



**UiO • University of Oslo**

# FlyNerd

---

Group project for course IN2000 Spring 2021

**Team 16**

**Bence Kalmar**, bencek

**Gert-Jan Haasnoot**, gertjanh

**Mohammed Abubaker Mohammed**, mohmo

**Piotr Koktkowski**, piotrkot

**Stela Ceaicovscaia**, stelac

**Yan Jiang**, yanji

project conducted under supervision of

**Sofie Tjønneland Urhaug**

## Table of Contents

Presentation.....	5
Team .....	6
User documentation .....	8
App's functionalities .....	8
Target group.....	8
How to start .....	8
How to navigate in Flynerd? .....	9
How to check the airplanes currently in the sky?.....	9
How to search for a flight?.....	10
How to check the airport weather of a given airport? .....	10
What else does the app offer?.....	10
Requirements specification and modelling .....	11
Brainstorming the project.....	11
Activity diagram .....	13
User stories and backlog.....	14
Majors use cases .....	16
Sequence diagram.....	21
Functional and non-functional requirements.....	21
Functional requirements.....	21
Non-functional requirements .....	22
Inclusive design .....	23
Colour and contrast .....	23
Navigation and hierarchy.....	24
Typography .....	25
Class diagram .....	26
Design patterns .....	26
Object-oriented principles and GRASP .....	26
Model-View-ViewModel .....	28
Repository .....	29
Product documentation .....	31
Coroutines and lifecycles .....	31
Lifecycles .....	32
Libraries.....	33
Fuel.....	33
Retrofit .....	33

Gson .....	34
Kotlin-csv .....	34
Java 1.8 desugaring .....	34
Lottie .....	35
Front-end .....	35
Homepage screen – the MapActivity .....	35
Flight status screen .....	35
Airport Weather screen .....	36
APIs .....	36
Google Maps .....	36
OpenSky Network .....	38
FlightStats .....	39
LocationForecast API .....	42
Test documentation .....	45
Static testing .....	45
Dynamic testing .....	46
Unit testing .....	47
Integration testing .....	48
System testing .....	48
Process documentation .....	49
Methodologies used in the project .....	49
Course of the project .....	51
Start phase .....	52
Development phase .....	52
Grooming phase .....	52
Reflections .....	53
Teamwork .....	53
Methodologies and project management .....	54
The app .....	55
The project in general .....	55
Influence of Corona situation .....	55
Requirements met .....	56
Final words .....	59
References .....	60
Attachments .....	63
Group contract .....	63

Interview questions .....	63
Interview consent form.....	64
Class diagram .....	65

## Presentation

Our group had its first meeting with all members present a few days after the kick-off. A few of us had attended the event and decided that it would be best to pick a case from the list of suggested cases. We created a google spreadsheet where each of us could cast their vote. In the end we decided to pick the FlyNerd case because apparently a lot of us liked planes. The original case description gave us the impression that the app was meant to be primarily targeted to people with a special interest in planes. Our first thought was plane spotters or drone enthusiasts. After conducting some initial interviews, we found out that majority of users are not plane spotters. None of us had a particular interest with planes either (besides travelling of course). This caused us to change our approach and make an app geared towards the average traveller.

Our app is designed to make traveling in a soon-to-come-post-corona-age more convenient. At some point, we all had to look up information as part of the travel planning process. In general, this process can be very exhausting. "what is the weather going to be like at place x", "what type of clothing should I pack", "is my plane going to be delayed?" and other questions require us to visit several websites to collect the information we need. Our app collects all this information and stores it at a convenient place – your pocket. You only need a flight number to see the current weather conditions at both the inbound and outbound airports as well as information about the flight. It can even display a list of your connecting flights to always have up-to-date information.

Furthermore, our app also has a map which displays all airborne planes in Norwegian airspace. We mainly did this to scratch a curiosity some people might have. We can envision situations where our plane is delayed and want to kill some time. The map can be a fun little distraction to not only kill some time, but also learn something about aviation. You can press the plane marker on the map to get some more information about the plane.

Finally, this app can also be very useful for your friends and family. If they have a flight number, they can check when the plane is going to land and whether it is delayed. In short, our app helps everyone to plan their day and save time to spend on more important matters.

## Team

Our team consisted of following members:

### **Bence**

3<sup>rd</sup> year student in bachelor programme Informatics: Programming and System Architecture. No prior practical experience with team-based software projects, agile methodologies, or android development.

### **Gert-Jan**

2<sup>nd</sup> year student in the bachelor program programming and system architecture. Has no prior experience with Android development. Interested in “human” side of project development. Appreciated the opportunity to work in a team and apply what we have learned these past two years.

### **Mohammed A. Mohammed**

2nd year student in Programming and System Architecture bachelor programme. Mohammed is one of the students who likes creative work and has high enthusiasm in programming and computer science in general. Besides studies at UiO, in his free time, he develops his programming skills using online platforms like Coursera, Udacity and Udemy. Exploring new technology is one of his hobbies. In the future, he plans to work as a full stack developer.

### **Piotr**

He also studies Programming and System Architecture. Besides studies, he works as a business analyst where he got experience in Python programming and leading projects. In the future, he plans to assume a similar role within IT industry – to be a link between the business and the development. Personally, he’s a big tea enthusiast and a board games player.

### **Stella**

Second year student in bachelor programme Informatics: design, use, interaction. Her professional interests revolve around UX and UI design.

## **Yan**

Second year student in bachelor program Informatics: design use and interaction. No prior Android-related development experience. Have basic knowledge of React Native app development.

Since the group members have different interests, strengths, skills, and different programming backgrounds, we decided to split our team into 2 groups: front- and back-end development team. There were 3 people in each team which worked well through the whole project. The front-end team consisted of Mohammed, Yan, and Stella. At the beginning of the project, Kai was also helping the design team, but as the amount of work for the back-end team increased, he decided to focus on the coding to ensure a high-quality code and database. As the point of contact between the user and the deliverable project solution, the design team also had different roles: UI and UX designer and quality assurance engineer. The responsibilities of the UI designers were to create user stories, making sure the solution adheres to the requirements, think about the user flow and sketch app wireframes, create app prototypes, build intuitive app interface and make changes according to users' feedback and testing data. The QA team, represented by Yan and Stella, was responsible for the recording test progress and document test case, tracking bugs throughout testing and identifying potential challenges users might face.

The back-end team, represented by Kai, Bence, Piotr and Mohammed, were responsible for the technical part of mobile app development: turning sketches, mock-ups, and wireframes into high-quality code, creating reliable and high-performance code and correcting app drawbacks and fixing bugs.

As a development process we decided to use Scrum as we saw many benefits of it. Although the Scrum development team is a self-managing one and we decided on work principles on our own, we still needed a person responsible for the right flow of the project, and that everyone works according to a plan. The role of team lead was played by Piotr as he showed qualities which allowed facilitating communication between the team members and ensured that the team had sufficient performance levels. Drawing up a project plan that shows a set of milestones and deadlines helped us to accomplish our tasks with a non-restrictive velocity.

## User documentation

### App's functionalities

The app offers the following functionalities: checking flight status and flight delay, checking weather at the airport, and showing airplanes in the sky over Norway.

### Target group

Following types of users are the target group for our app: passengers, plane spotters, people who want to pick somebody at the airport.

### How to start

First, click on the Flynerd icon to open our Flynerd app (Figure 1). Click on 'ok' to let the app use your location. We collect the location of the users only for the purpose of the map as we are using google map integration and mostly require location access. If you do not wish to give the permission, this may limit your use in our map.

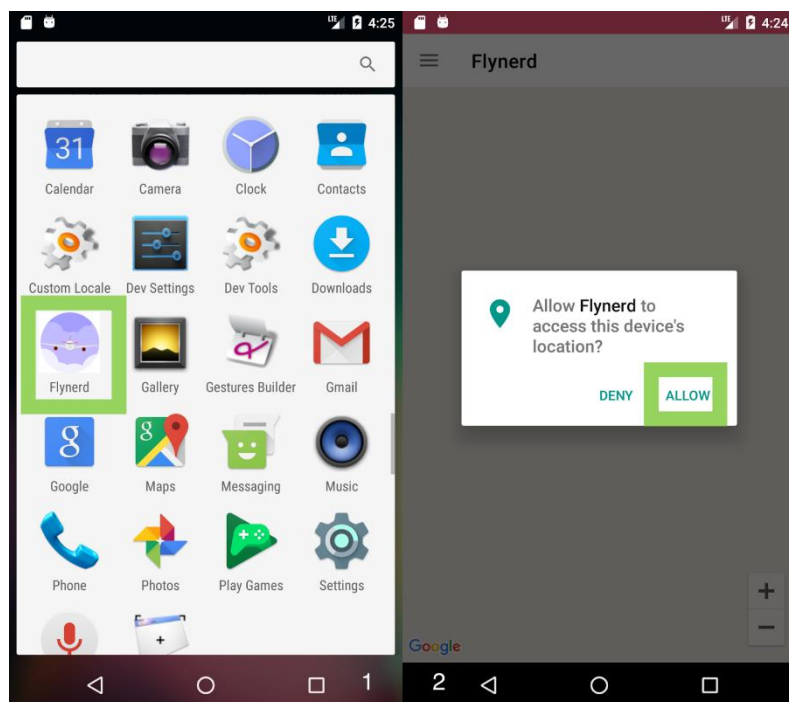


Figure 1. Screenshots showing 1) icon of the app, 2) location access request



### How to navigate in Flynerd?

In each of the pages, click on the hamburger menu in the upper left corner to see all the functions (Figure 2), click on one of them to jump into the page that you need.

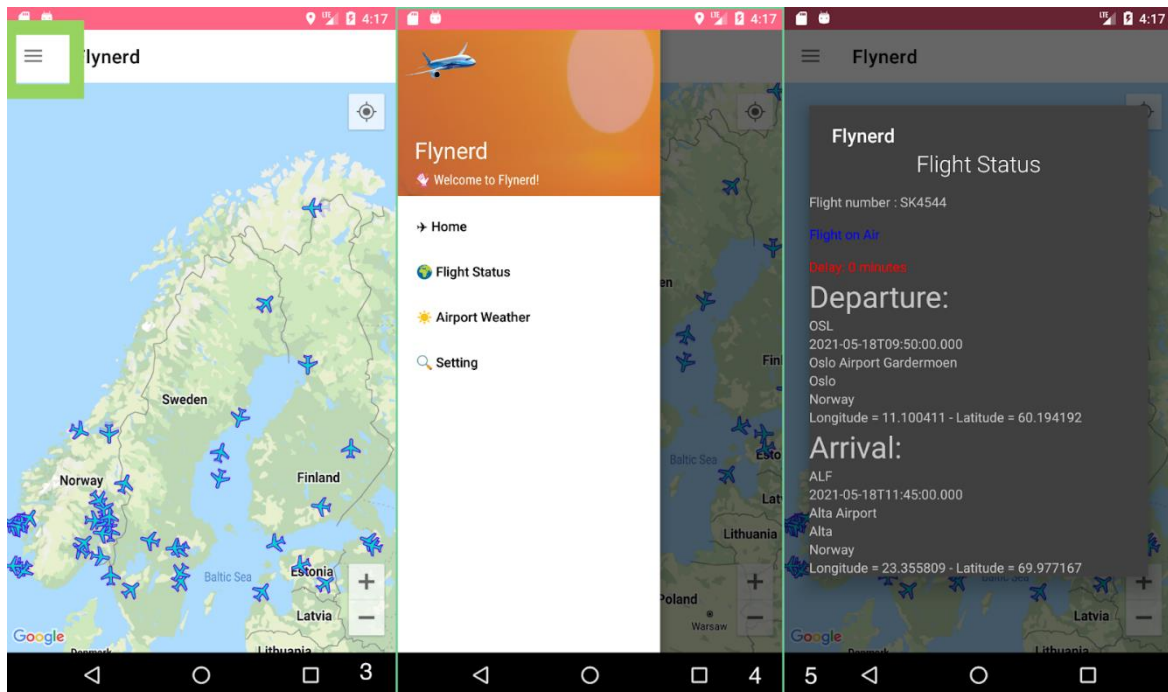


Figure 2. Screenshots showing 1) location of the menu button, 2) app's functions, 3) Flight status of a plane after tapping it.

### How to check the airplanes currently in the sky?

If you are on another page, first click the menu to navigate to "Home". Our "Home" page offers a map of all the airplanes currently in the sky (Figure 2). You can search for a specific flight by directly tapping on a plane to get more information about a flight.

### How to search for a flight?

If you are on another page, first click the menu button to navigate to “Flight status”. Then, type in the flight number in the search bar. Click on the search button. The flight information will be displayed shortly (Figure 3).

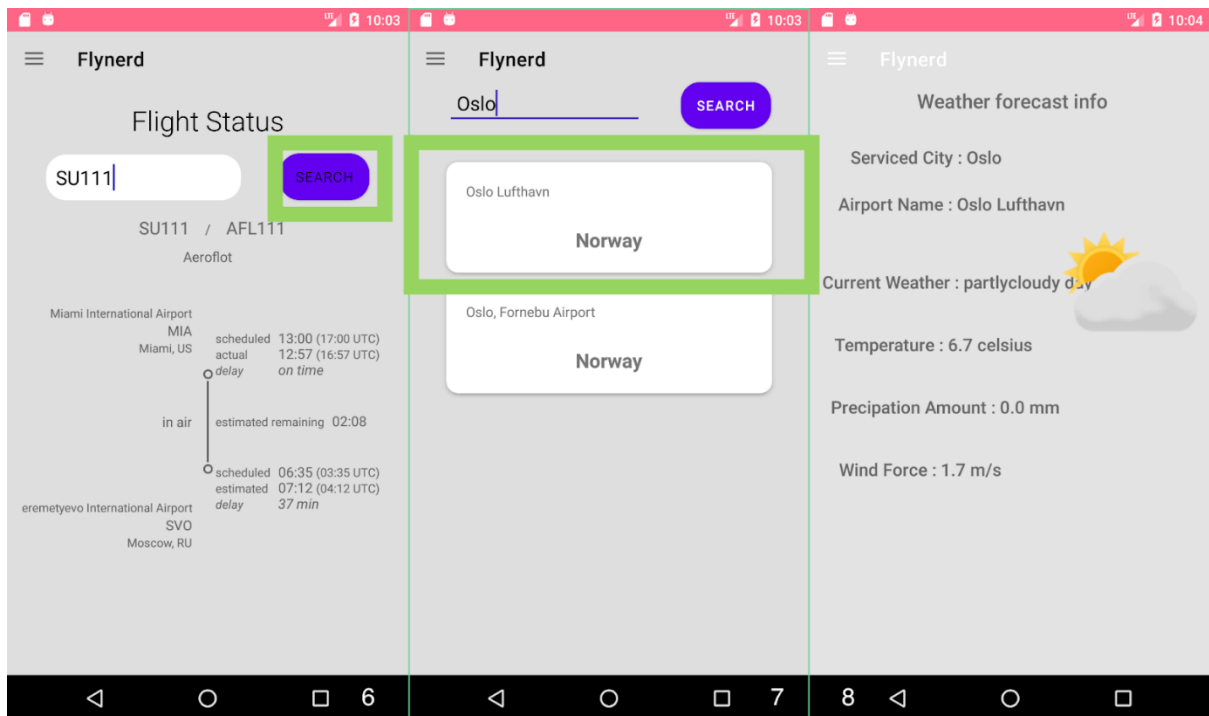


Figure 3. Screenshots showing 1) How to check flight status, 2) and 3) How to check weather at a chosen airport.

### How to check the airport weather of a given airport?

If you are not on the page of ‘Check Weather’, please first click the menu button and choose Check Weather. Then, type in the city in the search bar. Select the airport that you wish to check by clicking on it (Figure 3). The information at the airport will be displayed shortly.

