

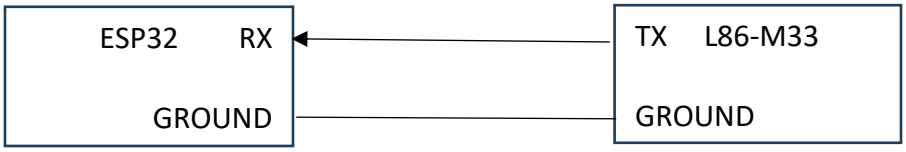

Figure 7.1 – ESP32 Microcontroller and GNSS Receiver Hardware Schematic

7.1.1 ESP32 Microcontroller

The development board used for the project was the ESPRESSIF ESP32-DevKitC V4 featuring the WROOM-32E module. The ESP32 was chosen for this project due to its built-in Wi-Fi capabilities, support for various communication protocols, and strong community support through open-source libraries. Specifically, its support for UART communication enabled reliable serial data transfer for receiving continuous NMEA data streams from the GNSS module [8].

In this project, the ESP32 played a central role in system communication. It established UART communication with the GNSS module via GPIO16, allowing it to receive real-time NMEA sentences streams. The board was programmed to parse $GPGSV and $GLGSV sentences, extract relevant information such as satellite PRN (Pseudo-Random Noise) codes, and send this data via HTTP POST requests to the backend server for further processing.

Programming of the ESP32 was done using the Arduino IDE, which provided a straightforward environment to write, compile, and upload code to the board. Libraries such as TinyGPS++, HardwareSerial, and WiFi.h were used to support GNSS data parsing and network communication. The ESP32 was connected to the local Wi-Fi network, allowing it to send data directly to the Node.js backend.

The ESP32 was powered through a standard USB connection, which provided 5V to the development board. This input was regulated down to 3.3V, supplying the necessary voltage for the ESP32's operation and ensuring compatibility with 3.3V logic-level communication used by peripheral devices such as the GNSS module [9]. This power setup made it easy to connect to a laptop during testing and deployment.

7.1.2 L86-M33 GNSS Receiver

The Quectel L86-M33 GNSS receiver was chosen for this project due to its affordability and availability. While it only supports the GPS and GLONASS constellations, it was an optimal choice given the budget constraints and limited availability of hardware components. Despite supporting fewer constellations, the L86-M33 provided reliable and accurate location data, meeting the project's requirements effectively.

The L86-M33 is a compact, high-performance GNSS receiver module manufactured by Quectel. It supports positioning using GPS and GLONASS satellite constellations, allowing for improved location accuracy and reliability by combining data from multiple satellite systems. The module features an internal patch antenna that is designed to receive signals in the L1 band – 1575.42 MHz for GPS and 1598.0625-1605.375 MHz for GLONASS [7].

The positioning data is based on a concept called trilateration, which states that at least three or four satellites are required for accurate positioning. Each satellite in orbit contains an atomic clock and broadcasts a signal embedded with a highly accurate timestamp. The receiver calculates its distance by measuring the time it takes from the satellites to reach the receiver using the formula:
Distance = Speed of Light * Time Difference.
By calculating distances from four satellites, the receiver can determine its precise location. These distances define spheres of possible locations around each satellite. The point at which at these spheres intersect determines the precise location of the receiver in three-dimensional space [1].

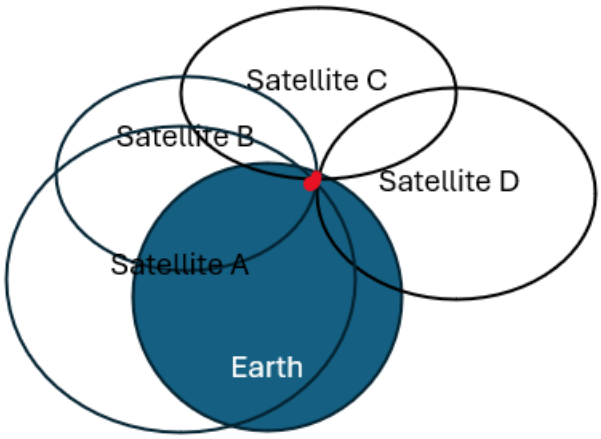

Figure 7.2 – Principle of Trilateration Using Four Satellites

The GNSS receiver can perform trilateration and provide accurate position information if it knows where the satellite is, the time when the signal was sent from the satellite, and the time the signal was received. However, several factors can affect the accuracy of this process. Environmental and technological challenges such as ionospheric disturbances, clock inaccuracies, and multipath signal reflection can introduce positioning errors. To mitigate this, the L86-M33 supports Satellite-Based Augmentation Systems (SBAS), which use a network of ground reference stations and geosynchronous satellites to estimate and correct errors caused by atmospheric interference and signal delays [10].

