§

Weer App

Jordy Neefs



Table of Contents

[Inleiding 2](#_Toc156257152)

[Functionele beschijving 3](#_Toc156257153)

[Home pagina 3](#_Toc156257154)

[Uurrooster 4](#_Toc156257155)

[Het weer 5](#_Toc156257156)

[Technische opbouw 6](#_Toc156257157)

[Rooster API 6](#_Toc156257158)

[Weather App 9](#_Toc156257159)

[LocationUtils 9](#_Toc156257160)

[Utils 10](#_Toc156257161)

[Schedule Api 11](#_Toc156257162)

[WeatherApi 13](#_Toc156257163)

[Navigatie 15](#_Toc156257164)

[MainActivity 16](#_Toc156257165)

[Home 17](#_Toc156257166)

[Weather 22](#_Toc156257167)

# Inleiding

Fietsen naar school is niet alleen een milieuvriendelijke keuze, maar het biedt ook tal van voordelen voor je gezondheid. Echter, we begrijpen allemaal dat niemand graag door de regen fietst. Om dit dagelijkse dilemma aan te pakken, heb ik besloten een app te ontwikkelen die het gemakkelijk maakt voor fietsers zoals jij om te weten of het tijdens je fietstocht naar school gaat regenen of niet.

Het doel van deze app is om fietsers te helpen bij het plannen van hun dagelijkse fietstochten door realtime weersinformatie te verstrekken. De focus ligt specifiek op het vermijden van regen tijdens het fietsen, waardoor gebruikers comfortabel en voorbereid naar school kunnen gaan. Deze app richt zich op studenten en iedereen die regelmatig met de fiets naar hun bestemming reist.

# Functionele beschijving

## Home pagina

De homepage van de app houdt het simpel en duidelijk. Zodra je de app opent, zie je meteen of je droog aankomt op de fiets of dat je beter de auto kunt pakken. Met de melding "It's not going to rain!" of "It's going to rain!", weet je direct waar je aan toe bent. En voor de zekerheid toont de app ook nog het beste vervoermiddel voor de dag, met een icoontje van een fiets of auto.

Onder deze weervoorspelling vind je meteen de tijden dat je les hebt, bijvoorbeeld "Your lessons go from 08:30 to 12:30". Zo zie je in één oogopslag van hoe laat tot hoe laat je les hebt. Het idee is om je ochtendroutine wat makkelijker te maken door alle nodige info op een plek te verzamelen, zodat je snel en zonder gedoe je dag kunt beginnen.

A blue screen with a black bicycle and white text

Description automatically generated

## Uurrooster

Op de "Schedule" pagina van de app krijg je snel een helder overzicht van je dagelijkse lesrooster. Het toont de start- en eindtijd van elke les, samen met de vaknaam of cursuscode, zoals "QM1682 Interdisciplinair project" of "QM1323 Soft skills". Het strakke, minimalistische design maakt het makkelijk om te zien wat je volgende klas is en hoe je dag eruitziet. Deze pagina is superhandig om in één oogopslag je schooldag te plannen. Plus, het past mooi bij de rest van de app, waardoor schakelen tussen weervoorspellingen en je rooster naadloos gaat.

A screenshot of a phone

Description automatically generated

## Het weer

Op de "Weather" pagina van de app vind je alles wat je moet weten over het weer. Bovenaan staat de huidige temperatuur groot in beeld, met een bijpassend weericoon, zodat je in één oogopslag de sfeer van de dag pakt. Daaronder staat je locatie om aan te duiden waar deze voorspelling voor is.

Onder de actuele temperatuur is er een sectie voor de uurlijkse voorspelling, handig voor als je je afvraagt of je je paraplu moet pakken voor je tripje naar buiten. Daaronder geeft de app een overzicht van de dagelijkse voorspellingen, met temperaturen en weersymbolen voor de komende dagen. Zo weet je precies wanneer je die barbecue het beste kunt plannen of wanneer het een binnenblijfdagje wordt.



# Technische opbouw

## Rooster API

Deze Python API is ontworpen om het uurrooster van studenten dynamisch op te halen van een iCalendar URL en dit vervolgens beschikbaar te maken via een webservice. Het maakt gebruik van de Flask web framework, wat een lichtgewicht en makkelijk te gebruiken tool is voor het creëren van webapplicaties in Python.