### What else does the app offer?

You can go to the “Setting” page and change the text size and text weight. As of now, it only does it for the settings page.

## Requirements specification and modelling

### Brainstorming the project

During our first design team meeting we identified the stakeholders that have different interests, purposes of use of the app, issues, and concerns. To represent and make sense of complexity, we used rich pictures as a system's thinking tool. It helped us to identify the main connections, influences, interactions, and relationships that we could take into consideration during the development process. After a long round of brainstorming, we came up with numerous stakeholders, interests, and conflict of interests. We designed our initial rich picture diagram using whatever representations we liked to capture our view of the situation.

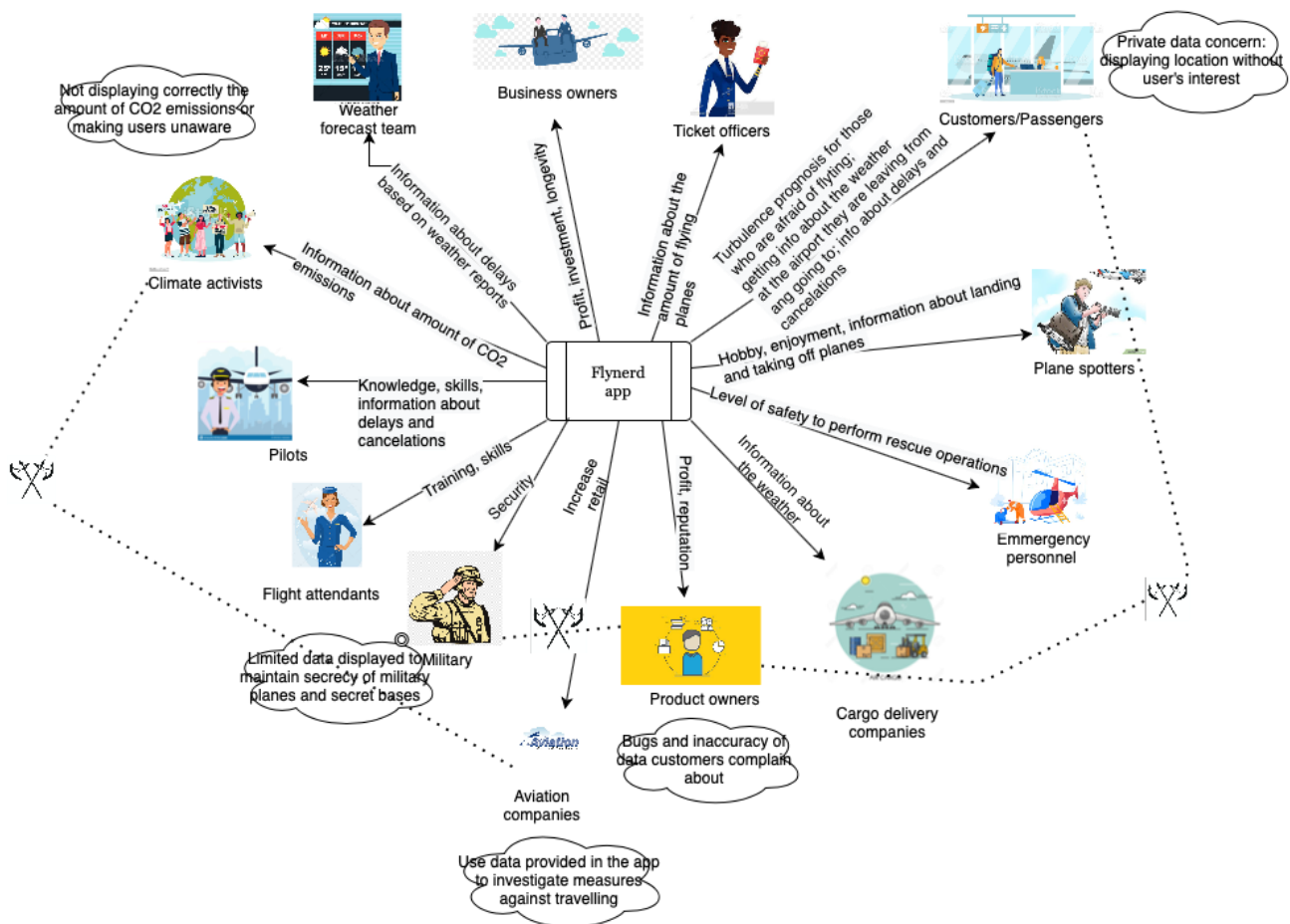


Figure 4. First version of rich picture.

However, after some reflections, we decided not to involve all the users in the development process because of their vastly different needs and requirements. Therefore, we decided to narrow down our scope to fewer stakeholders. This led to revising our initial rich picture.

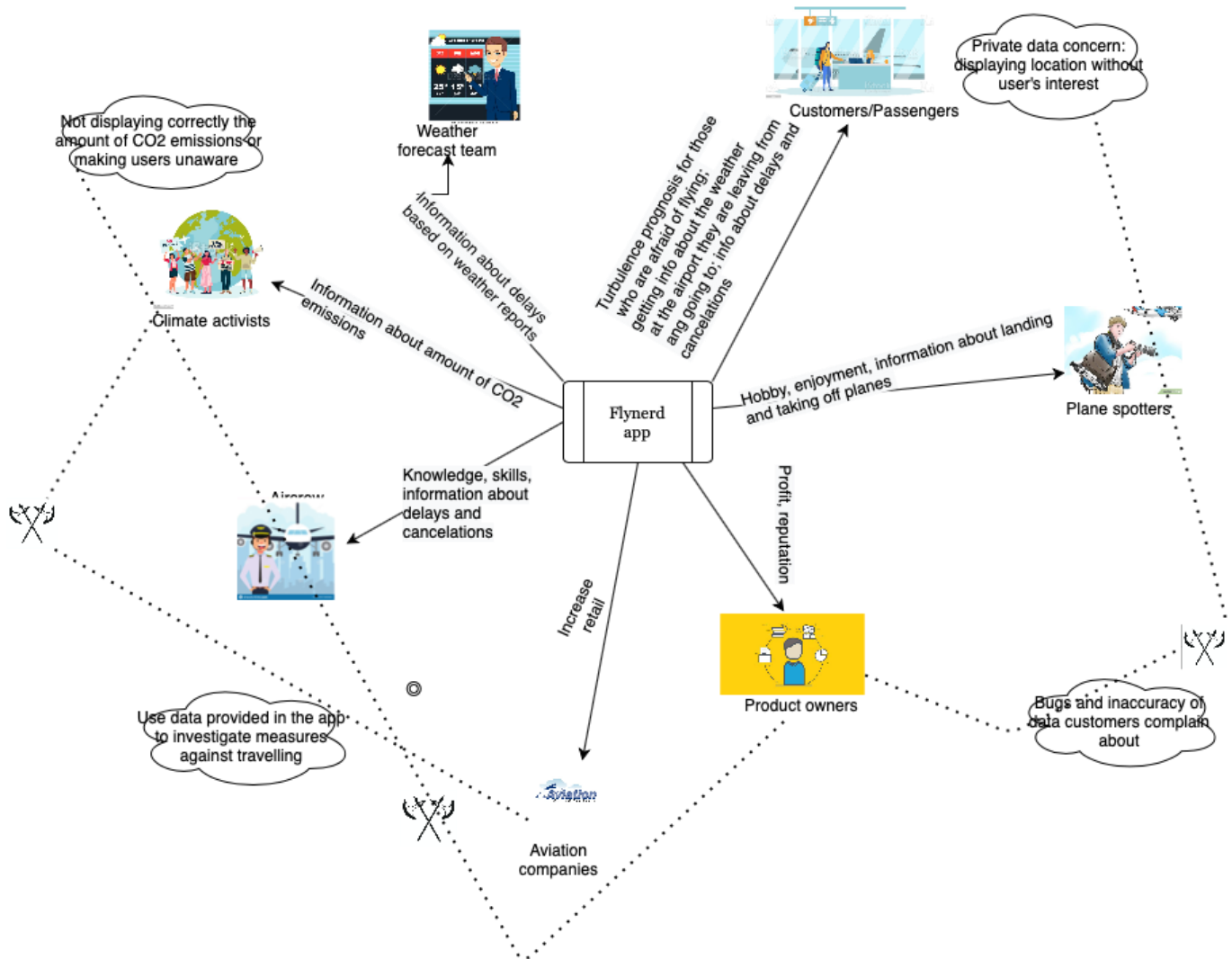


Figure 5. Revised version of rich picture.

The main stakeholders on the macro level are:

- Passenger whose interests are to get information about the flight status, weather, turbulence prognosis
- Aircrew who benefits from knowledge, skills, as well as information about flight delays and cancellations
- Plane spotters who want to know the time schedule of plane landing and taking off to pursue their hobby
- Climate activists who assess ecological and environmental awareness

On the meso level the stakeholders are represented by aviation companies and product owners with the interest of making a better profit. Their interests are to make a good product to become famous and in demand.

The first conflict identified is between the climate activists and product owners for not correctly displaying the amount of CO2 emissions or making users unaware of it at all. Another conflict is deduced between passengers and product owners in concerns of using private data and displaying location without user's consent. The driving force behind the conflict between aviation companies and climate activists is the use of data provided in the app to investigate measures against traveling. In addition, we don't disregard the conflict that can occur due to bugs in the app and inaccuracy of data that users can complain about.

### Activity diagram

As the next step, we made an activity diagram showing behaviour of the app from start to finish.

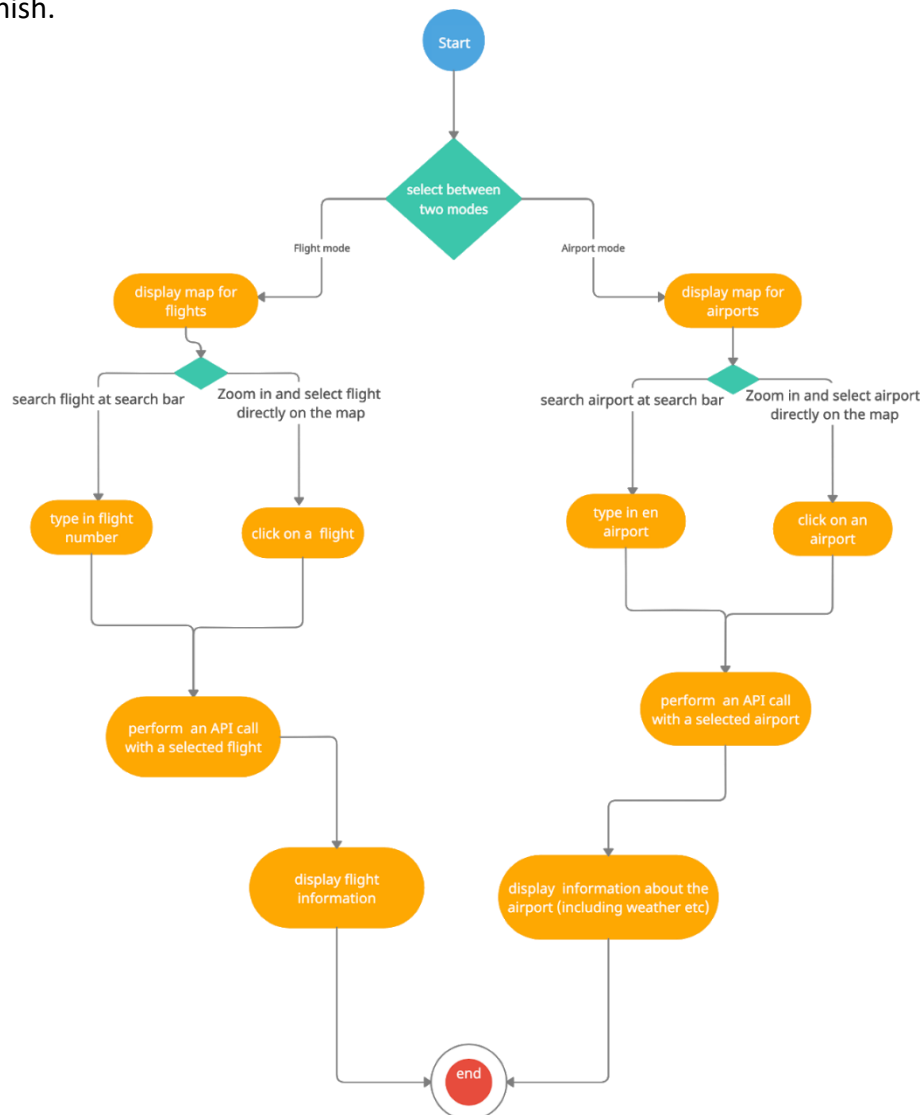


Figure 6. Activity diagram

This is the original version of the diagram which reflects how we first pictured what functions the app may have. It has mainly two functions, one is to check flight status and the other is to check weather. In each of the modes, we planned to display a map, and the user could also choose between a search bar and directly tap on the map to get a result.

However, after interviews and throughout analysis, we concluded that we want to focus on other user cases as well. So instead of a toggle switch, we chose to use a drawer navigation button for our app. Also, to keep the app simple and concise, we also removed the map from the weather part.

### User stories and backlog

After having done rich pictures and activity diagram, we were ready to decide the scope. We came up with the following user stories / functionalities. Bolded ones we considered important at the time.

- 1. As a plain enthusiast I want to see the current number of planes above Norway.**
- 2. I want to see planes over Norway on a map.**
- 3. When I click on the plane, I get detailed information about this flight.**
4. List of all planes above Norway with categories.
5. List of airports in Norway + detailed information about airports (e.g. weather forecast).
6. Depending on the end user - picking somebody from the airport.

Our next step was to perform interviews with potential users. This step gave us more insight into requirements we needed to consider as well as desired functionalities. Each of us interviewed one person. Following user stories were identified:

1. As a person who wants to pick somebody up from the airport, I want to know if the plane has landed and if it's delayed.
2. As a flight attendant, I want to know if the plane I am going to fly is delayed somewhere.
3. As a flight attendant, I want to have a list of the flights I'm going to take at a given day.
4. Extra feature: rating of flights etc.
5. As a person who is about to take a flight, I want to know what the weather at the outbound airport is and if there are any delays.

6. As a plane enthusiast, I want to be able to tap on the plane and get information about it.
7. I want to be able to know how many planes there are above Norway right now.

You can find interview questions and consent form in attachments.

After conducting the interviews and reducing the amount of user stories, we decided to mainly focus on these three:

1. As a person who wants to pick somebody up from the airport, I want to know if the plane has landed, so I can plan when I have to be at the pickup point.
2. As a person who is about to take a flight, I want to know what the weather at the outbound airport is, so I know what clothes to wear.
3. As an average user / flight attendant, I want to know if the plane I am going to fly is delayed, so I can plan what to do with my time.

We reached this conclusion after a voting we did using Miro.

