PUC Minas – Thomas More

Internship Project: AirVision

Realization

Seppe Volkaerts, Wesley Janse

Contents

1 Goal 3
2 Front-end 4
2.1 Analysis 4
2.1.1 Flutter 4
2.1.2 TensorFlow Lite object detection 5
2.1.3 Google maps integration 7
2.2 Mockup 7
2.2.1 First version 7
2.2.2 Final version 8
2.3 Implementation 9
2.3.1 Mobile application 9
2.3.2 Language/framework 9
2.3.3 Build System 10
2.3.4 Object detection 10
2.3.5 Georeferenced app development with google maps 13
2.3.6 Settings and debugging 15
2.3.7 Device sensors 16
2.3.8 REST API Communication 18
2.3.9 Dependencies 20
3 Back-end 22
3.1 Analysis 23
3.1.1 REST Web Service 23
3.1.2 Database 23
3.1.3 ADS-B ground station 24
3.1.4 Aircraft matching algorithm 25
3.2 Implementation 30
3.2.1 Programming language 30
3.2.2 Build System 30
3.2.3 Programming patterns 30
3.2.4 Database 31
3.2.5 ADS-B integration 31

3.2.6 Dependencies....33
3.2.7 Third-party API's....34
3.2.8 Downloading updates from a file server....37
3.2.9 OpenFlights....38
3.2.10 JSON serialization....41
3.3 Aircraft matching algorithm....41
3.3.1 Coordinate system conversion....44
3.3.2 Convert ECEF to 2D camera positions....46
4 Front-end to back-end communication....49
4.1 REST Specification....49
4.1.1 Response....49
4.1.2 Request....50
5 Testing....56
5.1 Test data....56
6 References....57

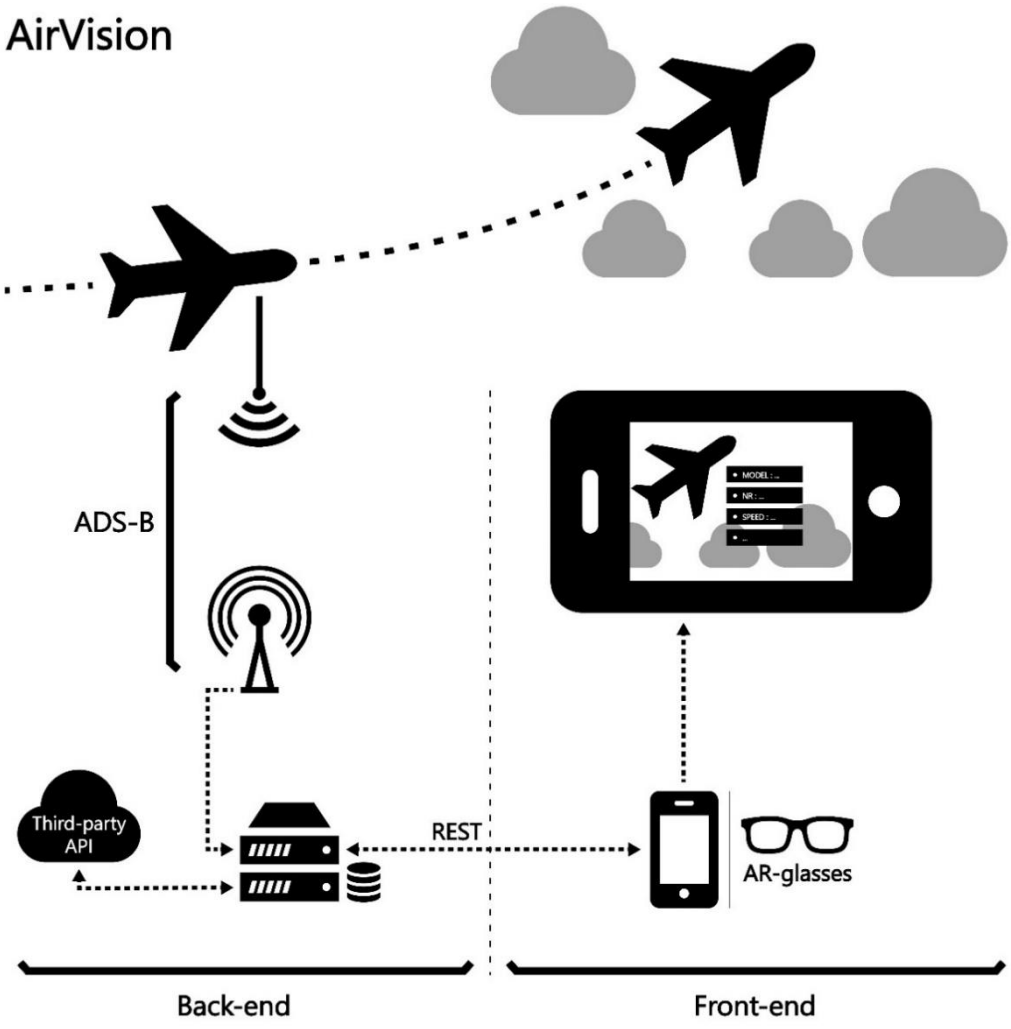
1 Goal

This project is application for mobile devices which can be used to see information about aircraft that the device is pointing at with its camera. There are some additional features such as an integrated map on which you can see the aircraft. Initially the application will be made for a mobile phone, in the future this can be extended to AR-glasses.

This document will describe the steps in the development of the application. Starting with an extended analysis, mockup, implementation and finally testing. The analysis mainly consists of acquiring information on algorithms, formulas, which data are needed to achieve the project. The implementation will go more in depth on how the analysis was implemented, which libraries or third-party API's were used for this, etc.

The application is split in a front-end, which is the mobile application. The back-end, which will handle all the data of aircraft, flights, etc. And finally, the REST API and its specification which will be used to communicate between the front and back-end.

The following image gives an overview of the structure of the project.

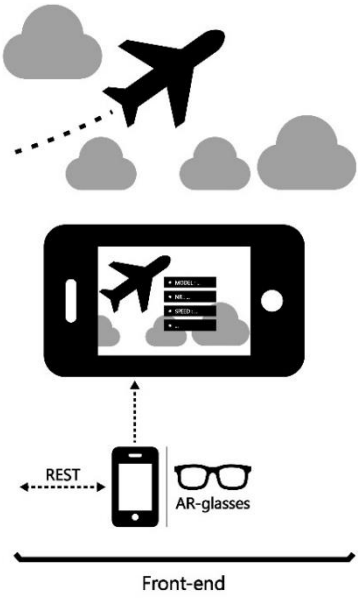


2 Front-end

2.1 Analysis

The front-end of the application consists of a few key components:

1. Flutter
2. TensorFlow Lite object detection
3. Google Maps integration

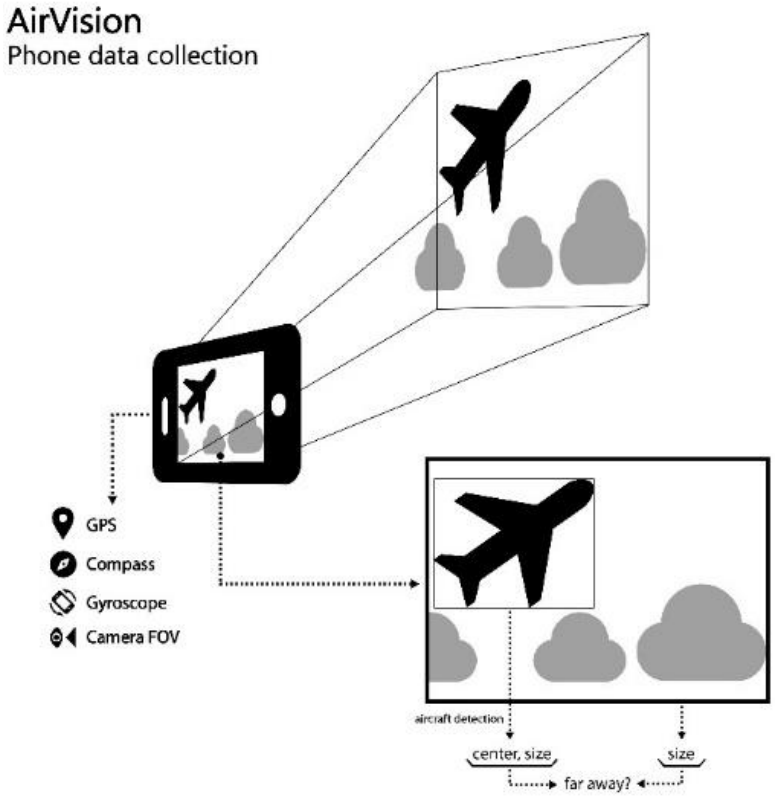


To retrieve the airplane's data, we will need to send some necessary information to the REST API:

1. The position of the device
2. The rotation of the device
3. The FOV (field of view) of the camera lens
4. The position and size of the aircraft(s) on the image captured by the camera (x, y, width, height)

2.1.1 Flutter

Flutter is Google's UI Toolkit that makes creating modern and interesting looking interfaces very easily compared to native Android or IOS. Flutter makes it possible to create cross-platform apps that will run on both Android, IOS and in the future even on Desktop. Because Flutter is open-source, there is a huge community behind it that makes Flutter improve everyday with the use of custom flutter packages.



2.1.2 TensorFlow Lite object detection

TensorFlow Lite is an open source deep learning framework for on-mobile device inference. We will be using this framework to accomplish aircraft detection on the mobile device.

The model's output will return an array of detected objects, in our case aircraft. The returned information will look something like the following:

Class	Confidence	Location
Airplane	0.90	[18,21,57,63]
Airplane	0.88	[100,20,180,140]
Airplane	0.32	[7,82,83,103]

The model returns a class, the confidence score, and the positions on screen in following order [TOP, LEFT, BOTTOM, RIGHT].

The position on screen will be used to calculate the airplane's center point on the viewport of the camera and this information along with the size of the detected area will be send to the backend to determine which aircraft is being seen by the camera.

2.1.2.1 Methods

To handle the object detection of this mobile application, we had to make a choice of what method we wanted to use: do we want real time object detection, or do we want an image object detection system where the user takes a picture of a plane instead of using the feed of the camera to detect an aircraft?

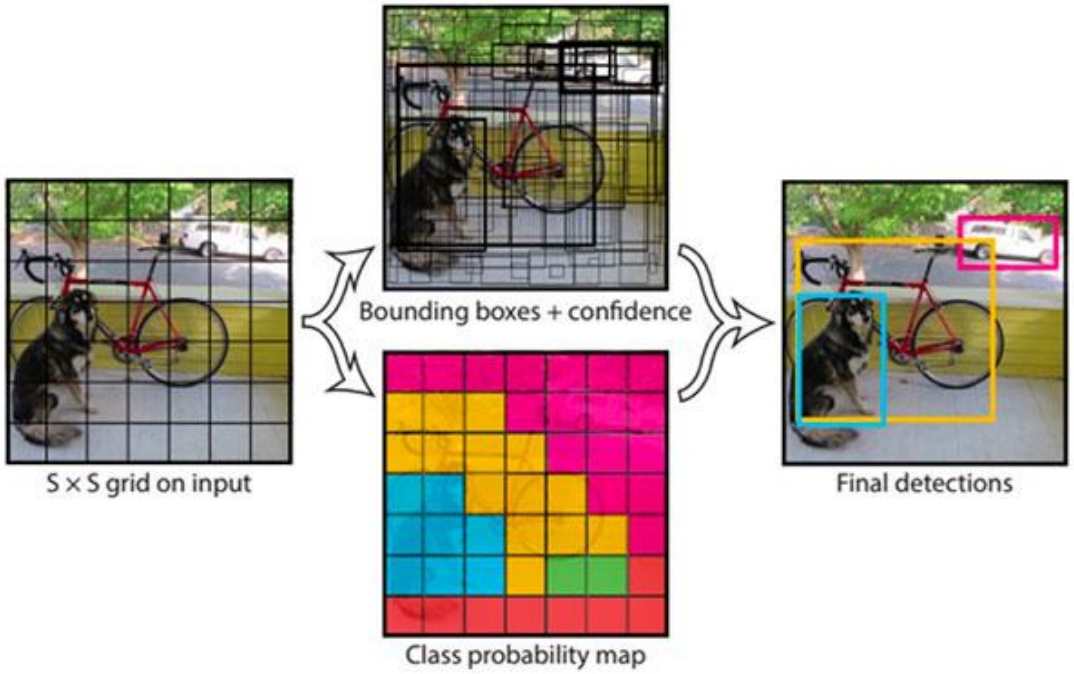
The two following methods YOLO and SSD each represent one side of this question. They are explained below with a simple illustration.

YOLO

YOLO also known as the You Only Look Once algorithm is a straightforward regression dilemma which takes an input image and learns the class possibilities with bounding box coordinates. YOLO divides every image into a grid of S x S and every grid predicts N bounding boxes and confidence. The confidence reflects the precision of the bounding box and whether the bounding box contains an object despite the defined class. YOLO even forecasts the classification score for every box for each class. You can merge both classes to work out the chance of every class being in attendance in a predicted box.