Het cache-systeem is geïmplementeerd om de efficiëntie te verhogen en onnodige verzoeken aan de iCalendar URL te vermijden. Dit is een eenvoudige Python dictionary die twee sleutels bevat: 'last\_update', om bij te houden wanneer de data voor het laatst is opgehaald, en 'data', om de daadwerkelijk opgehaalde evenementen op te slaan.

cache = {

    'last\_update': None,

    'data': None

}

def get\_cached\_events():

    now = datetime.now()

    if cache['last\_update'] is None or (now - cache['last\_update']) > timedelta(hours=24):

        url = 'https://rooster.ucll.be/ical/student/r0880665/hash/3db109d6d3d8002e4834232220980a09c33b966818c6e4ac9fe2916ff2e77bc4ad541c67c12d82590e64fee968349819afe7dd13083f11543ff11d548f20c369'

        cache['data'] = parse\_ics\_from\_url(url)

        cache['last\_update'] = now

    return cache['data']

De get\_cached\_events functie controleert of de opgeslagen data in de cache verouderd is (ouder dan 24 uur). Zo ja, dan haalt het de nieuwe gegevens op door parse\_ics\_from\_url aan te roepen met de gegeven iCalendar URL. Deze functie is een belangrijk onderdeel van de app omdat het zorgt voor een snelle laadtijd van de roosterdata door het vermijden van herhaalde verzoeken naar dezelfde bron.

De parse\_ics\_from\_url functie haalt data op van een gegeven URL, die verwijst naar een iCalendar-bestand. Met behulp van de requests bibliotheek wordt een GET-verzoek naar de URL gestuurd en de inhoud van de respons wordt omgezet in een Calendar object met behulp van de ics bibliotheek. Vervolgens worden de gebeurtenissen uit deze kalender geparsed en in een lijst van dictionaries geplaatst met de sleutels 'start', 'end' en 'summary' om respectievelijk de start- en eindtijden en de samenvatting van elk evenement op te slaan.

def parse\_ics\_from\_url(url):

    response = requests.get(url)

    response.raise\_for\_status()

    calendar = Calendar(response.text)

    events = []

    for event in calendar.events:

        events.append({

            'start': event.begin.datetime,

            'end': event.end.datetime,

            'summary': event.name

        })

    return events

De filter\_events\_by\_date functie filtert de opgehaalde evenementen op een specifieke datum. Dit maakt het mogelijk voor gebruikers om het uurrooster van een bepaalde dag op te vragen.

def filter\_events\_by\_date(events, date):

    return [event for event in events if event['start'].date() == date]

De get\_extreme\_times functie bepaalt de vroegste starttijd en laatste eindtijd uit een lijst van evenementen. Dit is nuttig om snel een overzicht te krijgen van de totale tijdsspanne van de lessen op een dag.

def get\_extreme\_times(events):

    if not events:

        return None

    start\_times = [event['start'].time() for event in events]

    end\_times = [event['end'].time() for event in events]

    return {

        'begin\_time': min(start\_times).strftime("%H:%M"),

        'end\_time': max(end\_times).strftime("%H:%M")

    }

De /lessons route is het toegangspunt van de API. Wanneer een GET-verzoek wordt gemaakt naar deze route, worden de evenementen voor de opgegeven datum (of de huidige datum als er geen is opgegeven) gefilterd en geretourneerd in een gestructureerd JSON-formaat samen met de extreme tijden. Dit maakt het voor de gebruiker mogelijk om via een eenvoudige HTTP-aanroep de lessen voor een specifieke dag op te vragen.

@app.route('/lessons')

def lessons():

    all\_events = get\_cached\_events()

    date\_str = request.args.get('date')

    date = datetime.now(timezone('Europe/Brussels')).date() if not date\_str else datetime.strptime(date\_str, '%Y-%m-%d').date()

    events = filter\_events\_by\_date(all\_events, date)

    lessons = [

        {

            "begin": event['start'].strftime("%H:%M"),

            "date": event['start'].strftime("%Y-%m-%d"),

            "end": event['end'].strftime("%H:%M"),

            "name": event['summary']

        }

        for event in events

    ]

    extreme\_times = get\_extreme\_times(events)

    return jsonify({

        "lessons": lessons,

        "extreme\_times": extreme\_times

    })

Tot slot, het onderstaande blok code is het standaardpatroon om een Flask applicatie te starten. Wanneer dit script direct wordt uitgevoerd, zal de Flask applicatie starten in debugmodus. Dit betekent dat het draait op een lokale ontwikkelserver en automatisch herstart bij wijzigingen in de code, wat handig is tijdens de ontwikkelingsfase.

if \_\_name\_\_ == '\_\_main\_\_':

    app.run(debug=True)

## Weather App

### LocationUtils

De LocationUtils klasse is een onderdeel van de weerapplicatie, verantwoordelijk voor het verkrijgen van de locatiegegevens van de gebruiker. Deze informatie is cruciaal om locatiespecifieke weersvoorspellingen te verstrekken.