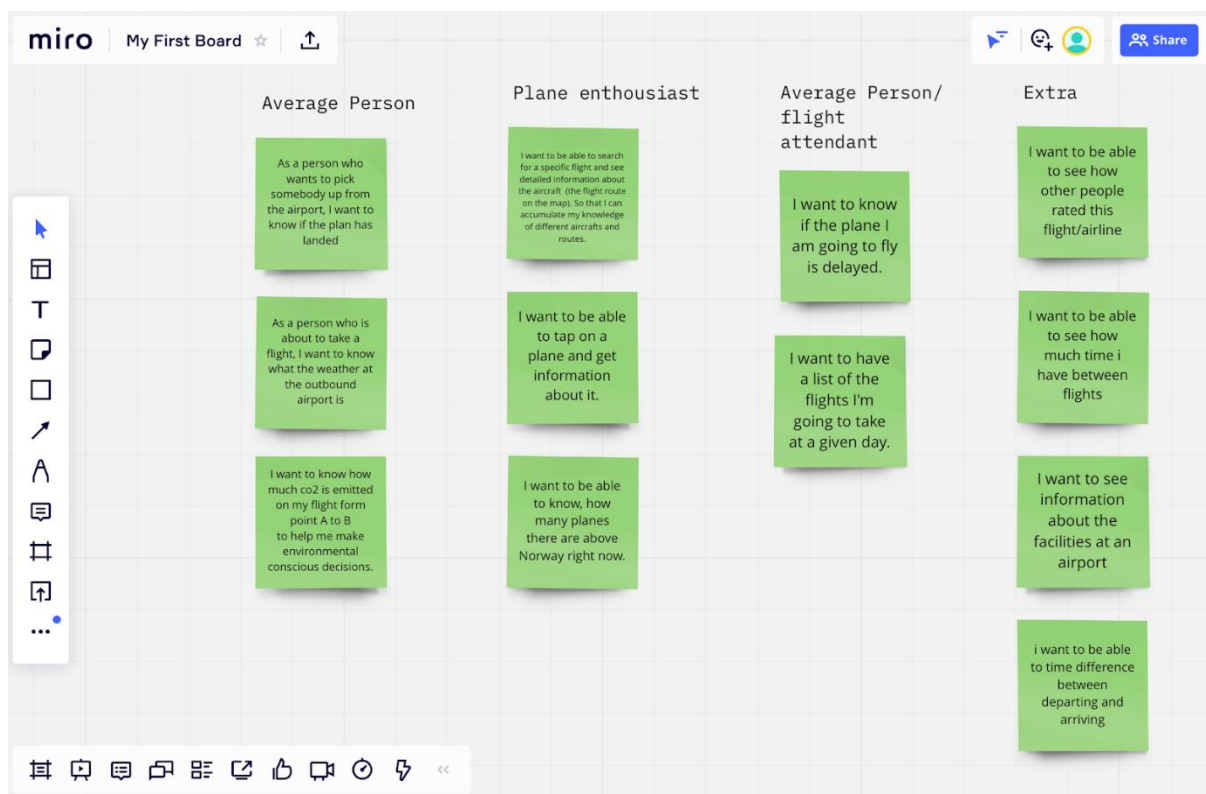


Figure 7. Miro board we used for voting.

By breaking down the user stories into smaller stories we experienced that the development process went easier and faster. Eventually our backlog took the following form:

Priority	User story
1	As a person who wants to pick someone up from the airport, I want to have a time estimation of when the plane is going to land, so I can plan my day efficiently.
1	As an average consumer/flight attendant who is going to travel/work, I want to know if my plane is delayed, so I can plan how to spend my time in advance.
1	As an average consumer, I want to know what the weather at the outbound airport is like, so I can plan what to wear in advance.
2	As a plane enthusiast I want to be able to search for a specific flight and see detailed information about the aircraft (the flight route on the map). So that I can accumulate my knowledge of different aircrafts and routes.
2	As a plane enthusiast, I want to be given the ability to tap on planes shown on the map, so I can get information about that specific flight.
3	As a plane enthusiast I want to know/see how many planes are airborne in Norwegian airspace, so I can satisfy my curiosity.
4	As an environmentally conscious consumer, I want to know how much CO <sub>2</sub> is emitted on my flight from point A to B to help me make environmental conscious decisions
4	As a flight attendant, I want to have a list of the flights I'm going to take on a given day, So I can plan my day more efficiently.
4	As an average consumer, I want to be able to see how other people rated this flight/airline.
4	As a consumer, I want to see information about the facilities at an airport
4	As a consumer, I want to be shown the time difference calculation of arriving and departing flights when I have a transfer, so I know how much time I have at the airport.

### Majors use cases

We used use cases throughout our project to define the flow of the app, explain how the overall system should behave and brainstorm what could go wrong. Use cases also helped us to negotiate which functions should be built and make sure that every member of the group has the same product vision.

Primarily, we designed a general use case diagram that showed the flow of the app regarding all the features bundled together that were discussed in the planning phase – display planes in air, show the weather forecast for the outbound airport, expose the list of airports and of planes identities (see Figure 8).





Figure 8. Initial use case diagram showing all major use cases

However, we found it too complicated and cramped and decided to spread it into two. Use cases for the delay and flight status were kept in a one diagram, as both can be reached by clicking on "flight status" icon on the menu. The "check weather at the airport" use case is pictured on a separate diagram. These three use cases are described below.

## Use case #1

Name: Check flight status

Actor: Traveller

Pre-condition: The user has downloaded the app and has access to the Internet

Post-condition: The user has checked the status of a flight

Main flow:

1. The user goes to the home page and clicks on the “check flight status”
2. The system displays the search screen and enables the user to enter the flight number
3. The user types the flight number and clicks on “search”
4. The system shows a loading animation
5. The system displays the status screen with a progress bar flight status

Alternative flow:

- 3.1 The user enters an invalid flight number
- 3.2 The system doesn't find the searched flight and displays an error message
- 3.3 The app goes back to step 2

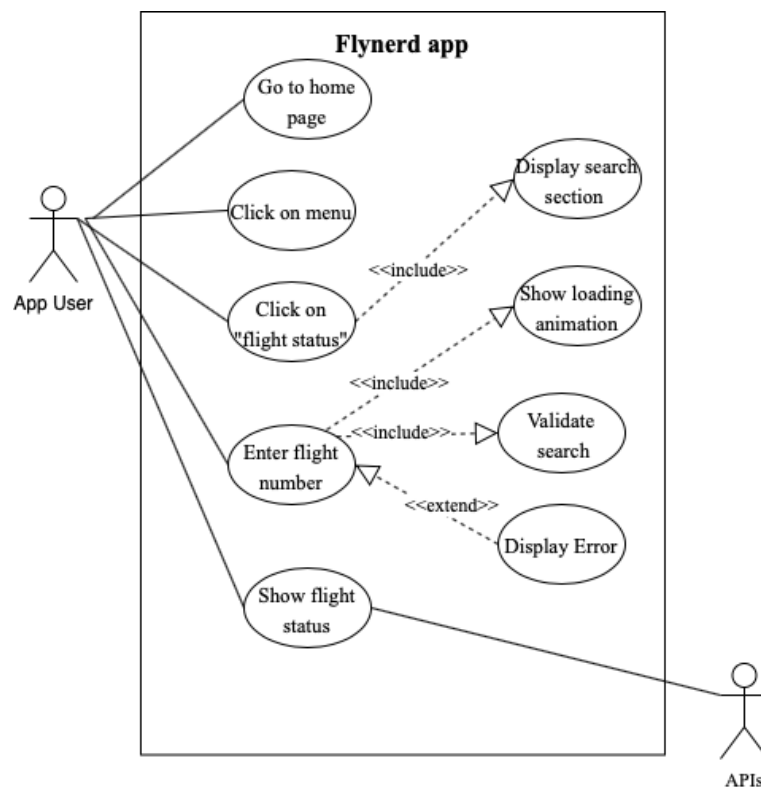


Figure 9. Use case diagram for use cases: check flight status, and check if a plan is delayed.

## **Use case #2**

Name: Check if a plane is delayed

Actor: Flight attendant

Pre-condition: The user has downloaded the app and has access to the Internet

Post-condition: The user has checked the status of a flight

Main flow:

1. The user goes to the home page and clicks on the “check flight status”.
2. The system displays the search screen and enables the user to enter the flight number.
3. The user types the flight number and clicks on “search”.
4. The system shows a loading animation.
5. The system displays the status screen with a progress bar flight status.

Alternative flow:

- 3.1 The user enters an invalid flight number.
- 3.2 The system doesn't find the searched flight and displays an error message.
- 3.3 The app goes back to step 2.

### Use case #3

Name: Check airport weather

Actor: Passenger

Pre-condition: The user has downloaded the app and has access to the Internet

Post-condition: The user has checked the weather of an airport

Main flow:

1. The user goes to the home page and clicks on the “check flight status” in navigation menu
2. The system displays the search screen and enables the user to enter the airport location
3. The user types the airport and clicks on “search” button
4. The system shows a loading animation
5. The system displays the weather result in a card

Alternative flow:

- 5.1 The user enters the airport name incorrectly
- 5.2 The system doesn't find the searched location and displays an error message
- 5.3 The app goes back to step 2

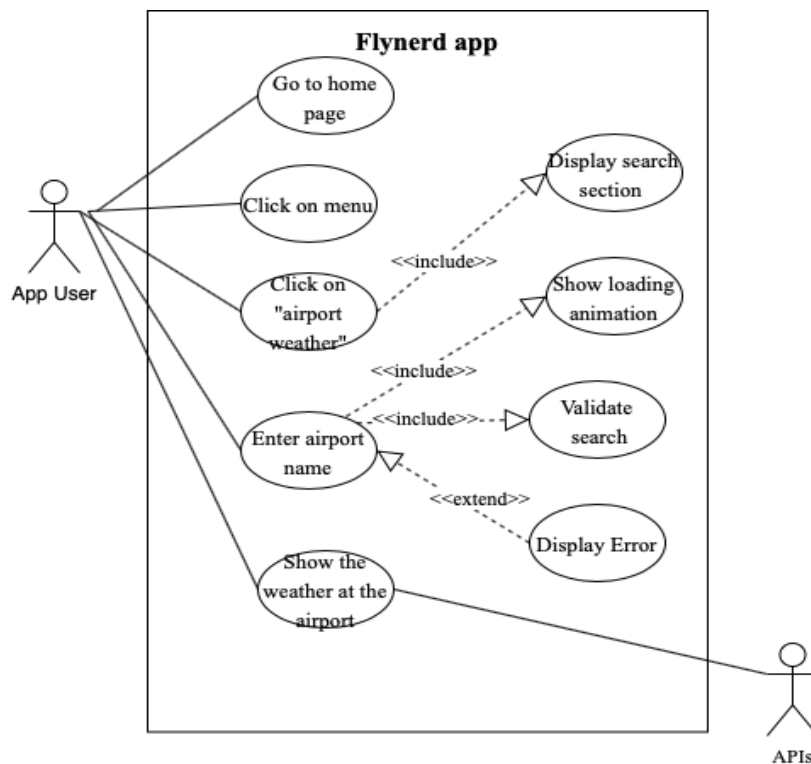


Figure 10. Use case diagram for “check airport weather” use case.

## Sequence diagram

The diagram shown below is a simplified version of the user's action sequences to check the weather forecast at a given airport.

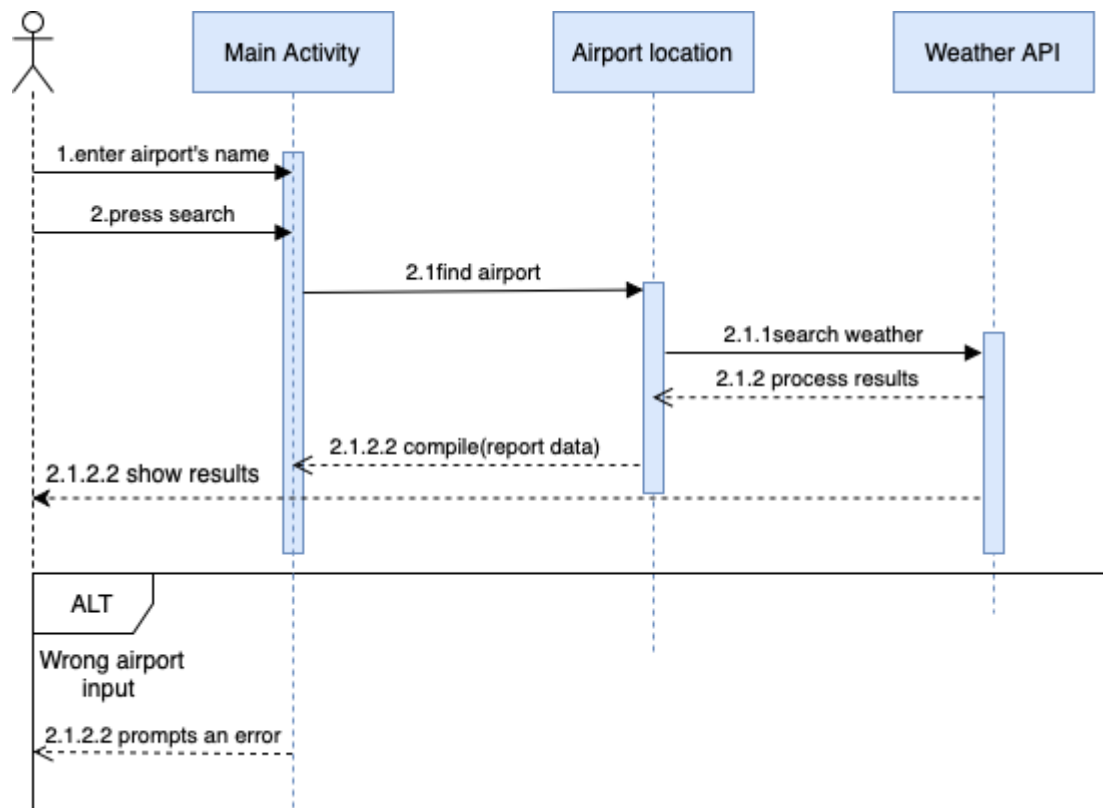


Figure 11. Sequence diagram

## Functional and non-functional requirements

Having in mind all design considerations and the backlog, we specified our functional and non-functional requirements.

### Functional requirements

Type	Nr	Specification
Must	1.0	Show overview of planes over Norway
	1.1	Show Norway's location on map (default view)
	1.2	Present information about a flight's duration and distance, flight number upon clicking.
	1.3	Detect user's location from GPS
Should	2.0	Display information of airports in a country
	2.1	Show weather at each airport according to the time entered by user
	2.2	Predict the turbulence in a specific area
	2.3	Show information about CO2 emissions
Can	3.0	Show data about delays and cancellations of flights (pick up feature)

## Non-functional requirements

Type	Nr	Specification
Performance	1.0	The application shouldn't take more than 3 seconds to load initial screen. It should not hindrance to the user input.
Scalability	1.1	The app should be capable of handling increased user's data without delay.
Responsiveness	1.2	When app gets interrupted by call, it should be able to save state and return to same state which was before it got interrupted.
Usability	1.3	The flow of the app should be easily understood without the help of any guideline or experts/manuals.
Reliability	1.4	When user performs actions, they should be acknowledged with confirmation.
Availability	1.5	Users can access the application everywhere, install it, look for regular updates and give feedback.
Screen adaption	1.6	The application should be able to render it's layout to different screen sizes.
Network coverage	1.7	The app should be able to handle slow connection. It should be able to look for WiFi if not available then automatically switch to mobile network.
Accessibility	1.8	The app should be developed according to universal design, for people of different physical possibilities.
Compatibility	1.9	The app must run on a different configuration, database, mobile devices and their versions.
Fetching data	1.10	Present data in the real time.
	1.11	System must fetch data from API upon the user's request of information.
	1.12	System must use the flights-API from OpenSky.
	1.13	System must use the weather-API from Meteorological Institute.
	1.14	System must use the turbulence-API from IPPC turbulence prognosis.
Organisational requirements	1.15	The app should be developed in Android Studio where Kotlin is used as programming language.
	1.16	App development must be performed with the agile process modelling Scrum.
External requirements	1.17	The system should follow the Personal Data Act and Data Protection Regulations (GDPR).

## Inclusive design

We want to build an app that can be used by most users. We have taken the inclusive design patterns and user-centered design patterns into consideration from the very early stage. We had meetings around it and discussed what should be implemented and what is practical to implement within our timeframe. We wanted to design the app in a way that it is intuitive to use, easy to understand and support accessibility needs.

