



Escola d'Enginyeria de Telecomunicació i
Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

DEGREE FINAL PROJECT

TITLE: Drone control and monitoring by means of a web application

DEGREE: Bachelor's degree in Telecommunications Systems

AUTHOR: Jaskirat Singh Atwal

ADVISOR: Miguel Valero García

DATE: September 8th , 2023

Title: Drone control and monitoring by means of a web application

Author: Jaskirat Singh Atwal

Advisor: Miguel Valero Garcia

Date: September 8th, 2023

Overview

This document is a report of my final degree project, which aims to design and implement a software framework for controlling and monitoring drones through a web application, built using the Vue.js framework. With this, I offer a contribution to the Drone Engineering Ecosystem (DEE) , an ecosystem dedicated to the control and monitoring of drones through different technologies in which students of the EETAC, a university that belongs to the UPC, can contribute and enhance the ecosystem while doing their bachelor or master's degree final project.

Currently, there is a desktop application in the ecosystem that does the tasks of drone monitoring, control and mission planning. However, as the technology evolves, there is a need for web app as the advantages of a web platform for drone control and monitoring are numerous and compelling. The main benefit is the enhanced accessibility as the only needs are Internet connection and a browser, that no matter whether it is in the laptop, tablet or on a smartphone. As technology continues to evolve, web applications undoubtedly stand at the forefront of innovation in all domains. So, the focus of this project is to provide a web platform for drone controlling and monitoring.

The culmination emerges as a remarkably professional and contemporary web application that empowers the user with countless levels of control over drone operations. Notably, users have the freedom to decide the drone's movements and actions, ensuring a seamless and intuitive interface that facilitates effortless navigation. The capabilities extend beyond mere control, as users can devise a diverse range of missions. One standout attribute is the ability to create geofences, a vital tool for ensuring safe and responsible drone operations. Moreover, the web app enables users to fine tune drone's flying parameters. This level of customization guarantees that the drone's performance aligns precisely with the intended goals.

The outcome of the efforts yield a high level of satisfaction as the web platform has been successfully built, enhancing the basic functionalities of the desktop application while introducing additional features. This achievement holds personal significance as well, contributing to my growth by deepening my proficiency in new programming languages. On the other hand, the broader implication is that the Drone Engineering Ecosystem now boasts a new platform for drone control and monitoring.

This achievement marks both, a personal milestone and a significant stride forward for the DEE.

Títol: Control i monitoratge de drons mitjançant una aplicació web

Autor: Jaskirat Singh Atwal

Director: Miguel Valero Garcia

Data: 8 de setembre de 2023

Resum

En el present document s'exposa la memòria del treball final de grau que té com a objectiu dissenyar i implementar una arquitectura software per al control i monitorització de drons mitjançant una aplicació web creada amb el framework Vue.js. Això suposa una contribució al Drone Engineering Ecosystem (DEE), un ecosistema dedicat al control i monitoratge de drons a través de diferents tecnologies on els estudiants de l'EETAC, una universitat pertanyent a la UPC, poden contribuir i millorar l'ecosistema mentre realitzen el seu treball final de grau o màster.

Actualment, a l'ecosistema hi ha una aplicació d'escriptori que compleix les funcions del control de drons i la planificació de missions. No obstant això, a mesura que la tecnologia avança, sorgeix la necessitat d'una aplicació web a causa dels nombrosos i convincents avantatges d'una plataforma web per al control i monitorització de drons. El principal benefici és l'accessibilitat, ja que només es necessita d'una connexió a Internet i un navegador, sense importar si es troba en un ordinador portàtil, tauleta o en un smartphone. A mesura que la tecnologia segueix evolucionant, les aplicacions web sens dubte es situen a l'avantguarda de la innovació en tots els àmbits. Per tant, l'enfocament d'aquest projecte és proporcionar una plataforma web pel control i monitoratge de drons.

La culminació d'això es presenta com una aplicació web sumament professional i contemporània que otorga a l'usuari un control sense límits sobre les operacions dels drons. Destacablement, els usuaris tenen la llibertat de decidir els moviments i les accions del dron, assegurant una interfície fluida i intuïtiva que facilita una navegació sense esforç. Les capacitats van més enllà del simple control, ja que els usuaris poden realitzar una àmplia gamma de missions. Un tret destacat és la capacitat de crear geofences, una eina vital per garantir operacions de drons segures i responsables. A més, l'aplicació web permet als usuaris ajustar amb precisió els paràmetres de vol del dron. Aquest nivell de personalització garanteix que el rendiment del dron s'alineï amb precisió amb els objectius previstos.

El resultat dels esforços genera un nivell alt de satisfacció, ja que s'ha construït amb èxit una plataforma web que millora les funcionalitats bàsiques de l'aplicació d'escriptori i alhora introdueix unes característiques addicionals. Aquest èxit té una importància personal també, ja que contribueix al meu creixement en aprofundir la meua habilitat en nous llenguatges de programació. D'altra banda, la implicació més àmplia és que el Drone Engineering Ecosystem ara compta amb una nova plataforma per al control i monitoratge de drones.

Aquesta fita marca un pas endavant tant personalment com pel DEE.

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to my parents who have been by my side through thick and thin. Thank you for being the pillars of support during moments of doubt and the cheerleaders during moments of triumph. Your encouragement and belief in me have been fundamental in overcoming obstacles and reaching this milestone.

A special note of thanks goes to my friends and classmates too, the true companions of this educational voyage. Your companionship, friendship, the shared challenges and joyous moments over these years have transformed the pursuit of knowledge into an unforgettable experience. The laughter, the study sessions, and the collaborative spirit have made the path to this achievement all more enjoyable.

Also, I would like to give a special mention to Miguel Valero Garcia, the advisor and supervisor of my final project, who has been an outstanding guide throughout the journey of this project, offering me his time and guidance.

INDEX

INTRODUCTION.....	1
 CHAPTER 1: DRONE ENGINEERING ECOSYSTEM (DEE).....	5
1.1 DEE Structure.....	5
1.1.1 Software on-board	6
1.1.2 Ground station software.....	7
1.2 Contribution to the DEE structure.....	8
1.3 Hardware: real UAV used in Drone Lab for tests	8
 CHAPTER 2: OBJECTIVES AND WORK METHODOLOGY	10
2.1 Main goals	10
2.2 Work plan and methodology	12
2.2.1 Work plan	12
2.2.2 Methodology.....	13
2.2.3 Relation with the supervisor	14
 CHAPTER 3: TECHNOLOGIES INVOLVED	15
3.1 Vue.js : JavaScript, HTML and CSS	15
3.1.1 Template	17
3.1.2 Script.....	18
3.1.3 Styles	19
3.2 Python & Dronekit.....	21
3.3 MQTT & Mosquitto : messaging protocol.....	21
3.3.1 MQTT communication example in the DEE	22
3.4 MAVLink protocol.....	24
3.5 Mission Planner	24
3.6 Summary: the complete workflow	24
 CHAPTER 4: DASHAPP, FUNDAMENTAL FUNCTIONALITIES	26
4.1 Basic functionalities	26
4.1.1 Drone control and monitoring page	28
4.1.2 Mission planning page.....	33

CHAPTER 5: DASHAPP, ADDITIONAL FUNCTIONALITIES	35
5.1 Enhanced interface	35
5.2 Geofence creation	36
5.2.1 Map Card	37
5.2.2 Fence's parameters configuration card	38
5.2.3 Buttons' card	38
5.3 Drone's parameters settings	39
 CHAPTER 6: DASHAPP'S VUE CODE STRUCTURE	 40
6.1 Code organization and basic files	40
6.2 DashApp's entrance pages	43
6.2.1 ConnectPage	43
6.2.2 EntrancePage	44
6.3 DashApp's components	46
6.3.1 Drone Free Guiding.....	47
6.3.2 Rest of the DashApp's components	49
 CHAPTER 7: AUTOPILOT'S ON-BOARD MODULE.....	 51
7.1 Autopilot's code structure.....	51
7.2 Contribution to the autopilot module: geofence creation and drone's parameter settings	55
7.2.1 Drone's parameters configuration	56
7.2.2 Geofence functionality.....	59
 CHAPTER 8: SIMULATION AND DRONE LAB TESTS	 62
8.1 Simulation tests in Mission Planner	62
8.1.1 Configuring the scenario	62
8.1.2 Testing DashApp functionalities	64
8.2 Drone Lab tests	69
 CHAPTER 9: DASHAPP'S INTEGRATION TO THE DEE'S GITHUB REPOSITORY.....	 72
9.1 Uploading the code	72
9.2 Documenting the repository	72
9.3 Tutorial videos	72

CHAPTER 10: CONCLUSIONS 74

10.1 Objectives accomplished 74

10.2 Possible future improvements in DashApp 74

10.3 Personal conclusions 75

REFERENCES..... 77

ANNEX A: CONNECT PAGE CODE 78

ANNEX B: ENTRANCE PAGE CODE..... 83

ANNEX C: DRONE FREE GUIDING’S COMPONENT CODE 88

ANNEX D: AUTOPILOT’S GEOFENCE CREATION CODE 98

ABBREVIATIONS

AI	Artificial intelligence
CBLL	Campus Baix Llobregat
DEE	Drone Engineering Ecosystem
DOM	Document Object Model
ESC	Electronic Speed Controller
LED	Light Emitting Diode
MAVLink	Micro Air Vehicle Link
MQTT	Messaging Queue Telemetry Transport
TCP/IP	Transmission Control Protocol / Internet Protocol
UAV	Unmanned Aerial Vehicle
UPC	Universitat Politècnica de Catalunya

INTRODUCTION

In an era defined by technological breakthroughs, drones have emerged as one of the most important innovations in the past decades. These devices, that previously belonged to the military personnel only, now being available to the general public, have unlocked a realm of possibilities that promise to reshape how we perceive and interact with the world around us in the present and will perceive in the future. From the simple filming purposes to navigating hazardous environments for search and rescue operations, drones' applications know no bounds.

The following document is also based on them, the drones. Concretely, by this project I will contribute to the Drone Engineering Ecosystem, from now on referring to it by DEE. As his own name indicates, DEE is an ecosystem that grants us the possibility to control a drone and perform various tasks with it through different technologies and applications. Like said, the ecosystem has various applications, some with more serious and mission planning purposes like the Dashboard and other with more gaming and joy purposes like the Drone Circus. It has to be taken into account that, DEE is an active ecosystem that has been and is being developed and updated frequently, mainly by students of the EETAC, an engineering school that belongs to the UPC, who seek to contribute to the ecosystem while doing their bachelor or master's degree final project.

Students participating in the DEE have consistently showcased an impressive array of technologies that they have studied as part of their degree program, as well as those they have explored independently on themselves. This rich diversity of technological influences has enriched the DEE with a collection of innovation that covers the entire spectrum of modern engineering. From aerodynamics and control systems to software development, students have tapped into a myriad of disciplines, each contributing a distinct facet to the evolution of the ecosystem. As we gaze toward the future, the potential for student contributions to the DEE becomes even more interesting. The rise of Artificial Intelligence (AI) is a reality, and it could change drone technology a lot. Students could use AI algorithms to enable autonomous navigation, adaptive learning, and real-time data analysis. By embracing AI driven solutions, the DEE stands to elevate its capabilities to another level, navigating complex environments and tasks with incredible precision.

The ecosystem is published and is available in a GitHub repository, so it is publicly accessible and every student who desires to participate and contribute can do it easily while doing their degree's final project.

Moreover, the inauguration of the Drone Lab, an outdoor drone laboratory within the CBL campus of the UPC, designed for testing drones and conducting experiments for the Drone Engineering Ecosystem (DEE) among other purposes, has sparked a notable surge of interest among the local community. This newfound curiosity surrounding drones has taken a positive turn, particularly within the local citizens.

One prominent manifestation of this enthusiasm can be seen in the neighbouring primary schools, which have expressed a keen desire to engage their students with drones. The establishment of the Drone Lab has provided an exciting opportunity for these schools to showcase drones as engaging educational tools, facilitating interaction and learning beyond the confines of traditional classrooms. The synergy between the Drone Lab and nearby primary schools underscores the DEE's broader impact on education and community engagement.

As a telecommunication engineering student myself, throughout the career we have dealt with each and every aspect that has to do with nowadays telecommunication systems. That is to say, delving deep from the hardware electronics to the software creating a combination of hardware and software that powers our interconnected world. Without any doubt software engineering has been one of my favourite subjects in my career as it is a realm of constant evolution and boundless creativity. The skills you acquire in software engineering can be used and applied across countless sectors. That is the main reason why I chose to contribute to the DEE, as this provided me to not only practice and improve my skills in programming languages already taught in my degree but also gave me the opportunity to learn new programming languages like Vue.js and Python or messaging protocols like MQTT and MAVLink that I could not learn in the formal degree's academic plan for a Telecommunications Systems' engineer.

By this project, I contributed to the DEE with the main objective of designing a software architecture for the control and monitoring of drones through a web application. This web application includes some additional functionalities while maintaining the actual ones from the already built desktop application in Python named as Dashboard, available in the DEE GitHub. **So, the core purpose of this project is to shift the existing Python and Tkinter based Dashboard into a web application (web app) version naming it DashApp.** This conversion responds to the contemporary demand for web accessibility, given the widespread utility and advantages associated with web apps in the present digital landscape.

Web apps provide a seamless and convenient user experience, as they can be accessed directly from web browsers without requiring users to download or install any software. This accessibility transcends device and operating system limitations, making it easier for a broader audience to interact with the Dashboard. Moreover, web apps enable real-time updates, facilitating efficient communication and data sharing among users. Furthermore, the transition to a web app format allows for scalability and adaptability. **In addition to conveying the Dashboard to a web app , I have also incorporated several significant enhancements to the Dashboard.**

In order to achieve this, I approached the project by dividing it into two distinct sections. The first section centers around the conversion of the existing Dashboard into its web app version. I refer to this aspect as the "Conveying the fundamental Dashboard functionalities."

This segment involves the meticulous migration of the Dashboard's core features and functionalities to a web-based platform. The second section encompasses the strategic augmentation of the Dashboard's capabilities through the integration of additional functionalities.

By approaching the project with a clear division into these two sections, I am able to tackle the dual objective of conveying the Dashboard to its web version while simultaneously fortifying it with features that enhance its value. This strategic approach ensures that the resulting web app not only upholds the original functionalities of the Dashboard but also offers an enriched experience that supports more effective control over the drone.

Concretely the functionalities to develop are:

1. Conveying the fundamental Dashboard functionalities
 - i. **Free guiding the drone.** The final user must be able to send basic commands to the drone. For instance, to arm, take off, land, etc.
 - ii. **Live picture/video stream.** The user must be able to see live video of everything that the drone camera is capturing.
 - iii. **Simple flight planning.** The user must be able to design a flight plan. For example, set a series of waypoints where the drone should go and take a picture if the user desires.
 - iv. **Showing telemetry data of the drone in real time.** The user must be able to see the telemetry data of the drone while controlling it.
2. Additional functionalities
 - i. **Inclusion Geofence creation.** The user must be able to create an inclusion geofence, whether it is a circular or polygonal one, wherever in the map of the Drone Lab so that the drone only can fly in that area. Additionally the user is able to set Geofence parameters like the maximum altitude, circular fence radius...
 - ii. **Setting drone parameters.** The user must be able to configure the drone's basic parameters like take off altitude, ground speed, RTL altitude, etc.
 - iii. **User interface functionalities:** the web app made takes the user experience to the next level. The app designed in this project has a modern makeover with a host of exciting new functionalities. This means that the web app has got a contemporary interface, for instance with the bright and dark mode functionality among others, with which the user can personalize its viewing experience according to its preference.

The present document is structured in 10 chapters in the pages that follow. Concretely, in the opening chapter, Drone Engineering Ecosystem (DEE), explains in detail the ecosystem. That is to say, how it is organized, which technologies are implemented on the different modules, how these modules communicate with each other and in which state the ecosystem stands currently.

Straight next, the second chapter focuses on both, the goals set and the work plan followed in order to accomplish the objectives settled prior elaborating the project. Moving on, the third chapter continues diving into theoretical architecture, in this case we will talk about the technologies and programming languages used to bring forth results.

The report takes a turn in the fourth and fifth chapter, where the focus finally shifts from the theoretical section to the practical one, explaining the web application itself. That means, describing what each and every functionality of the web-app does and how it was done. As we progress, it comes up the chapters where the tests done in both, the Mission Planner simulator and the Drone Lab of EETAC alongside with their results are explained. The practical section finishes with the code and tutorial videos' description.

Finally, there is the conclusions chapter alongside with bibliography and annex to conclude the project.

CHAPTER 1: DRONE ENGINEERING ECOSYSTEM (DEE)

With this first chapter, we start the theoretical section of the project. In this case, the following chapter talks in detail about the Drone Engineering Ecosystem where firstly it describes how the architecture of the ecosystem was when I first started working with it and how it stands now, after my contribution. In addition the real drone in question is also described in this chapter.

As introduced before, DEE is an active ecosystem published in a GitHub repository whose main objective is to control and monitor drones via different applications and technologies. This ecosystem is publically accessible in the GitHub repository (see [1]).

1.1 DEE Structure

The ecosystem's architecture when I first started to work with, it is the given in the following figure:

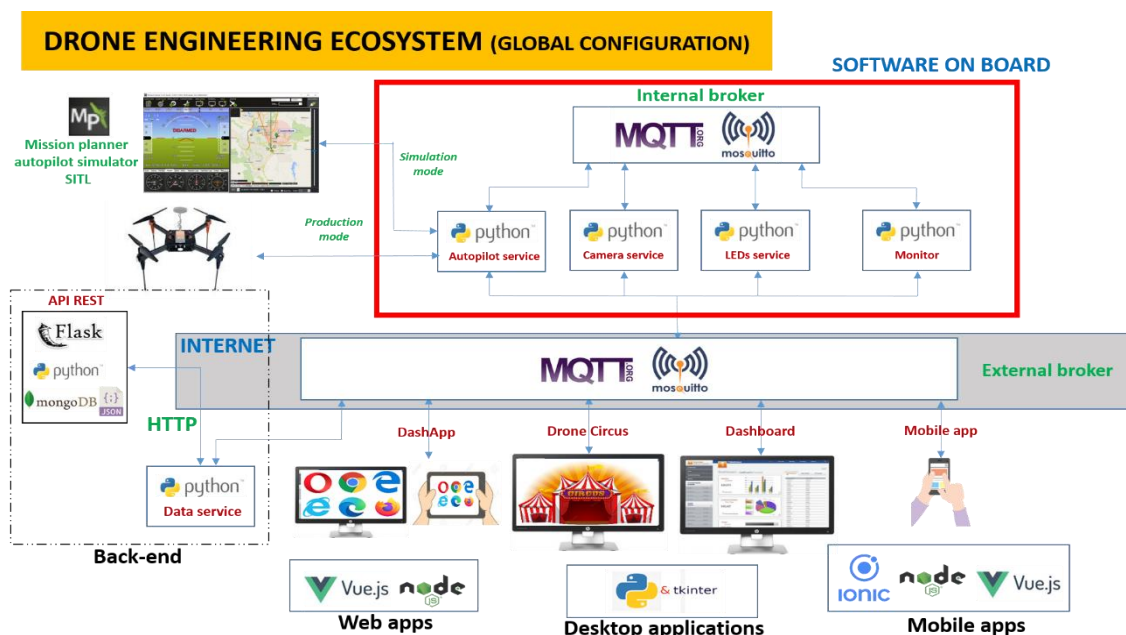


Fig. 1.1 Drone Engineering Ecosystem's architecture in February 2023

As shown in the figure above, the architecture of the ecosystem is divided into two big blocks: **Software on-board** and the **Ground station software** which includes modules of numerous front-end applications developed in different technologies. The on-board and ground station blocks communicate with each other through MQTT brokers (Message Queuing Telemetry Transport), a Mosquitto messaging protocol that is based on TCP/IP protocol.

It operates on a publish-subscribe model, where devices can publish messages to specific “topics” and other devices can subscribe to these topics to receive relevant messages.

1.1.1 Software on-board

The Software on-board block is the red box shown in the general architecture of the ecosystem.

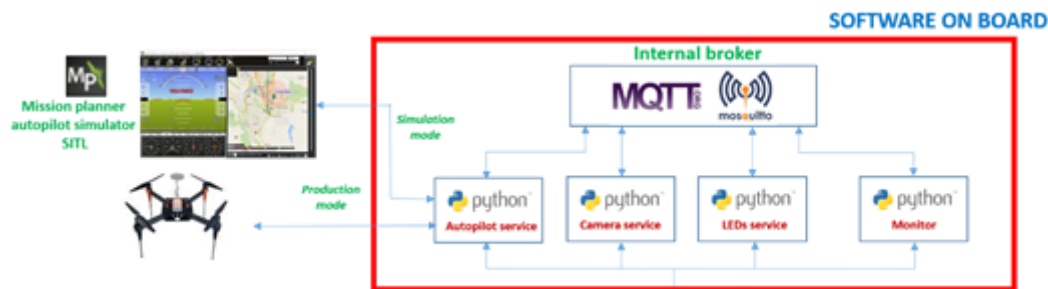


Fig. 1.2 Software on board’s architecture in February 2023

This block represents, as his own name indicates, each and every module coded in Python that is present within the drone’s Raspberry Pi. The main objective of these modules is to control the different tasks of the drone.

Concretely, this block is formed by 4 modules:

1. **Autopilot service:** this is the fundamental module, as it is responsible for executing the commands sent by the ground station such as arm the drone, take off, create a geofence, etc. The autopilot module is built in Python using the Dronekit library.
2. **Camera service:** camera module whose objective is to control the camera configured on the drone and send live video stream or take pictures with it among other things.
3. **LEDs service:** module that controls the LEDs of the drone and executes the orders of switch ON or switch OFF.
4. **Monitor:** a module that records on board drone data for future analysis.

The internal broker establishes the communication among all the on-board modules using the MQTT protocol alongside Mosquitto.

Furthermore, the drone’s functionalities are simulated by Mission Planner , a flight simulation application that in our case works as a ground station for a real drone. Mission Planner offers countless advantages for simulating drones, making it a very valuable tool for professionals. Its user-friendly interface facilitates easy mission planning and execution. Users can graphically design complex flight plans, waypoints, all while visualizing the simulated drone’s movements in real time. Mission Planner offers a perfect solution for pre testing drones before real world trials due to its advanced simulation capabilities.

By providing a realistic environment and accurate hardware emulation, it allows users to assess drone performance under various conditions, reducing risks and potential damages associated with physical testing, while also saving time.

1.1.2 Ground station software

Once described the on-board software, let's shift our attention now to the ground station software.

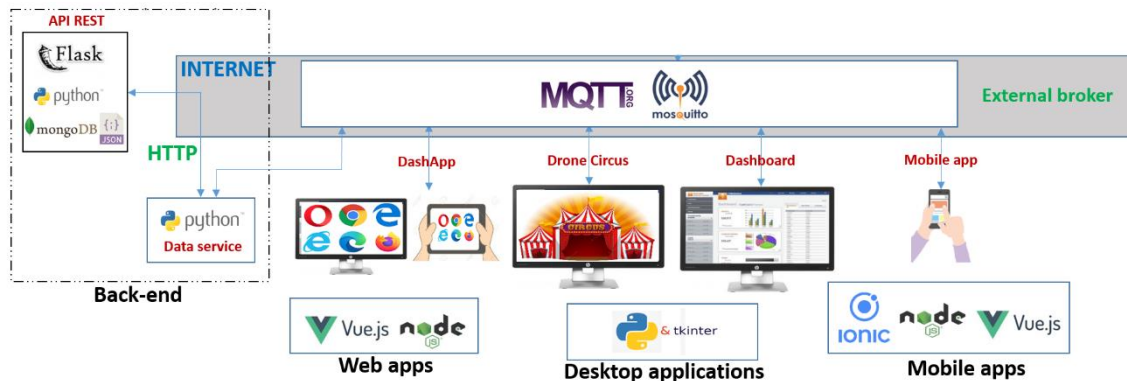


Fig. 1.3 Ground station's software architecture in February 2023

The modules available in the ground station are front-end applications that allow the user to control the drone in every way. Specifically, this possesses:

1. **Dashboard:** a desktop application made with Python and Tkinter (a package used to build a graphic interface in Python) that employs and uses all the services provided by the on-board modules (autopilot, camera, LEDs, etc).
2. **DashApp (to build):** the web application to create in the project in question. Made with Vue, and that will undertake most of the Dashboard functionalities plus adding some additional ones like the creation of the geofences or the settings of the parameters. In contrast with the conventional Dashboard, as this is a web app, it can be operated from any laptop, tablet or smartphone that has a browser connected to the Internet.
3. **DroneCircus:** another desktop application made with Python and Tkinter that allows the user to interact with the drone platform in a funny and joyful way. For example you can control the drone through making different faces such as a smiling face, sad face, etc.
4. **Mobile App:** a mobile web application made with Vue and Ionic that allows the user to employ a limited set of functionalities from a smartphone connected to the Internet.

1.2 Contribution to the DEE structure

In my role within the DEE, my contributions have encompassed two pivotal aspects of our project's evolution.

First and foremost, I headed the creation of DashApp, an innovative web application that extends the functionalities of our existing Dashboard. Simultaneously, my involvement extended to the on-board module: the autopilot's module. Here, I meticulously integrated code to accommodate the novel features introduced by the DashApp that will be explained in detail later on.

Lastly, completing my involvement in the ecosystem, I took charge in scripting the DashApp's segment within the Drone Engineering Ecosystem's Github repository. By documenting my contribution, I ensured transparency and the continuity of our project's development.