The L86-M33 GNSS receiver consists of several key components that enable it to accurately receive and process satellite signals. It includes a built-in patch antenna, which is responsible for receiving signals from GNSS satellites such as GPS and GLONASS [7].

The module contains a radio frequency (RF) Front-End, which performs two critical functions: it amplifies weak signals using a low-noise amplifier (LNA), and it filters out unwanted frequencies using a Surface Acoustic Wave (SAW) filter. This filtering ensures that only the relevant GNSS frequency bands – GPS L1 (1575.42 MHz) and GLONASS L1 (1598.0625–1605.375 MHz) – are passed through for further processing [7].

The baseband processor, which can be considered as a form of digital signal processor (DSP), handles the core GNSS signal processing. It performs correlation to match incoming signals to known satellite codes (each satellite uses a unique pseudo-random noise code), tracks signal phases to determine time delays, and uses trilateration to calculate the receiver's position, velocity, and time (PVT) [7].

The module communicates with external systems via a UART interface, operating at a default baud rate of 9600 bps. The processed GNSS data is transmitted in NMEA format to the ESP32 microcontroller for further use and display [7].

7.2 Software

The software component of the system is divided into three main parts: the embedded firmware running on the ESP32 microcontroller, a backend server built with Node.js, and a frontend web interface developed in React.js. The ESP32 collects GNSS data from the L86-M33 receiver and transmits satellite information to the backend via HTTP over Wi-Fi. The Node.js backend processes the data and serves it to the frontend. The React.js frontend provides a user-friendly interface to display the list of satellites currently visible over Galway, Ireland.

7.2.1 ESP32 Firmware (Arduino/C++)

The firmware running on the ESP32 was developed in C++ using the Arduino IDE. It is responsible for reading GNSS data from the L86-M33 module via UART, it parses specific NMEA sentences to extract satellite IDs, formats the data into JSON, and sends it to a remote server over Wi-Fi using an HTTP POST request.

To support these tasks, the program utilizes different libraries to manage connectivity, communication, and GNSS data parsing respectively. The WiFi.h library is used to establish Wi-Fi connectivity, while HTTPClient.h enables HTTP POST requests to the backend server [11]. The TinyGPS++.h library allows the parsing of NMEA data streams provided by the GNSS module [12]. Finally, the HardwareSerial.h is used to configure and manage the hardware UART communication channel between the ESP32 and the L86-M33 module [13].

The GNSS receiver outputs data in NMEA format, which is a standardized text-based protocol used by GNSS modules to transmit positioning and satellite data. An example NMEA sentence looks like this:
*$GPGSV, 3, 3, 11, 23, 08, 038, 32, 30, 06, 281, 25, 03, 03, 200, 40, 0*5B;*
Each sentence begins with a dollar sign, followed by an identifier like GPGSV (GPS) or GLGSV (GLONASS).

- The first number 3 represents the total number of GSV sentences in this group.
- Second number 3 represents the current sentence number (i.e., this is the third of three).
- 11 is the number of satellites visible
- The following values appear in groups of four and describe individual satellites
    - 23 is the PRN number.
    - 08 is the elevation angle (in degrees above the horizon).
    - 038 is the azimuth (the satellite's direction in degrees from true north).
    - 32 is the signal-to-noise ratio (SNR) indicating the strength and quality of the signal.

This pattern repeats for the next satellites in the sentence (e.g., PRN 30, 03). Each satellite is described using four consecutive values. The sentence ends with a checksum (*5B) which is used to verify that the data was received correctly without corruption [14].