## Colour and contrast

Colour can convey messages and emotions and is one of the most important virtual elements. When we choose the primary, secondary colours used in our app, we also consider colour availability. We follow WCAG 2.1 guidelines and tried to use colours that have high contrast ratios to help users with colour vision deficiency. For example, if a button has a dark background colour, we will use white colour text. We use tools like Colour Contrast Checker to check if the colour fulfils the requirements (Figure 12).

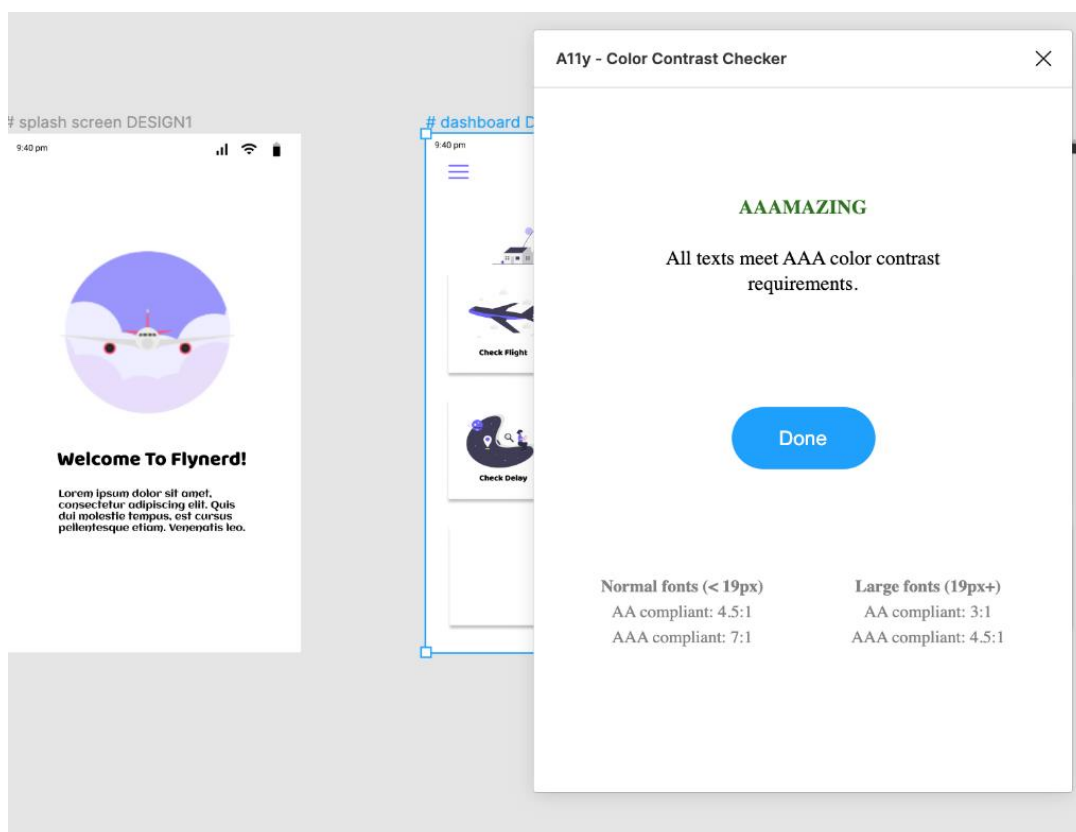


Figure 12. Screenshot of colour checker for our app.

## Navigation and hierarchy

One of the most important choices we made is choosing a navigation component for our app. We wanted to build a navigation component that empowers the users to interact with our app and supports its structure and purpose. We favoured bottom tab navigation in the beginning when we considered our app with mainly two functions (check flight status and check weather). However, as we progressed with our design process and collecting more data, it seemed that a navigation drawer menu was the most suitable choice (Figure 13). To enhance the user's understanding, we also used some pictures, icons or emojis in the navigation drawer menu (see in the app).

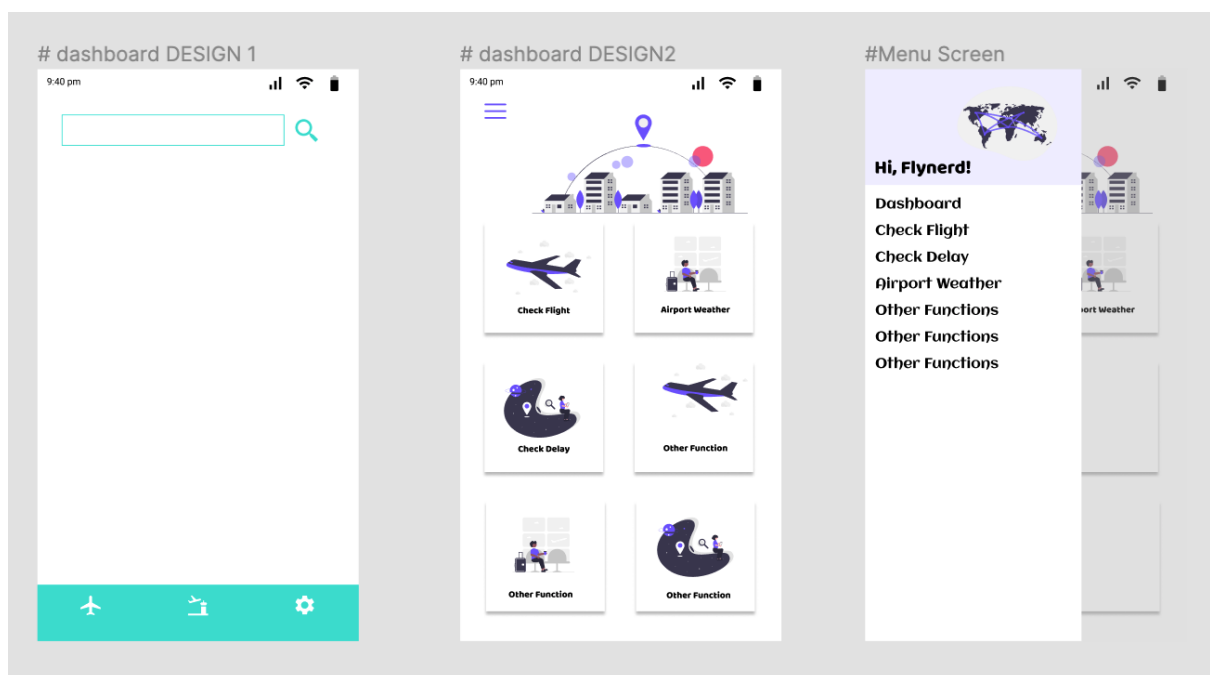


Figure 13. Designs of navigation menus

We also tried to emphasize visual hierarchy in our app to help describe the importance and sequence of different elements in each screen. We place important actions (hamburger menu) at the top left corner of the screen, showing important information in larger text font with primary colour. We also use grouping to show elements in a clean and readable order. Transitions between screens were also considered, and we tried to implement some animation in between.



## Typography

Typography can help to create a certain atmosphere and express identity. We choose fonts that are well-known and comfortable to read in our app. Texts are all measured in scalable pixels (sp) and left with sufficient line height and letter spacing (Understanding typography, 2021). In the research process, we have also learned about Google TalkBack and using alternative texts. After some consideration, we aimed to implement a function that the user can use to adjust the text size and text weight system wide.

In the implementation, we had some difficulties adjusting the text fonts in menu items. We tried changing the themes and different ways but couldn't find an optimal way to manipulate all the Textviews in the app. Manipulating all Textview manually is time-consuming and may potentially mess up the layout. In the end, we had to adjust to only implement this function for our setting page (Figure 15). When the user turns on the switch of Larger Text, the text font increases. If both switches are turned on, the text will be bold and in larger size. We think it is important that the user can adjust the text fonts at least to some extent.

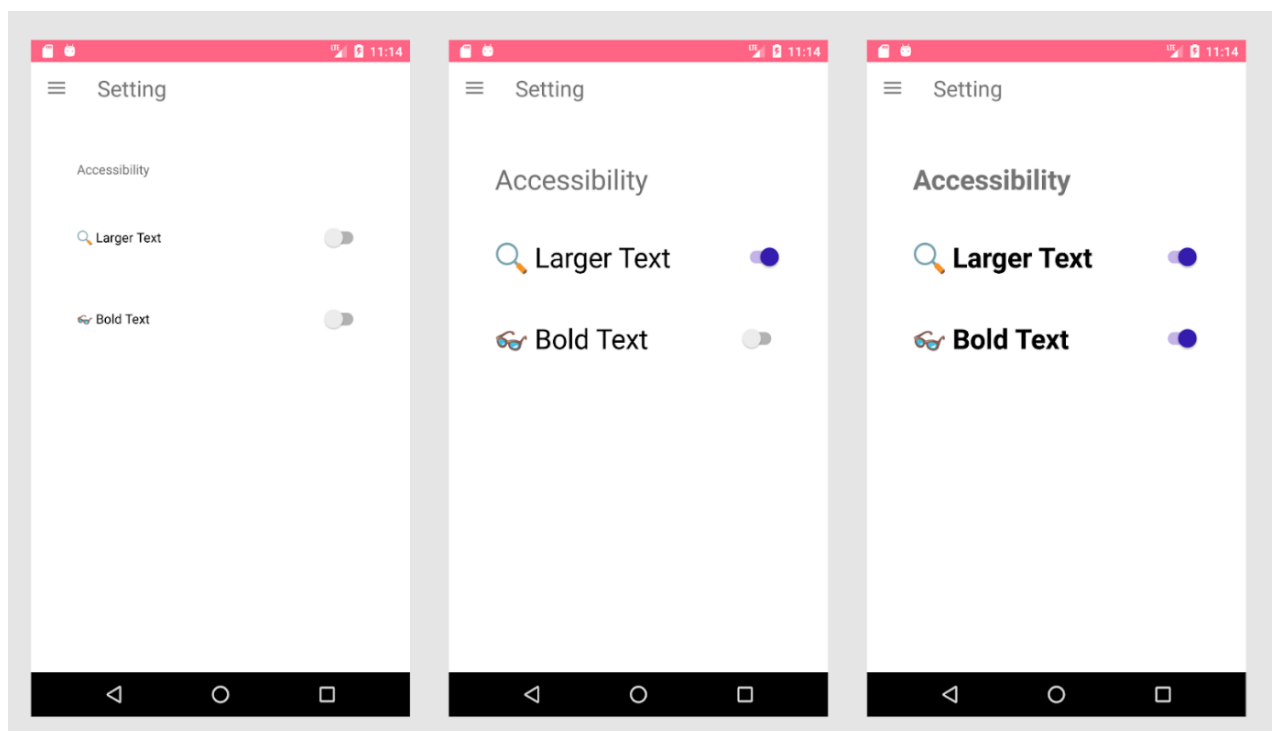


Figure 15. Screenshot of text font setting in app

## Class diagram

We made our class diagram after having coded a part of our backlog and we were updating it along the way. The below version shows major classes of our app and how they are related. Because of a big volume of our class diagram, we decided to put it into the attachments, [click here to be redirected to it.](#)

## Design patterns

A design pattern is a generalized, reusable template for solving a certain kind of problem within engineering contexts. It is abstracted as to provide only the scaffolding for building the solution, leaving decisions about how to fill in the specifics to the developer. One reason we use design patterns is that they are mature and extensively tested, so we can learn from the experience and avoid the mistakes of others, rather than continuously “reinventing the wheel”. They also serve as common mental models that developers of the project can lean on to understand the code. In this way, they play a vital role in ensuring the primary purpose of software architecture: to reduce complexity in development and communication (Sommerville, 2021). Here follows a quick overview of the most important design patterns used in our app.

## Object-oriented principles and GRASP

Object-oriented principles define a family of paradigms; we can only provide a summary of this very extensive topic. The primary unit of abstraction in object-oriented systems is an object: a self-contained instance of some class. The object's class defines its possible states and associated behaviour. The state is implemented via data fields, whereas the behaviour via methods. Access to an instance is mediated through its public-facing interface, which is defined as a subset of its fields and methods (Sommerville, 2021).

The principle of encapsulation refers to strict separation of interface and implementation. The main advantages it offers are improved flexibility in modifying implementation details, and improved workload distribution, as different classes can be developed and maintained with a high degree of independence (ibidem).

The General Responsibility Assignment Software Principles are a set of influential design patterns within object-oriented development. We touch on the most important GRASP patterns in this section. A fundamental object-oriented concept is generalization. This relies on a hierarchical approach to information modelling, where a more specific class (subclass) inherits from a more general one (superclass). Conceptually, any object that "is an" A also

implicitly "is a" B, if class A inherits from class B. In more concrete terms, if A inherits from B, then any object that is of type A is also of type B. We should in principle be able to substitute any object with an instance of some subclass of it. By default, a subclass inherits the implementation of its superclass; it may also override some parts in order to specialize behaviour. Overriding to achieve differences in behaviour of types is known as polymorphism (*ibidem*).

There are also some best practices regarding distribution of responsibilities among software units. By the expert principle, we should assign a responsibility to the object which contains the information to fulfil it. The extent of how well classes are grouped relative to their responsibilities is referred to as cohesion. High cohesion means classes that are grouped together have strongly related responsibilities. The extent of an object's dependence on, coordination with, and control over other objects is referred to as coupling. Low coupling means a class is constrained to a small, well-defined area of responsibility, with as few dependencies on other objects as possible. High cohesion and low coupling are desirable traits from an architecture standpoint, as they reduce complexity and redundancy and increase reusability (*ibidem*).

Object-oriented principles are integral to the JVM computing model and the languages targeting it, including the primary language of our app, Kotlin. Kotlin also has the concept of abstract classes and stand-alone interfaces. Abstract classes may be thought of as classes it does not make sense to have instances of. They may be partially implemented, and as such, they are not directly instantiable, only through subclasses that implement all the missing pieces. They are useful both to avoid duplication, as well as to enforce more abstract modelling requirements. Stand-alone interfaces are much like abstract classes, but with no state implementation. This lets interfaces make use of multiple inheritance, that is, a class or interface may inherit from multiple distinct interfaces. The stand-alone interfaces an object implements are therefore better thought of as defining separate aspects of that object's interface. When only a specific aspect is relevant to a method, it can take an argument of the interface type rather than a concrete type to be more flexible, for example.

Because of its close integration with the android OS and broader ecosystem, our app inherently relies on many object-oriented techniques. Throughout development, we tried to keep in mind the above described secondary principles to maintain high quality code and achieve the paradigm's full potential.

## Model-View-ViewModel

A common pattern for the overall structure of Android apps is the so-called Model-View-ViewModel pattern. The pattern is highly recommended by Android's maintainers, and the Android library is well-equipped to help implement it efficiently (ViewModel Overview: Android Developers, 2021). Poorly designed apps commonly assign far too much responsibility to activities; MVVM counteracts this to ensure a separation of concerns. We separate classes into three distinct kinds with distinct responsibilities:

1. A View is a component displaying UI and processing and forwarding user and OS events. It belongs in the front-end. Typically, this is implemented as an activity and/or fragments.
2. A Model handles the necessary data; it is responsible for reading from and updating various data locations. It belongs in the backend. This may be implemented as a simple API class, or preferably via the repository pattern.
3. A ViewModel, found in the Android standard library, is the connection between the front- and back-end. It contains the logic for reacting to user events and selecting and formatting data from the back-end. Typically owned by an activity or several fragments, and itself owning a model instance.

The components are structured as follows: some View components contain references to a ViewModel, which contains a reference to a Model component. Crucially, a ViewModel should have no knowledge whatsoever of its Views, and a Model should have no knowledge of the ViewModel using it. In general, this gives a dependency tree, and we get the associated benefits of minimal coupling. It is also well-suited to unit testing components. View elements are difficult to unit test programmatically because their existence is managed by and tied to the operating system. By delegating as much of the behaviour as possible to ViewModels, we can run our tests on ViewModel objects instead, whose lifecycles we control. We can write tests for a ViewModel that never need to use an actual activity or fragment, and similarly, we can test Models independent of the other components.