De checkLocationPermission methode controleert of de app de nodige locatiepermissies heeft gekregen van de gebruiker. Zonder deze permissies kan de app geen locatiegegevens ophalen.

fun checkLocationPermission(): Boolean {  
 return (  
 ContextCompat.checkSelfPermission(  
 context,  
 Manifest.permission.*ACCESS\_COARSE\_LOCATION* ) == PackageManager.*PERMISSION\_GRANTED* &&  
 ContextCompat.checkSelfPermission(  
 context,  
 Manifest.permission.*ACCESS\_FINE\_LOCATION* ) == PackageManager.*PERMISSION\_GRANTED* )  
}

Indien de nodige locatiepermissies nog niet zijn verleend, faciliteert de requestLocationPermission methode het aanvraagproces door de gebruiker een dialoogvenster te tonen waarin om toestemming wordt gevraagd.

fun requestLocationPermission(requestLocationPermission: (Boolean) -> Unit) {  
 requestPermissionLauncher.launch(  
 *arrayOf*(  
 Manifest.permission.*ACCESS\_COARSE\_LOCATION*,  
 Manifest.permission.*ACCESS\_FINE\_LOCATION* )  
 )  
}

Zodra de locatiepermissies zijn verleend, haalt de getLastLocation methode de meest recente locatie op. Deze gegevens worden vervolgens gebruikt om het weer te voorspellen op de huidige locatie van de gebruiker.

fun getLastLocation(onSuccess: (Double, Double) -> Unit) {  
 if (checkLocationPermission()) {  
 fusedLocationClient.*lastLocation*.addOnSuccessListener **{** location **->** if (location != null) {  
 onSuccess(location.*latitude*, location.*longitude*)  
 }  
 **}** }  
}

Door gebruik te maken van de FusedLocationProviderClient, biedt de LocationUtils klasse een geïntegreerde aanpak voor locatiebepaling, waarbij gebruik wordt gemaakt van meerdere bronnen om de locatie nauwkeurig en efficiënt te

private val fusedLocationClient: FusedLocationProviderClient =  
 LocationServices.getFusedLocationProviderClient(context)

De LocationUtils klasse maakt gebruik van de FusedLocationProviderClient uit de Google Play Services voor locatiebepaling. Deze client biedt een simpele interface voor het verkrijgen van de huidige locatie en is geoptimaliseerd om een balans te vinden tussen nauwkeurigheid, snelheid en energieverbruik.

### Utils

De Utils klasse biedt een set van hulpmethoden die worden gebruikt om datum- en tijdgegevens te verwerken binnen de weerapplicatie.

De getShortDayFromDayItem methode converteert een datum uit een ForecastdayItem naar een korte dagnotatie zoals 'Ma', 'Di', enz. Dit is handig voor het weergeven van weersvoorspellingen waarbij alleen de dag van de week van belang is.

fun getShortDayFromDayItem(dayItem: ForecastdayItem): String? {  
 val inputFormat = SimpleDateFormat("yyyy-MM-dd", Locale.getDefault())  
 val date = dayItem.date?.*let* **{** inputFormat.parse(**it**) **}** val outputFormat = SimpleDateFormat("EE", Locale.getDefault())  
 return date?.*let* **{** outputFormat.format(**it**) **}**}

Met getHourOnlyFromHourItem wordt de specifieke tijd, zoals '15:00', geëxtraheerd uit een HourItem object. Dit maakt het mogelijk om in de app enkel de relevante uurinformatie te tonen voor uurlijkse weersvoorspellingen.

fun getHourOnlyFromHourItem(hourItem: HourItem): String? {  
 val inputFormat = SimpleDateFormat("yyyy-MM-dd HH:mm", Locale.getDefault())  
 val outputFormat = SimpleDateFormat("HH:mm", Locale.getDefault())  
  
 val date = hourItem.time?.*let* **{** inputFormat.parse(**it**) **}** return date?.*let* **{** outputFormat.format(**it**) **}**}

extractHourFromStingTime neemt een tijd als string, bijvoorbeeld '08:30', en haalt hieruit het uur als een integer. Dit is bijzonder nuttig voor functies binnen de app die logica uitvoeren gebaseerd op het uur van de dag, zoals bepalen of het tijdens bepaalde lesuren gaat regenen.

fun extractHourFromStingTime(time: String): Int {  
 val hour = time.*split*(":")[0]  
 return hour.*toInt*()  
}

### Schedule Api

#### ScheduleApiConfig

De ScheduleApiConfig klasse fungeert als een configuratielaag voor het netwerkgedeelte van de app dat verantwoordelijk is voor het communiceren met de rooster-API.

class ScheduleApiConfig {  
 companion object {  
 fun getLessonsApiService(): ScheduleApiService {  
 }  
 }  
}

Een HttpLoggingInterceptor wordt toegevoegd aan de OkHttp-client om de details van de netwerkverzoeken en -responsen in de logs te zetten. Dit is vooral handig voor ontwikkelaars om snel netwerkgerelateerde problemen te identificeren en op te lossen.

val loggingInterceptor = HttpLoggingInterceptor()  
 .setLevel(HttpLoggingInterceptor.Level.*BODY*)