1.3 Hardware: real UAV used in Drone Lab for tests

A real Unmanned Aerial Vehicle, from now on UAV, is used for the real world tests in the Drone Lab. The drone is shown in the following figure:



Fig. 1.4 Drone used for DEE's tests in the Drone Lab

His main physical characteristics are:

- Formed by 4 motors, model HS2216 920KV
- 4 helices, model T-MOTOR T2045II

- 4 ESCs (Electronic Speed Controller) used for controlling the motors' speed, model Hexsoon 20^a.
- Drone's body
- Energy distributing plaque
- Battery
- Orange Cube Pixhawk flight controller (autopilot).
- GPS receiver
- Radio waves receiver
- Raspberry Pi microcontroller
- Camera used for taking pictures or recording the video stream.
- LED lights
- Altimeter

CHAPTER 2: OBJECTIVES AND WORK METHODOLOGY

Moving on, the second chapter revolves around the goals set prior to the elaboration of the DashApp and the work methodology followed so as to accomplish these objectives.

As previously stated, there was no web app in the ecosystem that fulfilled the Dashboard functionalities in its whole. So there was no web app that had the right to be called the Dashboard Web App (DashApp).

2.1 Main goals

First and foremost, this project had as its main objective the design of a software architecture for drone control and monitoring through a web application. In simple words, building a web app that had every or most of the Tkinter made desktop Dashboard functionalities.

The reason behind creating a web app is that using a web app for drone control and monitoring offers several advantages compared to a desktop application. One of the key advantages is accessibility. A web application can be accessed from any device with a browser, enabling seamless control and monitoring from smartphones, tablets, and computers, regardless of the operating system.

As a life law, nothing is perfect, so either it is a web app for drone control and monitoring and has its own disadvantages that will be discussed in his respective chapter.

So as to get the work done I established some targets for myself:

1. First and the most important goal is to build a web app user interface that does the same functions as the current Dashboard. In simple terms to convey the functionalities of the Dashboard to a web app. Specifically these functionalities are:
 - **Enabling autonomous drone control:** The final user should possess the capability to issue fundamental instructions to the drone, such as arming, initiating take-off, or commanding the directions to go.
 - **Streaming live video:** Users should have the capability to observe real-time video feed captured by the drone's onboard camera, granting them a dynamic perspective of the drone's surroundings.
 - **Mission and route planning:** Users should have the ability to formulate flight paths for the drone to follow.

That is to say defining a sequence of waypoints that guide its trajectory and even specifying actions like capturing photos at specific points.

- **Displaying real-time telemetry information:** Users should be able to monitor the drone's essential data in real-time as they actively control its movements. This telemetry data encompasses critical flight statistics and information that contribute to informed decision-making during drone operation.

2. Expanding the functionalities of the web app. That is to say add some extra functionalities to the DashApp that the actual Dashboard does not have. In my case, talking with the supervisor, we concluded that adding the **geofence functionality** to the web app would fit perfectly.

Additionally I personally aimed to add some more extra functionalities as I was progressing with the app. Introducing some important supplementary functionalities to the web app beyond its desktop counterpart holds paramount importance in elevating user engagement and enhancing the overall user experience. By introducing additional features like the geofence one, I am not only differentiating my web app from the existing desktop version but also tapping into the full potential of web technologies.

3. Another important objective of this project focuses on the **augmentation of the on-board autopilot module, making it compatible with the additional capabilities of the DashApp**. The intention is to empower the autopilot module with the ability to not only create geofences but also configure them according to specific parameters.

Additionally, the aim is to enhance the autopilot module by integrating parameter configuration functions. This means enabling the module to receive commands from the DashApp that would trigger changes in various drone parameters. Such adjustments could encompass ground speed, return-to-launch altitude, and return-to-launch speed, among others. This functionality empowers users to fine-tune the drone's behaviour and performance based on their specific needs.

4. The final goal is to integrate the DashApp into the DEE's GitHub repository. This entails an approach that involves uploading the DashApp's source code to the GitHub repository. In addition, the integration process extends to creating informative tutorial videos. These videos will serve as user-friendly guides, walking through various aspects of the DashApp's functionalities and providing step-by-step instructions.

Furthermore, an important aspect of this integration involves documenting comprehensively the DashApp's repository. The goal is to write clear and concise guides that elucidate how students can navigate and use the DashApp.

Looking forward, the integration also paves the way for potential enhancements and refinements. As the app becomes an integral part of the ecosystem, it opens the door for future contributions and developments. Students can collaborate to enhance features, fix bugs, and introduce innovative functionalities, thereby fostering a dynamic environment of continuous improvement.

Leaving aside these fundamental objectives for the project itself, my personal goal is to learn new programming languages. As a student that loved programming, my aspiration is to continuously expand my repertoire of programming languages and skills. By delving into new languages, I aim to broaden my problem solving capabilities and unlock innovative ways to approach challenges. This pursuit not only enriches my expertise but also equips me with the adaptability needed to thrive in the ever evolving landscape of technology.

2.2 Work plan and methodology

Next, I would like to take this opportunity to walk you through my work plan and the methodology I have put into action to ensure the efficient and timely execution of all tasks at hand.

2.2.1 Work plan

As shown in the **Fig. 2.1**, I elucidate my work plan using a Gantt diagram. This visual representation will provide a clear overview of the project timeline, task dependencies, and milestones, offering a comprehensive understanding of how I intend to manage and accomplish the ongoing responsibilities.

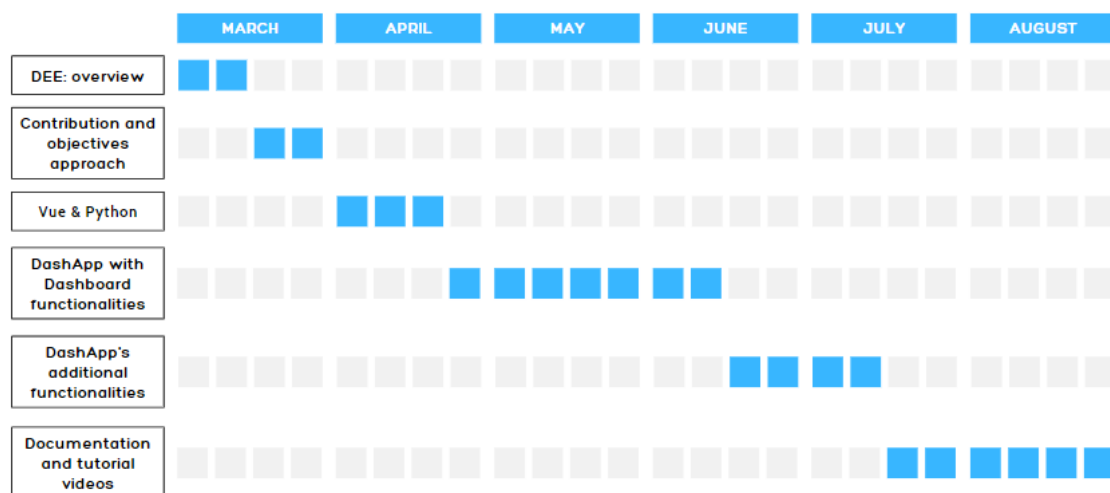


Fig. 2.1 Gantt diagram of the initial approach

2.2.2 Methodology

As illustrated on the Gantt diagram, practically the entire month of March was dedicated to delve deep into the Drone Engineering Ecosystem. Since the goal was to know in first hand the ecosystem's work plan so that I could finally decide my contribution and objectives. These first weeks also were to check if all the modules in the DEE were correctly working.

Moving on, over the first three April weeks, I have dedicated myself to an intensive learning endeavour, focusing on two distinct programming languages: Vue.js and Python. Vue.js will be harnessed for crafting the web application, capitalizing on its capabilities for seamless user interface design. In tandem, Python will serve a pivotal role in enhancing the drone's autopilot module, leveraging the already implemented dronekit library to facilitate autonomous flight operations. The investment of these three weeks has been instrumental in gaining a comprehensive understanding of Vue.js's frontend capabilities and Python's implementation as these newfound skills will be crucial in my contribution to the DEE.

As advancing in the timeline reflected in the provided Gantt diagram, it illustrates the progression of my efforts in building the web application. The bulk of May and the initial weeks of June were meticulously directed toward the development of the fundamental functionalities of the DashApp, a Vue and JavaScript based counterpart mirroring the functionalities of the Tkinter & Python desktop Dashboard. Subsequently, from mid-July through August, my focus shifted to a strategic expansion of the web app's capabilities. This period was marked for conceiving and implementing additional features, a decision proven astute for enhancing the app's utility.

This strategic allocation of time and effort has empowered the web app to transcend its initial scope, promising a more comprehensive and versatile user experience.

Finally we shifted gears in August as this was a month marked by focused activities, as indicated by the Gantt diagram. It was during this time that I directed my attention towards key tasks. First and foremost, a substantial portion of the month was dedicated to meticulously documenting the project, ensuring that the culmination of my work is conveyed with clarity and precision. Additionally, a significant effort was invested in conducting thorough tests within DroneLab, validating the robustness and reliability of the developed solutions. Lastly, in a bid to facilitate seamless user understanding and engagement of the DashApp done, I endeavoured to create tutorial videos and document the DEE's DashApp section.

This compilation of efforts underscores the culmination of this project, combining documentation, practical validation, and user friendly guidance to encapsulate the comprehensive scope of my work.

2.2.3 Relation with the supervisor

My relationship with the supervisor of my final degree project, Miguel Valero, has been fundamental in shaping the success of the project.

While the Gantt diagram may not explicitly reflect it, our collaboration was characterized by consistent and constructive interactions. Approximately every two weeks, I had meetings with Miguel to provide updates on my progress during that time frame. These meetings served as checkpoints, allowing me to showcase the advancements I had made and engage in productive discussions about the forthcoming tasks.

CHAPTER 3: TECHNOLOGIES INVOLVED

Transitioning to the third chapter, our focus now shifts to the technologies that underpin the development of the DashApp. This chapter delves into the heart of the project, exploring the essential tools and platforms that empower the creation and functionality of this innovative application.

At the forefront of these technologies is Vue.js (see [2]), a dynamic framework that serves as our web development partner.

Python also stands as a vital player in this project, driving the autopilot's core intelligence by integrating the Dronekit library. The synergy between Python and Dronekit facilitates the realization of a great autopilot.

At the heart of this ecosystem, effective communication takes an important role, where MQTT assumes the role of a crucial connector, harmonizing the conversation between the Vue.js created web application and the Python driven autopilot. This resilient communication protocol is the backbone of dependable information interchange, crucial for real time drone control and monitoring.

Integral to our technological arsenal are the MAVlink messages, which pave the way for drone's geofence configuration. These messages, acting as the language of communication, establish a clear channel for configuring the drone's geofence behaviour and characteristics.

Amidst these building blocks, the Mission Planner emerges as a significant asset. Serving as a simulator, the Mission Planner extends a platform to test and validate the drone's functionalities, ensuring a smooth transition from conception to real-world implementation.

In the subsequent chapter, we will witness these technologies culminating into the creation of a web application that not only controls and monitors a drone but also exemplifies the synergistic potential of modern software and hardware integration.

3.1 Vue.js : JavaScript, HTML and CSS

As introduced before, Vue.js (from now on referring it to simply as Vue) is a progressive JavaScript framework built to create interactive and dynamic user interfaces. Vue is known for its simplicity, flexibility, and ease of integration. In our case we use the Composition API instead of the Options API to develop the code in Vue. Composition API is a feature introduced in Vue 3 to provide a more flexible and organized way of creating and managing the logic within Vue components. That is to say, the Composition API is an alternative approach to structuring Vue components that aims to provide better organization, reusability, and manageability of component logic, especially in complex scenarios.

Vue is based on components, in other words in the idea of breaking down the user interface into smaller, self contained building blocks. These components encapsulate both the visual elements and the associated functionality of a particular part of a web application. A Vue component is like a mini web page with its own HTML, JavaScript, and even CSS code. It's a self contained unit that can be reused throughout the project. Components are formed by three main elements, the template, script and styles :

- **Template:** This is where you define the structure and layout of your component using HTML syntax. It's like creating a blueprint for how your component should look.
- **Script:** The script part is where you add the behaviour and functionality of your component using JavaScript. This is where you define functions, variables, and even data that your component needs to work properly.
- **Styles:** Just like regular web pages, components can also have their own styles. You can write CSS specific to your component to make it look unique (<styles scoped>). These styles won't affect other parts of your application.

Next, we will delve into the fundamental elements of Vue.js through a hands-on example of our very own web application as it is always better to understand the concepts by means of an example rather than mere descriptions. Through this example, you will witness firsthand how Vue's building blocks, like components, templates, scripts, and styles, come together to form a dynamic interface.

For instance, the following figure is the Dashboard.vue component of my app:

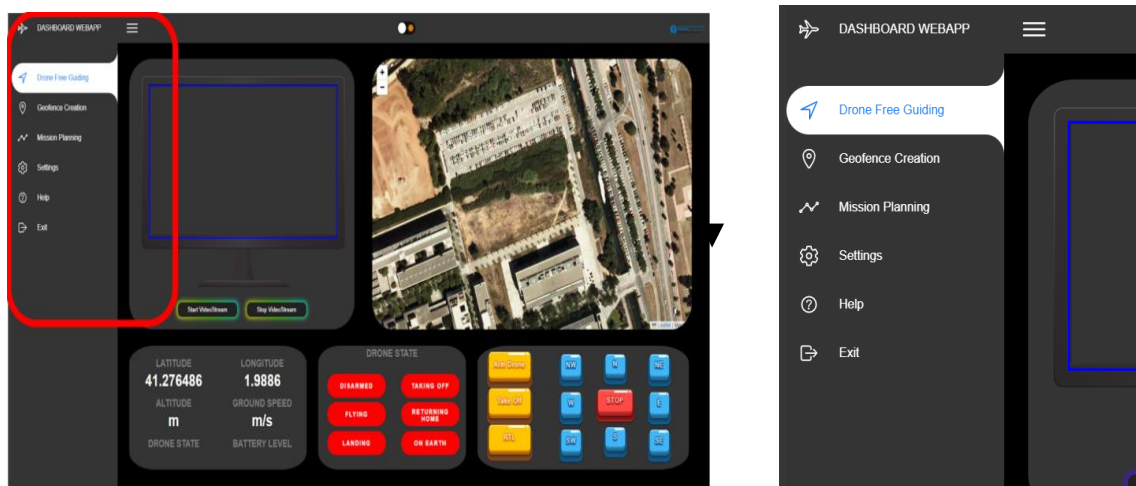


Fig. 3.1 Cropped *Dashboard.vue* component from the DashApp

In the realm of the DashApp development, the *Dashboard.vue* component serves as a hub of functionality, housing child components like *FreeGuiding.vue*, *Geofence.vue*, *MissionPlanning.vue* and more as illustrated in **Fig 3.2**.

This approach embraces componentization, breaking down the dashboard into manageable parts. The *Dashboard.vue* component orchestrates a range of visual and interactive elements, while its internal structure reveals five other subcomponents: *FreeGuiding.vue*, *Geofence.vue*, *MissionPlanning.vue*, *Settings.vue* and *Help.vue*.

This component driven design showcases the power of modular development, where distinct features are encapsulated for seamless integration. The result is a user friendly and feature rich dashboard that aligns with principles of modularity and reusability, highlighting the efficiency of component based architecture in modern web development.

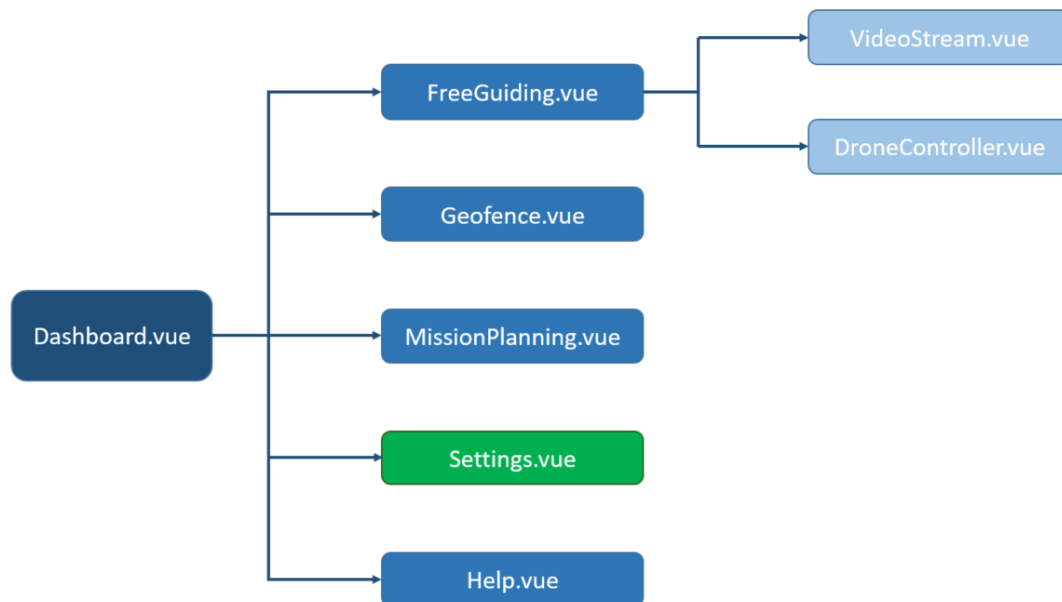


Fig. 3.2 Component distribution from the *Dashboard.vue* component

Next, by means of the *Settings.vue* component we will explain the Vue's fundamental three elements: template, script and styles.

3.1.1 Template

As introduced before Vue's template element is a fundamental part of Vue's syntax that allows you to define the structure of the user interface. It serves as a placeholder for HTML content, where we describe the layout, structure, and content of our component without worrying about JavaScript logic. The content within the template element is used to render the component's HTML whenever the component is used.

For instance, in the **Fig.3.3** it is shown the template element of the DashApp. Where the template creates a clean and organized user interface for configuring different parameters related to a drone's flying configuration. The use of classes, styling, and Vue directives ensures an interactive and visually appealing experience for users interacting with the configuration form as shown in **Fig.3.6**.

```

<template>
  <div style="width:100%; height: 100vh; display: grid; place-items: center;">
    <div class="settingsStyle">
      <div style="margin:1%">
        <p> DRONE'S PRE-CONFIGURATION PARAMETERS</p>
      </div>

      <div class="card">
        <h2 class="title"> Configure the Take-Off altitude (between 5 and 15):</h2>
        <input type="number" id="takeOffAltitude" name="takeOffAltitude" min="5" max="15">
        <button class="submitButton" @click="takeOffSubmitted"> Submit </button>
      </div>

      <div class="card">
        <h2 class="title"> Configure drone's Ground Speed in m/s (between 2 and 10):</h2>
        <input type="number" id="groundSpeed" name="groundSpeed" min="2" max="10">
        <button class="submitButton" @click="groundSpeedSubmitted"> Submit </button>
      </div>

      <div class="card">
        <h2 class="title"> Configure RTL maximum altitude (between 5 and 15):</h2>
        <input type="number" id="RTLAltitude" name="RTLAltitude" min="5" max="15">
        <button class="submitButton" @click="RTLAltitudeSubmitted"> Submit </button>
      </div>

      <div class="card">
        <h2 class="title"> Configure drone's Ground Speed in m/s (between 5 and 10):</h2>
        <input type="number" id="RTLSpeed" name="RTLSpeed" min="5" max="10">
        <button class="submitButton" @click="RTLSpeedSubmitted"> Submit </button>
      </div>
    </div>
  </div>
</template>

```

Fig. 3.3 Template element from the *Settings.vue* component

3.1.2 Script

Vue's script element is used to define the JavaScript logic for a Vue component. It contains the data, methods, lifecycle hooks, and other functionality that define the behaviour of the component. The code within the script element is written in JavaScript or in some cases in TypeScript and interacts with the template and other parts of the component.

In our case it is of the utmost importance to understand the JavaScript code shown in **Fig.3.5** in order to know Vue's working principle. Firstly, The “import” statement brings in necessary functions and libraries, such as “ref” for creating reactive variables, “inject” for accessing injected values, and “Swal” for using SweetAlert2 for pop-up notifications.

Next, the “setup()” function is the entry point for the Composition API logic. Inside this function, various data variables are created using the “ref” function to make them reactive. The variables in it like the “takeOffMaxAltitude” represent different configuration settings for a drone.

The “emitter” is injected using “inject('emitter')”, which allows the component to communicate with the parent component or other components of the application in order to pass variable values between them.

Moving on, the script defines several functions: `takeOffSubmitted`, `groundSpeedSubmitted`, `RTLAltitudeSubmitted`, and `RTLSpeedSubmitted`. These functions handle user input for configuring drone parameters and emit values to the parent component through the emitter.

Finally, the “return” statement at the end of the “setup()” function specifies the functions that will be exposed and accessible in the template for handling user interactions.

```
<script>
import { ref, inject, onMounted } from "vue";
import Swal from 'sweetalert2';

export default {
  components: {

  },

  setup() {
    const emitter = inject('emitter');
    let takeOffMaxAltitude = ref(5);
    emitter.emit('setTakeOffAltitude', "5");
    let groundSpeed = ref(4);
    emitter.emit('setGroundSpeed', "4");
    let RTLAltitude = ref(10000);
    emitter.emit('setRTLAltitude', "10000");
    let RTLSpeed = ref(5000);
    emitter.emit('setRTLSpeed', "5000");

    function takeOffSubmitted() { ...
    }

    function groundSpeedSubmitted(){ ...
    }

    function RTLAltitudeSubmitted(){ ...
    }

    function RTLSpeedSubmitted(){ ...
    }

    return {
      takeOffSubmitted,
      groundSpeedSubmitted,
      RTLAltitudeSubmitted,
      RTLSpeedSubmitted,
    }
  }
}
</script>
```

Fig. 3.4 Script element from the *Settings.vue* component

3.1.3 Styles

Eventually, the component elements are closed with the `<style>` element. The `<style>` element is used to define the CSS styles for a component. It allows us to apply visual styling to the components' HTML elements, ensuring a consistent and visually appealing user interface.

```

<style scoped>
:root {
  /* --blue: #2a2185; */
  --blue: #1D7DE8;
  --white: #fff;
  --gray: #f5f5f5;
  --black1: #222;
  --black2: #999;
}

0 references
.settingsStyle {
  width: 40%;
  height: 81%;
  border-style: solid;
  border-color: transparent;
  border-radius: 8%;
  background-color: #343434;
}

0 references
.submitButton {
  background-color: var(--blue);
  color: white;
  font-weight: bold;
  width: 25%;
  height: 21px;
  margin-left: 4%;
  border-radius: 20%;
}
</style>

```

Fig. 3.5 Style element from the *Settings.vue* component

Together, these three elements merge into the interactive user experience captured in the following figure.

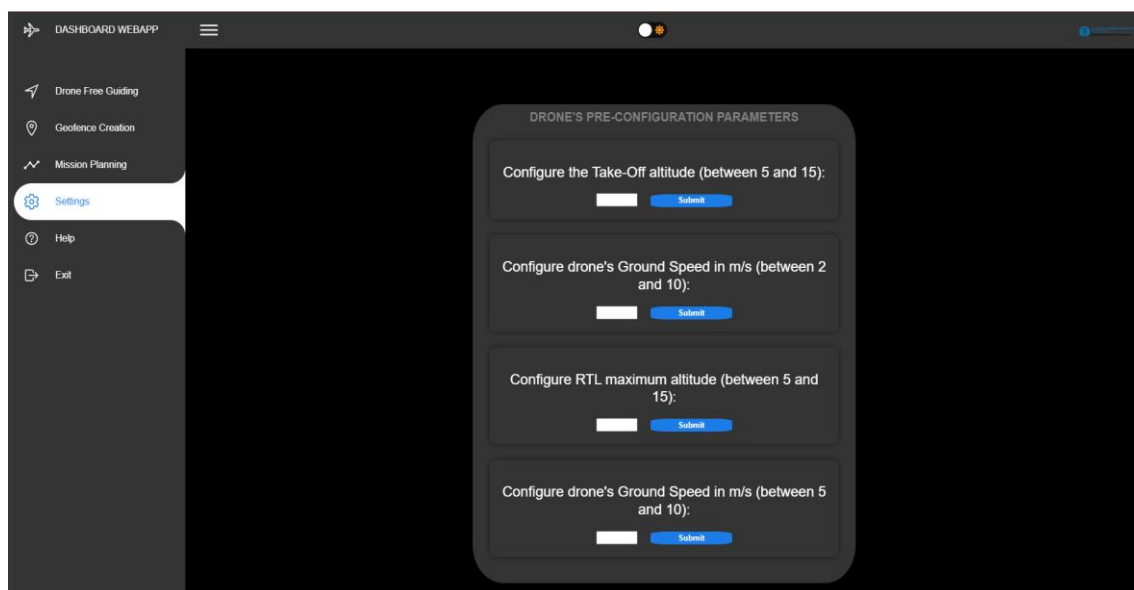


Fig. 3.6 *Settings.vue* component displayed on the browser

3.2 Python & Dronekit

Earlier on it was mentioned that all the on board modules were programmed using Python. Python is a versatile and widely used programming language known for its simplicity and readability. One of Python's key strengths is its extensive standard library, offering modules and packages that cover a wide range of tasks. Precisely, **Dronekit** is the library that it is used in the on board modules. Dronekit is a powerful open source toolkit designed to facilitate the development of applications and tools for UAVs. It provides a comprehensive set of tools, APIs, and resources that allow developers to interact with and control drones using the Python programming language. Some of the key features of this library are:

- Communication with drones
- Flight control
- Telemetry and monitoring
- Automation
- Mission planning
- Integration with external services

I dedicated my time to learn Python and Dronekit as all the on board modules were coded with them. Furthermore, the understanding of Python and Dronekit helped me to enhance and add some extra functionalities of the DashApp I was working on. As I integrated new features and capabilities into the DashApp, I had to integrate them on the autopilot module too as one of our goals was expanding the Dashboard's capabilities.

