FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Implementation of the new module into the Dronetag Mobile app in Flutter for planning, managing and coordinating drone fleets
Student:	Albert Moravec
Supervisor:	Ing. Lukáš Brchl
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

There is currently a lack of applications on the market that would allow drone pilots to conveniently manage and coordinate several aircraft within a single organization (drone fleet management). Most of the solutions are focused on only one drone manufacturer, have insufficient flight planning capabilities, or are simply overpriced. This thesis aims to utilize existing building blocks of the Dronetag Mobile app that already offers some drone coordination-related functionalities and extend for organization and fleet management use.

- Research, analyze and compare existing drone fleet management solutions.
- Define the functionalities of the planned fleet management module and design the backend extensions.
- Implement the functionalities in the Flutter framework in accordance with the latest trends.
- Test the application with real pilots, evaluate the results and suggest its future improvements.

Electronically approved by Ing. Michal Valenta, Ph.D. on 23 February 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor’s thesis

Implementation of the new module into the Dronetag Mobile app in Flutter for planning, managing and coordinating drone fleets

Albert Moravec

Department of Software Engineering
Supervisor: Ing. Lukáš Brchl

May 12, 2022

Acknowledgements

I would like to thank my family and friends for their constant support throughout my studies. I would also like to thank my supervisor Ing. Lukáš Brchl for always being there to discuss or help with every matter.

Declaration

Citation of this thesis

Moravec, Albert. *Implementation of the new module into the Dronetag Mobile app in Flutter for planning, managing and coordinating drone fleets.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Tato práce se soustředí na analýzu, návrh, implementaci a testování řešení pro správu dronů. Řešení by mělo umožňovat seskupení dronů a pilotů v rámci organizace a koordinaci letů s více piloty. Jsou vybrána a následně analyzována existující řešení a zhodnoceny jejich silné a slabé stránky. Dále je analyzována vnitřní funkčnost platformy Dronetag pro pochopení omezení kladených na vyvíjené řešení. Znalost fungování platformy Dronetag a silné a slabé stránky analyzovaných existujících řešení jsou poté použity pro analýzu nového řešení správy dronů. Dále je proveden návrh a implementce potřebných rozšíření platformy Dronetag. Rozšíření backendu je implementováno v jazyce Python za použití frameworku Django, rozšíření mobilní aplikace s použitím frameworku Flutter. Na závěr je funkční prototyp testován reálnými piloty za použití definovaných testovacích scénářů. Výstup z tohoto testování je poté využit k návrhu dalších vylepšení uživatelského zážitku a chování implementovaného řešení.

Klíčová slova bezpilotní létání, správa dronů, Flutter, Django, Python, Dronetag

Abstract

This bachelor's thesis focuses on analyzing, designing, implementing and testing drone fleet management solution. Fleet management solution should allow grouping drones and pilots into organizations and coordination of flights with multiple pilots involved. Existing fleet management solutions are selected and their analysis is then performed, assessing their strengths and weaknesses. Current Dronetag platform inner workings are analyzed for proper understanding of system constraints put on the developed solution. Strengths and weaknesses of the analyzed platforms and knowledge of the Dronetag platform is then applied for fleet management solution analysis. Needed extensions to the Dronetag platform are then designed and implemented. Backend extension is implemented in Python using the Django framework, mobile application extension then uses the Flutter framework. Finally, the working prototype is tested by real pilots using defined test scenarios. Output from execution of these scenarios is then used to propose further user experience improvements and behavior changes.

Keywords drones, fleet management, Flutter, Django, Python, Dronetag

Contents

List of Figures

List of Tables

List of Listings

Introduction

In recent years, drones have become increasingly popular among hobbyists and also in the commercial sector. According to the United States Federal Aviation Administration, the number of registered drones for commercial use increased from 12,000 registered units in 2016 to 488,000 in 2020 [1]. Companies have started to use drones to simplify tasks that have traditionally been difficult without them. Drones (and their operators) quickly became a necessity for technical inspections in high and hard-to-reach places, but also made place for themselves in agriculture, movie industry and photography as well.

Airspace prior to the expansion of drones to masses was tightly regulated by national aviation agencies. After drones became more popular, it became clear that additional regulation must be implemented to ensure civil and military aviation safety, alleviate privacy concerns, and prevent accidents. In the context of the European Union, this effort led to the creation of U-space*, which is a set of rules and systems that allow safe use of all types of drones and coordination with other types of aircraft. Since the required regulation needs to be discussed and implemented on both national and international levels, implementation progress is rather slow; however, it is beginning to take shape and should allow safer drone usage in the near future. At the core of this new regulation is mandatory drone registration for pilots and certain types of drones depending on their type and equipment they carry, as well as mandatory flight plan approval for flights inside restricted zones (e.g. city centers, airports). At the moment, the flight plan approval system is not yet in operation, but the infrastructure that will allow it is being built.

With increased drone usage in professional business, another problem arises. Apart from submitting flights for approval for more than one pilot at once, companies may want to coordinate job assignment for multiple drones and their pilots in one place and even supervise them in real time. After the flights themselves are completed, companies might want to collect and eval-

*More information on `https://www.sesarju.eu/U-space` uate data, such as detailed flight trajectory, distance flown, and flight hours accumulated for pilots and flight equipment. It would be natural to combine the mandatory flight plan registration process, data collection, and job coordination into one package.

This work will focus on designing the drone fleet management solution on top of the existing Dronetag platform. The final solution should allow for easy flight planning for multiple pilots, device management for organizations deploying drones, and review of historical data.

This thesis is divided into eight parts. The introduction will present the scope and goals of this thesis. In the first chapter, existing fleet management solutions will be presented and discussed. In the second chapter, current state of the Dronetag platform will be presented. In the third chapter, the preconditions and requirements of the proposed fleet management solution will be analyzed. In the fourth chapter, the proposed fleet management solution extensions to the Dronetag platform will be designed. In the fifth chapter, the solution will be implemented on top of the existing Dronetag platform backend in Python and the accompanying mobile app written using the Flutter framework. The sixth chapter will present the results of prototype testing with real pilots and their remarks on the presented solution and propose further improvements. The conclusion will then summarize the results of this thesis.

Motivation

The main motivation for choosing this thesis was my previous experience with the Flutter framework and the opportunity to work on a project that uses it. The second factor was the fact that I could shape the newly defined product domain together with other members of the Dronetag team and have relatively free hand on how the development is led. Importantly, the output of this thesis is a real and usable product that is going to impact the corporate drone market.

Objectives

The goal of this thesis is to analyze, design, and implement a fleet management solution suitable for organizations consisting of multiple drones and pilots.

First, similar existing solutions will be discussed and evaluated. Taking into account existing solutions, a possible fleet management system solution will be analyzed and designed according to the requirements and expected use cases. An analysis of the existing Dronetag platform and its inner workings will also be performed.

Using the knowledge acquired previously, the fleet management backend extension will be designed and implemented in Python on top of the existing Dronetag platform and properly tested. Following the backend implementation, an extension of the Dronetag mobile application will be designed and implemented. The resulting prototype will then be evaluated by real drone pilots, and further improvements will be suggested.

Chapter 1

Related Work

In this part, two fleet management solutions will be presented. FlyFreely [2] and DroneLogbook [3] will be described in detail, their workflow will be shown, and each platform's pros and cons will be assessed. Finally, the reason for omitting other solutions will be explained.

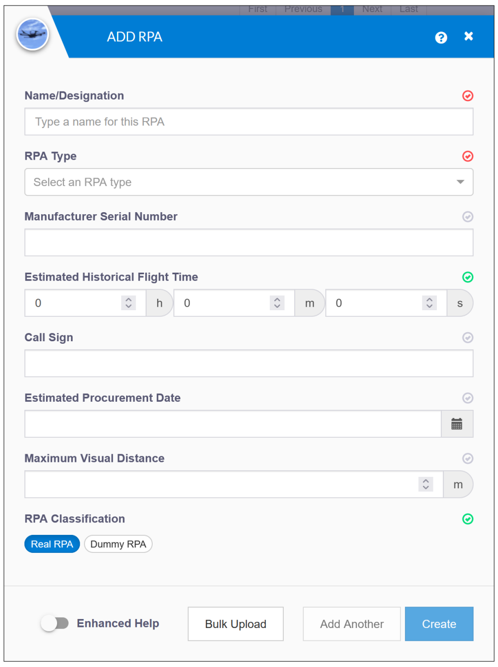
1.1 FlyFreely Platform

FlyFreely is a platform specialized in fleet management and general drone operation in the context of an organization [4]. It offers a free starter plan for evaluation and individual flight planning and tracking, as well as paid plans divided into three tiers depending on the size of your business and feature requirements [5].

It should be noted that the platform is specifically tailored to Australian drone regulations and offers additional features when used inside of Australia. There is also support for the United States regulations, and the United States is also the jurisdiction under which all operations outside of supported areas fall.

1.1.1 Registration Process

When first visiting the FlyFreely platform, the user is required to go through an on-boarding process. During this process, a role within an organization is chosen (depending on whether the user is directly involved in flight operations or in administrative), and then the user can choose to either join an existing organization or create a new one. In case a new organization is created, the on-boarding process continues, and the organization's name, jurisdiction, and aviation operations authority under which the company operates is chosen. For unsupported countries, "Generic / Unlisted" jurisdiction has to be chosen and generic operations authority has to be created.



Name/Designation

Type a name for this RPA

RPA Type

Select an RPA type

Manufacturer Serial Number

Estimated Historical Flight Time

0 h 0 m 0 s

Call Sign

Estimated Procurement Date

Maximum Visual Distance

m

RPA Classification

[✓ Real RPA] Dummy RPA

Figure 1.1: Add Aircraft Form (FlyFreely)

After completing all necessary steps, the operations dashboard containing everything related to the organization is shown. The core functionality covers the management of missions, drones, and organization personnel. The drone maintenance process, batteries and other additional drone equipment, location templates for mission planning purposes, organization documents, and useful links can also be managed, but these additional features will not be covered in further detail, as they were deemed unnecessary or replaced with alternative functionality in the final fleet management solution.

1.1.2 Personnel

Adding a new organization member starts with an e-mail invitation and role assignment. Roles give specific responsibilities; for example, allow members to plan missions, manage personnel, and observe or fly as a pilot in a mission. Once the person joins the organization, they are visible under the "Organization Personnel" tab. For pilots, a pilot license issued by organization's overseeing authority can be uploaded; however, when using generic operations authority (generally, when your local authority is not supported), file upload is disabled and not required. The pilot detail view also shows mission flight history and even allows manual log entry creation.

1.1.3 Aircraft

Creating a new aircraft is done using the form shown in Figure 1.1. To create a new aircraft, the name and type of aircraft are required. There is a large amount of predefined drones and other kinds of remotely piloted aircraft, and also the ability to create custom aircraft type in case a predefined one is not available. In addition to this required information, data like serial number, historical flight time, or call sign can also be added. When the aircraft is created, the user can upload manuals for the aircraft or add local authority's registration documents for the aircraft, similarly to pilot registration. Existing aircraft have a maintenance log and a flight log similar to the pilot flight log. Flights can also be added manually, or for Da-Jiang Innovations (DJI) drones they can be synced directly from DJI's cloud platform. The current status that determines whether the aircraft is suitable for flight, requires maintenance, or is retired is also tracked.

1.1.4 Missions

Missions are at the center of the FlyFreely platform, and the features described above are directly related to them. To create a mission, users must go through the four steps shown in the mission creation dialog.

First, there is the "Objectives" step where the mission name, objectives description, and mission location can be filled in. The mission location can be loaded from a previously created location template or drawn directly on the map. Upon clicking the "Draw Flight Area" button, another dialog containing a map and a set of drawing tools is shown. Drawing tools are divided into four categories – areas, markers, lines, and corridors. Polygons or corridors of the flight area can be marked as flight or no-flight zone, danger zone, or area of interest. Markers can be designated as points of interest, starting or landing points, or observer points. Lines can optionally be set as flight lines. The absolute minimum that has to be drawn is the flight zone polygon, which designates the area where pilots will be able to operate. Once the drawing is finished, the flight plan has to be named after which it can be saved.

After filling in all the information in the "Objectives" step, the "Resources" step becomes available. Here, it is required to enter mission type (test, training, or commercial) and a mission workflow defined by the organization's operations authority. After that, one or more aircraft have to be selected, and pilot in command of the mission has to be chosen. Optionally, additional crew members can be added along with additional notes for the crew.

Next up the "Mission planning" step must be completed. Here, the mission's planned time, duration, and visual line-of-sight operation mode, either visual line of sight (VLOS), extended visual line of sight (EVLOS), or beyond visual line of sight (BLOS) have to be entered. Optionally, maximum flight altitude, additional contact information, used radio frequencies, and related documents can be added.

After filling in all the necessary information, a dialog is shown informing the user that the mission can optionally be executed from the “Field App” mobile application.

1.1.5 Field App

FlyFreely “Field App” is a mobile application made specifically for the execution of missions on site. When opened, the user is presented with a list of current, canceled, and completed missions. The user can also enter an “Airspace Check” view with a map showing flight zones in the area. The information is very detailed in Australia and New Zealand, but globally only restricted areas, airports, and heliports are shown.

When an active mission is available, it can be selected and executed. To execute a mission flight, a pilot, an aircraft, and a battery pack must be selected. After confirming the configuration, flight tracking is started. There is no support for real-time aircraft position tracking, so the application only starts tracking flight time and allows viewing mission information. When the flight ends, the pilot can review the flight time, edit it, or insert additional flights manually. The pilot can then fly again or end the mission, and a recapitulation screen is shown.

1.1.6 Conclusion

FlyFreely provides a solution for the management of virtually every aspect of drone operations in an organization. It presents a concept of mission that allows organization members to organize and cooperate on a task. The platform also provides a wide range of supporting features related to personnel management, asset management, and other aspects. These features will also be considered for the final solution. The mobile app for field operation also provides improved user experience, as flights can be started without the need of a computer.

Positives:

- multiple pilots and aircraft in a single mission;
- complex organization asset management;
- advanced flight area drawing tools;
- battery management;
- maintenance workflows.

However, there are also some negative aspects. The whole platform is built with legal aspects of drone operation in mind, but currently only supports Australia and New Zealand, with limited support for the United States regulations. Other regions have to use an unintuitive workaround to use the platform. The whole platform is also very complex and hard to use at times, due to the large number of steps the user has to do to perform certain operations. Certain features seem even excessive and unnecessary for drone fleet operation. Finally, the functionality of the mobile app is quite limited at the time and no flight trajectory tracking is provided.

Negatives:

- heavy dependence on jurisdiction and regulations;
- unintuitive mission creation process;
- limited functionality of the Field App.

1.2 DroneLogbook Platform

DroneLogbook [3] is a mature and widely known platform among pilots, primarily used to maintain a personal flight log.

When creating an organization within DroneLogbook, the user first has to create a personal account. The process of creating an organization is then very simple – choosing the “Organization” section in the side menu and filling in the organization name and the field of activity in which the company operates. The organization is created immediately and receives an identifier (ID) which can be shared with other DroneLogbook users to join the organization. Users must bear in mind that organization members can no longer perform personal flights, and every new flight will be recorded under the organization. For this reason, it is mostly desirable to create a new account for organization use, even though the user can leave the organization to be able to perform personal flights again.