De OkHttp client is de motor onder de motorkap voor het maken van HTTP-verzoeken. In deze code wordt de client geconfigureerd met een loggingInterceptor. Dit is een handige tool tijdens de ontwikkeling omdat het alle inkomende en uitgaande HTTP-verzoeken logt, waardoor je een duidelijk inzicht krijgt in de netwerkactiviteit van je app.

val client = OkHttpClient.Builder()  
 .addInterceptor(loggingInterceptor)  
 .build()

Retrofit wordt gebruikt om een type-safe HTTP client voor je app te creëren. In deze configuratie:

* baseUrl: Stelt de basis URL in voor de API. Alle verzoeken vanuit de app gebruiken deze URL als uitgangspunt.
* addConverterFactory: Specificeert de GsonConverterFactory die Retrofit zal gebruiken om JSON te converteren naar Kotlin-objecten. Gson handelt de serialisatie en deserialisatie van de verzoeken en responsen af.
* client: Voegt de OkHttp client toe aan Retrofit. Dit betekent dat alle verzoeken die door Retrofit worden gemaakt, gebruik maken van de eerder geconfigureerde OkHttp client.

val retrofit = Retrofit.Builder()  
 .baseUrl("https://mobieleappsroosterapi.azurewebsites.net/")  
 .addConverterFactory(GsonConverterFactory.create())  
 .client(client)  
 .build()

Tot slot wordt de Retrofit builder gebruikt om een implementatie van de ScheduleApiService interface te creëren. Deze interface bevat methoden die overeenkomen met de verschillende endpoints van je API. Met Retrofit worden deze methoden omgezet in concrete oproepen naar de webserver, waarbij de resultaten worden verpakt in Call objecten die je kunt gebruiken om asynchrone of synchrone netwerkverzoeken uit te voeren.

return retrofit.create(ScheduleApiService::class.*java*)

#### ScheduleApiServive

De ScheduleApiService interface is een contract tussen de app en de server. Het specificeert de HTTP-verzoeken die door Retrofit worden uitgevoerd om de roosterdata op te halen.

interface ScheduleApiService {  
 @GET("lessons")  
 fun getLessons(  
 @Query("date") date: String,  
 ): Call<ScheduleResponse>  
}

* De @GET("lessons") annotatie verklaart een HTTP GET-verzoek naar het 'lessons' endpoint van de API. Dit is het punt waar de app de roosterinformatie zal opvragen.
* De getLessons methode wordt geassocieerd met de bovengenoemde GET-aanvraag en zal worden aangeroepen wanneer de app de roosterinformatie nodig heeft.
* De @Query("date") annotatie geeft aan dat de methode een query parameter verwacht met de naam 'date'. Retrofit zorgt ervoor dat wanneer de getLessons methode wordt aangeroepen, de gegeven 'date' parameter aan de URL wordt toegevoegd als een query parameter.

### WeatherApi

#### WeatherApiConfig In de weerapplicatie speelt WeatherApiConfig een vergelijkbare rol als ScheduleApiConfig, maar dan gericht op het ophalen van weergerelateerde gegevens. De uitleg is identiek aan de vorige dus ga niet alles uitleggen.

class WeatherApiConfig {  
  
 companion object {  
 fun getApiService(): WeatherApiService {  
  
 // API response interceptor  
 val loggingInterceptor = HttpLoggingInterceptor()  
 .setLevel(HttpLoggingInterceptor.Level.*BODY*)  
  
 // Client  
 val client = OkHttpClient.Builder()  
 .addInterceptor(loggingInterceptor)  
 .build()  
  
 // Retrofit  
 val retrofit = Retrofit.Builder()  
 .baseUrl("https://api.weatherapi.com/v1/")  
 .addConverterFactory(GsonConverterFactory.create())  
 .client(client)  
 .build()  
  
 return retrofit.create(WeatherApiService::class.*java*)  
 }  
  
 const val API\_KEY = "c717e43524db4057858193046230212"  
 }  
}

#### WeatherApiService

interface WeatherApiService {  
 @GET("current.json")  
 fun getCurrentWeather(  
 @Query("key") key: String = WeatherApiConfig.API\_KEY,  
 @Query("q") location: String,  
 @Query("aqi") aqi: String = "no",  
 @Query("alerts") alerts: String = "no",  
 @Query("lang") lang: String = "nl"  
 ): Call<CurrentWeatherResponse>  
  
 @GET("forecast.json")  
 fun getWeatherForecast(  
 @Query("key") key: String = WeatherApiConfig.API\_KEY,  
 @Query("q") location: String,  
 @Query("aqi") aqi: String = "no",  
 @Query("alerts") alerts: String = "no",  
 @Query("lang") lang: String = "nl",  
 @Query("days") days: Int = 7  
 ): Call<WeatherForecastResponse>  
}

* Key: de API key voor de API
* Q: de query parameter voor de locatie
* Aqi : air quality toevoegen of niet
* Alerts: alerts aan of uit zetten
* Lang: de taal
* Days: geef aan hoeveel dagen er in de forecast moeten zitten.

