Wprowadzenie do technologii mobilnych

Vehicle Fuelling Manager App

Wykonali: Kamil Skomro, Michał Warzecha, Jakub Serweta
Github: https://github.com/jserweta/VehicleManager

Tutorial

Opis wymagań funkcjonalnych

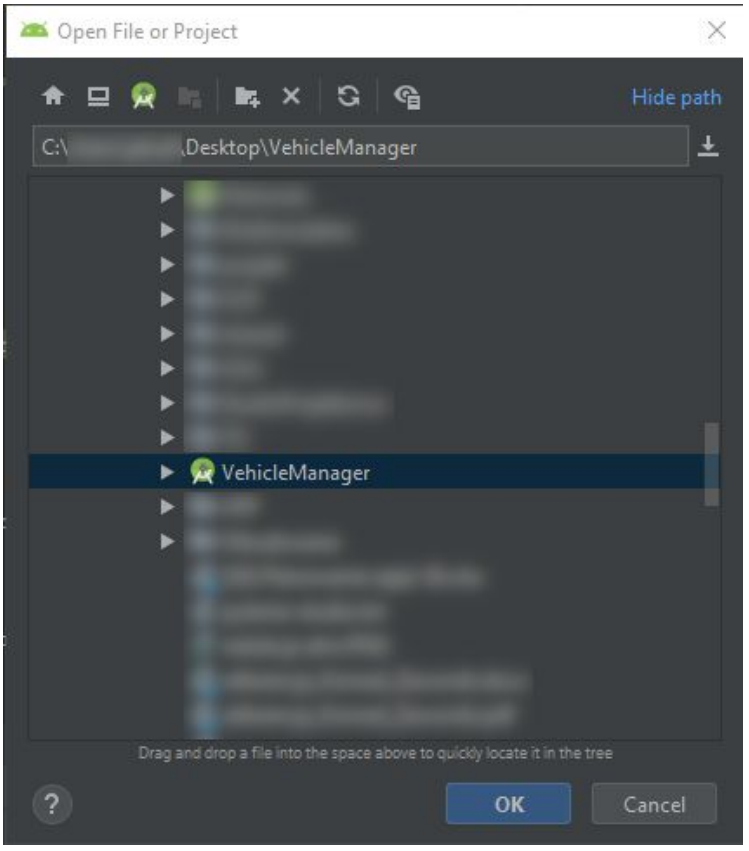
W naszej aplikacji użytkownik ma możliwość stworzenia pojazdu, dla którego będzie mógł notować sobie dane przy kolejnych tankowaniach (przebieg, cenę paliwa za litr, ilość zatankowanego paliwa). Aplikacja dla każdego pojazdu automatycznie wyliczy różnego rodzaju statystyki m.in.:

- spalanie pomiędzy tankowaniami
- statystyki dla bieżącego miesiąca i całej dostępnej historii:
  - całkowity przejechany dystans
  - zużyta ilość paliwa
  - średnie spalanie
  - całkowity poniesiony koszt paliwa
  - średni koszt przejechania jednego kilometra
  - najgorsze spalanie na 100km
  - najlepsze spalanie na 100km

Vehicle Fuelling Manager ma za zadanie ułatwić użytkownikowi obliczanie zużycia oraz prowadzenie historii poniesionych kosztów paliwa.

Inicjalizacja projektu

Po rozpakowaniu archiwum z projektem należy uruchomić Android Studio wybrać ***File -> Open,*** a następnie wybrać odpowiedni folder z listy (zostanie on odpowiednio oznaczony przez Android Studio)



Po wczytaniu projektu, należy wybrać emulator, który mamy zainstalowany i kliknąć przycisk **“Play”**

Realizacja GUI

Główne części GUI to:

- górny pasek z rozwijaną listą typu **Spinner,** która pozwala na wybór pojazdu którego dane chcemy oglądać
- środkowa część to “pojemnik” na fragmenty
- dolny pasek nawigacyjny zawierający trzy kategorie powoduje on podmienianie widoków w środkowej części
- oraz dwa widoki jeden umożliwiający dodawanie pojazdów, a drugi nowych tankowań

Opis tworzenia layout’ów

Najpierw przygotowaliśmy sobie potrzebne w aplikacji ikony, część z nich była dostępna bezpośrednio w bibliotekach dołączonych, natomiast druga część została wyeksportowana z darmowych grafik wektorowych dla aplikacji, za pomocą programu AdobeXD.

