Freya-The-Goddess
29/09/2022 - 04/11/2022

Custom App - “Victorian Turntables”

Links:

GitHub Repository: https://github.com/SDMD-2022/custom-app-FreyaCR

Report Video: https://youtu.be/wZ-Eif2Bs_k

Table of Contents:

Custom App Summary:

My custom app idea is to make a CRUD app for storing and editing information about railway turntables in Victoria. I already have an existing dataset in the form of a .csv file with surviving turntables and related metadata like location, status, rail gauge turntable length, etc.

The app will use a RecyclerView list in the main activity which will display the identifying data for each turntable. Clicking on one of the list items will launch a detail activity which will show a Google Map fragment with a location pin and all data about that turntable. Turntable data will be modified in the edit detail activity which will populate text inputs with the existing data to be modified. New turntables will be added from the main activity using the same edit detail activity. Pressing the back button from the edit detail activity will give the option save or discard any changes. UML activity diagrams of the UI flows for the most common user tasks are included in *Appendix 2*.

App Topic Background

Turntables are an important piece of railway infrastructure which allows locomotives to be turned around to face the other direction. A rotatable steel bridge with tracks on top is balanced in the middle by a pivot or bearing with the turntable usually inside a pit so the tracks are level. When the weight of the locomotive is balanced over the pivot the turntable can be rotated by hand with a bar at each end by as little as one person. Some later turntables were electrically powered to cut down on man power, especially in locomotive depots where many locomotives would be turned a day. A handful of niche examples were also powered by alternative methods including petrol, diesel and pneumatic.

The ability to turn locomotives was especially important in the days of steam, when leading funnel-end first provided much better visibility and comfort for the driver and fireman than leading tender first. This also applied to early diesel locomotives, which often had only one driver’s cab and leading short-hull first had better visibility than leading long-hull first. However modern diesel and electric locomotives usually have a driver’s cab at either end, so under normal circumstances don't need to be turned for normal operation. For this reason many turntables fell into disuse and disrepair, with many eventually removed and scrapped.

Some railway turntables in Victoria have survived to this day and a handful are still in use at locomotive depots, freight terminals and maintenance facilities. VicTrack and V/Line also maintain a number of turntables across country Victoria for use by heritage and tourist special services, often run by restored steam locomotives. There are also a number of turntables located on tourist railways for their private operations. Many of the surviving turntables are now heritage listed to be preserved as part of railway history, and I believe recording and documenting these parts of history is important. My custom app will allow me to keep a record of the surviving turntables that can be updated and amended.

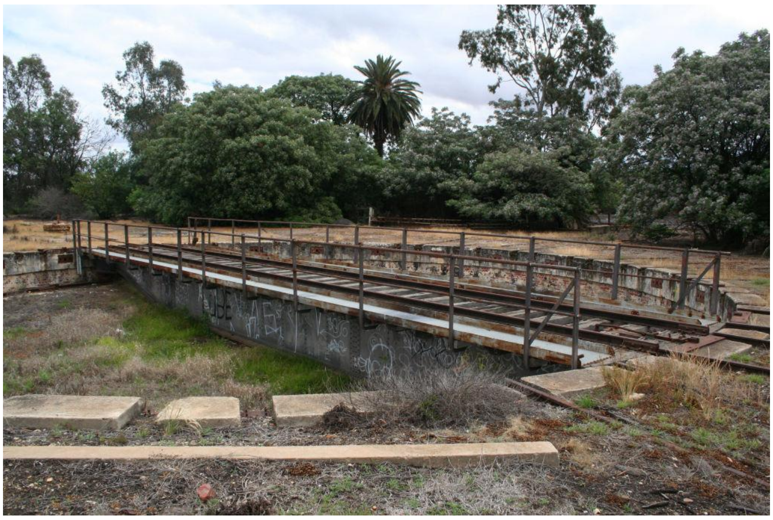


fig.1 - A typical example of a V.R. 70' manual turntable located at Benalla, Marcus Wong 8/11/2008 [https://railgallery.wongm.com/north-east-stations/benalla/D625_2526.jpg.html]

Knowledge Gaps and Solutions

Gap 1 - Select and implement app color palette and set up app themes for day and night mode:

As most of the railway turntables in Victoria were constructed and installed by the state-owned enterprise Victorian Railways (1959-1983) I wanted to make my app reflect the most prominent color scheme of the company with navy blue and golden yellow color scheme. Using the material.io color tool I selected the colors I wanted and checked their accessibility, a screenshot of the palette is in *fig.2*. I then added these colors and their light and dark variants into the `/res/values/colors.xml` file with appropriate names. I also added some colors for day and night text and background colors, which are a slightly blue grey and off white. I then modified the `themes.xml` files for day and night themes with the new color palette.

