Individual Software Project report

Branko Lekić

September 2024

Chapter 1

Goal, design, and implementation

The goal of this Individual Software Project (ISP) was to create a functional and user-friendly mobile application for time-series data forecasting using a model designed and trained by the student. This was accomplished via a cross-platform app written in C#/.NET MAUI, and a web server hosting the model written in Python. These parts are almost entirely distinct, in that they can be used elsewhere or extended independently.

1.1 Developer documentation

1.1.1 Mobile application (frontend)

Dependencies

The ”forecAstIng” app is written in the C# and XAML languages, targeting the .NET, and in particular .NET MAUI, frameworks. .NET MAUI is a modern, but relatively young, superset of Xamarin.Forms that provides cross-platform tools for creating native mobile and desktop apps with a single codebase for Windows, Android, Mac, and iOS.

It is recommended to use Windows and Visual Studio when working with the project's codebase. They are Microsoft products, similar to .NET, and as such have the most complete support for .NET MAUI features. To build the app from source:

1. Download the latest version of Visual Studio (VS) and .NET SDK.
2. Install the .NET Multi-platform App UI development workload in VS [1].
3. Clone the project repository [2] and open App/forecAstIng.sln in VS.

4. Install required NuGet packages via Project -> Manage NuGet packages... in the VS toolbar:
    - CommunityToolkit.Maui
    - CommunityToolkit.Mvvm
    - Microsoft.Extensions.Logging.Debug
    - Microsoft.Maui.Controls
    - Microsoft.Maui.Controls.Compatibility
    - Microsoft.NET.ILLink.Tasks
5. For Android development, set up the Android SDK inside of VS [3].
6. Choose a build target and Debug/Run the application.

A ”ServiceSecrets” class containing API keys is used in the source code but git ignored. This means the developer should register for their free keys for the various services listed under the ”APIs” heading, and add them using the conventions in the code (Same class name and same static variable name). This is an academic project and not a widely distributed commercial product, so the same free keys were used during the development. Additionally, for full functionality, it is necessary to start the Python web server and obtain a reachable address for it. More details can be found under Section 1.1.2.

APIs

APIs were used to source data which is displayed to the user:

1. Open-Meteo [4] - Free weather API from which historical, current, and forecast data (apart from ML model predictions) was obtained.
2. Geocoding [5] - Auxiliary API for turning address/ city name into coordinates (Geocoding) that are used with Open-Meteo, and the reverse process (Reverse geocoding) for more granular location information (Country, city, address, etc.)
3. Custom API for receiving ML model predictions. It is implemented with Flask in Python, and exposed to outside devices through the internet via SocketXP [6].
4. AlphaVantage [7] - Stock market data API with a free, albeit very limited, API key available. Used for finding stock ticker symbol from company name, intraday and daily data, as well as stock ”fundamentals” i.e. details about a company and it’s earnings.

Code architecture design

This project is developed in line with the MVVM (Model-View-ViewModel) architecture design pattern, which is standard practice for .NET MAUI applications. This pattern allows for the separation of data, notification, and UI logic, resulting in better encapsulation and modularity [8].

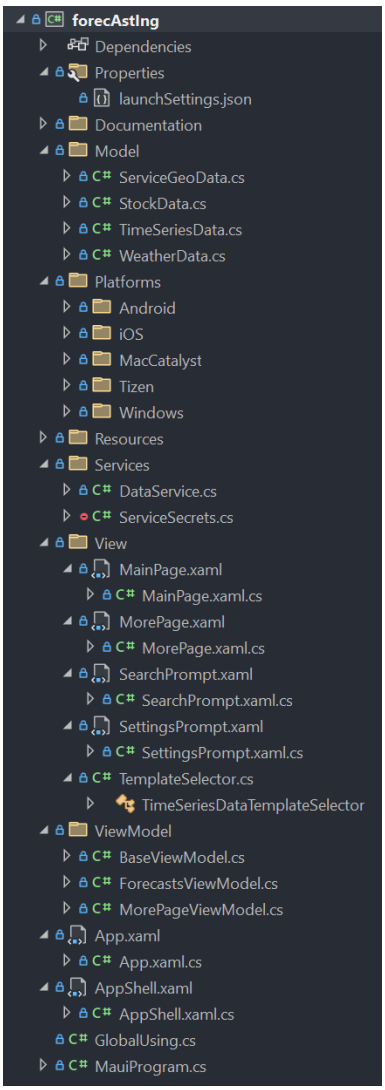


Figure 1.1: Application repository structure

Model The Model directory and namespace contain classes that hold data necessary for all preprocessing and/or displaying. The *TimeSeriesData.cs* file contains a base class of the same name which represents a common abstraction