So, total S x S x N boxes are forecasted. On the other hand, most of these boxes have lower confidence scores and if we set a doorstep say 30% confidence, we can get rid of most of them.

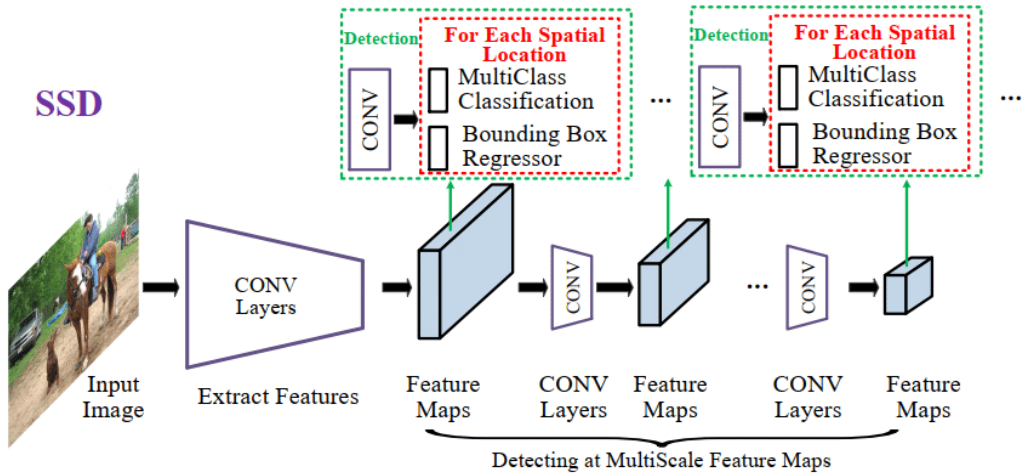
This is the YOLO method visualized:



SSD

The other option is to use Single Shot Detector also known as SSD. In theory this would be a better solution because its more accurate, but because we are looking to use real time object detection, we want the fastest method of working that we have available and that is the YOLO method. SSD is also a lot tougher to run on mobile devices because it demands more computing power than YOLO.

This is the SSD method visualized:



2.1.3 Google maps integration

Google maps will be used to display the current user's position and all aircraft currently in the vicinity of the user. The user will be able to set a max radius of aircraft to detect, this way it is easier to look for aircraft.

When a user taps on an aircraft that is currently being displayed on the map they will get more information about the aircraft, flight, etc.

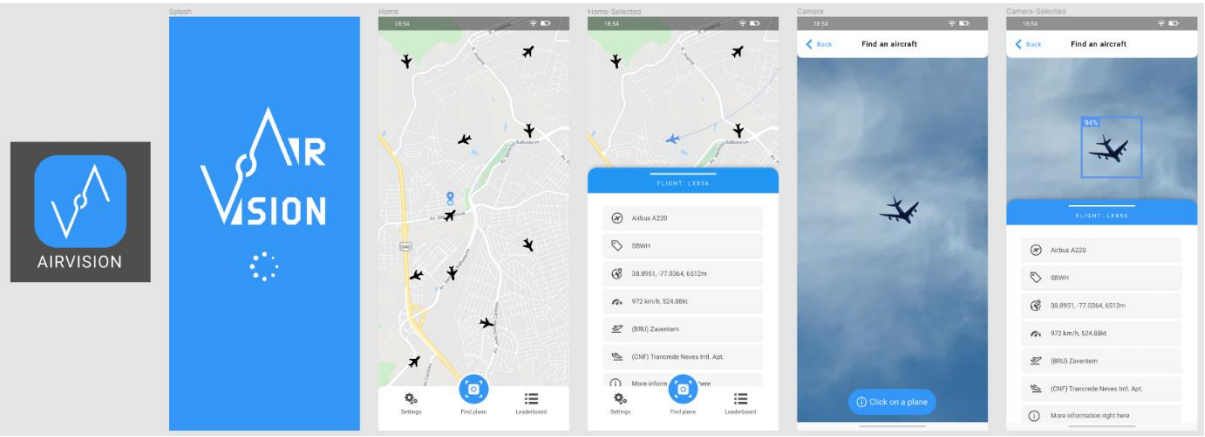
2.2 Mockup

To make the actual development of the app easier and faster, we first created some mockups.

There are 2 designs of the app. The first one we made at the start of the project. We started with a simple mockup so we could take this to our mentor to get a confirmation that the mockup is something similar to the concept that he had in mind.

2.2.1 First version

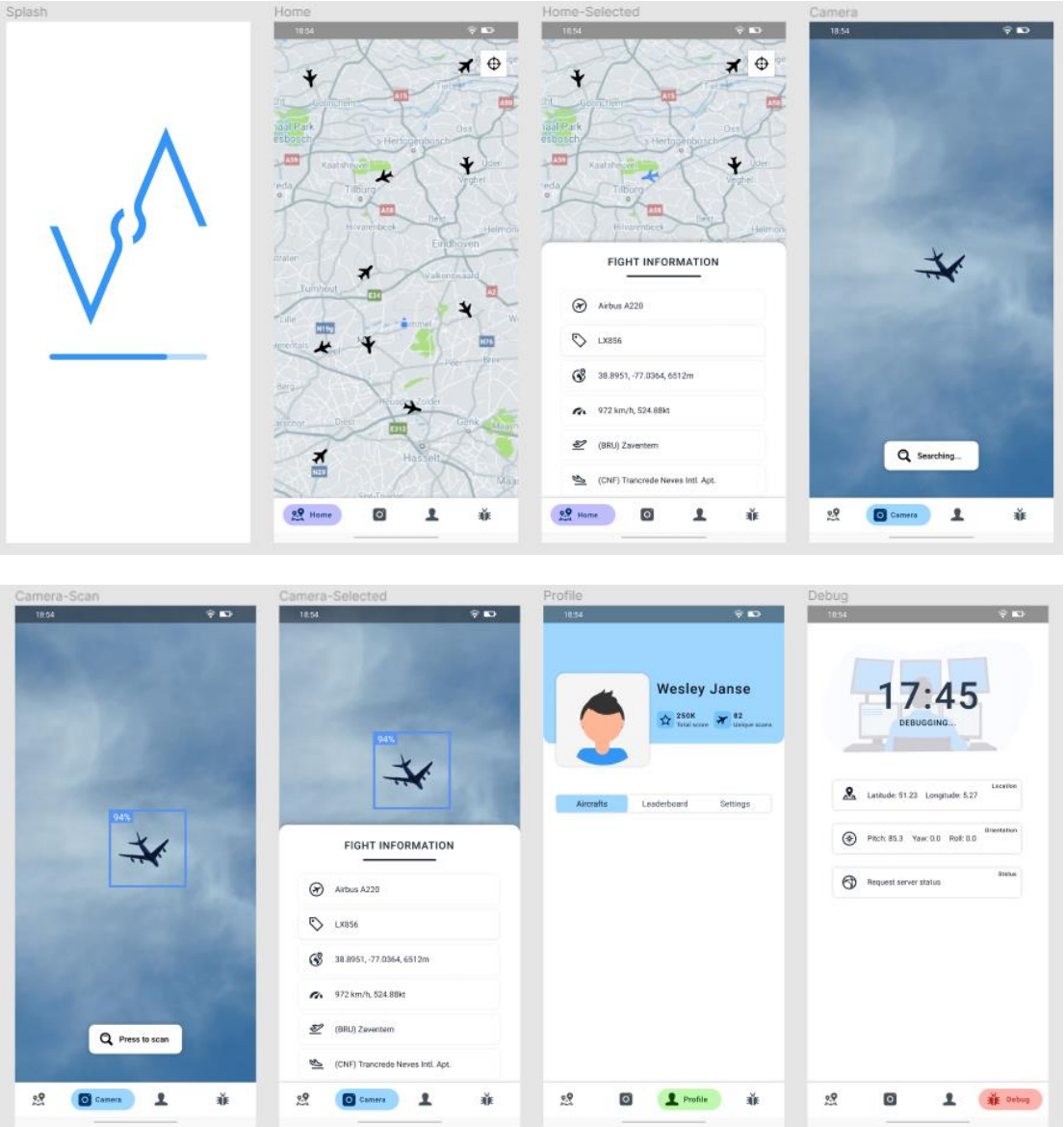
In the initial mockup you can see the first design of how we wanted to implement the map, with a simple bottom navigation bar making it easy for the user to navigate the application. In the center of this navigation bar was a button to access the camera functionality of the app, which you can see on the last two screenshots.



The logo and splash screen of the first mockup remain the same in the final version of the project.

2.2.2 Final version

While making the second mockup, we wanted to change the way the app navigates. In the previous version every button would open a new activity, therefore opening a separate screen. Thus the main focus was changing the bottom navigation to something more modern. Meanwhile we decided the change the style of the map with the purpose of making it look less like a Google Maps copy and therefore turning it into something unique.



Now the navigation works without opening new windows, it just slides the desired window into view.

Another aspect to note is that there is a ‘Profile’ screen in this mockup which later on in the application itself turns into the ‘Settings’ screen. As you will be able to see in the implementation part of this document.

2.3 Implementation

We will now go into more detail surrounding a number of components defined in the analysis

2.3.1 Mobile application

This section describes how the mobile application is implemented. This includes the use of language/framework, build system and dependencies. The application uses object-oriented programming.

2.3.2 Language/framework

The front-end framework used for the application is Flutter which in turn is written in the open-source Dart programming language, which is also used to write the application. Dart is a client-optimized language to create fast and optimized applications on all platforms. The programming language heavily focuses on building and customizing user interfaces on all platforms.

2.3.3 Build System

Gradle is the preferred build system for the Flutter/Dart programming environment. Flutter builds APKs through a long set of steps that ultimately defer to Gradle. The logic lives in many places: The Flutter framework, the Flutter app template, and a mix of both Dart and Gradle scripts.

Here follows a high-level description of the overall flow, which can be hard to follow given how many different places at once it lives and how asynchronous Gradle builds are. (These steps come straight from the Flutter Wiki.)

- The Flutter tool fetches any plugin dependencies of the Flutter app from pub and downloads it to the pub cache. This step follows normal pub get behavior and version solving.
- The Flutter tool writes project metadata to some local temporary files. This includes things like the location of the local Flutter SDK, and paths to all the newly downloaded plugins from pub on disk.
- The Flutter tool invokes Gradle to build the APK.
- Gradle reads the app's build.gradle, which applies the Flutter framework's flutter.gradle plugin.
- flutter.gradle adds the Flutter engine and all of its support library dependencies as a jar dependency to the Flutter app.
- flutter.gradle adds all of the Flutter plugins of an app as "subproject" Gradle dependencies (usually).
- Gradle compiles the Flutter app and all its subprojects (Flutter plugins) into a root APK.

Gradle invokes a lot of these steps at seemingly random and parallel times. That is why these steps are deliberately listed as bullet points and not as a numbered list.

2.3.4 Object detection

Here follows a brief description of what technology was used for object detection.

2.3.4.1 Tensorflow Lite

TensorFlow Lite is a set of tools to help developers run TensorFlow models on mobile, embedded, and IoT devices. It enables on-device machine learning inference with low latency and a small binary size, making it relatively easy to run on mobile-devices.

2.3.4.2 Method

The object detection method used in this application is called the You Only Look Once method also called YOLO.

Please refer to ***2.1.2.1 Methods*** to find more information about the You Only Look Once method.

2.3.4.3 Implementation

A necessary aspect during the implementation is real time object detection, so that every frame that the camera captures needed to be analyzed as fast as possible.