3.3 MQTT & Mosquitto : messaging protocol

On the one hand, the Message Queuing Telemetry Transport, usually referred as MQTT, is a lightweight and efficient communication protocol designed for exchanging messages between devices in a publish-subscribe pattern. MQTT is the most used messaging protocol among the Internet of Things as it operates on top of the TCP/IP protocol and is designed to minimize the overhead associated with traditional messaging protocols, making it suitable for resource constrained environments.

Mosquitto, on the other hand, is an open source MQTT broker implementation. An MQTT broker is a central server that acts as an intermediary for messages exchanged between MQTT clients (devices or applications). When an MQTT client wants to publish a message or subscribe to a topic, it communicates with the broker. The broker then handles the routing and delivery of messages between clients based on the topics they are interested in.

In summary, the relationship between MQTT and Mosquitto is that Mosquitto is a specific software implementation of an MQTT broker. It provides the infrastructure for MQTT communication, allowing devices and applications to send and receive messages using the MQTT protocol. Mosquitto enables seamless communication, message handling, and data exchange between MQTT clients in an IoT or messaging scenario.

The following is a workflow example of the publish-subscribe pattern:

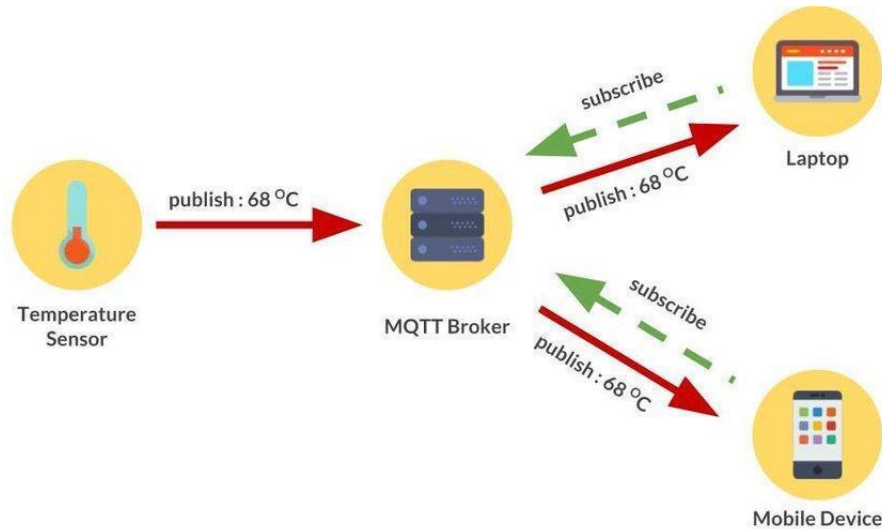


Fig. 3.7 Publish-Subscribe mechanism illustration (source: Google)

To understand the pattern, imagine a temperature sensor deployed in a room. This sensor measures the temperature and uses MQTT to communicate its data to both, the laptop and the mobile device. The laptop and the mobile device are both subscribers to the topic related to temperature data so they will receive the published messages.

To put it all together, the temperature sensor doesn't need to know who its specific receivers are. It just publishes data to a topic. The laptop and smartphone don't need to know where the data comes from; they just subscribe to topics they're interested in. The MQTT broker handles the matchmaking, making sure that data from the sensor reaches the devices that are subscribed to the relevant topic.

3.3.1 MQTT communication example in the DEE

To better understand, a more specific case is the one used in our ecosystem. In the DEE we use Mosquitto Brokers to facilitate the communication among the different modules. An example is shown in the **Fig.3.8** where the DashApp may subscribe to the topic "dashboard/autopilotService/telemetryInfo" in order to receive the telemetry information of the drone. Then if the autopilot module publishes a message in the broker with exactly this topic ("autopilotService/dashboard/telemetryInfo"), the Mosquitto Broker will occupy to send the message to the subscribed device, that in our case is the DashApp.

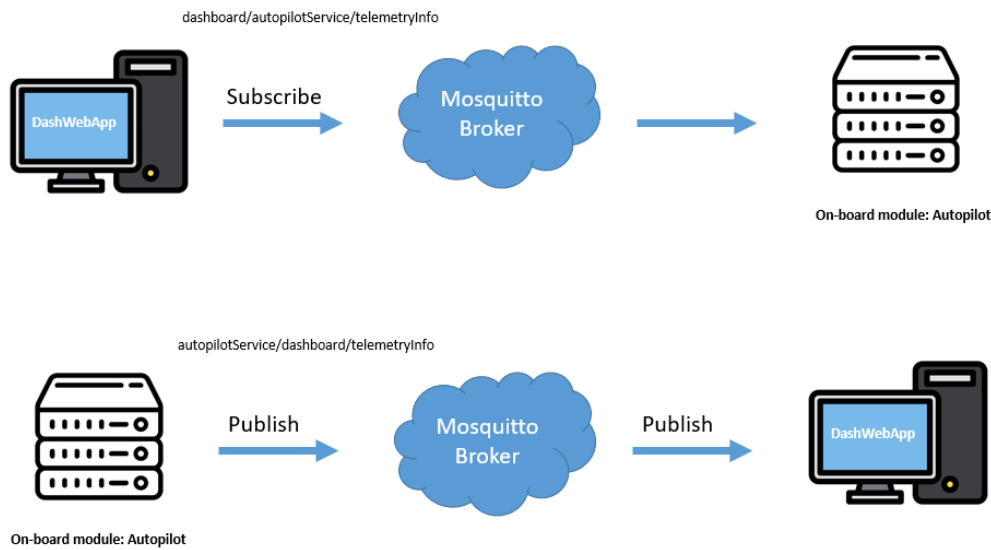


Fig. 3.8 DEE's publish-subscribe mechanism illustration

The topic convention used in the ecosystem for the publish-subscribe pattern is the following:

`name_of_the_origin_module/name_of_the_destination_module/command`

To fully understand all the parameters utilized in the MQTT workflow, you need to grasp the following aspects: Topic Definition, Subscribing, Publishing, and Delivering MQTT messages.

3.3.1.1 Topic Definition

A topic is a string identifier that represents a category or type of data. In this case, you might define a topic like “dashboard/autopilotService/telemetryInfo” to indicate that it's related to temperature readings from your home.

3.3.1.2 Subscribing

In the given example the DashApp wants to receive the telemetry data update from the autopilot module so it subscribes to the following topic:

`dashboard/autopilotService/telemetryInfo`

By subscribing, the web application tells the broker that he is interested in any messages published under the given topic.

3.3.1.3 Publishing

The autopilot module sends telemetry data periodically, concretely every 250 ms, so every 250 ms the modules “publishes” a message to the Mosquitto Broker. The message contains the telemetry information and is tagged with the topic:

`autopilotService/dashboard/telemetryInfo`

The MQTT broker receives the message and then forwards it to all subscribers who have expressed interest in the topic.

3.3.1.4 Delivery

Each time the autopilot publishes a new telemetry data reading with the topic, the MQTT broker forwards this message to the DashApp. The app receives the message, extracts the telemetry values, and displays it to the user.

3.4 MAVLink protocol

Micro Air Vehicle Link, better known as MAVLink, is a lightweight communication protocol that is widely used in the field of drone systems, particularly in the context of drones and ground based robotic vehicles. It's designed to facilitate communication between different components of a robotic system, enabling them to exchange commands, telemetry data, and other information. MAVLink is commonly used in drone simulations to simulate the interactions and communications that occur between various parts of a drone system. MAVLink serves as a communication bridge between the different components of a drone system, enabling them to exchange data and commands. In the context of drone simulation, MAVLink ensures that the communication patterns and interactions among components are accurately replicated, allowing for realistic testing and development without the need for physical hardware.

Particularly, in our case it has been used to create and receive Geofence messages from the autopilot module to the Mission Planner.

3.5 Mission Planner

Mission Planner has been the fundamental drone testing buddy throughout the whole DashApp implementation. Mission Planner is an open source ground control station software specifically designed for drone systems that adhere to the MAVLink communication protocol. Its versatility lies in its ability to bridge the gap between real world drone hardware and virtual environments, offering a range of functionalities tailored to simulation and testing.

3.6 Summary: the complete workflow

In summarizing this chapter, the ensuing illustration vividly demonstrates the interplay of the technologies outlined and explored throughout the discussion. Vue takes center stage as it was used to develop the dynamic web app, serving as the conductor of user interaction with the drone system. MQTT, represented as the intricate network of communication pathways, links the DashApp with the drone's on-board autopilot module, which was created using the Python programming language.

The communication and control culminates in the simulation realm, brought to life through the Mission Planner platform.

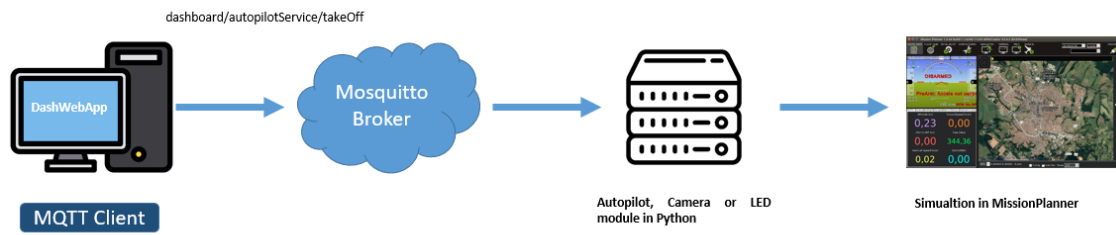


Fig. 3.9 Complete workflow of the technologies involved

CHAPTER 4: DASHAPP, FUNDAMENTAL FUNCTIONALITIES

In this chapter, we finally shift the gears from the theory section to the practical one as we eventually illustrate our DashApp's fundamental functionalities in their entirety. This entails delving into the core functionalities that define the existing Dashboard and their seamless adaptation into the web version.

So, this chapter immerses in the realm of the fundamental functionalities that form the core of the DashApp. These functionalities, pre-existing within the Python and Tkinter made Dashboard, have a bunch of interesting and essential features. From the freedom to guide the drone, streaming live video of what the drone sees, and designing simple flight plans, to the display of real-time telemetry data.

4.1 Basic functionalities

When first launching the DashApp the opening page that the user sees is the "entrance portal" to the app as shown in **Fig.4.1**. In other words, it presents an introductory page housing solely the "Connect" button, enabling users to access the Drone Engineering Ecosystem.

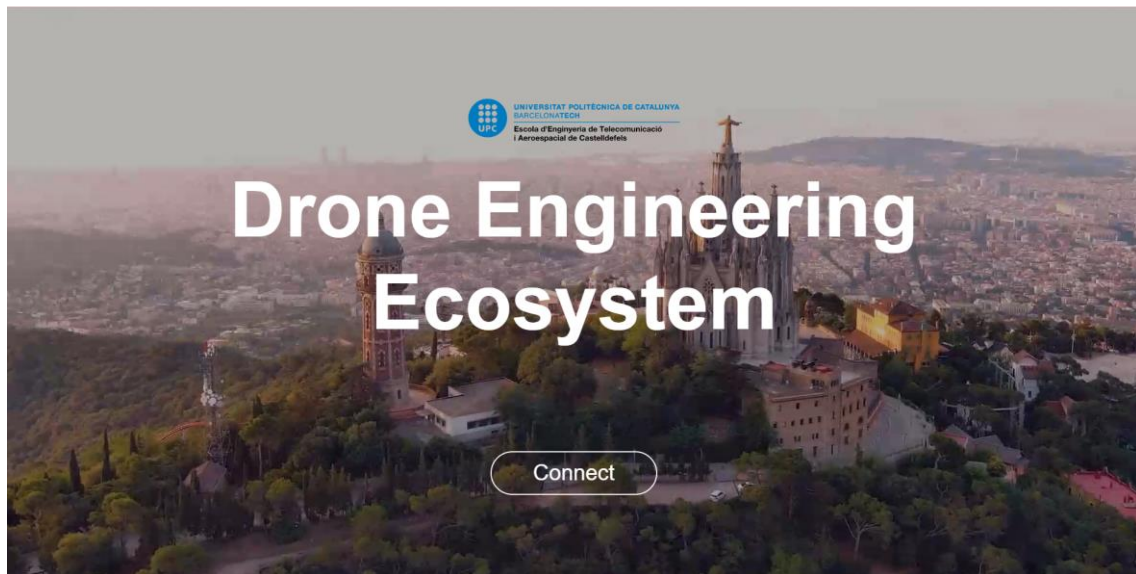


Fig. 4.1 Entrance portal to the Drone Engineering Ecosystem

Upon clicking the "Connect" button, a process is set into motion. The DashApp tries to establish a connection with the MQTT broker, an intermediary that facilitates communication between the app and the various DEE on-board modules. This connection is the bridge that allows your commands and requests to travel to the hardware, and the module's responses to find their way back to your screen. The outcome of the connection is displayed through SweetAlert pop-ups, which informs the user about either the failure or the success of the connection.

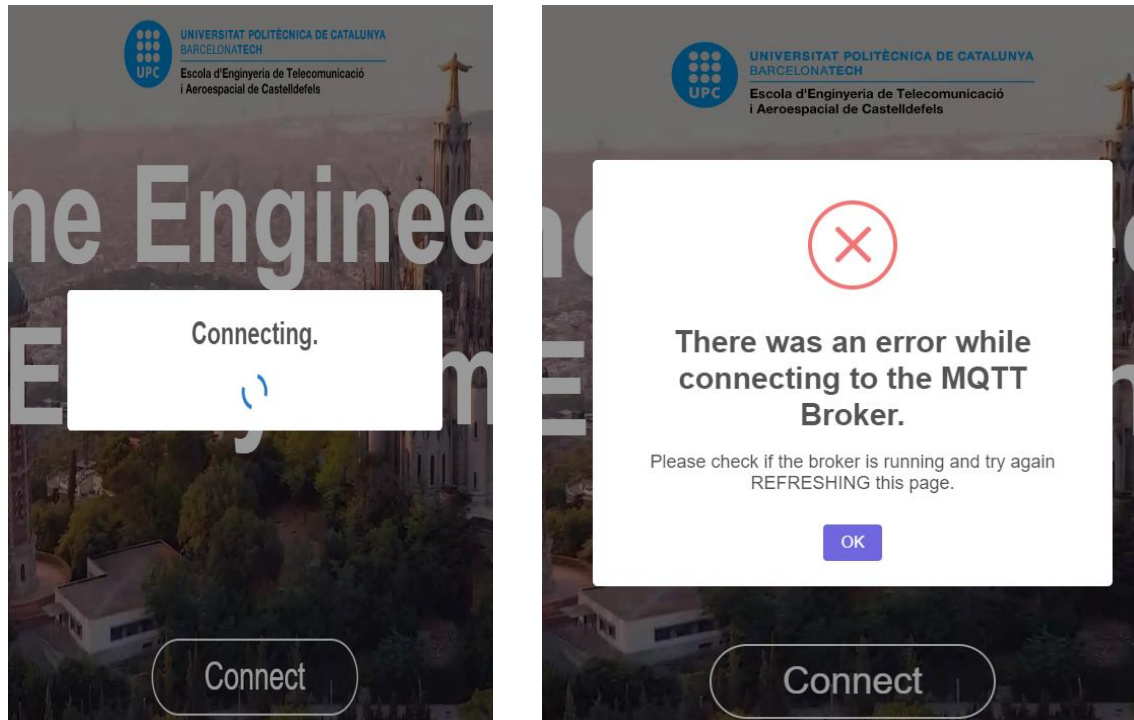


Fig. 4.2 SweetAlert pop-ups in case of connection failure

Once the connection has succeeded, the app unveils an additional preliminary component. This new element houses a series of interactive slides, each one presents the user with a distinct web app option among the available in the Drone Engineering Ecosystem to explore. Within these slides, the user is empowered to make a choice, selecting their desired destination within the ecosystem. Whether it is navigating with the Dashboard or delving into the captivating and funny world of the Drone Circus, these slides serve as a gateway, allowing users to direct their journey through the app's diverse offerings.

This approach holds a promising objective: **to integrate all the DEE's apps in their web edition into a unified platform.** This consolidation would provide a singular entry point to the entirety of the DEE's assortment of apps. As a hub for innovation and learning, this integrated platform will allow the future contributors to add their web apps on this unified platform.

Nevertheless we have to take into account that these slides represent an additional functionality within the app's interface. However, it is important to note that, currently, **the only active app available within this web interface is the DashApp.** While the other exciting applications like the Drone Circus are in the pipeline for development, so they are not accessible in their web edition yet.

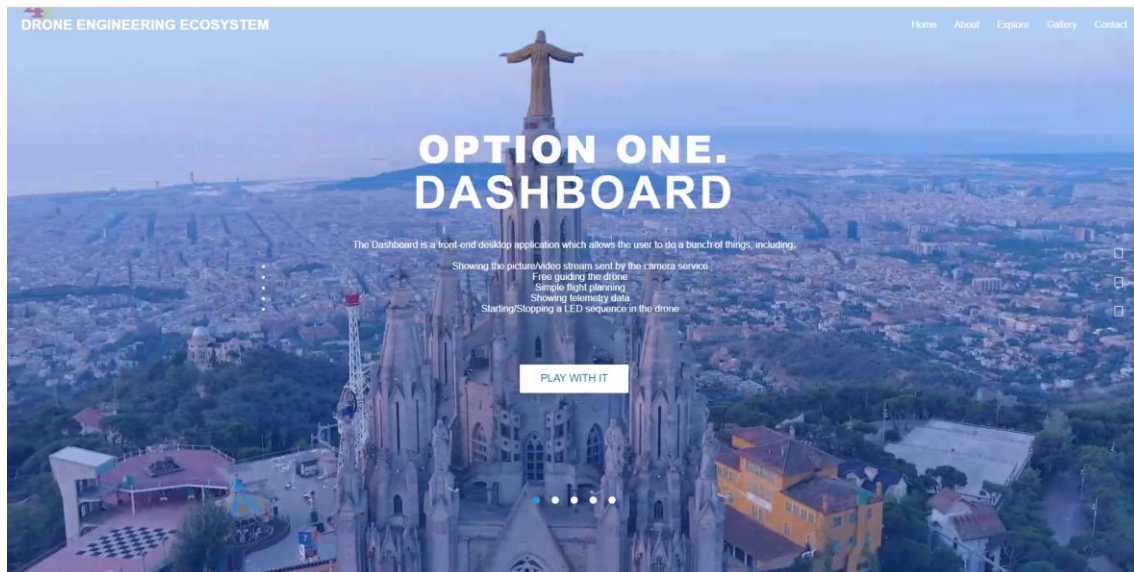


Fig. 4.3 Web app selection page

4.1.1 Drone control and monitoring page

In essence, the component above was another preliminary page featuring solely the entry point to the different apps, in our particular case affording the user access to the DashApp. And now, finally, we step into the DashApp itself as depicted in the figure below:

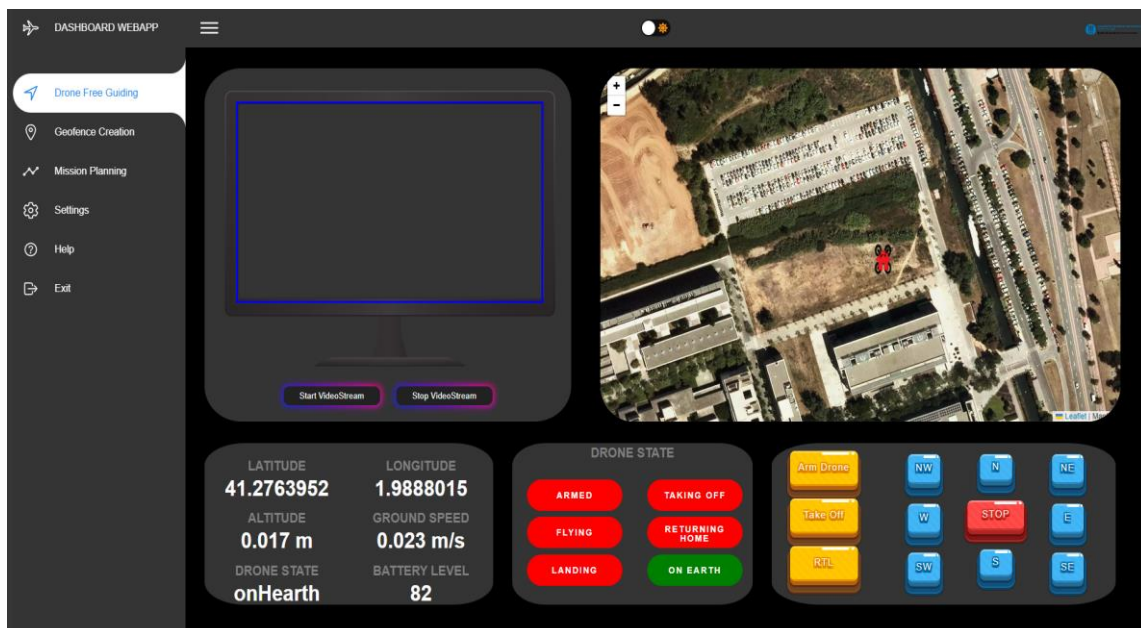


Fig. 4.4 DashApp's Dashboard window

As shown in the figure above, a prominent navigation bar graces the left side of the interface. This navigation bar serves as a menu gateway to a myriad of components, providing easy access to the diverse functions that the Dashboard offers to the user, such as:

- **Drone Free Guiding:** Unleash the drone's potential by assuming manual control, allowing you to guide it with precision and finesse.

-
- **Geofence Creation:** Define virtual boundaries, ensuring the drone's operations remain within designated areas for safety and compliance.
-
- **Mission Planning:** Craft intricate flight paths, orchestrating the drone's trajectory with a strategic layout that corresponds to your objectives.
-
- **Settings:** where the user has the ability to configure the drone's parameters precisely to the user's liking. The user is able to set essential parameters such as takeoff altitude, groundspeed, and more.

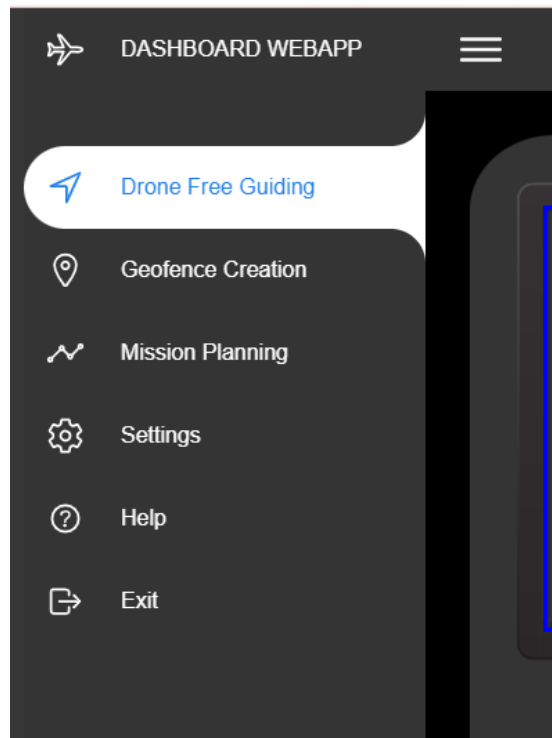


Fig. 4.5 Dashboard's navigation bar

Undoubtedly, the current page stands as the principal and central page of this web application, bearing paramount significance. Here, within the drone's control and monitoring domain, the user holds the reins of command. In this singular space, the user has granted the privilege of free guiding the drone, orchestrating its movements with precision. As the drone navigates, the telemetry data shows all the information in his respective card, keeping the user informed about the drone's vital metrics. Furthermore, a live video stream can be brought forth, providing the user with a window into the drone's perspective.

An important point to note is that prior to free guiding the drone, there exists the possibility to set the drone's parameters in the Settings' page. Here, you wield the power to configure some important parameters, from the initial takeoff altitude to the ground speed among others. In the same way the user is able to create an inclusion geofence in the "Geofence Creation" page.

By creating these restricted areas, the user ensures the drone's adherence to designated spaces, fostering both safety and compliance.

In essence, this page encapsulates a symphony of control, information, and strategic planning. It perfectly defines the Drone Engineering Ecosystem's control and monitoring interface, allowing the user to not just fly a drone, but to command an entire experience.

Moving forward, our focus shifts to the exploration of each individual section within the current page. Screenshots of the page will guide us through this comprehensive journey, unveiling every facet. My approach will involve a systematic progression, where the screenshots will illustrate each section from the top to the bottom and from the left to the right. By adhering to this layout, I aim to provide you with a clear and intuitive understanding of the individual "cards" that constitute the essence of this interface. From the map shown to the drone's commanding buttons, each segment will be unveiled in a comprehensive manner.

4.1.1.1 Video Stream

Displayed below is the Video Stream's card, within which, a monitor image takes center stage, ready to host the dynamic visual feed captured from the drone's perspective. Upon selecting the "Start VideoStream" button, the monitor will come to life, displaying the drone's live video stream.