After creating an organization, the user becomes its administrator, and new options appear in the user interface. The most important of all available functions are “Inventory”, “Personnel”, “Flights”, and “Plan mission”. Additional sections “Projects”, “Maintenance”, “Inspections”, and “Incidents” are also available, but will not be covered in detail.

1.2.1 Personnel Management

Under personnel management, organization members can be added or modified. Adding a new member this way immediately creates a new account for them, which means that users with an existing account they wish to use in the organization must join using organization ID instead of being added manually.

After creating or inviting a new user, organization roles can be assigned, giving access to various fleet management options within the organization. Skills and skill ratings can also be assigned to individual users in the section “Skills” with an optional expiration date for the assigned skill.

Pilots in the organization need to have a pilot role assigned to them to be able to set them as pilots in organization flights. In addition, pilot license and allowed types of aircraft can be assigned to each pilot; however, this information does not affect functionality in any way, apart from being visible on the pilot’s profile.

1.2.2 Customers and Projects

Organizations can create their own customer database with basic information (name, address, and description). Existing customers can be assigned to a project during creation. Projects serve as a container for one or more related flights and additional information, such as project documentation, flight incidents, and time flown.

1.2.3 Inventory

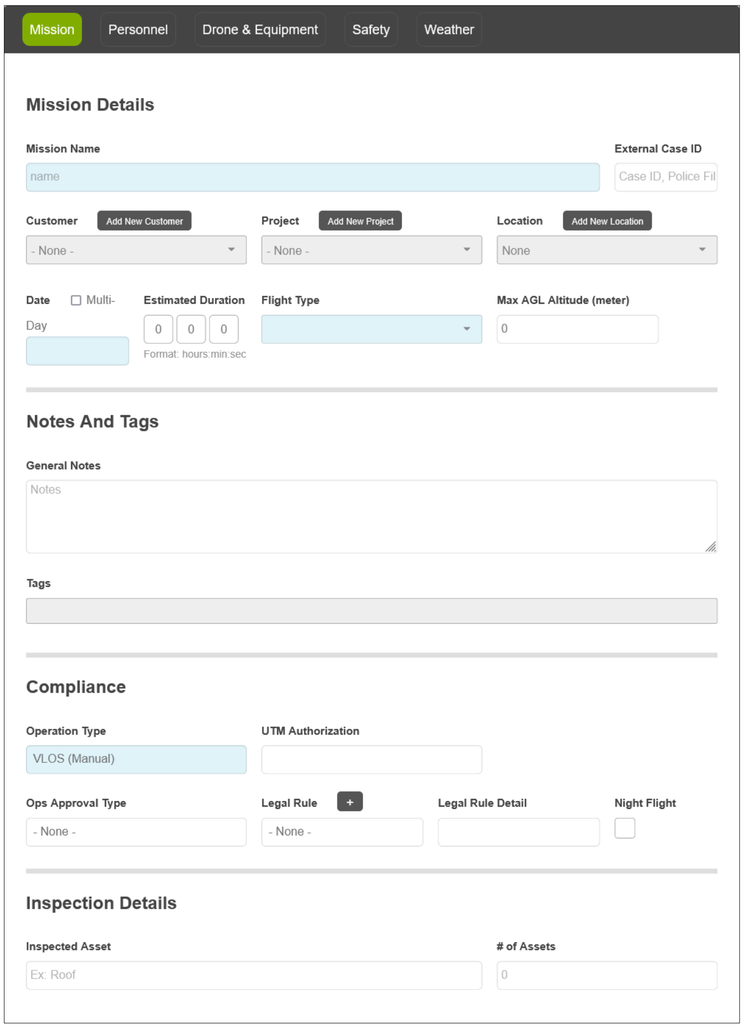
Inventory is a place for management of the assets of the organization. Here drones, batteries, and additional equipment can be registered under the organization ownership. Kits can then be created to tie the aforementioned assets together.

1.2.4 Missions

Missions are an organizational unit that group multiple flights in one place, usually expected to span only one day.

The mission creation form (shown in Figure 1.2) is almost identical to the “Manual Detailed Form” used to add manual flight log entries, hinting that missions might be implemented as an extension of current flight functionality. The process consists of 5 separate steps where most of the input data is optional. The first “Mission” section requires basic information, mission name, date, duration, flight type (commercial, hobby, test, etc.), and visual operation type (VLOS, BLOS, and other advanced types are available). Apart from the required data, mission can optionally be assigned to a project and customer and additional legal compliance information can be added.

In the next “Personnel” step, pilots, ground support, and other members of the mission are added. The assignment of a pilot is limited to only one “main” pilot. Other members of the mission must be assigned as a ground support crew or “others”. This limitation likely stems from missions being derived from single flight; however, it does not affect ability to assign flights from other organization members, even those not listed as mission crew, to mission.



Mission Details

Mission Name	name
External Case ID	Case ID, Police Fil
Customer	- None -
Project	- None -
Location	None
Date	Multi-Day
Estimated Duration	0 0 0 (Format: hours:min:sec)
Flight Type	
Max AGL Altitude (meter)	0

Notes And Tags

General Notes	Notes
Tags	

Compliance

Operation Type	VLOS (Manual)
UTM Authorization	
Ops Approval Type	- None -
Legal Rule	- None -
Legal Rule Detail	
Night Flight	

Inspection Details

Inspected Asset	Ex: Roof
# of Assets	0

Figure 1.2: Create Mission Form (DroneLogbook)

In the third “Drone & Equipment” step at least one drone has to be selected for the mission, and additional drones can optionally be added. Only the selected primary drone can be checked for availability, displaying a schedule with planned missions or flights for the selected drone. Necessary equipment and carried payload can also be specified here.

Fourth “Safety” step can optionally specify what safety measures must be taken during the mission.

The fifth “Weather” step allows the user to manually or automatically fill in the relevant weather data for the mission.

Upon filling in all the necessary information and clicking “Save”, the user is taken to a separate screen where the mission flight area can be defined. Here a map with a side bar containing drawn entities is displayed, and the user can use line, polygon, or point tool to draw into the map. The map data defined here can act as a guide for the pilots when in flight, but do not have any direct connection to mission flights themselves. The confirmation of the area finishes the mission creation process.

When the mission is created, it appears in the organization calendar, and the organization assets used in the mission are marked as in use (so that they can be looked up based on their availability for other missions). When mission is over, it has to be finalized manually, at which point the user can select logged flights that belong to the mission. Any logged organization flight that occurred during the mission time window can be added here without taking other flight parameters into account. Mission flights cannot be planned as part of the mission; they can only be added when (or after) the mission is finished. This behavior is consistent with how flights are handled in DroneLogbook – they are usually added once concluded and are not planned beforehand.

1.2.5 Flights

Individual flights can be added to DroneLogbook in multiple ways. There are two manual flight creation methods (simplified or detailed); however, automatic data upload from other cloud platforms (or manual data upload from exported log files) is usually preferred, as DroneLogbook integrates with many drone manufacturer cloud platforms.

The detailed manual mode is almost identical to mission planning and will not be further described. The simplified mode allows creating flight using a single dialog by filling in name, date and time, flight type, location, drone, and battery used. Operation mode (VLOS, BLOS, etc.), project, customer, and equipment can be optionally entered as well.

Automated flight creation can be done by uploading a flight log file (a wide range of formats is accepted) or by syncing with an external service (Autel Sync, DJI Sync and Skydio Sync are currently supported).

1.2.6 Conclusion

DroneLogbook is a go-to platform for many people due to its maturity and wide range of supported data sources for flight imports, making it easy to log flights even without supporting real-time tracking by the platform itself (unlike the Dronetag platform, which will be covered later). It provides convenient and easy-to-use tools for organizing flights and allows tracking of various operational data without overwhelming the user. However, the relationship between missions and flights seems very confusing and complicated to understand properly. That is mainly because flights are completely separated from the mission and are only added after the mission is finished, and also because all of the mission data, apart from date and time, do not have to correspond to the flight data – mission region, members, and aircraft are only informative and can differ from actual flight data.

This possible data inconsistency is also visible in other parts of the platform, showing that the features were probably implemented incrementally without proper integration into the system. One such example would be “Projects”, where the customer for the project can be selected; however, when creating a mission or flight, the project and the customer are selected separately and independently of each other.

Positives:

- wide range of supported flight data;
- easy to use user interface;
- pilot’s approved aircraft model tracking;
- multiple pilots, aircraft and flights allowed under mission.

Negatives:

- mission flights cannot be planned in advance;
- consistency issues with mission flights;
- mission data are only informative;
- personal flights not allowed when in organization.

1.3 Other Platforms

Fleet management platforms other than FlyFreely and DroneLogbook were also considered for further analysis, but were left out because paid subscriptions were required for access and a demo request was ignored or declined. For certain considered platforms, it was also recognized that they only work with very specific hardware or are made for a single specific field of operation rather than general fleet management.

Michal Skipala analyzed other significant platforms in his analysis of the Dronetag fleet management solution [6].

Chapter 2

Dronetag Platform

This section presents the Dronetag platform, its principles, functionality, and internal design. This thesis' primary goal is to extend this platform, which is why the platform, its internal design, and features related to fleet management are going to be described in detail.

2.1 Platform Introduction

The core of the Dronetag platform is drone in-flight identification and real-time tracking. Dronetag achieves this by using a small and lightweight battery-powered box attached to a drone. This box ("device" from now on) then sends real-time data to the Dronetag platform during flight. By using the device, pilots can make any drone comply with the new European regulation that requires all drones to identify themselves during flight and also enjoy the added value of being able to track their drone in real time [7].

2.2 Usage Overview

Currently, users can use the Dronetag platform to view information about other's flights and plan and execute their own flights. To be able to plan a flight, the user must have an aircraft and a device registered on the Dronetag platform.

2.2.1 Aircraft and Device Creation

The creation of aircraft is a very simple process. When creating a new aircraft, the user has to fill in its name, and then he can select the aircraft model from a predefined list or fill in the aircraft weight, class, and endurance manually (values can be overridden even when a predefined model is selected).

Device registration is also very simple – after providing a device serial number and a name, the device is assigned to the user's account and can be used.

2.2.2 Flight

With the aircraft and device registered, a flight can be planned by clicking "Plan a new flight" button in the map view. A set of planning steps appears on the right. By clicking on the map, a take-off point is set, and then the desired flight region is defined by drawing either a circle or a polygon and setting flight height range. Then in the "Date & Time" step, start and end time of the flight can be set. In the "Identification" step, an aircraft and a device with which the pilot will fly are selected. In the last "Flight Properties" step, visibility (allowing everyone to view the telemetry data), operation mode, and operation category can be set. After clicking the "Briefing" button, a card with a flight overview is shown and the flight can be confirmed.

Flights can, however, be created by pressing a button on the device which automatically creates a new flight with default device configured for that particular user. If a pre-planned flight is found when the device button is pressed, the planned flight is started instead of creating a new one.

When in a flight, Dronetag can be configured to send notifications to the pilot about potentially dangerous events. That can, for example, mean leaving designated flight region or someone else entering your region or flying in close proximity to you.

Planned or past flights are listed on user's flight list screen. Detail of ended flight shows flight trajectory which can be replayed using recorded telemetry data.

2.2.3 Fleet Management Support

Currently, Dronetag does not offer any fleet management capabilities and only supports single pilot flights with one aircraft and device being used at a time. There was an attempt to partially implement fleet management functionality in the backend, preparing for future expansion, but these parts will be omitted in the description for clarity, as the implementation will be done from scratch.

2.3 Architecture

The Dronetag platform consists of the following parts:

- Frontend
  - React Web App
  - Flutter Mobile App

- Backend
  - PostgreSQL Database
  - Django Backend
  - Live Service
  - Constrained Application Protocol (CoAP) Proxy
  - Redis
- Devices

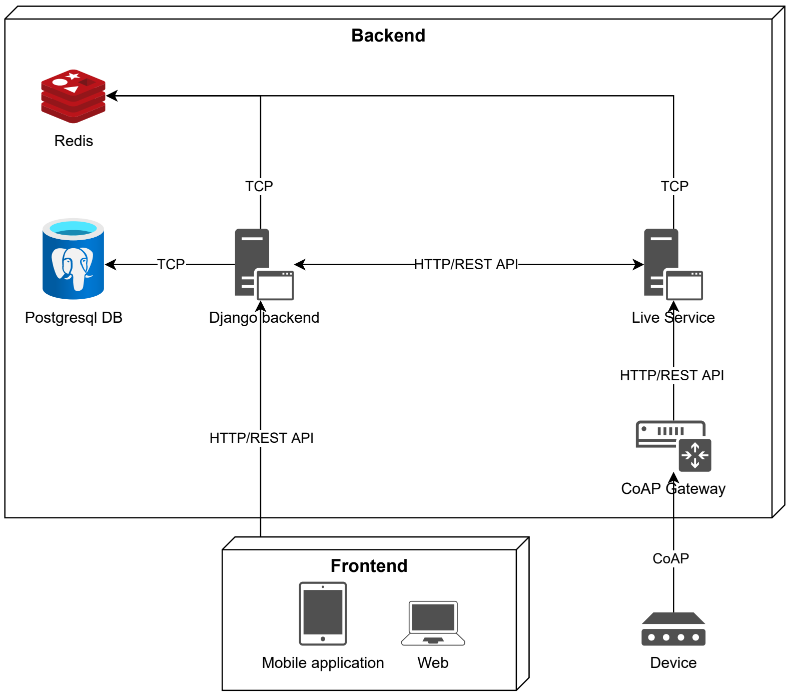


Figure 2.1: Current Dronetag Platform Architecture

The primary platform data storage is PostgreSQL [8] relational database (called “database” onward), which stores all persistent data. This database is connected to the Django [9] backend (called just “backend”) which provides application programming interface (API) for most of the user-facing functionality via Hypertext Transfer Protocol (HTTP). The API is designed with Representational State Transfer (REST) principles [10] in mind. The backend also defines and manages the persistent data being stored in the database. The frontend and mobile application can retrieve and manipulate the data using the backend HTTP REST API.

The devices communicate only with the Live Service using the CoAP protocol. This communication is translated by the CoAP proxy to standard HTTP

REST API requests that the Live Service handles. The Live Service and the backend then communicate using each other's HTTP REST API, this way the Live Service can request flight start or end and the backend can request all the telemetry data when the flight is finished. Additionally, the Live Service provides Websocket connection which allows frontend and mobile application to ingest telemetry data in real time without having to contact the backend.

Both backend and Live Service make use of Redis as an all-purpose storage for short-lived data (for caching purposes, etc.).

As this thesis' goal is to extend the backend part and implement new functionality in the mobile app, the Django backend and the Flutter mobile app will be now presented in further detail.

2.4 Django Backend

Dronetag backend HTTP API is based on the Django framework, a web application framework written in the Python programming language. Django is described as a framework that makes it possible to create web applications from concept to working solution in a matter of hours. It takes care of common development tasks, like user authentication or content administration [11].

> *Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. [12]*

Other external libraries are used to make API development easier (only the most relevant to fleet management implementation are listed):

- **Celery** – manages scheduling and execution of planned tasks [13],
- **Django REST Framework** – simplifies development of REST API using Django framework [14],
- **Sendgrid** – facilitates e-mail deilivery [15]
- **Shapely** – provides utility features for working with geospatial data [16],
- **Spectacular** – generates standardized REST API schema for documentation [17].