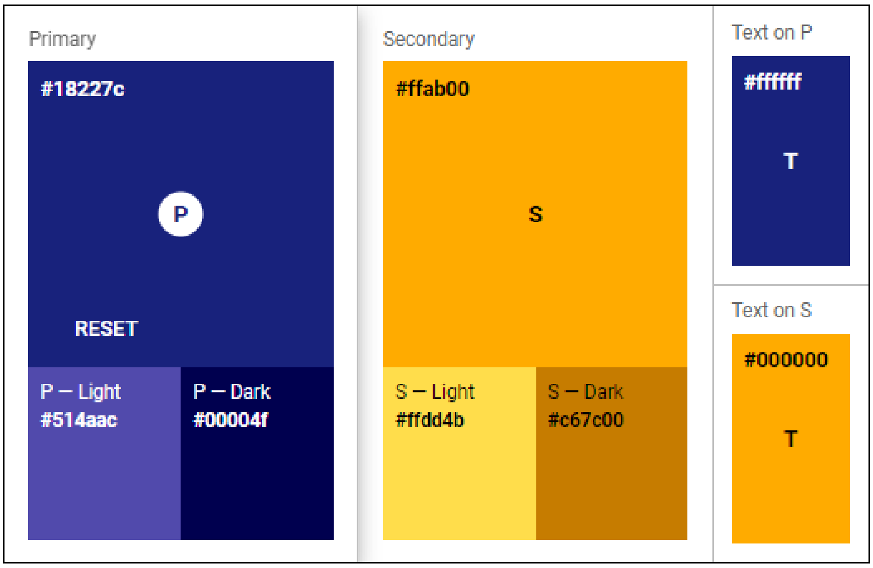


fig.2 - Primary and secondary color palette selected using the material.io color tool

Gap 2 - Sketch app layout design for required activities and fragments

I created some rough layout sketches on paper to visualise the activities and views required to build my custom app. The main activity will have a `RecyclerView` list and I wanted each row to show the identifying information for each turntable. This includes TextViews for the turntable name, location, status strings left justified and the length in feet and power type at the right of the row. The rows will also contain an `ImageView` with a custom icon that will be colored based on status, this icon will be designed in *Gap 3*.

The detail activity will have a Google Maps fragment in the top half of the screen which will have a map pin at the location of the turntable. The bottom half of the screen will contain a scrollable table of the turntable data with each row having a bold label `TextView` and another `TextView` for the data.

The edit detail activity will contain a form for editing the turntable data fields. Most fields are text inputs, with some of them being limited to number data types. Some of the fields will use `Spinner` dropdown menus to let the user select from a list of predefined options. This activity will be used both for editing existing turntables by pre-populating the fields when launched, and for adding new turntables with blank fields.

I scanned my layout sketches and shared them with my partner and a few friends to get some preliminary feedback. Scans of my paper sketches are included in *fig.3* on the next page. These layouts are subject to change as the app is created and the functionality is implemented, as well as from feedback later in the mini conference.

