25.04.2023

Project document for Weather Data Dashboard

1. Personal information

Abdi-Ra'uf Salah 100322581

Computer Science *(tietotekniikka)*

First year

2. General description

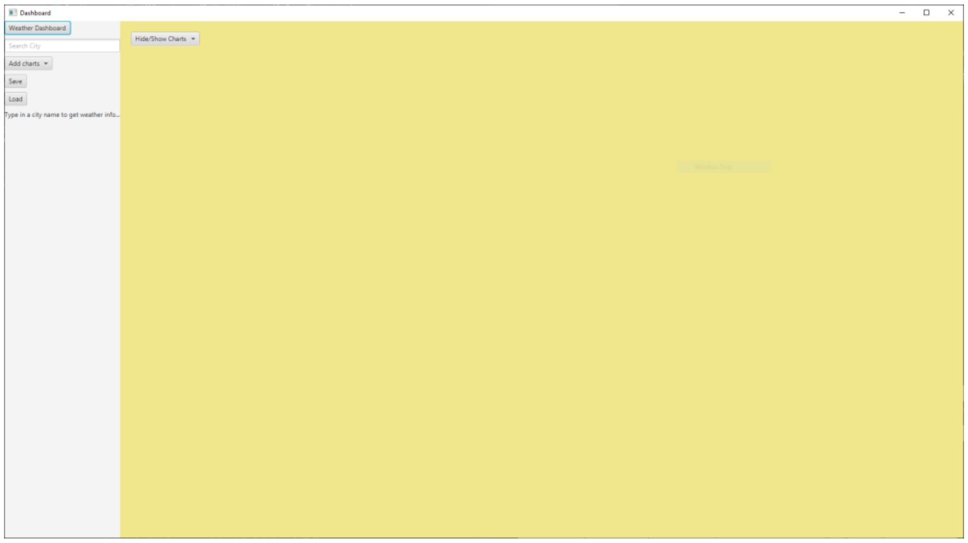
The project is about a Data Dashboard that visualizes data from different cities. The whole project is built with ScalaFX, JavaFX (partially) and Circe libraries. The dashboard itself has different types of components, such as scatterplot, column plot, pie chart and a 'Card' component.

When you first load up the program it is empty, to this empty dashboard you add all these different components. Once you've done this you can input a city name into the text field on the top left of the program. As soon as you hit enter it will get the weather data from openweathermap.org and then it visualizes this data on to our charts.

All of the different charts can be resized with their own sliders, they can be duplicated as well as removed by right clicking them. When you hover over a chart's data point it will reveal more detailed information about that data point. I've implemented everything from the moderate requirements, and I've implemented the first demanding task and the last demanding task. The first demanding task was loading and refreshing dynamic data from the internet (this is the weather data). So I would say I've implemented the work at a moderate but slightly demanding level.

3. User interface

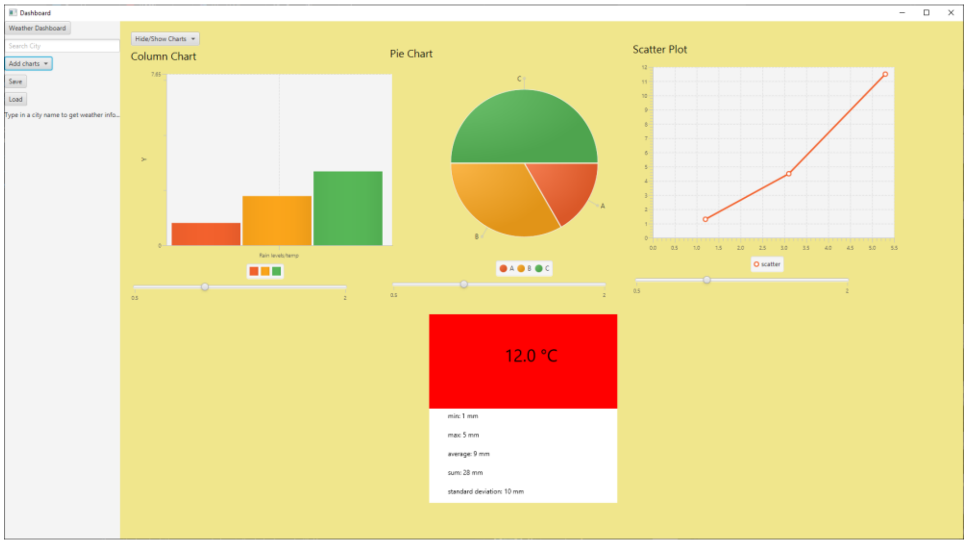
So how do you use the program? To launch the program, we go to the class Dashboard located in the dashboard.GUI package. When the program launches for the first time, you get an empty dashboard screen, like this:



From the left side you can input a city name, you can add various components such as scatter plot, column chart, pie chart and card component. Under that we can save the current state of the dashboard and its data with the button 'Save' and likewise we can load this data that we saved with the button 'Load', under the buttons we have the current date and city of our data.