In my code I am using the extractSatelliteID function to isolate and extract the satellite PRN codes from the NMEA sentences. As the sentence is a comma-separated string, the function begins by iterating through each character while counting the commas.

```
String extractSatelliteID(String nmeaSentence) {
  int commas = 0;
  bool getID = false;
  String satelliteIDs = "";
  bool firstSatelliteID = true;


  for (int i = 0; i < nmeaSentence.length(); i++) {
    char character = nmeaSentence[i];

    if (character == ',') {
      commas++;
```

Figure 7.3 - Iterating Through NMEA Sentence to Count Commas and Extract Satellite IDs

The satellite IDs are known to appear at predictable intervals; they start at the fourth comma and repeat every four fields thereafter. To detect those positions, the function uses the condition (commas >= 4 && (commas - 4) % 4 == 0). When this is true, it marks the next value as satellite ID and extracts it using substring() function.

```
      if (commas >= 4 && (commas - 4) % 4 == 0) {
        getID = true;
      }
      else {
        getID = false;
      }
    }
    else if (getID) {
      //Extract satellite ID
      int startIndex = i;
      int endIndex = nmeaSentence.indexOf(",", startIndex);
      if (endIndex == -1) {
        endIndex = nmeaSentence.length();
      }
      String satID = nmeaSentence.substring(startIndex, endIndex);
```

Figure 7.4 - Extracting Satellite ID Based on Comma Count and Delimiters in NMEA Sentence

The extractSatelliteID function includes safeguards to detect corrupted or malformed data. It checks if the extracted ID is longer than three characters, in which case it is invalid and the parsing stops. It also checks if the ID contains an *, in which case it is part of the checksum and it is not a valid satellite ID.

```
//if NMEA sentence is corrupted.
if(satID.length() > 3) {
  break;
}

//Ignore IDs with '*'
if (satID.indexOf('*') != -1) {
  break;
}
```

Figure 7.5 – Handling Corrupted or Invalid Satellite IDs in NMEA Sentence

Valid satellite IDs are concatenated into a single comma-separated string. The firstSatelliteID Boolean ensures the correct placement of commas between the different IDs. Once the parsing is complete, the function returns a list containing the list of satellite IDs, ready to be formatted as JSON and sent to the backend server.

```
      if (satID.length() > 0) {
        if (!firstSatelliteID) {
          satelliteIDs += ",";
        }
        satelliteIDs += satID;
        firstSatelliteID = false;
      }
      i = endIndex - 1;
    }
  }
  return satelliteIDs;
}
```

Figure 7.6 - Concatenating Valid Satellite IDs and Handling String Formatting

The core logic of the firmware is implemented in the loop() function, which continuously reads and processes data from the GNSS module. The firmware initializes two strings: sentence to store the full NMEA sentence, and satelliteList to build a JSON array of satellite objects.

Once a complete sentence is received, the program checks whether it begins with $GPGSV (GPS data) or $GLGSV (GLONASS data) using the startsWith() function. Each type is handled separately to properly tag the satellites with their respective GNSS system. The sentence is passed to the extractSatelliteID() function, which returns a comma-separated string of satellite IDs, and they are stored in the String GPSsatelliteId or GLONASSsatelliteId variables, depending on the sentence type.

```
if (sentence.startsWith("$GPGSV")) {
  String GPSsatelliteId = extractSatelliteID(sentence);
  if (GPSsatelliteId.length() > 0) {
    int firstIndex = 0, lastIndex = 0;
    while ((lastIndex = GPSsatelliteId.indexOf(",", firstIndex)) != -1) {                //
      String GPSid = GPSsatelliteId.substring(firstIndex, lastIndex);
      satelliteList += "{\"id\":\"" + GPSid + "\",\"type\":\"GPS\",\"country\":\"USA\"},";
      firstIndex = lastIndex + 1;
    }
  }
```

Figure 7.7- Extracting and Formatting GPS Satellite IDs into JSON

The program then uses string manipulation to iterate through the IDs, extract each one, and format it into a JSON object like this: *{"id":"03","type":"GPS","country":"USA"}*. These JSON objects are added to the satelliteList string. The same procedure is repeated for GLONASS satellites.