An at least equally important benefit of this pattern is the result of how Android manages UI component lifecycles. In short, every View component is, at any given point in time, in some state of its lifecycle: created, started, paused, resumed, destroyed, and so on. The operating system manages what components are in what state based on user interaction,

resource usage, and other factors. State transition events trigger the respective methods (onStart, onPause, etc.) of the components; overriding these lets us take appropriate action. A component may go through several destructions and re-creations, for instance on screen rotation. Whenever this happens, anything stored in the activity's fields is lost (Understand the Activity Lifecycle: Android Developers, 2021). A ViewModel does not have a lifecycle, nor is it dependent upon the lifecycle of the View it is attached to. It is handled specially so that it remains in memory from the time an activity is initialized until it is discarded. The data it stores therefore remains undisturbed by lifecycle events if the activity is not discarded, so the UI can restore its state from it following re-creation.

We make use of MVVM throughout the app. The activities such as ForecastActivity and MapActivity all have their respective ViewModel subclasses (e.g. ForecastViewModel and MapViewModel). Each exposes the data meant for the activity as LiveData values, and each contains references to whatever backends it gets the data from.

## Repository

The repository pattern ensures that there is a clean, uniform abstraction level encapsulating all backend data sources. A repository class translates app-specific types into raw data, makes necessary retrievals, and converts raw response data into app-specific data. It acts as an interface through which backend data is accessed by the rest of the app. As such, the other modules need not concern themselves with how the data is stored and can focus on their own responsibilities. More concretely, repositories handle typical bookkeeping tasks like API authentication and local caching (Guide To App Architecture: Android Developers, 2021). They may also coordinate and unify multiple heterogeneous data sources. For instance, they might keep a permanent local SQL data store that is updated on demand by fetching from the REST API or fetch different aspects of the data from different APIs. In such cases, it is also the repository's responsibility to serve as a single source of truth for the data in question, by checking for and handling any consistency errors.

For a subset of our data sources we opted to use the repository pattern. Retrofit library services access the APIs and are in turn controlled by repository classes. For example, the FlightStatusService endpoint byFlightNumberArrivingOn takes flight id as a String-Int pair, and date as an Int triple. The repository class FlightStatusRepository customizes its service instance to insert appropriate authentication query parameters for each request. Its endpoint

method `byFlightIdArrivingOn` maps `FlightId` and `LocalDate` objects to the primitives mentioned. We also throw exceptions on various response errors and convert the response to app-specific data structures here. The app also has an example of local caching in the `OpenSkyRepository` class. With respect to permanent storage, we judged that it was out of scope with the amount of development time available.

## Product documentation

### Coroutines and lifecycles

To avoid unresponsive UI and freezes, Android development requires heavy computations and I/O tasks to be performed without blocking the main thread. For such non-blocking operations, we chose coroutines. They enjoy first-class support in Kotlin and are safer and easier to use than conventional Java concurrency. Further, coroutines are the recommended idiomatic way of asynchronous development under Android, so the choice was quite natural and did not require a lot of consideration.

A coroutine is essentially a suspendable computation sequence. Execution of the sequence may be suspended on a step, and subsequently resumed from that same step, with the same state (Kotlin Coroutines, 2021). Think of a suspended computation as a copy of the memory and CPU state at the point of suspension. On resumption, this copy replaces the current contents of memory and CPU, and the coroutine can continue executing; from its perspective, as if no interruption has occurred. A coroutine implements a non-blocking operation by suspending for the duration of the operation and allowing other coroutines to run in the meanwhile. Coroutines are managed by dispatchers that decide when to launch or resume coroutines, and what threads to use. As such, coroutines can also act as lightweight concurrent threads.

Kotlin only has barebones built-in coroutine support in the language, in the form of suspend functions. `Suspend` functions represent suspendable computations; they are only callable from other suspend functions. Everything else is implemented by the `kotlinx-coroutines-*` family of libraries (Kotlinx.coroutines reference documentation, 2021). Creating and launching coroutines is accomplished via functions called coroutine builders, that take suspend functions. There are many different builders for different use cases. For our app, it suffices to be familiar with `launch`, which launches a new coroutine without blocking the current thread.

The context of a coroutine is an associated data structure containing arbitrary objects, used for various purposes. Among other things, a custom dispatcher can be defined as part of a coroutine's context. We usually switch contexts using `withContext`, which launches a new coroutine with the given context, and suspends the current one until the new one completes. Each coroutine executes under a scope as part of its context; this is called structured concurrency (Coroutines Basics: Kotlin, 2021). Any coroutines created within the

context of another one is considered children of the latter. A scope controls the lifetime of its coroutines; once the scope terminates, it cancels all its coroutines. Moreover, children are automatically cancelled once the parent is cancelled. If we take care to scope our coroutines to the proper lifetimes, this hierarchy ensures that there are no resource leaks from runaway coroutines.

## Lifecycles

The Android architecture provides first-class support for so-called lifecycle-aware scoping, in the `lifecycle-*-ktx` family of libraries. These libraries define scopes for all lifecycles' components that cancel their coroutines once the component reaches a destroyed state (Use Kotlin Coroutines with Architecture Components, 2021). In our app, we make use of `lifecycleScope` and `viewModelScope` to restrict our asynchronous operations appropriately.

To exemplify these different aspects of coroutines, let us look at the implementation of continuously updating the aircrafts' current locations. There is an infinite update loop inside a coroutine. In each iteration, it:

1. Fetches new states from the repository in a suspending manner,
2. Updates the LiveData with the states, and
3. Calls the library primitive `delay` to suspend until the next update cycle.

These coroutine uses the IO dispatcher, which executes coroutines on a dedicated I/O thread to free up the main thread. The coroutine is launched under the `viewModelScope` of the ViewModel, so it is cancelled when the ViewModel is destroyed. Without further restrictions, the loop would keep running and making network calls regardless of the state of the activity (e.g. even when the app is sent to the background). To avoid this, we make a custom `MutableLiveData` subclass for the aircraft states, where we override the `onActive` and `onInactive` methods. These are called when the number of active observers changes from 0 to 1 or 1 to 0, respectively. Method `onActive` launches the coroutine and saves its job, a handle that can be used to cancel the coroutine; method `onInactive` uses the job to do just that. Since an observer is only counted as active if the lifecycle is in state started or resumed, updates are automatically paused when the activity leaves these states and resumed when it re-enters one.



## Libraries

### Fuel

The Fuel library is a simple http networking library kotlin/android. It provides support for basic http verbs and Kotlin's coroutine module. This makes it possible to perform both asynchronous and blocking requests. The library is modular which made it easy to select and include the dependencies we needed (Fuel documentation, 2021).

The Fuel library was used to make simple API call in the `ForecastViewModel` class. The call consisted of a simple get call to MET's location forecast 2.0 API. This was done in conjunction with a call to the library's header method. This was necessary to add UA identification to the request to satisfy conditions set by MET's terms of service.

Students of the course were introduced to the library early in the semester. Its simplicity resulted in one of our team members favouring its usage. In retrospect, it would have been easier to stick to just one HTTP request library to limit the amount of dependencies used across the whole project. Most of the app uses the retrofit library to make API calls. No clear reason can be given for this choice. Due to corona, the backend team was forced to work on the code in isolation. The `ForecastViewModel` was written mostly by one author. While we did frequent code reviews, no comment was made on the library's usage. It simply had not crossed our mind to specify one single http networking library for our whole project.

### Retrofit

Our app utilizes the retrofit library to build most of the API backend. Retrofit was chosen because of its declarative convenience, its native support for Kotlin coroutines, and existing integration with gson. It works by defining a new interface, conventionally called a service interface, whose methods correspond to endpoints of a REST API. Each method is annotated specifying how to access the endpoint, e.g. `@GET("states/all")` means to send an HTTP GET request to the relative URL `states/all` appended to the base URL (Retrofit, 201). Parameters to each method may also be annotated specifying how to incorporate that parameter's value into the request, e.g. `@Query("icao24")` means that the argument is to be appended as a query parameter like `?icao24=c0ffee`. Retrofit natively supports coroutines, meaning that service methods may be marked `suspend` to take advantage of non-blocking execution.

The response from the endpoint is automatically converted to the `return` type. When we create an instance of a service interface, we must provide some converters. Converters are special objects retrofit uses to try and convert the response to the specified type. We use a converter that integrates gson into retrofit called `retrofit-gson`.

### Gson

Gson is a JSON serialization and deserialization framework for the Java environment, developed by Google. Gson converts between JSON and java objects using reflection, requiring no special effort on the part of the developer besides writing the data classes (Gson user guide, 2021). We chose this library because of its ease of use, as well as its maturity and active support. It also has pre-existing integration with retrofit, so all in all an excellent choice. Let us also remark that some APIs use ISO-8601 formatting for their dates. Since gson does not include type adapters for java standard library types, we need to customize the gson converter. Using the `registerLocalDateTime` and `registerZonedDateTime` methods, we register custom datetime type adapters that can parse the given format.

### Kotlin-csv

To read the csv file, we made use of a third-party library called `kotlin-csv`. The library is created by a GitHub user under the moniker `doyaaaaken` (Kotlin-csv user documentation, 2021). No other library was considered as it offered everything we required: reading of csv files with possibility to do this asynchronously.

The library was used in the coroutine method `createDb` of the `AirportsViewModel` class. It was used to asynchronously read the csv file `intair_city.csv` row by row. The information provided by each row was subsequently used to create `Airport` objects for the locally stored database.

### Java 1.8 desugaring

For Android API versions below 26, significant parts of the Java 8+ standard library are unavailable. In our app, the `java.time` package turned out to be very convenient, especially for handling the dates and times retrieved from the APIs. This is Java's revised standard API for dates, times, instants, and durations, whose usage is preferred over the legacy `util` classes (Java API Reference, 2021). Firstly, the Android gradle plugin lets us enable a subset of Java 8 language features for any API level. This is accomplished by setting compile options

`sourceCompatibility` and `targetCompatibility`, as well as the kotlin option `jvmTarget` to 1.8. Secondly, to actually use the parts of the stdlib we require, we also enable compile option `coreLibraryDesugaringEnabled`. With this, we can include special dependency `desugar_jdk_libs` and use the standard library types normally in the code. The library types are injected by performing “desugaring”, i.e. rewriting the bytecode to use types from this library at runtime instead (Use Java 8 language features and APIs: Android Developers, 2021).

## Lottie

Lottie is a library that can be used for Android, iOS, Web, and Windows which “parses Adobe After Effects animations exported as json with Bodymoving and renders them natively on mobile and on the web” (Lottie docs, 2021). We use it for our splash screen and for the loading animation.

## Front-end

### Homepage screen – the MapActivity

This is the most important activity in our app and displays the aircraft in airspace over a defined geographical boundary. The activity uses the `MapActivity` class that implements several `googleMap` interface functions. An instance of `LatLngBounds` is responsible for setting Norway as a centre of the map. Then, it executes the `openSkyRepository` function and uses `openSky` API to load aircrafts on the activities `linearLayout` view, `activity_map.xml`.

Furthermore, it implements `onMarkerClick` listener to listen to the user interaction so that when a user taps on an aircraft, it displays the flight number and a dialog alert call (android Appcompat feature, `AppCompat.Dialog.Alert`) which is declared in `AndroidManifest.xml`. As an intent activity, it displays a mini-popup-window containing the flight status information which is executed by the `flightStatusPopupWindow` activity classes and their corresponding internal data classes and functions.

### Flight status screen

This screen uses `FlightStatusInfoSearchFlight` class activity that executes `flightsFetch()` as the main function of the activity to fetch the important flight status

data from the flight status API up on user request with flight number as input. The activity extends `android AppCompatActivity` and implements several `NavigationView` interface functions for the navigation menus. It uses `flight_status_main.xml` layout which implements `RecyclerView` (displaying flight status info as main display), `NavigationView` (displaying main menu on top) and other layout such as `DrawerLayout` (supporting `NavigationView`), `ConstraintLayout` (containing views accepting flight number, `editText` as input field & search button) as well as `LottieAnimationView` (as loading interval animation).

#### Airport Weather screen

The main activity class responsible for this screen, fetching airport and live weather forecast data is `AirportsListActivity`. It extends the `AppCompatActivity()` and implements the `NavigationView` interface. The activity sets `activity_weather.xml` layout as content view. This layout implements `RecyclerView` (displaying a Norwegian airport), `ConstraintLayout` (holding the search button, `autoCompleteTextView` and toolbar) and `DrawerLayout` (holding relative layout and `NavigationView` for navigation menu)

The `recyclerView` implements a class of `airportAdapter` as `recyclerView` adapter's `viewHolder`. The class takes the `mutableList` of an `airport` as a `viewModel` matching the value of live data which was assigned as a mutable list of an airport in the `AirportsListViewModel` class.

The search button implements `setOnClickListener`, executing `matchAirportWithCity` function which takes the city (string) as a parameter. Once the search argument matches the airport city in the database, the `viewModel` displays the list of matched airports for this city. Then, upon user interaction with this result, it displays live weather data on the intent activity `ForecastActivity`.

#### APIs

##### Google Maps

Showing a map of Norway with some additional information is one of our basic requirements. There is a variety of map APIs and corresponding android libraries with this functionality. During the initial discovery phase, `MapBox` (`Maps SDK for Android`, 2021), `OpenStreetMap` (`OpenStreetMap wiki. Frameworks`, 2021), and `Google Maps` were explored.

Despite its seemingly high quality and rich feature set, MapBox was quickly discarded as it is not free. OpenStreetMap is completely open and free, but the largest libraries available for it are all lacking in some aspect or another. For example, osmdroid is severely under-documented and lacking in features, and WhirlyGlobe-Maply, while adequate, contains unacceptable bugs. This left Google's Maps SDK service, which is free without usage restrictions, and is additionally actively supported and maintained by Google. Not only is the service, therefore, very widely used and robust, but it also has built-in integration with Android as well as Android Studio (Google Maps Platform. Maps SDK for Android overview, 2021).

The service is part of Google's Cloud Platform and as such is not completely open and requires a registered account to set up. A project was created, and an API key generated for it on the Cloud Platform. Following usual convention, the key is stored in the app as a string resource in the `google_maps_api.xml` file. The library used is provided by the `play-services-maps` dependency. This library provides the developer with a relatively simple java interface that can be directly plugged in on the front-end. The API key is made available to the library as a meta-data element in the manifest file. The manifest file also contains permissions required by the library: internet and access fine location cover our needs.

We integrate the map into the app using a `SupportMapFragment` fragment. This fragment automatically manages the lifecycle of the map object and all network communication with the API in the background. The map is a simple fragment element in the layout file which is managed in the corresponding map activity. Here, we request the fragment and the layout manager to notify us once the map is ready and the layout is done, respectively. Once both events have occurred, we initialize the camera and UI, and enable the my-location feature.

As the requisite access fine location permission is considered a dangerous permission by android, it must be requested from the user at runtime. We do this in the usual way: call `requestPermissions` with an array of permission constants and an integer request code, and override `onRequestPermissionsResult` to first check that the request code matches, and we enable the feature if the permission is granted. The library then uses the device's GPS or other location features to display current location on the map. If the user denies the request, the feature is simply not available.

The rest of the map activity makes use of markers to show aircrafts on the map. A Marker is a simple local overlay that marks a specified point on the map. Based on received data, we continuously update positions of the map markers based on each aircraft's speed and track. The maps API represents coordinates on the Earth's surface in decimal degrees in the common WGS 84 coordinate system. The markers are also positioned using such coordinates, meaning they adjust accordingly as the map camera changes. This is convenient for interoperability with our other information sources, but also means finding aircraft positions involves some non-trivial calculations. This functionality is encapsulated by the `MapAircraftMovement` class and its `calculatePosition` method.