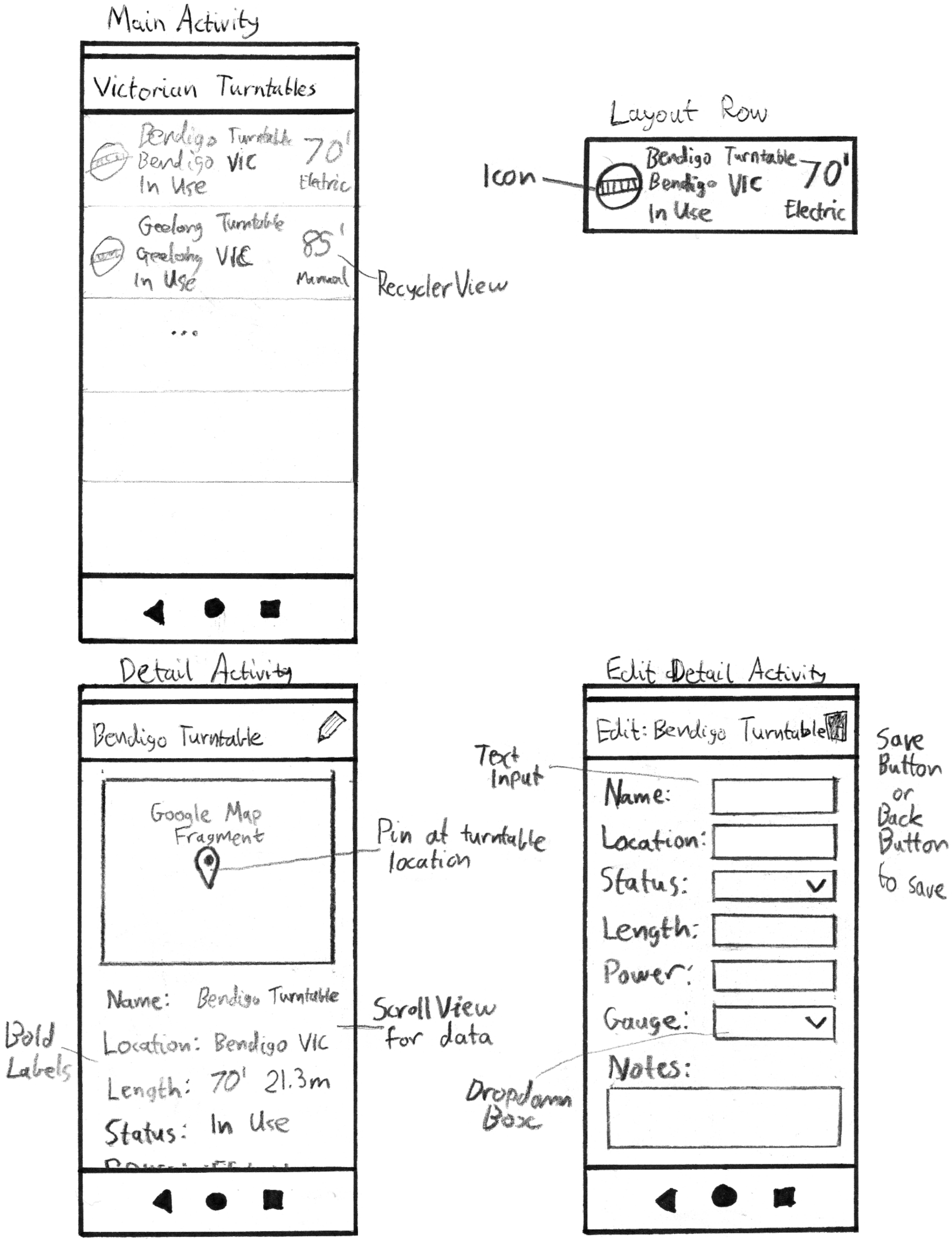


fig.3 - App design sketches for main activity, main row, detail activity and edit detail activity

Gap 3 - Create icon for app and implement as launcher icon and drawable icon

I wanted to design my own vector icon for my custom app which would be used for the app's home screen icon as well as utilised within the app's design. Vector graphics are good for this as unlike raster images they can be easily scaled and colored programmatically. Android also allows vector graphics to be layered together so components can easily be reused for multiple different implementations. To design my vector graphics I used InkScape, a free open-source vector graphics editor. I used primitive shapes (circle, rectangle and rounded rectangle) and boolean operations (union, difference, intersection) to combine and modify the shapes to my design.

I knew my icon would be a simplified drawing of a railway turntable from above. This would include a circle or ring with railway tracks running horizontally through the circle. I came up with four slightly different versions of the design and to share and decide which I would use as shown in *fig.4*. After getting the opinions of my partner and a few friends as well my own opinions I chose the second version as I like the shape and think it will integrate well into the app.

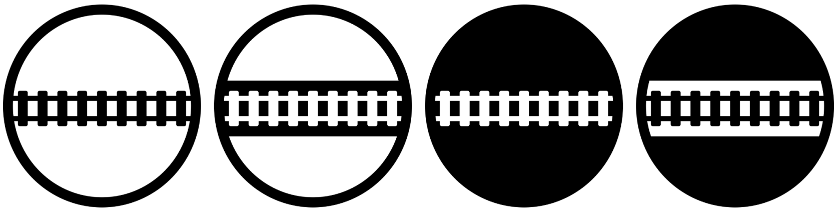


fig.4 - Experimental designs of turntable icon for the app, the second design was chosen for the final icon

I then made sure that the second design was all combined into one path and exported to the correct sizes. For the app icon foreground the vector should be exported at 108dp x 108dp with the icon within the 72dp x 72dp safe zone, and I also exported a background graphic of a square filling the canvas. For use in the app layouts I exported the icon vector as 24dp x 24dp filling the canvas.