for any concrete instance of time series data. *StockData.cs* and *WeatherData.cs* hold classes of the same name that inherit from TimeSeriesData, as well as extend it with their specific attributes, such as precipitation for weather, and fundamentals for stocks. Along with *ServiceGeoData.cs* the structure of each object is informed by their respective APIs. Further, certain additions such as [property_name]_current in WeatherData, and TimeSeriesDataLite class exist for more convenient manipulation in the View.

View The View, i.e. GUI of the application, is written almost entirely in XAML. XAML gets compiled down to the Common Language Runtime (CLR), meaning every element has a C# equivalent, however, it is standard practice, and more readable, to use it when working in .NET MAUI. Each different component (*MainPage*, *MorePage*, *SearchPrompt*, *SettingsPrompt*) have their own .xaml file accompanied by a .xaml.cs codebehind file used almost exclusively for initializing the component in the page/popup constructor and registering the binding context (accompanying view model), if applicable. Data binding inside of the view is naturally done by the developer, but the boilerplate code for notification and event delegates and ICommands is autogenerated by the CommunityToolkit.Mvvm package using tags in the view model for e.g. button press commands or watched variables that affect the GUI such as selected tab etc.

ViewModel The ViewModel is a logical bridge between the Model and View, in the sense that it prepares the data for the View to consume it as simply as possible, and handles watching for changes in variables to notify the View. To this end, the *MainPage* interfaces with the *ForecastsViewModel*, and *MorePage* with the *MorePageViewModel*. Most core functionality is implemented inside of *ForecastsViewModel*, and its accompanying *DataService* which is responsible for communicating with data servers through the aforementioned APIs. Async functions are utilized to maintain GUI responsiveness on the main thread.

Other files and directories The *Resources* directory contains various icons used in the app, fonts, and colors and styles that aid in keeping the code more DRY since they define default styles, both light and dark themes, for all the controls in XAML files.
The *Platforms* directory contains subdirectories for all the .NET MAUI supported platforms with platform-specific launch settings and respective ”Main” functions.
The *MauiProgram.cs*, *App.xaml(.cs)*, and *AppShell.xaml(.cs)* files represent a wrap for the entire application to handle certain high-level functions such as registering and automatically constructing view <-> view model pairs, holding app-wide

resources such as styles, and registering transient navigation targets (MainPage <-> MorePage..). Additional comments and reasoning on specific choices are present in the code.

1.1.2 Web server and trained ML model (backend)

Dependencies

The web server and ML parts are written in Python. The directory *ML Model Server* is the root directory for this part of the project. The direct requirements for the project are the following libraries:

- openmeteo_requests
- requests_cache
- retry_requests
- pandas
- numpy
- scikit-learn
- flask

All of the mentioned, and more, implicitly required packages are listed in the *requirements.txt* file. It is recommended to use a python virtual environment, and install the dependencies with the `pip install -r requirements.txt` command.

Model pipeline

Data processing To train a model, it is first necessary to collect and process training data. For the current implementation, hourly data on 2m temperature, 2m relative humidity, apparent temperature, precipitation, and 10m wind speed from January 1st 2000 to December 31st 2023 in Prague was used. This is a proof of concept with room to expand the dataset to larger time frames and/or more places on the Earth; ECMWF provides data from 1940 on a 31km grid [9]. This data was obtained through the aforementioned open-meteo API and has a similar shape to data in the mobile app (*WeatherData.cs*), however, it is loaded through the open_meteo library, in the *weather_data_service.py* file contained in the Services directory.
After getting the raw data, the function:

`data_preprocess_pipeline(data, hours_in_input, hours_in_output)` first extracts cyclical time features from the date column in the raw data, giving two two-part (sin, cos) features intended to help the model encode seasonal effects of months in a year, and hours in a day, on the weather. Then, the aforementioned weather variables are all scaled via scikit_learn's StandardScaler which removes the mean and scales to unit variance [10]. Finally, since we are dealing with time series data, any output depends only on data points before it, so the data is prepared by concatenating a specified number of hours in the input, and its matching (immediately following) specified number of hours in the output; this represents one training sample.