Below a step-by-step explanation of how Tensorflow object-detection is implemented in the app:

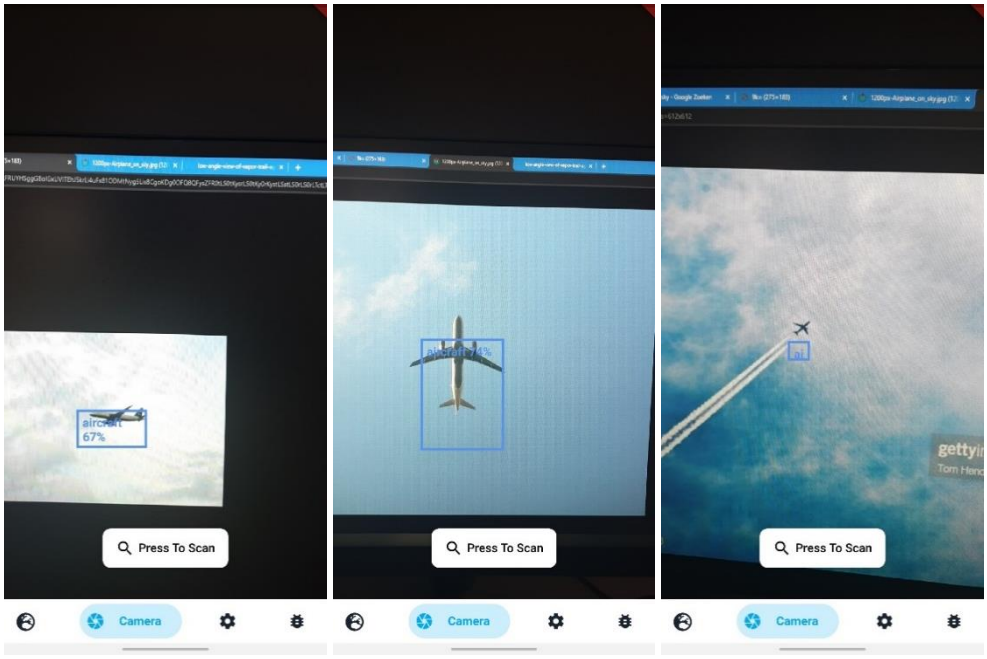
1. Start a stream that sends every frame the camera captures to a function.
2. Use Tensorflow to detect a possible object on a frame.
3. Convert the image into a bytesList so that Tensorflow can work with the data.
4. When an object is found complete the Future and execute a function.
5. Send the data to a function that draws the squares on the screen in real-time.

Here is an example:

```
1 controller.startImageStream((CameraImage img) async{
  if (!isDetecting) {
    isDetecting = true;
   2 Tflite.detectObjectOnFrame(
    3 bytesList: img.planes.map((plane) {
        return plane.bytes;
      }).toList(),
      model: 'Tiny YOLOv2',
      imageHeight: img.height,
      imageWidth: img.width,
      imageMean: 0,
      imageStd: 255.0,
      numResultsPerClass: 1,
      threshold: 0.2,
    ).then((recognitions) {
    4 updateRecognitions(recognitions, img.height, img.width);
      isDetecting = false;
    });
  }
});
```

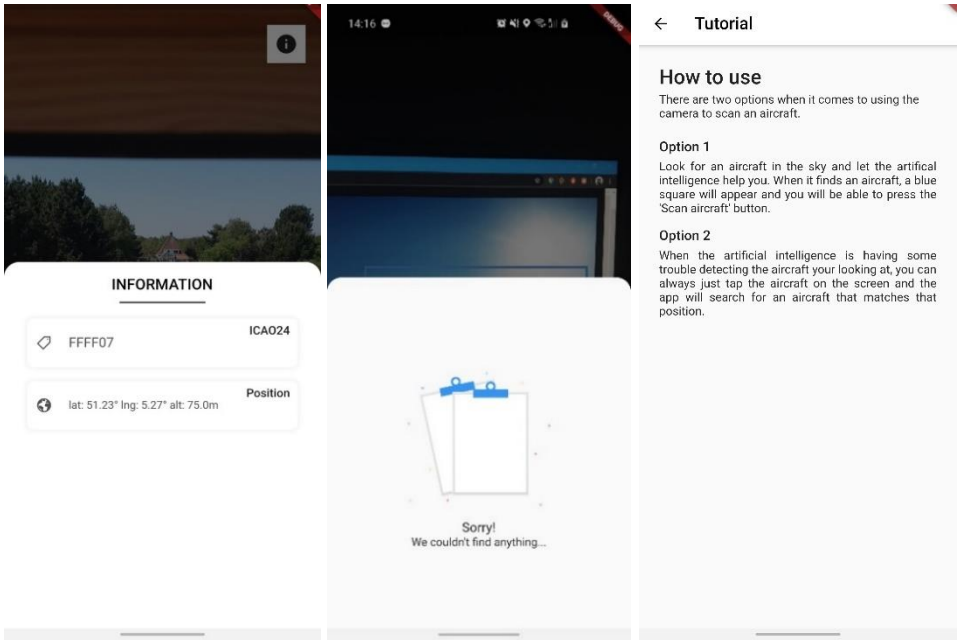
2.3.4.4 Model and confidence rate

For speeding up the development process we are using a pre-trained model from the Tensorflow team that includes aircraft recognition. Because of this the detection accuracy and confidence are not always perfect. You will find some example screenshots below.



Although it is not perfect, it is still good enough to detect aircraft that are close and aircraft that are a bit further in the sky. Making it possible to use our algorithm to search for a matching aircraft and display the information to the user. If the object detection is not able to detect an aircraft, you can also tap the screen on the location of the aircraft.

Some example screenshots of the app showing information after a scan. **Note:** The scanned aircraft in the screenshot is test therefore fake data. When a real aircraft is scanned the information is the same as shown in ***2.3.5.1 Examples.***



2.3.5 Georeferenced app development with google maps

Another big part of the mobile application is using a map which shows you all the aircraft categorized by their weight class by using different icons to represent the different weight classes.

By tapping on an aircraft, the app will show all the data that is available of that specific aircraft.

In order to make it possible to have custom icons which actually act as a marker we had to write a custom function that gets the bytes from a specific asset, since you cannot use an image as a marker.

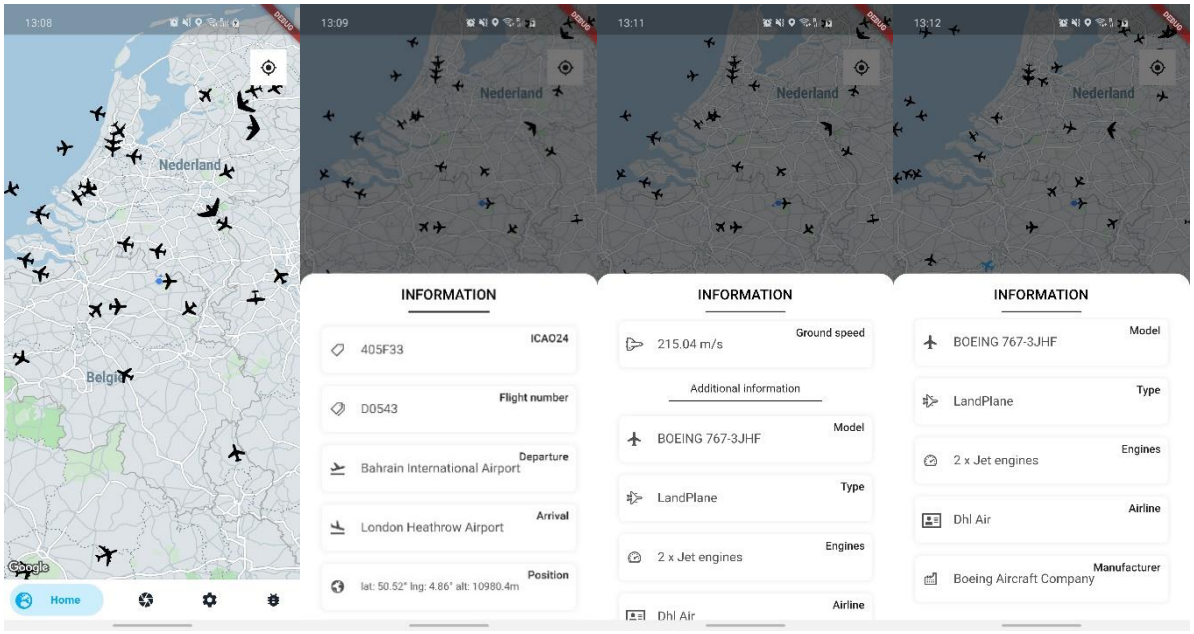
This is the function that makes it all possible, it loads in an image from a specific path and returns a List of integers so that the google maps library can work with the data.

```
Future<Uint8List> getBytesFromAsset(String path, int width) async {
  ByteData data = await rootBundle.load(path);
  ui.Codec codec = await ui.instantiateImageCodec(data.buffer.asUint8List(),
      targetWidth: width);
  ui.FrameInfo fi = await codec.getNextFrame();
  return (await fi.image.toByteData(format: ui.ImageByteFormat.png))
      .buffer
      .asUint8List();
}
```

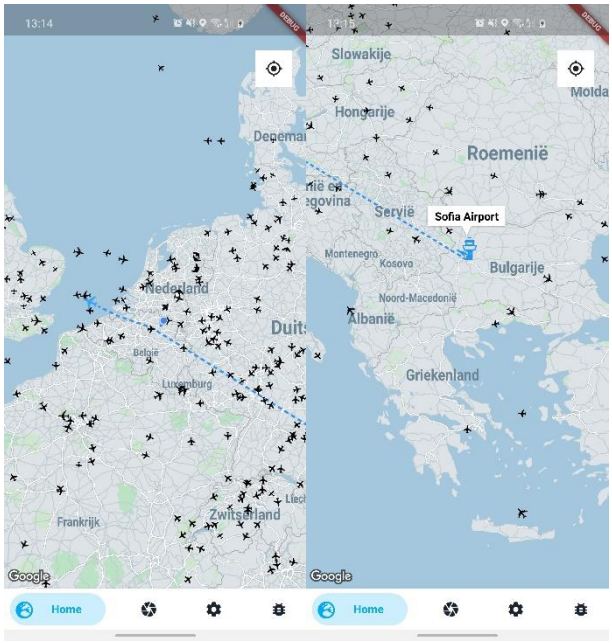
In this case the function returns a Uint8List the Uint8List consumes less memory than a normal List, because it is known from the beginning that each elements size is a single byte.

2.3.5.1 Examples

Some example screenshots can be found below, you can see the different icons and information available to the user. You can find an overview of the available data from the REST API at ***3.2.5.1 Model.***



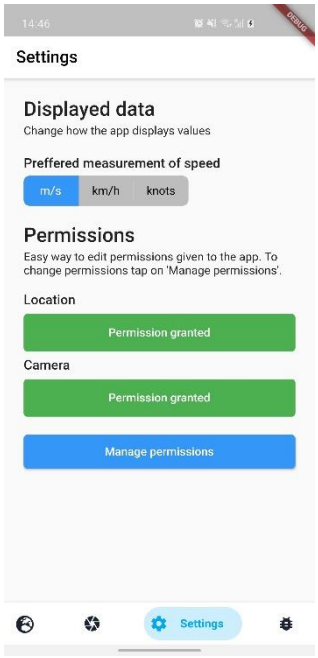
If the flightpath information is available, the app shows the route the aircraft has taken, including the departure airport.



2.3.6 Settings and debugging

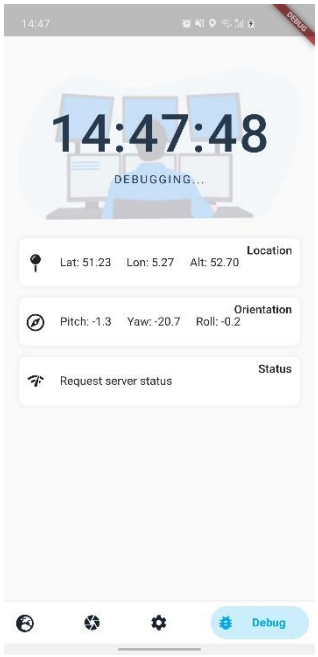
2.3.6.1 Settings

For some extra customizability we created a settings screen where the user can easily change whether they want to use m/s, km/h or knots as their preferred measurement of speed. This screen also functions as a way to manage the permissions given to the app.



2.3.6.2 Debugging

To make development easier for us we decided to make a debugging screen showing us the sensors working in real-time in order to determine how they actually work.



2.3.7 Device sensors

An important aspect of making sure the aircraft detection actually works is sending it the right sensor data. In order to accomplish this we also had to dig into the native android code to invoke a method that gets the required sensor information.

In order to invoke a native method you have to setup a methodChannel which you can use to invoke these methods from the Dart code itself.

This is an example of a methodChannel:

```
new MethodChannel(flutterEngine.getDartExecutor().getBinaryMessenger(),
ORIENTATION_CHANNEL)
        .setMethodCallHandler(this::handleOrientationChannel);
```

After that you need to setup a handler that will handle the method calls and activate the right method.

For example:

```
private void handleOrientationChannel(
    MethodCall call, MethodChannel.Result result) {
  switch (call.method) {
    case "getQuaternion":
      getOrientationQuaternion(result);
      break;
    case "getEstimatedAccuracy":
      getEstimatedAccuracy(result);
      break;
    case "start":
      startOrientationService(result);
      break;
    case "stop":
      stopOrientationService(result);
      break;
    default:
      result.notImplemented();
      break;
  }
}
```

2.3.7.1 Orientation

We need to know where the device is facing, we accomplish this by using an orientation service that tracks the device's orientation. When the sensor detects that it has changed, it will store that rotation data into a quaternion and when it is time to send a request to the API the app can simple request the latest quaternion data and send that information to the server.

Please refer to ***3.3.1.3 Quaternion*** to get more information about quaternions.

2.3.7.2 Location

Retrieving the location data is a lot simpler than retrieving the rotation data. There are numerous packages that make it easy to do this. Simply setup a location Listener that looks whether the location had changed then store that data into a variable consisting of a Geodetic Position and update the state.