To ensure valid formatting, the last comma is removed from the satelliteList before the closing bracket is added. The result should look like this:
[{"id":"03","type":"GPS","country":"USA"},{"id":"12","type":"GLONASS","country":"Russia"}]
Once the JSON array is ready, it is passed to the sendSatellitesToServer() function, which sends the data to a remote server over Wi-Fi using an HTTP POST request. A delay of five seconds is added to avoid sending data too frequently, maintaining a balance between performance and network efficiency.

```
if (satelliteList.length() > 1) {
  satelliteList.remove(satelliteList.length() - 1);
  satelliteList += "]";
  //Serial.println(satelliteList);
  sendSatellitesToServer(satelliteList);
}
delay(5000);
```

Figure 7.8 - Finalizing and Sending Satellite Data in JSON Format

The sendSatellitesToServer function first checks whether the ESP32 is connected to Wi-Fi using the WiFi.status() function. If a connection is available, it initializes an HTTP connection to the specified server using the http.begin() and sets the request header to *"Content-Type: application/json"* to ensure the server understands the format of the incoming data.

The satellite data is wrapped into a single JSON object with the key "satellites", and the complete payload is constructed as a string. The payload is sent to the server using an HTTP POST request and the response code returned by the server is captured in httpStatusCode and printed to the serial monitor to provide feedback whether the transmission was successful.

```
void sendSatellitesToServer(const String& satelliteList) {
  if (WiFi.status() == WL_CONNECTED) {
    http.begin(serverURL);
    http.addHeader("Content-Type", "application/json");

    String payLoad = "{\"satellites\":" + satelliteList + "}";
    int httpStatusCode = http.POST(payLoad);

    if (httpStatusCode > 0) {
      Serial.println("Data sent: " + String(httpStatusCode));
    }
    else {
      Serial.println("Error sending data: " + String(httpStatusCode));
    }
    http.end();
  }
  else {
    Serial.println("WiFi not connected");
  }
}
```

7.9 – sendSatellitesToServer function

7.2.2 Backend Server (Node.js)

The backend server was implemented using Node.js, with the Express.js framework. Node.js a JavaScript runtime environment that allows JavaScript to be executed outside of a web browser, making it ideal for building fast and efficient backend services [15]. The Express.js is a web framework that handles web requests, sets up routes, and manages middleware [16]. The primary responsibilities of the backend are receiving data from the ESP32, fetching real-time satellite information from the CelesTrak API (Application Programming Interface), and to combine the two datasets into a single API response for the frontend.

The ESP32 sends satellite information via an HTTP POST request to the */api/sendMessage* endpoint. The backend extracts the satellites array from the request body and stores it in a global variable gnssSatellites.

```
//API endpoint to receive GNSS satellite data from ESP32
router.post('/api/sendMessage', function (req, res, next) {
  const { satellites } = req.body  // Extract 'satellites' array from request body

  if(satellites) {
    gnssSatellites = satellites //Update global GNSS satellite data
    console.log("Received GNSS satellite data:", gnssSatellites)
    res.status(200).json({ message: "GNSS data received successfully", satellites})
  } else {
    console.log("No GNSS data received")
    res.status(400).json({ error: "No GNSS data received" })
  }
});
```

Figure 7.10 - API Endpoint to Receive and Process GNSS Satellite Data

The backend has an asynchronous function that fetches the TLE data of active satellites from the CelesTrak API. TLE contains the orbital parameters of a satellite, used to compute its position and velocity at a given time. The function sends a GET request to CelesTrak's API using fetch() and retrieves the data as raw text. This text is structured in groups of three lines per satellite:

- The first line contains the satellite name
- The second line includes orbital parameters such as inclination and eccentricity
- The third line includes data used to compute velocity and position

```
//Fetch satellite data from celestrak (TLE format)
async function fetchCelestrakData() {
  try {
    const response = await fetch("https://celestrak.org/NORAD/elements/gp.php?GROUP=active&FORMAT=tle")
    const tleData = await response.text() //fetch the response as text

    //Split the TLE data into lines, remove empty lines and extra spaces
    let rows = tleData.split("\n").map(row => row.trim()).filter(row => row.length > 0)
    let mySatellites = []    //Array to store the processed satellites
```