Dolny pasek nawigacyjny

Zaczynając od dolnego paska nawigacyjnego. W folderze *color* stworzyliśmy plik *bnv_tab_item_foreground.xml*, który definiuje kolor ikon wyświetlanych przez *BottomNavigationView* w momencie gdy są aktywne lub nie.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_selected="true" android:color="@color/colorAccent" />
    <item android:color="@color/white"  />
</selector>
```

BottomNavigationView posiada interfejs `OnNavigationItemSelectedListener` która pozwala zmieniać kolor ikon w zależności od *state_selected.*

Wyświetlany tekst oraz przypisanie ikon do paska nawigacji zostało zdefiniowane w folderze *menu/bottom_nav.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/main_page"
        android:title="Strona główna"
        android:icon="@drawable/ic_home_24px"/>

    <item...>

    <item...>

</menu>
```

Powyższe pliki edytujące wygląd naszego paska nawigacyjnego zostały do niego dodane za pomocą odpowiednich atrybutów:

```
<!-- Bottom menu -->
<com.google.android.material.bottomnavigation.BottomNavigationView
    app:itemIconTint="@color/bnv_tab_item_foreground"
    app:itemTextColor="@color/bnv_tab_item_foreground"
    app:menu="@menu/bottom_nav"
```

Podmiana zawartości środkowej części ekranu jest możliwa dzięki zastosowaniu `BottomNavigationView` który posiada metodę `OnNavigationItemSelectedListener` pozwalającą przechwycić, który przycisk został kliknięty, wtedy wywołuje stworzoną funkcję ***replaceFragment().***

```
private fun replaceFragment(fragment: Fragment){
    val fragmentTransaction : FragmentTransaction = supportFragmentManager.beginTransaction()
    fragmentTransaction.replace(R.id.fragmentContainer, fragment)
    fragmentTransaction.commit()
}
```

Funkcja ta za pomocą metody *replace* podmienia zawartość elementu o podanym *id* w pliku XML (*activity_main.xml*), na odpowiedni plik (*MainPageFragment*, *RefuellinListFragment*, *VehicleManagerFragment*). Następnie w poszczególnych fragmentach ustawiamy odpowiednie layouty w funkcji *onCreateView* np.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    val view: View = inflater.inflate(R.layout.fragment_main_page, container, attachToRoot: false)
```

Spinner - lista rozwijalna

Aby dodać listę rozwijalną należy w layout-cie umieścić w odpowiednim miejscu element o nazwie *Spinner* i nadać mu id.

```
<Spinner
    android:id="@+id/vehicle_spinner_list"
```

Następnie przypisujemy go do zmiennej w naszym *Activity*.

```
selectVehicleSpinner = findViewById(R.id.vehicle_spinner_list)
```

Aby móc w wierszach naszej listy wyświetlać inne elementy niż zwykłe napisy w niestandardowy sposób należy zdefiniować własny *Adapter* (w naszym przypadku przedefiniowaliśmy ArrayAdapter, gdyż chcemy wyświetlać listę samochodów). Nasz adapter będzie wyświetlał encje typu vehicle więc w konstruktorze musimy podać mu listę z samochodami.

```
class VehicleAdapter(context: Context, @LayoutRes private val layoutResource: Int, vehicleList: List<Vehicle>):
    ArrayAdapter<Vehicle>(context, layoutResource, vehicleList) {
```

Wygląd wierszy został zdefiniowany przy pomocy dwóch elementów:

- dla tła - *drawer/spinner_bg.xml* (który został użyty w kontenerze w którym znajduje się spinner)

- dla wyglądu wierszy - *vehicle_spinner_row.xml* (użyty w metodzie *initView* odpowiedzialnej za inicjalizację *View* dla każdego z wierszy)

W elemencie *vehicle_spinner_row.xml* mamy odpowiednie pola typu *TextView*, dzięki czemu możemy wyświetlić w nim więcej informacji w bardziej estetyczny sposób. Jest on przypisywany do Spinnera w klasie *VehicleAdapter*, podobnie jak dane, które mają zostać w nim wyświetlone.