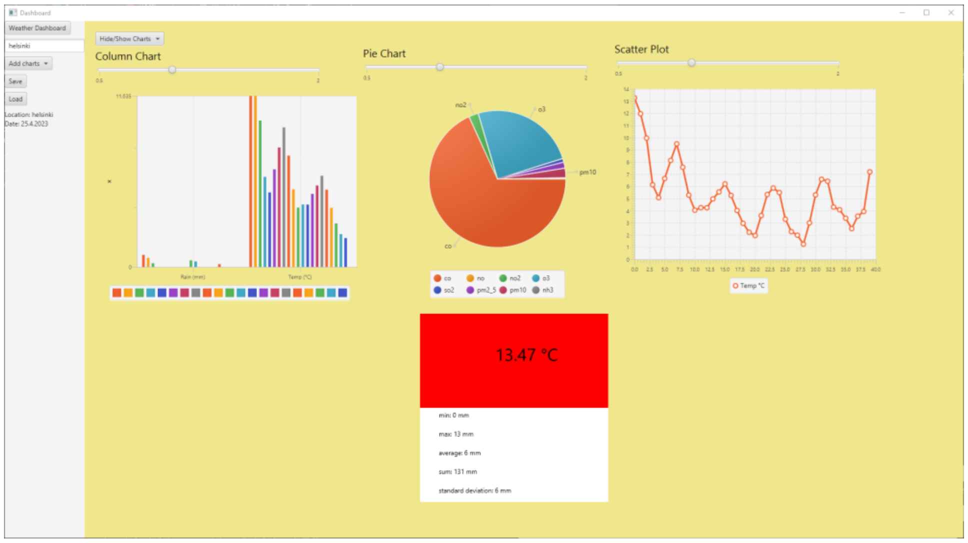
Let's start by adding charts with the 'add charts' button.

We can move around the components to look like this:



We notice that each chart has a slider, with this you can adjust the sizes of the charts. Also, if we right click the charts we can duplicate/remove them. From the button on the top of the components we can hide/show them. We cannot duplicate the card component since I didn't really think it is necessary, but you can still remove them.

Next let's input a city name, to get its data for this example let's type in 'helsinki' into the search city text field and hit enter:



Location: helsinki
Date: 25.4.2023

13.47 °C

min: 0 mm
max: 13 mm
average: 6 mm
sum: 131 mm
standard deviation: 6 mm

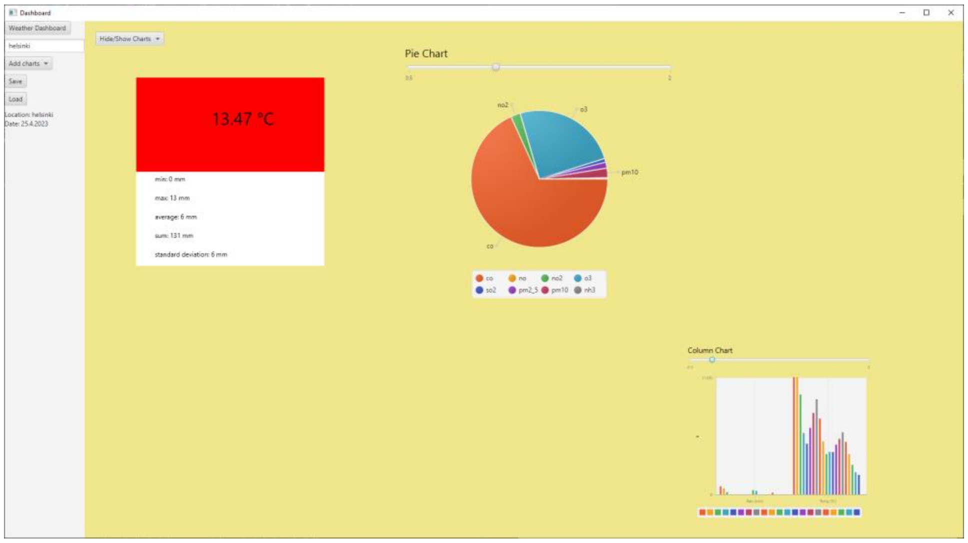
We immediately get the weather data from Helsinki. You can also notice that under the ‘Load’ button on the left side the text updates to show the current city and the current data’s date.

Let’s see what each component does:

The column chart on the top contains the rainfall levels and the temperature of the same date and time. The pie chart displays air pollution levels. The scatter plot displays temperature of 5 days in 3-hour intervals. And finally, the card component displays the current temperature on the top and in the bottom, it calculates 5-day rainfall data’s max/min/average/sum/standard deviation. The card's top part changes color if it is over 0 degrees to red and if it is less than or equal to 0 degrees it changes to blue.