### Navigatie

#### Mobile\_navigation.xml

<?xml version="1.0" encoding="utf-8"?>  
<navigation xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:app="http://schemas.android.com/apk/res-auto"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:id="@+id/mobile\_navigation"  
 app:startDestination="@+id/navigation\_home">  
  
 <fragment  
 android:id="@+id/navigation\_schedule\_list"  
 android:name="com.example.weatherapp.schedule.view.ScheduleListFragment"  
 android:label="@string/title\_schedule"  
 tools:layout="@layout/fragment\_schedule" />  
  
 <fragment  
 android:id="@+id/navigation\_home"  
 android:name="com.example.weatherapp.home.view.HomeFragment"  
 android:label="@string/home\_title"  
 tools:layout="@layout/fragment\_weather\_full" />  
  
 <fragment  
 android:id="@+id/navigation\_weather\_full"  
 android:name="com.example.weatherapp.weather.view.WeatherFullFragment"  
 android:label="@string/title\_weather"  
 tools:layout="@layout/fragment\_weather\_full" />  
  
  
</navigation>

Hier wordt aangegeven welke navigatie-elementen er allemaal zijn. Ik heb 3 navigaties en elk wordt gekoppeld aan een fragment.

#### Bottom\_nav\_menu.xml

<?xml version="1.0" encoding="utf-8"?>  
<menu xmlns:android="http://schemas.android.com/apk/res/android">  
 <item  
 android:id="@+id/navigation\_schedule\_list"  
 android:icon="@drawable/baseline\_format\_list\_bulleted\_24"  
 android:title="@string/schedule\_title" />  
  
 <item  
 android:id="@+id/navigation\_home"  
 android:icon="@drawable/baseline\_home\_24"  
 android:title="@string/home\_title" />  
  
 <item  
 android:id="@+id/navigation\_weather\_full"  
 android:icon="@drawable/baseline\_cloud\_24"  
 android:title="@string/weather\_title" />  
</menu>

Dit is de “Bottom nav” dit is het menu waar je tussen de fragments kunt wisselen. Dit is het deel wat je daadwerkelijk ziet. Elk item is gekoppeld aan een navigatie uit het vorige deel.

### MainActivity

De main activity is zeg maar de wrapper van mijn applicatie.

De main activity bestaat uit 3 onderdelen

#### Bottom navigatie

<com.google.android.material.bottomnavigation.BottomNavigationView  
 android:id="@+id/nav\_view"  
 android:layout\_width="0dp"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginStart="0dp"  
 android:layout\_marginEnd="0dp"  
 android:background="?android:attr/windowBackground"  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintLeft\_toLeftOf="parent"  
 app:layout\_constraintRight\_toRightOf="parent"  
 app:menu="@menu/bottom\_nav\_menu" />

val navView: BottomNavigationView = binding.navView  
  
val navHostFragment = *supportFragmentManager* .findFragmentById(R.id.*nav\_host\_fragment\_activity\_main*) as NavHostFragment  
val navController = navHostFragment.navController  
val appBarConfiguration = *AppBarConfiguration*(  
 *setOf*(  
 R.id.*navigation\_schedule\_list*, R.id.*navigation\_home*, R.id.*navigation\_weather\_full* )  
)  
*setupActionBarWithNavController*(navController, appBarConfiguration)  
  
navView.*setupWithNavController*(navController)

Dit is de navigatie onderaan het scherm. Deze bevat dus de 3 items “Schedule”, “Home” en “Weather”. In de code wordt deze “BottomNavigationView” gekoppeld aan de “nav\_host\_fragment\_activity\_main”, hierin worden de fragments geladen.

#### ProgressBar

<ProgressBar  
 android:id="@+id/progressBar"  
 style="@style/Widget.AppCompat.ProgressBar"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:visibility="gone"  
 app:layout\_constraintTop\_toTopOf="parent"  
 app:layout\_constraintBottom\_toBottomOf="parent"  
 app:layout\_constraintLeft\_toLeftOf="parent"  
 app:layout\_constraintRight\_toRightOf="parent"/>

fun showLoading(isLoading: Boolean) {  
 binding.progressBar.*visibility* = if (isLoading) View.*VISIBLE* else View.*GONE* binding.navHostFragmentActivityMain.*visibility* = if (isLoading) View.*GONE* else View.*VISIBLE*}

De progressbar dient als laadicoon zolang de data nog niet geladen is. Deze wordt in het midden van het scherm getoond. Met de functie “showLoading” kun je het icoontje tonen. Deze functie zorgt ervoor dat het icoon getoond wordt en de rest niet. Als de data geladen is wordt het icoon terug onzichtbaar en het huidige fragment weer zichtbaar.