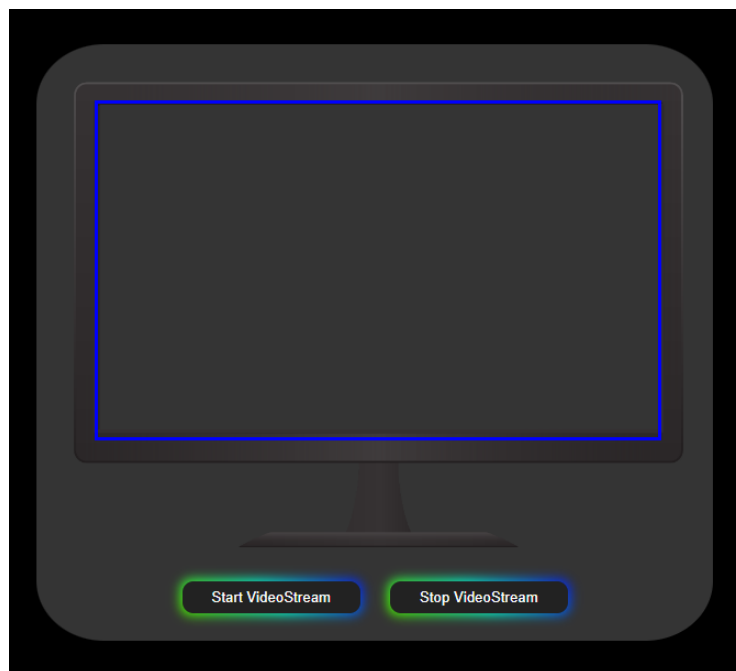


Fig. 4.6 Video Stream card inside the Drone Free Guiding component

A single click initiates this immersive experience, enabling the user to witness real time scenes as the drone navigates its surroundings. Conversely, the option to conclude the live stream lies in the "Stop VideoStream" button. With a mere tap, you can bring the broadcast to an end, relinquishing the visual connection between the drone and the user.

4.1.1.2 Map

Now, we shift our attention to the Map's card. Within it the user will encounter a drone and a home icon, both thoughtfully positioned on the Drone Lab's map. The home icon refers to the geographical point from which the drone will take off and in case of Return To Launch, will land. This card is designed to provide the user with a comprehensive visual representation of the drone's navigation journey.

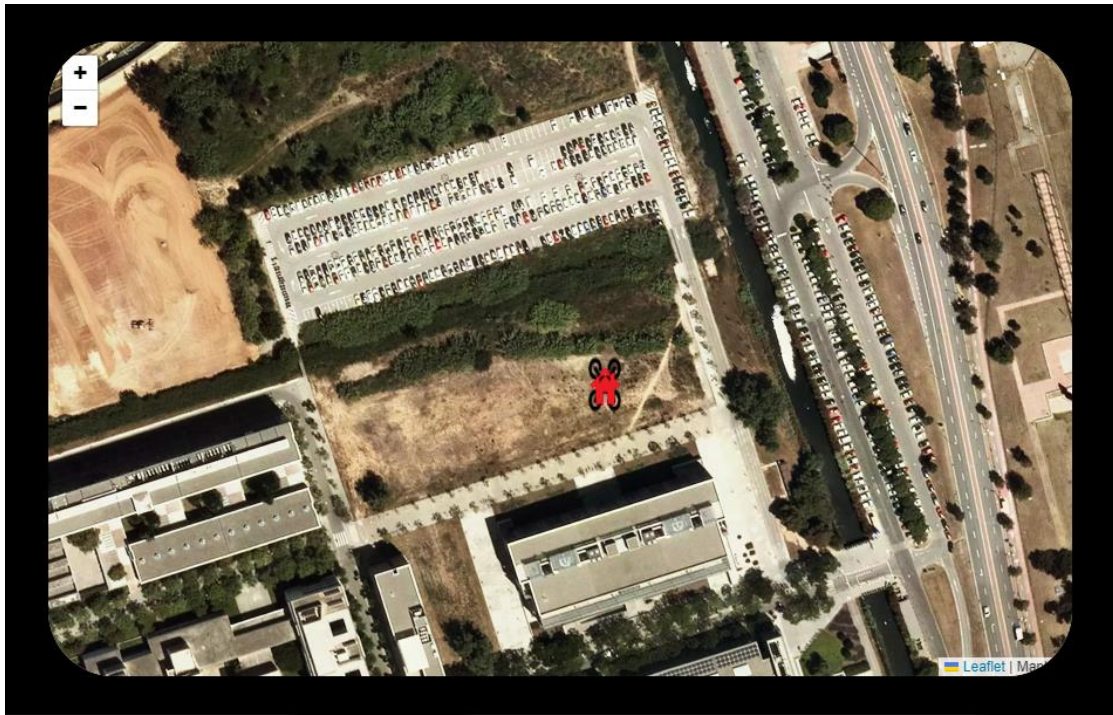


Fig. 4.7 Map card inside the Drone Free Guiding component

Upon this section of the Drone Free Guiding component, the user will be presented with a display as the drone icon starts its virtual flight. The drone will navigate through the map, moving with a ground speed configured to match the user's preferences. As the user guides the UAV, its icon will smoothly sail across the map, tracing a path that the user directed it to follow. By integrating this feature, I aim to enhance the user's understanding of the drone's geographical positioning and its interactions with the environment.

4.1.1.3 Telemetry information

Next up is the "Telemetry Info" card. This card serves as a hub for real time telemetry data, offering a comprehensive snapshot of the drone's crucial metrics as it navigates through the Map's card. This card is shown in **Fig.4.8**.

Within this section, the user will find a live stream of the drone's essential information, including the drone's latitude, longitude, altitude, ground speed, and more. As the drone takes its course across the map, these data points will be updated in real time, allowing you to stay attuned to its movements and current state.

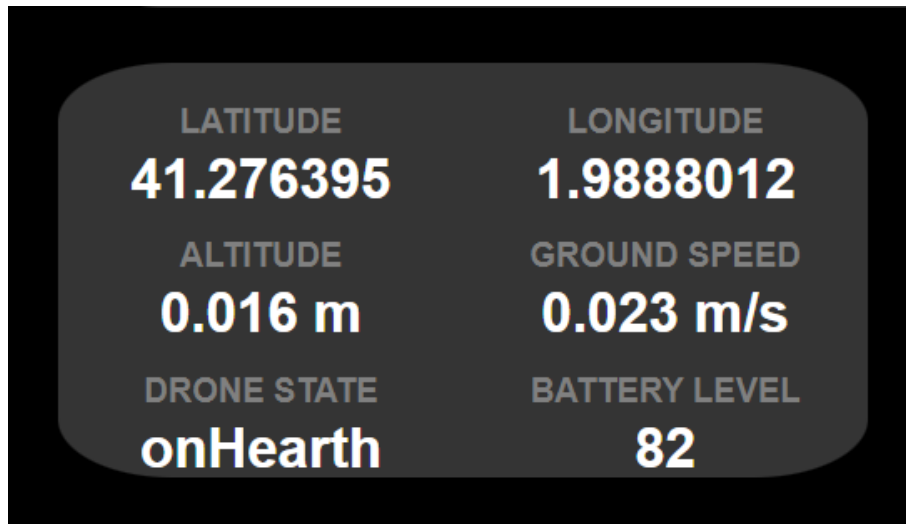


Fig. 4.8 Telemetry Information card inside the Drone Free Guiding component

4.1.1.4 Drone states

Alike to the Telemetry Information card, is the "Drone State" card as a parallel feature in the interface. This card serves as a visual representation of the drone's operational status, encapsulating its various states through a set of buttons. Each button, initially displayed in red, symbolizes a distinct state of the drone's journey. These states include "Armed," "Taking Off," "Flying," "Returning Home," "Landing," and "On Earth."

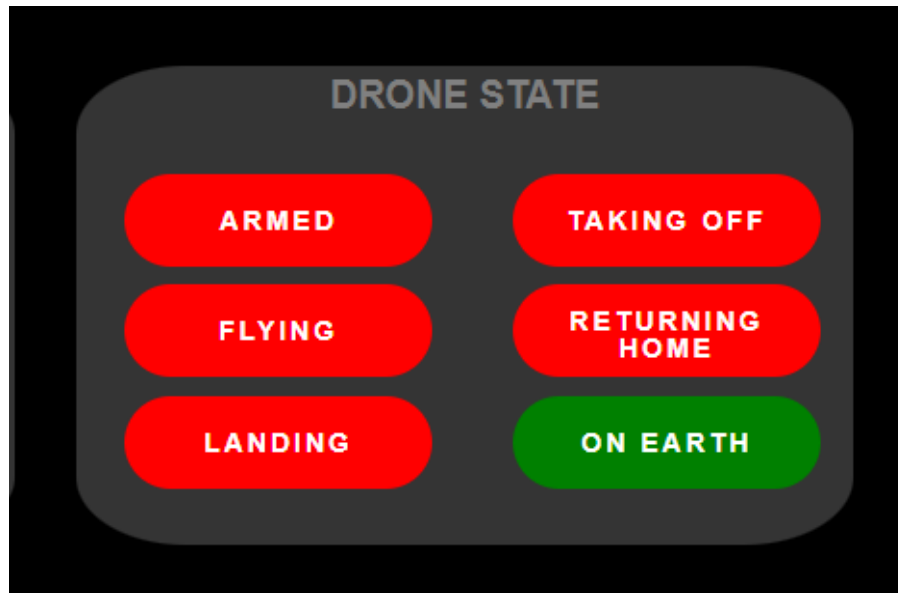


Fig. 4.9 Drone State card inside the Drone Free Guiding component

As the drone progresses through its flight, these buttons undergo a colour transformation, shifting from red to green, to signify the drone's successful transition into the corresponding state. This feature grants the user immediate insight into the drone's real time progress, allowing the user to track its movement and actions.

By offering this clear visual representation, I aim to provide the user with a simplified and accessible means of staying informed about the drone's activities.

4.1.1.5 Drone's controller

At last comes the most important card, the Drone's Controller card. This card holds a central role, as it serves as the command hub for orchestrating the drone's actions. Notably, this card plays a vital role in initiating actions that trigger subsequent responses within other cards, such as the Map, Telemetry Info and the Drone State cards.

The "Drone's Controller" card main objective is to send direct commands to the drone's on-board autopilot module. This card empowers the user with a range of commands at their disposal, each with its distinct purpose. These commands include essential actions such as:

- **Arm Drone:** Initiate the readiness of the drone, preparing it for operation.
- **Take Off:** Elevate the drone into the skies, starting its flight.
- **Return to Launch (RTL):** Command the drone to navigate back to its designated launch point.
- **Stop:** Terminate the drone's motion instantaneously.
- **Go North, North-East, South, etc:** specify directional commands to guide the drone's movement in the desired compass direction.

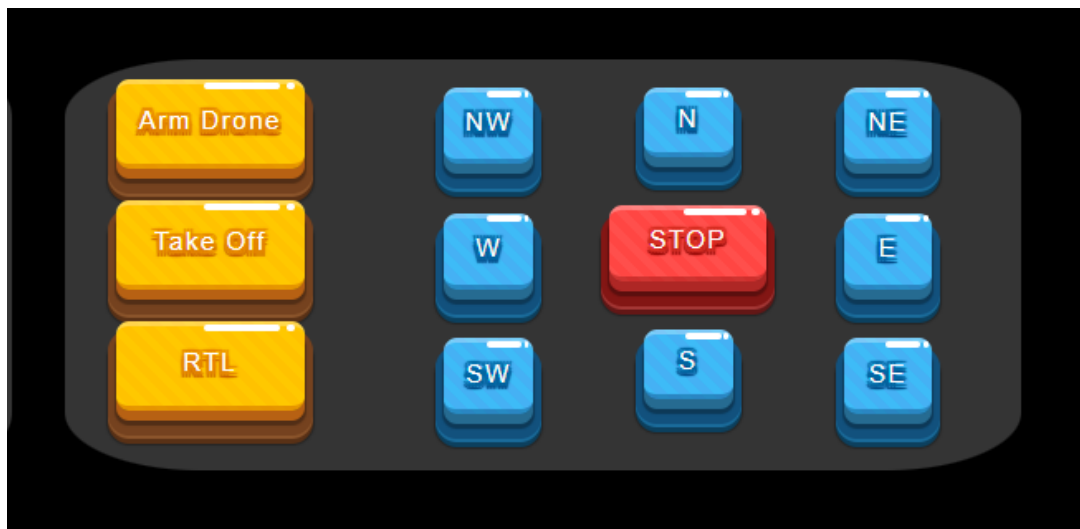


Fig. 4.10 Drone's Controller card inside the Drone Free Guiding component

4.1.2 Mission planning page

After exploring the drone's drone control and monitoring page in detail, the Mission Planning page operates in a similar manner but with a distinct objective. This component comprises four cards, each serving a specific purpose.

The first card features a map leaflet, which facilitates the design of a mission. Users can draw waypoints on the map, establishing a route for the drone to follow, with the added functionality of capturing pictures at selected waypoints.

Additionally, a telemetry information card is included to display real-time data about the drone's status as it follows the planned route.

Another card showcases the selected waypoints, and users can utilize radio buttons to determine whether the drone should take pictures at specific waypoints.

Concluding the page, a button card contains two buttons. The first button clears the waypoints drawn on the map, allowing users to refine their mission plan. The second button triggers the transmission of the mission plan to the autopilot module for execution. This design ensures that the Mission Planning component efficiently aids users in crafting and executing drone missions with enhanced precision.

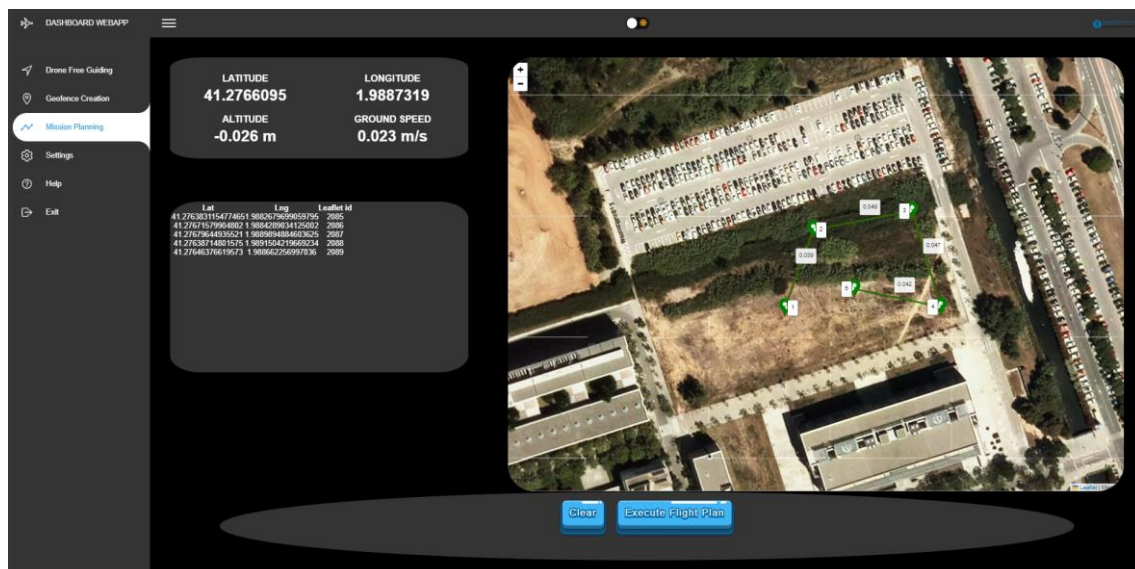


Fig. 4.11 DashApp's Mission Planning window

CHAPTER 5: DASHAPP, ADDITIONAL FUNCTIONALITIES

Moving on, the second phase of the DashApp's development was to implement the significant additional functionalities. For this purpose, this chapter unveils the interesting range of additional functionalities integrated into DashApp. These enhancements stem from meticulous development efforts, enhancing every facet of the user interaction with the dashboard. Among the numerous innovations, **the user will be able to create inclusion geofences** to establish virtual boundaries, **a configuration pathway to set drone parameters**, and **a list of other intuitive user interface functionalities** that make DashApp an evolved and updated Dashboard.

So, once having familiarized ourselves with the fundamental capabilities of the DashApp, it's now time to delve into its supplementary features, meticulously designed to elevate the Dashboard's utility and appeal. These additional functionalities extend the app's versatility, enriching the user's experience in many ways. Among the additional features, lie the following:

- Enhanced interface look
- Geofence creation
- Drone's critical parameters configuration

5.1 Enhanced interface

To initiate, let's first start with our focus set on the enhanced interface features. As mentioned many times before, the DashApp is designed to align with the latest trends in web application development, incorporating contemporary elements to elevate the user's experience. Among many features, one that has gained widespread popularity in modern web apps is the bright and dark mode feature. This versatile feature allows users to toggle between two distinct visual themes: bright mode, which presents a clean and well lit interface, and dark mode, which employs a darker colour scheme, reducing strain on the eyes in low light conditions.

The DashApp proudly incorporates the dark and bright mode feature. As you explore the interface, you will notice the presence of a top bar that houses essential elements, including the menu button and a toggle button in the middle dedicated to switching between bright and dark modes. This button serves as a dynamic switch, allowing users to seamlessly alternate between the two modes with a simple click.



Fig. 5.1 DashApp's top bar

This innovative functionality grants users the autonomy to align the app's appearance with their preferences. When this toggle button is activated, the app's appearance transforms seamlessly between dark and bright modes. This empowers users to select the mode that resonates most with their preferences and complements their work environment, ultimately fostering a personalized and comfortable interaction with the DashApp. The following figure illustrates the DashApp's when it is used in bright mode:



Fig. 5.2 DashApp's appearance in bright mode

5.2 Geofence creation

Moving on, the most important additional functionality that the app has is undoubtedly the inclusion geofence creation feature.

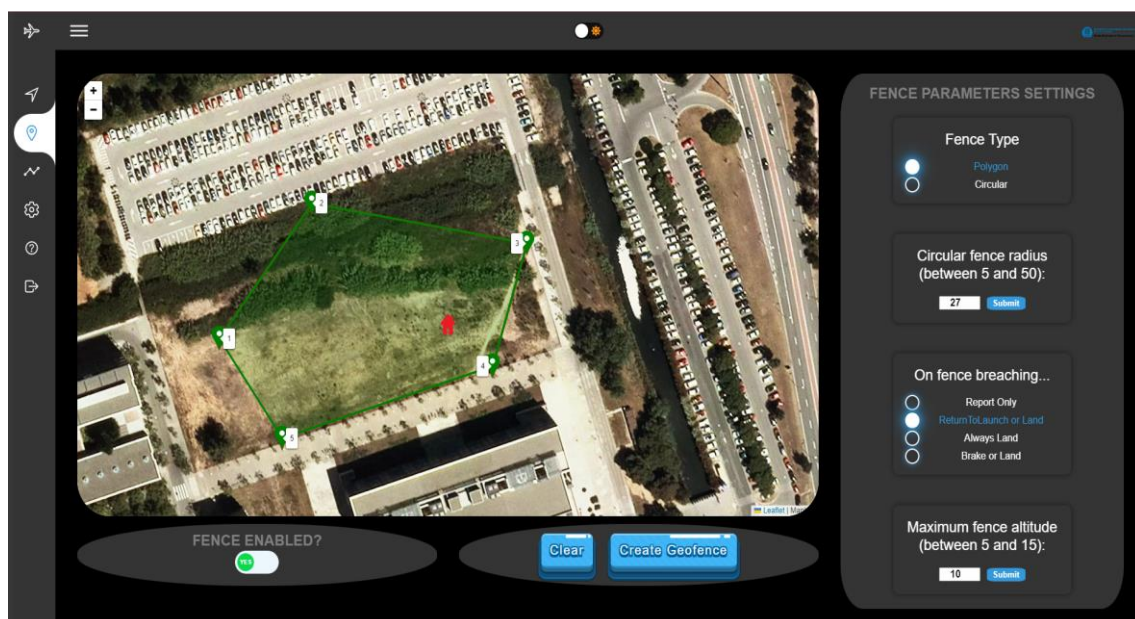


Fig. 5.3 Geofence creation component window

The Geofence Creation page is thoughtfully structured into three distinct cards, each serving a specific purpose. Let's walk through this systematic arrangement:

5.2.1 Map Card

At the forefront is the Map Leaflet card, designed to facilitate the creation of geofences. Within this card, you will find an interactive map interface where the user can outline the shape of the geofence. Whether you're defining a polygonal or circular geofence, this is where the groundwork is laid.



Fig. 5.4 Polygonal geofence



Fig. 5.5 Circular geofence

5.2.2 Fence's parameters configuration card

Following the map leaflet, the Fence Parameters Configuration card takes center stage. This card empowers the user to configure their geofence's behaviour. Here, you can set a range of parameters such as the fence type, be it circular or polygonal, in case it is circular, define its maximum radius, the maximum altitude the drone is allowed to reach within the fence, and the action you want the drone to take if it breaches the fence.

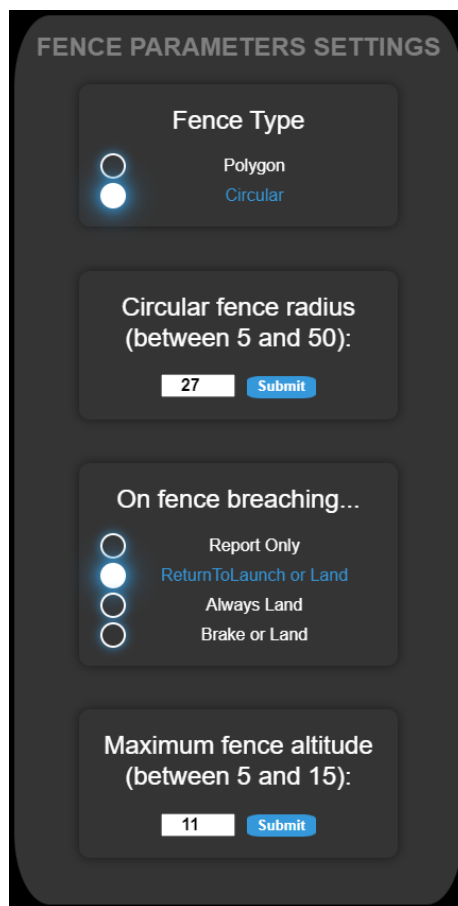


Fig. 5.6 Geofence settings

5.2.3 Buttons' card

The final cards introduce some key elements. Firstly, a toggle button grants you the ability to activate or deactivate the geofence as needed. Secondly, a distinct button empowers you to transmit the finalized geofence settings to the on-board autopilot module. Whereas there is also a button to clean the geofence drawn in the map in case the user wants to modify it.



Fig. 5.7 Geofence's buttons' card

Collectively, these three cards within the Geofence Creation page create an intuitive process for defining and configuring geofences. From drawing shapes to parameter settings and practical implementation, this comprehensive structure aims to empower the user with a robust geofencing solution tailored to your specific operational requirements.

5.3 Drone's parameters settings

In the subsequent component we will explore the Drone's flight parameters settings. Within this section, users are granted the capability to set critical flight parameters, ensuring a desired flight experience. Here, the focus centers on configuring parameters that shape the drone's behaviour throughout its flight:

1. **Take Off Altitude:** This parameter dictates the height at which the drone will ascend when initiating takeoff. It plays a pivotal role in determining the drone's initial flight altitude.
2. **Drone's Ground Speed:** Ground speed refers to the drone's horizontal speed as it moves across the terrain. By adjusting this parameter, users can set the drone's desired pace during flight.
3. **Return to Launch (RTL) Maximum Altitude:** In the context of Return to Launch, this parameter sets the maximum altitude the drone can reach while executing its return journey to the launch point.
4. **Return to Launch (RTL) Ground Speed:** This parameter controls the drone's speed during its Return to Launch operation, determining the pace at which it retraces its flight path back to its starting point.

It's important to note that users possess the flexibility to select which parameters to configure. There's no obligation to set all the available parameters, as users can choose to adjust only those that align with their flight requirements. The remaining parameters will retain default values, ensuring a seamless flight experience even when certain configurations are left untouched.

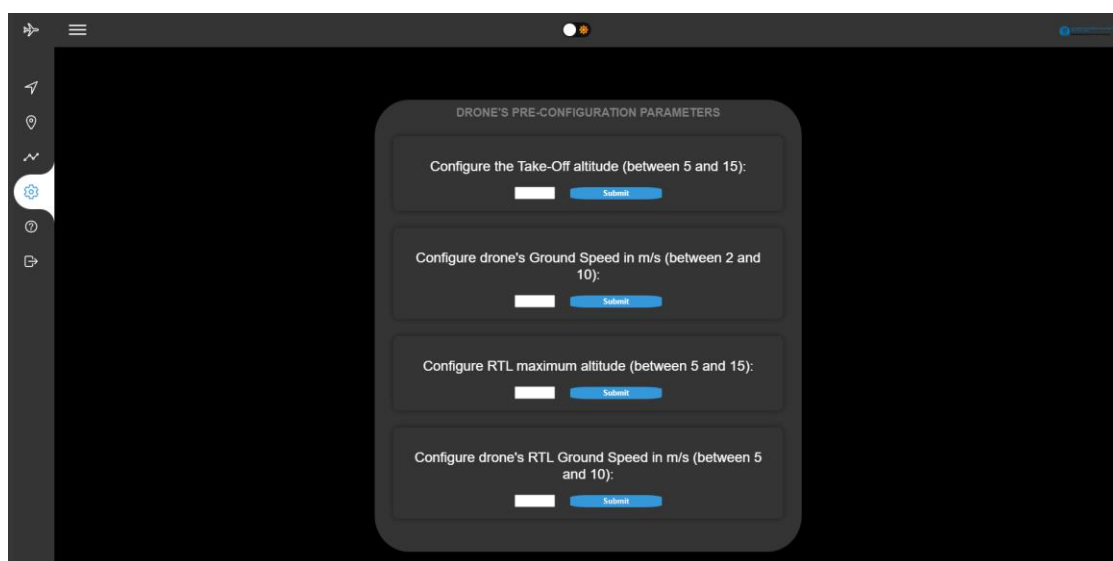


Fig. 5.8 . Settings component window

CHAPTER 6: DASHAPP'S VUE CODE STRUCTURE

Proceeding into the sixth chapter of the project, the focus turns toward an exploration of Vue's code structure and organizational framework. So, this chapter is dedicated to unravelling the intricate layers that compose the architecture of the DashApp, making clear on how its code components are structured, linked together, and meticulously organized to create an efficient application. We will navigate the various layers of Dashapp's codebase, examining the arrangement of files, directories, and modules that collectively contribute to its functionality.