Figure 7.11 - Fetching and Processing TLE Satellite Data from Celestrak

The function iterates through the data in blocks of three lines to extract valid TLE sets. Each set is converted into a satellite object using the twoline2satrec() method from the satellite.js library. The current satellite position is then calculated using the propagate() function, which outputs coordinates in the Earth-Centered Inertial (ECI) frame. Finally, the ECI coordinates are transformed into geodetic coordinates (latitude and longitude) using Greenwich Mean Sidereal Time (GMST) via the eciToGeodetic() function.

```
//Process data in sets of three lines (Satellite name, tle line 1 and tle line 2)
for(let i = 0; i < rows.length; i += 3) {
  if(i + 2 < rows.length) {   //Ensure there are three valid lines
    let satName = rows[i]    //First line: satellite name
    let tleLine1 = rows[i + 1]
    let tleLine2 = rows[i + 2]

    try {
      //convert the TLE data to a satellite object
      const satelliteRecord = satellite.twoline2satrec(tleLine1, tleLine2)
      const currentTime = new Date()     //Get current time

      //Calculate satellite's position in Earth-Centered Inertial (ECI) coordinates
      const satPositionVelocity = satellite.propagate(satelliteRecord, currentTime)

      //Check if position data is available
      if(satPositionVelocity.position) {
        const ECIposition = satPositionVelocity.position   //Get satellite position in ECI coordinates
        const gmstTime = satellite.gstime(currentTime)     //Compute Greenwich Mean Sidereal Time
        const geodeticPosition = satellite.eciToGeodetic(ECIposition, gmstTime)  // Convert ECI to latitude/longitude

        const lat = satellite.degreesLat(geodeticPosition.latitude)    // Convert latitude to degrees
        const long = satellite.degreesLong(geodeticPosition.longitude)
```

Figure 7.12 - Parsing TLE Data and Calculating Satellite Position in ECI Coordinates

The Earth-Centered Inertial coordinate system is a 3-dimensional reference frame used to track the position of objects in space, such as satellites. It is centered at the center of the Earth but does not rotate with the Earth. It remains fixed relative to the stars, and it is used because it gives a stable reference point for calculating satellite positions as they move through space [17].

Satellites move very fast and orbit the Earth in complex paths. The ECI system helps to calculate their exact location in space at any given time. But, because the ECI coordinates are not easy to understand on a map, they are converted into regular latitude and longitude values. This conversion allows the server to determine if a satellite is currently near a specific location on Earth [18].

At the end of the fetchCelestrakData() function, the dataset is filtered to include only satellites whose calculated positions fall within +- 5 degrees latitude and longitude of Galway, Ireland. The coordinates for Galway are defined at the beginning of the code as 53.2709°N, -9.0627°W. This makes the data location specific.

```
//Check if the satellite is near Galway(within +- degrees latitude/longitude)
if(Math.abs(lat - galwayLat) < 5 && Math.abs(long - galwayLong) < 5) {
  mySatellites.push({
    name: satName,
    latitude: lat,
    longitude: long,
  })
}
```

Figure 7.13 - Filtering Satellites Near Galway Based on Latitude and Longitude

The reason for using a range of +- 5 degrees is to focus on satellites that are near Galway in the sky. 5 degrees is wide enough to catch satellites that might be visible, but not too wide that it includes satellites too far away. If a small number like 1 or 2 was used, we might miss nearby satellites and if a large number like 10 degrees was used, we would get too many satellites that are not near Galway.

When we say “within 5 degrees,” we are talking about a distance measured in latitude and longitude; how far away a satellite is from Galway on the globe. The code is checking if the satellite’s latitude is between 48.27° and 58.27° and if the longitude is between -14.06° and -4.06°. If both conditions are true, the satellite is considered to be close enough to Galway [19].

Lastly, the Node.js backend combines both the ESP32 and the CelesTrak data sources into a unified API response. This combined dataset is returned to the frontend through the root GET endpoint (/).

```
//API endpoint to get both Celestrak and GNSS satellite data
router.get('/', async function (req, res, next) {
  const celestrakSatellites = await fetchCelestrakData()
  res.json({
    celestrakData: celestrakSatellites,   //Processed Celestrak data
    gnssData: gnssSatellites
  })
})
```

Figure 7.14 - API Endpoint to Retrieve Both Celestrak and GNSS Satellite Data

7.2.3 Frontend Interface (React.js)

The user interface for the project was developed using React.js. React is a JavaScript library for building interactive and dynamic user interfaces [20]. In this project, React was used to build the frontend interface that interacts with the backend, presenting real-time satellite data in a clear and organized manner to the user.