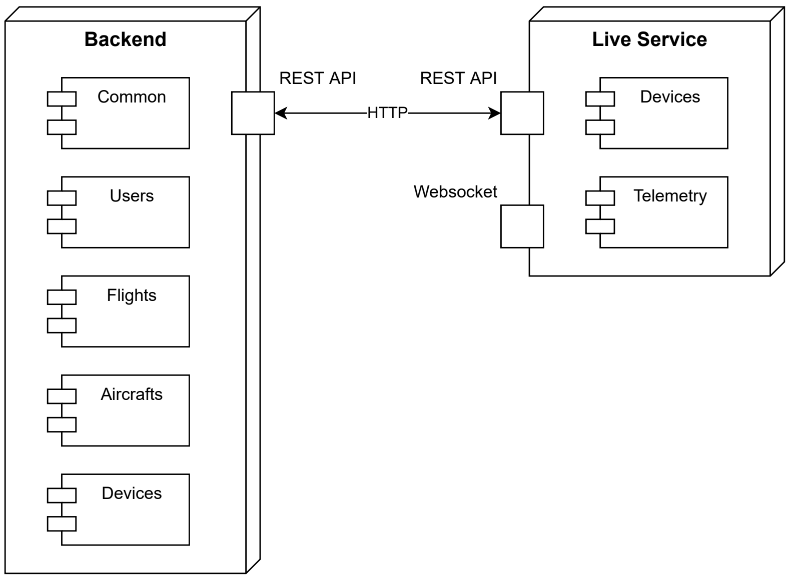


Figure 2.2: Current Dronetag Backend and Live Service Architecture

Using the Django framework and additional libraries, REST API is able to provide convenient features, like filtering based on Uniform Resource Locator (URL) query parameters, paging for more efficient loading, and automated endpoint documentation generation, without having to write large amounts of custom code or complicated configuration, as most of the functionality is provided out-of-the-box.

The API consists of a set of modules (which will be called "apps" from now on, following the Django terminology). Each app is responsible for small subsets of API and encapsulate functionality that is closely related (except for the "common" module, which contains shared functionality that other modules can use). Most apps follow the same code structure:

- **migrations** – directory containing automatically generated database transformations that are applied when the data structure changes,
- **templates** – directory containing HTML templates,
- **admins.py** – contains definition of administration forms available to Dronetag administrators,
- **apps.py** – contains app configuration,
- **fields.py** – contains definitions of model fields specific to the app,
- **mixins.py** – contains class extensions shared among app classes,
- **models.py** – contains data definitions, constraints, and validation rules upon which the database schema is generated,

- **permissions.py** – defines access policies for the application views,
- **serializers.py** – contains input and output data models and validation rules,
- **tasks.py** – contains Celery tasks,
- **urls.py** – defines available endpoints and views that handle them,
- **utils.py** – contains supporting functionality,
- **views.py** – contains modules that handle API requests for defined endpoints.

There are currently more than 10 apps present, but only the following apps relevant to fleet management will be described:

- Aircraft
- Device
- Flight
- User

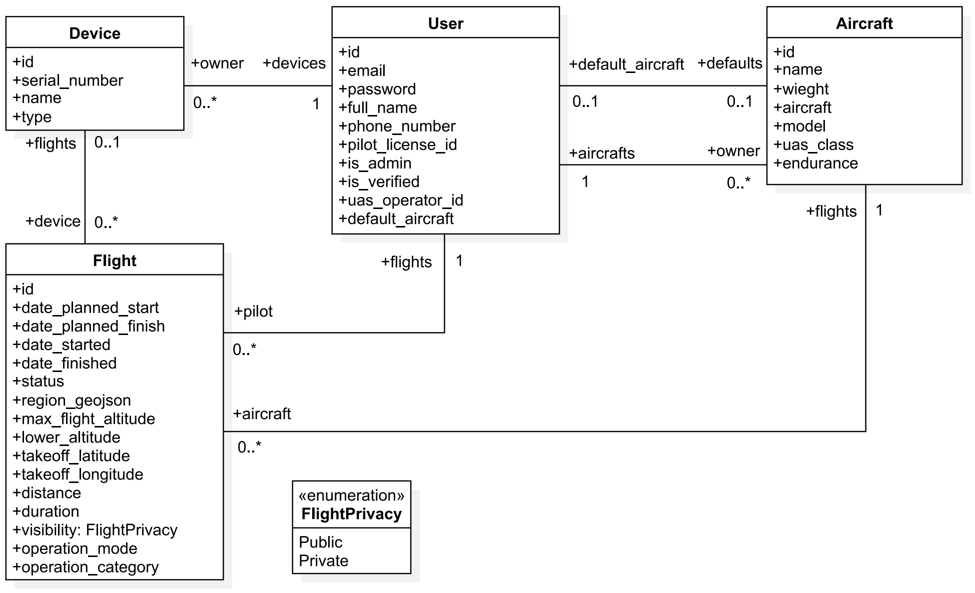


Figure 2.3: Current Domain Model

Each app defines its set of data models. These data models can form relationships with other models in the same app or even a different app. The simplified data model in Figure 2.3 describes the most important entities and their properties and relationships.

2.4.1 Aircraft App

This app contains all the application functionality regarding aircraft management. Available prefilled aircraft vendors and models are defined here. It provides API endpoints to list owned aircraft, create new aircraft or remove existing ones, and retrieve and update information on owned aircraft. It also allows listing predefined aircraft vendors and their models for the aircraft creation process.

2.4.2 Device App

This app defines the representation of devices in the Dronetag platform. It handles the device registration process and allows listing, modifying, and deleting owned devices. Functionality that allows the device itself to update its own data (through Live Service) is also implemented. Finally, the device status history can be uploaded or retrieved.

2.4.3 Flight App

This app contains code which is crucial to the usability of the entire platform. The flight and telemetry representation, as well as the endpoints that allow flight planning, listing, or deletion, are defined here. The flight start request from the device is implemented here as a dedicated endpoint that looks up a planned flight or creates a new one if none is found and sets it to an in-progress state. Public endpoint for identifying flight by announced device identifier is also implemented here. Flight telemetry data are processed here as they are uploaded by the Live Service, and they can also be exported to file for further processing.

The app also defines tasks that are run after the flight is finished – flight distance calculation and flight thumbnail image generation (this image shows a quick flight area and trajectory overview to the user) – and a cleanup task that marks expired flights as canceled if a planned flight is never executed.

2.4.4 User App

The User app is built on top of an existing Django functionality that provides user management out of the box. Basic user is extended with additional fields, such as the default aircraft or identifier assigned by the aviation authority (unmanned aircraft system (UAS) operator ID). Apart from standard user management functionality – user listing, registration, password reset, or e-mail validation – this module also handles tracking of allowed flight hours per month (in coordination with the Flight app) and also handles user mobile notification preferences.

2.5 Flutter Mobile Application

Platform functionality is provided to the user through the Dronetag web application or the Dronetag mobile application. This section will cover the mobile application in detail.

Dronetag mobile application is available for the Android and iOS operating systems and provides functionality similar to the web application with additional focus on flight operation, allowing the user to start a new flight immediately and observe detailed flight information.

2.5.1 Flutter Framework

Dronetag mobile app utilizes the Flutter framework, *"an open source framework by Google for building beautiful, natively compiled, multi-platform applications from a single codebase"* [18]. Flutter and the applications that use it are written in the Dart programming language [19].

> *Dart is a client-optimized language for developing fast apps on any platform. Its goal is to offer the most productive programming language for multi-platform development, paired with a flexible execution runtime platform for app frameworks. . . . Dart also forms the foundation of Flutter. Dart provides the language and runtimes that power Flutter apps, but Dart also supports many core developer tasks like formatting, analyzing, and testing code. [20]*

Dronetag chose Flutter for multiple reasons. Because Flutter applications can run on both Android and iOS with a single codebase [19], there is no need to develop two separate applications for each platform, significantly reducing development costs by reducing development time and reducing the burden of having to support two applications. Flutter also supports hot-reload when developing the application, which means that the application does not have to be recompiled when code is changed and only the relevant part is replaced directly in the running application [19] which also saves a lot of time in development. Another important attribute of Flutter is its performance when it comes to user interaction. Flutter draws each component of the user interface on its own canvas instead of relying on native implementations, allowing better portability between different platforms and better rendering performance [21]. Writing platform-specific code is still possible and native features can be used easily using "Method channels" which bridge native and Dart/Flutter code [21].

Flutter uses a modern approach to define and control the user interface.

> *In most traditional UI frameworks, the user interface's initial state is described once and then separately updated by user code at runtime, in response to events. One challenge of this approach is*

that, as the application grows in complexity, the developer needs to be aware of how state changes cascade throughout the entire UI. . . . Flutter, along with other reactive frameworks, takes an alternative approach to this problem, by explicitly decoupling the user interface from its underlying state. . . . you only create the UI description, and the framework takes care of using that one configuration to both create and/or update the user interface as appropriate. [21]

This means that the user interface, on an abstract level, is defined as a function of the application state, which is evaluated when the state changes, returning the user interface representation for that particular state. This approach allows for a number of different programming paradigms to be used when building Flutter applications. Composition is the primary paradigm used by Flutter in the description of user interface (UI).That is because UI in Flutter is represented by elements with a narrow scope of behavior that are composed to create one functional whole [19].

Another important paradigm is functional programming. Since every widget can be described as a function receiving state and returning its representation in UI, it is very easy to compose widgets or perform various transformations utilizing functional programming techniques [19].

2.5.2 State Management

As mentioned in the previous section, the user interface is a function of the application state. To avoid re-rendering the whole application every time there is a change in the application state, it is necessary to split the state into smaller parts. Since state changes can come from different sources, it is desirable to manage them in one place. It is also a good practice to separate the application's business logic that drives application state changes from the presentation layer where the state is processed into the user interface. This is handled using the business logic component (BLoC) pattern and the **Bloc** library that implements this pattern.

"This design pattern helps to separate presentation from business logic. Following the BLoC pattern facilitates testability and reusability. This package abstracts reactive aspects of the pattern allowing developers to focus on writing the business logic." [22]

The term BLoC describes both the pattern and a component that maintains the state and utilizes the pattern. The rules for the implementation of the BLoC pattern were defined by its author, Paolo Soares [23].

BLoC Rules:

- *Input/outputs are simple sinks/streams only.*
- *All dependencies must be injectable and platform agnostic.*

- *No platform branching is allowed.*
- *The actual implementation can be anything if rules 1-3 are followed.*

Widget Rules:

- *Each "complex enough" widget has a related BLoC.*
- *Widgets do not format the inputs they send to the BLoC.*
- *Widgets should display the BLoCs state with as little formatting as possible.*
- *If you do have platform branching, it should be dependent on a single bool state emitted by a BLoC.*

The **Bloc** library used in the Dronetag mobile application helps to utilize the BLoC pattern and makes development faster and easier, and also allows better code reuse and better testability [24]. To simplify the usage of BLoCs for cases where simple function calls would be sufficient to change the state, a *cubit* was integrated into the **Bloc** library, providing the same interface as the full-featured BLoC, but without the need to define input and output data streams and transformation of input events into new state [25].

Correctly defined cubit consists of state type, which defines the data that the cubit will provide to the user interface, initial state and methods that can be called from outside to transition to new state. The cubit then exposes a stream where new states are emitted. The user interface then listens for new data in the stream [26].

2.5.3 Dependency Injection

The application consists of multiple components that are divided into multiple layers and depend on each other. According to dependency inversion principle (DIP), the dependencies of the source code should refer only to abstractions, not to concrete modules. This is because every change to abstract interface requires changes in its concrete implementation, but changing concrete implementation does not always lead to a change of abstract interface. Therefore, relying on interfaces rather than concrete modules reduces volatility, and developers should always look to make abstract interfaces as stable as possible [27].

To achieve complete decoupling of the abstract interface and concrete implementation, a service locator is used. Service locator is a globally available registry where concrete implementations of services are registered and from which they can be retrieved upon request [28]. Service locator in the Dronetag application is provided by the **GetIt** library [29]. On application startup, a factory producing the concrete implementation or a singleton providing the concrete implementation is registered for each abstract interface.

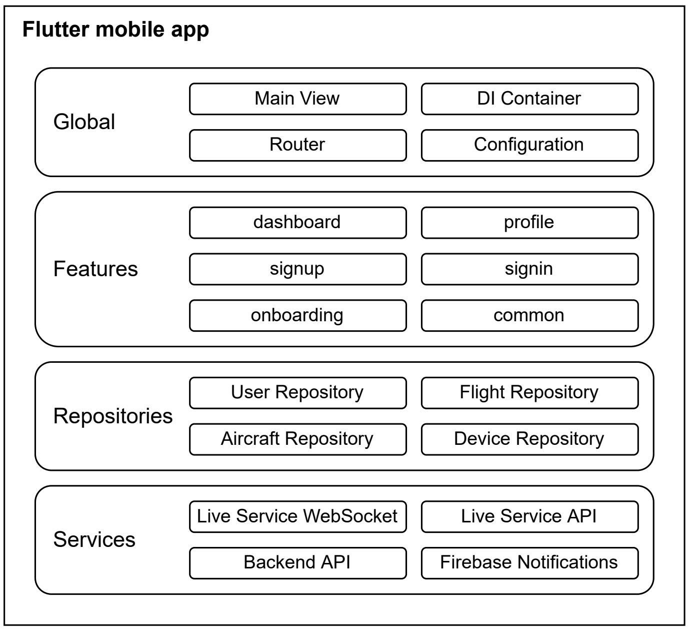


Figure 2.4: Mobile Application Structure

Components that utilize these abstract interfaces can then request concrete implementations from the service locator without directly depending on them [29].

Most components in the Dronetag mobile application require their dependencies during instantiation (as a constructor parameter) rather than requesting them from the service locator by themselves. This approach effectively turns the service locator into a container for *constructor injection* where dependencies are resolved based on the constructor parameters [28]. The *constructor injection* approach produces an even more flexible architecture, as components necessarily do not have to be instantiated with dependencies from the service locator, but can also be instantiated with dependencies provided using other means or a combination of both.

2.5.4 Application Structure

This section will describe the internal structure of the mobile application. Figure 2.4 shows a simplified diagram showing identified application layers (some modules have been omitted for clarity). The architecture can be divided into the following 4 layers:

Services provide lowest level of abstraction above the data sources used in the application. Modules in this layer provide means of communication with external data sources outside of the application and expose specific methods that the application can use to retrieve or send data to the data source.

Repositories contain modules for each standalone resource that the application can manipulate. Each module contains methods for manipulating the resource using modules from the *services* layer.

Features contain modules that reflect the structure of the user interface. Each feature module contains a single screen or a set of closely related screens, widgets that are used exclusively on those screens, and BLoCs (more precisely cubits) for those screens. Each BLoC handles part of the state of the feature in the application and provides methods that the user interface or external sources can call to trigger state change. These methods make use of the *services* and *repositories* layers to retrieve the data necessary for the transition to the new desired state.

Global handles the configuration and start process of the application, the registration of components into the service locator for dependency injection, and the configuration of the router and the registration of the available routes, so that the user can transition between the application screens. Additionally, the global application state is defined and managed on this layer. The global state is divided into multiple BLoCs exposed using the application context.

All layers make use of data models that are defined in the `models` package. These models provide data representation used across the application.

Chapter 3

Analysis

This section will describe fleet management functionality, its functional and non-functional requirements, and identified use cases. The requirements will define the expected fleet management behavior and will also affect design and implementation decisions. The presented solution analysis comes from current Dronetag user requests, previously analyzed existing solutions, and discussion between the author and other members developing the Dronetag platform.

