Your first task at EN-CO

# Task description

In this task you’ll have to write a web application, processing and displaying sailplane (engineless aircraft) flights. With your app, pilots will be able to upload their flights to a server where flights get automatically processed and later approved by an admin.

While writing the app, you’ll encounter the following topics: processing text and Excel files, working with databases (Entity Framework, LINQ), user authentication (Identity), background processing, and a bit of JavaScript will pop up as well on client side ☺

The look of the site doesn’t really matter, the goal of this task is to practice solving backend problems that often occur in real projects.

The application can be written over either .NET Framework or .NET Core, it’s up to you ☺

The exact system functions menu by menu can be found below. At the end of the document you’ll also find a little help for processing files containing flight logs.

# Roles

## Pilot

The pilot of the given flight who logs into the app to upload his/her flight. This role requires logging in.

## Administrator

It’s his/her job to approve flights uploaded by pilots. This role requires logging in.

# Menu items

## Menu items accessible by pilots

### Logfile upload

The only thing present in the page is a flight uploader. User browses a certain flight log file and then presses the upload button. This sends the file to the server. (upon the completion of upload the user is presented with a success message on the screen)

### My flights

This page displays the flights uploaded by the pilot in a table. As for the implementation of the table, choose the simplest solution, no sorting or paging is needed.

A row of the table contains the following information:

* Date of flight
* Duration (length of flight in time)
* Place of takeoff
* Place of landing
* Status (under approval, approved, rejected)
* A button for navigating to details view

### Flight – details view

In the details view one can find the same data that was in the table, plus a map view that displays the flight log as a polyline made from GPS fixes. You may pick a map provider you like the most ( eg. Google Maps, Bing Maps, OpenStreetMap)

## Menu items accessible by administrators

### Flights

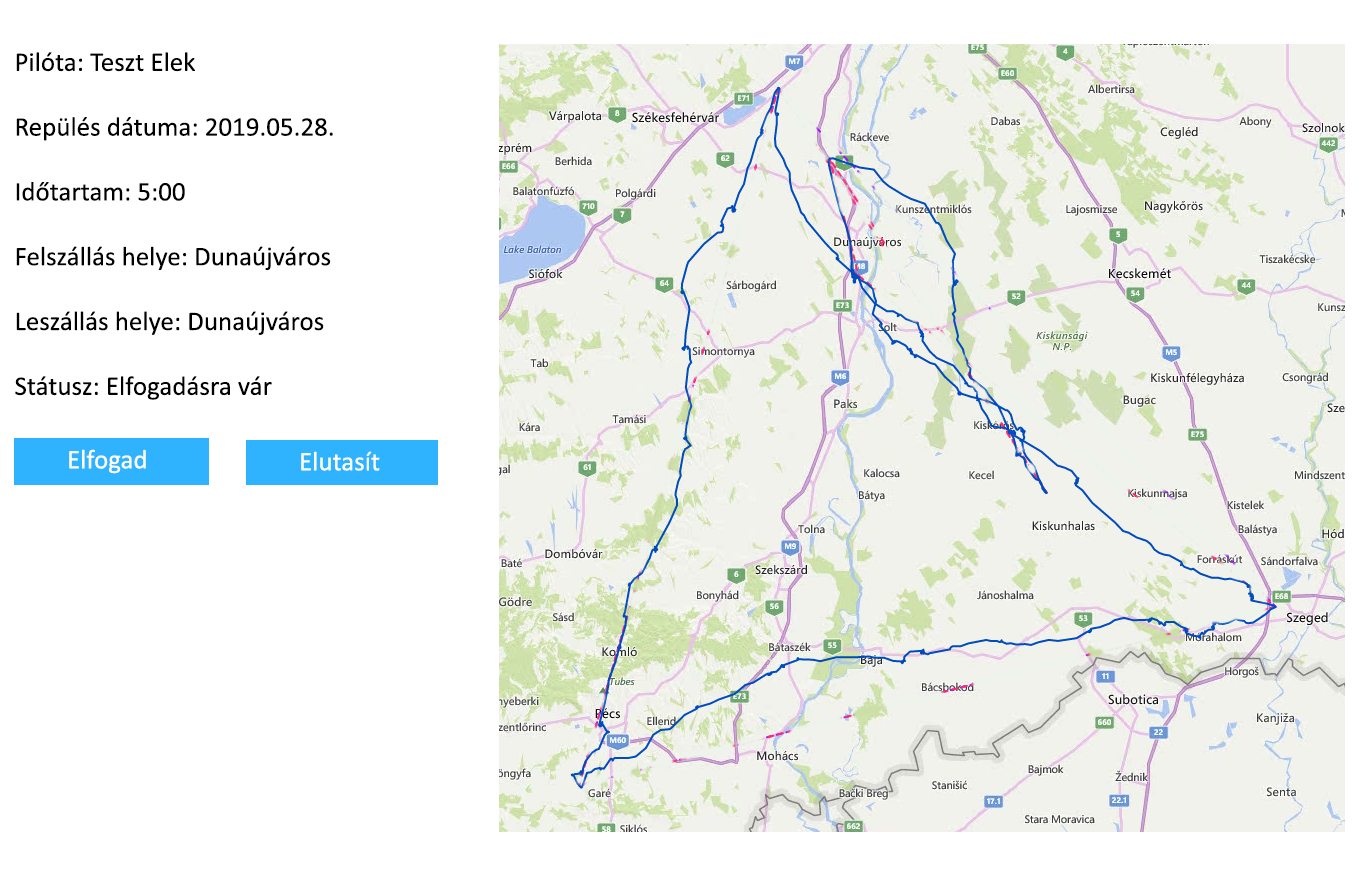
The user is presented with a dropdown on the top of the page with the following items: ’under approval, approved, rejected’. By changing the selected value of the dropdown the table below displays flights with the corresponding status only. By default flights with ’under approval’ status are displayed in the table. Data found in a row of the table is basically the same as described above, but it contains the name of the pilot as well.