In essence, this serves as a comprehensive guide to the inner workings of Dashapp's codebase, providing a clear understanding of how its structure and organization contribute to the overall functionality and robustness of the web application.

6.1 Code organization and basic files

As mentioned before, the DashApp is built using the Vue.js framework. Vue follows a specific file structure that helps organize the project into distinct parts, making development more manageable. The following figure shows how DashApp's directories and files are organized:

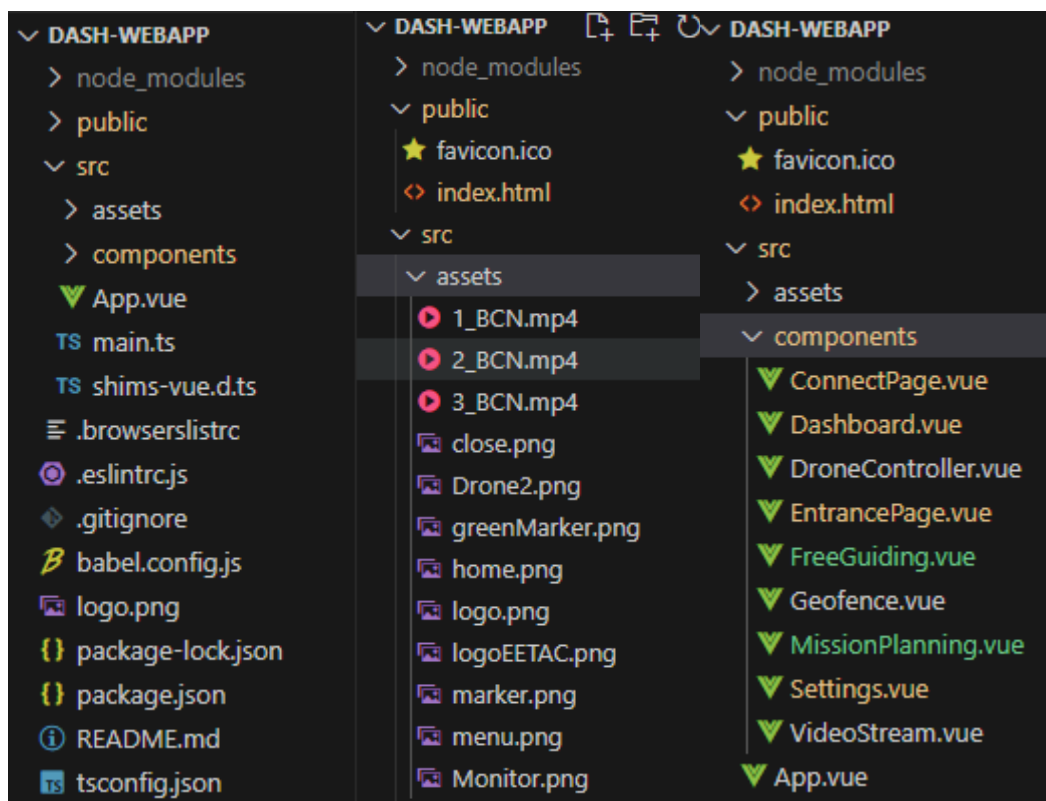


Fig. 6.1 File tree of the web application

The file structure of a Vue.js project is designed to keep the code organized and modular. Concretely, the "src" directory contains most of the application's logic, with subdirectories for assets and components. The "public" directory holds static assets and the main HTML file. The "node_modules" directory manages the project's dependencies. This separation of concerns makes it easier to maintain, develop, and scale a Vue.js application. Specifically the key directories and files are:

1. **node_modules:** This directory contains all the dependencies and libraries that the project relies on. It's usually generated and managed by a package manager like npm or Yarn.
2. **public:** The "public" directory is where you place static assets that you want to be publicly accessible, such as images, fonts, and the main HTML file (index.html).
3. **src:** The "src" directory is where most of the application code resides. It includes several important files and subdirectories, such as:
 - **assets:** This is where we place our static assets that need to be processed, like stylesheets, images, and fonts. Like shown on **Fig.6.1**, in the "assets" is where we have the images and videos used in the app.
 - **components:** In the "components" directory, we find Vue component files. Vue components are reusable building blocks that can be combined to create complex interfaces.
 - **App.vue:** This is the root Vue component of the application. It serves as a container for all other components and defines the overall layout and structure of your app. It usually contains the layout structure and potentially the navigation elements that persist across all pages. In our case this is not used.
 - **main.ts :** This is the entry point of the application. It initializes the Vue app by creating an instance of the Vue constructor and mounting it to an HTML element in the "index.html" file.
4. **index.html:** This is the main HTML file of your application. It serves as a template for your app and includes the mount point where your Vue app is rendered.

The diagram in **Fig.6.2** illustrates the hierarchical structure of the components within my web app. It outlines the relationship between parent and child components, providing a clear overview of the app's architecture.

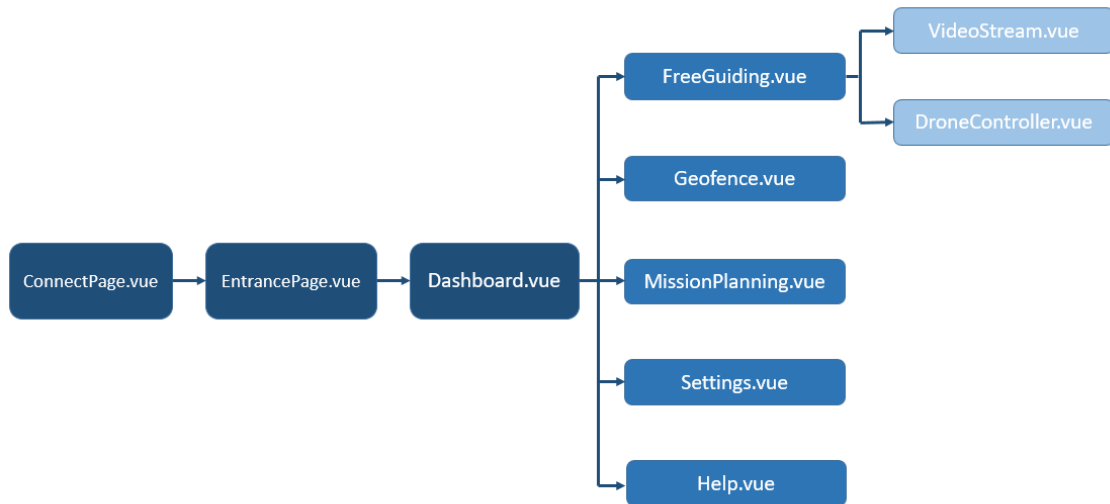


Fig. 6.2 Component tree of the web application

- **ConnectPage.vue:** serving as the initial entry point, this page establishes a connection with the MQTT Broker. If the connection is successful, users proceed to the next stage, otherwise they remain on this page.
- **EntrancePage.vue:** upon successful connection with the MQTT Broker, users are directed to the EntrancePage. Here, they have access to various available web apps, with Dashboard.vue being the current sole option.
- **Dashboard.vue (DashApp):** this is the core of the web app, the Dashboard itself, presenting users with a wide range of functionalities over drone control and monitoring. Within Dashboard.vue, various child components are integrated, each one forms a different window in the app:
 - **FreeGuiding.vue:** this component enables manual drone control and monitoring alongside with the live video stream from the drone's perspective. It further consists of two sub-components:
 - **VideoStream.vue:** displays live video streaming from the drone's perspective.
 - **DroneController.vue:** empowers users to send commands to the drone's autopilot module.
 - **Geofence.vue:** enables users to create virtual boundaries for the drone's operations.
 - **MissionPlanning.vue:** component that gives to the user the ability to design mission routes.

- **Settings.vue:** within this component, users are granted the ability to configure significant flight parameters. Here, the focus centers on setting parameters that shape the drone's behaviour throughout its flight.
- **Help.vue:** a component that includes DashApp's basic help information for the user

6.2 DashApp's entrance pages

As previously described, our web app presents users with an entry sequence, featuring the *ConnectPage* and *EntrancePage* as its preliminary stages. These pages serve as the gateway to the experience that lies within the app's core functionalities. Through this thoughtfully designed entry process, users are guided seamlessly from establishing a connection with the MQTT broker to having access to the web Dashboard itself.

Continuing along, we'll now provide an overview of how the codebase for these entry pages is structured and what specific functions they encompass.

6.2.1 ConnectPage

The *template*, *script* and *styles* element codes for the *ConnectPage* are attached to Annex A.

6.2.1.1 Template

Overall, this *ConnectPage* template sets up the visual layout for the initial page of the web app. It provides a background video, a navigation UPC logo, a title, and a "Connect" button. If the connection is successful, users are directed to the *EntrancePage* component; otherwise, they remain on the *ConnectPage* to retry the connection.

6.2.1.2 Script

The *script* element begins by importing various functionalities and components. These include utilities from Vue 3 like *defineComponent*, *ref*, and *onMounted*, which help in defining and managing components and their reactivity. The *provide* and *inject* functions allow data sharing between components, while *EntrancePage* is another component used within this script. Additionally, *Swal* from the *SweetAlert2* library is employed for displaying visually appealing pop-up messages, and the *mqtt* library provides MQTT related functionalities.

Moving on, the script defines the component behaviour within a setup function. Here, several reactive variables and functions are set up for use within the component's scope. Among these is the definition of a critical function named "toggle". This function is tied to the interaction triggered by clicking the "Connect" button within the app. This function dynamically switches the state of a variable, determining access to the "EntrancePage" component.

When the MQTT connection succeeds, the function toggles the variable, granting access. In case of a failed connection, it advises users, through a *SweetAlert* pop-up, to refresh the page for another attempt. This interaction strategy ensures seamless user engagement, enabling access to the "EntrancePage" upon successful MQTT connection or prompting a practical retry approach for unresolved connections.

6.2.1.2 Styles

The *styles* element within a Vue app is where the visual design and presentation aspects of a specific component are defined. It serves as the creative canvas for customizing how a particular component appears to users. Through CSS rules, properties, and styles, we can shape the component's layout, colors, typography, and overall aesthetic appeal.

For more detailed insights into how styles are utilized and defined in the *ConnectPage*, you can refer to the code provided in Annex A. The following figure illustrates *ConnectPage*'s window applying all three elements:

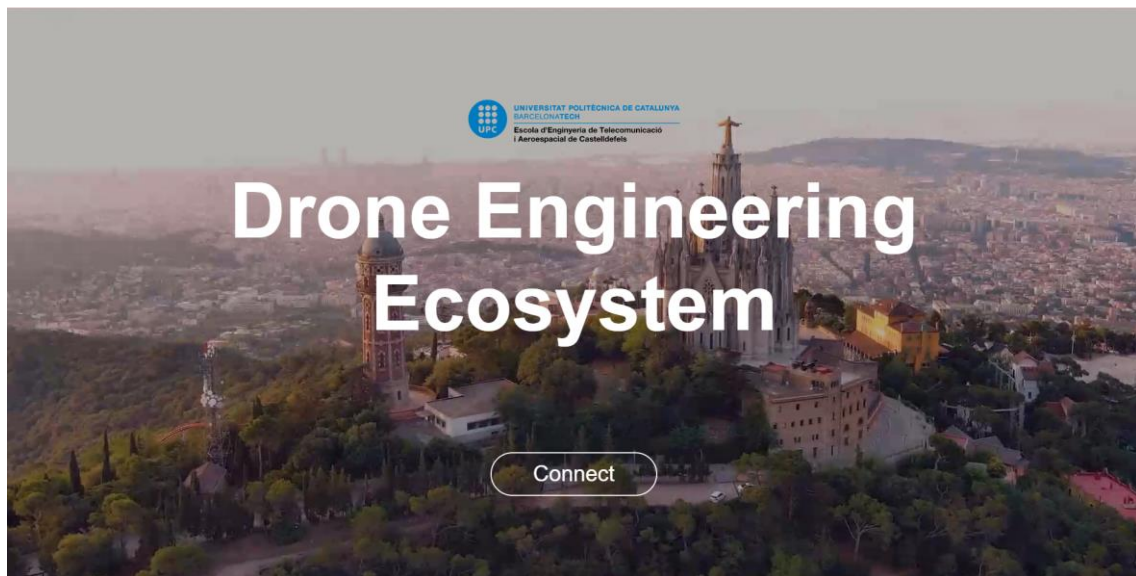


Fig. 6.3 ConnectPage's window view

6.2.2 EntrancePage

For better comprehension of the descriptions please check the *template*, *script* and *styles* elements code that is attached to Annex B.

6.2.2.1 Template

At the forefront, the template integrates the *Dashboard* component, displayed only when the *openDash* variable holds a truthy value. The "header" section introduces the app's identity, showcasing a brand name ("Drone Engineering Ecosystem") as a clickable link. A menu button and navigation menu follow, offering links to "Home," "About," "Explore," "Gallery," and "Contact".

Then, a section encompasses a dynamic display that cycles through video slides. Each slide is accompanied by corresponding content. The content includes titles and descriptions related to distinct features of the app's functionality. For instance, it describes the "Dashboard" component's features, such as camera streaming, drone control, flight planning, telemetry data display, and LED sequencing. Buttons are interspersed within the content. The template also includes navigation controls for the video slides, permitting users to navigate between different sections.

The concluding touch incorporates social media icons for GitHub, YouTube, and Twitter at the right of the page. These icons serve as direct links to relevant profiles and channels. In summary, this template not only presents information about the app's offerings but also facilitates user interaction through buttons and navigation controls.

6.2.2.2 Script

Progressing to the *script* element, it starts by importing essential dependencies like the MQTT library, Vue components and utilities. These are integral for enabling MQTT communication, defining components, and managing reactivity in the app.

The *script* defines the core component, including the *Dashboard* component as a nested part of the structure. The script then defines a couple of methods to handle user interactions. The *menuBTNClicked* function toggles the app's navigation menu's visibility. The *navBTNClicked* function orchestrates the visual navigation between different sections of the app based on user interaction with navigation buttons.

In the *setup* function, the script sets up reactive variables:

- *dashAccess*: Tracks whether access to the *Dashboard* is granted or not.
- *openDash*: Controls whether the *Dashboard* component is displayed or not.

The function *dashAccessPermission* is intended to be triggered upon user interaction. When invoked, it toggles the *dashAccess* and *openDash* variables, thereby allowing the *Dashboard* component to be accessed and displayed.

Together, these three elements merge into the interactive user experience captured in the next figure.

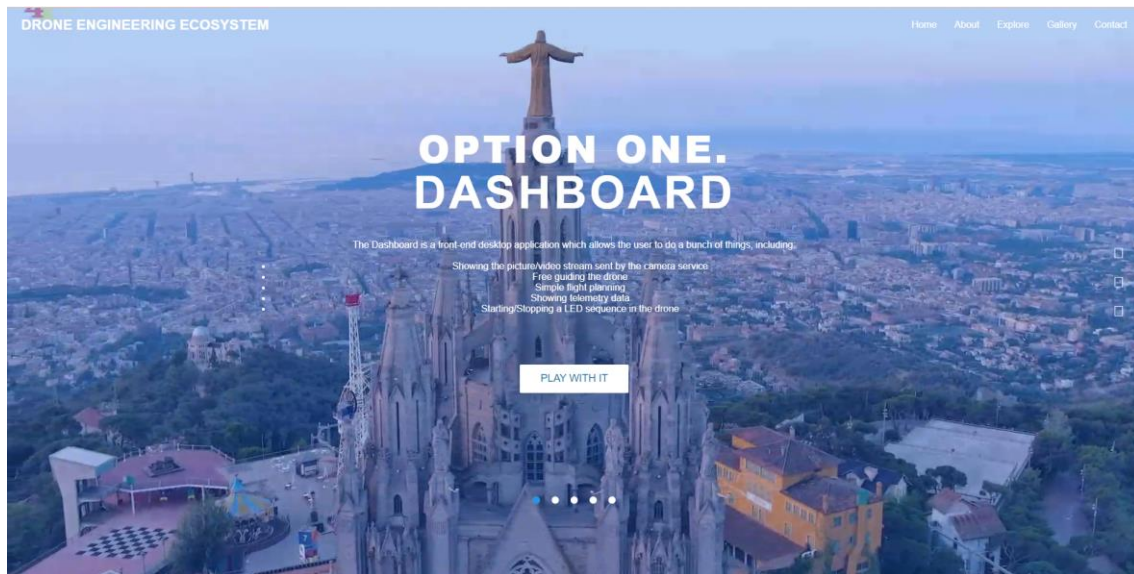


Fig. 6.4 EntrancePage's window view

6.3 DashApp's components

After exploring the preliminary entrance components that pave the way for accessing the heart of the application, the focus now shifts to the component known as the *Dashboard*. This component embodies the core functionality and features of the application, encapsulating a range of advanced tools and capabilities.

Unlike the previous components that served as introductions and provided glimpses into the DashApp itself, the *Dashboard* opens the door to drone control and monitoring. However, in this context, we will refrain from delving into an exhaustive code breakdown, as the intricacies of the *Dashboard* component's implementation are extensive. Rather than dissecting each line of code, we will focus on the broader essence and significance of the DashApp. For that reason, the styles applied to these components will not be described, as they are considered standard CSS code and adhere to the individual design specifications of each component. As such, these styles are not extensively detailed here, as they primarily determine the visual aesthetics and layout of the components.

The *Dashboard* component acts as a centralized component where users can wield a multitude of functionalities, including controlling drones, viewing live camera feeds, strategizing flight plans, accessing telemetry data. All this accessing Dashboard's child components. In the figure below, a vibrant red line outlines the window designated for the *Dashboard* component. This red-bordered area symbolizes the central hub where users select which DashApp functionality they want to use through the menu options on the left.

Adjacent to this red zone, a bold green line encircles another window. This red-bordered space signifies the area where child components will emerge. Child components, internally linked to the *Dashboard*, pop up within the green-bordered region, showcasing specific Dashboard functionalities such as:

- Drone Free Guiding
- Geofence Creation
- Mission Planning
- Settings
- Help

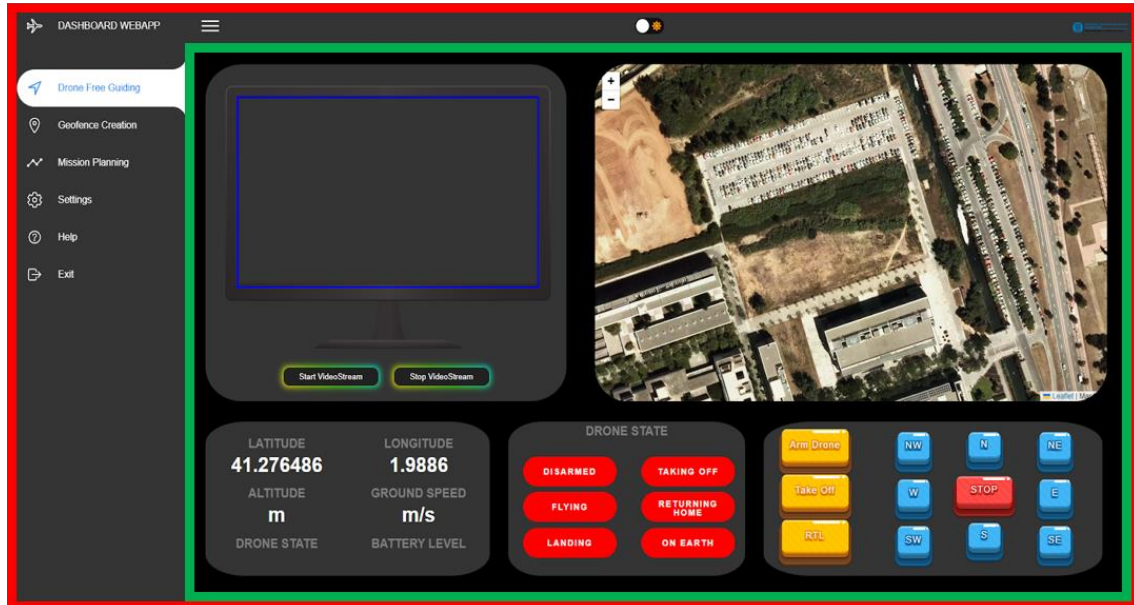


Fig. 6.5 Child components window within the parent component: Dashboard

6.3.1 Drone Free Guiding

For better comprehension of the descriptions please check the *template* and *script* elements code that is attached to Annex C.

6.3.1.1 Template

The template of the *Drone Free Guiding* component is designed with a clear and organized layout that partitions the screen into five distinct sections, each corresponding to a specific card. This deliberate segmentation ensures a structured and user-friendly display of information and functionalities. Within these sections, several key elements come into play, each contributing to the comprehensive user experience:

1. Video Stream and Drone Controller card: The *Drone Free Guiding* component incorporates dedicated spaces for both the Video Stream and the Drone Controller. These two cards operate as independent components, each fulfilling a unique role. The Video Stream card displays live video content, providing users with a visual feed from the drone's camera while the Drone Controller card offers controls for manoeuvring the drone, allowing users to interact with the drone's movements in real-time. The synergy between these components is evident as they seamlessly communicate with the on-board autopilot module via the MQTT broker, facilitating effective command transmission and video stream data reception.

2. **Map Card:** Another distinctive card within the component in question features a dynamic map display using Leaflet. This map card provides users with a geographical overview of the drone's location, enhancing situational awareness and aiding in navigation.
3. **Telemetry Info card:** The Telemetry Info card is designed to present real-time data to users. It comprises textual information that dynamically updates as data is received. This card serves as a valuable resource for users to monitor and track the drone's essential telemetry details.
4. **Drone State card:** The Drone State card introduces a set of buttons that are not only informative but also interactive. These buttons change colour in correspondence with the drone's state transitions. This allows users to quickly grasp the current state of the drone, contributing to a more intuitive and streamlined user experience.

As demonstrated in the visual representation in **Fig 6.5**, the red line highlights the area occupied by the parent component, the Drone Free Guiding. Within this space we have integrated the two of his child components: the green dotted card showcases the VideoStream component's display, while the adjacent blue dotted card hosts the DroneController component's functionality.

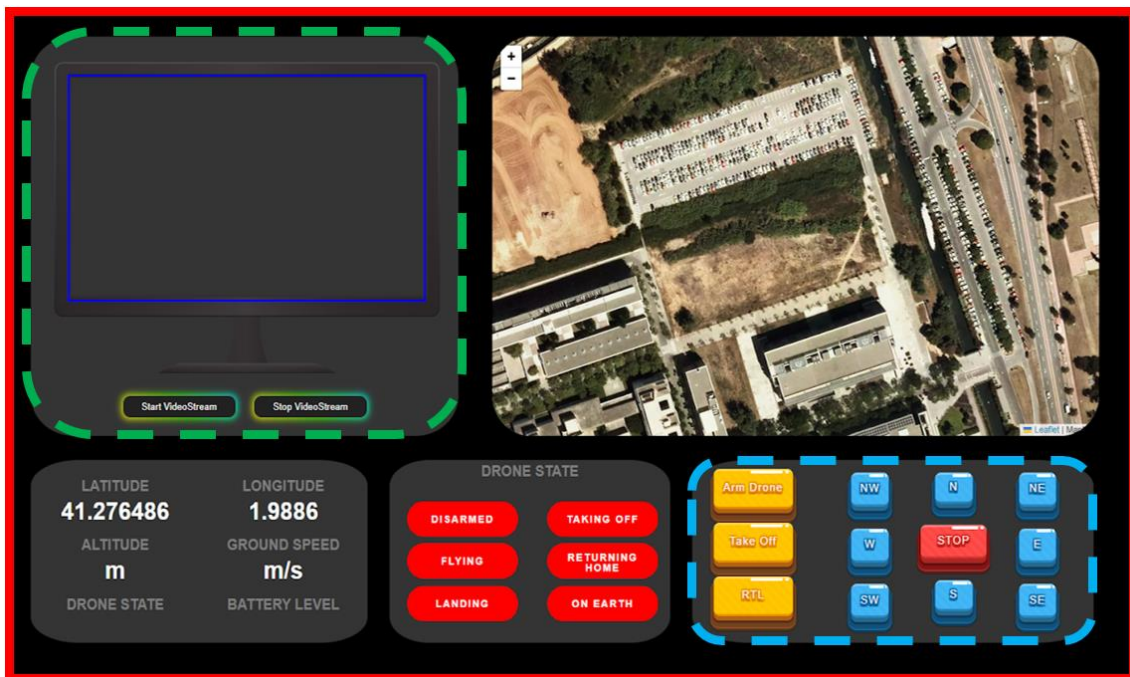


Fig. 6.6 Child components window within the parent component: Drone Free Guiding

6.3.1.2 Script

The script section of the current component is crucial for drone control as it houses a multitude of important functions and variables that significantly enhance its functionality.