Please refer to ***4.1.2.1 Geodetic position*** to get more information about Geodetic Positions.

2.3.7.3 Camera FOV

To setup the virtual camera to detect which aircraft the mobile device is looking at it will need the camera's FOV (field of view). The method below gets the FOV from the camera:

```
private void getCameraFov(MethodCall call, MethodChannel.Result result) {
  String name = call.arguments();

  try {
   CameraCharacteristics info = this.cameraManager.getCameraCharacteristics(name);
   SizeF sensorSize = info.get(CameraCharacteristics.SENSOR_INFO_PHYSICAL_SIZE);

   float[] focalLengths =
        info.get(CameraCharacteristics.LENS_INFO_AVAILABLE_FOCAL_LENGTHS);

   if (sensorSize == null || focalLengths == null)
     throw new IllegalStateException("Essential camera info isn't available.");

     float focalLength = focalLengths[0];

     // Short edge of the screen
     double x = Math.min(sensorSize.getHeight(), sensorSize.getWidth());
     // Longest edge of the screen
     double y = Math.max(sensorSize.getHeight(), sensorSize.getWidth());

     double[] fov = new double[2];
     fov[0] = Math.toDegrees(2 * Math.atan(x / (2 * focalLength)));
     fov[1] = Math.toDegrees(2 * Math.atan(y / (2 * focalLength)));

     result.success(fov);
  } catch (CameraAccessException e) {
     throw throwUnchecked(e);
  }
}
```

2.3.8 REST API Communication

In this section you will find more information about how the mobile application communicates with the REST Web Service.

Please refer to ***3.1.1 REST Web Service*** for more information about the REST Web Service itself*.*

2.3.8.1 Models

To get an overview of all the models used in the front-end of this application you can refer to the database model ***2.3.8.1 Models.***

2.3.8.2 Making a request

One problem that we ran into with flutter and its HTTP package was that it does not allow GET requests with a body. To counter this issue, we had a few options, we could use a POST request that returns the data, or we can use queryParameters. Ultimately, we decided to go with the POST request method. But it is also possible to use queryParameters if we ever ran into problems using POST.

Here is an example request:

```
Future<AircraftState> getPositionalData(String icao24) async {
  Map<String, String> headers = {'Content-Type': 'application/json'};

  String body = '''{
    "icao24": "$icao24"
  }''';

  http.Response responseData = await http.post(
    baseURL + '/api/v1/aircraft/state/get',
    body: body,
    headers: headers,
  );

  if (responseData.statusCode == 200) {
    var tagObjsJson = jsonDecode(responseData.body)["data"];
    AircraftState aircrafts = AircraftState.fromJson(tagObjsJson);
    return aircrafts;
  } else {
    return Future.error(jsonDecode(responseData.body)['error']['message']);
  }
}
```

As you can see, when the request is successful the responseData is decoded using JSON and then fed into the Factory method of the targeted class. Please refer to ***2.3.8.3 Request conversion to object.***

2.3.8.3 Request conversion to object

To make sure the JSON data from the REST API can be used in the app the data needs to be casted into an object. There is no straightforward way of doing this in Flutter, so to solve this you need to write a Factory method that instantiates an object from the JSON data.

Here is an example of a class with a Factory method:

```
class AircraftInfo{
    AircraftInfo(this.icao24,this.model, this.owner, this.type,
this.weightCategory, this.manufacturer, this.engines);

  String icao24;
  String model;
  String owner;
  String type;
  String weightCategory;
  Manufacturer manufacturer;
  Engines engines;

  factory AircraftInfo.fromJson(dynamic json) {
    return AircraftInfo(
      json['icao24'] as String,
      json['model'] as String,
      json['owner'] as String,
      json['type'] as String,
      json['weightCategory'] as String,
      Manufacturer.fromJson(json['manufacturer']),
      Engines.fromJson(json['engines']),
    );
  }
}
```

As you can see all the JSON data is casted into the datatype it should be and then created in the constructor of the class.

2.3.9 Dependencies

Here follows a list of the used dependencies in the mobile application.

Flutter SDK

SDK used to build Flutter applications.

Material:dart

Flutter widgets implementing Material Design, used for fast and optimized UI/UX creation.

Dart:math

Mathematical constants and functions, plus a random number generator.

Dart:async

Support for asynchronous programming, with classes such as Future and Stream.

Dart:typed_data

Lists that efficiently handle fixed sized data (for example, unsigned 8 byte integers) and SIMD numeric types.

Dart:ui

This library exposes the lowest-level services that Flutter frameworks use to bootstrap applications, such as classes for driving the input, graphics text, layout, and rendering subsystems.

Dart:convert

Encoders and decoders for converting between different data representations, including JSON and UTF-8.

In addition to converters for common data representations, this library provides support for implementing converters in a way which makes them easy to chain and to use with streams.

Sensors

A Flutter plugin to access the accelerometer and gyroscope sensors.

Camera

A Flutter plugin for iOS and Android allowing access to the device cameras.

Location

This plugin for Flutter handles getting location on Android and iOS. It also provides callbacks when location is changed.

Google_maps_flutter

A Flutter plugin that provides a Google Maps widget.

Tflite

A Flutter plugin for accessing TensorFlow Lite API. Supports image classification, object detection (SSD and YOLO), Pix2Pix and Deeplab and PoseNet on both iOS and Android.

Http

A composable, Future-based library for making HTTP requests.

This package contains a set of high-level functions and classes that make it easy to consume HTTP resources. It is platform-independent and can be used on both the command-line and the browser.

Permission_handler

This plugin provides a cross-platform (iOS, Android) API to request permissions and check their status.

App_settings

A Flutter plugin for opening iOS and Android phone settings from an app.

Vector_math

A Vector math library for 2D and 3D applications.

Flutter_spinkit

A collection of loading indicators animated with flutter.

Font_awesome_flutter

The Font Awesome Icon pack available as set of Flutter Icons.

Line_icons

Icon library based on Awesome Line Icons.

Cupertino_icons

This is an asset repo containing the default set of icon assets used by Flutter's Cupertino widgets.

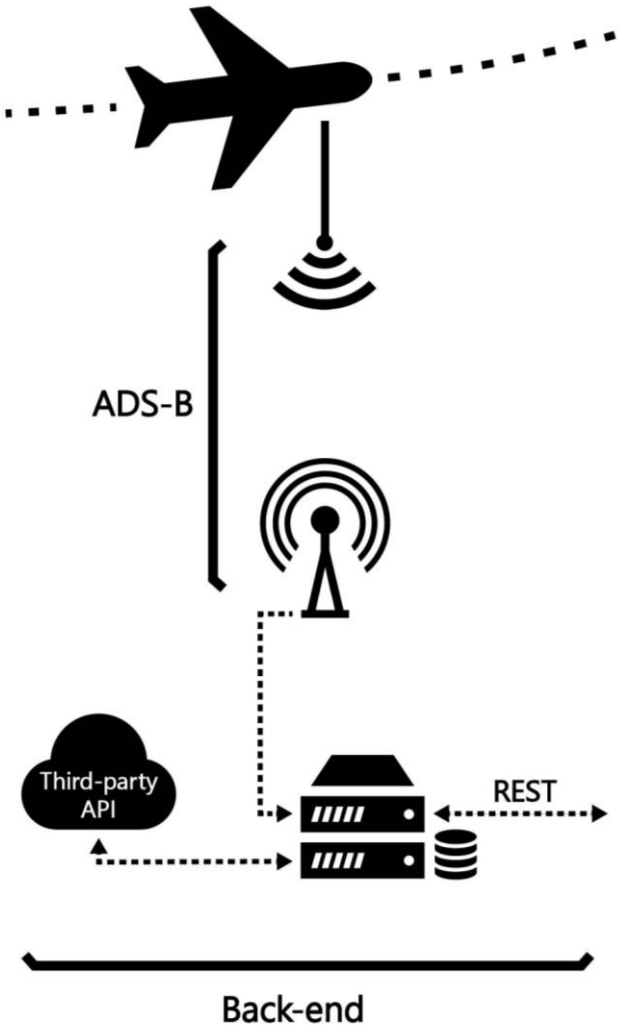
A lot of these dependencies came from *pub.dev* which is a website where flutter developers can upload their custom-made packages so that other developers can use them.

3 Back-end

The back-end will provide all the information to the front-end through a REST API. The back-end will collect necessary information from multiple sources. This can range from third-party API's to the ADS-B ground station which will collect broadcasted packets. The first section of this chapter will analyze the requirements of the back-end, which algorithms will be used, which data is required to use implement the algorithms. The specification of the REST API falls also under this section, but has its own chapter ***4.1 REST Specification***.

The back-end consists of a few key components (the image below visualizes this):

1. REST Web Service
2. Database
3. ADS-B ground station
4. Third-party API's



3.1 Analysis

The server consists of a REST Web Service, database, and a decoder to process messages received from the ADS-B ground station.

3.1.1 REST Web Service

The REST Web Service will be used for the communication between the front-end and the back-end. It is essential that there is a clear specification which will represent the data, which can be found in ***4.1 REST Specification.*** The REST API will mainly be used to retrieve information from the back-end. No data from the front-end is currently stored in the back-end.

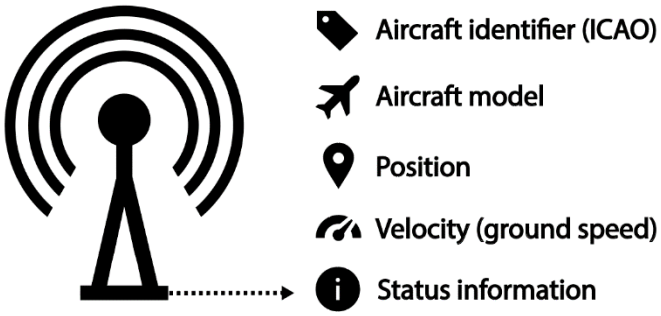
3.1.2 Database

The database will in first instance be used to store the data received from the ADS-B ground station and from other sources like the third-party API's.

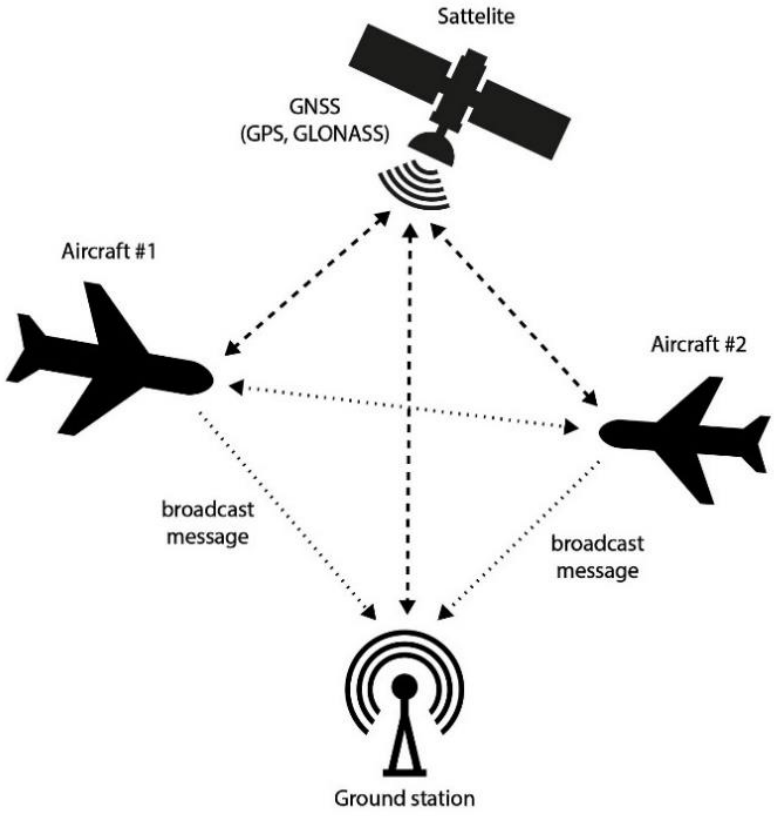
The used database is an SQL database, the structure of all the data that is used will be fixed. The ADS-B has a specification of which data is contained; it is possible that the structure may change in the future because of new additions.

3.1.3 ADS-B ground station

ADS-B, short for Automatic Dependent Surveillance – Broadcast is used by aircraft to share information about where they are located, at what speed they are moving, the status of the aircraft, emergency statuses, etc. Using a ground station, it is possible to read out these broadcasted messages. These messages can be used in combination with the data retrieved from third party API's. The advantage of using the ground station is that we will be able to control the data and there will not be a limit in throughput and the latest version of the data. The range of the ADS-B can cover up to 370 km.



Below is an image of a complete ADS-B system, not just the ground station that the software will focus on. Aircraft will also communicate with each other and communicate with satellites to determine their positions.