With the required SVG files generated I imported them into the project using Android Asset Studio to convert the SVG files to XML vector resources in `/res/drawable`. I then modified the resources' fill colors using color resources from the color palette, including using a radial gradient for the launcher icon background. With the foreground and background XML resources I then used the Asset Studio to generate app icons including legacy icons. This generates in the `/res/mipmap` folder an anydpi XML resource which combines both layers (used for android API 26 and above), and also creates 5 different sized PNG images for legacy devices. The app launcher icons are generated for square and round app icons which will be displayed based on the user's device preferences and OS version. The last thing I did was convert the PNG images to WEBP files following convention as they are a bit smaller. To apply the new launcher icon I modified `AndroidManifest.xml` so the `android:icon` and `android:roundIcon` attributes point to the mipmap resources. The final app launcher icon design is shown in *fig.5*.

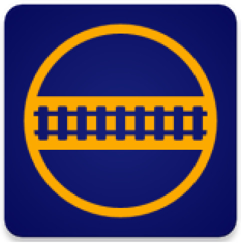


fig.5 - Final app launcher icon, using theme colors and icon vector

Gap 4 - Create layout XML resources for main activity, detail activity and edit detail activity

Based on my layout sketches that I created earlier I built the layout XML files for the required activities and fragments for the app. Whilst building the layouts I followed conventions by externalising colors, strings, dimensions and styles into the appropriate files in the `/res/values` folder. Externalising values from the code to separate files allows greater flexibility as multiple versions of a resource can exist for different device states like theme and orientation. As each of the layouts has been designed with dynamic spacing and externalised values there was no need to implement landscape layout resources for any of the activities in this app.

The main activity was a simple layout with only a `RecyclerView` to be populated by code. Using `RecyclerView` is the preferred method for creating scrolling lists over the old `ListView` method as performance can be greatly increased. The `RecyclerView` list will be populated with rows using the `row_layout_main.xml` layout, which contains the custom turntable icon and the text views for the identifying turntable data. The adapter to handle the RecyclerView and populate rows will be created in *Gap 6*.

The detail activity has a Google Maps fragment which will be handled in code later, and has a `TableLayout` for the detailed information with bold labels and normal text data `TextViews` which will be populated with the turntable data when the detail activity launches.

The edit detail activity has a similar `TableLayout` to the detail activity, with bold label `TextViews` for each field. The text input fields will use a `TextInputLayout` with a `TextInputEditText` inside which can get string input from the user and show validation errors. The dropdown inputs use `Spinner` views, which will be implemented in *Gap 11*.