#### FragmentContainerView

<androidx.fragment.app.FragmentContainerView  
 android:id="@+id/nav\_host\_fragment\_activity\_main"  
 android:name="androidx.navigation.fragment.NavHostFragment"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 app:defaultNavHost="true"  
 app:layout\_constraintBottom\_toTopOf="@id/nav\_view"  
 app:layout\_constraintLeft\_toLeftOf="parent"  
 app:layout\_constraintRight\_toRightOf="parent"  
 app:layout\_constraintTop\_toTopOf="parent"  
 app:navGraph="@navigation/mobile\_navigation" />

Dit is de container waar het fragment in geladen worden.

### Home

Elke view is gelijkaardig opgebouwd. Ik ga eerst de gemeenschappelijke dingen uitleggen zodat ik later hiernaar kan verwijzen:

override fun onCreateView(  
 inflater: LayoutInflater, container: ViewGroup?,  
 savedInstanceState: Bundle?  
): View? {  
 // Inflate the layout for this fragment  
 return inflater.inflate(R.layout.*fragment\_home*, container, false)  
}

De “onCreateView” functie genereerd telkens de juiste instantie van het juiste fragment, in dit geval het “fragment\_home”

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
 super.onViewCreated(view, savedInstanceState)  
  
 binding = DataBindingUtil.bind(view)!!  
 homeViewModel = HomeViewModel()  
 binding.*viewModel* = homeViewModel  
 binding.*lifecycleOwner* = *viewLifecycleOwner*  
  
 subscribe()  
}

In deze functie wordt de databinding opgezet. De “subscribe” functie is een functie waar ik alle observers in steek.

private fun subscribe(){  
 homeViewModel.isLoading.observe(*viewLifecycleOwner*) **{** isLoading **->** (*activity* as? MainActivity)?.showLoading(isLoading)  
 **}** homeViewModel.isError.observe(*viewLifecycleOwner*) **{** isError **->** if (isError) {  
 *context*?.*let* **{** (*activity* as? MainActivity)?.showError(**it**, homeViewModel.errorMessage) **}** }  
 **}**}

Elke view gebruikt “isLoading” en “isError”, deze 2 staan in de mainActivity. De “isLoading” dient voor het laadicoon weer te geven. De “isError” geeft een error toast weer als er een error is. In het ViewModel kun je dan dit doen.

private val \_isLoading = MutableLiveData<Boolean>()  
val isLoading: LiveData<Boolean> get() = \_isLoading  
  
private val \_isError = MutableLiveData<Boolean>()  
val isError: LiveData<Boolean> get() = \_isError  
  
var errorMessage: String = ""  
 private set

fun getStartHour(date: String) {  
 \_isLoading.*value* = true  
 \_isError.*value* = false  
  
 val client = ScheduleApiConfig.getLessonsApiService().getLessons(date = date)  
  
 client.enqueue(object : Callback<ScheduleResponse> {  
 override fun onResponse(  
 call: Call<ScheduleResponse>,  
 response: Response<ScheduleResponse>  
 ) {  
 val responseBody = response.body()  
 if (!response.*isSuccessful* || responseBody == null) {  
 onError("Data Processing Error")  
 return  
 }  
 }

override fun onFailure(call: Call<ScheduleResponse>, t: Throwable) {  
 onError(t.message)  
 t.printStackTrace()  
 }  
 })  
}

private fun onError(inputMessage: String?) {  
 val message = if (inputMessage.*isNullOrBlank*() or inputMessage.*isNullOrEmpty*()) "Unknown Error"  
 else inputMessage  
  
 errorMessage = StringBuilder("ERROR: ")  
 .append("$message some data may not displayed properly").toString()  
  
 \_isError.*value* = true  
 \_isLoading.*value* = false  
}

Elke view bevat wel een API-call, in deze code wordt gebruik gemaakt van de “isLoading” en “isError”, als er een error is wordt de functie “onError” aangeroepen en deze voegt dan de errormelding toe aan de “errorMessage”. Deze wordt daarna in de observer getoond als toats message.

Voor elke API-call heb je een client nodig. Als de call gedaan is wordt de functie “onResponse” uitgevoerd. Hierin gaan we eerst kijken of de “responseBody” iets van data bevat. Als dit niet zo is wordt de error getoond. Moest de call in zijn geheel falen wordt er ook een error getoond.

Om de databinding mogelijk te maken moet er in de xml nog iets toegevoegd worden namelijk:

<data>  
 <variable  
 name="viewModel"  
 type="com.example.weatherapp.home.viewmodel.HomeViewModel" />  
</data>

Dit zorgt ervoor dat de juiste data gebruikt wordt.