```
private fun initView(position: Int, convertView: View?, parent: ViewGroup?): View {
    val cView: View = convertView ?: LayoutInflater.from(context).
        inflate(R.layout.vehicle_spinner_row, parent,  attachToRoot: false)

    val initialLetter: TextView = cView.spinner_letter
    val nameLabel: TextView = cView.spinner_vehicle_name
    val mileageLabel: TextView = cView.spinner_vehicle_mileage

    val currentVehicle: Vehicle? = getItem(position)

    if (currentVehicle != null) {
        initialLetter.text = currentVehicle.name.first().toString()
        nameLabel.text = currentVehicle.name
        mileageLabel.text = String.format("%d km", currentVehicle.mileage)
    }

    return cView
}
```

Dzięki metodzie `onItemSelectedListener` dostępnej dla elementu **Spinner** zapisujemy pozycję na jakiej jest on ustawiony oraz odświeżamy zawartość fragmentu, co pozwala na wyświetlenie danych dla odpowiedniego pojazdu.

W momencie dodawania nowego pojazdu konieczne jest odświeżanie zawartości Spinnera, co zrealizowaliśmy za pomocą broadcastu (opis niżej).

Pozostałe elementy

Pozostałe elementy GUI (layouty oraz fragmenty) zostały stworzone na bardzo podobnej zasadzie, dlatego postanowiliśmy opisać sposób tworzenia tylko jednego z nich.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/drawer_general_layout"
    tools:context=".MainActivity">

    <!--Top bar layout-->
    <androidx.constraintlayout.widget.ConstraintLayout
        android:id="@+id/constraint_layout"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <!-- Spinner -->
        <androidx.constraintlayout.widget.ConstraintLayout...>

        <!--Fragment container-->
        <androidx.constraintlayout.widget.ConstraintLayout
            android:id="@+id/fragmentContainer"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            app:layout_constraintBottom_toTopOf="@+id/bottom_nav"
            app:layout_constraintTop_toBottomOf="@id/top_car_select_wrapper">
        </androidx.constraintlayout.widget.ConstraintLayout>

        <!-- Bottom menu -->
        <com.google.android.material.bottomnavigation.BottomNavigationView
            android:id="@+id/bottom_nav"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:background="@color/colorPrimary"
            app:itemIconTint="@color/bnv_tab_item_foreground"
            app:itemTextColor="@color/bnv_tab_item_foreground"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:menu="@menu/bottom_nav" />
    </androidx.constraintlayout.widget.ConstraintLayout>

</androidx.drawerlayout.widget.DrawerLayout>
```

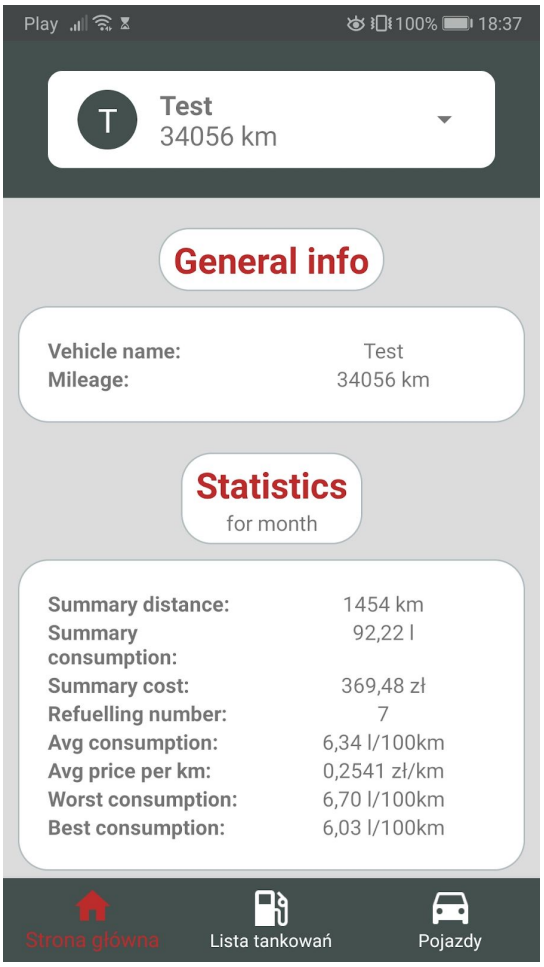
Powyższe zdjęcie przedstawia plik XML *activity_main.xml* jednak pozostałe stworzyliśmy na podobnej zasadzie, a mianowicie głównie używaliśmy *ConstraintLayout,* który pozwala na łatwe dopasowywanie elementów względem siebie łącząc je połączeniami “constraint”.

Każdy element ma swoje *id* dzięki któremu możemy stosować połączenia. Dzięki *id* elementy rozróżniane są w klasach i możemy uzyskać dostęp poprzez ***findViewById().***

Adaptowanie danych do wyświetlenia w GUI

Dla poszczególnych fragmentów zostały stworzone klasy - *adapter*. Mają one za zadanie wpisywać dane pobrane z warstwy danych w odpowiednie pola.