3.1.4 Aircraft matching algorithm

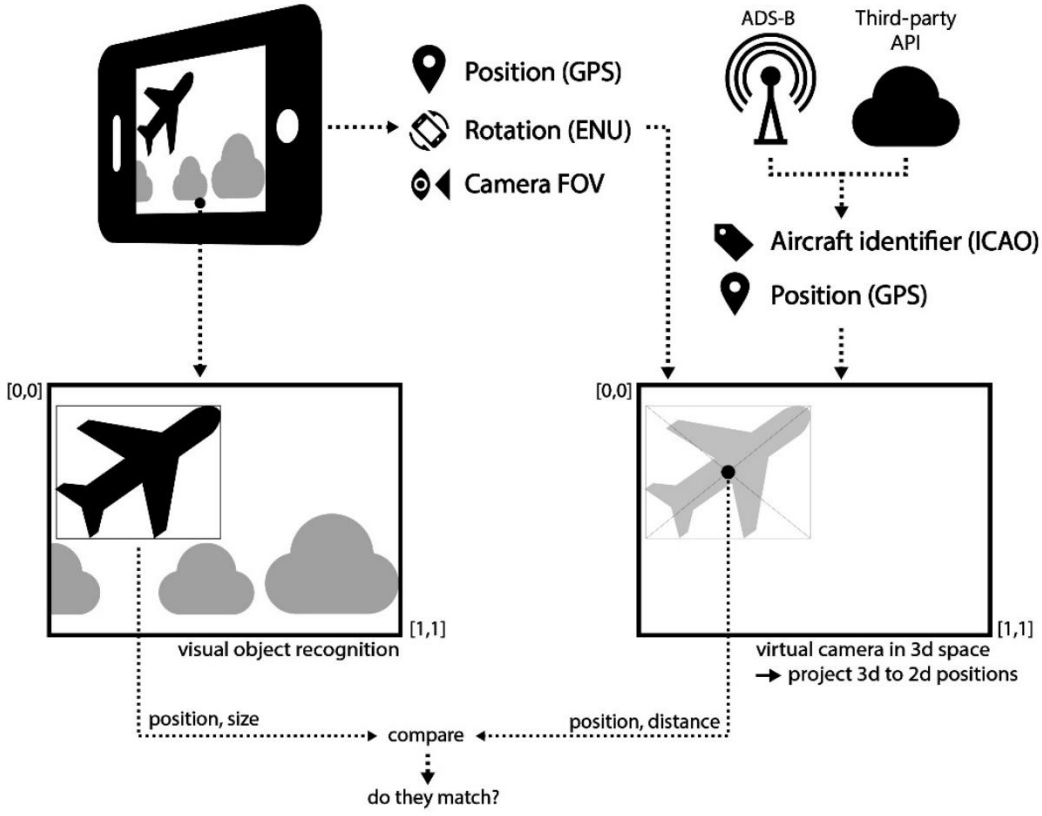
3.1.4.1 The problem

The back end receives information from the front end about the aircraft that is seen by the physical camera. This information needs to be used to figure out which aircraft this is, e.g. identifier, model, position, etc.

To do this, the physical camera data (2D) must be compared to the 3D world where the aircraft are located, and the position is known.

A solution for this is recreating the physical camera as a virtual one which is possible with the position, rotation and FOV. The information from the virtual camera can then be used to compare to the physical one, to see if they match.

The following diagram gives an idea of the steps. On the left side is the image captured by the physical camera, from this image aircraft are analyzed using image recognition to find their positions. On the right side is the virtual camera, which is located at the same position as the physical one but is aware of all the aircraft in the 3D space. The aircraft in the 3D space need to be mapped to a 2D view. Once we have two 2D views, we can use these to compare to each other. The only difference in data is that the image recognition will provide the size of the aircraft, from the virtual camera we just have a single point but the distance between virtual camera and aircraft can be used in this case.



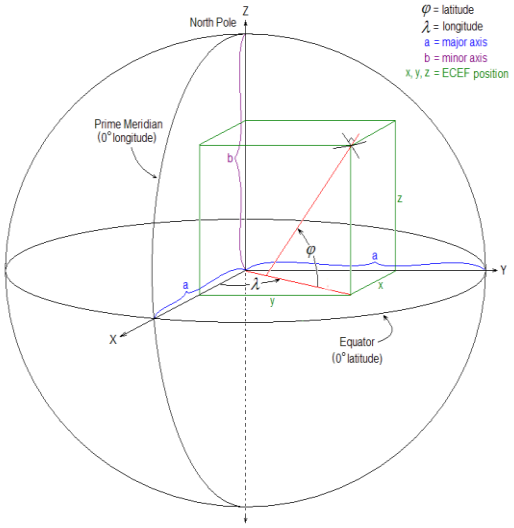
3.1.4.2 Coordinate systems

Another problem is that there are multiple coordinate systems, to make calculations, everything needs to be in the same coordinate system. This also needs to be a cartesian coordinate system because the virtual camera will only use this coordinate system. The ECEF system is the earth-centered, earth-fixed cartesian coordinate system which will be used for this.

Geodetic to ECEF

The geodetic coordinates (latitude, longitude, altitude) provided by the GPS in the aircraft and mobile device will need to be converted to the ECEF coordinate system.

The following image shows the relation between the two coordinate systems.



The following formulas will be used to convert (latitude ϕ, longitude λ, height h (altitude)) to the ECEF system:

$$X = (N(\phi) + h)\cos\phi\cos\lambda$$
$$Y = (N(\phi) + h)\cos\phi\sin\lambda$$
$$Z = \left(\frac{b^2}{a^2}N(\phi) + h\right)\sin\phi$$

where

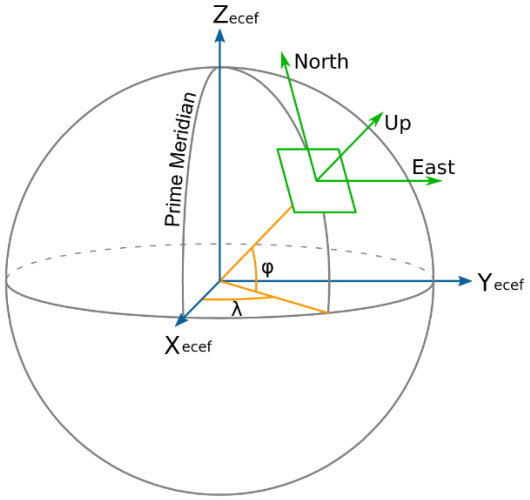
$$N(\phi) = \frac{a^2}{\sqrt{a^2\cos^2\phi + b^2\sin^2\phi}}$$

The *a* and *b* are respectively the equatorial radius (semi-major axis) and the polar radius (semi-minor axis) of the Earth.

a = 6 378,1370 km
b = 6 356,7523 km

ENU to ECEF

The rotation provided by the mobile device is also in a different coordinate system, so it will also have to be converted to the ECEF coordinate system. The system used by the mobile device is the LTP (local tangent plane), which is based on the local vertical direction. The following image shows the positioning of the system, which depends on the coordinates you are at.



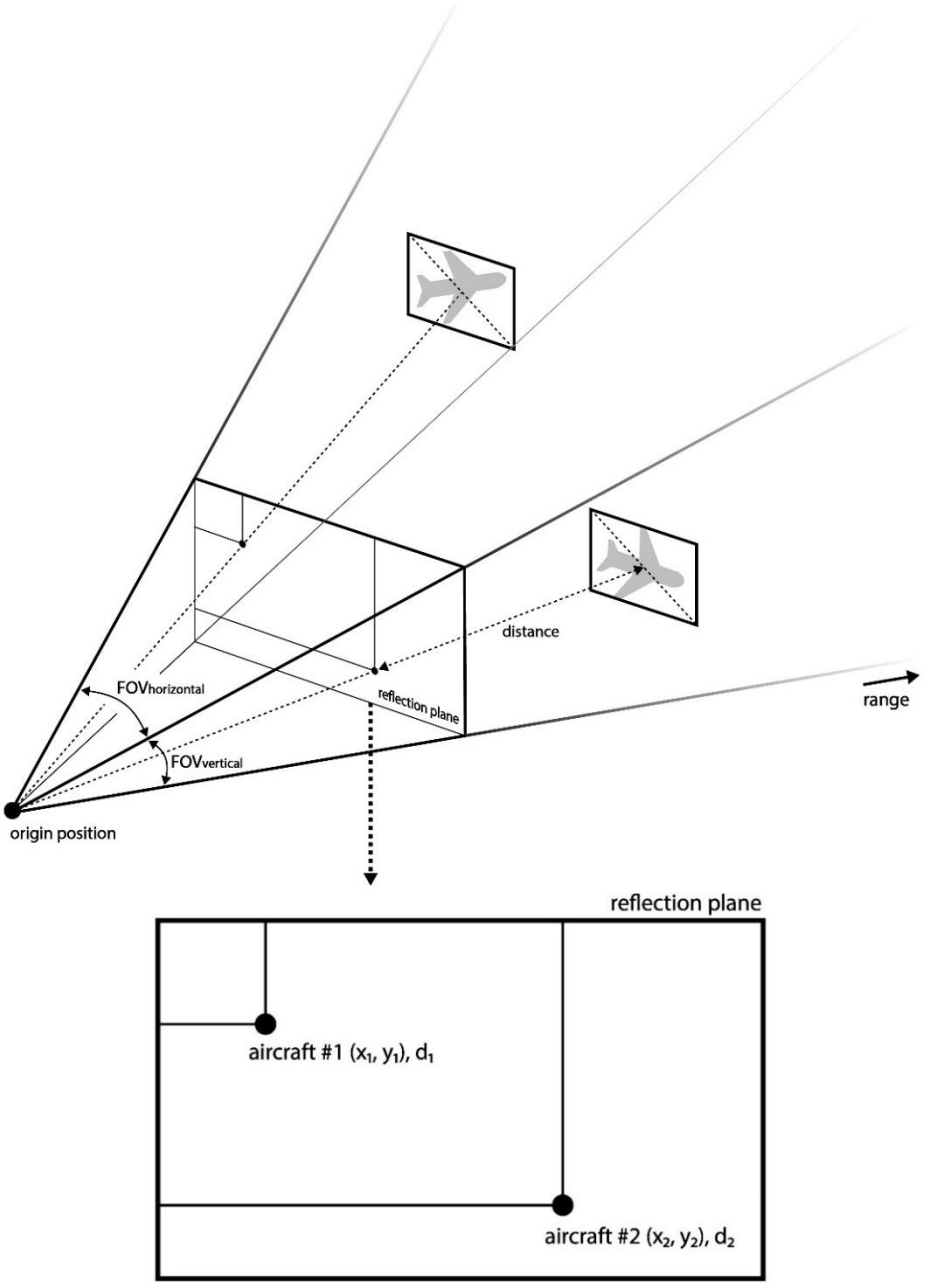
There are two variants in which these coordinates can be described, ENU (east, north, up) and NED (north, east, down) coordinates, the one used by our algorithm is ENU.

The following formula will be used to convert a vector in the ENU coordinate system (z, y, z) to the ECEF system (X, Y, Z). The matrix in the formula is a rotation matrix.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} -\sin\lambda & -\sin\phi\cos\lambda & \cos\phi\cos\lambda \\ \cos\lambda & -\sin\phi\sin\lambda & \cos\phi\sin\lambda \\ 0 & \cos\phi & \sin\phi \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

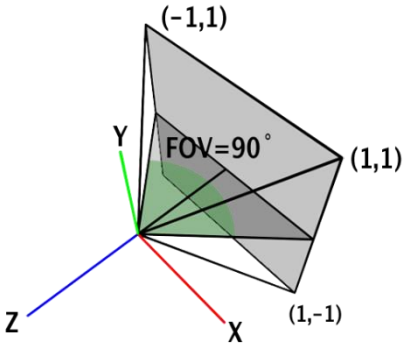
3.1.4.3 Convert ECEF to 2D camera positions

The following image gives an idea of how the virtual camera will work. Aircraft within a predefined radius will be taken into consideration. If they are within the view of the virtual camera, they will be reflected on a reflection plane. Which is the plane where the 3D positions are converted to 2D positions. Another name for this kind of camera is a pinhole camera. The camera can have any kind of rotation within the ECEF system.



Converting 3D ECEF coordinates to the 2D coordinates on the reflection plane is done in two steps.

The first one is to transform the ECEF coordinates to coordinates relative to the camera coordinate system. In this coordinate system every point with +Z will be behind the camera view. The X-axis is left or right and the Y-axis is up or down. The figure on the right shows this visually.



Once the coordinates are within the camera coordinate system, they can be projected on a plane. The figure shows a plane with the bounds (-1, 1) to (1, -1). If the coordinates fall outside of the plane or if the point is behind the camera view, the point will not be visible to the camera, so it will be ignored. More specifically the aircraft will not be visible to the virtual camera.

All the valid points will afterwards be mapped to (0, 0) (top left) to (1, 1) (bottom right) to match an image coordinate system, these coordinates will be compared to the ones send by the mobile phone.