Verder bevat de homepage nog de volgende functie:

fun checkWeatherForCities(firstHour: Int) {  
 var goingToRainTemp = false  
 for (city in cityList) {  
 getWeatherForCity(city, firstHour) **{** willItRain **->** if (willItRain == 1) {  
 goingToRainTemp = true  
 }  
 **}** }  
 goingToRain.postValue(goingToRainTemp)  
  
 \_isLoading.*value* = false  
}

Deze functie loopt over alle steden die op mijn route liggen, de steden zitten nu gewoon in een hard coded list omdat ik de routeberekening niet werkende kreeg. Deze functie gaat kijken of het in een van de gemeentes gaat regenen. De waarde wordt doorgestuurd naar volgende functie:

homeViewModel.goingToRain.observe(*viewLifecycleOwner*) **{** goingToRain **->** if (goingToRain) {  
 imgCondition.setImageResource(R.drawable.*baseline\_directions\_car\_filled\_24*)  
 homeViewModel.rainingText.postValue(*context*?.getString(R.string.*going\_to\_rain*))  
 } else {  
 imgCondition.setImageResource(R.drawable.*baseline\_pedal\_bike\_24*)  
 homeViewModel.rainingText.postValue(*context*?.getString(R.string.*not\_going\_to\_rain*))  
 }  
**}**

Hier wordt dan de juiste tekst en icoon geladen in de view

<TextView  
 android:id="@+id/tv\_going\_to\_rain"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:textSize="24sp"  
 android:textStyle="bold"  
 android:textColor="@android:color/white"  
 android:text="@{viewModel.rainingText}" />  
  
<TextView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:textSize="18sp"  
 android:textStyle="italic"  
 android:textColor="@android:color/white"  
 android:text="@string/consider\_transportation\_options\_text"  
 android:layout\_marginTop="8dp" />  
  
<ImageView  
 android:id="@+id/img\_vehicle"  
 android:layout\_width="120dp"  
 android:layout\_height="120dp"  
 android:layout\_marginTop="16dp"  
 android:layout\_marginBottom="16dp"  
 android:contentDescription="@string/img\_vehicle\_description" />  
  
<TextView  
 android:id="@+id/tv\_time"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:textSize="20sp"  
 android:textColor="@android:color/white"  
 android:text="@{`Your lessons go from ` + viewModel.timeBegin + ` to ` + viewModel.timeEnd}"  
 android:layout\_marginTop="16dp" />

### Weather

Hier wordt dus heel veel hetzelfde gedaan als in de homepage. Hier wordt wel gebruik gemaakt van een recyclerView, dit wordt gebruik om lijsten te maken, ik het het hier gebruikt voor de daily en hourly items weer te geven.  
  
dit is de xml:  
<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android"  
 xmlns:tools="http://schemas.android.com/tools"  
 android:background="@drawable/background\_gradient">  
  
 <data>  
 <variable  
 name="viewModel"  
 type="com.example.weatherapp.weather.viewmodel.WeatherFullViewModel"/>  
 </data>  
  
 <LinearLayout  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent"  
 android:orientation="vertical"  
 android:gravity="center\_horizontal"  
 tools:context=".view.weather.WeatherFullFragment">  
  
 <ImageView  
 android:id="@+id/img\_condition"  
 android:layout\_width="80dp"  
 android:layout\_height="80dp"  
 android:contentDescription="@string/weather\_icon\_description" />  
  
 <TextView  
 android:id="@+id/tv\_temperature"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:text="@{viewModel.temperature}"  
 android:textSize="60sp" />  
  
 <TextView  
 android:id="@+id/tv\_country"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:textSize="20sp"  
 android:text="@{viewModel.region}"/>  
  
 <TextView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginTop="20dp"  
 android:textSize="20sp"  
 android:text="@string/hourly\_forecast"/>  
  
 <androidx.recyclerview.widget.RecyclerView  
 android:id="@+id/hourlyForecastRecyclerView"  
 android:layout\_width="match\_parent"  
 android:layout\_height="wrap\_content" />  
  
 <TextView  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:layout\_marginTop="20dp"  
 android:textSize="20sp"  
 android:text="@string/daily\_forecast"/>  
  
 <androidx.recyclerview.widget.RecyclerView  
 android:id="@+id/dailyForecastRecyclerView"  
 android:layout\_width="match\_parent"  
 android:layout\_height="match\_parent" />  
 </LinearLayout>  
  
</layout>

Op deze pagina komt dus alles van het weer. Zowel het huidige weer als het weer op elk uur van de dag alsook het weer voor de komende dagen.

Om een lijst te gebruiken moet je de volgende dingen doen.