- Spinner - *VehicleAdapter*
- Strona główna - tutaj dane przypisywane są w *MainPageFragment*
- Lista tankowań - *RefuellingAdapter*
- Pojazdy - *VehicleListAdapter*



Strona główna to miejsce, w którym wyświetlane są ogólne informacje oraz średnie statystyki (z ostatniego miesiąca oraz całej dostępnej historii) odnośnie pojazdu wybranego w górnej liście.

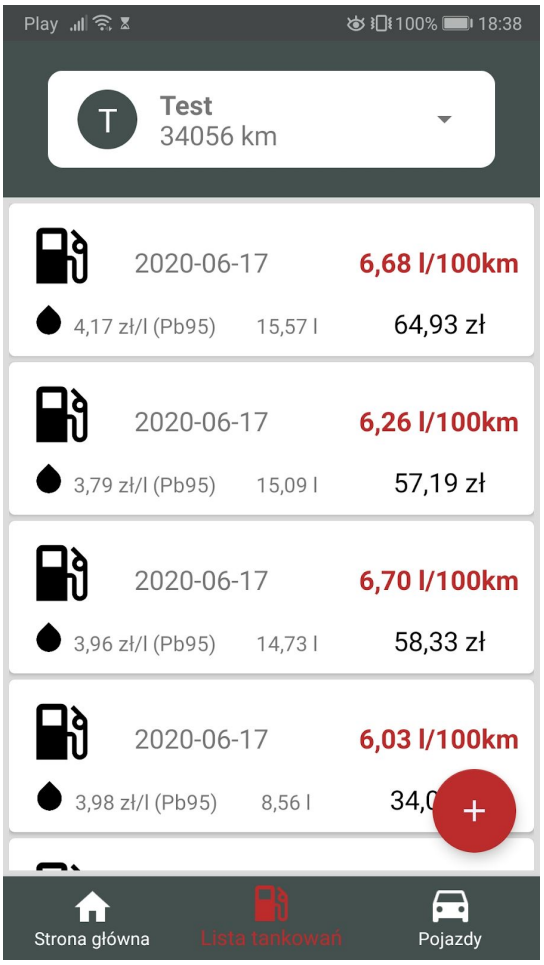
Za wygląd tła w poszczególnych elementach odpowiada zdefiniowany element *drawable/header_layout_bg.xml,* nadawany za pomocą atrybutu `android:background="@drawable/header_layout_bg"`

Dane wyświetlane w tym widoku obliczane są w klasie *RefuellingStatistics* i pobieramy je za pomocą metod:

```
val wholePeriodStat : Statistics = refuellingStatistics!!.getStatisticsForWholePeriod(currentVehicle!!.getId())
val currentMonthStat : Statistics = refuellingStatistics!!.getStatisticsForCurrentMonth(currentVehicle!!.getId())
```

Aby dodać poszczególne wartości do odpowiednich elementów w klasie *MainPageFragment* dla każdego elementu tworzymy zmienną przypisujemy jej element o odpowiednim *id* z pliku XML, a następnie przypisujemy wartość.

```
private var vehicleNameText: TextView? = null
vehicleNameText = view.findViewById(R.id.vehicle_name)
vehicleNameText!!.text = currentVehicle!!.name
```

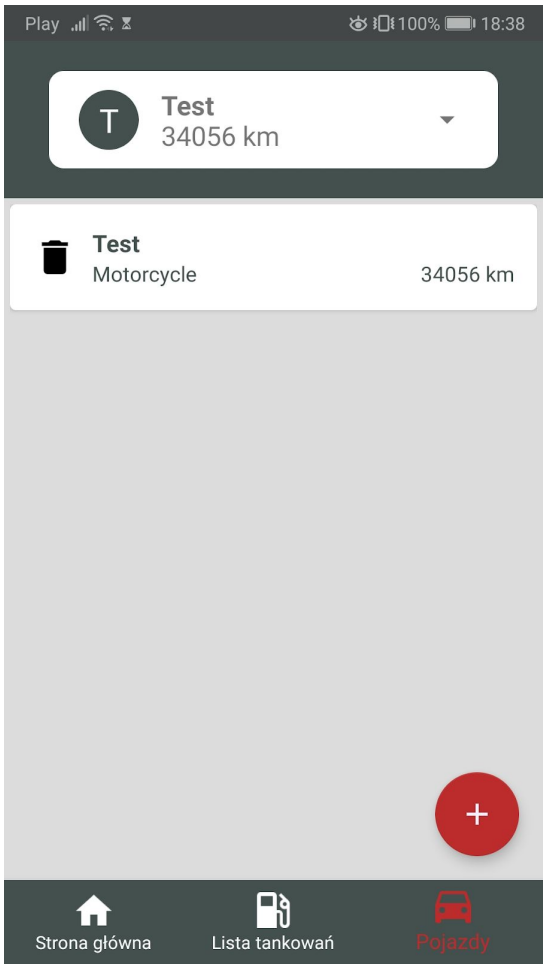


Lista tankowań - wyświetlane są tutaj wszystkie tankowania przypisane do, wybranego w elemencie Spinner, pojazdu, jakie dodał użytkownik.