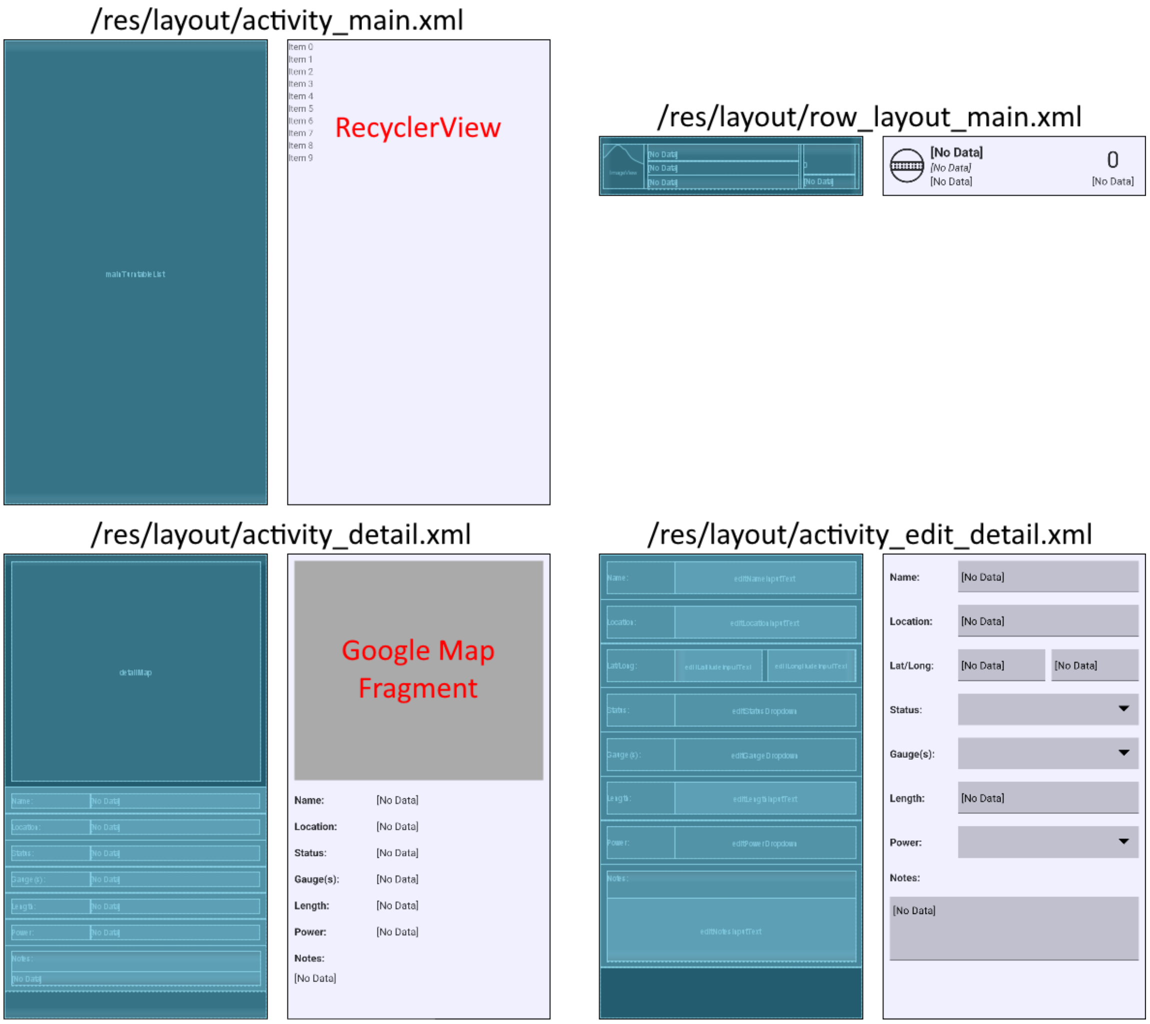


fig.6 - Android Studio layout design tool for each activity and fragment layout

Gap 5 - Create and implement Room database into app

The turntable data will be persistently stored in the app using a Room database. Room is an Android implementation of SQLite database that is more robust and cross-compatible than using SQLite alone. The addition of a number of Kotlin annotations reduces the amount of repetitive boilerplate code needed to set up and access the database. I followed the steps in the Google CodeLab "Room with a View" to implement the Room database into my custom app, adapting the simple database code to meet the more complex requirements of my app and data. A number of new classes and files had to be created to manage and interact with the Room database, following conventions and best practices.

`TurntableEntity` is a data class that is used to store and interact with the turntable data from the database in memory. Similar to the parcelable data classes used in previous tasks this holds the data fields in an object for each turntable. The data class implements the `@Entity` annotation which handles a lot of boilerplate code behind-the-scenes. Each data variable in the class constructor represents a database column and is annotated with `@ColumnInfo(name="")` giving each field a database column reference. The primary key will be automatically generated and shouldn't be specified in the class constructor, so it gets defined in the class body using the `@PrimaryKey(autoGenerate=true)` annotation along with the `@ColumnInfo` annotation.

The `TurntableEntity` class also contains some additional functions for interacting with the turntable data for use in the app. The `lengthMeters()` function converts the `lengthFeet` integer into the length in meters and returns a double of the value. The `lengthString()` function returns a human readable string containing the length in feet and the length in metres in parentheses. The function `gaugeString()` similarly returns a human readable string but for the gauges of the turntable, which are labeled with their classification (NG/SG/BG) and if there are multiple gauges they will be separated by slash delimiters.