### OpenSky Network

There are not many easily available sources of live aircraft tracking data. We initially considered the OpenSky API for this, given in the recommended APIs section of the case description. As this API turned out to be well suited to our use case, as well as being free and unrestricted for non-commercial purposes, it was not necessary to try other possibilities. The OpenSky Network is a collaborative effort that continuously compiles information from ADS-B messages broadcast by aircraft and makes it available through their service. It is a public JSON-based REST API and does not require identification to access (The OpenSky Network API documentation, 2021).

There are multiple endpoints with different types of information. In our app, we make use of only a single endpoint, `states/all`, which returns near real-time information about the states of aircraft, such as ICAO-24 address, velocity, and position. ICAO-24 addresses are 24-bit integers uniquely identifying each aircraft transponder; they are conventionally given as 6-digit hexadecimal numbers. The endpoint is optionally restricted to an area by a coordinate bounding box, and likewise optionally to a list of specific ICAO-24 addresses. For anonymous users such as our app, the endpoint has a resolution of 10 seconds. In other words, the data is always time stamped with the most recent multiple of 10 seconds. We take this into account in our implementation.

We use retrofit to access the API, with the gson converter integration, since the responses are JSON. Our endpoint method returns an `OpenSkyStatesResponse`, which holds a timestamp and an array of so-called state vectors. A state vector holds the current state of some aircraft. It is a fixed-length JSON array, where each element has a well-defined

type and semantics in the documentation. OpenSky takes this approach rather than JSON objects to spare bandwidth. As the converter has no way of inferring these types and semantics, some of the information is missing from the returned objects.

The service is further encapsulated into a repository, `OpenSkyRepository`, as per the repository pattern. The repository converts API responses to objects with type safety and meaningful variable names, aiding convenient and error-free usage. The endpoint method takes less cumbersome parameter types, e.g. as `LatLngBounds` instead of four individual coordinates, and transparently converts them when calling the service. The repository also has the responsibility of enforcing the 10-second rate limiting via in-memory caching. Essentially, a map of method arguments to aircraft states is maintained. This is the first thing checked on before each request. If it already contains an up-to-date object for the given arguments (timestamped at or after the most recent multiple of 10 seconds), then that is returned; otherwise, the service object is used to fetch more up-to-date data, and the cache is updated with it.

The rest of the file is devoted to the private functions for the necessary conversions. The data structures yielded are found in `Aircraft.kt`. Worth mentioning is that points in time are represented as Unix timestamps, i.e. seconds since the epoch 1970-01-01 00:00:00. They are converted to Java standard library `Instant`s. The repository also has a `statesAllNew` method that always fetches new data instead of returning cached data. If up-to-date cached data is available, it first suspends until the next update, then it fetches from the service (and updates the cache as usual). It is used to implement the live update loop for our map activity.

## FlightStats

There are many available options for accessing flight status and delay information. The following APIs were considered:

- OAG flight info (OAG Aviation Worldwide Limited. OAG Flight Status, 2021)
- aviationstack real-time flights (Aviationstack. API documentation, 2021)
- flightlookup TimeTable Lookup (TimeTable Lookup API Documentation, 2021)
- AeroDataBox flight (AeroDataBox API, 2021)
- flightstats flight status (Cirium Flex API Reference, 2021)

All the above are paid, but either have free access restricted by a fixed monthly quota, or a free trial available. The OAG and aviationstack APIs proved too cumbersome to obtain trial keys for. The flightlookup API does not have look up by flight id. This left AeroDataBox and flightstats. Both alternatives provide the necessary data to calculate delays. However, the flightstats API is both more detailed, and pre-calculates the delays for you in the response, which is perfect for our use case. For these reasons we chose the flightstats API.

The only part of the API we need is the `flight/status` JSON endpoint. It takes as inputs a flight id and a date of arrival, and returns detailed information about the flight, its status, as well as related airports and airlines. A flight id is the combination of an airline code and a number, uniquely identifying a specific flight of the airline. A flight has a start and end airport and may have zero or more intervening stops. An airline code is a globally unique identifier for an airline (Airline codes, 2021). There are two standards for airline codes in use, managed by the two largest aircraft organizations:

- IATA two alphanumeric characters; for example: `QR` (Qatar Airways), `B6` (JetBlue Airways), `5X` (UPS Airlines).
- ICAO three alphabetic characters; for example: `QTR` (Qatar Airways), `JBU` (JetBlue Airways), `UPS` (UPS Airlines)

Flight numbers are identifiers for flights. They are unique internally to and assigned by a given airline (Flight number, 2021). Each flight number consists of:

1. 1-4 arbitrary digits.
2. An optional *operational suffix* (one alphabetic character) giving other information that is irrelevant to the app

Examples of valid flight ids are: `QR0579` (Qatar Airways DEL-DOH), `JBU1` (Jetblue Airways JFK-FLL), `5X18` (UPS Airlines CGN-HKG), `AY15Z` (Finnair HEL-JFK, with operational suffix). The canonical form of flight ids is all-uppercase, without spaces. Some APIs accept spaces and lowercase letters, some do not; some APIs might require separated airline codes and flight numbers. Most APIs do not appear to handle operational suffixes at all. To be safe, we first normalize any input flight number by:

1. Removing whitespace anywhere in the string.
2. Converting all characters to uppercase.
3. Removing the last character if it is a letter (operational suffix)



Then we split the flight number into airline code + number:

1. If the third character is a digit, parse as IATA.
2. Otherwise, parse as ICAO.

If any error is encountered in this process, we reject the flight number. We let the endpoints take care of further validation, and only check whether the response indicates an invalid flight number. The above described concept is encapsulated in class `FlightId`, with subclasses for IATA and ICAO flight ids, and a parse method that returns the appropriate one.

The FlightStats API also uses the Retrofit library. We separate functionality between the service class `FlightStatusService` and repository class `FlightStatusRepository`. We use the gson converter for converting into the raw data structures of the service class. The API specifies a special error object that may be present in each response, indicating an error. This must be checked after any API fetch.

Since it is unnecessary, this repository class currently implements no caching, but it is responsible for authentication. An application ID and application key must be passed as query parameters in every request (How to Use the Cirium Flex APIs, 2021). The repository adds a so-called interceptor to the HTTP client used by Retrofit that appends the query parameters to the URL before the request is made. The ID and key are stored as string resources in file `flightstats_api.xml`. Our app currently uses a trial account for this purpose (currently set to automatically disable after 2021-06-20). There is a hierarchy of custom exceptions to be thrown for unexpected response contents or explicit errors returned by the API. The rest of the file contains private functions converting service types to more convenient types that better correspond to the model in use in the app.

These types are found in file `Flight.kt`, with largely self-explanatory semantics. However, the concept of flight juncture may appear confusing at first. It is simply a generalization of an arrival and/or departure of the flight and corresponding airport. A departure or arrival juncture is a juncture the flight departs from, or arrives at, respectively. A mid juncture is one of the zero or more stops along the flight, and as such it is both an arrival and a departure juncture. Since java does not allow multiple class inheritance, modelling these concepts was done using interfaces. Concrete implementations used by the repository are the `Impl`-suffixed private classes.

### LocationForecast API

One of the basic requirements for the app was to display weather forecast information at the outgoing and inbound airports. Because of limitations set by the project requirements, only API's offered by MET were considered.

Initially we explored the TAF/METAR API for its apparent relevancy. TAF/METAR relates to weather reports specifically meant for aviation purposes. These include basic reports like air temperature and precipitation forecasts. While the information offered by the API was relevant, we eventually decided against it, however. There were many arguments to support this decision. First, the TAF/METAR API offered a lot of information that an average consumer has no use for. Secondly, TAF/METAR reports are highly abbreviated codes, which make it cumbersome to work with. We would have to dedicate a significant portion of our time and energy to document and implement decoding tools to work with the data. Finally, the API only offered data in txt or xml format which would require a parser and therefore more resources from the development team. Because of these arguments, we finally decided to use the location forecast 2.0 API. This API offers data in JSON format and only requires longitude and latitude coordinates to fetch forecast data. The API's adaptability along with its ease of use motivated us to favour this API over TAF/METAR.

All API's offered by MET are free to use within the boundaries set by the terms of service. MET sets a variety of conditions but only a handful directly relate to the functioning of our app (Locationforecast API documentation by MET, 2021). The most notable of these conditions relate to user identification and bandwidth. First, every call to the API must be made with a User-Agent (UA) header for authentication. The UA must include the application name and some sort of contact information. Either an email address or a link to a company website satisfies this requirement. Finally, MET restricts the amount of API calls an application can make per second. Currently this restriction is set to 20 calls/second per API. Failure to satisfy these two requirements will result in the app being blocked from MET's offered services (MET APIs terms of service, 2021).

### *MVVM design pattern*

It is important to mention that the implementation of the weather forecast functionality relies on two controllers, namely `AirportsListActivity` and `ForecastActivity`. The `AirportsListActivity` serves as a search and selection

screen. It provides an overview of airport objects that matched the user's search criteria. Selection of an item will redirect the user to a forecast activity. The forecast activity is responsible for displaying the weather forecast information of a specific airport to the user.

When designing the app, we attempted to follow key principles of GRASP as best as we could (GRASP (object-oriented design), 2021). The `AirportsListActivity` and `ForecastActivity` are both UI controllers. To optimize app performance and avoid unnecessary bloating of the class, we strived to maintain high cohesion across our classes. Since UI controllers should only be responsible for managing user interaction and displaying UI components, a clearer division of responsibilities was necessary.

Both controller classes have partner classes that implement the `ViewModel` interface. These classes are called `AirportsListViewModel` and `ForecastViewModel`, respectively. The main responsibilities of these classes are to prepare and retain data for their partnering activity classes.

#### *Airport database creation and usage*

As mentioned above, the API takes as inputs latitude and longitude coordinates to fetch forecast data of a specific location. To make the API calls, the app relies on information stored in instances of the `Airport` class. The `Airport` class contains relevant data about the airport like its geographical location expressed in longitude/latitude coordinates. A database of `Airport` objects was required to enable the activity to search through, match and list the results.

Since responsibility of data preparation is assigned to the `AirportsListViewModel`, this class handles the creation of the database. This agrees with the creator principle as specified by GRASP. The class contains methods to create the database asynchronously. Object creation is based on a csv file called `intair_city.csv` which can be found in the `raw` subdirectory of the resource folder. This csv file contains all the required information of all known international airports in the world.

Initialization of the database is performed asynchronously at start-up of the activity by calling the `AirportsListViewModel` method `createDb`. It takes as input the context of the class that calls it and the resource-ID of a csv file. It reads the entire database row by row and provides the information needed to create `Airport` objects. As mentioned earlier, we use a third-party library to enable the method to read the csv-file asynchronously.

Since I/O operations tend to be resource consuming operations, we thought it necessary to use a `viewModelScope` for this method. The database is represented by a `MutableList` of `Airport` objects. This is stored in the `LiveData` variable `AirportsLiveData` for future use.

#### *API call*

Selection of an item in the `recyclerview` initializes a forecast activity. The selected `Airport` object is transferred to this activity where the instance variables `latitude` and `longitude` are used as arguments in the method `callForecastAPI`. This method is contained in the `ForecastViewModel` class.

As with the `AirportsListActivity`, a partnering `ViewModel` class is responsible for the preparation and maintenance of data. The partnering `ViewModel` class to `ForecastActivity` is called `ForecastViewModel`. The method `callForecastAPI` uses the `Fuel` library to get responses from the API. Responses are returned in JSON format and converted into java objects using the `gson` library. These objects containing the weather forecast information are subsequently stored in their respective `airport` objects.

In retrospect, it was not necessary to store instances of the `Forecast` class in the `Airport` objects. The `LiveData` variable of `ForecastViewModel` would have been enough to display forecast information.

The file containing all the classes required to create forecast objects is called `Forecast.kt`. A JSON to kotlin plugin was used to generate all the data classes required. These classes are used to convert the JSON response of the location forecast API. Even though a lot of the information offered by the API response is left unused, we decided to retain it anyway. This was done with an eye to the future, as we could see this data being used in future iterations of the project. For example, the response contains forecast information for the next couple of hours. Currently, our app only displays the current forecast for the next hour or so. A future iteration could have shown a forecast for the next 6 to 24 hours. We did not have the time to implement this however, so the class is left unaltered in our file repository.

## Test documentation

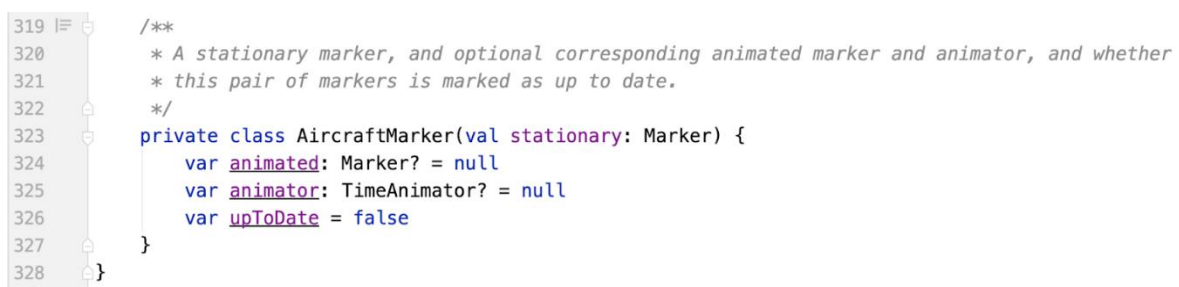
As for our testing documentation we divided our testing into 2 parts: static testing that is performed on requirements, design, or code without executing the code, and dynamic testing - executing the code and using different tools for this purpose.

The main goal of our testing processes is to identify errors, gaps or missing requirements in contrast to actual requirements, to ensure a stable codebase and feel comfortable when introducing changes, without worrying that it will “break” the code that has already been implemented.

### Static testing

Static test techniques provide a powerful way to improve the quality and productivity of software development by assisting us to recognize and fix our own defects early in the software development process (Graham, 2012). At the basic level, to look for potential improvements in the code and to prevent us from accumulating technical debt we used tools in Android Studio (IntelliJ) that pointed to places where we had bad code practice or small errors such as unused variables. The static testing technique we used is code review.

We executed code reviews in GitHub by emphasising all the changes that pull request (PR) introduced. Conversations happened alongside the code. In case there were some questionable parts, we left detailed comments on code syntax or asked questions about structure inline (Figure 16). The PR was never merged before we came in agreement in all our discussions. The comments were typically written on the variable names or whether a function solved the problem it was supposed to solve. Along with looking for bugs and misunderstandings in the code, the reviewers commented on the readability and understandability of the code. It was also discussed how we would proceed further.



```
319 |  
320 |  
321 |  
322 |  
323 |  
324 |  
325 |  
326 |  
327 |  
328 |  
... |  
/**  
 * A stationary marker, and optional corresponding animated marker and animator, and whether  
 * this pair of markers is marked as up to date.  
 */  
private class AircraftMarker(val stationary: Marker) {  
    var animated: Marker? = null  
    var animator: TimeAnimator? = null  
    var upToDate = false  
}
```

Figure 16. Screenshot of code review.

To ensure that code reviews were executed on each new line code, we modified the configuration on GitHub to “protect” the default main branch. Thus, it was not possible to push straight to main, so that the new code had to be introduced through a pull request. As an extra measure, we use peer review and we think that each commit should be approved by another person, not the author, to be merged to master.