3.2 Implementation

This section describes how the back-end server is implemented, this includes the use of language, build system, dependencies, integration with ADS-B, etc. The application heavily uses concurrency through coroutines. Because of the high use of concurrency, this will be combined with functional programming.

3.2.1 Programming language

The programming language used for the back-end is Kotlin, targeting the JVM (Java Virtual Machine). Kotlin is a cross-platform, statically typed, general-purpose programming language. It is designed to interoperate fully with Java, this means that the software can also make use of Java code and libraries.

3.2.2 Build System

For the Kotlin programming environment, Gradle is the preferred build system. This also a very popular option for Java programming. Gradle will handle all the dependencies that are needed to run the application, if you have Java installed. There is also very good integration with IDE's, this makes it that you can basically clone the project, load it and you are almost good to go.

3.2.3 Programming patterns

There are some concepts that are heavily used in the development of the software. The first one is asynchonous programming, starting from the REST web service to the communication with each individual third-party API. Due the nature of asynchonous programming, most data structures are immutable. This means that each time an immutable object needs to be modified, a new one will be created. This allows the objects to be safely passed around.

Functional programming is also heavily used in the software, which works nicely with the immutable objects and asynchonous programming.

3.2.4 Database

The database of choice is PostgreSQL, it is a free and open source relational database system. Some of its specific language features will be used by the back-end server. Due the nature of the database framework that will be used, named Exposed; it would not be a lot of work to switch to different database system.

The exposed library very mature but was missing some features. So, some extensions needed to be written. The following feature were added:

- Abs (absolute) operator
- Transformation between data types was added so it is easy to define columns with custom types. See the data types table in the database model.
- Upsert operator, it is an update operator and if the PK does not exist it will be added to the table.
- Distinct by, known as DISTINCT ON in Postgres. It allows you to get distinct entries for a specific column, instead of all the columns when using the normal DISTINCT.
  This feature is very language specific, so currently only Postgres is supported by the software because of this feature.

3.2.5 ADS-B integration

ADS-B packets are sent with a 1090Mhz radio frequency, so the information can be received using an antenna, using an USB port to interface with the server.

ADS-B packets that are received through the USB port would be decoded according to the ADS-B packet specification; this will not be covered here but a link can be found below. There is a Java library which can be used to decode the packets, so this does not need be reimplemented (see ***LibADSB***, in 3.2.4 Dependencies).

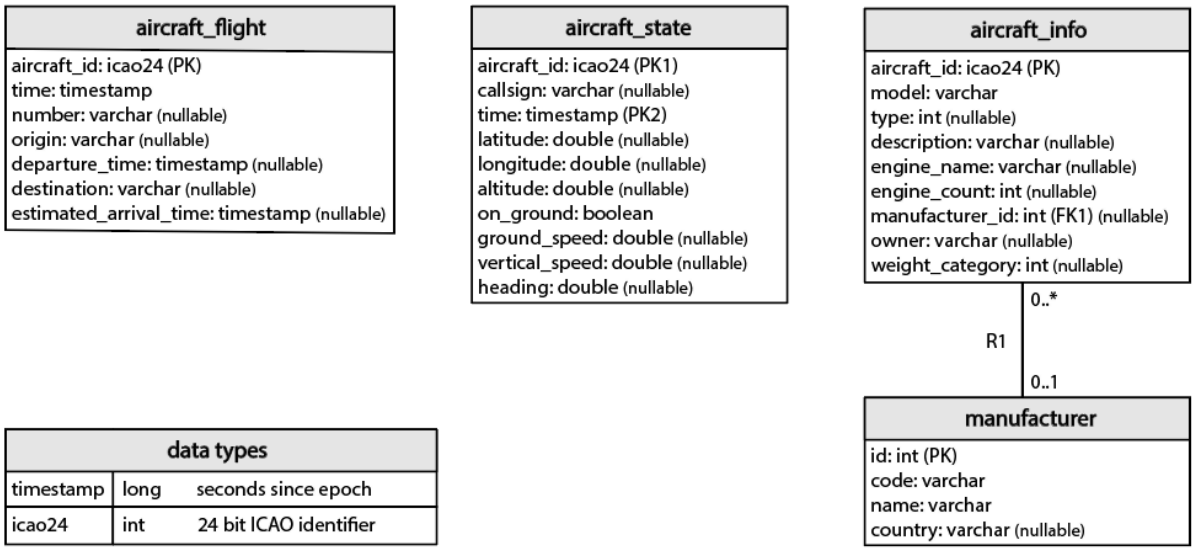
The ADS-B ground station was never installed because of a limitation caused by the corona crisis. This limitation does however not block the development of the application since third-party API's can provide enough information to accommodate for this. However, not having our own ground station binds us to the use of other services, this includes their limitations.

ADS-B specification: https://mode-s.org/decode/

3.2.5.1 Model

The database model is quite simple, mainly because the REST API does not require a more complex one. This is also not the focus of the project, although it would be possible to reduce the amount of duplicate data stored.

The following model gives an idea of which tables are used by the software. Also note the data types, these are somewhat “custom” data types which are defined with Exposed and used by multiple tables.



***aircraft*_flight**

Is the current flight an aircraft is executing if it is known. Only recent data will be kept in the database.

***aircraft*_state**

Is the state of an aircraft at a specific timestamp. Only recent data will be kept in the database.

A lot of fields are nullable, because the nature of ADS-B packets, which provide only a small piece of data each time. Merging data helps with this, but there are no guarantees.

***aircraft*_info**

Is various information about an aircraft and manufacturer, if it is known.

3.2.6 Dependencies

Kotlin Coroutines

Kotlin library for asynchronous programming using the coroutine design pattern.

Kotlin Serialization

Kotlin library for serialization, will be used to transform Objects to JSON and vice-versa.

Ktor

Kotlin library for networking, this library is used for the REST API as well as for the communication with the third-party API's. This library also utilizes Kotlin coroutines and serialization. Ktor also depends on some other libraries like Netty and Apache Http Client.

Netty

A non-blocking, I/O client-server framework, in this project it will not be used directly. Ktor will use this library under the hood to create the server endpoint.

Apache Http Client

Java library that will handle http requests, in this project it will not be used directly. Ktor will use this library under the hood to create http clients.

Arrow

Kotlin library for functional programming.

Csv Parser

Kotlin library to read CSV files, some third-party endpoints provide CSV files.

Guava

Java library with common utilities which can be used across a lot of projects.

Exposed

Kotlin library for database management. Using this library and some extra extensions there is no need to manually write SQL statements, this will all be handled by this library.

HikariCP

A (JDBC, Java Database Connectivity) database connection pool, this will prevent the constant recreation of database connection, but rather reuse connections as needed.

PostgreSQL JDBC Driver

A (JDBC, Java Database Connectivity) database driver for PostgreSQL.

Caffeine

A high-performance caching library for Java.

Log4j2

A logging framework for Java, which will handle all messages that are logged.

Disruptor

A high-performance inter-thread messaging library used by log4j2 for async loggers.

Math

A math library which has representations of vectors, quaternions, matrices, etc.

LibADSB

An ADSB message decoder, is used to decode raw data that is received by the ADSB ground station.

jSerialComm

A java library for serial communication, is used to read data from or write data to serial ports. This is used for communication with the ADSB ground station.

JUnit Jupiter Engine

A java unit testing engine.

3.2.7 Third-party API's

During the implementation there are multiple sources of data were used. This allows a check whether data is reliable and it allows gap within the data to be filled when certain data is absent from a certain source. Using multiple sources for the same piece of information also provides redundancy to a certain degree. This section will explain which sources are used and how the data is processed or requested.

3.2.7.1 OpenSky Network

The OpenSky Network is a non-profit community-based receiver network that collects air traffic surveillance data. OpenSky keep the complete unfiltered raw data and makes it accessible to academic and institutional researchers.

The REST API is the first things that is being from this source, more specifically getting all the active aircrafts and their states (position, identifier, ground speed, etc.). However, this source is not always very stable, so only relying on this service for state data is not viable. This is where Flightradar24 will come into play.

The second thing that is being used is aircraft registration and manufacturer datasets, which are hosted on a file server. These are gigantic CSV files that will be parsed and loaded into a database. AirVision will automatically check for updates, if they are available, they will be downloaded, and the database will be updated. See ***3.2.8 Downloading updates from a file server***.

Website:

https://opensky-network.org

Rest API specification:

https://opensky-network.org/apidoc/rest.html

Aircraft dataset

Url:

https://opensky-network.org/datasets/metadata/aircraftDatabase.csv

Fields

The following table explains the fields that were used to populate the database. There is a lot of missing data in the dataset, but sometimes information can be extracted from multiple fields, not every field is useful or reliable.

NAME	VALUE
icao24	ICAO aircraft registration number
registration	registration code, if known
manufacturericao	manufacturer ICAO code, if known
manufacturername	manufacturer name, if known
model	aircraft model
typecode	aircraft type code, is tied to the model
serialnumber	serial number
linenumber	not useful, is almost always empty
icaoaircrafttype	the aircraft description, if known, see below
operator	the operator fields are not useful, they are mostly empty, it is better
operatorcallsign	to rely on the *owner* field
operatoricao	
operatoriata	
owner	name of the owner or *Private*, if known
testreg	not useful, is almost always empty
registered	the date when the aircraft was registered, if known
reguntil	the date until the aircraft is registered, if known
status	not useful, is always empty
built	not useful, should be the build date, but they make no sense
firstflightdate	not useful, is always empty
seatconfiguration	not useful, is always empty
engines	engine series, if known
modes	not reliable, is always FALSE
adsb	not reliable, is always FALSE
acars	not reliable, is always FALSE
notes	not useful, is always empty
categoryDescription	contains sometimes info about the weight category, or if it is glider

Aircraft description

Some examples of aircraft descriptions are L1P/S, H2T, L2J, etc. The following tables explain what each character in the description represents.

Aircraft type

The first character represents the type.

CODE	TYPE
L	Land based
S	Seaplane
A	Amphibian
H	Helicopter
D	Dirigible, also known as Airship
G	Glider

Number of engines

The second character represents the number of engines that are present in the aircraft.

Types of engines

The third character represents the type of engine.

CODE	TYPE
P	Piston
T	Turboprop
J	Turbo-Jet

Weight class

The weight class is represented as a suffix, although it is not used in this dataset. The weight class will be able to be extracted from the ***categoryDescription***.

CODE	TYPE
/S	Small – U.S. designated aircraft of less than 12 500 lbs. (5670 kg)
/L	Large – U.S. designated aircraft of more than 12 500 lbs. (5670 kg) but less than 300 000 lbs. (13608 kg)
/H	Heavy – Aircraft of 300 000 lbs. (13608 kg) or more

Manufacturer dataset

Url:

https://opensky-network.org/datasets/metadata/doc8643Manufacturers.csv

Fields

The following fields are present in the manufacturer's dataset.

NAME	VALUE
Code	manufacturer ICAO code
Name	manufacturer name followed by the country in round brackets

There are however problems with directly using this dataset, because a lot of manufacturers in the aircraft dataset do not exist here, or do not use the code.

A way to try to minimize the duplication is to also match manufacturers using their names, ignoring case to try to get the best results. However, the data is not very consistent, so there will be inevitably duplicate manufacturers with slightly different names. Manufacturers of which the code are not known but are used by aircraft are also stored in the database, with a code generated based on the name.

3.2.8 Downloading updates from a file server

Downloading files from a file server can easily be done by performing a GET request on a target URL. However, getting updates from a file server without redownloading the file every time can be done as following. Assuming that it is supported.

Most file servers will provide the header field named ***Last-Modified***, which can be used to retrieve the time and date when the file was last updated. Combining this with the ***HEAD*** HTTP method, it is possible to only request for the headers and retrieve the last modified time and date without having to download the file. Then check against the last modified time that is stored locally, and only download the file if needed.

This is very convenient if you want to check for updates frequently or if the files are big.

3.2.9 OpenFlights

OpenFlights provides free airline, airport, and route data. The data that is taken from this source is one with all the known the airports. Including their position, ICAO, IATA, name, country, and city. Every now and then the server will redownload the data file to update for newly added airports. The download endpoint does not provide the last modified time, so the file needs to be redownloaded every time we want to check for updates.

Website:

https://openflights.org/faq

Airport dataset

This dataset looks like an CSV file, but *null* is represented by *\N*. Each row represents one airport and the index in the following table represents each column. There is no header.