Model training The structure of the program is such that the `data_preprocess_pipeline` function gets called first from inside the `train_MLPRegressor` function, after which it is split into training and testing data, and scikit_learn's MLPRegressor [11] model is trained, the score on the test data printed, and the model (along with data processing transformers) is passed back for use, or compressed saving.

Web server access

The program is run from the *main.py* file, which also defines the `/forecast` route on the flask web server through which model predictions can be obtained. The route takes three parameters:

- *lat* - latitude of desired location
- *lon* - longitude of desired location
- *date* - first day of forecasting

The "main" function first checks the `PULL_DATA_FROM_INTERNET` and `TRAIN_MODEL_ANEW` flags that the developer sets to indicate whether or not to get data from the open-meteo API, and train the model on it again. All data and models are stored in a locally generated Resources directory, from which they are also read. Finally, it attempts to load the model and opens the web server.
To be able to communicate with devices other than the local computer on which the program is run, it is necessary to open the flask web server to listen on every interface (`host=0.0.0.0`). Additionally, it is necessary to download and connect to the SocketXP Cloud Gateway [6]:

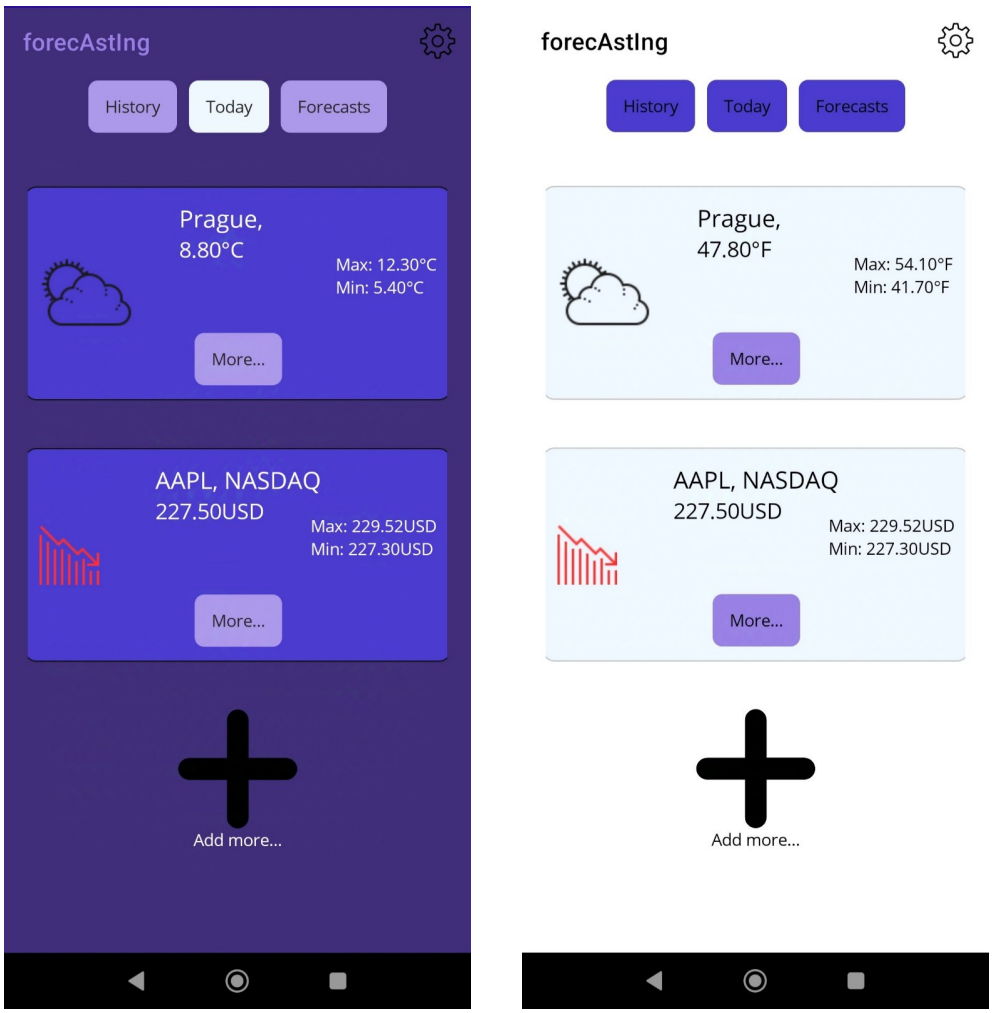
```
.\socketxp.exe login [your auth token]
.\socketxp.exe connect http://localhost:80
```