### Flight – detailed view

This view is basically the same as that of the pilots’ with the exception of having two extra buttons on the screen (’approve’, ’reject’) if the given flight is in ’under approval’ status.

Should the admin click on one of these buttons, the flight is transferred into the corresponding status and the page gets reloaded. (showing the new status after reload)

A wireframe of how the page should look like:   
(sorry, had no time to rewrite the labels in the picture ☺ )



### Airfields

Besides flights, the administrator also manages a database of airfields. (this will start making sense when you see the flight processing task ☺) In this menu the admin is able to upload data to the database from an Excel file which contains the following info for each airfield:

* Name
* Latitude
* Longitude

The page itself looks the same as the log uploading page of the pilot, with a sole file uploader control and an upload button. By pressing the button the user can upload the Excel file to the server, which gets processed right away, saving the airfields into the database. After the call is returned the user is presented with a success message on the screen.

# Processing the GPS log

Now that everything is prepared for the arrival of the first log file, let the fun part begin ☺

As it was mentioned at flight upload, upon pressing the upload button the only thing that happens is that the file gets uploaded to the server. The actual processing of the file will begin afterwards by a background process. (you may choose the background processing library you like the most, we suggest using Hangfire) The process runs at given intervals (a short interval is favorable for testing purposes) and finds out whether there are uploaded files it should process.

*(Hint: when uploading the GPS log, a db log entry should be created that indicates the presence of a file that’s waiting for processing. When the process runs, it checks the content of the table, processes the files belonging to the db entries and clears the table when finished. This way you can always keep track of files waiting to be processed.)*

As long as a log file is not processed neither the pilot nor the admin sees it in their flight lists.

What shall the processing function do?

The GPS log is a simple text file with .igc extension. You may have a look at it with Notepad. Though it contains loads of information on the flight, you’ll only need 2 kinds of records (lines) of it as described below.

## Date record

The second line of the file is always going to look like this. This provides the date of flight.

HFDTE250519

You may identify this line in the file by its letters. Numbers identify the day the log was created. The format is DDMMYY, so in this case it’s the 25th May, 2019.