At its core, the script leverages the *onMounted* function provided by the Vue framework. This function is a lifecycle hook that runs automatically when a component is mounted (added to the DOM). It's a convenient way to perform tasks, such as data initialization or setting up event listeners, once the component is ready to interact with.

The *onMounted* function enables the component to actively receive data from the MQTT broker, particularly when the data pertains to the "telemetryInfo" topic. By employing this function, the component promptly reacts to incoming messages and updates the reactive values tied to drone parameters. This real-time update ensures that telemetry information is instantaneously showcased in the Telemetry Info card, offering users an instant glimpse of drone data. Moreover, the data is used to pinpoint and reflect the drone's precise location on the map card. The function also undertakes the task of creating a dynamic map using Leaflet. This is pivotal in providing users with a visually engaging representation of the drone's geographical location.

As we progress in the *script* element, two fundamental functions, named "changeState" and "drawGeofence", further augment the component's capabilities. The "changeState" function plays an important role in modifying the appearance of the buttons within the Drone State card. It receives the current drone state as a parameter and based on this information, alters the colour of the buttons, visually signifying the drone's state to users.

On the other hand, the "drawGeofence" function holds the responsibility of rendering the geofence on the map. This becomes particularly relevant when the geofence is created and submitted through the Geofence Creation component. By receiving the necessary settings from the Geofence component through an *emitter*, this function seamlessly translates the geofence creation into a tangible visual representation on the map. An *emitter* is a mechanism for sending and receiving data between components. It allows components to communicate with each other by emitting events with data that other components can listen for and react to.

Summarising, the script section serves as the engine that drives the core functionalities of the component. By employing sophisticated interactions with MQTT data, dynamic map creation, and the execution of essential functions, this *script* empowers users to monitor, control, and visualise drone related data and actions in a seamless and intuitive manner.

6.3.2 Rest of the DashApp's components

The remaining components of the DashApp, including Drone Free Guiding, Geofence Creation, Mission Planning, and Settings, follow a similar structural pattern as discussed earlier. While each component has a unique purpose, the template, script, and styles of these components align with the fundamental framework established in the previous component descriptions. However, in the interest of avoiding redundancy and keeping this chapter concise, a detailed breakdown of these components will not be provided.

It's worth noting that these components adopt a similar organization and development approach, leveraging templates for user interface layout, scripts for functionalities, and styles for visual design. Each component addresses distinct aspects of the application, addressing specific functionalities and user interactions. By maintaining a consistent architecture across components, the DashApp ensures a cohesive user understanding of the code while accommodating diverse use cases.

The intention behind this approach is to present a comprehensive overview of the application's structure, allowing readers to grasp the design principles without delving into excessive detail. This way, the narrative maintains a dynamic pace and avoids overwhelming readers with repetitive information.

CHAPTER 7: AUTOPILOT'S ON-BOARD MODULE

Continuing with our work, the forthcoming chapter delves into the autopilot on-board module. This module, programmed in Python, serves as the heart of the drone's software infrastructure. In essence, the drone autopilot orchestrates and controls the drone's flight operations, ensuring stability and adherence to predefined flight paths.

In this context, the autopilot module has undergone many enhancements in order to work with DashApp. Specifically, I have **undertaken modifications into the autopilot module to integrate the geofence functionality alongside the feature of configuring various drone parameters**. Geofence functionality holds immense importance as it was first implemented in the DashApp in question and it establishes virtual boundaries, restricting the drone's flight within predefined areas for safety and regulatory compliance.

The autopilot's codebase is meticulously crafted using the DroneKit library, a Python toolkit tailored for drone communication and control. Through this library, our autopilot communicates with the drone, facilitates mission planning, and monitors telemetry data for precise navigation.

While the DEE encompasses other on-board modules like the Camera Service and LEDs' Service, our focus within this chapter remains firmly on the autopilot module. This choice stems from the important role it plays in implementing the geofence functionalities and expanding the drone's operational capabilities. The geofence feature's integration within the autopilot marks a significant milestone in achieving controlled and secure flight operations.

7.1 Autopilot's code structure

In order to better comprehend the subsequent functionalities integrated into this module, it's essential to first grasp the basic code structure of the autopilot. This will provide a crucial understanding to later explore the additional enhancements added to the module. By familiarizing ourselves with the core structure of the autopilot's codebase, we establish a clear starting point from which we can delve into the intricacies of the newly incorporated features.

At first, the autopilot module initiates its operation by importing several essential libraries. Among these, two libraries stand out: DroneKit and PahoMQTT. The inclusion of these libraries is significant, as they facilitate seamless communication between the autopilot and the local broker. DroneKit equips the module with the tools required for effective drone communication and control, enabling dynamic interaction with the drone's functionalities.

On the other hand, PahoMQTT empowers the module to establish communication with the local broker, enhancing the autopilot's ability to exchange information in real-time.

To establish a communication channel between the DashApp and the autopilot, a well-orchestrated sequence of actions takes place. This enables the autopilot to accurately interpret and execute commands received from the DashApp. Firstly, upon initialization, the autopilot employs the "AutopilotService" function, exemplified by **Fig.7.1**.

```
def AutopilotService(connection_mode, operation_mode, external_broker, username, password):
    global op_mode
    global external_client
    global internal_client
    global state

    state = 'disconnected'

    print('Connection mode: ', connection_mode)
    print('Operation mode: ', operation_mode)
    op_mode = operation_mode

    # The internal broker is always (global or local mode) at localhost:1884
    internal_broker_address = "localhost"
    internal_broker_port = 1884

    if connection_mode == 'global':
        external_broker_address = external_broker
    else:
        external_broker_address = 'localhost'

    print('External broker: ', external_broker_address)

    # the external broker must run always in port 8000
    external_broker_port = 8000

    external_client = mqtt.Client("Autopilot_external", transport="websockets")
    print('yo external client: ', external_client)
    if external_broker_address == 'classpip.upc.edu':
        external_client.username_pw_set(username, password)

    external_client.on_message = on_external_message
    external_client.connect(external_broker_address, external_broker_port)

    internal_client = mqtt.Client("Autopilot_internal")
    internal_client.on_message = on_internal_message
    internal_client.connect(internal_broker_address, internal_broker_port)

    print("Waiting...")
    external_client.subscribe("/autopilotService/#", 2)
    internal_client.subscribe("/autopilotService/#")
    internal_client.loop_start()
    internal_client.loop_start()
    external_client.loop_forever()
    external_client.loop_start()
```

Fig. 7.1 Autopilot module's *AutopilotService* function

This function serves as the one that sets the stage for receiving and responding to commands received by the DashApp. As part of its initialization, the function establishes connections and configurations based on specified parameters such as "connection_mode," "operation_mode," "external_broker," "username," and "password." These parameters are set on the script when running the autopilot module. In our case are the shown in the figure below:

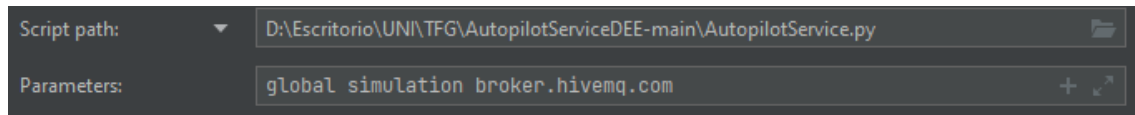


Fig. 7.2 Autopilot module's running configuration

In our case as we use the "broker.hivemq.com" as an external broker the "username" and "password" parameters are set to none. In the case we choose the "classpip.upc.edu" as an external broker then we must provide these additional parameters, the "username" and "password".

Moving on, the function's journey begins by defining global variables and determining the "operation_mode". The "external_broker_address" is configured based on the connection mode, whether "global" or "local" (in our case global). The function then creates both an external and an internal MQTT client. The external client, named "Autopilot_external," is tailored to connect via websockets, with optional authentication if the external broker's address requires it. This client is equipped to handle incoming messages and is connected to the external broker using the specified address and port.

Simultaneously, the internal client, named "Autopilot_internal," is configured to handle messages in a similar way. It connects to the internal broker at the defined address and port. Upon establishment, both clients are primed to listen for incoming messages, initiating their respective functions, "on_external_message" and "on_internal_message," as highlighted in **Fig.7.3**. These functions, designed to process incoming messages, are fundamental in the execution of commands.

```
def on_internal_message(client, userdata, message):
    global internal_client
    process_message(message, internal_client)

def on_external_message(client, userdata, message):
    global external_client
    process_message(message, external_client)
```

Fig. 7.3 Autopilot module's listening functions

Finally, the "AutopilotService" function subscribes to MQTT topics using both internal and external clients, allowing the autopilot to receive all the messages whose receiver is him.

Subsequently, as outlined in the previous figure, the incoming message is passed to the "process_message" function. This function, which stands as the coordinator, efficiently delegates the task to the relevant client (be it the internal or external client). This is achieved through the use of global variables, ensuring that the intended message recipient is properly identified.

Once the message's destination is established, the "process_message" function proceeds to solve the specifics of the message. As exemplified in the figure below, the function parses the message's content to determine the command's nature, such as "takeOff" or "returnToLaunch." Based on the command's identification, the function takes decisive action. For instance, if the command is "takeOff," the autopilot's response is immediate and calculated. The altitude data embedded within the message is extracted, and a designated thread initiates the takeoff procedure, leveraging the "take_off" function. This thread-based approach ensures parallel execution, optimizing the drone's responsiveness.

```
def process_message(message, client): # AQUI ES DONDE SE CONECTA CON EL SIMULADOR DE MISSION PLANNER
    global vehicle
    global direction
    global go
    global sending_telemetry_info
    global sending_topic
    global op_mode
    global sending_topic
    global state

    splited = message.topic.split("/")
    origin = splited[0]
    command = splited[2]
    sending_topic = "autopilotService/" + origin

    if command == "takeOff":
        altitude = int(message.payload.decode("utf-8")) # Lo puse yo
        print(altitude)
        state = 'takingOff'
        # w = threading.Thread(target=take_off, args=[5, ]) ----- antes de mis cambios
        w = threading.Thread(target=take_off, args=[altitude, "true"])
        w.start()

    if command == "returnToLaunch":
        # stop the process of getting positions
        vehicle.mode = dronekit.VehicleMode("RTL")
        state = 'returningHome'
        direction = "RTL"
        go = True
        w = threading.Thread(target=returning)
        w.start()
```

Fig. 7.4 Function that processes the message received from the DashApp

Below is given a comprehensive table detailing the complete list of commands that the Dashboard had the capability to send to the autopilot module before my contributions. These commands provide instructions to the drone, shaping its behaviour and actions.

Command	Description	Payload
connect	Connect with the simulator or the flight controller depending on the operation mode	No
disconnect	Disconnect from the simulator or the flight controller depending on the operation mode	No
armDrone	Arms the drone	No
takeOff	Get the drone take off to reach the default altitude	No
returnToLaunch	Go to launch position	No
land	Land the drone	No
disarmDrone	Disarm the drone	No
go	Move in certain direction	"North", "South", "East", "West", "NorthWest", "NorthEast", "SouthWest", "SouthEast"
executeFlightPlan	Execute the flight plan	A json object specifying the flight plan.

Table 7.1. Commands that the autopilot had before my contribution.

In essence, this meticulously orchestrated interplay of these functions establishes a robust framework for command reception, interpretation, and execution.

7.2 Contribution to the autopilot module: geofence creation and drone's parameter settings

Once seen the working methodology of the autopilot module, let's now delve into the functionalities that I added to this specific on-board module in order to enhance and make it compatible with the DashApp additional features.

With my contribution, these are the new commands and functionalities that the autopilot can manoeuvre with :

Command	Description	Payload
circularGeofence	Create a circular geofence with the center point on the home location and radius sent in the payload.	A json object specifying the circular fence and his configurations.
polygonalGeofence	Create a polygonal geofence with the json object sent in the payload. This object carries the vertices of the fence alongside the fence settings as: fence maximum altitude, what to do when breaching the fence, etc.	A json object specifying the polygonal fence and his configurations.
fenceEnable	Enable or disable the fence functionality	“0” or “1” (0 is fence disabled and 1 is fence enabled)
setGroundSpeed	Configure drone’s ground speed	The ground speed value
setRTLAltitude	Configure RTL’s altitude	The RTL altitude
setRTLSpeed	Configure drone’s RTL ground speed	The RTL’s ground speed value
takeOff	Take off the drone to the altitude given in the payload	Altitude to reach

Table 7.2. Additional commands that the autopilot has after my contribution.

7.2.1 Drone’s parameters configuration

Let's begin with the simple functionalities first, starting with the drone's parameter configuration functions within the autopilot module. This will provide us with an understanding of how drone's parameters can be configured and managed within the autopilot system.

As explained before, the “process_message” function is the one assigned to treat the incoming command from the DashApp and call the specified function once it has been clarified the received command. An example of the command sent by the DashApp could be:

```
client.publish("dashBoard/autopilotService/setGroundSpeed", groundSpeed.value);
```

Fig. 7.5 Parameter configuring command sent by the DashApp

Here we see a MQTT protocol message being sent by defining the:

- **Transmitter module:** dashBoard (DashApp)
- **Receiver module:** autopilotService (Autopilot on-board module)
- **Command:** setGroundSpeed
- **Message payload:** groundSpeed.value (the number to set)

The following snippet of this function shows how are being processed the commands of drone's parameters settings sent by the DashApp.

```
if command == "setGroundSpeed":
    gSpeed = int(message.payload.decode("utf-8"))
    print("GroundSpeed to set: ", gSpeed)
    w = threading.Thread(target=set_ground_speed, args=[gSpeed])
    w.start()

if command == "setRTLAltitude":
    RTL_Altitude = int(message.payload.decode("utf-8"))
    print("GroundSpeed to set (cm): ", RTL_Altitude)
    w = threading.Thread(target=set_RTL_Altitude, args=[RTL_Altitude])
    w.start()

if command == "setRTLSpeed":
    RTL_Speed = int(message.payload.decode("utf-8"))
    print("GroundSpeed to set (cm/s): ", RTL_Speed)
    w = threading.Thread(target=set_RTL_Speed, args=[RTL_Speed])
    w.start()
```

Fig. 7.6 Drone's parameters configuration message being processed

Certainly, let's dive into the code snippet provided. As mentioned, this code is nested within the “process_message” function, which is responsible for handling incoming messages and triggering relevant actions within the autopilot module.

If the received command is "setGroundSpeed," the code extracts the payload of the message to obtain the desired ground speed value. This value is then converted to an integer using “int(message.payload.decode("utf-8"))”. The ground speed represents the speed at which the drone moves along the ground. The extracted value is printed for verification purposes.

Subsequently, a new thread “w” is created to execute the “set_ground_speed” function, passing the extracted ground speed as an argument. This function is designed to set the drone's ground speed to the specified value:

```
def set_ground_speed(ground_speed):  
    global speed  
  
    print("Entra en la función de SET GROUND SPEED")  
    speed = ground_speed
```

Fig. 7.7 Function to set the drone's ground speed

Similarly, if the command received is "setRTLAltitude," the payload of the message is extracted to retrieve the desired return-to-launch (RTL) altitude. The extracted value is converted to an integer. The RTL altitude refers to the height above the takeoff point where the drone will return in case of a "Return to Launch" command. The extracted value is printed, and a thread (w) is created to execute the “set_RTL_Altitude” function, passing the RTL altitude as an argument. This function is responsible for configuring the RTL altitude of the drone:

```
def set_RTL_Altitude(RTL_Altitude):  
    global vehicle  
  
    print("Entra en la función de RTL Altitude")  
    vehicle.parameters['RTL_ALT'] = RTL_Altitude
```

Fig. 7.8 Function to set the drone's RTL altitude

Lastly, with the same *modus operandi*, if the command received is "setRTLSpeed," the code follows a similar pattern. The payload is extracted to obtain the desired RTL speed. The extracted value, representing the speed at which the drone returns to its launch point, is printed. The same thread is initiated to execute the “set_RTL_Speed” function, with the RTL speed value as an argument. The “set_RTL_Speed” function adjusts the RTL speed configuration of the drone:

```
def set_RTL_Speed(RTL_Speed):  
    global vehicle  
  
    print("Entra en la función de RTL Speed")  
    vehicle.parameters['RTL_SPEED'] = RTL_Speed
```

Fig. 7.9 Function to set the drone's RTL ground speed

In essence, this code snippet enables the autopilot module to respond to specific commands from the DashApp by configuring various parameters of the drone.

The utilization of threading allows these configuration changes to occur asynchronously, ensuring smooth and efficient execution without disrupting other processes.

7.2.2 Geofence functionality

In a similar way, the autopilot module handles geofence commands that are received and processed based on the type of specified geofence. This differentiation occurs between circular and polygonal geofences, as illustrated in the code below:

```
if command == 'circularGeofence':
    circular_configuration = str(message.payload.decode("utf-8"))
    x = threading.Thread(target=create_circular_geofence, args=[circular_configuration])
    x.start()

if command == 'polygonalGeofence':
    polygonal_configuration = str(message.payload.decode("utf-8"))
    x = threading.Thread(target=create_polygonal_geofence, args=[polygonal_configuration])
    x.start()
```

Fig. 7.10 Geofence commands processing

In the particular case of the geofence messages, when the autopilot module receives the command "circularGeofence," the payload of the message is extracted, containing JSON-encoded data that defines the configuration of the circular geofence. This information is converted to a string using "str(message.payload.decode("utf-8"))". Subsequently, a new thread (x) is initiated to execute the "create_circular_geofence" function. This function is designed to create a circular geofence based on the provided configuration JSON. The same process happens with the polygonal geofence type.

By segregating geofence commands into "circularGeofence" and "polygonalGeofence," the autopilot module can distinguish between the two types of geofences and initiate the respective functions to create the appropriate geofence shape. This approach allows the autopilot to dynamically respond to commands from the DashApp and implement geofences of different shapes to ensure safe drone operations within designated areas. Threading facilitates the concurrent execution of geofence creation processes, contributing to the efficiency of the autopilot's operations.

7.2.2.1 Geofence functionality's code structure

For better comprehension, please check the geofence's code that is attached to Annex D.

The geofence functionality offered in the autopilot module is designed to create an **inclusion geofence**. This type of geofence defines a virtual boundary within which the drone is allowed to operate.

If the drone crosses this boundary, predefined actions such as returning to the launch point (RTL) can be triggered. In essence, the inclusion geofence confines the drone's movements within a specified area to ensure its safe operation and prevent it from venturing into restricted zones. For example, in the following figure the drone only can fly in the green area:



Fig. 7.11 Inclusion geofence

While the current implementation focuses on the inclusion geofence, the initial intention was to also include an exclusion geofence functionality too. An exclusion geofence would work in the opposite manner (it would define areas where the drone is explicitly prohibited from entering). This is particularly useful for marking off areas such as no-fly zones or sensitive locations.

Regrettably, due to certain constraints or challenges, the implementation of the exclusion geofence wasn't achievable within the current scope of the project. However, this presents an exciting opportunity for future contributors to the Drone Engineering Ecosystem. One potential proposal is the development and integration of the exclusion geofence functionality. This enhancement would further enhance the drone's safety and versatility, enabling it to avoid specific zones that should be off-limits for its operations. Such a contribution would be valuable for expanding the capabilities of the drone.

Now let's delve into the methodology behind the inclusion geofence creation. Remember that the code is attached to the Annex D.

The code attached, is responsible for processing and implementing the creation of a polygonal inclusion geofence within the autopilot module. Here's an overview of how the code works:

1. **Initialization and data parsing** : The function "create_polygonal_geofence" receives a JSON string containing information about the polygonal geofence. This information includes fence type, action, maximum altitude, radius, and a list of vertices of the geofence polygon created by the user in the DashApp.

2. **Setting geofence parameters:** The code extracts relevant information from the JSON data and sets various geofence parameters within the vehicle's autopilot. These parameters include fence type, action (e.g., Return To Launch), maximum altitude, etc.
3. **Vertex Processing:** Subsequently the function processes the list of vertices received in the JSON data. It iterates through each vertex, extracting latitude and longitude values, and adds them to a "fence_list" that will define the geofence boundary.
4. **Closing the Geofence:** To close the polygon, the code adds the coordinates of the first vertex to the "fence_list" as this is the home location..
5. **MAVLink Messaging:** The code uses MAVLink messaging to communicate with the drone's autopilot. It sends a series of MAVLink messages to set and retrieve parameter values associated with the geofence.
6. **Looping and Conditions:** The function uses a series of loops to wait for specific conditions to be met before proceeding. It listens for incoming parameter value messages related to the geofence configuration and uses these messages to verify and confirm the settings.
7. **Uploading Fence Points:** The code iterates through each vertex in the "fence_list". For each vertex, it creates MAVLink messages to upload fence point information to the drone's autopilot.
8. **Finalization:** After all fence points are uploaded successfully, the geofence configuration is completed, and a confirmation message is printed.

Overall, this function demonstrates how the autopilot module processes incoming geofence commands from the DashApp, sets geofence parameters, uploads the geofence polygon's vertices, and verifies the configuration to ensure proper functionality.

CHAPTER 8: SIMULATION AND DRONE LAB TESTS

Now that we've explored the structure and construction of the DashApp, the next step is to put it to the test. We'll be conducting tests in two environments: simulation using the Mission Planner and in real-life scenarios within EETAC's Drone Lab. This comprehensive testing will ensure that the DashApp functions seamlessly across different contexts.

In addition to the testing process, we'll provide a set of step-by-step instructions to verify the proper functionality of the DashApp. It's important to note that the modules within the DEE ecosystem are subject to ongoing updates and improvements. By establishing these verification steps, we aim to offer future users and developers a reliable way to assess whether all the app's features are functioning correctly. This ensures that the DashApp remains a valuable tool for drone control and monitoring, both now and in the future.

8.1 Simulation tests in Mission Planner

Our first step in this process is to prepare the scenario. This involves ensuring that all modules, both on-board and ground station software, are set up in sync and ready to go.

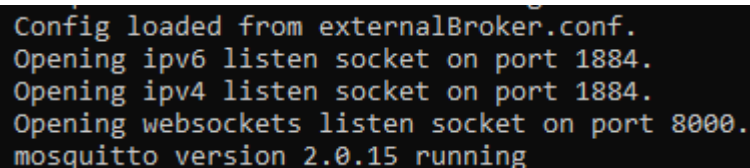
In our scenario, this encompasses the on-board modules, such as the Autopilot and the Camera Service, as well as the ground station software applications, primarily the DashApp itself. This synchronization extends to the communicational broker, MQTT, which facilitates communication between these modules, and the Mission Planner, a critical tool for the test simulation.

This meticulous preparation sets the stage for comprehensive testing and ensures that all elements of the system are in sync and ready for evaluation.

8.1.1 Configuring the scenario

To configure the scenario for testing, several key steps must be followed. These steps are crucial to ensure that all components work seamlessly together. Here are the steps of the configuration process:

1. **MQTT Broker:** The first step is to launch the MQTT broker, specifically on port 1884. This broker will act as the communication hub for all the modules involved.

A terminal window with a black background and light blue/green text. The text shows the MQTT broker (mosquitto) starting up, loading configuration from externalBroker.conf, opening listen sockets for IPv6, IPv4, and websockets on port 1884, and reporting its version as 2.0.15.

```
Config loaded from externalBroker.conf.  
Opening ipv6 listen socket on port 1884.  
Opening ipv4 listen socket on port 1884.  
Opening websockets listen socket on port 8000.  
mosquitto version 2.0.15 running
```

Fig. 8.1 External broker running on port 1884

2. **Mission Planner Simulator:** As the test operates in simulation mode, the Mission Planner simulator is initiated in simulation mode with the multirotor. This provides a virtual environment for testing the drone's behavior and interactions

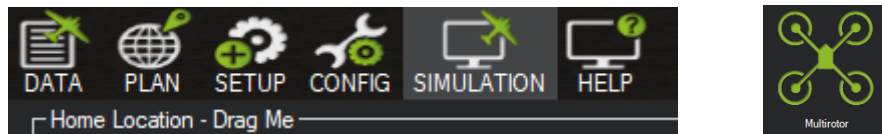


Fig. 8.2 Mission Planner in simulation mode with a multirotor

3. **Initialize On-Board Modules:** The next critical step is to initialize both the camera and autopilot modules. These modules are set to operate in global and simulation modes. Importantly, they are configured to use the public broker "broker.hivemq.com" as an external broker. This external broker serves as a bridge to connect the on-board modules to the broader network.

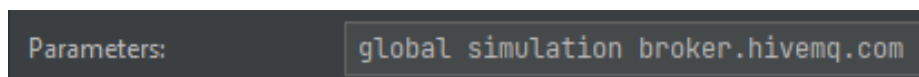


Fig. 8.3 On-board modules' running script configuration

4. **Start the DashApp:** Finally, the DashApp is started. This web application acts as the user interface for controlling and monitoring the drone's behavior. It connects to the MQTT broker, enabling users to send commands and receive. If an error message like the one given below appears, the user must check the broker which the DashApp is trying to connect to.

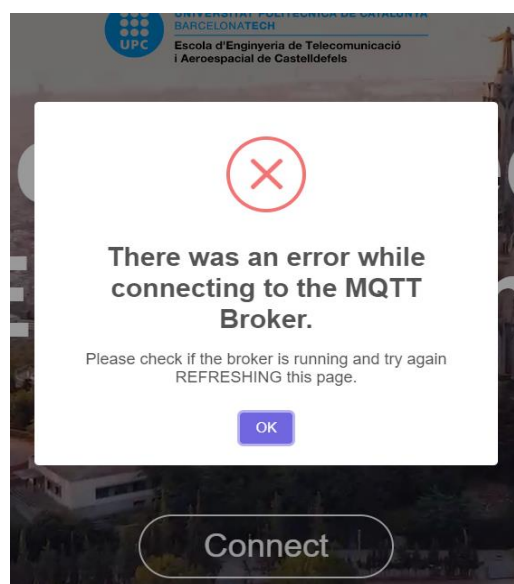


Fig. 8.4 DashApp's error message on connecting to the MQTT Broker

By following these steps, the scenario is fully configured and ready for testing. Each component plays a crucial role in ensuring the system's functionality and effectiveness during simulation testing.