This enables access over the internet with the public URL generated as the output of the previous command, for example, the one used in the mobile app is:

https://branko-lekic02-d1bb0f5a-e682-4dba-bd68-9597c19a314f.socketxp.com
When a properly formatted request is received by the web server, the function `get_forecast` is called. This function first separates the request parameters, and passes them along with the model and data transformers to the function `get_requested_forecast`. There, inference data corresponding to a number of hours (same as previous `hours_in_input`) before the forecast start date, is gathered, transformed the same as input training data, and fed into the model. The output is then prepared in JSON format by first using the output data transformer to preform an inverse transformation such that we get sensible prediction data. After, the data frame is parsed into an object which resembles open-meteo's weather data (*DataModel/weather_prediction_data.py*), and finally serialized into a JSON string. The response is then sent back to the requestee.
In general, the benefit of this approach is that a larger model can be hosted on a powerful machine, and not locally on smartphone. The data is sent over the HTTP protocol in a relatively light JSON format.

1.2 User documentation

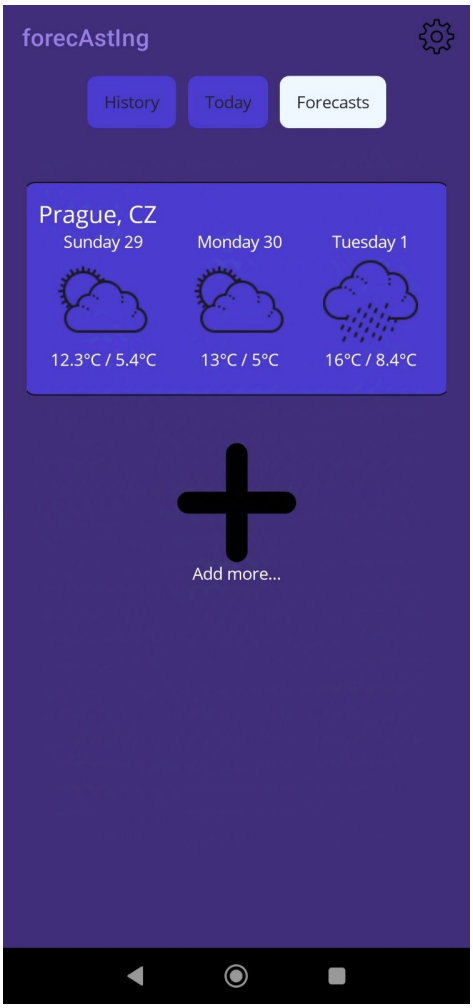
After the web server is launched and open to the internet, the end user can simply launch the application and get full functionality of the app as it stands currently.
This app allows for tracking multiple locations for weather forecasts, as well as ticker symbols for stock data. Locations/ symbols are added by clicking on the "+" icon labeled "Add more..." on the main page, entering the name of the location or stock, and clicking on either "Add location" or "Add stock" respectively. On application startup, it will attempt to locate the device and provide weather data for that location automatically.
The main page consists of 3 tabs - "History" "Today" "Forecasts" - located at the top of the page. Navigating between them presents the relevant data. On the "Today" page, today's conditions are presented, including current, maximum, and minimum temperature/price for each respective location/stock. Clicking the "More..." button opens the details page for the selected location/stock with hourly data and more detailed information; the back arrow can be pressed to return to the main page. The "History" and "Forecasts" tabs have similar layouts, with the first showing past values, and the second predicted ones. Each location/stock presents a scrolling view of 2/3/5 (default) days of historical/forecast data. Clicking on any of the entries, which are annotated by dates they represent, opens the aforementioned details page for that date. From the main page, entries can be deleted by swiping them to the left. Pulling down or clicking on the button of the tab currently opened will refresh the data.