3.1 Functional Requirements

The functional requirements listed below describe the specific functionality expected from the final solution. Individual requirements are given a code and a short description.

The proposed features try to maintain two key principles. First of all, the solution should constrain user actions as little as possible. Second, there should be as little excessive functionality as possible.

FR01 Organization management

A user without an organization will be able to create an organization. The user who created the organization will become its owner. The owner will be able to change the organization name, description, manage global parameters, and delete the organization. Deleting an organization will result in irreversible deletion of all data related to the organization.

FR02 Organization asset management

Any organization member will be able to transfer his aircraft or device to organization ownership. Aircraft and devices owned by the organization will not be available for personal flights. Any member will also be able to transfer any organization aircraft or device into his ownership.

FR03 Organization member management

Any organization member will be able to invite new members to the organization using their e-mail address. The invited user will be able to accept the invitation via a direct link or in the Dronetag web app if they already have an account and are not a member of an organization. Any member can leave the organization. Any member can be removed from the organization by the organization owner. Leaving the organization irreversibly deletes all data related to the member.

FR04 Organization team management

Team will be a named list of selected organization members. Any organization member can create a team. Any member can also rename a team, add other organization members into a team, or delete a team.

FR05 Mission management

Any organization member will be able to create a mission. The member that created the mission will become the mission owner and the first member with coordinator privileges. The owner retains coordinator privileges even if he is not a member of the mission or is not designated as the mission coordinator explicitly. Coordinators will be able to rename the mission or delete it, assign start and end time, and designate mission area.

FR06 Mission member management

Mission coordinator will be able to add members to the mission, remove members from the mission, and assign pilot and coordinator roles to members in the mission.

FR07 Mission flight planning

Coordinator will be able to create, view, edit, or remove all flight plans of every mission pilot. Mission will consist of multiple flight plans, zero or one for each mission pilot. The flight area of these flight plans must be entirely inside of the mission area.

The mission pilot will be able to view flight plans of other mission pilots, but will be able to only create, edit, or remove his own flight plan.

FR08 Mission flight execution

Mission pilots will be able to execute the planned flight at the specified time in the same way as with a personal flight.

3.2 Non-Functional Requirements

Non-functional requirements describe technical requirements that the final solution is expected to meet. All of them have to be accounted for in the design and implementation, which is why they will only be given a code and description.

NFR01 Functionality will be available via HTTP REST API

The functionality will be available via HTTP protocol and API will be built using REST principles.

NRF02 Functionality will be available in the mobile application

Users will be able to view organization and mission details in the Dronetag mobile application.

NRF03 Functionality will be divided into multiple components

Solution functionality will be logically divided into multiple components so that the application remains maintainable and extendable.

3.3 Use Case Analysis

Modeling system use cases consists of defining actions that a user can perform and defining user roles under which the user will be able to perform these actions. This type of model requires analyzing the application from the user's point of view, which helps to understand the system as a whole and uncovering potentially missed features that the system might require for functioning properly.

The use case model defined here is partially based on preexisting mobile application design (see Appendix A) that defined part of the user workflow regarding fleet management. It was decided that this preexisting design will be reused and adapted to correspond to defined functional requirements. For this reason, the use cases were mostly taken from the application design. The original mobile application design was created by Marián Hlaváč and adapted by Michal Skipala.

3.3.1 Actors

In this section, the actors in the system and their relationship will be identified and described.

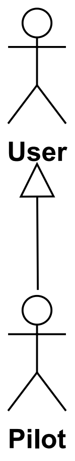


Figure 3.1: Actor Generalization Example

An actor is something or someone that interacts with the target system to produce an observable result. . . . Actor models the roles of real users using the system for different purposes or systems that interact with the software system, such as border system that receives input from the main system. [30]

Most of the roles in the system are specialized cases of a general role, Dronetag user. This is described by *generalization* relationship. In Figure 3.1 an example of a relationship between pilot and user shows that user is a *generalization* of pilot. This notation describes that pilot can perform the same actions as user, however, user cannot perform the same actions as pilot unless he is an instance of pilot himself.

The following actors were identified:

- **User without organization** – standard Dronetag user which is not a member of any organization,
- **Organization member** – standard user in an organization,
- **Organization owner** – user that created the organization of which he is a member of,
- **Mission owner** – user in an organization who created the mission,
- **Mission member** – user in an organization who is a member of the mission,
- **Mission pilot** – member of a mission in which he is designated as a pilot,

- **Mission coordinator** – member of a mission in which he is designated as a coordinator,
- **Timed task executor** – system component that automatically starts planned missions.

3.3.2 Use Cases

This section will describe the business functionality that an actor will be able to use in the system.

> *A use case is one instance of how an actor would use a software system to activate a business function that is a service offered by the system and to produce a result. With use cases it is possible to specify all services offered to users by the system when use cases are related to actors these directly specify functionality of the system. [30]*

Each use case maps to one or more actors in the system. Due to the nature of the system, most use cases correspond to standard create, read, update, delete (CRUD) operations. View operation use cases encompass both listing and detail information viewing, and manage operation use cases group both edit and delete operations, excluding cases where these operations are listed separately.

- Organization management
  - UC01 Create organization
  - UC02 Manage organization
  - UC03 Leave organization
  - UC04 Delete organization
  - UC05 Change organization preferences
- Organization asset management
  - UC06 View organization assets
  - UC07 Transfer assets
- Organization member management
  - UC08 View organization members
  - UC09 Remove organization members
  - UC10 Invite user to organization
  - UC11 Resolve organization invitations

- Organization team management
  - UC12 View teams
  - UC13 Create team
  - UC14 Manage team
  - UC15 View team members
  - UC16 Manage team members
- Mission management
  - UC17 View missions
  - UC18 Create mission
  - UC19 Manage mission
  - UC20 Change mission preferences
- Mission coordination
  - UC21 View mission members
  - UC22 Manage mission members
  - UC23 Assign mission member roles
  - UC24 View mission flight plans
  - UC25 Create mission flight plan
  - UC26 Edit mission flight plan
  - UC27 Delete mission flight plan
- Mission execution
  - UC28 Start mission
  - UC29 End mission
  - UC30 Execute mission flight

3.3.3 Use Case Model

Using the information in the previous sections, actors were associated with corresponding use cases that they will be able to execute in the system. For clarity, this model has been split into two parts, one part shown in Figure 3.2, focusing on organization, asset, and team management, and the other part shown in Figure 3.3, focusing on mission planning and execution.

3.3.4 Use Case Coverage

Functional requirements of a complete system should cover every defined use case. To validate that the system was analyzed and designed correctly, use cases are checked to have at least one corresponding functional requirement that covers them. This validation is shown in Table 3.1.

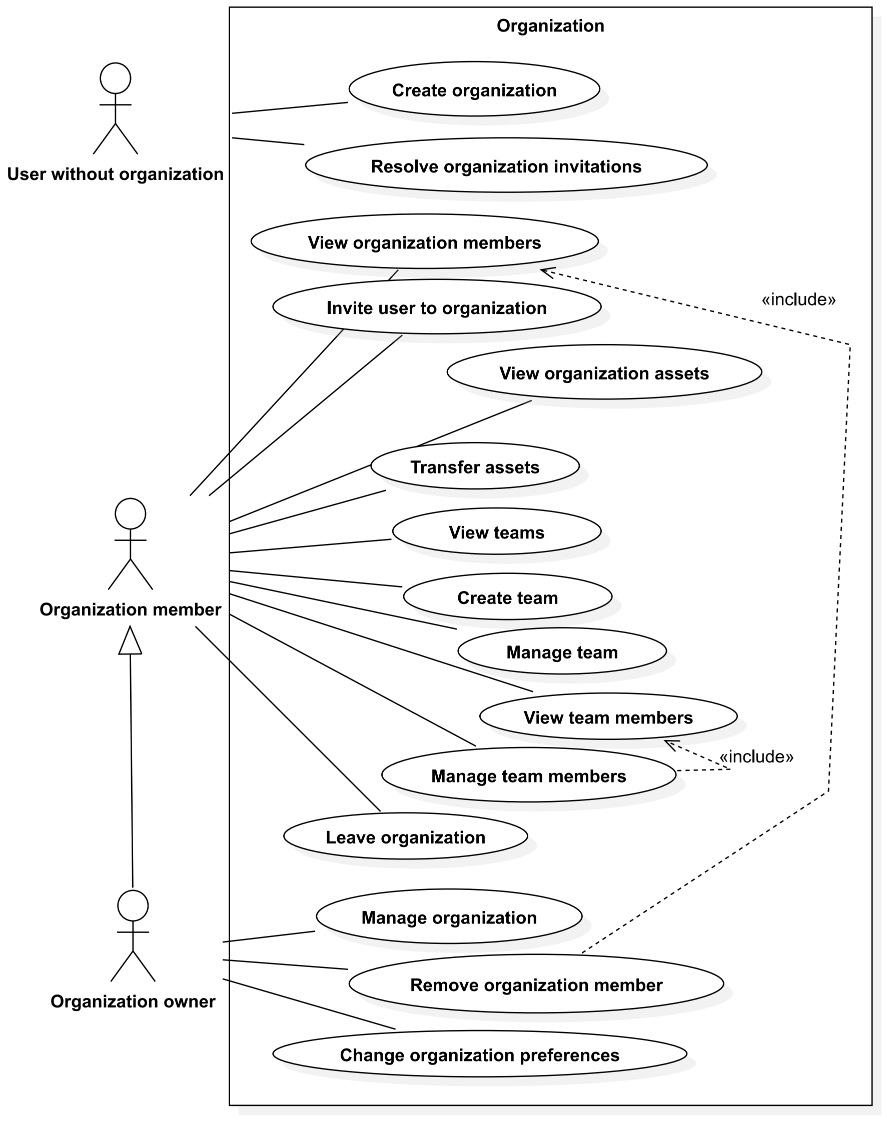


Figure 3.2: Organization Use Case Model

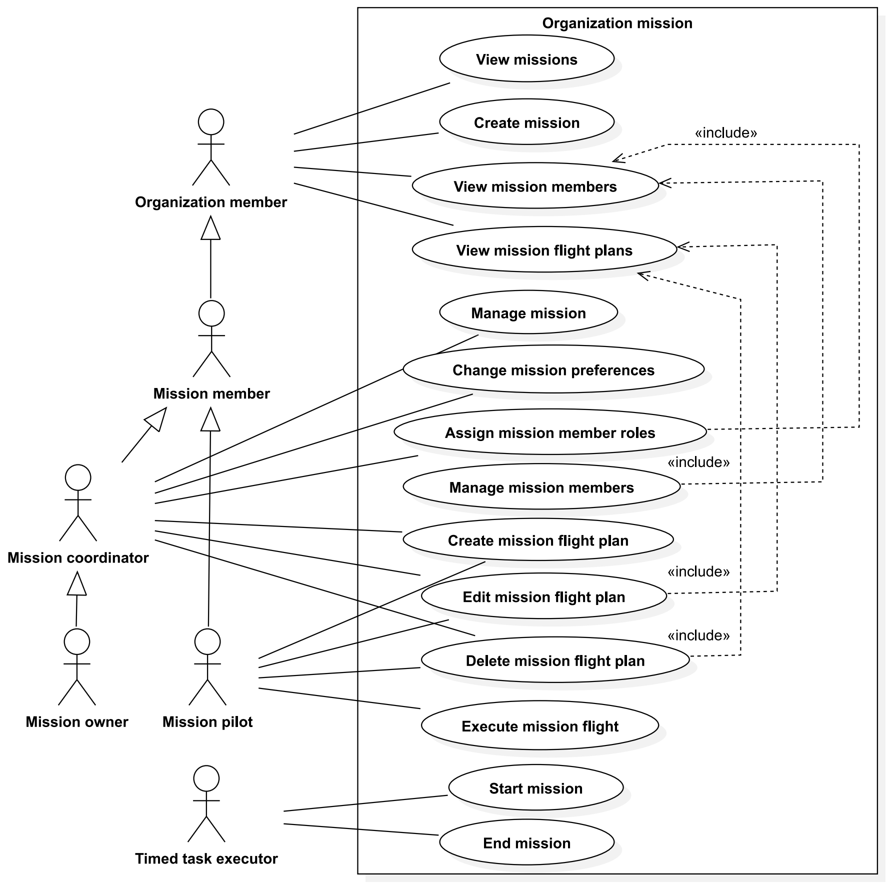


Figure 3.3: Mission Use Case Model

	Functional Requirements							
Use Case	FR01	FR02	FR03	FR04	FR05	FR06	FR07	FR08
UC01	*							
UC02	*							
UC03			*					
UC04	*							
UC05	*							
UC06		*						
UC07		*						
UC08			*					
UC09			*					
UC10			*					
UC11			*					
UC12				*				
UC13				*				
UC14				*				
UC15				*				
UC16				*				
UC17					*			
UC18					*			
UC19					*			
UC20					*			
UC21						*		
UC22						*		
UC23						*		
UC24							*	
UC25							*	
UC26							*	
UC27							*	
UC28								*
UC29								*
UC30								*

Table 3.1: Use Case Coverage

Chapter 4

Design

This chapter will describe the steps taken when designing the fleet management solution, extending the current Dronetag backend and mobile application. As the business logic for the solution will be implemented in the backend part of the Dronetag platform, the domain model extension will first be presented. Afterwards, the design of new modules and extension of existing modules covering the new functionality will be discussed.

4.1 Backend Extension

Given the functional requirements in the analysis presented previously, most of the functionality is to be implemented as an addition to the existing Dronetag backend. The fact that the fleet management solution must be integrated into an existing system had consequences on the design and implementation decisions made. Newly added modules have to make use of existing functionality without making any significant changes to it unless absolutely necessary, since each such change would require a thorough analysis of impact on current platform operation.

With this in mind, the goal is still to integrate the solution into the existing infrastructure as tightly as possible, reusing the existing functionality as much as possible. This is why a certain amount of change to existing modules is expected for successful integration.

4.1.1 Domain Model

The first step in designing the fleet management solution analyzed in the previous chapter was to define new entities in the system. The newly added entities and their relations are visualized in Figure 4.1.