W prawym dolnym rogu pojawia się przycisk, który przenosi do aktywności dodawania nowego tankowania. Jest to `com.google.android.material.floatingactionbutton.FloatingActionButton`, dlatego może pojawiać się on ponad innymi elementami GUI.

Lista tankowań została stworzona za pomocą **RecyclerView** (`com.agh.wtm.vehiclemanager.util.EmptyRecyclerView`), który zdefiniowaliśmy aby móc dodać metody sprawdzające czy lista jest pusta, a następnie móc podmienić ją na napis “There are no refuellings yet”.

Wygląd wierszy RecyclerView również został zdefiniowany (plik *fuelling_item.xml)*. Jest on dodawany za pomocą atrybutu `tools:listitem="@layout/fuelling_item"`.



Lista pojazdów - wyświetla pojazdy, które zostały dodane przez użytkownika.

Podobnie jak lista tankowań została stworzona za pomocą **RecyclerView** z zdefiniowanym wyglądem wierszy (plik *vehicle_list_row.xml)* i aktywowany za pomocą `tools:listitem="@layout/vehicle list row"`.
Gdy w bazie nie ma żadnych pojazdów, automatycznie uruchamiany jest ekran dodawania nowego pojazdu, który można również uruchomić za pomocą przycisku *FloatinActionButton* prawym dolnym rogu.

Na każdym elemencie z *RecyclerView* pojawia się symbol kosza, który umożliwia usunięcie pojazdu z bazy, wraz z którym usuwane są odpowiadające mu tankowania. Realizują to poniższe metody:

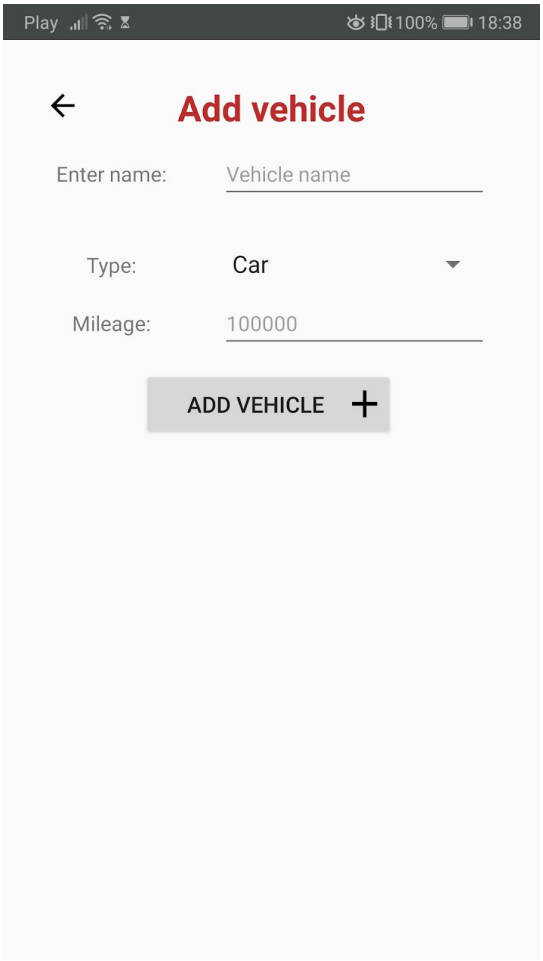
```
dbHelper!!.deleteById(Vehicles, vehicleToRemove.getId())
dbHelper!!.deleteFuellingsForVehicle(vehicleToRemove.getId())
```