The frontend was developed using a component-based design, where the user interface is broken down into smaller, reusable, and independent components that manage different parts of the data display and interaction. The key components used to display the satellite data are the GnssList and CelestrakList components. Both components contain smaller sub-components like GnssItem and CelestrakItem, which handle the display of individual satellite details such as satellite ID, latitude, and longitude.

The application contains an app.js file which serves as the root of the React app. It imports global styles and acts as a wrapper for the entire application component. It is used to render the HomePage component, which serves as the main interface for the user.

```
import '../styles/globals.css'


function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}

export default MyApp
```

Figure 7.15 - App.js File as the Root of the React Application

The HomePage component is responsible for displaying the satellite data. The component receives the information from the backend and passes it to two child components GnssList and CelestrakList, which display the GNSS and CelesTrak satellites in separate lists. The useEffect hook is used to fetch the data once when the page loads, ensuring the satellite information is displayed in real-time to the user.

```
function HomePage() {
  const [gnssSatellites, setGnssSatellites] = useState([]);
  const [celestrakSatellites, setCelestrakSatellites] = useState([]);

  useEffect(() => {
    fetchSatellites();
  }, []);

  async function fetchSatellites() {
    const response = await fetch("/api/get-satellites")
    let data = await response.json()
    setGnssSatellites(data.gnssSatellites || [ any
    setCelestrakSatellites(data.celestrakSatellites || [])
  }

  //combine both GNSS and celestrak satellites into a single array
  //const allSatellites = [...gnssSatellites, ...celestrakSatellites]    /
  const navigationSatellites = [gnssSatellites]
  const otherSatellites = [celestrakSatellites]

  //<SatelliteList satellites={allSatellites} />

  return (
    <div>
      <h1>Satellite Follower App</h1>
      <p>Satellites Near Galway:</p>
      <GnssList gnssSatellites={gnssSatellites} />
      <CelestrakList celestrakSatellites={celestrakSatellites} />
    </div>
  )
}

export default HomePage
```

Figure 7.16 - HomePage Component to Fetch and Display GNSS and Celestrak Satellite Data

The HomePage component calls the handler function in the */api/get-satellites.* The handler function is an asynchronous function that handles incoming requests to the API. When a request is made, it sends a GET request to the backend server to fetch the satellite data. This data is then processed and returned in JSON format.

```
async function handler(req, res) {
  const response = await fetch("http://localhost:8000/")
  const data = await response.json()
  console.log(data)
  //res.json(data);
  res.json({
    gnssSatellites: data.gnssData || [],
    celestrakSatellites: data.celestrakData || []
  })
}

export default handler
```

7.17 - API Handler to Fetch and Return GNSS and Celestrak Satellite Data

The frontend has a small, reusable CelestrakItem component which is responsible for displaying individual satellite details from the CelesTrak dataset. It receives its data through props which include the satellite's ID, latitude, and longitude. The component uses conditional rendering to display latitude and longitude only if the values are available.

```
function CelestrakItem(props) {
  return (
    <li className={classes.item}>
      <h2>{props.id}</h2>
      {props.latitude && (
        <h3 className={classes.coordinateWrapper}>
          {props.latitude} <span className={classes.lat}>(Latitude)</span>
        </h3>
      )}
      {props.longitude && (
        <h4 className={classes.coordinateWrapper}>
          {props.longitude} <span className={classes.long}>(Longitude)</span>
        </h4>
      )}
    </li>
  );
}

export default CelestrakItem;
```

Figure 7.18 - Component for Displaying Individual Celestrak Satellite Information

The CelestrakItem has a parent component called CelestrakList which is responsible for displaying a list of satellites from the CelesTrak dataset. It receives an array of satellites as a prop (props.celestrakSatellites) and loops through it using then.map() function. For each satellite, it renders a reusable CelestrakItem component, passing along relevant information like the satellite's ID, latitude, and longitude. The component uses conditional rendering, so if the list of satellites exists and has items, it displays them. If the list is empty, it will show a message "No Satellite From Celestrak".

```
function CelestrakList(props) {
    return (
        <ul className={classes.list}>
            {props.celestrakSatellites && props.celestrakSatellites.length > 0 ? (
                props.celestrakSatellites.map((satellite) => (
                    <CelestrakItem
                    key={satellite.satelliteId || satellite.name}
                    id={satellite.id || satellite.name}
                    latitude={satellite.latitude}
                    longitude={satellite.longitude}
                    />
                ))
            ):(
                <p>No Satellites From Celestrak</p>
            )}
        </ul>
    )
}

export default CelestrakList
```