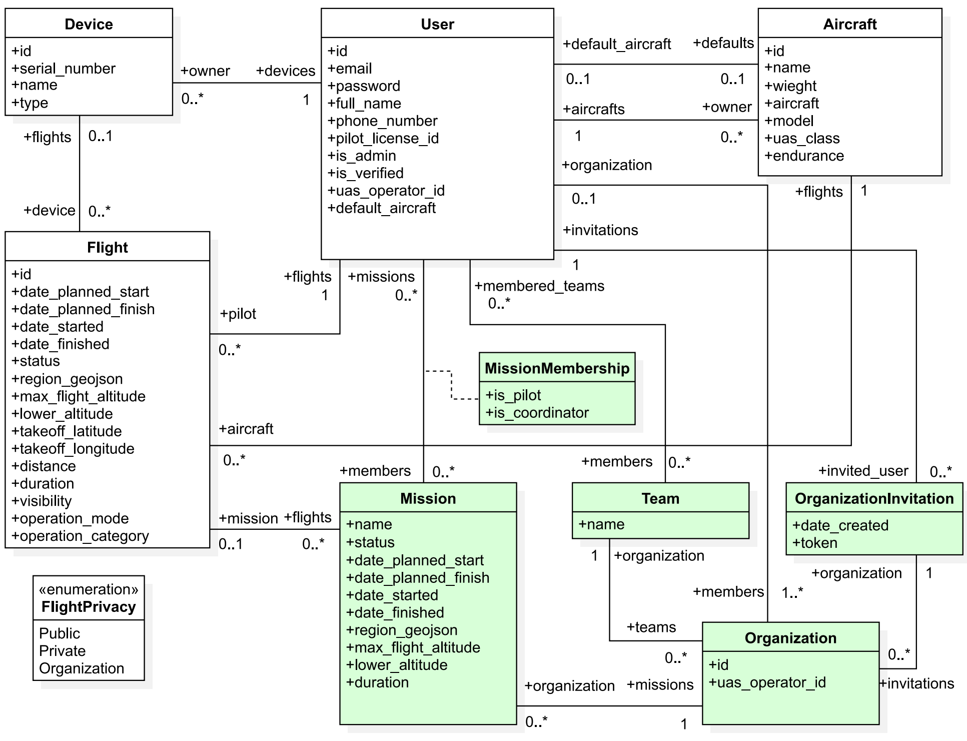


Figure 4.1: Updated Domain Model

4.1.2 Modules

The second step after defining new entities that extend the system is their division into modules. The current Dronetag backend has its functionality divided into multiple apps, structurally isolating the functionality not only on a business logic level, but also on the application logic level. All modules together still form a monolithic system and can depend on each other, but each app is trying to achieve high cohesion and low coupling by following the single-responsibility principle at the module level [31]. This principle had to be respected when defining new modules for the system.

To correctly determine the scope of each module, it was also necessary to consider current modules in terms of their granularity. As the platform is being actively developed by the Dronetag team, it was necessary to coordinate the design with their expectations on each module's scope to make sure the scope is not too broad or too narrow (producing too many small modules or too few large modules) and consistent with current practices.

After discussing possible options with the team, it was decided to split the newly implemented functionality into 3 modules, each taking care of one aspect of fleet management.

- `organization` – This module will be responsible for creating and managing the organization entity itself. Operations for creating, updating, and deleting organization will be implemented here. Additionally, this module will be responsible for the management of organization members and their invitations, providing endpoints for listing organization members and invitations, as well as endpoints for removing members from the organization, inviting new members, and resolving invitations.

- `mission` – This module will take care of the organization missions domain. Basic mission management, such as creating, editing, and deleting missions, will be implemented here. Adding and removing mission members and assigning roles to them will also be in the scope of this module.

- `team` – This module will handle the retrieval and management of teams. That includes listing, creating, editing, and deleting teams, as well as listing, adding, and removing team members.

In addition, existing modules will be modified to handle additional functionality related to fleet management.

- `aircraft` – This module will additionally be responsible for aircraft ownership transfer to and from an organization.

- `device` – Ownership transfer to and from an organization will have to be handled in this module, similarly to the `aircraft` module. Additionally, special handling in case of organization deletion has to be also implemented here.

- `flight` – The flight module will be extended to support mission flight planning. This way, current flight functionality will not be significantly disrupted, and mission flight execution will be the same as personal flight execution.

- `users` – The users module will newly handle organization invitations for users who are not members of any organization.

4.2 Mobile Application Extension

The next step after designing and implementing the backend part of the fleet management solution is the design and implementation of the newly provided functionality into Dronetag mobile application. Once again the design and implementation had to follow Dronetag architectural practices and code style to allow easy cooperation across the whole codebase, new functionality included.

The application architecture in Chapter 2.5.4 provided clear guidance on how to properly extend the application.

4.2.1 Service Layer

As described previously in Chapter 2.5.4, the service layer communicates directly with external data sources used in the application. Since fleet management is provided entirely by the Dronetag backend and is exposed using REST API, only the `BackendApiRestClient` had to be modified to provide access to new or updated HTTP endpoints.

4.2.2 Repositories

The repository layer groups functions for manipulating resources by resource type rather than data source. In most cases, each resource has applicable CRUD operation calls implemented on this layer and additionally provides non-standard operations if there are any. To properly extend this layer, new resources and modified functionality for existing resources had to be identified to design new repositories and modify existing ones. All newly implemented or updated operations are based on the changes described above, since all of them will be provided by `BackendApiRestClient`.

It was decided that 3 new repositories with the following functionality will be created to cover the fleet management functionality.

- **Organization repository**
  - read, create and update[*] operations for organization;
  - organization members list and remove operations;
  - organization invitations list and create operations;
  - organization preferences read and update operations.
- **Mission repository**
  - missions list and mission detail retrieval operations;
  - mission members listing.
- **Team repository**
  - teams list, create, update, and delete operations;
  - team members list, add, and remove operations.

Apart from adding new repositories, existing repositories have to be updated for the new functionality as well. These updates closely resemble the backend changes that had to be done. The **Device repository** and **Aircraft repository** will additionally need to cover new ownership transfer capability and also allow filtering assets (aircraft or devices) by organization or user that owns them. The **Flight repository** also had to be modified to allow filtering listed flights by mission.

[*]It was decided that delete operation will not be available in mobile application.

4.2.3 Features

The feature modules provide a user interface that allows the user to use the functionality of the Dronetag platform. This is why features, unlike repositories and services, are designed according to the user interface prototype rather than the backend API. The user interface defined in the feature modules then displays and allows the user to manipulate the resources defined in the repositories.

The current feature modules in the Dronetag mobile application are built around a small feature scope that is available to the user. They usually consist of a screen or a small set of screens that the user is able to view in the application and code that handles data fetching and user input handling for those specific screens. Feature modules are described in more detail in Chapter 2.5.4.

It was decided that the new functionality will be split into the following 5 modules:

- `organization` – This module will be responsible for showing basic organization information to the user (screens shown in Figure A.2).
- `organization_assets` – This module will be responsible for screens displaying organization assets and transfer of assets to or from organization (contains screens in Figure A.3).
- `organization_creation` – This module will only contain the organization creation process shown in Figure A.1 (designed similarly to existing `onboarding` feature package).
- `organization_members` – This module will contain the list of organization members (screen shown in Figure A.4).
- `mission` – This module will only be responsible for mission functionality, specifically listing missions and displaying detail of an existing mission, as shown in Figure A.7. In the future, this module could also be expanded with additional mission functionality, such as managing mission notifications and other preferences.
- `team` – This module will be responsible for displaying existing teams and managing teams and their members (partial prototypes shown in Figure A.5 and A.6).

4.2.4 Global Layer

To properly integrate previously designed modules into the application, it was necessary to modify two components that provide global functionality. New repositories and BLoCs had to be registered in the service locator to make them available for dependency injection. Then additional routes representing new screens were added to the router configuration, making them available for navigation across the application.

Chapter 5

Implementation

This chapter will focus on implementation of the design described in the previous chapter. It will be split into two parts, the first part describing backend implementation and the second part describing mobile application implementation.

5.1 Backend

Chapter 4.1 described necessary changes to the existing Dronetag backend structure. This section will describe the steps taken to implement new modules, some of the changes to existing modules, and important details of the implementation itself.

5.1.1 App Initialization

The first necessary step was to create new apps described in Chapter 4.1.2 according to the preexisting structure, which was described in Chapter 2.4. Once the structure was established, it was necessary to register each app so that it is properly initialized on backend startup. Django apps are identified and registered using their name defined in the `apps.py` source file (Listing 5.1 shows configuration of the organization app). The apps are then registered in the backend settings directory by adding the application name to the list of installed apps.

```python
from django.apps import AppConfig

class OrganizationConfig(AppConfig):
  name = "backend.organization"
```

Listing 5.1: Organization App Configuration

```
@receiver(pre_save, sender=Mission)
def mission_before_change(sender, instance, *args, **kwargs):
  check_mission_flight_altitudes(instance)
  check_mission_flight_regions(instance)
```

Listing 5.2: Mission Pre-Save Signal Handler Example

5.1.2 Models

The next step after creating and registering all new modules was the definition of data models. As shown in Figure 4.1, models can form relationships with other modules, which is why models had to be created at once, so that all models are available and usable for relationship definitions.

Data models are represented by Python classes which inherit from the `django.db.models.Model` class. Each model class field defines a database column that will be created in the database. Model fields can be configured to set database column constraints (e.g. nullability, default value) or configuration of constructed model instances, for example, inverse relation name for fields representing relation.

In addition to fields, validation rules can also be defined programmatically for each model. Django models even allow defining table-wide constraints on models, meaning that validation rules can also be expressed using SQL queries if necessary. Table-wide constraints were used in the implementation to enforce a single owner for each device or aircraft. The check enforces that both user and organization cannot be set as owners at the same time.

An important feature of Django models is signal dispatching. Each model instance sends a signal before and after it is saved or deleted. These signals can be listened to by any registered app, allowing other parts of the system to react to data changes when necessary. Data change handlers can be decoupled from the model definition, which reduces code complexity of the model itself and allows for better separation of concerns.

One example of signal usage is the mission change handler – when mission changes, it is necessary to validate that all flights still fulfill mission constraints and raise validation exception in case they do not (example of such handler is shown in Listing 5.2). Another perfect usage of signals was scheduling of delayed tasks that are to be run when mission is created or updated.

5.1.3 Endpoints

After defining the database schema, it was necessary to define HTTP endpoints that will be responsible for manipulating previously defined entities. Endpoints are defined for each app in the `urls.py` file where each route is assigned to a single view class. When an HTTP request is received, Django looks up the responsible view and calls the appropriate handler. Whether

```
urlpatterns = [
  re_path(
    r"^$",
    OrganizationCreateView.as_view(),
    name="list_create",
  ),
  path(
    r"<uuid:org_id>", OrganizationByIdView.as_view(),
    name="organization",
  ),
  # and others...
]
```

Listing 5.3: Endpoint Path Registration

the view handles particular HTTP method is entirely dependent on the view implementation and is not handled by the router, as seen in Listing 5.3. This approach pushes more responsibility onto the views, but makes it harder to implement individual HTTP handlers as they have to be implemented inside a single view class, which is not always desirable.

5.1.4 Views

Each view handling HTTP requests had to be implemented in the `views.py` file of each module. The **Django REST Framework (DRF)** library was used for view implementation, reducing the amount of code necessary for standard REST endpoints to an absolute minimum. Views are implemented as classes inheriting from `GenericAPIView` which defines the standard behavior for REST API views. Implementation of each HTTP verb handler is inherited from `GenericAPIView` children, for example, the `ListCreateAPIView` class which provides the `GET` handler returning a list of entities and the `POST` handler for creating a new entity.

The view behavior is configured using the following attributes:

- `queryset` – database entities upon which the view will operate,
- `permission_classes` – list of classes that determine the view and entity access policy,
- `serializer_class` – class defining the input and output data structure and handling,
- `lookup_url_kwarg` – name of the path parameter that contains the primary entity identifier;

Additionally, filtering and ordering behavior can also be configured using view attributes. Since attributes are defined statically and do not depend on the received request, it is also allowed to override attributes with methods (for example, `queryset` can be overridden by the `get_queryset` method, as shown in Listing 5.4). Since methods are called on specific instance of the view, they receive the whole request context and can operate on request data, allowing to filter the queryset depending on user or select serializer class based on HTTP verb that is being handled.

```
class OrganizationMembersView(
  LoggingMixin,
  OrganizationMixin,
  ListAPIView
):
  """
  Implementation of /organizations/{org_id}/members
  """

  permission_classes = [permissions.IsAuthenticated, IsMember]
  serializer_class = OrganizationMemberSerializer

  filter_backends = [OrderingFilter]
  ordering_fields = "__all__"
  pagination_class = LimitOffsetPaginationHeaders

  def get_queryset(self):
    organization = self.get_object(IsMember())

    return organization.members.all()
```

Listing 5.4: Organization Members View

5.1.5 Serializers

An important aspect of request handling are serializers. Serializer classes define input and output data models which might differ from backend data models. Serializer definitions are very similar to model definitions as serializer fields are defined as class attributes. However, to further reduce the amount of code necessary, serializers can be derived from models, defining all the model fields automatically instead of having to define them by hand.

Serializer fields can be marked as read-only to prevent writing to certain model fields, but still return them in response data. The input data can also be programmatically validated in serializers. This allows multiple input

```python
class IsOwner(permissions.BasePermission):
  def has_object_permission(self, request, view, obj):
    # Read access for organization members
    if request.method in permissions.SAFE_METHODS:
      return request.user in obj.members.all()

    if request.user:
      return obj.owner == request.user

    return False
```

Listing 5.5: Organization Ownership Permission Class

definitions and validation rules for one data model. Such an approach is used for handling personal and mission flights, where mission flights use different serializers with different required fields and validation rules, but creating the same type of resource as personal flight.

Serializers also handle the process of creating or updating a model instance, so that the view only looks up existing resource, instantiates the serializer, provides input data to the serializer, and returns the serializer output. In case additional steps are necessary, all the methods can be overridden to add custom behavior. For more complicated logic, entire request handlers are overridden with custom logic, only reusing bits of `GenericAPIView` functionality.

5.1.6 Permissions

Views were first implemented without taking user permissions into account, which simplified testing the initial implementation and reduced possible surface for errors. After the initial implementation was completed, permission classes and other means of enforcing correct behavior were implemented and added to the views. Almost all views check that the current user is authenticated to use the platform. This check is used for all use cases where a regular user can be the actor, and it is implemented using the `permissions.IsAuthenticated` permission class provided by Django.

For more specific permission checks where membership or ownership matters, custom permission classes are implemented. An example of a permission class that checks the ownership of an organization is shown in the Listing 5.5. Permission classes define rules for views by defining the `has_permission` method, which has access to request context, but does not know which resource is being accessed. For this reason, permission can be granted or revoked based on user data or input data, but not based on attributes of the resource being accessed. For resource access permissions, the `has_object_permission` method has to be implemented. This method is called when `get_object` method is called in view to retrieve the resource. The resource object is passed to the `has_object_permission` handler, where access permission can be granted or revoked based on the attributes of the resource object.

5.2 Mobile Application

During design of the mobile application extension the process progressed from the bottom layers of the application upward. The implementation started from the top, implementing user interface first and lower-lever functionality later. Not only did this approach feel more natural from the developer's point of view, but it also limited the amount of unnecessary or unused code because all the code written stemmed from what was directly available to the application's end user.

5.2.1 Features

The mobile application UI is divided into feature modules, and since the modules do not depend on each other, it was not necessary to implement them all at once. For each module, the first step was to implement the screens and lay out the widgets so that the user interface corresponds to the user interface prototype shown in Appendix A. This first implementation did not use any external data and only showed static content without any state management being used in this phase.

With screens and widgets in place, it was necessary to implement cubits that provide state and external input handling logic for each screen and their widgets. As mentioned in Chapter 2.5.2, each cubit requires a state type and an initial state. Dronetag team decided to use abstract class as a base state type with concrete subclasses defining different possible states the screen can have. An example of this hierarchy is shown in Figure 5.1. All cubits used for fleet management implementation use this type of hierarchy, as it allows for easy handling of loading and error states. The initial state of cubits utilizing this hierarchy is usually the state representing the loading phase, as the screen is expected to call cubit's data fetch method upon being displayed, transitioning to the loading state anyway.

Implementation of user and external input handling which affects screen state is also defined in cubits. The user interface is expected to call cubit methods to handle user input and produce state changes that lead to changes in the UI. For example, when the user wants to update current organization data using the pull-to-refresh mechanism, the `loadOrganization` method is called, and `OrganizationLoadInProgress` state is emitted using the `emit` method. To enforce handling of all the state logic inside the cubit, the `emit` method is marked as protected. This ensures that it is only called within cubit methods, as required by the BLoC pattern.