Next let’s try the different things that you can edit in the dashboard. With the slider on each chart component, we can resize them, by right clicking we can duplicate them and with the button hide/show we can hide/show them. Let’s resize the column chart to be smaller and let’s remove the scatterplot that visualizes our temperature on different times of the day. We can also move around the components to different places of the UI like I stated before, let’s move the card component to the top left and column to the bottom right-

Our dashboard will now look like this:

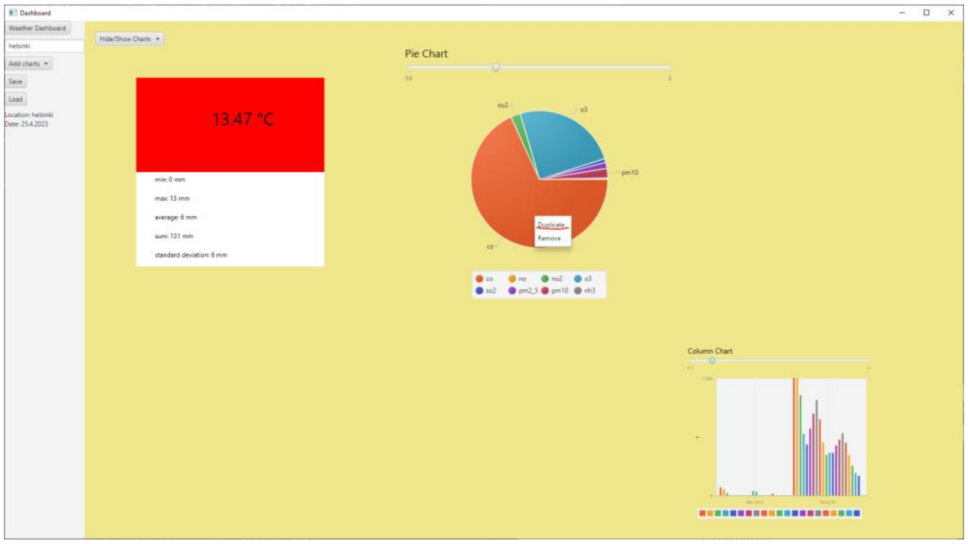


Location: helsinki
Date: 25.4.2023

13.47 °C

min: 0 mm
max: 13 mm
average: 6 mm
sum: 131 mm
standard deviation: 6 mm

We can duplicate each chart with right clicking let’s try that on our pie chart like this:

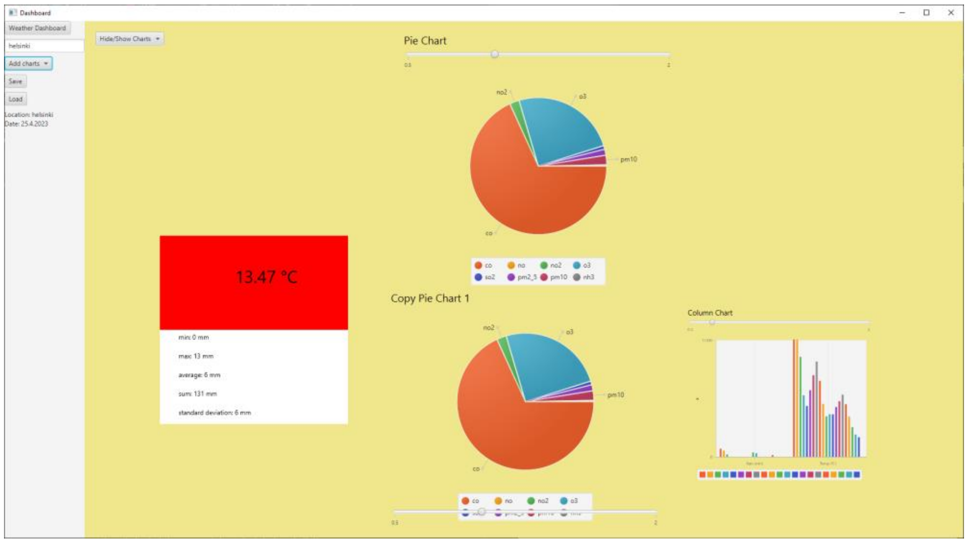


Location: helsinki
Date: 25.4.2023

13.47 °C

min: 0 mm
max: 13 mm
average: 6 mm
sum: 131 mm
standard deviation: 6 mm

And then:

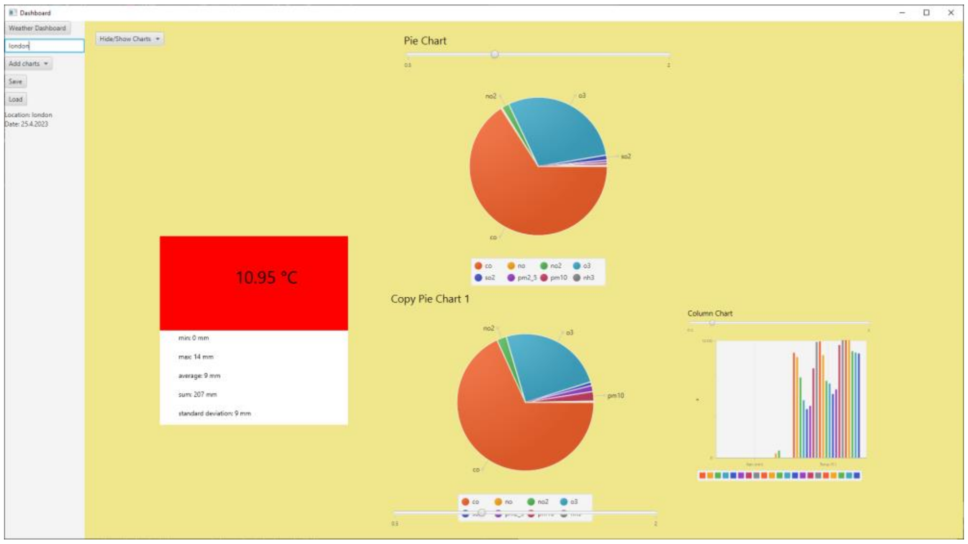


Location: helsinki
Date: 25.4.2023

13.47 °C

min: 0 mm
max: 13 mm
average: 6 mm
sum: 131 mm
standard deviation: 6 mm

Let’s try to search the weather data from ‘London’:



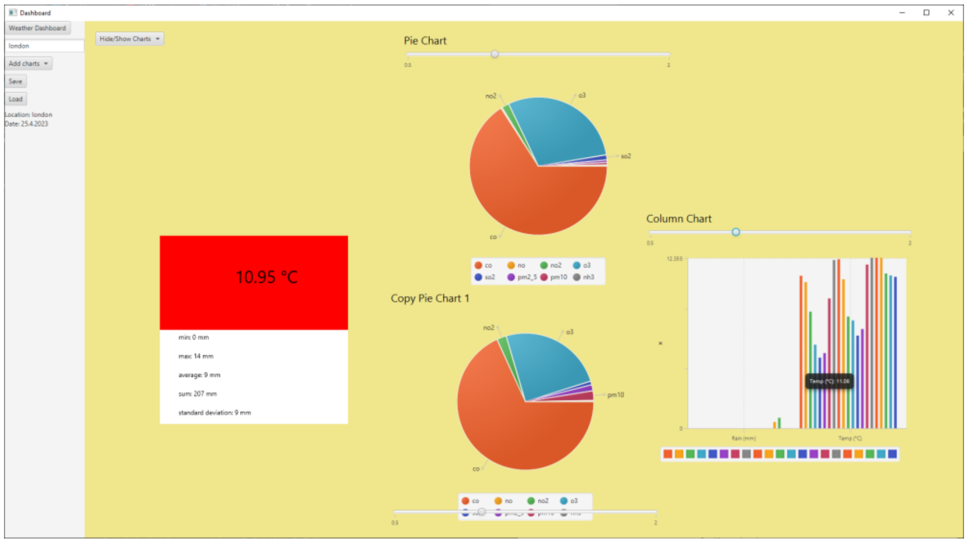
Location: london
Date: 25.4.2023

10.95 °C

min: 0 mm
max: 14 mm
average: 9 mm
sum: 207 mm
standard deviation: 9 mm

We notice that the pie chart duplicate from Helsinki stays the same, but the ‘parent’ component changes. This is all intended since I thought if we can duplicate a chart, it is to compare weather data between two cities.

One other neat thing that each component has is the tooltips, for example reading the data on the column chart could be a bit hard, to quickly read one data points actual data we can hover over it and a tooltip will show up. Let's try it on our column chart, we get this (every component has this except the card):

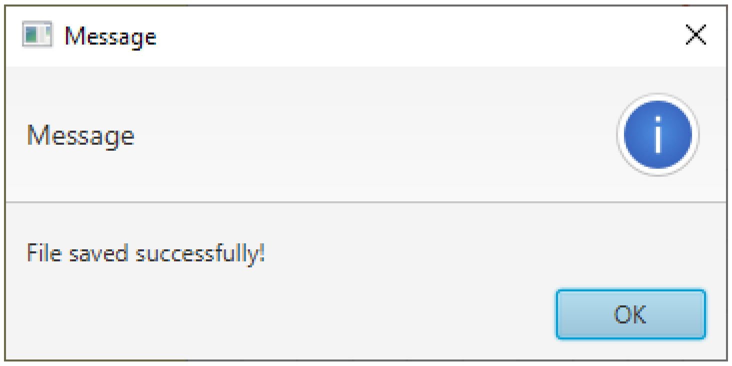


Location: london
Date: 25.4.2023

10.95 °C

min: 0 mm
max: 14 mm
average: 9 mm
sum: 207 mm
standard deviation: 9 mm

You can also delete the duplicated chart components, let’s try that add back our scatter plot and make a save of the dashboard with the button ‘Save’:



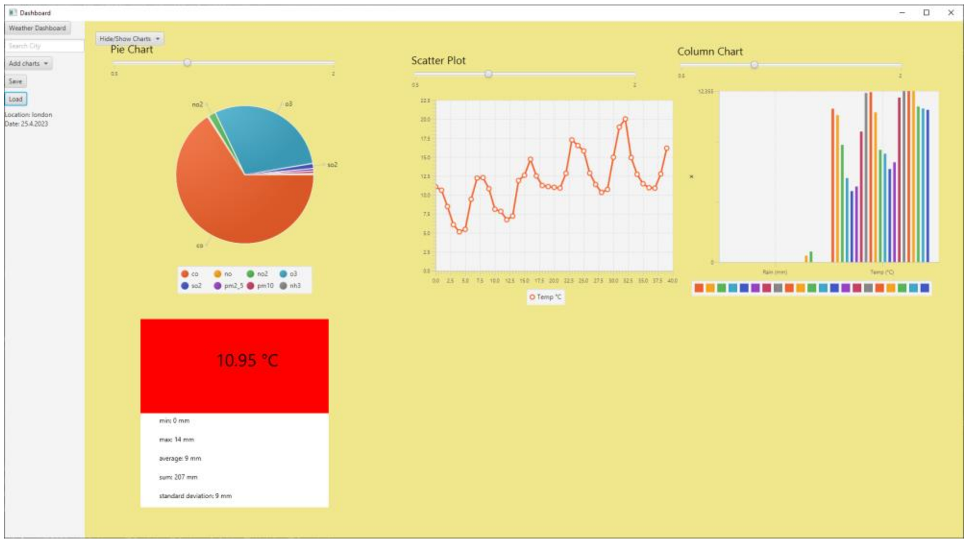
Message

Message

File saved successfully!

Once we save the dashboard, we get a prompt like this, that tells us if everything got saved successfully.

Now if we close our program and relaunch it and press the button ‘Load’:



The dashboard will look like this (Disclaimer: when I added back in the scatterplot I moved everything around to fit nicely, the save button also saves the positions of the components).

This is how the program works in general, and how to use it.

4. Program structure

My program structure changed almost completely from the technical plan. The program is split into two packages. Dashboard.GUI and backend.API packages. The dashboard.GUI contains the main component classes and the actual program (UI), backend.API contains classes for the API, that get different data from it. Dasboard.GUI also is the class that writes the saves data etc.

Let's start off by looking at the different components that draw the charts and the card.

4.1 Components Trait, Card Class, ColumnChart Class, Pie Class and ScatterPlot Class

Components class contains an abstract method component that creates a Pane object, that each chart/plot/card class contains. The Card class takes as its parameters Seq[Seq[Double] (here the rain data) and a temp that is a double. From the list it calculates the min, max, avrg, sum and standard deviation. After that it builds the "picture" from these calculations and puts them on a Rectangle object.

The ColumnChart class takes two lists as it's parameters a dataSet: Seq[Seq[Double]] and dataSetx: ObservableBuffer[String], from these two lists the class builds different XYChart.Data points and puts them into a XYChart.Series that then the column chart is drawn with these data points using ScalaFX built in BarChart object, and again this is done in the method component.

The Pie Class takes two lists as well as it's parameters here a Seq[Int] and Seq[String]. The functionality is the same as the ColumnChart class but instead of XYChart.Data points we create PieChart.Data points and again from these data points we draw the chart using the component method, and with ScalaFX object PieChart we draw the actual pie chart.

The ScatterPlot class takes one parameter a dataSet: Seq[(Double, Double)]. The structure of the class is almost indentical to the ColumnChart class, we put the data into a XYChart.Data and then into XYChart.Series as a whole and draw the scatter plot using our component method and the ScalaFX built in LineChart object.

4.2 Dashboard Class

This class is basically our main class for the whole program, since it gets all the data from our backend and then puts them into lists that are then drawn using the above classes, the data is also saved to files and loaded from files in this class as well. The class also handles all the user input etc. So the Dashboard class is connected to every single other class of the program structure.

Each chart is put into a VBox, that contains the chart itself, a title and a slider to adjust the size of the box. These chart 'boxes' are then manually added to the dashboard's GridPane variable when the user adds them in the UI. The first scatterPlotBox contains comments that explain what each part does, so it's better to look at them instead of explaining them here. After we create the charts, we create the cardBox. The difference between the cardBox and the 'chart' boxes is that you cannot duplicate it and you cannot adjust its size (all of this is intentional), since I didn't really think it is necessary. After all of the component boxes we have a variable that changes the nodes position when you click and drag them, there is a short explanation in the comments of the code. Everything else is outlined in the code.