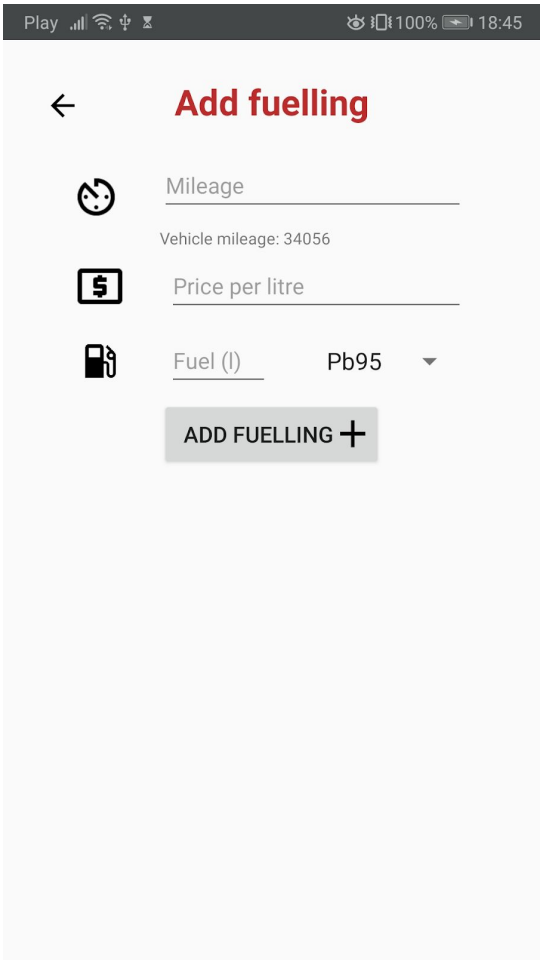
Do wypełnienia danych w dwóch powyższych fragmentach odpowiednio służy klasa *RefuellingAdapter* oraz *VehicleListAdapter,* a dokładnie metoda `override fun onBindViewHolder` posiadająca dostęp do poszczególnych pól za pomocą atrybutu *holder.* W klasie *RefuellingAdapter* obliczany jest również koszt tankowania oraz spalanie. Pozostałe dane pobierane są bezpośrednio z bazy.

```
override fun onBindViewHolder(holder: VehicleViewHolder, position: Int) {
    val currentItem : Vehicle  = vehicleList[position]

    holder.vehicleNameField.text = currentItem.name
    holder.vehicleMileage.text = String.format("%d km", currentItem.mileage)
    holder.vehicleTypeField.text = currentItem.type.toString()

}
```





Dodawanie pojazdu oraz **tankowania** - aktywności pozwala dodać nowy pojazd/tankowanie. Dostępne są trzy typy pojazdów (Car, Motorcycle, Lorry) oraz trzy typy paliwa (Pb95, Pb98, ON)

W momencie, gdy nie posiadamy w bazie żadnego pojazdu automatycznie zostaniemy przekierowani do widoku dodaj pojazd, a także ukryta zostaje strzałka oraz wyłączony systemowy przycisk powrotu.

Po kliknięciu przycisku add oprócz dodania wartości do bazy konieczne jest odświeżenie zawartości Spinnera oraz w obydwóch przypadkach zawartości listy RecyclerView, aby umożliwić natychmiastowe odświeżanie korzystamy z **Broadcastu** jest on wysyłany z *AddFuellingActivity* oraz *AddVehicleActivity* do *MainActivity* i zamykana jest aktywność

```
val intent = Intent( action: "com.agh.wtm.vehiclemanager.UPDATE_SPINNER")
sendBroadcast(intent)
finish()
```

Wartość przekazywana w *Intent* to filtr, dzięki któremu *MainActivity* wie, że są to dane przeznaczone dla niej.

Broadcast receiver:

```
val updateSpinnerFilter = IntentFilter( action: "com.agh.wtm.vehiclemanager.UPDATE_SPINNER")
broadcastUpdateSpinner = object : BroadcastReceiver(){
    override fun onReceive(context: Context, intent: Intent) {
        updateSpinner()
        var fragment: Fragment? = supportFragmentManager.fragments.last()
        if(fragment != null && fragment is VehicleManagerFragment) {
            fragment.updateVehicleList()
        }
    }
}
registerReceiver(broadcastUpdateSpinner, updateSpinnerFilter)
```

Po otrzymaniu komunikatu realizujemy uaktualnienie listy pojazdów, powiadamiając *VehicleAdapterList,* że dane uległy zmianie:

```
fun updateVehicleList() {
    vehicleListAdapter!!.vehicleList = getVehicles()
    vehicleListAdapter!!.notifyDataSetChanged()
}
```

Za odświeżenie fragmentów odpowiada również metoda `refreshFragment()` wywoływania przy każdym przywróceniu MainActivity w nadpisanej metodzie `override fun onResume().`

```
private fun refreshFragment() {
    val currentFragment: Fragment = supportFragmentManager.findFragmentById(R.id.fragmentContainer)!!
    val fragTransaction: FragmentTransaction = supportFragmentManager.beginTransaction()
    fragTransaction.detach(currentFragment)
    fragTransaction.attach(currentFragment)
    fragTransaction.commit()
}
```