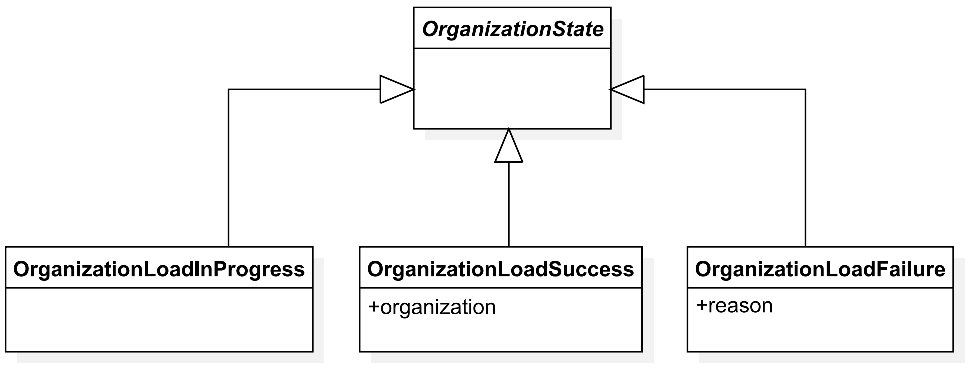


Figure 5.1: Cubit State Hierarchy

```
Future<void> loadOrganization() async {
  emit(OrganizationLoadInProgress());

  try {
    final organization =
      await repository.fetchOrganization(id);

    emit(OrganizationLoadSuccess(organization));
  } catch (exception) {
    emit(OrganizationLoadFailure(exception));
  }
}
```

Listing 5.6: Cubit Load Organization Method Example

5.2.2 Repositories and Backend Communication

Implementing the repositories and the extension of the service layer was very simple and straightforward, as all new functionality was part of the REST API communication layer. For this reason, all the implemented changes in the services layer occurred in the `BackendApiRestClient` class, and the new endpoint calls were implemented the same way as the existing API calls. The repositories then simply grouped the newly created API calls according to the design defined in Chapter 4.2.2.

```
Future<Organization> updateOrganization({
  required String id,
  required String name,
}) async {
  final response = await request(
    HttpMethod.put,
    'organizations/$id',
    jsonBody: {
      'name': name,
    },
    authenticated: true,
  );

  return convertToObject(
    response,
    (o) =>
      Organization.fromJson(o as Map<String, dynamic>),
  );
}
```

Listing 5.7: Update Organization Backend Call

5.3 Mobile Application Limitations

During functional analysis, it was recognized that performing some of the fleet management features in the mobile application made little to no sense. An example of such functionality is mission planning, especially mission region and flight region planning, which requires more precise control and can be done more conveniently on a computer rather than on a smartphone. Mission planning is expected to be done beforehand and usually requires the coordinator to draw multiple flight areas, which would be inconvenient on a smartphone. For this reason, it was decided that for now the mobile application will focus mainly on displaying organization and mission information, while the Dronetag web application will focus on active management.

Additionally, it became apparent that the mobile prototype does not cover everything necessary for the proper implementation and presentation of the fleet management solution to the user. After consulting the supervisor, it was decided that designing user interface is out of scope of this thesis and that the functionality of the mobile application will be reduced accordingly.

Chapter 6

Evaluation

This chapter will describe the processes used for evaluation of the backend implementation and the implementation of the mobile application, and discussion of future extensions to the developed solution. The first section will talk about the backend testing processes that were used to evaluate the implemented solution. The second section will then explain the user testing methodology used for the mobile application and the reason why automated testing was not used. The third part will then propose possible improvements and additions that could be developed in the future.

6.1 Backend

After implementing the backend part of the solution, it was necessary to validate that the implementation correctly follows the requirements defined in Chapter 3. This validation was done using a combination of two methods, automated testing and manual validation of functionality using applications that utilized the newly implemented features.

In [32] Steve McConnell defined the following test categories:

Unit testing *is the execution of a complete class, routine, or small program that has been written by a single programmer or team of programmers, which is tested in isolation from the more complete system.*

Component testing *is the execution of a class, package, small program, or other program element that involves the work of multiple programmers or programming teams, which is tested in isolation from the more complete system.*

Integration testing *is the combined execution of two or more classes, packages, components, or subsystems that have been created by multiple programmers or programming teams. This* *kind of testing typically starts as soon as there are two classes to test and continues until the entire system is complete.*

***System testing** is the execution of the software in its final configuration, including integration with other software and hardware systems. It tests for security, performance, resource loss, timing problems, and other issues that can't be tested at lower levels of integration.*

Before the implementation of the fleet management solution, the Dronetag platform featured a limited number of automated tests for its existing functionality. These tests would most likely be classified as unit tests. Tests falling under other category, especially automated integration or system tests, were not recognized in the codebase and for this reason only unit testing was used for automated validation of code.

6.1.1 Unit Testing

The Django framework provides the necessary functionality and tooling for convenient unit testing of application code [33]. The **DRF** library then also provides tools for easier testing of REST API part of the backend [34].

The structure of tests themselves was laid out according to the recommended Django testing practices [33] and the existing conventions used by the Dronetag team. This lead to creation of separate test modules for each functional module in each new app (e.g. `test_views.py`, `test_models.py`). Much of the testing setup needed was reused from existing code in the `common` module. Since the `common` module had also provided test data for all the backend unit tests, it was necessary to add new test data for fleet management there.

The most important part of unit testing was testing of view modules of each new app. A test case class inheriting from **DRF**'s `APITestCase` was created for each class inside of a view module. Various test cases that validate the functional requirements defined in Chapter 3.1 were then implemented with the help of existing API tests. An example of such test is shown in the Listing 6.1.

```python
def test_delete_unauthorized(self):
  """Unauthorize users cannot delete organization"""
  url = reverse(
    "organizations:organization",
    kwargs={
      "version": "v1",
      "org_id": Organization.objects.first().id
    },
  )
  client.force_authenticate(user=None)
  response = client.delete(url)
  self.assertEqual(
    response.status_code,
    status.HTTP_401_UNAUTHORIZED,
    response.data,
  )
  self.assertEqual(error_unauthorized, response.data)
```

Listing 6.1: API Test Example

6.2 Mobile Application

The evaluation of the mobile application was performed exclusively by manual testing done by the developer in the first phase of testing and different types of users in the second testing phase. After consulting the supervisor, it was decided that automated testing will not be used for the mobile application, as Dronetag currently does not have any testing methodology implemented for the Flutter mobile application, and performing research on testing methodology and implementation of one would be outside of scope of this thesis.

6.2.1 User Test Scenarios

For comparability and consistency between user tests performed, a set of test scenarios that the user will go through was created. Each scenario tests a particular use case or multiple use cases, validating that they were implemented properly. Scenarios were performed one by one and each user was asked to perform its listed steps. Users were observed and were not provided with any assistance unless absolutely necessary. After each test scenario, they were asked for feedback on user experience of the feature tested. The collected information was then used to evaluate the necessary changes to the concept of fleet management and the design of the mobile application UI.

TS01 Create organization

In this test case user is is logged in the application, starts on the application dashboard, and is not currently member of any organization. His task is to find where to create a new organization inside the application. Once the user finds the form, he must create an organization named “Drone” and invite an additional member with e-mail address `m1@dronetag.cz`. The test scenario ends successfully when the user finishes the creation process and enters the organization detail screen.

TS02 Rename organization

This test scenario starts on the organization detail screen. The user is asked to find the screen where the organization name can be changed. When the user successfully enters the screen, he is asked to rename the organization to “Drone Ops” and go back to the organization detail screen. The test scenario ends successfully when the user gets back to the organization detail screen.

TS03 Transfer assets

In this test case the user starts on the organization detail screen. His user account currently owns two aircraft and two devices. The user is asked to transfer aircraft named “Drone Ops Aircraft” and device named ‘’‘Drone Ops Device” to organization ownership. Once the transfer is completed, the user is asked to view the list of assets currently owned by the organization, at which point the test case successfully ends.

TS04 Invite new member

In this test case, the user starts on the organization detail screen and has to view list of current organization members. After successfully listing organization members, the user has to invite new organization member with e-mail `m2@dronetag.cz` into the organization. This test case ends when the user is successfully invited.

TS05 View organization missions

In this test case, the user starts on the organization detail screen. The user is asked to find a list of organization missions and identify missions which are currently in progress. The user is then asked to open detailed information about finished mission and identify coordinators in the mission. After that, the user has to display the flight plan of a mission pilot who is not a coordinator. The user is then asked to describe what he sees on the map. After that, the user has to find the UI option to hide all flight plans at once. The test scenario ends successfully when the user hides all the flight plans displayed using the “Hide all flight plans” menu option.

TS06 Leave organization

This test scenario starts on organization detail screen, where the user is asked to leave the organization he is currently in. The test scenario ends successfully when the user successfully leaves the organization.

6.2.2 User Testing Output

The user testing scenarios described above were designed to evaluate user experience and uncover potential problems in the user interface design, logic of implemented user interactions, or usage patterns that were not recognized during analysis and design. After going through the defined scenarios with professional pilots, experienced Dronetag users and users who had never used Dronetag before, multiple usability issues and also a few minor bugs were identified.

The following user experience inconveniences arisen:

- When creating a new organization, after a user is invited, keyboard is not dismissed and covers the “Continue” button, preventing the user from finishing the process.
- Users expect the “Save” button on “Manage organization” to take them back to the “Organization” screen.
- “Transfer assets” button on the “Assets” screen is difficult to find for users inexperienced with the Dronetag mobile application.
- Users generally try to find asset transfer option under that particular asset’s detail screen outside of the organization management.
- The test subjects expected list of invited users on the “Members” screen (this was included in the original prototype, but the implementation is awaiting improved design).
- Experienced pilots expressed that the button to show or hide all mission plans seems to be unnecessary and that they would at least expect it near the mission member list.
- Users would generally like to further distinguish pilots from other mission members.

Incorrect behavior found:

- When creating a new organization, the text field for invited user’s e-mail does not show validation errors, even though the validation itself works correctly.

- Changing the organization name would not propagate to the “Profile” screen.

Apart from the issues mentioned above, most of the test subjects said that the mission detail screen seemed intuitive and easy to use. The “Members” screen functionality also seemed sufficient for users, and process for leaving the organization felt straightforward to them.

User feedback from the user testing performed provided valuable feedback for future user interface design iterations and will help with the development of the platform in the future. Thanks to feedback from different types of users, the Dronetag team can make the platform more accessible for everyone. Testing also helped uncover incorrect behavior that was not recognized during development.

6.3 Future Work

The implemented solution, as described in this thesis, will already provide major usability improvement for commercial subjects using the Dronetag platform, once it is generally available. Still, this is only the beginning for Dronetag fleet management.

The development process of the fleet management solution showed a clear vision of how the system should operate, and the implemented backend extension provides solid base upon which the platform’s commercial customer operations can build. However, current Dronetag user interface design of the mobile application, as well as the web application is not entirely prepared for the rollout of fleet management to the public. The next steps should therefore be the proper adaptation of the Dronetag design concept for the new fleet management functionality and implementation of this new design. Properly finalizing the design will also allow implementation of team functionality for the mobile application.

From a functional point of view, development in the short term should primarily focus on improving current functionality and its user experience, not on adding new features. An exception to this would be implementation of organization teams for the mobile application and implementation of real-time notifications about mission events, as this feature was proposed as an addition during initial analysis, provides great added value to end users, and already has initial support in the implemented backend part of the solution. Additionally, the backend architecture would benefit from separating business logic into an entirely new layer, as fleet management added a great amount of complexity which was sometimes difficult to handle using the current architecture.

In the long term, fleet management offers many possible ways for future expansion, which is most likely going to be driven by customer requests, and current fleet management design and implementation is built with this in mind. Inspiration for new features could be, for example, taken from solutions analyzed in Chapter 1. One such feature that was discussed during solution analysis was drone battery management, as it would allow organizations to easily track battery wear and manage maintenance cycles, but there are many other possible features to choose from.

Conclusion

A detailed analysis of existing fleet management solutions and an assessment of their capabilities was presented. This analysis was then applied during the design of the new fleet management solution.

After analyzing existing fleet management solutions, the current state of the Dronetag platform was analyzed to obtain an overview of how the platform works internally and what the constraints are for the proposed solution. This analysis covered both a high-level overview of the entire platform and a more in-depth analysis of the Django backend and the Flutter mobile application, where the fleet management solution was implemented. The code structure and conventions used were identified so that the implemented solution follows the Dronetag team code style.

Taking into account previous analyses, Dronetag team member remarks, and Dronetag user feedback, the new fleet management solution was analyzed, and its requirements were collected.

The collected requirements were then used to design an extended Dronetag backend architecture that would provide fleet management functionality. With the required functionality and backend design available, mobile application extensions were designed in a similar fashion. However, the scope of the mobile application was stripped of functionality that was deemed unnecessary at the time, for example, mission planning.

Fleet management was then implemented according to the previously defined design. New modules handling designed features were created on the backend and a new section of the Flutter mobile application was added, utilizing these new backend features.

After the implementation of the solution on the backend was finished, unit tests were written for the backend implementation. Because there was no existing testing methodology for the Flutter mobile app, automated testing was omitted for the mobile application. The final prototype of the mobile application was then tested with real pilots, who presented their opinions comments about usability and possible improvements of the implemented solution.

Possible future expansion, taking feedback into account, was presented in Chapter 6.3. Current backend implementation already supports vast majority of required functionality, however, additional changes regarding user experience proposed by users in testing might require additional backend changes. The scope of implemented mobile application functionality was limited, and further expansion is planned for later date.

The designed solution provides a good base for future expansion of functionality, and the implemented functionality simplifies fleet management, even though the implementation was just a prototype that will be revised. Further steps that would allow finalization of the solution and its availability to users were also presented in this thesis; however, it will still take additional time for that to happen due to involvement of external factors.

Bibliography

1. FEDERAL AVIATION ADMINISTRATION. *FAA Aerospace Forecast Fiscal Years 2021-2041* [online]. FAA Aerospace Forecasts, 2021. [visited on 2022-04-24]. Available from: `https://www.faa.gov/data_research/aviation/aerospace_forecasts/media/FY2021-41_FAA_Aerospace_Forecast.pdf`.
2. FLYFREELY PTY LTD. *Drone Management Platform* [online]. FlyFreely Drone Management Platform, 2022. [visited on 2022-05-11]. Available from: `https://www.flyfreely.io/`.
3. DRONELOGBOOK. *DroneLogbook* [online]. DroneLogbook, 2022. [visited on 2022-05-11]. Available from: `https://www.dronelogbook.com`.
4. FLYFREELY PTY LTD. *About* [online]. FlyFreely Drone Management Platform, 2021. [visited on 2022-04-24]. Available from: `https://www.flyfreely.io/about/`.
5. FLYFREELY PTY LTD. *Pricing* [online]. FlyFreely Drone Management Platform, 2021. [visited on 2022-04-24]. Available from: `https://www.flyfreely.io/pricing/`.
6. SKIPALA, Michal. *Implementation of the new module into the Dronetag web application for planning, managing and coordinating drone fleets.* Praha, 2022. Bachelor's Thesis. Czech Technical University in Prague, Faculty of Information Technology.
7. DRONETAG S.R.O. *Dronetag Mini* [online]. Dronetag, 2022. [visited on 2022-05-06]. Available from: `https://dronetag.cz/products/mini/`.
8. THE POSTGRESQL GLOBAL DEVELOPMENT GROUP. *PostgreSQL: The world's most advanced open source database* [online]. Postgresql.org, 2019. [visited on 2022-05-11]. Available from: `https://www.postgresql.org/`.