Let's now look at the backend.API:

4.3 DateFormat Class

This class is fairly small, it contains case classes for the most common keys that the JSON from the API contains. I've also implemented three functions that transform the given Long integer into hours, weekday or date.

4.4 Forecast Class

This class contains case classes that have the keys/types that the JSON from the API contains. Basically, what we do is match the names and the types of the values with the ones in the API call, once we do this, we decode the data. In this class we have to methods our getForeData that takes a place as it's parameter, and getForecastData that takes coordinates (lattitude and longitude) as it's parameters. We use two functions inside these functions that are called from the API class that we will look into in the end.

4.5 WeatherData Class

The structure of this class is almost exactly the same as Forecast data, but instead of a list of different weather components, we only get a single one. The methods are the same as well.

4.6 AirPollution Class

This class is small as well, since the API call only contains two components, where the second component is a list of three components. The methods in this class are the same as the other weather classes.

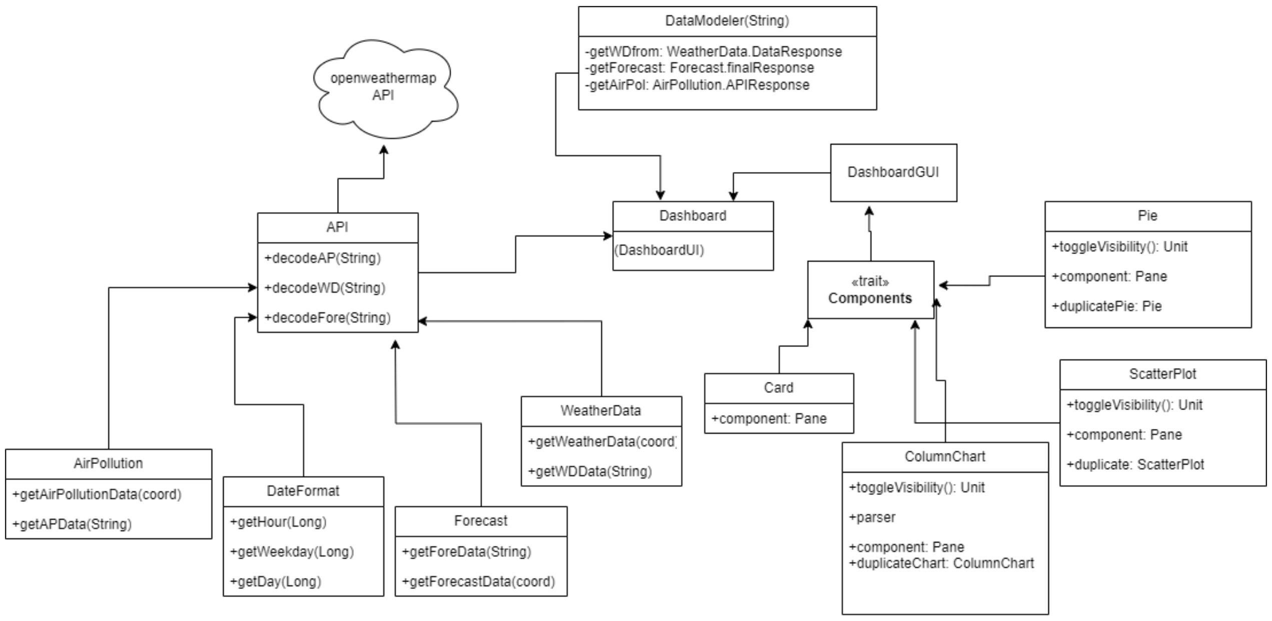
4.7 API Class

Finally let's look at the API class in the backend.API, this class contains all the parsing methods. To parse through the JSON file, we're using Circe library. The methods are commented on in the code so no further explanations are required.

4.7 UML diagram of the program structure

This differs a little bit from the technical plan:

….



...

5. Algorithms

The algorithms I use in my program aren't really that advanced, the most common and the most important algorithms are the first lines of code in each components class. These methods or variables take the data from the parameters that the class gets and turns them into data that can be use with the ScalaFX chart components (like the XYchart.Series). I wrote in the technical plan "For my project I will be implementing a few algorithms to perform the needed calculations and sort data. One of the algorithms will work as a data collection, it collects the needed data such as the air quality, temperature and rainfall levels into a list. After collecting the data, it is then moved to the data processing algorithm which then performs the necessary calculations such as the mean, maximum and minimum and standard deviation. The formulas we will use are in order" and for the most part this is what I have.

6. Data structures

The data structures I use in this program consist of different Sequences. There isn't really a good reason I am using sequences. But the structure of some of the sequences are Seq[Seq[Double]] since I have data from different times and by doing it this way I can separate them. Other than that, there isn't anything fancy about them.