Async Task

W naszym projekcie użyliśmy *AsyncTask* do pobrania z internetu obrazu który następnie wyświetlamy przy pomocy *ImageView.* W tym celu w fragmencie w którym znajduje się obrazek stworzyliśmy prywatną klasę implementującą *AsyncTask.* W naszym przypadku typem *Params* jest *String* gdyż chcemy móc przekazać adres do naszego zdjęcia przy pomocy właśnie napisu. Typ *Result* to *Drawable* które to może później wyświetlić w *ImageView* używając metody *setImageDrawable.*

Nasza klasa pobierająca zdjęcie w całości wygląda tak:

```
private class DownloadImage(val imageViewBanner: ImageView): AsyncTask<String, Void, Drawable>() {

    private fun downloadImage(_url: String): Drawable {
        return try {
            val url = URL(_url)
            val input: InputStream = url.openStream()
            val buf = BufferedInputStream(input)
            val bMap: Bitmap = BitmapFactory.decodeStream(buf)

            input.close()
            buf.close()
            BitmapDrawable(Resources.getSystem(), bMap)
        } catch (e: Exception) {
            ColorDrawable( color: 0)
        }
    }

    override fun doInBackground(vararg params: String?): Drawable {
        return downloadImage(params[0]!!)
    }

    override fun onPostExecute(result: Drawable?) {
        if(result != null) {
            imageViewBanner.setImageDrawable(result)
        }
    }
}
```

W razie niepowodzenia stworzymy Drawable które w całości jest wypełnione kolorem.

Przykładowe użycie:

```
DownloadImage(imageViewBanner)
    .execute( ...params: "https://images/test.png")
```

Jak widać, aby uruchomić *AsyncTask* należy wywołać metodę execute podając argument (lub argumenty) które potem będą dostępne w metodzie *doInBackground* naszego zadania asynchronicznego. Jak widzimy obiekt zwracany z tej metody musi mieć typ podany przez nas jako *Result (czyli ostatni type parametr dla AsyncTask),* a *params* są typu podanego jako pierwszy dla *AsyncTask.* Drugi parametr to *Progress* i może być używany jeśli chcielibyśmy wyświetlić progres naszego zadania (my tego nigdzie nie używamy więc ustawiliśmy typ Void).

W metodzie *onPostExecute* możemy użyć rezultat naszego zadania zaraz po tym jak zostanie ono ukończone.

Omówienie realizacji logiki biznesowej

Jedyna logika biznesowa w naszej aplikacji to obliczanie statystyk dla tankowań. W tym celu stworzyliśmy klasę *RefuellingStatistics* która przyjmuje w konstruktorze instancje *VehicleDBHelper*-a. Będzie nam ona potrzebna do wyciągania informacji o tankowaniach z bazy danych dla danego pojazdu.

Metodami które udostępnia nam ta klasa są *getStatisticsForCurrentMonth* i *getStatisticsForWholePeriod,* które przyjmują jako parametr id pojazdu dla którego chcemy otrzymać statystyki. Wszystkie obliczane przez nas statystyki są zwracana w *data class* o nazwie *Statistics* o następującej strukturze:

```
data class Statistics(
    val summaryDistance: Double,
    val summaryConsumption: Double,
    val avgConsumption: Double,
    val avgPricePerKilometer: Double,
    val summaryCost: Double,
    val minConsumption: Double,
    val maxConsumption: Double,
    val refuellingsNumber: Int
)
```

Szczegółów związanych z ich obliczaniem nie będziemy tutaj objaśniać gdyż polegają one na prostych operacjach, typu obliczanie średniej itp.

Omówienie realizacji warstwy danych

Tabele

Tabela pojazdy

- id pojazdu
- typ pojazdu (enum)
- nazwa pojazdu
- przebieg
- przebieg przy ostatnim tankowaniu

Tabela tankowania

- id tankowania
- id pojazdu
- data tankowania
- ilość zatankowanego paliwa
- cena paliwa
- typ paliwa (enum)
- przebieg przy ostatnim tankowaniu
- przebieg (w momencie tankowania)

W naszej aplikacji korzystaliśmy z SQLite i zdecydowaliśmy się na implementację opartą na definiowanie tabel dla zasobów przy pomocy poniższego interfejsu:

```
interface Table<T> : BaseColumns {
    val tableName: String
    val sqlCreateEntries: String
    val sqlDeleteEntries: String
    fun values(entity: T): ContentValues
    fun fromCursor(cursor: Cursor): T
}
```

Powyższy interfejs jak i jego implementacje przechowujemy w obiekcie *VehicleContract*. Kontrakt ten zawiera wszystkie informacje o strukturze naszej bazy danych. Dodatkowo tworzymy listę *tables* przechowującą wszystkie implementacje tego interfejsu, po to aby przy tworzeniu bazy danych można było stworzyć tabele iterując się po tabelach należących do bazy.