Figure 7.19 - Component for Displaying a List of Celestrak Satellites

The frontend includes GnssItem and GnssList components which follow the same parent-child structure as the Celestrak components. GnssList acts as the parent component, receiving an array of GNSS satellite data as props and loops through this array to render individual GnssItem components. Each GnssItem is a reusable child component responsible for displaying the details of a single satellite, such as its ID, type, and country of origin. This design keeps the code modular and makes it easier to manage and update different parts of the user interface.

7.3 UART Communication

UART (Universal Asynchronous Receiver/Transmitter) is a serial communication used for asynchronous data transmission between devices. Because UART transmits data asynchronously, it does not require a clock signal. Instead, it relies on a predefined baud rate that both devices agree upon. Data is transmitted in the form of packets that include start and stop bits, which ensure that the receiver can correctly interpret each byte of information [21].

In this project, UART was used to facilitate communication between the ESP32 microcontroller and the L86-M33 GNSS module. The ESP32 uses the GPIO (General Peripheral Input Output) 16 on the UART1 port as the RX (Receive) pin. The TXD1 pin on the L86-M33 module's UART port, is used to transmit data and it is used for NMEA output [7]. The communication between the ESP32 and the GNSS module operates at a baud rate of 9600 bps (bits per second), determining the speed at which the data is transmitted.

When the GNSS module sends data, it transmits NMEA sentences which are strings that provide satellite position and time information. This information is sent over the UART connection and the ESP32 listens for incoming data on the RX pin and interprets the sentences received.

The communication is handled programmatically using the HardwareSerial class provided by the Arduino framework, which allows the ESP32 to use its secondary UART1 port for dedicated communication with external modules like the GNSS receiver [13].

8 Ethics

Although this project does not collect any personal data, it's important to be aware of ethical issues related to location-based technologies. Systems that deal with satellite and GNSS data can raise concerns around privacy and data security, especially if user locations are tracked or stored.

For this project, only publicly available satellite data is used, and no personal or sensitive information is processed. The goal of the project is educational and focuses on transparency and responsible use of technology. However, it is important to consider how similar systems might be used in the real world, and to make sure they are designed and used in a way that respects people's privacy and rights.

9 Conclusion

The outcome of this project is a working prototype of a satellite tracking system that uses a GNSS module and an ESP32 microcontroller to collect satellite data, which is then sent to a Node.js backend server and displayed in a frontend interface built with React.js. The system successfully captures NMEA sentences and transmits them over a UART connection, while also fetching additional satellite data from the CelesTrak public API source. The combined dataset is then displayed in a web-based interface, making the information easily accessible.

Throughout the development process, a strong focus was placed on modular design and real-time data flow. The frontend interface was built using a component-based approach, which allows for better organization, reusability, and easier maintenance. Backend endpoints were designed to handle data fetching and processing efficiently, while the ESP32 handled the communication with the GNSS module reliably. The system was tested using real data, and the results show that both sources, GNSS and CelesTrak, are successfully integrated and clearly visualized.

This project showcases how embedded systems, API communication, and web technologies can be combined to create an end-to-end solution. It serves as a strong foundation for future development, such as adding mapping features, enhancing user interactivity, or expanding data sources. The experience gained during this project has strengthened my full-stack web development skills and provided me with a solid foundation in satellite communication technologies.

10 Appendix

10.1 Bill of Materials (BOM)

Item	Quantity	Manufacturer	Sourced From	Cost
ESP32-DevKitC V4	1	ESPRESSIF	ATU Galway	Estimated 13.50
L86-M33	1	Quectel	ie.rs-online.com	13.41
Wire Jumpers Male to Male	3	MIKROE	ATU Galway	3.60
Wire Jumpers Male to Female	1	MIKROE	ATU Galway	3.60
Solderless Breadboard	1	Velleman	ATU Galway	3.95
Straight Through Hole Pin Header	1	RS PRO	ie.rs-online.com	14.92
Molex USB 2.0 cable	1	Molex	ie.rs-online.com	14.46

10.2 Source Code

This project's source code can be found at the following links:

https://github.com/oskark22/Project-Engineering-2024-25

https://github.com/oskark22/Project-Engineering-2024-25_ESP32

11 References

[1] "Global Navigation Satellite System (GNSS) and Satellite Navigation Explained," ADVANCED NAVIGATION, [Online]. Available: https://www.advancednavigation.com/tech-articles/global-navigation-satellite-system-gnss-and-satellite-navigation-explained/. [Accessed 2 February 2025].