9. DJANGO SOFTWARE FOUNDATION. *The Web framework for perfectionists with deadlines* [online]. Djangoproject.com, 2019. [visited on 2022-05-11]. Available from: `https://www.djangoproject.com/`.
10. HOGUIN, Loïc. *REST principles* [online]. Nine Nines, 2018. [visited on 2022-05-11]. Available from: `https://ninenines.eu/docs/en/cowboy/2.8/guide/rest_principles/`.
11. DJANGO SOFTWARE FOUNDATION. *Django overview* [online]. Djangoproject.com, 2019. [visited on 2022-04-24]. Available from: `https://www.djangoproject.com/start/overview/`.
12. PYTHON SOFTWARE FOUNDATION. *What is Python? Executive Summary* [online]. Python.org, 2019. [visited on 2022-04-24]. Available from: `https://www.python.org/doc/essays/blurb/`.
13. SOLEM, Ask. *Periodic Tasks* [online]. Celery 5.2.6 Documentation, 2021. [visited on 2022-05-07]. Available from: `https://docs.celeryq.dev/en/stable/userguide/periodic-tasks.html`.
14. DJANGO REST FRAMEWORK COMMUNITY. *django-rest-framework* [online]. GitHub, 2022. [visited on 2022-05-07]. Available from: `https://github.com/encode/django-rest-framework`.
15. SENDGRID. *sendgrid-python* [online]. GitHub, 2022. [visited on 2022-05-07]. Available from: `https://github.com/sendgrid/sendgrid-python`.
16. GILLIES, Sean. *The Shapely User Manual* [online]. Shapely 1.6 documentation, 2022. [visited on 2022-05-07]. Available from: `https://shapely.readthedocs.io/en/stable/manual.html`.
17. FRANZEL, T. *drf-spectacular* [online]. GitHub, 2022. [visited on 2022-05-07]. Available from: `https://github.com/tfranzel/drf-spectacular`.
18. FLUTTER. *Flutter - Beautiful native apps in record time* [online]. Flutter.dev, 2019. [visited on 2022-04-25]. Available from: `https://flutter.dev/`.
19. FLUTTER. *FAQ* [online]. Flutter, 2022. [visited on 2022-05-05]. Available from: `https://docs.flutter.dev/resources/faq`.
20. DART COMMUNITY. *Dart Overview* [online]. dart.dev, 2022. [visited on 2022-04-27]. Available from: `https://dart.dev/overview`.
21. FLUTTER. *Flutter Architectural Overview* [online]. Flutter, 2021. [visited on 2022-05-05]. Available from: `https://docs.flutter.dev/resources/architectural-overview`.
22. ANGELOV, Felix. *Bloc* [online]. GitHub, 2022. [visited on 2022-05-10]. Available from: `https://github.com/felangel/bloc/blob/e05a8cbfbf50dce35af4a2c66a53a1416b362613/packages/bloc/README.md`.

23. SOARES, Paolo. *Flutter / AngularDart – Code sharing, better together* [online]. YouTube, 2018. [visited on 2022-05-04]. Available from: `https://www.youtube.com/watch?v=PLHln7wHgPE`.

24. ANGELOV, Felix. *Why Bloc?* [online]. Bloc State Management Library, 2020. [visited on 2022-05-04]. Available from: `https://bloclibrary.dev/#/whybloc`.

25. ANGELOV, Felix. *Merge Cubit into Bloc* [online]. GitHub, 2020. [visited on 2022-05-04]. Available from: `https://github.com/felangel/cubit/issues/69`.

26. ANGELOV, Felix. *Core Concepts* [online]. Bloc State Management Library, 2021. [visited on 2022-05-05]. Available from: `https://bloclibrary.dev/#/coreconcepts`.

27. MARTIN, Robert C. *Clean Architecture: a craftsman's guide to software structure and design.* Prentice Hall, 2018.

28. FOWLER, Martin. *Inversion of Control Containers and the Dependency Injection pattern* [online]. martinfowler.com, 2004. [visited on 2022-05-05]. Available from: `https://martinfowler.com/articles/injection.html`.

29. BURKHART, Thomas; COMMUNITY, Flutter. *GetIt* [online]. GitHub, 2022. [visited on 2022-05-05]. Available from: `https://github.com/fluttercommunity/get_it`.

30. JALLOUL, Ghinwa. *UML by Example.* Cambridge University Press, Cop, 2004.

31. MARTIN, Robert C. *Clean code: a handbook of agile software craftsmanship.* Prentice Hall, 2009.

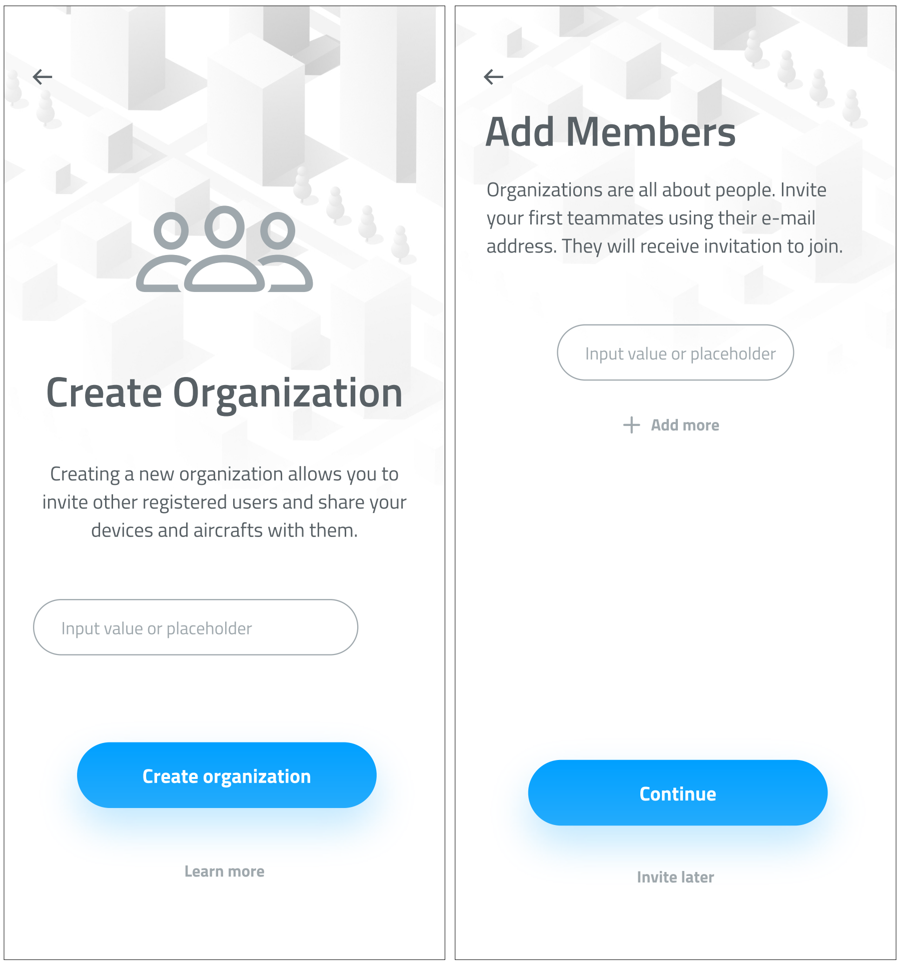
32. MCCONNELL, Steve. *Code Complete.* 2nd ed. Microsoft Press, 2004.

33. DJANGO SOFTWARE FOUNDATION. *Writing and running tests* [online]. Django documentation, 2022. [visited on 2022-05-09]. Available from: `https://docs.djangoproject.com/en/4.0/topics/testing/overview/`.

34. DJANGO REST FRAMEWORK COMMUNITY. *Testing* [online]. Django REST framework, 2022. [visited on 2022-05-09]. Available from: `https://www.django-rest-framework.org/api-guide/testing/`.

Appendix A

User Interface Prototypes



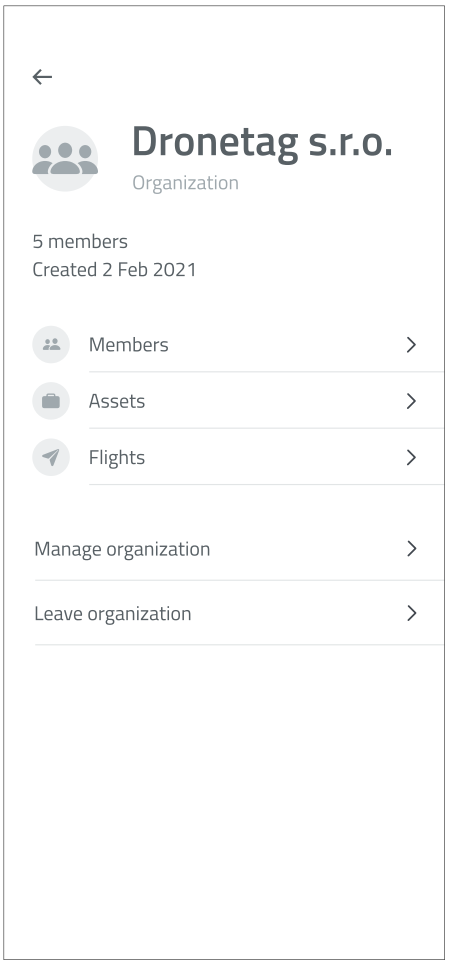
Create Organization

Creating a new organization allows you to invite other registered users and share your devices and aircrafts with them.

Add Members

Organizations are all about people. Invite your first teammates using their e-mail address. They will receive invitation to join.

Figure A.1: Create Organization Process Prototypes



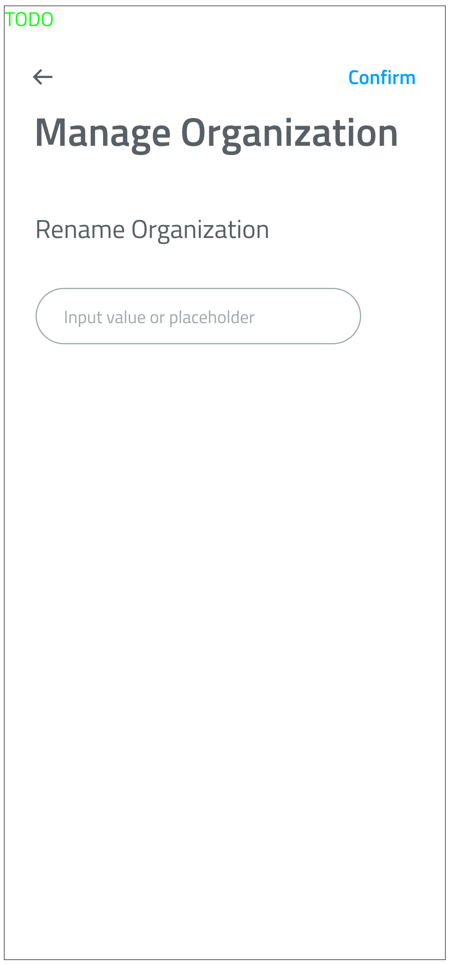
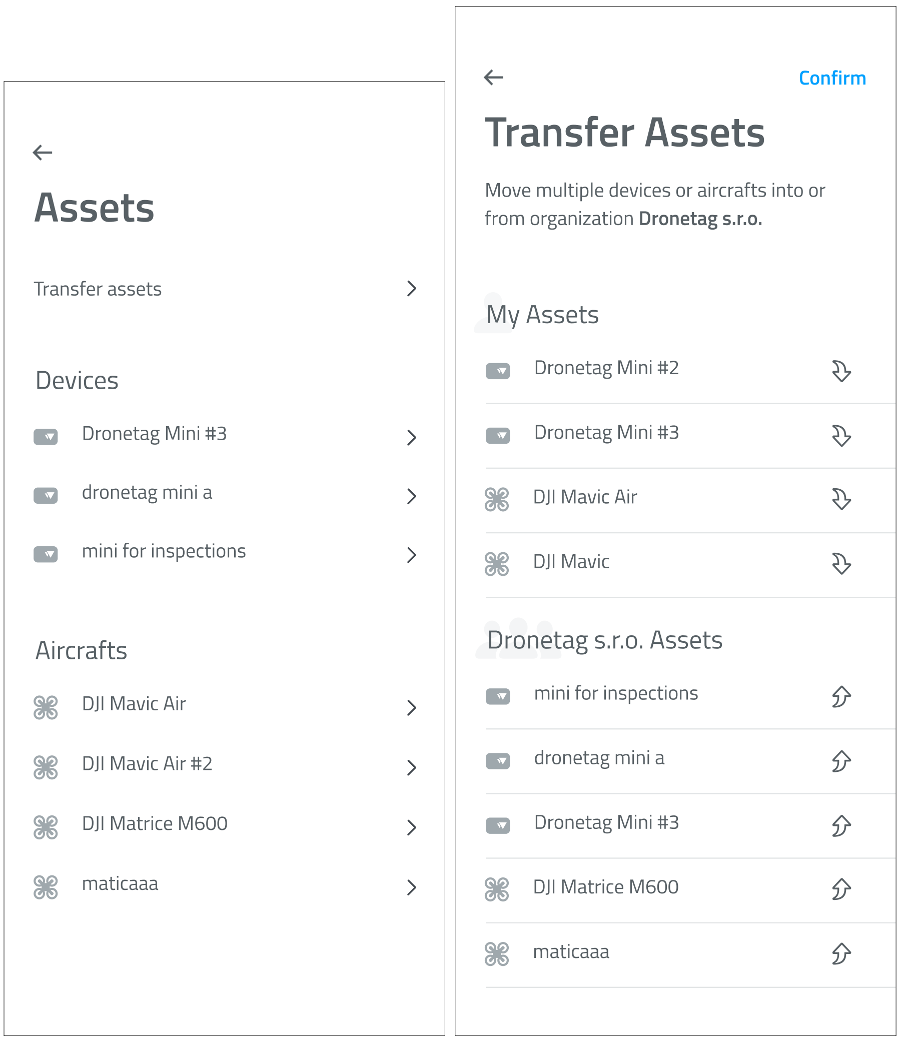