Do komunikacji z bazą danych używana jest klasa *VehicleDBHelper* rozszerzająca *SQLiteOpenHelper*. Jeśli baza danych o nazwie podanej w konstruktorze *SQLiteOpenHelper* nie istnieje to klasa ta będzie chciała ją utworzyć, zatem potrzebujemy zaimplementować metodę onCreate:

```
override fun onCreate(db: SQLiteDatabase) {
    tables.forEach { it: VehicleContract.Table<*>
        db.execSQL(it.sqlCreateEntries)
    }
}
```

Dzięki temu, że każda z tabel posiada zdefiniowany “przepis” na utworzenie się możemy to zrobić przy pomocy pętli forEach. Podobnie postępujemy w przypadku metody onUpgrade, najpierw usuwamy każdą tabele (w definicji każdej tabeli mamy również odpowiednie zapytanie SQL to jej usunięcia), a następnie wywołujemy metodę onCreate.

Dodatkowo w klasie *VehicleDBHelper* dodajemy metody pozwalające na dodawanie, modyfikowanie, usuwanie i pobieranie naszych encji. Robimy to przy pomocy dostępnej wewnątrz klasy instancji *SQLiteDatabase* o nazwie *writableDatabase*.
Udostępnia ona m.in. metody *update, insert, query* i *delete.* Do użycia metod *insert* i *update* potrzebujemy znać nazwę tabeli oraz naszą encje zapisaną przy pomocy *ContentValues.*
W przypadku metody update nakładamy ograniczenie na przyjmowanie encje. Muszą one implementować interfejs Entity, czyli defacto mieć zdefiniowaną metodą getId(). Potrzebne jest to po to, aby zmodyfikować tylko encje o danym id. Gdybyśmy nie podali tego ograniczenia modyfikowali byśmy wszystkie encje w tabeli, a tego nie chcemy. Informacje o tym jak konwertować encje do *ContentValues* również umieściliśmy wcześniej w definicji tabeli.

```
fun <T> insert(contract: VehicleContract.Table<T>, entity: T): Long {
    val entityValue : ContentValues = contract.values(entity)
    return writableDatabase.insert(contract.tableName, nullColumnHack: null, entityValue)
}

fun <T: Entity> update(contract: VehicleContract.Table<T>, entity: T): Int {
    val entityValue : ContentValues = contract.values(entity)
    return writableDatabase.update(contract.tableName, entityValue, whereClause: "_id=?", arrayOf(entity.getId().toString()))
}

fun <T> getAll(contract: VehicleContract.Table<T>): List<T> {
    val c : Cursor! = writableDatabase.query(contract.tableName, columns: null, selection: null, selectionArgs: null, groupBy: null, having: null, orderBy: null)
    return generateSequence { if (c.moveToNext()) c else null }
        .map { contract.fromCursor(it) }
        .toList()
}

fun <T> getById(contract: VehicleContract.Table<T>, id: Int): T? {
    val c : Cursor! = writableDatabase.query(contract.tableName, columns: null, selection: "_id=?", arrayOf(id.toString()), groupBy: null, having: null, orderBy: null)
    return generateSequence { if (c.moveToNext()) c else null }
        .map { contract.fromCursor(it) }
        .toList().getOrNull( index: 0)
}
```

W przypadku metod *getAll()* i *getById()* otrzymujemy Cursor, który jest iteratorem. Z każdego elementu iteratora możemy wydobyć nasze encje przy pomocy napisanej wcześniej implementacji metody *fromCursor*.

Przykładowe implementacje metod *fromCursor* i *toContentValue* dla encji Vehicle widać poniżej:

```
override fun toContentValues(vehicle: Vehicle): ContentValues {
    return ContentValues().apply { this: ContentValues
        put(COLUMN_NAME_NAME, vehicle.name)
        put(COLUMN_NAME_TYPE, vehicle.type.toString())
        put(COLUMN_NAME_MILEAGE, vehicle.mileage)
    }
}
override fun fromCursor(cursor: Cursor): Vehicle {
    return Vehicle(
        cursor.getInt( columnIndex: 0),
        cursor.getString( columnIndex: 1),
        Vehicle.VehicleType.valueOf(cursor.getString( columnIndex: 2)),
        cursor.getInt( columnIndex: 3)
    )
}
```

Teraz mamy już cały ORM na potrzeby naszej aplikacji.