The `TurntableEntity` class also contains the functions required for drawing the tinted custom icon for the main activity rows. The `statusColor()` function returns the correct color integer matching the status, which is fetched from the theme attributes so that the correct light or dark variant is used based on the device theme. The `drawIcon()` function creates and returns a custom `Drawable` icon with the correct colors. The foreground and background layers from the `/res/drawable` folder are retrieved, then the foreground is tinted with the text color from the theme attributes and the background is tinted with the color from `statusColor()`. As both of these functions require context to access theme attributes and drawable resources they both take a `Context` object as a parameter. This context will come from the `MainRowAdapter.ViewHolder.bind()` function which will call `drawIcon()` when populating the `RecyclerView` rows, which will be implemented in Gap 6.

As the TurntableEntity objects will need to be passed between activities the objects must be able to be serialized, which in Kotlin is done by making the class extend the Parcelable class. However due to clashes between the `@Entity` and `@Parcelize` annotations the latter cannot be used, therefore the parcelable implementation must be done manually. This required overriding the `writeToParcel()` function which converts the `TurntableEntity` into a `Parcel` object. A custom constructor function is also created which converts the `Parcel` object back into a `TurntableEntity` with all properties populated.

As the `gauges` field is a `MutableList<Int>` it cannot be directly parsed into the SQLite database nor used directly in a `Parcelable` implementation. A TypeConverter is a class which can convert one type to another, in this case converting `MutableList<Int>` into a `String` and vice versa. The `GaugeListTypeConverter` class is created with two functions `gaugeListToString()` and `stringToGaugeList()` which are both annotated with `@TypeConverter`. The `gaugeListToString()` function uses the `.joinToString` Kotlin collection lambda to join the list into a string separated by a slash delimiter. The `stringToGaugeList()` function uses the `.split` lambda to separate the string by the slash delimiter, then uses the `.map` lambda to run `.toInt()` on each value and return a list of strings, and finally `.toMutableList()` is used to make the list mutable. The type converter is implemented into the `TurntableEntity` data class using the `@TypeConverters(GaugeListTypeConverter::class)` annotation.

To define the database itself the abstract class `TurntableDatabase` is created which inherits `RoomDatabase`. An abstract function `turntableDao()` is created which will return the data access object (DAO). The class has a `companion object` which contains the constructor functions that will retrieve the instance of `TurntableDatabase` if it exists, otherwise will create a new instance.

The data access object (DAO) is an interface that maps the SQLite database queries to Kotlin functions for use in the app. The interface `TurntableDao` is created and annotated with `@Dao`. The functions inside are annotated to pass an SQLite query using the annotation `@Query("")`. Room also has special DAO annotations `@Insert`, `@Update`, and `@Delete` for those database queries, which take a `TurntableEntity` object as a parameter and handle the actual SQLite database query behind-the-scenes. These functions are defined with the `suspend` keyword which allows the function to be run asynchronously and can be stopped and resumed as computing resources allow.

The repository class `TurntableRepository` interacts with the DAO and is used to manage multiple data sources where applicable. For this app only one data source exists however it is still good practice to implement the Room database with a repository for access. The repository contains the `suspend` functions `insert()`, `update()`, `delete()` and `deleteAll()` which each call the equivalent DAO function. These functions are annotated with `@WorkerThread` which denotes that the methods should only be called in the worker thread and should not be run on the UI thread.

The database view model is the primary communication method between the user interface and the database via the repository. The `TurntableViewModel` class was created which takes the repository as a constructor parameter. It holds the `LiveData<List>` of the turntable entities in the `allTurntables` variable The view model also contains the functions `insert()`, `update()`, `delete()` and `deleteAll()` which call the equivalent functions from the repository in a coroutine.

The `TurntableViewModel.kt` file also contains the `TurntableViewModelFactory` class which inherits `ViewModelProvider.Factory`. The `create()` function gets the existing view model if an instance already exists, otherwise creates the `TurntableViewModel` object.

To make sure that only one instance of the database exists for the app an application class `TurntableApplication` is implemented. This holds two variables `database` and `repository` for persistently holding both outside of the activity lifecycles. The variables are declared `by lazy` which means they are only created as they are needed. The application class also holds the coroutine scope for the database actions to be performed in the background. To implement the application class into the app `AndroidManifest.xml` is modified, adding the `android:name=".TurntableApplication"` attribute to the `<application>` tag.

To access the Room database from the activities the `TurntableViewModel` is used. The view model is stored in a class level variable `turntableViewModel` which uses the `TurntableViewModelFactory` to retrieve the view model if an instance exists, else it creates a new instance. The view model functions `insert()`, `update()`, `delete()` and `deleteAll()` can then be used in the activities as required to interact with the database. The main activity and detail activity both implement the `TurntableViewModel` as each needs access to the database.