7. Files and Internet access

The program deals with two types of files, text file and JSON files. The text file stores the API key apieKey.txt from openweatherdata.org. The JSON file is the file that the program fetches from the API, and we decode this data using Circe library and with Scala case classes.

The program accesses the internet to fetch data from the API using the HTTP/HTTPS protocol. We use the Source.fromURL method from the scala.io package to read data from the URLs. If we have a connection failure, we have a Alert that displays the error message. Everything will work fine if the apiKey.txt file contains our API key, then we can fetch data from the API and display it in the GUI.

The second JSON file is created in the class dashboard. The dashboard contains different data, components, state of the components and the locations of these components. With the save methods created in the program a JSON file is created where we store all of this data. The structure of the consists of case classes and matching key value pairs and writing them into the file, and correspondingly reading from it. If reading differs in any way let's say if we purposefully break the JSON file we get an error. The dashboard JSON is only created once the user clicks the 'save' button in the UI, but after that we can load from it freely.

8. Testing

During the creation of the program the testing was a bit rough. In many cases I tested my code by printing the values to console. But for the most part I used the GUI to test since the program leans heavily towards the visual side. Honestly, the testing process could've been a lot better. I used the debugger tool a lot in the backend.API package, because I had a lot of problems with reading the JSON (in many cases in the match-case structures I got a Failure, with the debugger tool I could check what went wrong and by doing so could find the problems). I also made sure that the program doesn't load corrupt JSON files that are saved in the root, I did this by changing the keys in the JSON and made sure every key is unique in the case class structure in the dashboard class.

9. Known bugs and missing features

One of the noticeable bugs happens when you add components to the dashboard, once we add all of them, we notice they move each other and when they do they move each other's sliders, so they're not aligned in the middle anymore. This could easily be fixed by making the different components not overlap, perhaps using setSpacing methods or insets built in VBox and scalaFX, or putting the different VBoxes into their own Rectangle objects.

This isn't really a bug but just a missing feature, but when we duplicate a chart and save the state of the dashboard to a file, and we reopen and load the file the duplicated chart is not displayed. I didn't really have enough time to complete this, but I could've done it easily by saving the duplicate chart components data to a list, and once I save the dashboard state, I save the list in the file as well. I would then just load up the duplicated charts similarly like I did in the other chart components.

One time when I loaded the dashboard from file, the pie chart got added on top of the hide/show button and it moved it down. But I've failed to recreate it ever since, and I really have no idea why it happened.

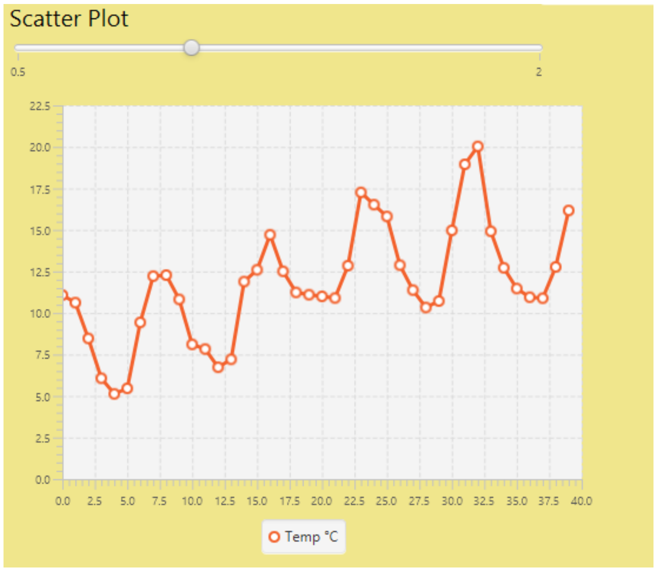
If I am being honest, I think I fixed most of the bugs with variables, conditions and Alerts. But like I said there could be some bugs appearing out of nowhere and just disappearing.

This isn't really a bug as well but when you type the city name wrong into the text field you get an error in the console. I added an alert to notify you if you've done so.

When you start up the program for the first time and you add charts, then search for weather data from a city. You can notice that the slider under the chart component moves from the bottom to the top under the label, this is actually intended since the VBox children are in a list, and when you search for weather data from another city it replaces the last chart element with the new one (in the middle of the list; index 1). So we remove the old chart from the list, and we add the new one as the last index of the list, this pushes the slider to the middle of that list, putting it under the label. I've added conditions for this, so that we don't accidentally replace any other children of the VBox.

When you type in a city the sidebar shrinks, this is since the sidebar (on the left) doesn't have a set size, but changes according to the length of the text. It shrinks since the text that displays the data and current city is shorter length than the first text. This isn't really a bug that breaks to program but can be fixed by setting the width of the sidebar.

On the scatter plot component, the X-axis is not explained or doesn't have anything logical to it.