## GPS fix record

This contains a GPS fix describing the position of the sailplane in a given moment. This will make the reconstruction of the flight on a map possible.

This is how such a ’B’ record looks like:

**B1040334619875N01844550E**A006740071501609

**B**: This identifies this line as a GPS fix. Actually you may discard any record (apart from the date record) that doesn’t begin with B. Though ’B’ records are in chronological order in the file, bare in mind that there might be records of other types between them.

**104033:** This provides the UTC time, when the fix was created. This can be resolved as 10 hrs 40 minutes 33 seconds. For the sake of simplicity the ’duration’ field may be computed from the time difference of the first and last fixes of the flight.

**4619875N:** Latitude of the coordinate. Resolution: 46 degrees 19.875 minutes, North.

**NOTE:** You’ll need the classic format that describes latitude as a decimal fraction of degrees. Should the geometry class in high school be a long time ago, we’d like to point out that 1 degree = 60 minutes ☺ If you’d like to check your calculation, we suggest using this site:

<https://www.directionsmag.com/site/latlong-converter/>

**01844550E:** Longitude of the coordinate. Resolution: 018 degrees 44.550 minutes, East. As you can see, degrees are always given in 3 digits, even when it’s less than 100. Processing this number is the same as described above.

In coordinates letters ’N’ and ’E’ can be considered unchanged, since you’re given igc files containing flights from Hungary only. So no need to deal with the other hemisphere of Earth.

Rest of the record contains data that are not needed for solving the task.

# Using airfields

As it’s been mentioned already, airfields can be imported into the database from an Excel file. For processing this Excel file, we suggest using a nuget package called ’EPPlus’ which is really easy to use.

While processing the GPS log – as you might have already found out – you’ll need the coordinates of airfields to find the place of takeoff and landing. For this purpose please use the following rules:

Place of takeoff: The airfield whose coordinates are closer than 3km to the first fix of the log. (There’ll be exactly one airfield matching this criteria)

Place of landing: The same rule applies, however sailplanes don’t always land at airfields. Unfortunately they sometimes end up somewhere else, eg. a plow field. If you don’t find an airfield for the last fix, then the word ’Outlanding’ shall appear in the flight list.

*Help: For the sake of simplicity we’ll measure distance on the surface of a sphere, not a geoid. This can be done by the following simple function. (parameters of the function are custom classes, they’re actually no more than a pair of latitude and longitude values)*

public double CalculateDistance(GlobalPoint start, GlobalPoint end)

{

double rlat1 = Math.PI \* start.Latitude / 180;

double rlat2 = Math.PI \* end.Latitude / 180;

double theta = start.Longitude - end.Longitude;

double rtheta = Math.PI \* theta / 180;

double dist =

Math.Sin(rlat1) \* Math.Sin(rlat2) + Math.Cos(rlat1) \*

Math.Cos(rlat2) \* Math.Cos(rtheta);

dist = Math.Acos(dist);

dist = dist \* 180 / Math.PI;

dist = dist \* 60 \* 1.1515;

return dist \* 1609.344; // Méterben adja vissza a távolságot

}

# Technical requirements

In the end a few things we’d like to mention that should be included in your solution:

We’d like this to be a well structured project. To achieve this, you should create the following projects in your solution:

* Model: Db entities (we’d like you to use a code-first db ☺)
* DAL: Db access (ApplicationContext, DbSets)
* BLL: All business logic, Manager classes for the Db, DTOs
* Web: ViewModels (using Automapper for DTO conversion is suggested), Controllers, UI, etc.

Misc:

* Using dependency injection is mandatory with Autofac, or in case of .NET Core with the built in IoC container
* As for routing, the default Controller/Action/Id solution is preferred to be replaced with attribute routing
* Always use Guid as an identifier in the database. (don’t use int)
* Everything else, like using Unit of work or Repository is optional if it’s more convenient for you, but they’re not required.
* Code shall be well commented. If the purpose of a class or a function is not trivial, please add a description in summary.

**We wish you good luck for the task! Should you have any questions, feel free to ask your mentor! ☺**