8.1.2 Testing DashApp functionalities

With the scenario correctly configured, the next phase in the testing process is to thoroughly evaluate every functionality of the DashApp. This comprehensive testing ensures that all the app's features and modules are operational and responsive.

8.1.2.1 Drone Free Guiding component validation process

To grasp the validation process more effectively, it's crucial to recall the component's visual structure:

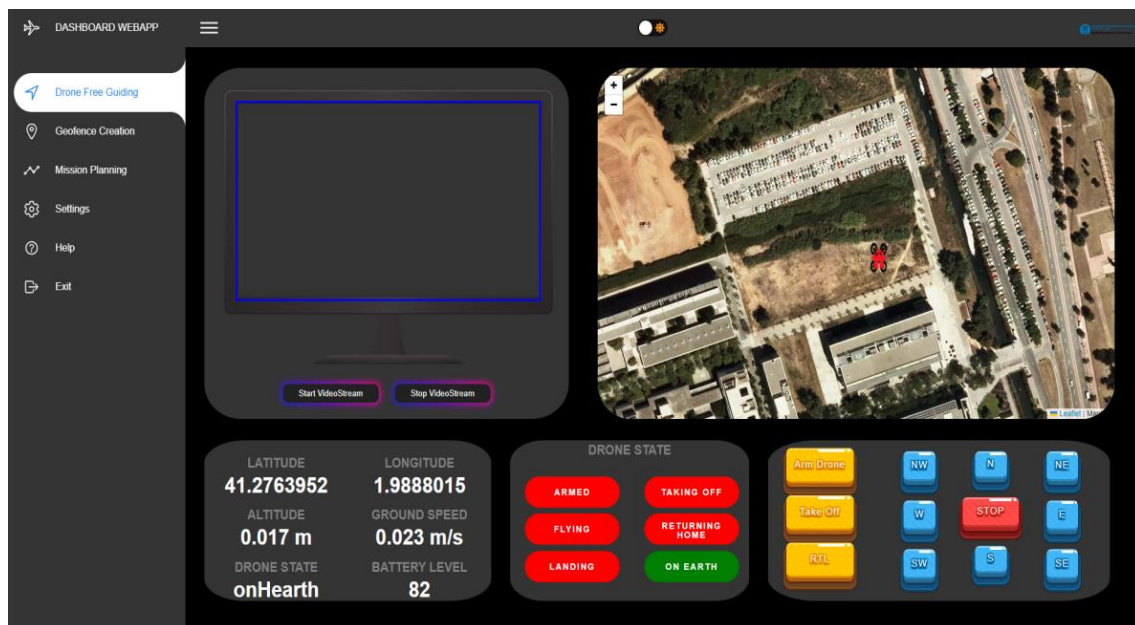


Fig. 8.5 Drone Free Guiding component's window

The following is the validation process that the user must follow to ensure the correct functioning of the Drone Free Guiding component.

1. **Basic commands:** Functions like arming, taking off, and landing the drone should execute smoothly without glitches. Users should be able to interact with the drone effortlessly through the DashApp.
2. **Telemetry data:** Telemetry information, including altitude, speed, and drone's coordinates data, should be displayed accurately on its corresponded card. This data ensures users have instant information about the drone's status.
3. **Drone movement on the map:** When users send commands for the drone to follow a specific path, it's essential to verify that the drone indeed moves as instructed on the map displayed in the DashApp.

This confirms that the navigation commands are accurately communicated to the drone.

4. **Drone state buttons:** The colour changes in the drone state buttons should accurately reflect the drone's actual state. For example, if the drone is in a "Taking Off" state, the corresponding button should be displayed in the green colour. Similarly, when the drone reaches other states like "Flying" or "OnEarth," the buttons should update accordingly. This visual feedback is crucial for users to monitor the drone's status.
5. **Live video streaming:** The DashApp should provide a real-time video stream from the drone's camera.

8.1.2.2 Geofencing

In order to better understand the further validation process, let's first take a look at how the Geofence's component is structured:

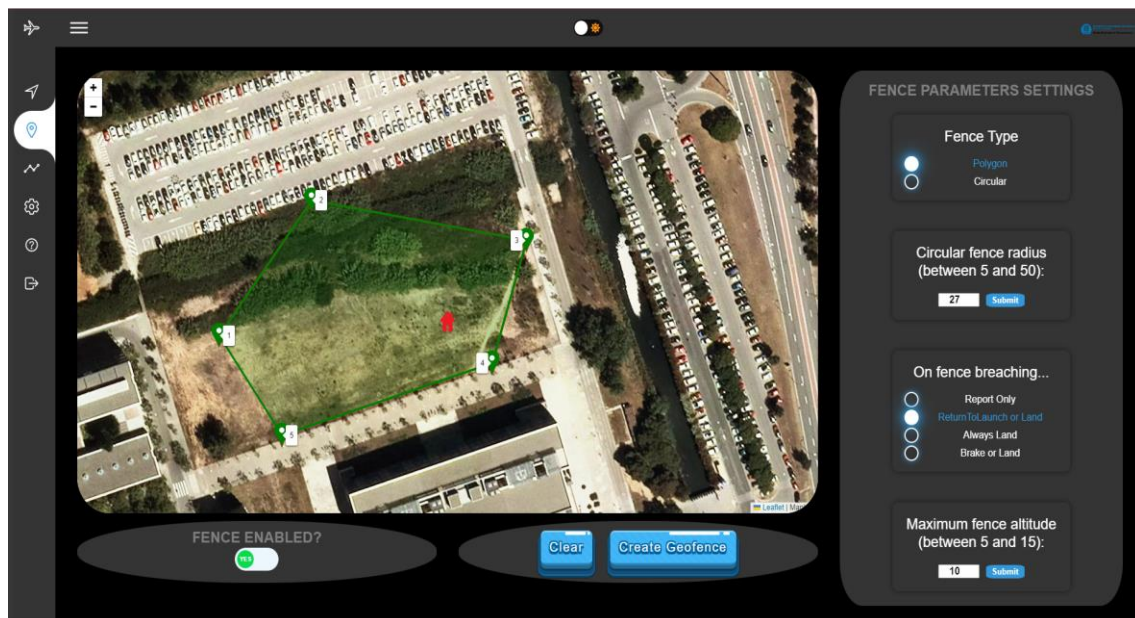


Fig. 8.6 Geofence component's window

Here is the prescribed validation procedure users should adhere to in order to confirm the proper operation of the Geofence component.

1. **Geofence Creation:** Begin by accessing the Geofence Creation component within the DashApp. Here's a step-by-step validation process:
 - **Circular or Polygonal Geofence Selection:** Depending on the desired geofence type (circular or polygonal), ensure that the appropriate option is selected.
 - **Drawing the Geofence**:** Use the provided tools to draw the geofence boundary on the map. Validate that:

- The geofence can be created by accurately marking the area of interest.
 - In the case of a circular geofence, ensure it has the correct radius.
 - For a polygonal geofence, confirm that the vertices accurately define the boundary.
 - Submission of Geofence Configuration: Once the geofence is drawn, submit the configuration. Verify that the configuration is correctly processed and sent to the autopilot service for geofence creation.
2. **Geofence Visibility:** After submitting the geofence configuration, users should check if the geofence is correctly displayed on the Free Guiding component's map. Validate that:
- The geofence boundary is visible on the map card.
 - The geofence shape (circular or polygonal) matches the intended design.
3. **Fence Enabled Button:** It's important to check the "Fence Enabled" button, which should be enabled (ON) if the user intends to use the geofence functionality during drone operations. Validate that:
- Enabling the geofence does not interfere with other drone functions.
 - Disabling the geofence does not affect other mission parameters.
4. **Geofence Functionality in Mission:** During a planned mission, validate the geofence's effectiveness:
- Ensure that the drone respects the geofence boundaries. If the drone approaches or breaches the geofence, it should respond according to the predefined action (e.g., Return To Launch).
 - Verify that the geofence triggers the appropriate actions, such as RTL (Return To Launch), as expected.
5. **Geofence configuration editing:** If the user needs to modify the geofence, access the editing tools in the Geofence Creation component. Validate that:
- Changes to the geofence shape, size, or location can be made.
 - Submitting edited configurations updates the geofence effectively.
6. **Clear Geofence:** Test the "Clear" button to ensure it removes the geofence from the map and deactivates its functionality.

By following these steps, users can thoroughly validate the Geofence component's functionality within the DashApp, ensuring that geofences are created accurately and enabled as needed.

8.1.2.3 Mission Planning

To facilitate a clearer understanding of the upcoming validation process, let's begin by observing the architecture of the Mission Planning component:

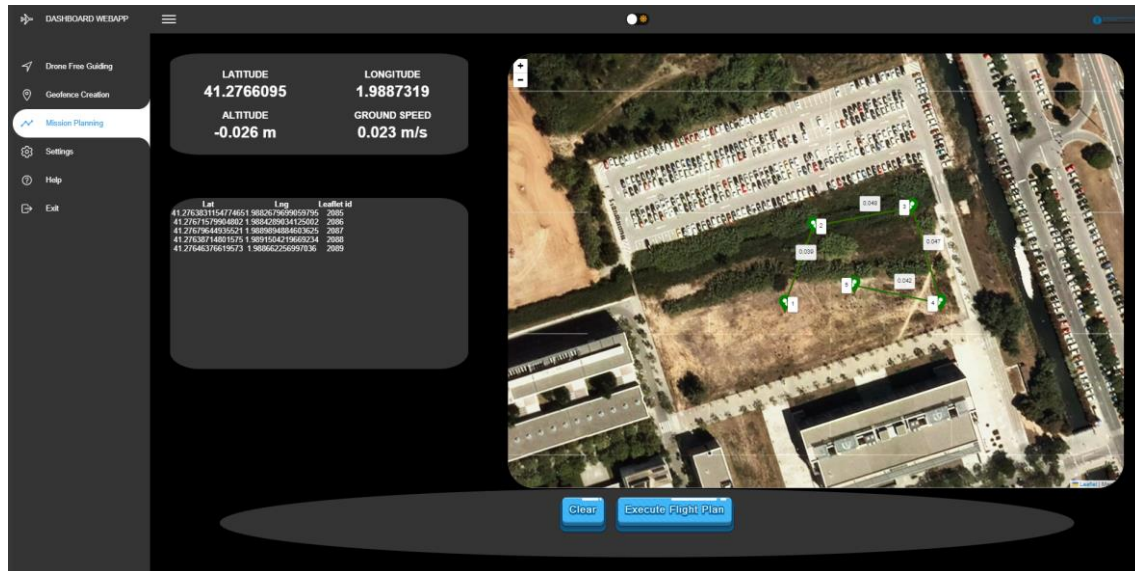


Fig. 8.7 Mission Planning component's window

The subsequent steps outline the validation protocol that users must undertake to ensure the Mission Planning's component functions correctly.

1. **Map Leaflet Card:** Begin by inspecting the map leaflet card, which serves for designing the drone's mission. Users should:
 - Ensure that waypoints, representing specific locations the drone will visit, are accurately placed on the map.
 - Confirm that the waypoints are arranged in the desired order of visitation.
2. **Telemetry Info Card:** Check the telemetry info card, where real-time data about the drone's status is displayed. Validate that:
 - The telemetry data is continuously updated as the drone follows the planned route.
 - Key parameters such as altitude, speed, and coordinates of the drone are being correctly reported.
3. **Selected Waypoints Card:** In this card, the waypoints selected by the user for actions (e.g., taking a picture) should be reviewed:
 - Ensure that the correct waypoints are marked for special actions.
 - Verify that the action (e.g., taking a picture) is set as intended for each selected waypoint.

4. **Mission Planning Buttons Card:** Examine the button card containing options for clearing the mission plan and sending it to the autopilot:
 - Confirm that the "Clear" button clears the map of any previously drawn waypoints.
 - Verify that the "Execute Flight Plan" or similar button sends the current mission plan to the autopilot for execution. Observe that the autopilot acknowledges receipt and starts executing the mission.
5. **Monitoring Execution:** As the drone follows the planned route, closely monitor its progress on the map leaflet card and the telemetry info card. Ensure that it adheres to the defined waypoints and performs any specified actions accurately.

By systematically reviewing these components and conducting real-time monitoring, users can validate the accuracy and functionality of their flight planning within the DashApp, ensuring that the drone follows the intended mission path and performs actions as desired.

8.1.2.4 Drone's parameter settings

Following the same explanation structure as the other sections, let's first see the component's structure:

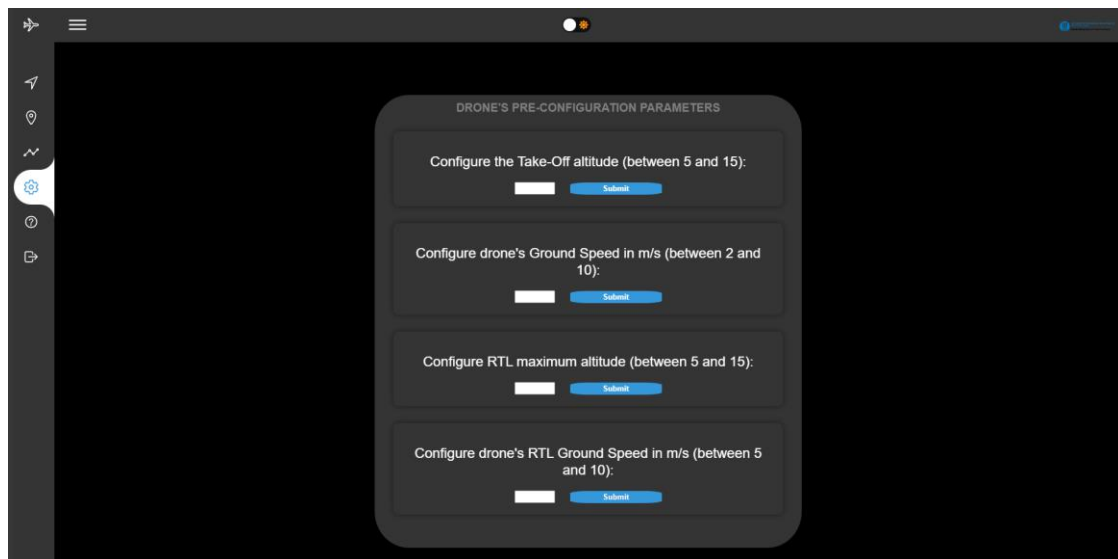


Fig. 8.8 Settings component's window

Before arming and flying the drone, users should verify that the parameters they've set and submitted are accurately configured when the drone is in flight. However if the user does not correctly define the parameters, an error message should appear.

These parameters play a significant role in defining how the drone behaves during its flight, including its flight characteristics, speed, altitude, and other essential parameters.

By confirming the parameter settings align with the intended flight plan requirements, users can maintain precise control over the drone's actions and ensure that it operates in accordance with their preferences and objectives.

8.2 Drone Lab tests

Having tested the DashApp in the simulated environment using Mission Planner, the next step involves conducting real world tests with a real drone in the Drone Lab.



Fig. 8.9 Drone ready to go for the execution of the tests in the Drone Lab

So, in this section, we will delve into the tests performed in the Drone Lab and present the outcomes and observations of it. Before conducting drone lab tests with the drone shown above, several previous steps need to be taken to ensure safe tests in production mode. Below I outline the key preparatory measures undertaken to operate in production mode, with a focus on our specific case working in global mode.

To prepare for the Drone Lab tests, we began by transitioning from simulation mode to production mode. This mode is essential for running the on-board services on the drone's computer, ensuring that the DashApp functions optimally during real-world operations.

In production mode, all on-board services (autopilot service, camera service and LEDs service) must be downloaded and installed on the drone's on-board computer, that is to say on the Raspberry Pi. This step guarantees that the services are available for execution during drone missions.

Moving on, the `boot.py` file, which plays a critical role in initializing the drone's software components, was executed to suit the requirements of production mode.

In our specific case, we worked in global mode, which meant configuring the DashApp to communicate with a broker. In our case “`classpip.upc.edu`” broker was used instead of the “`broker.hivemq.com`” that was employed in the simulation mode. This choice was made to suit the specific requirements and network settings of our Drone Lab environment as the “`broker.hivemq.com`” was very slow.

Before taking off for the drone lab tests, we conducted thorough check-ups to verify that all systems were functioning correctly:

1. CameraService Testing: We confirmed the proper operation of the CameraService, ensuring that the DashApp could capture and process images and video streams from the drone's camera system.
2. Drone Parameters Settings: All drone parameters were meticulously reviewed and adjusted as needed. This included fine-tuning flight parameters, calibrating sensors, and configuring the drone's behavior according to the specific mission requirements.
3. Communication Checks: We verified that communication between the DashApp and the drone's flight controller was stable and responsive. Any latency or connection issues were addressed before the actual flight tests.

By following these preparatory steps, we ensured that the drone and the DashApp were ready for rigorous testing in the Drone Lab environment. These measures not only enhance the safety of the tests but also set the stage for successful real-world drone missions.

Once configured and prepared everything, in the Drone Lab, we conducted a series of tests to assess the drone's control commands using our DashApp. These tests aimed to evaluate the drone's response to various commands, including arming, taking off, directional movements (such as north, east, north-east, etc), and returning to the launch point.

I am pleased to report that these tests were executed successfully, with the drone performing flawlessly in response to the commands sent through our DashApp. The precision and reliability of the drone's execution of these commands were highly promising, highlighting the effectiveness of our control system and the seamless integration between the DashApp and the drone's flight controller.

The following figure shows the drone flying during the execution of the tests:



Fig. 8.10 Drone taking off during the execution of the flying commands

The ability to command the drone accurately and have it respond as intended is a significant achievement. These successful tests represent a critical milestone in the development and validation of our drone control system, bringing us one step closer to deploying this technology in practical real-world scenarios with confidence and precision.

CHAPTER 9: DASHAPP'S INTEGRATION TO THE DEE'S GITHUB REPOSITORY

This chapter marks the culmination of the project as it outlines the integration of the DashApp into the DEE's GitHub repository.

This process involves uploading the DashApp's code, meticulously documenting the DashApp's section within the repository, and creating tutorial videos. These resources are aimed at helping future students interested in enhancing the application or simply utilizing its features effectively.

9.1 Uploading the code

One of the primary objectives of our project was to integrate the DashApp into the DEE's GitHub repository. Uploading the code signifies the commitment to open-source collaboration and the willingness to share our work with other people that are interested in enhancing the ecosystem. By doing so, we have made the DashApp easily accessible to everyone interested in using or building upon it.

9.2 Documenting the repository

Proper documentation is the backbone of any successful open-source project. I successfully documented the DashApp's section in the DEE's GitHub repository. This documentation serves as a comprehensive guide, enabling developers, often students, and users to understand the project's architecture, functionalities, and how to contribute effectively. The aim was to make it easier for anyone to navigate and utilize DashApp's capabilities.

9.3 Tutorial videos

Recognizing the importance of accessible learning resources, I have created tutorial videos to accompany the DashApp. These videos are designed to help the future students who wish to enhance the application or simply make use of its features. These tutorials provide step-by-step instructions for working with the DashApp.

Specifically, two distinct videos have been recorded to address different aspects of the application:

1. DashApp's functionalities showcase

In this video, we delve into the core functionalities of the DashApp. We offer a detailed walkthrough, demonstrating how to effectively utilize the various features and tools that the application has to offer. From connecting to the MQTT Broker to the use of different functionalities, this video provides a comprehensive overview of what the DashApp can do.

Whether you are a student eager to just explore the application's capabilities or a possible contributor looking to enhance the application, this video is for understanding the DashApp's workflow.

2. Code navigation:

The second video created focuses on the inner workings of the DashApp by providing a guide on how to navigate the codebase. This tutorial video is designed for students and aspiring contributors interested in understanding the code structure, architecture, and the logic behind the DashApp. We take you through the different sections of the code and explain key components. Whether you aim to contribute to the project or simply gain insights into the codebase, this video will be an invaluable resource to help you get started.

These two videos, together, provide a learning experience for anyone interested in the DashApp. They not only showcase its capabilities but also empower users with the knowledge they need to engage with the application.

I encourage you to explore these tutorial videos to deepen your understanding of the DashApp's functionality and codebase. They are available for your convenience in the DronsEETAC Youtube channel that you can find on the DEE's GitHub repository (see [1]).

This chapter signifies the successful conclusion of our project. We have not only integrated the DashApp into DEE's GitHub repository but also contributed to its continued growth and improvement. The code is now available for collaboration, the documentation is comprehensive, and the tutorial videos are ready to guide the way for future students.

CHAPTER 10: CONCLUSIONS

In this concluding chapter, I will present the final conclusions of my project. We'll begin by revisiting the initial objectives we set out to achieve and the ones we successfully accomplished. Next, we'll explore potential improvements for future contributors that can enhance the DashApp. Lastly, I will share my personal final opinion on the overall project.

10.1 Objectives accomplished

As explained in detail in the second chapter of this document, the principle goals of this project was to:

1. Shift the fundamental desktop Dashboard functionalities to a web app.
2. Add extra functionalities to the DashApp.
3. Make the autopilot module compatible with the additional functionalities inserted in the DashApp.
4. Integration of the DashApp within DEE's GitHub repository.

Our first primary goal was to transition the fundamental functionalities of the desktop Dashboard into a web app, which we successfully accomplished. We created a user-friendly interface that replicated the core functions of the Python made Dashboard and ensured that they were equally accessible and efficient in the DashApp.

In addition to migrating existing functionalities, we aimed to enhance the DashApp by adding new features. We successfully achieved this goal by introducing new capabilities, such as the inclusion of geofence functionality and the ability to set drone flight parameters, alongside significant improvements to the user interface.

Our next objective was to augment the autopilot module's capabilities to ensure compatibility with the newly added features in the DashApp. Through meticulous development and testing, we have successfully expanded the autopilot module's functionalities to make it functional with the DashApp's enhanced capabilities.

The final goal was to integrate the DashApp into DEE's GitHub repository. I am pleased to report that this crucial milestone has been accomplished, making the DashApp readily accessible to the future users and contributors.

10.2 Possible future improvements in DashApp

While we have met the objectives initially set, we have to acknowledge that there is always room for improvement:

1. **Addition of an exclusion geofence.** One promising avenue for improvement is the incorporation of an **exclusion geofence** feature.

This addition would empower users to choose between inclusion and exclusion geofences or even employ both simultaneously. This added flexibility in defining geographical boundaries can significantly enhance the DashApp's versatility, making it even more adaptable to various use cases and mission plans to test in the Drone Lab.

2. **Integration of all DEE's web apps into a unified platform.** Imagine a consolidated platform that provides a singular entry point to access all of DEE's web applications. This integration effort can unify the diverse range of DEE's web apps into a cohesive ecosystem.

The web app developed in this project, can serve as the foundation for this unified platform as we provide a platform to it, accommodating other web apps like the Drone Circus in its web edition. This consolidation not only facilitates user access but also empowers the ease of navigation.

3. **Enhancement of Mission Planning functionalities.** Mission planning is at the heart of any drone-related application. Future contributors to the DashApp can focus on expanding and refining the Mission Planning functionalities. The vast landscape of possibilities for creating mission plans presents an exciting challenge. By providing more robust and customizable mission planning tools, the DashApp can serve a wide variety of scenarios and mission requirements.

In summary, the future of the DashApp could be very impressive if the right enhancements are made. By adding an exclusion geofence, integrating DEE's web apps into a unified platform, and enhancing mission planning functionalities, we can continue to build upon the strong application established in this project.

These improvements will not only make the DashApp more powerful and versatile but also contribute to the growth and efficiency of the ecosystem's applications.

10.3 Personal conclusions

Developing the DashApp and contributing to the DEE has been an immensely rewarding journey, and it has left me with a profound sense of personal accomplishment. This project has not only allowed me to make a meaningful contribution to the DEE but has also been an incredible learning experience. One of the most significant benefits of this project has been the opportunity to gain practice and experience in programming languages that were not covered in my formal education. Languages like Vue.js and Python were beyond the scope of my degree program, but through this project, I had the chance to learn them.

Beyond programming languages, I delved into the workflows of communication protocols like MQTT and MAVLink. Understanding how these protocols facilitate the exchange of data and commands between the web app, autopilot module, and the drone itself was a fascinating and educational journey.

What truly stands out is how all these diverse technologies came together to bring the project to life. We successfully created a system where commands initiated from a web app were transmitted through a messaging protocol to the autopilot

module, which then communicated with the drone. Witnessing the integration of these technologies to achieve a common goal was not only intellectually satisfying but also a testament to the power of collaboration and innovation.

The final product, now integrated into the DEE, brings me immense satisfaction, knowing that the DashApp serves as a valuable tool within DEE, enhancing its capabilities and reach, fills me with pride and happiness.