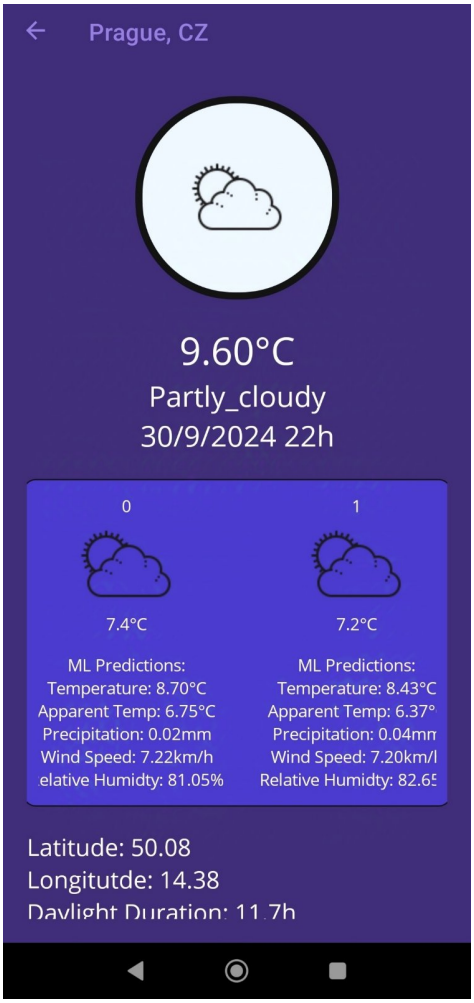
(a) Dark mode, Celcius

(b) Light mode, Fahrneheit

Figure 1.2: Pictures of the main page, where the adding and details buttons can be seen, as well as the settings icon and 3 time frame tabs