In het Fragment:  
class WeatherFullFragment : Fragment() {  
 private lateinit var weatherFullViewModel: WeatherFullViewModel  
 private lateinit var hourlyForecastRecyclerView: RecyclerView  
  
 override fun onCreateView(  
 inflater: LayoutInflater, container: ViewGroup?,  
 savedInstanceState: Bundle?  
 ): View? {  
 // Inflate the layout for this fragment  
 return inflater.inflate(R.layout.*fragment\_weather\_full*, container, false)  
 }  
  
 override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
 super.onViewCreated(view, savedInstanceState)  
 hourlyForecastRecyclerView = view.findViewById(R.id.*hourlyForecastRecyclerView*)  
 hourlyForecastRecyclerView.*layoutManager* = LinearLayoutManager(*context*, LinearLayoutManager.*HORIZONTAL*, false)  
 hourlyForecastRecyclerView.*adapter* = HourlyForecastAdapter()  
  
 subscribe()  
  
 weatherFullViewModel.getWeatherData("${MainActivity.latitude},${MainActivity.longitude}")  
 }  
  
 private fun subscribe() { weatherFullViewModel.hourlyForecastItemList.observe(*viewLifecycleOwner*) **{** hourItemList **->** hourItemList?.*let* **{** (hourlyForecastRecyclerView.*adapter* as? HourlyForecastAdapter)?.updateData(**it**)  
 **}  
 }** }  
}

Het belangrijkste stuk hieruit is het volgende:

hourlyForecastRecyclerView = view.findViewById(R.id.*hourlyForecastRecyclerView*)  
hourlyForecastRecyclerView.*layoutManager* = LinearLayoutManager(*context*, LinearLayoutManager.*HORIZONTAL*, false)  
hourlyForecastRecyclerView.*adapter* = HourlyForecastAdapter()

Hier wordt de adapter gekoppeld aan de recyclerview

weatherFullViewModel.hourlyForecastItemList.observe(*viewLifecycleOwner*) **{** hourItemList **->** hourItemList?.*let* **{** (hourlyForecastRecyclerView.*adapter* as? HourlyForecastAdapter)?.updateData(**it**)  
 **}  
}**

Hier worden de items geladen in de view dmv de adapter

De adapter werkt eigenlijk als middleware tussen het viewmodel en de view.

Het hourly item in de view bavat volgende velden:

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:orientation="vertical"  
 android:gravity="center"  
 android:padding="16dp">  
  
 <TextView  
 android:id="@+id/tvHour"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:textSize="14sp"  
 android:gravity="center" />  
  
 <ImageView  
 android:id="@+id/img\_condition"  
 android:layout\_width="60dp"  
 android:layout\_height="60dp"  
 android:layout\_marginTop="10dp"  
 android:layout\_marginBottom="10dp"  
 android:contentDescription="@string/weather\_icon\_description" />

<TextView  
 android:id="@+id/tvTemperature"  
 android:layout\_width="wrap\_content"  
 android:layout\_height="wrap\_content"  
 android:textSize="14sp"  
 android:gravity="center" />  
</LinearLayout>

In de adapter word dat telkens een nieuwe instantie van dit item gemaakt en toegevoegd aan de lijst.

package com.example.weatherapp.weather.view  
  
import android.annotation.SuppressLint  
import android.util.Log  
import android.view.LayoutInflater  
import android.view.ViewGroup  
import android.widget.TextView  
import androidx.recyclerview.widget.RecyclerView  
import com.example.weatherapp.R  
import com.example.weatherapp.databinding.DailyWeatherForecastItemBinding  
import com.example.weatherapp.utils.Utils  
import com.example.weatherapp.weather.model.ForecastdayItem  
  
class DailyForecastAdapter : RecyclerView.Adapter<DailyForecastAdapter.ViewHolder>() {  
 private var dayItems: List<ForecastdayItem> = *listOf*()  
 inner class ViewHolder(private val binding: DailyWeatherForecastItemBinding) :  
 RecyclerView.ViewHolder(binding.*root*) {  
 @SuppressLint("SetTextI18n")  
 fun bind(dayItem: ForecastdayItem) {  
 binding.*apply* **{** tvDay.*text* = Utils.getShortDayFromDayItem(dayItem)  
 tvChanceOfRain.*text* = "${dayItem.day?.dailyChanceOfRain}%"  
 tvMinMax.*text* = "${dayItem.day?.maxtempC}°/ ${dayItem.day?.mintempC}°"  
 **}** }  
 }  
  
 override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {  
 val binding = DailyWeatherForecastItemBinding.inflate(  
 LayoutInflater.from(parent.*context*),  
 parent,  
 false  
 )  
 return ViewHolder(binding)  
 }  
  
 @SuppressLint("NotifyDataSetChanged")  
 fun updateData(newDayItems: List<ForecastdayItem>) {  
 dayItems = newDayItems.*toMutableList*()  
 notifyDataSetChanged()  
 }  
  
 override fun onBindViewHolder(holder: ViewHolder, position: Int) {  
 val dayItem = dayItems[position]  
 holder.bind(dayItem)  
 }  
  
 override fun getItemCount(): Int {  
 return dayItems.size  
 }  
}