Figure A.2: Organization Detail and Management Prototypes



Assets

Transfer assets

Devices

Dronetag Mini #3

dronetag mini a

mini for inspections

Aircrafts

DJI Mavic Air

DJI Mavic Air #2

DJI Matrice M600

maticaaa

Confirm

Transfer Assets

Move multiple devices or aircrafts into or from organization **Dronetag s.r.o.**

My Assets

Dronetag Mini #2

Dronetag Mini #3

DJI Mavic Air

DJI Mavic

Dronetag s.r.o. Assets

mini for inspections

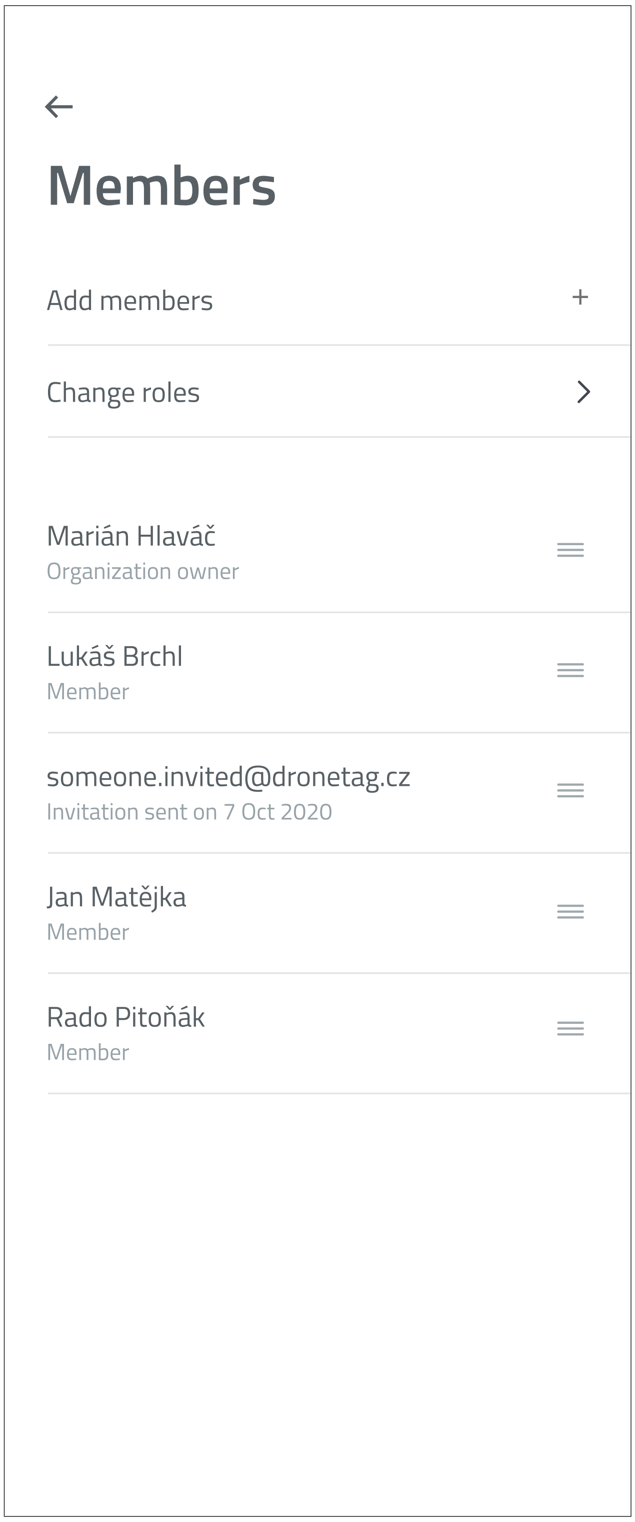
dronetag mini a

Dronetag Mini #3

DJI Matrice M600

maticaaa

Figure A.3: Organization Asset Management Prototypes



Members

Add members

Change roles

Marián Hlaváč
Organization owner

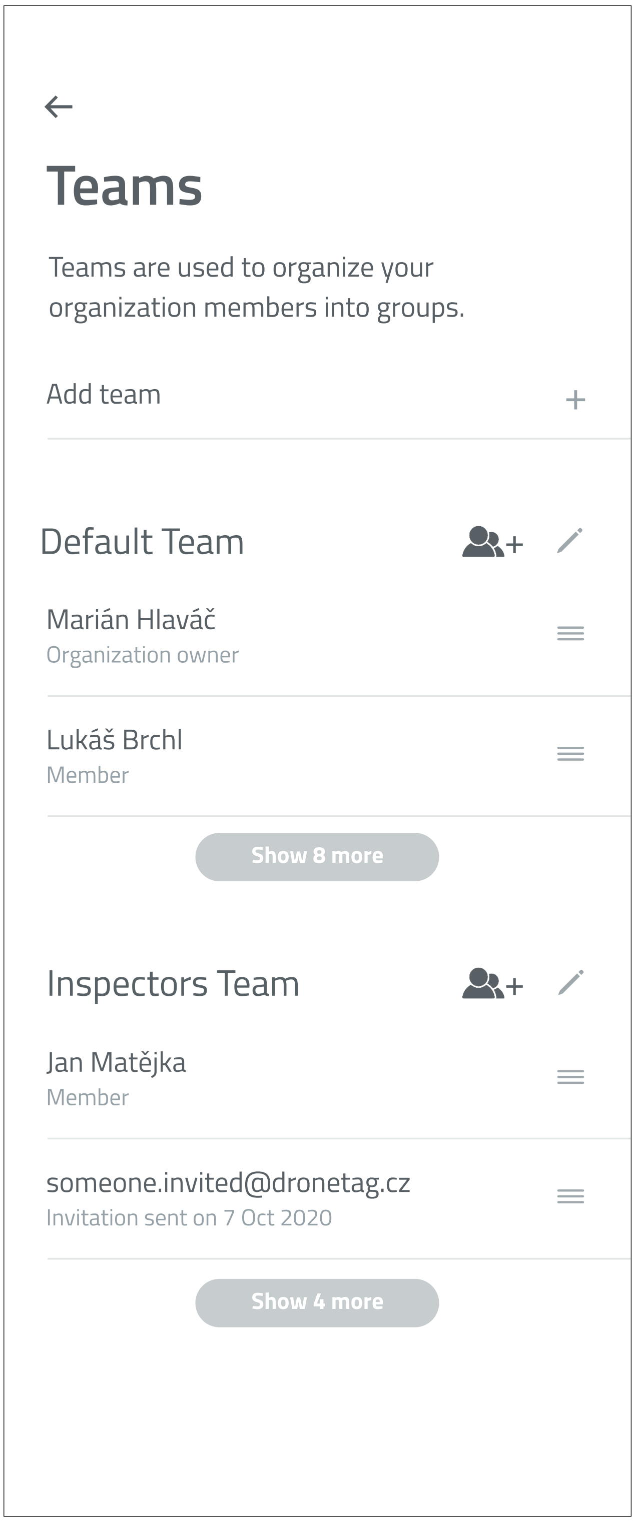
Lukáš Brchl
Member

someone.invited@dronetag.cz
Invitation sent on 7 Oct 2020

Jan Matějka
Member

Rado Pitoňák
Member

Figure A.4: Organization Members Screen Prototype



Teams

Teams are used to organize your organization members into groups.

Add team

Default Team

Marián Hlaváč
Organization owner

Lukáš Brchl
Member

Show 8 more

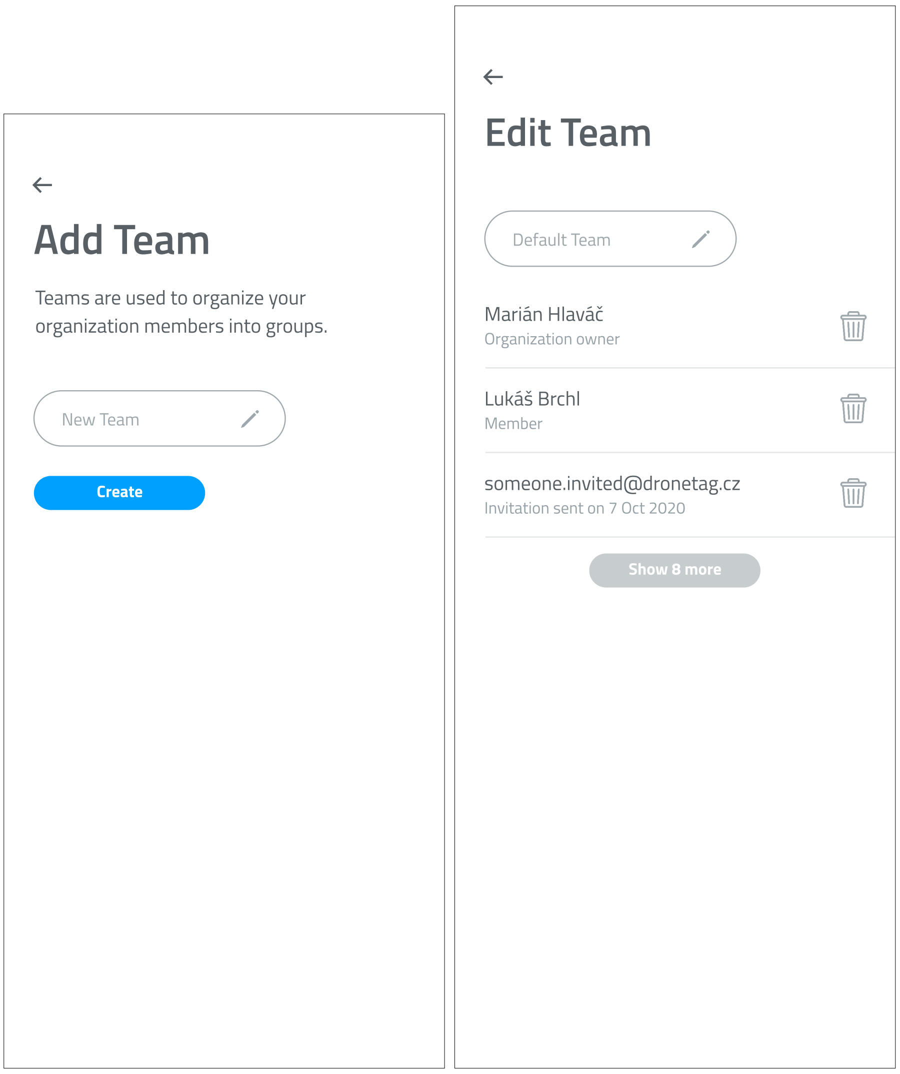
Inspectors Team

Jan Matějka
Member

someone.invited@dronetag.cz
Invitation sent on 7 Oct 2020

Show 4 more

Figure A.5: Team Detail Screen Prototype



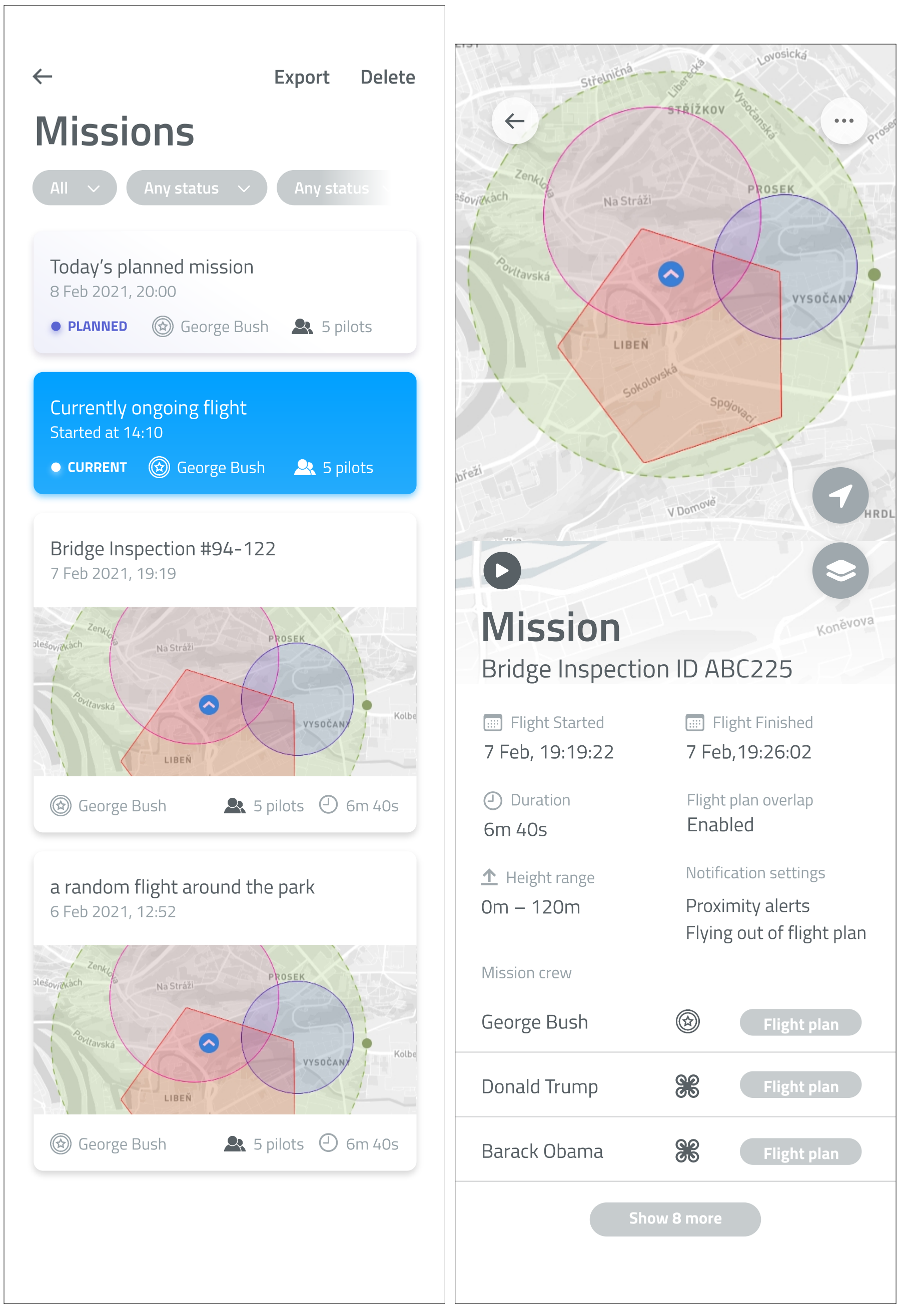
Teams are used to organize your organization members into groups.

Marián Hlaváč
Organization owner

Lukáš Brchl
Member

someone.invited@dronetag.cz
Invitation sent on 7 Oct 2020

Figure A.6: Team Add and Edit Prototypes



Export Delete

Missions

All · Any status · Any status

Today's planned mission
8 Feb 2021, 20:00
PLANNED · George Bush · 5 pilots

Currently ongoing flight
Started at 14:10
CURRENT · George Bush · 5 pilots

Bridge Inspection #94-122
7 Feb 2021, 19:19
George Bush · 5 pilots · 6m 40s

a random flight around the park
6 Feb 2021, 12:52
George Bush · 5 pilots · 6m 40s

Mission
Bridge Inspection ID ABC225

Flight Started: 7 Feb, 19:19:22	Flight Finished: 7 Feb,19:26:02
Duration: 6m 40s	Flight plan overlap: Enabled
Height range: 0m – 120m	Notification settings: Proximity alerts, Flying out of flight plan

Mission crew

George Bush	Flight plan
Donald Trump	Flight plan
Barack Obama	Flight plan

Show 8 more

Figure A.7: Organization Mission Prototypes

APPENDIX **B**

Acronyms

API application programming interface.

BLoC business logic component.

BLOS beyond visual line of sight.

CoAP Constrained Application Protocol.

CRUD create, read, update, delete.

DIP dependency inversion principle.

DJI Da-Jiang Innovations.

DRF Django REST Framework.

EVLOS extended visual line of sight.

HTTP Hypertext Transfer Protocol.

ID identifier.

REST Representational State Transfer.

UAS unmanned aircraft system.

UI user interface.

URL Uniform Resource Locator.

VLOS visual line of sight.

Appendix C

Contents of Enclosed SD Card

```
readme.txt ............ description of the SD card contents
src ......................... the directory of source codes
  thesis ...... the directory containing LaTeX source codes of the thesis
text ....................... the thesis text directory
  thesis.pdf ............... the thesis text in PDF format
```