When the app is first launched the database should be populated with the default turntable data from my existing dataset, read from the file `/res/raw/default_turntable_data.csv`. This is done in a new function `readDefaultTurntableData()` in `MainActivity.kt` that reads each line of the file and creates a `TurntableEntity` object from the line which gets inserted into the database with the `insert()` function. To avoid blocking the UI thread when reading the CSV file the `readDefaultTurntableData()` function is launched in a coroutine. This is done by surrounding the function call in `lifecycleScope.launch {}`, which launches the function with a coroutine scope meant for lifecycle aware functions. This is done both in the `onCreate()` method on the first launch as well from the reset data options menu item later in *Gap 7*.

To detect the first launch of the app, shared preferences are used as they can persistently store simple data types, in this case a `Boolean` value. The activity shared preferences are retrieved using `getPreferences(MODE_PRIVATE)`, then the `"firstLaunch"` preference is attempted to be fetched using `getBoolean()`, which will either return the existing value from the shared preferences or if the preference cant be found will return the default value of `true`. An `if` statement checks if the returned value is `true` which means it is the first launch, therefore `readDefaultTurntableData()` is called in a coroutine to populate the database. Then the `"firstLaunch"` shared preference is set to `false` so the next launch won't trigger the database to be re-populated.

Gap 6 - Create RecyclerView list adapter and implement with Room database data

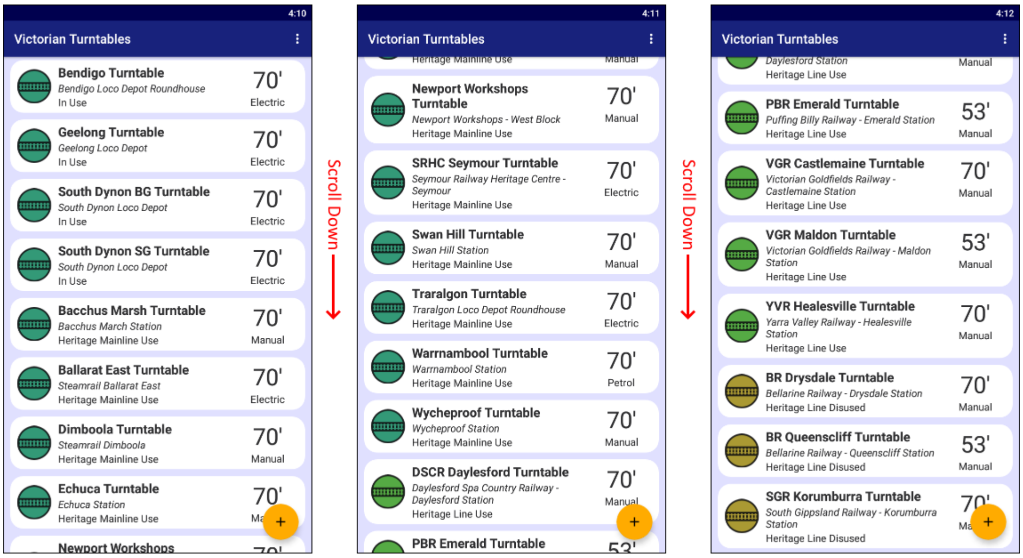
Also following the CodeLab I created the list adapter class and view holder class for the main activity `RecyclerView`. The class `MainRowAdapter` was created and inherits `ListAdapter` which is similar to the `RecyclerView.Adapter` class used in previous tasks. The `onCreateViewHolder()` function is overridden, which creates the `ViewHolder` objects by calling the custom constructor function `ViewHolder.create()`. The `onBindViewHolder()` function is also overridden which gets the current turntable using `getItem()` and calls the `ViewHolder.bind()` function.

The `ViewHolder` class is defined inside the `MainRowAdapter` class which inherits `Recycler.ViewHolder`. The `ViewHolder.bind()` function is created which finds the row's TextViews and populates them with turntable data. It also calls `TurntableEntity.drawIcon()` and puts the returned `Drawable` resource into the `ImageView` of the row. The `bind()` function also creates the click listener which logs the event and calls back the `launchDetailActivity()` function in `MainActivity.kt`, passing the `TurntableEntity` as a parameter to be included as a parcelable intent extra.