Url:

https://github.com/jpatokal/openflights/blob/master/data/airports.dat

Fields:

INDEX	NAME	VALUE
0	Airport ID	Unique OpenFlights identifier
1	Name	Name of the airport, can contain the city name
2	City	City where the airport is located
3	Country	Country where the airport is located
4	IATA	3-letter IATA code, if known
5	ICAO	4-letter ICAO code, if known
6	Latitude	Latitude
7	Longitude	Longitude
8	Altitude	Altitude, in feet
9	Timezone	Hours offset from UTC
10	Tz database time zone	Time zone in "tz" (Olson) format
11	Type	Is always *airport*
12	Source	Source of this data

3.2.9.1 Flightradar24

Flightradar24 is a flight tracker, unlike OpenSky Network this is mainly a paid service, especially for a REST API. We can only rely on endpoints that are publicly available mainly for the purposes of their website. Like OpenSky Network does this service also provide state data. But it also provides information about the flight itself, like the origin and destination airports, the model of the aircraft, the waypoints of the flight, etc.

Website:

https://www.flightradar24.com

Rest API specificiation

Base url: https://data-live.flightradar24.com/

Get all aircraft

This is what Flightradar24 uses to get all the aircraft that should be displayed on a map. Not every option is known, because it is not a publicly available API.

API path: /zones/fcgi/feed.js

API parameters

NAME	VALUE
bounds	north, south (latitude), west and east (longitude) joined together with ,
faa	0 or 1, whether FAA information should be included
flarm	0 or 1, whether FLARM technology data is to be included, defaults to 1
mlat	0 or 1, whether MLAT technology data is to be included, defaults to 1
adsb	0 or 1, whether ADSB technology data is to be included, defaults to 1
air	0 or 1, whether the aircraft should be in the sky or not, defaults to 1
gnd	0 or 1, whether the aircraft should be on the ground or not, defaults to 0
vehicles	0 or 1, whether inactive aircraft should be included, defaults to 0
gliders	0 or 1, whether gliders should be included, defaults to 0
estimated	0 or 1, whether estimated positions are allowed, defaults to 0
..other..	

Response (json object)

NAME	VALUE
full_count	amount of aircraft entries
version	version
..entries..	an entry is represented as an array mapped to the **flight identifier**; a flight identifier is a hexadecimal string of 8 characters. So, a 32-bit integer number.

Entry (json array)

INDEX	VALUE
0	ICAO aircraft registration number
1	latitude
2	longitude
3	heading, in degrees
4	altitude, in feet
5	ground speed, in knots, can be 0
6	squawk code
7	F24 radar source id
8	ICAO aircraft type designator
9	registration number
10	timestamp, in seconds since epoch
11	origin airport IATA code, can be empty
12	destination airport IATA code, can be empty
13	flight number, can be empty
14	0 or 1, whether the aircraft is on the ground
15	vertical speed, in ft/min
16	ICAO ATC call signature
17	0 or 1, whether it is a glider

Example: https://data-live.flightradar24.com/zones/fcgi/feed.js?gnd=1&gliders=1

Get flight information

This is what Flightradar24 uses to request more specific information about an aircraft and its flight.

API path: /clickhandler
API parameters:

NAME	VALUE
flight	**flight identifier** that was returned by the ***get all aircraft*** call
version	1.5

This information was found by looking through another project that interacts with the API: https://github.com/derhuerst/flightradar24-client/

3.2.10 JSON serialization

All the json serialization goes through the ***Kotlin Serialization*** framework. This library can transform data objects into JSON and back.

For types that are not in our control, because they are from external libraries or for list-based structures, it is necessary to write some extra code. The library is also designed to serialize to binary data so specifying custom serializers is a bit more complicated.

The ***Ktor*** library also supports ***Kotlin Serialization***, which means that it can serialize the objects when they are send to the client or when deserialize them when they are received. This minimizes the amount of handling we need for JSON objects, while having an explicit contract of used object types and their fields.

All supported object types are either annotated with `@Serializable` or they have a serializer which is annotated with `@Serializer`.

3.3 Aircraft matching algorithm

This section is the implementation of what was described in the analysis. The image below gives a small recap on what was described. Every step of the process will now be described in detail, and with some extra things that are deemed necessary.

Firstly, let us look at the information that the mobile device can provide to the back-end. Please refer to the 4.1.2.4 Request visible aircraft states section, in this section you see that a timestamp, position, rotation, field of view and the detections are send to the back-end. This information combined with the information about the surrounding aircraft will be used to determine which ones are in the view.

The formulas in section ***3.3.1 Coordinate system conversion*** are used in matching algorithm.

The following steps give a rough idea of how the request is processed:

1. First define some variables.

```
geodeticPosition = request.position
enuRotation = request.rotation
fov = request.fov
imageDetections = request.aircrafts
time = request.time
```

2. Define the ENU transform, this is just the position and rotation that were received from the mobile device. The rotation is a quaternion, which is what will be used to work with rotations.

```
enuTransform = EnuTransform(geodeticPosition, enuRotation)
```

3. Convert to ECEF transform. This converts the geodetic position to an ECEF position and the ENU transform to an ECEF transform. The transformation is based on the formulas found in the analysis.

```
ecefTransform = enuTransform.toEcefTransform()
```

4. Get the surrounding aircraft at the geodetic position, within a range of n degrees (by default 0.185 degrees). Also supply the time at which the mobile device data was collected.

```
surroundingAircraft = findAircraft(geodeticPosition, range, time)
```

5. Construct a virtual camera with the FOV (field of view) of the mobile device camera and the ECEF transform.

```
camera = Camera.ofPerspective(fov).withTransform(ecefTransform)
```

6. Now, let us define a function "`match`" which will try to match the surrounding aircraft with the view of the camera and the image detections of the mobile device.

```
match(camera)
```

   a. For each aircraft in surroundingAircraft, transform each entry as the following: Convert its position to an ECEF position. Then convert the ECEF position to a camera view position.

```
aircraftEcefPosition = aircraft.position.toEcefPosition()
aircraftViewPosition = camera.toViewPosition(aircraftEcefPosition)
(aircraft, aircraftEcefPosition, aircraftViewPosition)
```

   b. Filter out every entry where the view position is null, a null view position implies that the aircraft was not in the view of the camera.
   c. Sort the remaining entries based in the distance between the camera position and the ECEF position of each entry. This results in a sequence of entries that are in the camera view where the closest entries are first in the sequence.

```
closestInView = (apply steps 1 – 3)
```

d. Now, the aircraft that has been detected by the image recognition must be mapped accordingly to the ones that are in the view. The image detections must be mapped by distance, to do this the size on the screen will be used.
   The distance will only be used if two image detections have a recognizable difference in size on the screen. When two detections are similar, the position on the screen will be used instead.

   Each image detection gets a screen size group, calculated based on the x and y sizes. After grouping them, they will be sorted by the size (area) of each group, the value is inverted so bigger sizes appear first in the sequence.

```
imageDetectionGroups = imageDetections
    .groupBy(detection => screenSizeGroup(detection))
    .sortBy(screenSizeGroup => -(screenSizeGroup.x * screenSizeGroup.y))
```

e. After grouping them by similar size, each group will be processed separately. Within each group, take n entries (size of the group) from the aircraft that are closest in view. Then, find for each image detection the closest entry, each entry may only be used once.
f. After this, every image detection has found a matching aircraft. Or, if not sufficient aircraft are available in closest in view, a null will be returned instead for that image detection.

7. At this point the match function is used for the camera. If the match function provides an exact output aircraft count that matches the request image detection count, then just return the result.
   Otherwise the user is to retry with a slightly different stance, by rotating the camera a few degrees in each direction (left, up, right, down), to see if this gives a better one. This should counteract some problems caused by sensor inaccuracy.

3.3.1 Coordinate system conversion

This section deals with the implementations of the conversion functions between coordinate systems.

3.3.1.1 Geodetic position to ECEF position

This is a literal conversion of the function which was defined before, refer to ***3.1.4.2 Coordinate systems*** for more information.

```
private const val A = 6378137.0 // equatorial radius [m]
private const val B = 6356752.3 // polar radius [m]

private const val A_POW2 = A * A
private const val B_POW2 = B * B

private const val B_POW2_DIV_BY_A_POW2 = B_POW2 / A_POW2

/**
 * Converts the [GeodeticPosition] to an ECEF (earth-centered, earth-fixed)
 * position, the unit of each component (x, y, z) is in meters.
 */
fun GeodeticPosition.toEcefPosition(): Vector3d {
  val radLat = degToRad(latitude)
  val radLon = degToRad(longitude)

  val sinLat = sin(radLat)
  val cosLat = cos(radLat)
  val sinLon = sin(radLon)
  val cosLon = cos(radLon)

  val n = A_POW2 / sqrt(A_POW2 * cosLat * cosLat + B_POW2 * sinLat * sinLat)
  val x = (n + altitude) * cosLat * cosLon
  val y = (n + altitude) * cosLat * sinLon
  val z = (B_POW2_DIV_BY_A_POW2 * n + altitude) * sinLat

  return Vector3d(x, y, z)
}
```

3.3.1.2 ENU rotation to ECEF rotation

This conversion requires that the position is also known as the function which was defined before in ***3.1.4.2 Coordinate systems*** is used to transform vectors from ENU to ECEF. The ENU rotation will first be converted to a direction vector, after that will the rotation matrix be applied. Finally, the rotated vector is converted back to a rotation.

The `getEnuToEcefRotationMatrix` function implements the rotation matrix, which is literally taken from the original formula.

For working with rotations it is preferred to use quaternions, which is a representation of a rotation. Please refer to ***3.3.1.3 Quaternion*** for more information on quaternions.

```
/**
 * Converts the [EnuTransform] to an ECEF (earth-centered, earth-fixed)
 * transform, the unit of each position component (x, y, z) is in meters.
 */
fun EnuTransform.toEcefTransform(): Transform {
  val ecefPosition = position.toEcefPosition()

  val enuToEcefMatrix = position.getEnuToEcefRotationMatrix()
  // Convert the original rotation to a vector using the unit z vector (0,0,1),
  // 1. Rotate the unit z vector (0,0,1) with the original rotation
  // 2. Transform the resulting vector from ENU to ECEF
  // 3. Create a new rotation between the start unit z vector and the
  //    transformed vector
  val ecefRotation = Quaterniond.fromRotationTo(
      Vector3d.UNIT_Z, enuToEcefMatrix.transform(rotation.rotate(Vector3d.UNIT_Z)))

  return Transform(ecefPosition, ecefRotation)
}

/**
 * Gets a rotation matrix that can be used to convert from an ENU coordinate system
 * at the target position to the ECEF coordinate system.
 */
private fun GeodeticPosition.getEnuToEcefRotationMatrix(): Matrix3d {
  val radLat = degToRad(latitude)
  val radLon = degToRad(longitude)

  val sinLat = sin(radLat)
  val cosLat = cos(radLat)
  val sinLon = sin(radLon)
  val cosLon = cos(radLon)

  return Matrix3d.from(
      -sinLon, -sinLat * cosLon, cosLat * cosLon,
      cosLon, -sinLat * sinLon, cosLat * sinLon,
      0.0, cosLat, sinLat
  )
}
```

3.3.1.3 Quaternion

A quaternion is a four-element vector (x, y, z, w) that can be used to represent a rotation in 3D coordinate system. Technically, it consists of a single real number and 3 complex numbers, but that is something that we can disregard. We only need to know how to apply them.

An advantage of using quaternions is that it does not suffer from the “gimbal lock”. Which is the case for Euler angles (yaw, pitch, roll), Euler angles cannot measure the orientation when the pitch angle approaches +/- 90 degrees.

3.3.2 Convert ECEF to 2D camera positions

Before converting ECEF coordinates to camera positions, a camera object needs to be created. A camera object consists of a ECEF position, rotation and a projection matrix which is created using the FOV (field of view). The matrix is created using a library function.

The following function is used to create the camera using the FOV, after that, the position and rotation will be applied. The far and near values are clipping planes, which is not very relevant for this application, the range just needs to be big enough.

```
/**
 * Creates a new [Camera] from the given [fov], where the x component
 * represents the horizontal and y the vertical FOV (field of view).
 */
fun ofPerspective(fov: Vector2d): Camera {
  val near = 0.1
  val far = 1000.0
  val aspect = fov.x / fov.y
  return Camera(Matrix4d.createPerspective(fov.x, aspect, near, far))
}
```

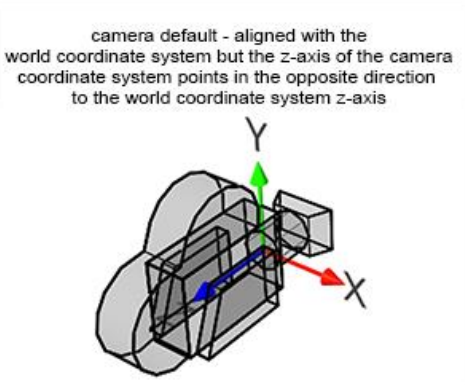
The camera has also a property to create the view matrix, which will be used to map positions to its coordinate system.

```
/**
 * The transformation matrix to transform world coordinates to
 * the camera coordinate system.
 */
val viewMatrix: Matrix4d by lazy {
  val rotationMatrix = Matrix4d.createRotation(transform.rotation.invert())
  val positionMatrix = Matrix4d.createTranslation(transform.position.negate())
  rotationMatrix.mul(positionMatrix)
}
```

Then, create the camera using the FOV and the apply the transform (position and rotation).

```
val camera = Camera.ofPerspective(fov).withTransform(transform)
```

Now, this camera can be used to transform 3D positions to its 2D plane. The positions are first remapped to the camera coordinate system, which can be seen in the image. Then, its already possible to see whether the point lies behind the camera or not. Then, the 3D position on the camera coordinate system needs to be mapped to the 2D plane. After which everything that lies outside the plane is filtered. There is an epsilon value used for edge cases, mainly for unit tests.