It affected the development process positively and everyone understood better the code that was merged to main. In addition, it made the development process faster since we were always updated on the code. Considering that no one in the team had previous experience with Kotlin, reading each other’s code helped us learn and get a deeper understanding of the programming language. As everyone in the group had different views and product visions, we needed to spend some time to come to a common understanding of the app functionalities. Therefore, we tried to not be too critical towards each other’s PR. However, after some time and work on adding new functionalities and fixing bugs, we were satisfied with the trade-off.

Another method we used for static testing was automatic formatting of the code. For each commit we formatted the code with the built-in formats of Android Studio (by pressing Command + Option +L). It was used as a standard of how a code should look like so that the code is readable for everyone.

### Dynamic testing

Under Dynamic Testing, a code is executed. It checks for functional behaviour of the software system and overall performance of the system. The main objective of this testing is to confirm that the software product works in conformance with the set requirements. Dynamic testing executes the software and validates the output with the expected outcome (Graham, 2012).

We focused on testing the function utility and that the methods parse and retrieve data from APIs. Therefore, we didn’t use a targeted planning of testing, but tested the code that didn’t have to do anything with the UI. Tests helped us feel confident about making changes without introducing new hidden bugs.

We executed testing at 4 different levels which helped us to check behaviour and performance for software testing. At each level we tried to recognize missing areas and reconciliation between the development lifecycle states. These central test levels in system

development are unit testing, integration testing, system testing and user testing (acceptance testing) (ibidem).

## Unit testing

None of us have previous testing experience with Android Studio. As Implementing the functions was our first priority, we started our testing procedure at a relatively late stage of our development. We choose to write local unit tests because it runs faster and does not require the fidelity associated with running tests on an emulator. We store the source files for local unit tests at module-name/src/test/java/ and mainly use Junit 5 and Mockito for testing.

We tested individual components with the purpose to validate that each unit of the code performs as expected. We wrote some unit tests for our `AirportList` class (Figure 17). We use Mockito to mock-up the context and create some mock-up objects and then check if the function returns a result as expected using `assertEquals()`.

One-unit testing technique we used is black box testing which involves testing of user interface along with input and output, to make sure that the function behaved as supposed with expected and unexpected input, and handle edge cases.

After all, we noticed that testing led to a better design of the code. We also learned from our experience that it is important to start testing in the early stage and testing driven development (TDD) is also something that we can use in our future projects.

```
9  class AirportsListTest {
10  //mockup context and airportList
11      private val mockAirportsList : MutableList<Airport> = mutableListOf()
12      private val mockContext: Context = Mockito.mock(Context::class.java)
13      private val testAirportsList: AirportsList = AirportsList(mockContext, 0, mockAirportsList)
14  //check if the it creates a right airport
15      @Test
16      fun testCreateAirport() {
17          val expectedAirport = Airport("ICA01","City1", "Name1","Country1", 0.0, 0.0)
18          val createdAirport = testAirportsList.createAirport("ICA01","City1", "Name1","Country1", 0.0, 0.0)
19          assertEquals(expectedAirport.ICA0, createdAirport.ICA0)
20          assertEquals(expectedAirport.city, createdAirport.city)
21          assertEquals(expectedAirport.country, createdAirport.country)
22          assertEquals(expectedAirport.name, createdAirport.name)
23          assertEquals(expectedAirport.latitude, createdAirport.latitude, 0.0)
24          assertEquals(expectedAirport.longitude, createdAirport.longitude, 0.0)
25      }
```

Figure 17. Screenshot of unit testing

### Integration testing

The purpose of this level of testing is to expose defects in the interaction between different software models when they are integrated (Graham, 2012). Integration Testing focuses on checking data communication amongst these modules. During our development process, integration testing was under prioritized. We rationalised it by the fact that there were many changes from week to week, and the first weeks we used to test different components. There was no guarantee that the functionality we created one day would be a part of the app next week, and to spend time on writing continuously new integration tests would go at the expense of the app's functionality. We decided to focus on bootstrapping our app with new functionalities to make it robust and write tests when dependencies fell into place. A good way to ensure that the tests were run for every new PR, was to set up CI (continuous integration) in GitHub, that would automatically run all our tests. It is a favourable way to make testing as easy as possible.

### System testing

System testing involves testing the software. In our project, we tested the application we had created by using a specification-based testing technique called use-case testing. As our starting point, we thought which use-case the system should implement and tested that these use-cases were performed flawlessly.

By the end of the project we focused on the UI tests after all modules and components had been integrated. Unlike the unit test or integration test, a UI test is not limited to a module or a unit of the application, it tests the application as a whole (database storage and components state update among other things). At this level we check how the application handles user actions, ensure that users do not encounter unexpected results and check whether visual elements are displayed and work correctly.



## Process documentation

### Methodologies used in the project

In our project, we incorporated methods of Scrum and Kanban adjusting them to our needs. The elements we implemented or tried to implement from Scrum are product backlog, timeboxed sprints, team velocity, roles within the team, daily scrums, time boxed sprints and self-organizing teams. As supporting tools, we used Slack for communication and Notion and G-suite to organize our work.

Very early into the project, we decided to assign scrum roles and divide into self-organizing teams. We had two teams who were responsible for the backend and frontend of our app. The front-end team consisted of three members and the backend team had four – one person worked on both teams throughout the project because of their personal interests. The teams were responsible for delivering certain features of the app and were self-organizing meaning that they both took ownership of their tasks without needing a “push” from the outside.

Besides the developers, the only scrum role we adapted was scrum master. However, the tasks he was doing were broader than the definition specifies. Besides making sure the methodology is implemented, with time, he also assumed the role of project manager. He was responsible for leading the project, planning meetings, and taking notes, assigning tasks at times, and making sure that everything outside of the development process works well. When it comes to the role of product owner – there wasn’t a person in our team who took this role as we didn’t feel the need for that. We took a democratic approach to our product ownership and decided collectively which features we want to implement and assessed progress with them along the way.

Another element we adapted from Scrum is product backlog. At the first stage of our project, we defined use cases and user stories by brainstorming and conducting potential user interviews. The backlog had a form of a table we stored in Notion. Each element had an assigned priority from 1 to 4 as you could see in the previous chapter. The backlog served us as a guide and kept us focused on the most important parts. We didn’t add, however, extra things that according to scrum methodology could be added e.g. “Fix the bug XYZ”, “Implemented changes in the weather forecasts”. Such things were decided on the go when planning the sprints.

Our sprints usually lasted one or two weeks. In an ideal world, each would last at least two weeks, but we decided we meet more often and have more frequent sprints because we had only around 10 weeks to develop the whole app. Each sprint had two fixed obligatory meetings – planning sessions and retrospectives. During planning meetings, we discussed at what stage we are and what needs to be done further. Our retros had a fixed structure and usually were very fast unless there were extra issues or concerns we needed to discuss. During the meetings we always asked us these three questions: what's going well during the sprint, what could have been done better, and what steps we need to take to improve our cooperation. Additionally, each sprint had its backlog. For this purpose, we used Notion's feature to create a Kanban board. Each post-it was a task assigned to certain people that needed to be completed at a given sprint. Each task had one of three statues: not started, working on, or completed. Additionally, they were assigned to sprints so we could filter our board by the sprint/person of interest.

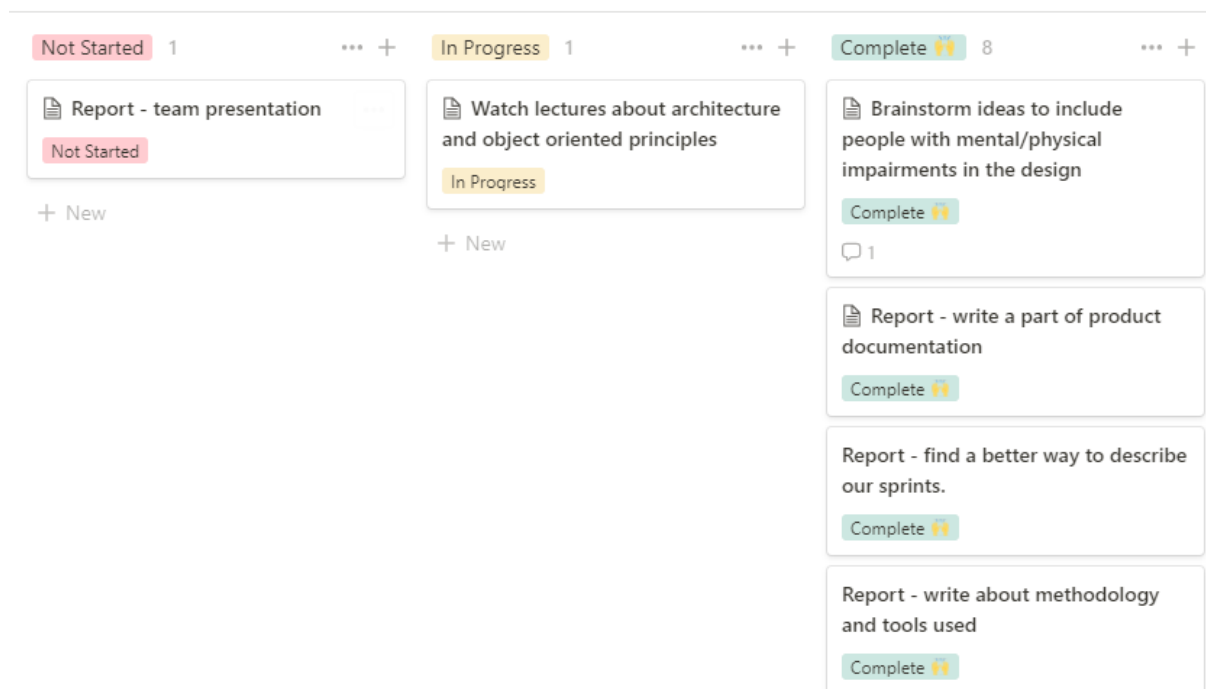


Figure 18. Screenshot of our Kanban board for one of the sprints in Notion

Finally, the element we tried to introduce from Scrum were daily scrums and team velocity using story points. Unfortunately, we weren't successful. We found it difficult to estimate story points for each and single tasks and we decided not to use them. Another reason was that everybody was taking tasks according to their capacities at the time. Additionally, the team was very good at fulfilling the tasks we said we would do during the sprints. We had very few tasks that needed to be moved to the next sprint because of capacity issues. In terms of daily scrums – we weren't successful because of our vastly different schedules and for the reasons mentioned earlier – that we were doing our tasks regardless of if we had daily scrums or not. We tried them a few times and found them not very useful for us.

It is notable to mention that when forming our group, we came up with a team contract which you can find in the attachments.

### Course of the project

Our project was divided into a total of 8 sprints with varying length from one to three weeks. Below table shows, in general terms, how we were progressing through the project with every sprint. The whole project can be roughly divided into three phases: start, development and finish. With some tasks overlapping between the development and finish phases.

	Group forming	Deciding the scope	Learning about technologies	Main backend work	Main frontend work	Polishing the app	Test and code reviews	Working on the report
Sprint	Start phase			Development phase		Grooming phase		
1								
2								
3								
4								
5								
6								
7								
8								

### Start phase

The first phase of our projects focused mainly on the “soft” part of things. At that point we were forming the group and deciding how we will work together – we reached agreement on the methodologies and tasks division.

Additionally, we narrowed down scope of the project. We used brainstorming and use interviews to decide what user stories we might want to implement. Eventually, by voting, we decided on our backlog. At this phase we also made some diagrams for the final report.

Finally, what was crucial for our project, we familiarized ourselves with the available APIs and decided which ones we would choose basing on the data they provide and their user-friendliness. To achieve that, we also needed to learn about nomenclature used in aviation.

### Development phase

The second phase focused mainly on product development. For a couple of weeks, we were working solely on the app itself. We encountered certain challenges with APIs that brought us back to starting phase when we needed to reconsider which ones to use. It caused another challenge – we had situation where two people were implementing the same functionality but using either the same or different API effectively wasting time of one person. To solve that, as a team, we made a decision whose code we were going to use further into the project.

This phase overlaps to a certain extend with the last phase – we wanted to start working on the report early. We decided what exactly needed to be written and wrote what we could at a given point of time. Furthermore, we reviewed each other’s code.

### Grooming phase

The final stage of the projects consisted mainly of final touches, writing the report, and generally finalizing the project. We made sure that what we implemented up to this point is well commented, the code is understandable etc. Additionally, we decided not to implement “can have” features from our backlog because we preferred to polish what we had. In this phase, we also continued with code reviews and did testing.

## Reflections

To conclude the project, we conducted an anonymous survey among the team members to collect data about how each individual experienced working on the project. The survey consisted of four sections and covered the areas teamwork, methodologies and project management, the app, and the project in general. Each section consisted of a set of multiple-choice questions and one open question. This gave everyone the space to freely voice final thoughts and opinions. Additionally, in the following section, we included observations we had made over the course of the project.

### Teamwork

In general, everyone was satisfied with the teamwork. According to the survey, everyone felt respected and safe to express opinions and ideas. The survey also showed that everyone agreed that tasks were divided according to people's preferences and capabilities.

Despite these achievements, the team answered all across the board on questions like how the team communicated with each other and the desire to influence team dynamics. This was reflected in a few comments we received to the final open questions. Apparently, there were suspicions about to what degree team members really were voicing their concerns and opinions. Some of us felt like there were some unspoken issues persisting throughout the project. Despite attempts to address issues during retro meetings, these never came to light. This particular comment also addresses the issue of perceived hesitation regarding communication. This is an interesting remark since most answers regarding communication were on the positive end of the scale .

There were also comments addressing a lack of transparency regarding what other team members were doing. Our team was divided in two self-governing groups, responsible for the back-end and front-end respectively. Apparently, it was not always clear how the front-end team collaborated and what they were working on. One comment gave expression to a wish of letting the front-end team present their progress more frequently. The same comment also wished for more collaboration on the back-end team. The writer felt more could have been implemented if several programmers were working on a use case at the same time. The final version of the app has a feature which contains a flaw that could have been fixed sooner. The writer thinks the reason this was left in the app was due to the group dynamics and the tendency by the members to work in isolation on code. An example is given about a situation where two members of the team were working on their own

implementation of a single use case. This wasted the time of one team member, and the writer wished this situation was addressed sooner in the project.

These comments somehow align with how a lot of our meetings were going. It wasn't unusual that a meeting was dominated by 1 - 3 people without a lot of input from other team members. We believe that everybody is a valuable member of our team who has knowledge and skills to deliver parts of the projects in which they specialize. We think that the project could have gone smoother and our app could have been way better if everybody in the team was sharing their ideas and doubts with the rest of the team.

Despite all the above comments that may shed negative light on our teamwork, one thing needs to be mentioned and emphasised. The team did amazing work in the given time frame. Every single team member delivered tasks they said they would deliver. We had very few delays that usually were connected to secondary functionalities and didn't cause serious delays in the project. Of course, there are many things that could have gone better in terms of teamwork and communication, but we did a great job in the end. Moreover, we were doing tasks that actually were fun to us no matter if it was design, development or project management. We think that's a major success.

### Methodologies and project management

The survey shows that everyone was satisfied with the frequency of the meetings and what they covered. Team members were also generally satisfied with the chosen tools, saying that they covered our needs. An interesting conclusion drawn from the survey is that everyone felt like they could have done more.

In terms of satisfaction regarding how we executed the sprints, the team answers across the board. While most answers were at the positive end of the results, some people weren't as satisfied and thought that we could have managed sprints better.

This is reflected in a comment which stated that the group failed to properly define and establish a healthy set of practices for using certain tools and techniques. The writer expresses a dissatisfaction in how the Kanban board was used as an example. The comment mentions that there was a lack of a unified practice and how people didn't take enough responsibility for creating, maintaining, and updating the tasks they were assigned to. The same comment also mentions difficulties establishing a structure for the obligatory meetings during the first half of the project, which is a symptom of the same above-mentioned issue.