[2] "Understanding GPS NMEA Sentences: GPGGA, GPGLL, GPVTG, and GPRMC," RF Wireless World, [Online]. Available: https://www.rfwireless-world.com/terminology/other-wireless/gps-nmea-sentences. [Accessed 15 February 2025].

[3] "What is PRN code?," Spirent, 1 October 2011. [Online]. Available: https://www.spirent.com/blogs/2011-10-17_what_is_prn_code. [Accessed 16 February 2025].

[4] T. Kruczek, "Demystifying the USSPACECOM Two-Line Element Set Format," Keep Track, 7 October 2023. [Online]. Available: https://www.keeptrack.space/deep-dive/two-line-element-set/. [Accessed 21 February 2025].

[5] "satellite.js v1.2.0," npm, Inc, 2014. [Online]. Available: https://www.npmjs.com/package/satellite.js/v/1.3.0. [Accessed 13 March 2025].

[6] "Universal Asynchronous Receiver Transmitter (UART) Protocol," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/universal-asynchronous-receiver-transmitter-uart-protocol/. [Accessed 14 April 2025].

[7] L. Quectel Wireless Solutions Co., "Mouser Electronics," [Online]. Available: https://www.mouser.ie/datasheet/2/1052/QWSC_S_A0007269014_1-2576162.pdf. [Accessed 15 November 2024].

[8] "ESP32-DevKitC V4," Espressif Systems (Shanghai) CO., LTD, [Online]. Available: https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32/esp32-devkitc/user_guide.html#what-you-need. [Accessed 25 January 2025].

[9] "ESP32 dev kit power options," Tech Explorations, [Online]. Available: https://techexplorations.com/guides/esp32/begin/power/. [Accessed 10 April 2025].

[10] "Satellite Based Augmentation System (SBAS)," Hexagon | NovAtel, [Online]. Available: https://novatel.com/an-introduction-to-gnss/resolving-errors/sbas. [Accessed 25 February 2025].

[11] "HttpClient," Arduino, [Online]. Available: https://docs.arduino.cc/libraries/httpclient/. [Accessed 10 February 2025].

[12] M. Hart, "Arduiniana," [Online]. Available: https://arduiniana.org/libraries/tinygpsplus/. [Accessed 10 February 2025].

[13] "[ESP32] use of HardwareSerial Library," Programming VIP, [Online]. Available: https://programming.vip/docs/esp32-use-of-hardwareserial-library.html. [Accessed 20 February 2025].

[14] G. Baddeley, "GPS - NMEA sentence information," [Online]. Available: https://aprs.gids.nl/nmea/#gsa. [Accessed 15 December 2024].

[15] "Introduction to Node.js," Node.js, [Online]. Available: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs. [Accessed 10 March 2025].

[16] "Express.js Tutorial," GeeksforGeeks , [Online]. Available: https://www.geeksforgeeks.org/express-js/. [Accessed 23 April 2025].

[17] "Coordinate Systems for Navigation," The MathWorks, Inc, [Online]. Available: https://www.mathworks.com/help/aerotbx/ug/coordinate-systems-for-navigation.html. [Accessed 13 March 2025].

[18] "Orbital Coordinate Systems, Part I," CelesTrak, [Online]. Available: https://celestrak.org/columns/v02n01/. [Accessed 13 March 2025].

[19] "Latitude, Longitude and Coordinate System Grids," GISGeography, [Online]. Available: https://gisgeography.com/latitude-longitude-coordinates/. [Accessed 13 March 2025].

[20] "React Introduction," W3Schools, [Online]. Available: https://www.w3schools.com/react/react_intro.asp. [Accessed 25 October 2024].

[21] "Basics of UART Communication," Circuit Basics, [Online]. Available: https://www.circuitbasics.com/basics-uart-communication/. [Accessed 25 April 2025].