The following code is used to achieve this.

```
/**
 * Converts the 3d point to a 2d point within the camera view. Returns
 * null if the point is not within the camera view.
 *
 * The returned position is within bounds [0,0] to [1,1]
 */
fun Vector3d.toViewPosition(camera: Camera): Vector2d? {
  // Camera Coordinate System
  // https://www.scratchapixel.com/images/upload/perspective-matrix/camera.png

  // Information on camera projection
  // https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-
projection-matrix/building-basic-perspective-projection-matrix

  // Transform the point in the world (ECEF) coordinate system
  // to the camera coordinate system, see the image above
  var pos = camera.viewMatrix.transform(this)
  // The point is behind the camera, so not visible
  if (pos.z > 0)
    return null

  // Project the point (in the camera coordinate system) to the
  // camera projection plane, everything in range -1.0..1.0 in
  // the x and y direction is within the visible area of the
  // camera
  pos = camera.projectionMatrix.transform(pos)

  // Convert the projection output to an image coordinate system
  // so top left is (0,0), bottom right is (1,1)

  // -1.0..1.0 -> 0.0..1.0
  // 0.0 is left, 1.0 is right
  val x = (pos.x + 1.0) / 2.0
  // -1.0..1.0 -> 1.0..0.0
  // 0.0 is top, 1.0 is bottom
  val y = 1.0 - (pos.y + 1.0) / 2.0

  // Check whether the values are valid, using a epsilon
  // to fix errors on edge cases
  if (x !in -Epsilon..1.0 + Epsilon || y !in -Epsilon..1.0 + Epsilon)
    return null

  return Vector2d(
      x.coerceIn(0.0, 1.0),
      y.coerceIn(0.0, 1.0))
}

private fun Matrix4d.transform(vector: Vector3d): Vector3d {
  val transformed = transform(vector.toVector4(1.0))
  // Normalize if w is different than 1 (convert from homogeneous to cartesian coordinates)
  return if (transformed.w != 1.0) {
    transformed.toVector3().div(transformed.w)
  } else {
    transformed.toVector3()
  }
}
```

3.3.2.1 Unit test

This part of the code contains unit tests, as this essential method is hard to test with just real-life data. There were an number of issues with this part of the code as well, so proper unit tests are very important.

Every unit test creates a camera with a specific rotation, for example rotated 90 degrees up or left. Then, some fixed 3D points will be converted to 2D positions and compared. The epsilon used in the `toViewPosition` function is for this very important, because most of the points are edge cases.

The following code is an example of a single unit test.

```
private fun assertSamePoint(expected: Vector2d, actual: Vector2d?) {
  check(actual != null) { "Expected $expected, but got $actual"}
  assert(actual.distanceSquared(expected) < Epsilon) {
    "Expected $expected, but got $actual" }
}

@Test
fun `check within camera view 1`() {
  val transform = Transform(Vector3d.ZERO,
      Quaterniond.fromAngleDegAxis(45.0, Vector3d.UNIT_Y))
  val camera = Camera.ofPerspective(Vector2d(90.0, 90.0))
      .withTransform(transform)

  // Left
  assertSamePoint(Vector2d(0.0, 0.5), Vector3d(-1.0, 0.0, 0.0).toViewPosition(camera))
  // Center
  assertSamePoint(Vector2d(0.5, 0.5), Vector3d(-1.0, 0.0, -1.0).toViewPosition(camera))
  // Right
  assertSamePoint(Vector2d(1.0, 0.5), Vector3d(0.0, 0.0, -1.0).toViewPosition(camera))
}
```

4 Front-end to back-end communication

The front-end will communicate with the back-end REST service to retrieve information for the application. The front-end does not have to rely on other web API's for its functionalities. This should be handled by the back-end to keep a clean interface between the back and front end.

4.1 REST Specification

This section describes the REST specification. All the data is represented as JSON, which will be used in requests and responses. Most operations support GET and POST http methods, why it makes sense to use GET solely for retrieving data. Not all libraries properly support using GET in combination with a body.

4.1.1 Response

There are two types of responses, it can either be a success or a failure.

4.1.1.1 Success

If a request is successful, the API will respond with a JSON object which contains a *data* field. This data field will contain all the data depending on the request type, as is described below.

Example:

```
{
  "data": {
    // response data
  }
}
```

4.1.1.2 Error

If a request were to result in an error, the API would respond with a JSON object which contains an *error* field. The field object contains the error code, which will also be the HTTP response code. And an optional message which explains why the error occurred.

Example:

```
{
  "error": {
    "code": 404,
    "message": "optional message"
  }
}
```

4.1.2 Request

Optional fields in a request or response implies that the field may be absent or contain null.

4.1.2.1 Geodetic position

A geodetic position is represented by an array, the values at each index can are described below. The length of the array is always 2 or 3.

Index	Name	Type	Description
0	latitude	double	The latitude, in degrees.
1	longitude	double	The longitude, in degrees.
2	altitude	double	The altitude, in meters. Is *optional*.

4.1.2.2 Geodetic bounds

Property	Type	Description
min	geodetic position	The minimum. (north-west, top-left)
max	geodetic position	The maximum. (south-east, bottom-right)

4.1.2.3 Aircraft state

The following data will be present every time data for an aircraft state is returned. These are common properties.

Property	Type	Description
time	int	Timestamp at which the data was captured. Unix-time in seconds.
icao24	string	Unique ICAO 24-bit address of the aircraft. Can be used in later requests.
position	geodetic position	The position of the aircraft.
velocity	float	The velocity (ground speed) of the aircraft (in m/s). Is *optional*.
vertical_rate	float	The vertical speed of the aircraft (in m/s). Is *optional*.
heading	float	The heading of the aricraft (in degrees).
weight_category	string	The weight category. Is *optional*. Can be any value of `Ultralight`, `Light`, `Normal`, `Heavy` or `VeryHeavy`.

4.1.2.4 Request visible aircraft states

These are ttempts to request info from visible aircraft that are captured by a camera. Multiple aircraft can be requested. There is no guarantee that a matching aircraft will be found for every request.

Operation

`GET|POST /v1/aircraft/state/visible`

Fields

Property	Type	Description
time	int	Timestamp at which the device data was collected. Unix-time in seconds.
position	*geodetic position*	The position of the GPS of the device.
rotation	array [x, y, z, w]	The rotation quaternion of the mobile device.
fov	array [x, y]	The angles (in degrees) that represent the FOV (Field of View) of the camera.
aircrafts	array of *aircraft*	See below.

Aircraft

This represents an aircraft that was detected on the photo that was captured by a device. The position and size are scaled accordingly to the proportions of the captured photo, so that they are within the bounds [0,0] and [1,1] where [0,0] is the top left position and [1,1] is the bottom right position.

Take note of the index of the aircraft within the array, the position of the aircraft will be used in the response of the request.

Property	Type	Description
position	array [x, y]	The center position of the aircraft on the screen of the device.
size	array [width, height]	The size of the aircraft on the screen of the device.

Reponse data

Property	Type	Description
states	array of *nullable aircraft state*	The state at a specific index will be non-null if it was matched successfully.

4.1.2.5 Request aircraft states

Requests all the states of aircraft that are currently flying. Optionally within bounds and at a specific time.

Operation

`GET|POST /v1/aircraft/state/all`

Fields

Property	Type	Description
time	int	The time at which the state was applicable. Unix-time in seconds. Is *optional*.
bounds	geodetic bounds	The bounds where the aircraft should be located within. Is *optional*.

Response data

Property	Type	Description
aircrafts	array of *aircraft state*	The found aircraft within the area around the device position.

4.1.2.6 Request aircraft state

Requests the state of a specific aircraft.

Operation

`GET|POST /v1/aircraft/state/get`

Fields

Property	Type	Description
icao24	string	Unique ICAO 24-bit address of the aircraft.

Response data

aircraft state

4.1.2.7 Request aircraft info

Requests information about a specific aircraft. Like the model, manufacturer, owner, etc.

Operation

`GET|POST /v1/aircraft/info`

Fields

Property	Type	Description
icao24	string	Unique ICAO 24-bit address of the aircraft.

Response

Property	Type	Description
icao24	string	Unique ICAO 24-bit address of the aircraft.
model	string	Name of the model.
owner	string	Name of the owner. Is *optional*.
type	string	The type. Is *optional*. Can be any value of *LandPlane*, *SeaPlane*, *Amphibian*, *Helicopter*, *Dirigible, or Glider*.
manufacturer	manufacturer	Manufacturer. Is *optional*.
weight_category	string	The weight category. Is *optional*. Can be any value of `Ultralight`, `Light`, `Normal`, `Heavy`, or `VeryHeavy`.
engines	engines	Engines information. Is *optional*.

Manufacturer

Property	Type	Description
name	string	The name.
country	string	The country. Is *optional*.

Engines

Property	Type	Description
type	string	The type. Is *optional*. Can be any value of *Piston*, *Turboprop*, or *Jet*.
count	int	The number of engines. Is *optional*.
name	string	The engines name. Is *optional*.

4.1.2.8 Request aircraft flight

Requests data for the current flight of a specific aircraft. The response can contain a track and flight info. Both will be missing if not information could be found.

Operation

`GET|POST /v1/aircraft/flight`

Property	Type	Description
icao24	string	Unique ICAO 24-bit address of the aircraft.

Response data

Property	Type	Description
icao24	string	Unique ICAO 24-bit address of the aircraft.
number	string	The flight number. Is *optional*.
departure_airport	airport	The departure airport. Is *optional*.
arrival_airport	airport	The arrival airport. Is *optional*.
estimated_arrival_time	int	The estimated arrival time. Unix-time in seconds. Is *optional*.
waypoints	array of waypoint	The trajectory the aircraft is following. Is *optional*.

Waypoint

Property	Type	Description
time	int	Timestamp at which the point will be reached. Unix-time in seconds.
pos	geodetic position	The position.

Airport

Property	Type	Description
icao	string	The ICAO code.
iata	string	The IATA code.
name	string	The name. Can include the city name.
city	string	The name of the city.
country	string	The name of the country.
position	geodetic position	The position.

5 Testing

Setting up a reliable test environment for the application is not easy, because it relies on the fact that there are aircraft flying around us. Since this application is being developed during the COVID-19 pandemic, this issue is even lager because there are almost no flights. So, the opportunity to test on real aircraft is limited.

In both cases, we need to setup an integration test environment which is consistent so we can get consistent results. Location data was collected from some noticeable locations in the direct surroundings, each of these locations represents a test aircraft. These test aircraft are static.

Some parts of the code also have unit tests, specifically ones that are hard to test in real-life situations or ones that need be working before making assumptions in real-life situations.

5.1 Test data

The following JSON is an entry of test data, which is paired to an photo of the location. All test aircraft used an ICAO24 starting with FFFF, which should be well out the range of used identifiers by real aircraft. The red dot in the photo marks the spot where the aircraft is located.

```
{
  "icao24": "FFFF01",
  "model": "Test Aircraft 1",
  "position": [
    51.03227347222222,
    4.677712083333334,
    83
  ]
}
```

6 References

Flutter. (n.d.). *Flutter Documentation*. Retrieved from Flutter: https://flutter.dev/docs

Klimushyn, M. (n.d.). *How Flutter apps are compiled with Gradle for Android*. Retrieved from https://github.com/: https://github.com/flutter/flutter/wiki/How-Flutter-apps-are-compiled-with-Gradle-for-Android

Rosebrock, A. (2018, November 12). *YOLO object detection with OpenCV*. Retrieved from pyimagesearch: https://www.pyimagesearch.com/2018/11/12/yolo-object-detection-with-opencv/

Scratchapixel. (n.d.). *The Perspective and Orthographic Projection Matrix*. Retrieved from Scratchapixel: https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/building-basic-perspective-projection-matrix

Tensorflow. (n.d.). *Object detection*. Retrieved from Tensorflow: https://www.tensorflow.org/lite/models/object_detection/overview

Wikipedia. (n.d.). *Axes conventions*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Axes_conventions#Ground_reference_frames:_ENU_and_NED

Wikipedia. (n.d.). *ECEF*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/ECEF

Wikipedia. (n.d.). *Geographic coordinate conversion*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Geographic_coordinate_conversion#From_geodetic_to_ECEF_coordinates

Wikipedia. (n.d.). *Local tangent plane coordinates*. Retrieved from https://en.wikipedia.org/wiki/Local_tangent_plane_coordinates