The writer felt that this resulted in defining inconcrete tasks. The writer did mention that we improved a lot over the course of the project but wished we had adapted a bit quicker.

### The app

In terms of the app itself, there seems to be a consensus that our app is a finished app ready for submission, all things considered. The team seems satisfied with the quality of the code and is proud of what has been achieved. Most team members liked the design of our app. There was an individual who didn't like the design all, however. To each their own, as they say.

Answers to the question what team members would work on if we had more time show a big degree of unanimity. The entire team would have spent more time and resources on improving the use cases we have instead of implementing more. This could be achieved by improving the UI, cleaning up the code, and adding additional features to implemented use cases.

### The project in general

All in all, everyone agreed that it was a pleasant experience to work on the project in this team. We unanimously agreed that it was a valuable, educational experience which allowed us to improve our technical and soft skills. The team did answer all across the board on the question whether the course is worth 20 study points. No comments were provided to give substance to people's choices.

During the final meetings we had around the day of submission, the team expressed general satisfaction with how the project went. We were happy that we achieved what we wanted. The process, all in all, went smooth despite some bumps on the road. On the day of submission, we didn't need to rush anything and could polish both the app and the report.

### Influence of Corona situation

We believe conference tools can never fully replace physical meetings. A lot of team building happens outside of the meetings. Conducting projects solely digital is too artificial for meaningful human connection to develop. It is easier to engage in conversation with a team member after meeting physically. This is not the case in digital meetings where there are more obstacles to talk with another person one on one. It feels weird to hang around and chat with people you barely know in a digital environment somehow. Something could also be said

about how a screen can act as something to hide behind. People are less motivated to talk when there are no people in their physical surroundings. All in all, we believe Corona has negatively affected the team culture.

On the other hand, however, having everything digital makes us save time on commuting etc. It is also important to be able to work fully digitally on IT projects since more and more companies (especially after Corona) allow fully remote work. Having such a big project in corona conditions gave us insights into what our workdays may at some point look like. It also taught us how to structure our days with regards to meetings and when we work on the project. Nonetheless the pros don't outweigh the cons - the whole team missed an opportunity to physically meet and enjoy each other's company. We all need physical meetings with fellow humans especially after almost 1.5 years in lockdown. For those of us who started studies in 2019, 3 out of 4 semesters were digital, which takes a lot from the study experience. It is not uncommon that students suffer from mental health problems because of that. We find this situation frustrating but there is nothing we can do. Maybe the content of the course could somehow facilitate team building activities by giving us e.g. Ideas of how to create team spirit when everything is online?

### Requirements met

In this section we would like to quickly summarize what functional and non-functional requirements we managed to implement. Since a picture is worth a thousand words, we put here the requirements tables and mark with green requirements that we met to a satisfactory degree.

#### Functional requirements

Type	Nr	Specification
Must	1.0	Show overview of planes over Norway
	1.1	Show Norway's location on map (default view)
	1.2	Present information about a flight's duration and distance, flight number upon clicking.
	1.3	Detect user's location from GPS
Should	2.0	Display information of airports in a country
	2.1	Show weather at each airport according to the time entered by user
	2.2	Predict the turbulence in a specific area
	2.3	Show information about CO2 emissions
Can	3.0	Show data about delays and cancellations of flights (pick up feature)



## Non-functional requirements

Type	Nr	Specification
Performance	1.0	The application shouldn't take more than 3 seconds to load initial screen. It should not hindrance to the user input.
Scalability	1.1	The app should be capable of handling increased user's data without delay.
Responsiveness	1.2	When app gets interrupted by call, it should be able to save state and return to same state which was before it got interrupted.
Usability	1.3	The flow of the app should be easily understood without the help of any guideline or experts/manuals.
Reliability	1.4	When user performs actions, they should be acknowledged with confirmation.
Availability	1.5	Users can access the application everywhere, install it, look for regular updates and give feedback.
Screen adaption	1.6	The application should be able to render its layout to different screen sizes.
Network coverage	1.7	The app should be able to handle slow connection. It should be able to look for WiFi if not available then automatically switch to mobile network.
Accessibility	1.8	The app should be developed according to universal design, for people of different physical possibilities.
Compatibility	1.9	The app must run on a different configuration, database, mobile devices and their versions.
Fetching data	1.10	Present data in the real time.
	1.11	System must fetch data from API upon the user's request of information.
	1.12	System must use the flights-API from OpenSky.
	1.13	System must use the weather-API from Meteorological Institute.
	1.14	System must use the turbulence-API from IPPC turbulence prognosis.
Organisational requirements	1.15	The app should be developed in Android Studio where Kotlin is used as programming language.
	1.16	App development must be performed with the agile process modelling Scrum.
External requirements	1.17	The system should follow the Personal Data Act and Data Protection Regulations (GDPR).

As for our backlog:

Priority	User story
1	As a person who wants to pick someone up from the airport, I want to have a time estimation of when the plane is going to land, so I can plan my day efficiently.
1	As an average consumer/flight attendant who is going to travel/work, I want to know if my plane is delayed, so I can plan how to spend my time in advance.
1	As an average consumer, I want to know what the weather at the outbound airport is like, so I can plan what to wear in advance.
2	As a plane enthusiast I want to be able to search for a specific flight and see detailed information about the aircraft (the flight route on the map). So that I can accumulate my knowledge of different aircrafts and routes.
2	As a plane enthusiast, I want to be given the ability to tap on planes shown on the map, so I can get information about that specific flight.
3	As a plane enthusiast I want to know/see how many planes are airborne in Norwegian airspace, so I can satisfy my curiosity.
4	As an environmentally conscious consumer, I want to know how much CO <sub>2</sub> is emitted on my flight from point A to B to help me make environmental conscious decisions
4	As a flight attendant, I want to have a list of the flights I'm going to take on a given day, So I can plan my day more efficiently.
4	As an average consumer, I want to be able to see how other people rated this flight/airline.
4	As a consumer, I want to see information about the facilities at an airport
4	As a consumer, I want to be shown the time difference calculation of arriving and departing flights when I have a transfer, so I know how much time I have at the airport.

Both in terms of the requirements and our backlog, we managed to implement all most important functionalities and user stories. In our final survey we asked each other what things we would implement had we had more time and resources. They were: combine the search bar with the map in one screen, better setting options, more unified UI, showing airports and flight paths on the map. We think we would make our code even cleaner and more readable. Regarding the user story *“As a plane enthusiast, I want to be given the ability to tap on planes shown on the map, so I can get information about that specific flight.”*. We managed to implement this feature, but very late in the project, we realized it is not perfect – it shows all kinds of aircraft in the Norwegian airspace. When user clicks on a marker that does not belong to a commercial flight, no data is displayed. Non commercial flights er e.g. private jets, helicopters etc.

## Final words

The project we worked on during the past almost 3 months was a valuable learning experience to all of us. We have learned a lot about teamwork, how we ourselves behave in a team setting. Moreover, we improved our coding skills. If we had to do a project one more time, there are things that we would like to do better. For sure, we would facilitate team communication in a better and more open way making extra sure that each team member freely expresses their ideas and shares knowledge with others. In terms of the design process – when we look at our backlog and requirements completions, it may seem that we were too ambitious at the beginning. And that's true. Initially we thought about many use cases that we could possibly implement. Our lack of experience told that's that, of course, we could do that. Fortunately, we did great job at prioritizing our tasks and, in the end, implemented everything with highest priority.

The work we have done gave all of us ideas about what we need to improve in terms of working with others. This learning experience will help us in the future as it gave us import insight on how we behave and what roles in the team we prefer to assume.

## References

AeroDataBox API. (n.d.). Retrieved March 19, 2021, from <https://doc.aerodatabox.com/>

Airline codes. (2021, March 24). Retrieved April 05, 2021, from

[https://en.wikipedia.org/wiki/Airline\\_codes](https://en.wikipedia.org/wiki/Airline_codes)

Aviationstack. API documentation. (n.d.). Retrieved March 21, 2021, from

<https://aviationstack.com/documentation>

Black, R., Veenendaal, E. V., & Graham, D. (2012). *Foundations of software testing ISTQB certification*. Australia: Cengage Learning.

Cirium Flex API Reference. (n.d.). Retrieved March 18, 2021, from

<https://developer.flightstats.com/api-docs>

Coroutines basics: Kotlin. (n.d.). Retrieved May 17, 2021, from

<https://kotlinlang.org/docs/coroutines-basics.html>

Doyaaaaaaken. (n.d.). Kotlin-csv user documentation. Retrieved April 1, 2021, from

<https://github.com/doyaaaaaaken/kotlin-csv>

Flight number. (2021, March 19). Retrieved April 04, 2021, from

[https://en.wikipedia.org/wiki/Flight\\_number](https://en.wikipedia.org/wiki/Flight_number)

Google. (2020, May 02). Gson user guide. Retrieved May 16, 2021, from

<https://github.com/google/gson/blob/master/UserGuide.md>

Google Maps Platform. Maps SDK for Android overview. (n.d.). Retrieved May 15, 2021,

from <https://developers.google.com/maps/documentation/android-sdk/overview>

GRASP (object-oriented design). (2021, February 25). Retrieved May 10, 2021, from

[https://en.wikipedia.org/wiki/GRASP\\_\(object-oriented\\_design\)#::~text=The  
different\\_patterns\\_and\\_principles,to\\_many\\_software\\_development\\_projects](https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)#::~text=The_different_patterns_and_principles,to_many_software_development_projects).

Guide to app architecture: Android Developers. (n.d.). Retrieved May 21, 2021, from

<https://developer.android.com/jetpack/guide>

How to Use the Cirium Flex APIs. (n.d.). Retrieved April 3, 2021, from

[https://developer.flightstats.com/api-docs/how\\_to](https://developer.flightstats.com/api-docs/how_to)

Java API Reference. (2021, March 31). Retrieved May 10, 2021, from

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/time/package-summary.html>

Kittinunf. (n.d.). Fuel documentation. Retrieved May 9, 2021, from

<https://github.com/kittinunf/fuel>

Kotlin. (2020, November 26). Kotlin Coroutines. Retrieved May 12, 2021, from

<https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md>

Kotlinx.coroutines reference documentation. (n.d.). Retrieved May 12, 2021, from

<https://kotlin.github.io/kotlinx.coroutines/>

Locationforecast API documentation by MET. (n.d.). Retrieved April 4, 2021, from

<https://api.met.no/weatherapi/locationforecast/2.0/documentation>

Lottie docs. (n.d.). Retrieved April 19, 2021, from <https://airbnb.io/lottie/#/README>

Maps SDK for Android. (n.d.). Retrieved May 11, 2021, from

<https://docs.mapbox.com/android/maps/guides/>

MET APIs terms of service. (n.d.). Retrieved March 20, 2021, from

<https://api.met.no/doc/TermsOfService>

OpenStreetMap wiki. Frameworks. (n.d.). Retrieved May 12, 2021, from

<https://wiki.openstreetmap.org/wiki/Frameworks>

Retrofit. (n.d.). Retrieved May 11, 2021, from <https://square.github.io/retrofit/>

Sommerville, I. (2021). *Engineering software products an introduction to modern software engineering*. Harlow, England: Pearson.

The OpenSky Network API documentation. (n.d.). Retrieved May 14, 2021, from

<https://opensky-network.org/apidoc/>

TimeTable Lookup API Documentation. (n.d.). Retrieved March 21, 2021, from

<https://rapidapi.com/flightlookup/api/timetable-lookup>

Understand the Activity Lifecycle : Android Developers. (n.d.). Retrieved May 19, 2021,

from <https://developer.android.com/guide/components/activities/activity-lifecycle>

Understanding typography. (n.d.). Retrieved May 20, 2021, from

<https://material.io/design/typography/understanding-typography.html#readability>

Use Java 8 language features and APIs: Android Developers. (n.d.). Retrieved May 15, 2021,

from <https://developer.android.com/studio/write/java8-support>

Use Kotlin coroutines with Architecture components. (n.d.). Retrieved May 12, 2021, from

<https://developer.android.com/topic/libraries/architecture/coroutines>

ViewModel Overview : Android Developers. (n.d.). Retrieved May 19, 2021, from

<https://developer.android.com/topic/libraries/architecture/viewmodel>

Worldwide, O. A. (n.d.). OAG Aviation Worldwide Limited. OAG Flight Status. Retrieved April

21, 2021, from <https://www.oag.com/flight-status-data>

## Attachments

### Group contract

1. Do your best to be at meetings (and on time).
2. If you are not present during a standup, send us a Slack message.
3. Presence at retro and planning meetings is obligatory.
4. Should a conflict arise, scrum master will try to solve it. If a solution cannot be found, we will ask veiledler to intervene.
5. Be prepared for the meetings.
6. Be honest - don't be afraid to say that you didn't manage something, you have problems with something or fucked up.
7. Be nice to your team
8. Assume full responsibility for your task.
9. We ask each other for help if we face difficulties.
10. We expect everybody to read what others write (documentation) and we document our work.
11. We reduce uncertainty by clearly defining our roles and tasks

### Interview questions

1. How often do you fly?
2. Have you ever used similar apps?
3. What kinds of apps did you use this?
4. Why did you use those apps?
5. What did these apps do well?
6. what core functionality should this app consist as your opinion? (Everyone)
7. How satisfied where you with this app's functionality?
8. How did you find out that your flight had been delayed/cancelled? How would you describe this experience? (For those who don't use such apps)
9. What do you expect the app to do? What do you think the most important functionality would be?

### Interview consent form

We are a student group in the course IN2000 “Software engineering with a project work” at the Department of Informatics, University of Oslo. The project group consists of Mohammed Abubeker Mohammed, Bence Richard Kalmar, Yan Jiang, Stela Ceaicovscaia, Gert-Jan Haasnoot and Piotr Grzegorz Kotkowski.

Our course teacher is Yngve Lindsjörn e-mail: ynglin@ifi.uio.no, phone: +47 22840856

Our project is aimed to develop a FlyNerd app that gives an overview of planes in air over Norway as well as shows the weather at the airports. We will also write a report on this. Therefore, we wish to interview you on the topic of requirements to app functionalities. The interview will take 20-30 minutes.

Everyone in the interview will be anonymous in the report. Notes from the interview will only be read by us - students in the project group, and the teachers in the course. We don't anticipate that there are any risks associated with your participation, but you have the right to stop the interview or withdraw from the research at any time.

If you wish to withdraw your consent or have any questions, you may reach us via this email address: stelac@ifi.uio.no

This consent form is necessary for us to ensure that you understand the purpose of your involvement and that you agree to the conditions of your participation. Would you therefore read the accompanying information sheet and then sign this form to certify that you approve the following:

1. The interview might be recorded. If it is, transcript will be produced you will be sent the transcript and given the opportunity to correct any factual errors.
2. Access to the interview transcript will be limited to participants of our project group and university colleagues with whom he might collaborate as part of the research process
3. Any summary interview content, or direct quotations from the interview will be anonymized so that you cannot be identified, and care will be taken to ensure that other information in the interview that could identify yourself is not revealed
4. Any variation of the conditions above will only occur with your further explicit approval



All or part of the content of your interview may be used:

1. In our research report.
2. On UiO website and in other media that we may produce such as spoken, presentations, in an archive of the project as noted above.

By signing this form, I agree that:

1. I am voluntarily taking part in this project. I understand that I don't have to take part, and I can stop the interview at any time.
2. The transcribed interview or extracts from it may be used as described above.
3. I have read the Information sheet.
4. I can request a copy of the transcript of my interview and may make edits I feel necessary to ensure the effectiveness of any agreement made about confidentiality.
5. I have been able to ask any questions I might have, and I understand that I am free to contact the researcher with any questions I may have in the future.

---

Name and date

---

Signature