The X-axis represents 5 days with 3-hour steps. And like the legend says the Y axis represents the temperature.

I could've added something in the form of (1.1,1.2,1.3...5.3) to represent the days and the hours.

The program is missing multiple-series scatter plot, and unfortunately I didn't have a lot of time to make it. But if I had time, it would've been as easy as creating a second XYChart.Series class and adding it to my chart since now I only have one and I would have a multiple-series scatter plot.

I am also missing the ability to click on a single data point and show more information about the data point, this could've easily been done by adding a Rectangle object that contains information once a data point is clicked. I already have tooltips so with the same concept this could be easily done, but due to time constraints I don’t have it implemented.

I am also partially missing specifying colors for individual components/data points/axis, since now I have specific colors for each data point for example on the pie chart but there isn’t an ability to change them in the GUI. This could easily be done with ScalaFX chart component built in methods, and creating a context menu buttons for each data point on the axis/component.

10. 3 best sides and 3 weaknesses

So what are the 3 best sides and 3 weaknesses?

I would say that the first best side of the program is that it is flawless in terms of updating the data when you search for a new location. The other best side is saving the dashboard to a file, not considering that you cannot save the duplicate components. I think that loading from the saved file is flawless and only has a few bugs that occur sometimes. The last best side of the program is its easability, the UI layout is simple to understand and doesn’t really require that much effort to learn.

The first weakness I would say is that we cannot save to file the whole state of the dashboard, like I said before the duplicate chart components and their data cannot be saved. Another weakness is that the components can overlap, this can cause a few problems such as misplacement of other components such as buttons when we add a chart/card component.

The last weakness I can think of is that some of the charts contain useless data, for example in the pie chart we have air pollution levels of different gases. But this data is not in percentages that you would like to have on a pie chart, but instead pollutant concentration in µg/m^3. This isn't really a major weakness but just something that I wanted to point out.

A bonus weakness is that the code is bloated, there is a lot of useless stuff that could be coded better.

11. Deviations from the plan, realized process and schedule

I deviated from the initial technical plan a lot. I started creating the project a month after the initial plan, so about a month ago. But the steps of creating the project were about the same as in the technical plan.

This is what I wrote in the technical plan:

“By 6.3. I would implement all the components for the user interface, I will also have the Data class ready with the required methods for parsing through the data.

By 20.3. I would implement additional UI elements such as the hovering over and removing Components methods, I will also implement the GET methods for different locations.

By 4.4 I would have implemented the complete program, during the next few weeks I will keep refining the program and test it to make sure there aren’t any errors or bugs hidden in the code.

Honestly my schedule would be a little more flexible, but I will try to stay on a fixed schedule.”

The time estimate to create the project was close to reality. But it was bad to procrastinate and start creating the project so late, since I could've missed the deadline if I got stuck on a problem. But I learned that creating something to visualize first like the UI, then creating the backend/algorithms makes the whole process easier since you see what you're working on.

12. Final evaluation

To summarize, most of my problems was with the components etc. And I would say most of it comes down to the class dashboard, since I get the data from the backend.API there and I display it. I think it would've been smarter to implement the different chart "sections" in their own classes where they would get the data from the backend.API, and then display them. This would've made duplicating the charts a lot easier since I could've made a class inside the class that let's me duplicate each chart component. By doing it I could then store the duplicate charts classes data in the files easily, and also, I could've made it so each chart class is in their own ScalaFX Pane so they don't overlap. These small changes would've made the program structure a lot better and made the program function as intended.

Another thing which is terrible with the class dashboard is readability. Since everything is created there, it is hard to find bugs and implement new stuff, and doing the above changes would've made it easier to read and extend.

But all in all, everything works as intended and there are few little bugs here and there. I would say the things that were required to implement that I did implement, work perfectly fine and are executed as intended.

13. References

Most of my information came from YouTube and stackoverflow.com. So, I might be missing some stuff but here they are:

https://fi.wikipedia.org/wiki/MVC-arkkitehtuuri

https://www.playframework.com/documentation/2.8.x/ScalaHome

https://circe.github.io/circe/

https://openweathermap.org/api

https://gist.github.com/navicore/7973711f300f00f9d878026eaf84bed2

https://www.youtube.com/watch?v=GDtBbBYgvoQ

https://www.youtube.com/watch?v=uCtKqp2tdrg&t=1983s

https://stackoverflow.com/questions/46818766/scalafx-create-linechart-in-scala

https://stackoverflow.com/questions/36023189/scalafx-javafx-stage-content-does-not-resize-on-window-resize

https://stackoverflow.com/questions/68114411/hbox-not-resizing-in-scalafx

ScalaFX documentation obviously.

https://github.com/scalafx/scalafx/blob/master/scalafx-demos/src/main/scala/scalafx/scene/chart/ScatterChartDemo.scala

https://github.com/scalafx/scalafx-ensemble