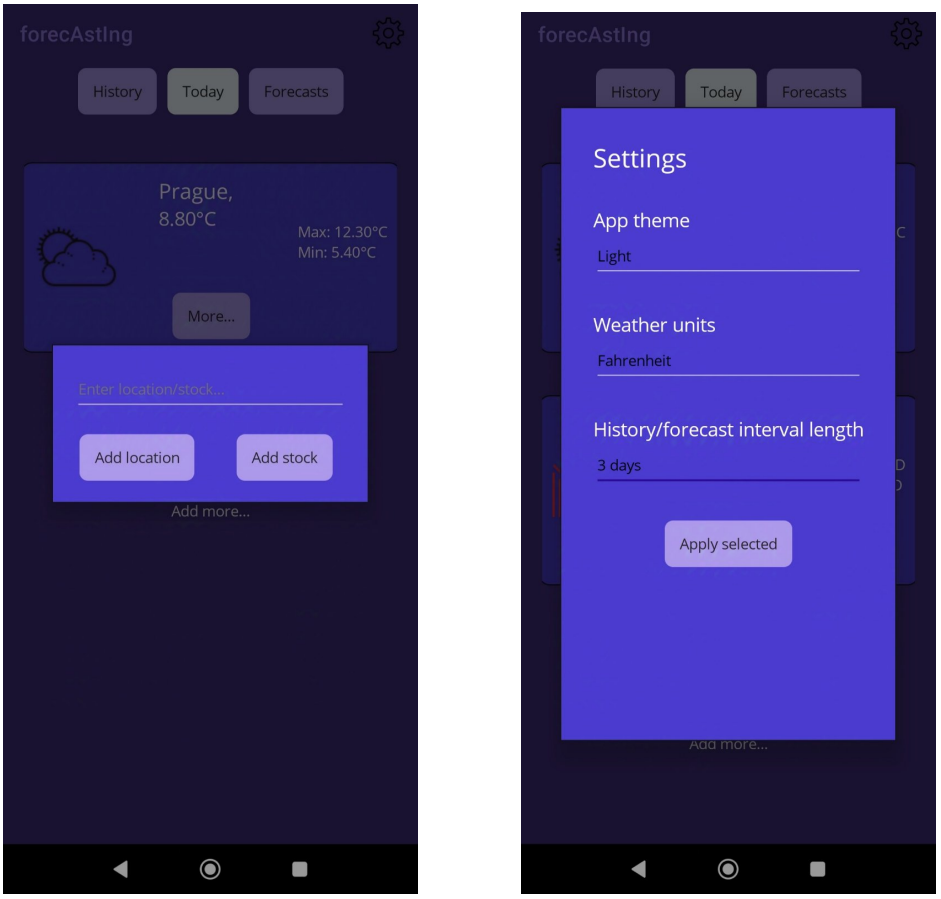


(a) Forecasts tab showing 3 days of forecasts



(b) More page showing ML model predictions below open-meteo's predictions, as well as detailed information about the location stretching downwards

Figure 1.3: Forecasts tab, and more page accessed from it (September 30)



(a) Search prompt for adding new locations/stocks

(b) Settings prompt for choosing app theme, temperature unit, and forecast intervals

Figure 1.4: Search and settings prompts

Settings can be found in the top right corner when clicking on the gear icon. There, number of history/forecast days (2/3/5), temperature units (Fahrenheit/ Celsius), and light/dark mode (light, dark, system default) can be selected.

1.3 Potential future improvements

Model type The main extension of this project to the bachelor thesis will be experimenting with different model structures. Such models may be MLP and TSM Mixer, LSTM and GRU RNNs, Reformer and Informer transformer models, etc. All of them are tailored to time series data, with some representing the state-of-the-art performance. So far, experimentation, in particular with the size and number of hidden layers in the MLP model and number of hours in input and output, have been done manually, and results recorded in comments inside of code. This process should be automated, such that results can be obtained for many different parameters at once.

Training data and evaluation As briefly mentioned, training data is currently limited to Prague, and the same features that are going in are the ones being predicted. It would make sense to include locations from different areas of the world, particularly along the North-South direction, where variations in climate are extreme. As far as input features are concerned, physics models which offer good performance on weather forecasting use many variables to calculate temperature, precipitation, etc. It might be favorable, then, to also use e.g. wind speed at different heights, or cloud cover in the inputs, even if we are not predicting them. A potentially valuable addition might also be better visualisation capabilities on the model side, such as tracking predictions over a period of time, it would then also be possible to compare these predictions to real world values, after some time.

On device model Finally, an on device, lighter model could be useful, especially as a fallback option whenever no internet connection is available. Libraries such as TensorFlow Lite might be a good choice for this.

Links and resources

[1] Microsoft guide for .NET MAUI installation. https://learn.microsoft.com/en-us/dotnet/maui/get-started/installation.

[2] ”forecAstIng” project repository. https://gitlab.mff.cuni.cz/lekib/forecasting.

[3] Get started with Android in Visual Studio. https://learn.microsoft.com/en-us/dotnet/maui/get-started/first-app?view=net-maui-8.0&tabs=vswin&pivots=devices-android.

[4] Open-Meteo API Documentation. https://open-meteo.com/en/docs.

[5] Geocoding API. https://geocode.maps.co.

[6] SocketXP. Remote access a web app from internet. https://www.socketxp.com/iot/remote-access-iot-web-app-from-internet.

[7] AlphaVantage API Documentation. https://www.alphavantage.co/documentation.

[8] Microsoft Learn MVVM. https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm.

[9] ECMWF ERA5 Reanalysis Dataset. https://www.ecmwf.int/en/forecasts/dataset/ecmwf-reanalysis-v5

[10] Scikit_learn documentation. StandardScaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

[11] Scikit_learn documentation. MLPRegressor. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html