REFERENCES

- [1] Valero, M. Drone Engineering Ecosystem's GitHub repository. Available on:
<https://github.com/dronsEETAC/DroneEngineeringEcosystemDEE>
- [2] Vue.js. Documentation available on :
<https://vuejs.org/>
- [3] W3Schools. JavaScript tutorial. Available on:
<https://www.w3schools.com/js/>
- [4] Valero, M. Vue tutorial adapted to the DEE. Available on:
<https://youtu.be/XCn9stPZ4iY?si=4zThn1qiLpLG9BsH>
- [5] SweetAlert2. Available on:
<https://sweetalert2.github.io/>
- [6] MAVLink. Common MAVLink messages. Available on::
<https://mavlink.io/en/messages/common.html>
- [7] DroneKit. Documentation available on::
<https://dronekit.io/>
- [8] MQTT. Documentation available on:
<https://mqtt.org/>
- [9] W3Schools. Python tutorial. Available on:
<https://www.w3schools.com/python/>
- [10] W3Schools. CSS tutorial. Available on:
<https://www.w3schools.com/css/>
- [11] ArduPilot. PyMavlink's geofence discussion. Available on:
<https://discuss.ardupilot.org/t/pymavlink-geofence/90526>
- [12] StackOverflow. Geofence with DroneKit and PyMavlink. Available on:
<https://stackoverflow.com/questions/47364392/geofence-with-pymavlink-or-dronekit-python>

ANNEX A: CONNECT PAGE CODE

A.1 Template

```
<template>
  <EntrancePage v-if="open"></EntrancePage>
  <div v-if="!open">
    <div class="hero">
      <video class="back-video" src="@/assets/2_BCN.mp4" autoplay muted
        loop></video>
      <nav>
        
      </nav>
      <div class="contenido">
        <h1>Drone Engineering Ecosystem</h1>
        <a v-if="!connected" @click="toggle"> Connect </a>
      </div>
    </div>
  </div>
</template>
```

A.2 Script

```
<script>
import { defineComponent, ref, onMounted, provide, inject } from "vue";
import EntrancePage from "./EntrancePage.vue";
import Swal from "sweetalert2"; // npm i -S vue-sweetalert2
import mqtt, { MqttClient } from "mqtt"; // npm install mqtt --save
import mitt from "mitt"; // npm i mitt

export default defineComponent({
  components: {
    EntrancePage,
  },
});
```

```
setup() {  
  let connected = ref(false);  
  let open = ref(false);  
  const client = inject('mqttClient');  
  const mqttConnect = inject('mqttConnected');  
  console.log("mqttConnect in ConnectPage.vue", mqttConnect);  
  
  async function toggle() {  
    try {  
      Swal.fire({  
        title: "Connecting.",  
        didOpen: () => {  
          Swal.showLoading();  
        }  
      },);  
      await new Promise(resolve => setTimeout(resolve, 1000));  
      if (mqttConnect == true) {  
        Swal.hideLoading();  
        Swal.close();  
        client.publish("dashboard/autopilotService/connect", "");  
        connected.value = !connected.value;  
        open.value = true;  
      }  
      else {  
        Swal.hideLoading();  
        Swal.close();  
  
        //console.log("esto se ejecuta primero");  
        throw new Error("");  
      }  
    }  
    catch (error) {  
      connected.value = false;  
      open.value = false;  
    }  
  }  
}
```

```
    console.log("Error connecting to MQTT broker");
    Swal.fire({
      icon: "error",
      title: "There was an error while connecting to the MQTT
      Broker.",
      text: "Please check if the broker is running and try again
      REFRESHING this page.",
    });
  }
}
return {
  toggle,
  connected,
  open,
};
},
});
</script>
```

A.3 Styles

```
<style scoped>
.hero {
  width: 100%;
  height: 100vh;
  background-image: linear-gradient(rgba(12, 3, 51, 0.3), rgba(12, 3, 51,
0.3));
  position: relative;
  padding: 0 5%;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
}

.logo {
```

```
width: 377px;
}

.contenido {
  text-align: center;
}

.contenido h1 {
  font-size: 130px;
  color: #fff;
  font-weight: 600;
  margin-bottom: 10%;
  transition: 0.5s;
}

.contenido h1:hover {
  -webkit-text-stroke: 5px #fff;
  color: transparent;
}

.contenido a {
  text-decoration: none;
  display: inline-block;
  color: #fff;
  font-size: 37px;
  border: 2px solid #fff;
  padding: 14px 70px;
  border-radius: 50px;
  margin-top: 50;
  cursor: pointer;
}

.contenido a:hover {
  -webkit-text-stroke: 1px #fff;
```

```
background: rgb(12, 197, 12);
color: #fff;
}

.back-video {
  position: absolute;
  width: 100%;
  right: 0;
  bottom: 0;
  z-index: -1;
}

@media (min-aspect-ratio: 16/9) {
  .back-video {
    width: 100%;
    height: auto;
  }
}

@media (max-aspect-ratio: 16/9) {
  .back-video {
    width: auto;
    height: 100%;
  }
}

</style>
```

ANNEX B: ENTRANCE PAGE CODE

B.1 Template

```
<template>

<Dashboard v-if = "openDash"></Dashboard>

<header>
  <a href="#" class="brand">Drone Engineering Ecosystem</a>
  <div class="menu-btn" @click="menuBTNClicked"></div>
  <div class="navigation">
    <div class="navigation-items">
      <a href="#">Home</a>
      <a href="#">About</a>
      <a href="#">Explore</a>
      <a href="#">Gallery</a>
      <a href="#">Contact</a>
    </div>
  </div>
</header>

<section class="home">
  <video class="video-slide active" src="@/assets/3_BCN.mp4"
    autoplay muted loop></video>
  <video class="video-slide" src="@/assets/2_BCN.mp4" autoplay muted
    loop></video>
  <video class="video-slide" src="@/assets/1_BCN.mp4" autoplay muted
    loop></video>
  <video class="video-slide" src="@/assets/3_BCN.mp4" autoplay muted
    loop></video>
  <video class="video-slide" src="@/assets/2_BCN.mp4" autoplay muted
    loop></video>
  <div class="content active">
    <h1> Option one.
    <br>
```

```

    <span>DASHBOARD</span>

</h1>
<ul>
  The Dashboard is a front-end desktop application which allows
  the user to do a bunch of things, including:

  <br>
  <br>
  <ul style="list-style-type: disc; margin-left: 20px;">
    <li>Showing the picture/video stream sent by the camera
    service</li>
    <li>Free guiding the drone</li>
    <li>Simple flight planning</li>
    <li>Showing telemetry data</li>
    <li>Starting/Stopping a LED sequence in the drone</li>
  </ul>
</ul>

<button @click = "dashAccessPermission"> PLAY WITH IT </button>
</div>

<div class="content">
  <h1> slide two.<br><span>Drone Circus</span></h1>
  <p> Write description here.
  </p>
  <button> PLAY WITH IT</button>
</div>

<div class="content">
  <h1> Slide Three.<br><span>All time Record</span></h1>
  <p> Write description here.
  </p>
  <button> SHOW RECORD LIST</button>
</div>

<div class="content">
  <h1> Slide Four.<br><span>Gallery</span></h1>
  <p> Write description here.
  </p>
  <button> READ MORE</button>

```



```

</div>

<div class="content">
  <h1> Slide Five.<br><span>Contact</span></h1>
  <p> Write description here.
  </p>
  <button> READ MORE</button>
</div>

<div class="media-icons">
  <a href="github.com/dronseETAC/DroneEngineeringEcosystemDEE"
    target="_blank"><i class="fab fa-github"></i></a>
  <a href="https://www.youtube.com/@dronseetac" target="_blank"><i
    class="fab fa-youtube"></i></a>
  <a href="#" target="_blank"><i class="fab fa-twitter"></i></a>
</div>

<div class="slider-navigation">
  <div class="nav-btn active" @click="navBTNClicked(0)"></div>
  <div class="nav-btn" @click="navBTNClicked(1)"></div>
  <div class="nav-btn" @click="navBTNClicked(2)"></div>
  <div class="nav-btn" @click="navBTNClicked(3)"></div>
  <div class="nav-btn" @click="navBTNClicked(4)"></div>
</div>
</section>
</template>

```

B.2 Script

```

<script>

import mqtt, {MqttClient} from 'mqtt';
import { defineComponent, inject, ref } from 'vue';
import Dashboard from "./Dashboard.vue";

export default defineComponent({
  name: 'App',
  components: {
    Dashboard,

```

```
  },

  methods: {
    menuBTNClicked(){
      const menuBtn = document.querySelector(".menu-btn")
      const navigation = document.querySelector(".navigation")
      menuBtn.classList.toggle("active");
      navigation.classList.toggle("active");
    },
    navBTNClicked(i){
      const btns = document.querySelectorAll(".nav-btn");
      const slides = document.querySelectorAll(".video-slide");
      const contents = document.querySelectorAll(".content");

      btns.forEach((btn)=> {
        btn.classList.remove("active");
      });

      slides.forEach((slide)=> {
        slide.classList.remove("active");
      });

      contents.forEach((content)=> {
        content.classList.remove("active");
      });

      btns[i].classList.add("active");
      slides[i].classList.add("active");
      contents[i].classList.add("active");
    },
  },
  setup () {
    let dashAccess = ref (false);
    let openDash = ref(false);
```

```
const client = inject('mqttClient');

function dashAccessPermission(){
  client.publish("dashBoard/autopilotService/getHome", "");
  dashAccess.value = true;
  openDash.value = true;
  console.log("Set to true");
}
return {
  dashAccessPermission,
  dashAccess,
  openDash,
}
}
});
</script>
```

ANNEX C: DRONE FREE GUIDING'S COMPONENT CODE

C.1 Template

```
<template>
  <div style="display:flex; height:63%;width:100%;">
    <div class="videoStreamStyle">
      <VideoStream></VideoStream>
    </div>
    <div class="mapStyle" id="map">
    </div>
  </div>

  <div style="display:flex;height:30%; width:100%">
    <div style="width: 30%;" class="telemetryInfoStyle">
      <div style="width: 50%; margin: 5%;">
        <p class="pTitleStyle"> LATITUDE </p>
        <p class="pValueStyle"> {{ droneLatitude }} </p>
        <p class="pTitleStyle" style="margin-top: 8%;"> ALTITUDE </p>
        <p class="pValueStyle"> {{ droneAltitude }} m </p>
        <p class="pTitleStyle" style="margin-top: 8%;"> DRONE STATE </p>
        <p class="pValueStyle"> {{ droneState }} </p>
      </div>
      <div style="width: 50%; margin: 5%;">
        <p class="pTitleStyle"> LONGITUDE </p>
        <p class="pValueStyle"> {{ droneLongitude }}</p>
        <p class="pTitleStyle" style="margin-top: 8%;"> GROUND SPEED </p>
        <p class="pValueStyle"> {{ droneGroundSpeed }} m/s</p>
        <p class="pTitleStyle" style="margin-top: 8%;"> BATTERY LEVEL</p>
        <p class="pValueStyle"> {{ droneBattery }}</p>
      </div>
    </div>
  </div>
```

```

<div style="width: 25%;" class="droneStateStyle">
  <div style="margin:1%">
    <p class="pTitleStyle"> DRONE STATE</p>
  </div>
  <div style="display: flex;">
    <div style="width: 50%; margin: 5%;">
      <button class="stateButton" id="armedDisarmed"> DISARMED
      </button>

      <button class="stateButton" id="flying"> FLYING </button>
      <button class="stateButton" id="landing"> LANDING </button>
    </div>
    <div style="width: 50%; margin: 5%;">
      <button class="stateButton" id="takingOff"> TAKING OFF
      </button>

      <button class="stateButton" id="returningHome"> RETURNING HOME
      </button>

      <button class="stateButton" id="onHearth"> ON EARTH </button>
    </div>
  </div>
</div>
</div>
<div style="width: 39.33%;" class="Controller">
  <DroneController></DroneController>
</div>
</div>
</template>

```

C.2 Script

```

<script>
import { ref, inject, onMounted } from "vue";
import leaflet from 'leaflet'
import "leaflet/dist/leaflet.css";
import VideoStream from './VideoStream.vue';
import DroneController from './DroneController.vue';
import droneImage from "@/assets/Drone2.png"
import markerImageGreen from "@/assets/greenMarker.png"

```

```
import markerImageRed from "@assets/marker.png";
import homeImage from "@assets/home.png";

export default {
  components: {
    VideoStream,
    DroneController
  },
  setup() {
    const client = inject('mqttClient');
    const emitter = inject('emitter');
    let map;
    let telemetryInfo = ref(undefined);
    let droneLatitude = ref(41.276486);
    let droneLongitude = ref(1.9886);
    let droneGroundSpeed = ref(undefined);
    let droneAltitude = ref(undefined);
    let droneBattery = ref(undefined);
    let droneState = ref(undefined);
    let homeLatitude = ref(undefined);
    let homeLongitude = ref(undefined);
    let homePosReceived = ref(false);
    let armedDisarmedButton = undefined;
    let flyingButton = undefined;
    let takingOffButton = undefined;
    let returningHomeButton = undefined;
    let landingButton = undefined;
    let onEarthButton = undefined;
    let armed = ref(false);
    let flying = ref(false);
    let fencePoints = ref([]);
    let circleCP = ref(undefined);
    let fenceType = ref(undefined);
    let fenceRadius = ref(undefined);
```

```
emitter.on('armedBool', (armedBool) => {
  armed.value = armedBool;
});
emitter.on('fencePoints', (fPoints) => {
  fencePoints.value = fPoints;
  drawGeofence();
});
emitter.on('circleCP', (circleCenterPoint) => {
  circleCP.value = circleCenterPoint;
  drawGeofence();
});
emitter.on('fenceType', (fType) => {
  fenceType.value = fType;
});
emitter.on('fenceRadius', (fRadius) => {
  fenceRadius.value = fRadius;
});
var iconOptions = {
  iconUrl: droneImage,
  iconSize: [30, 50]
}
var droneIcon = leaflet.icon(iconOptions);
var markerOptions = {
  icon: droneIcon,
  draggable: true,
}

var iconOptionsRed = {
  iconUrl: markerImageRed,
  iconSize: [25, 40]
}
var markerIconRed = leaflet.icon(iconOptionsRed);
```

```
var markerOptionsRed = {
  icon: markerIconRed,
  draggable: true,
}

var iconOptionsGreen = {
  iconUrl: markerImageGreen,
  iconSize: [25, 40]
}

var markerIconGreen = leaflet.icon(iconOptionsGreen);
var markerOptionsGreen = {
  icon: markerIconGreen,
  draggable: true,
}

var iconOptionsHome = {
  iconUrl: homeImage,
  iconSize: [25, 40]
}

var markerIconHome = leaflet.icon(iconOptionsHome);

var markerOptionsHome = {
  icon: markerIconHome,
  draggable: false,
}

onMounted(() => {
  armedDisarmedButton = document.getElementById('armedDisarmed');
  flyingButton = document.getElementById('flying');
  takingOffButton = document.getElementById('takingOff');
  returningHomeButton = document.getElementById('returningHome');
  landingButton = document.getElementById('landing');
  onEarthButton = document.getElementById('onHearth');

  client.subscribe("autopilotService/dashBoard/telemetryInfo")
  client.on("message", function (topic, message) {
```



```

    if (topic == "autopilotService/dashBoard/telemetryInfo") {
        telemetryInfo.value = message;

        let telemetryInfoString = ref(undefined);
        telemetryInfoString.value = new TextDecoder("utf-8").decode(telemetryInfo.value);
        var telemetryInfoJSON = JSON.parse(telemetryInfoString.value);
        droneLatitude.value = telemetryInfoJSON["lat"];
        emitter.emit('droneLatitude', droneLatitude.value);
        droneLongitude.value = telemetryInfoJSON["lon"];
        emitter.emit('droneLongitude', droneLongitude.value);

        if (homePosReceived.value == false) {
            homeLatitude.value = droneLatitude.value;
            emitter.emit('homeLatitude', homeLatitude.value);
            homeLongitude.value = droneLongitude.value;
            emitter.emit('homeLongitude', homeLongitude.value);
            homePosReceived.value = true;
        }
        let cut = telemetryInfoJSON["groundSpeed"].toString();
        droneGroundSpeed.value = cut.slice(0, 5);
        emitter.emit('droneGroundSpeed', droneGroundSpeed.value);

        droneAltitude.value = telemetryInfoJSON["altitude"];
        emitter.emit('droneAltitude', droneAltitude.value);

        droneBattery.value = telemetryInfoJSON["battery"];
        droneState.value = telemetryInfoJSON["state"];
        changeState(droneState.value);
        markerPosUpdate();
    }
})

let token =
"pk.eyJ1IjoiamFza2llIiwiaYSI6ImNsZmdueGMyMzA1YnozdnJzY2pneHR5ODUifQ.6TxzG0
ureYxRZITNJPVQFw"

```

```
map = leaflet.map('map').setView([41.276486, 1.9886], 18);

leaflet.tileLayer('https://api.mapbox.com/v4/mapbox.satellite/{z}/{x}/{y}@2x.png?access_token=' + token, {
  maxZoom: 23,
  attribution: 'Mapbox'
}).addTo(map);

let marker = null;
let markerPosUpdate = () => {
  if (marker) {
    marker.remove();
  }
  marker = leaflet.marker([droneLatitude.value,
    droneLongitude.value], markerOptions).addTo(map);
  let homeMarker = leaflet.marker([homeLatitude.value,
    homeLongitude.value], markerOptionsHome).addTo(map);
};
})

function changeState(state) {
  if (state == "armed") {
    armed.value = true;
    emitter.emit('armedBool', armed.value);

    armedDisarmedButton.innerHTML = 'ARMED';
    armedDisarmedButton.classList.remove('disarmed', 'arming');
    armedDisarmedButton.classList.add('armed');
  }
  if (state == "arming") {
    armedDisarmedButton.innerHTML = 'ARMING';
    armedDisarmedButton.classList.remove('disarmed', 'armed');
    armedDisarmedButton.classList.add('arming');
  }
  if (state == "disarmed") {
    armedDisarmedButton.innerHTML = 'DISARMED';
    armedDisarmedButton.classList.remove('armed', 'arming');
```

```
    armedDisarmedButton.classList.add('disarmed');
  }
  if (state == 'takingOff') {
    onEarthButton.classList.remove('Green');
    landingButton.classList.remove('Green');
    takingOffButton.classList.add('Green');
  }
  if (state == 'flying') {
    flying.value = true;
    emitter.emit('flyingBool', flying.value);

    onEarthButton.classList.remove('Green');
    takingOffButton.classList.remove('Green');
    flyingButton.classList.add('Green');
  }
  if (state == 'landing') {
    flying.value = false;
    emitter.emit('flyingBool', flying.value);

    onEarthButton.classList.remove('Green');
    landingButton.classList.add('Green');
  }
  if (state == 'returningHome') {
    flying.value = false;
    emitter.emit('flyingBool', flying.value);

    onEarthButton.classList.remove('Green');
    returningHomeButton.classList.add('Green');
  }
  if (state == 'onHearth') {
    armed.value = false;
    flying.value = false;
    emitter.emit('armedBool', armed.value);
    emitter.emit('flyingBool', flying.value);
  }
}
```

```
onEarthButton.classList.add('Green');
returningHomeButton.classList.remove('Green');
landingButton.classList.remove('Green');
flyingButton.classList.remove('Green');
takingOffButton.classList.remove('Green');
armedDisarmedButton.classList.remove('armed');
armedDisarmedButton.classList.add('disarmed');
}
}

function drawGeofence() {
  console.log("ENTRA DRAW GEOFENCE")
  map.eachLayer((layer) => {
    if (layer['_latlng'] != undefined)
      layer.remove();
    if (layer['_path'] != undefined)
      layer.remove();
  });
  if (fenceType.value == '5') {
    for (let i = 0; i < fencePoints.value.length; i++) {
      console.log("Entra dentro el FOR de DrawGeofence");
      console.log("Fence point: ", fencePoints.value[i]);
      leaflet.marker(fencePoints.value[i],
markerOptionsGreen).addTo(map);
    }
    let last = fencePoints.value[fencePoints.value.length - 1];
    let first = fencePoints.value[0];
    leaflet.polyline([last, first], { color: 'green' }).addTo(map);
    leaflet.polyline(fencePoints.value, { color: 'green'
}).addTo(map);

    let polygon = leaflet.polygon(fencePoints.value, { color: 'green',
fillColor: 'green', fillOpacity: 0.2 }).addTo(map);
  }
  if (fenceType.value == '7') {
    console.log("Entraaa")
```

```
    console.log(circleCP.value)
    let circle = leaflet.circle(circleCP.value, {
      color: 'blue',
      fillColor: 'blue',
      fillOpacity: 0.2,
      radius: fenceRadius.value
    }).addTo(map);
  }
}

return {
  map,
  droneAltitude,
  droneBattery,
  droneGroundSpeed,
  droneLatitude,
  droneLongitude,
  droneState,
  homeLatitude,
  homeLongitude,
  homePosReceived,
  changeState,
  drawGeofence,
}
}
}
</script>
```

ANNEX D: AUTOPILOT'S GEOFENCE CREATION CODE

```
def create_polygonal_geofence(vertices_json):
    global vehicle
    global fence_action_original
    global i

    # vehicle.parameters['FENCE_ENABLE'] = 1

    fence_data = json.loads(vertices_json)
    print("Fence Data: ", fence_data)
    fence_type = int(fence_data[0])
    fence_action = int(fence_data[1])
    fence_alt_max = int(fence_data[2])
    fence_radius = int(fence_data[3])

    # introduce FENCE_TOTAL and FENCE_ACTION as byte array and do not use parameter index
    FENCE_TOTAL = "FENCE_TOTAL".encode(encoding="utf-8")
    FENCE_ACTION = "FENCE_ACTION".encode(encoding="utf-8")
    FENCE_ALT_MAX = "FENCE_ALT_MAX".encode(encoding="utf-8")
    FENCE_RADIUS = "FENCE_RADIUS".encode(encoding="utf-8")
    PARAM_INDEX = -1

    vehicle.parameters['FENCE_TYPE'] = fence_type # 5 = Polygon and MaxAlt geofence, 7 = Circular and MaxAlt geofence
    vehicle.parameters['FENCE_ACTION'] = fence_action # RTL (Return To Launch) if geofence breached
    vehicle.parameters['FENCE_ALT_MAX'] = fence_alt_max
    vehicle.parameters['FENCE_RADIUS'] = fence_radius

    # Add the ReturnToLaunch coordinates as first object in fence_list
    fence_list = [(41.2763653, 1.9886234)]

    vertices = fence_data[4]
    print("VERTICES", vertices)

    for vertex in vertices: # Access the nested list of dictionaries
        lat = vertex['lat']
        lng = vertex['lng']
        fence_list.append((lat, lng))

    # Add the first vertex again at the end to close the fence
    fence_list.append((vertices[0]['lat'], vertices[0]['lng']))
    print("FENCE_LIST: ", fence_list)

    # create PARAM_REQUEST_READ message
    message = dialect.MAVLink_param_request_read_message(target_system=0,
                                                         target_component=0,
                                                         param_id=FENCE_ACTION,
                                                         param_index=PARAM_INDEX)

    # send PARAM_REQUEST_READ message to the vehicle
    vehicle.send_mavlink(message)

    @vehicle.on_message('PARAM_VALUE')
    def listener(self, name, msg):
        print('msg dentro listener: %s' % msg.param_id)
        message_received(msg)

    def message_received(msg):
        global parameter_received
        global messageRx
        global i

        messageRx = msg
        # print('mensajes después de pasar por la función: %s' % messageRx)
```

```

1     if msg.param_id == 'FENCE_ACTION':
2         parameter_received = True
3         # print("Entra if parameter received")
4         i = i + 1
5         print('i = ', i)
6     if msg.param_id == 'FENCE_TOTAL':
7         i = 0
8         print('i = ', i)

9
10    while i <= 0:
11        pass

12    # wait until get FENCE_ACTION
13    if i == 1:
14        global messageRx

15        print("entra en primer i")

16        print("Mensaje recibido:", messageRx)

17        message = messageRx.to_dict()

18        if message["param_id"] == "FENCE_ACTION":
19            # get the original fence action parameter from vehicle
20            fence_action_original = int(message["param_value"])

21            print('FENCE_ACTION parameter original:', fence_action_original)

22            # break the loop

23            # create parameter set message
24            message = dialect.MAVLink_param_set_message(target_system=0,
25                                                         target_component=0,
26                                                         param_id=FENCE_ACTION,
27                                                         param_value=dialect.FENCE_ACTION_NONE,
28                                                         param_type=dialect.MAV_PARAM_TYPE_REAL32)

29            # send parameter set message to the vehicle
30            vehicle.send_mavlink(message)

31    # run until parameter set successfully
32    while i <= 1:
33        pass

34    if i == 2:

35        print("Entra en el while de i ==2 ")

36        # convert the message to dictionary
37        message = messageRx.to_dict()

38        # make sure this parameter value message is for FENCE_ACTION
39        if message["param_id"] == "FENCE_ACTION":
40            # make sure that parameter value reset successfully
41            if int(message["param_value"]) == dialect.FENCE_ACTION_NONE:
42                print("FENCE_ACTION reset to 0 successfully")

43            # should send param set message again
44            else:
45                print("Failed to reset FENCE_ACTION to 0, trying again")

46            # create parameter set message
47            message = dialect.MAVLink_param_set_message(target_system=0,
48                                                         target_component=0,
49                                                         param_id=FENCE_TOTAL,
50                                                         param_value=0,
51                                                         param_type=dialect.MAV_PARAM_TYPE_REAL32)

52            # send parameter set message to the vehicle
53            vehicle.send_mavlink(message)

```

```
if i == 4:

    message = messageRx.to_dict()

    if int(message["param_value"]) == fence_action_original:
        print('FENCE_ACTION set to original value {0} successfully'.format(fence_action_original))
        # Enable the geofence on the drone

        # should send param set message again
    else:
        print('Failed to set FENCE_ACTION to original value {0}'.format(fence_action_original))

print("All the fence items uploaded successfully")
```