The `ViewHolder` class has a companion object with a custom constructor function `create()` that is called from `MainRowAdapter.onCreateViewHolder()`. It takes the parent `ViewGroup` and the click listener callback function as parameters. It then uses a `LayoutInflator` to inflate the row layout from the `row_layout_main.xml` resource file and returns the `ViewHolder` object.

To implement the adapter into the main activity the function `setupRecyclerView()` was created which gets called from `onCreate().` The `RecyclerView` is fetched by ID and put into a variable so that its properties can be modified. A new `LinearLayoutManager` object is created and assigned to the `RecyclerView .layoutManager` property. The `MainRowAdapter` is then created passing the `launchDetailActivity()` function as the click listener callback. The adapter is then assigned to the `RecyclerView.adapter` property.

To update the `RecyclerView` when the database changes a `LiveData` observer is set up in `onCreate()` with a callback. This means that the `RecyclerView` will always show the updated list regardless of the source of `LiveData` changes. The `LiveData` variable `allTurntables` in the `turntableViewModel` has the `.observe()` method called on it which sets the lambda callback that is triggered whenever the data changes. Inside the callback the `adapter.submitList()` function is called to send the updated `List<TurntableEntity>` to the list adapter which will update the `RecyclerView`.



4:10
Victorian Turntables

Bendigo Turntable
Bendigo Loco Depot Roundhouse
In Use
70'
Electric

Geelong Turntable
Geelong Loco Depot
In Use
70'
Electric

South Dynon BG Turntable
South Dynon Loco Depot
In Use
70'
Electric

South Dynon SG Turntable
South Dynon Loco Depot
In Use
70'
Electric

Bacchus Marsh Turntable
Bacchus March Station
Heritage Mainline Use
70'
Manual

Ballarat East Turntable
Steamrail Ballarat East
Heritage Mainline Use
70'
Electric

Dimboola Turntable
Steamrail Dimboola
Heritage Mainline Use
70'
Manual

Echuca Turntable
Echuca Station
Heritage Mainline Use
70'
Ma

Newport Workshops
70'

4:11
Victorian Turntables

Heritage Mainline Use
Manual

Newport Workshops Turntable
Newport Workshops - West Block
Heritage Mainline Use
70'
Manual

SRHC Seymour Turntable
Seymour Railway Heritage Centre - Seymour
Heritage Mainline Use
70'
Electric

Swan Hill Turntable
Swan Hill Station
Heritage Mainline Use
70'
Manual

Traralgon Turntable
Traralgon Loco Depot Roundhouse
Heritage Mainline Use
70'
Electric

Warrnambool Turntable
Warrnambool Station
Heritage Mainline Use
70'
Petrol

Wycheproof Turntable
Wycheproof Station
Heritage Mainline Use
70'
Manual

DSCR Daylesford Turntable
Daylesford Spa Country Railway - Daylesford Station
Heritage Line Use
70'
Manual

PBR Emerald Turntable
53'

4:12
Victorian Turntables

Daylesford Station
Heritage Line Use
Manual

PBR Emerald Turntable
Puffing Billy Railway - Emerald Station
Heritage Line Use
53'
Manual

VGR Castlemaine Turntable
Victorian Goldfields Railway - Castlemaine Station
Heritage Line Use
70'
Manual

VGR Maldon Turntable
Victorian Goldfields Railway - Maldon Station
Heritage Line Use
53'
Manual

YVR Healesville Turntable
Yarra Valley Railway - Healesville Station
Heritage Line Use
70'
Manual

BR Drysdale Turntable
Bellarine Railway - Drysdale Station
Heritage Line Disused
70'
Manual

BR Queenscliff Turntable
Bellarine Railway - Queenscliff Station
Heritage Line Disused
53'
Manual

SGR Korumburra Turntable
South Gippsland Railway - Korumburra Station
Heritage Line Disused
70'
Ma

fig.7 - Main activity RecyclerView list showing custom icons and turntable identifying data

Gap 7 - Create options menu actions for main activity and detail activity

To create options menu actions in the main activity I first used the Asset Studio tool to create the icon vector resources in the `/res/drawable` folder. The icons were tinted with the `?attr/ColorOnPrimary` theme attribute so they will appear the correct colour in the action bar.

The menu resources were then created in the `/res/menu` folder in the files `options_menu_main.xml` and `options_menu_detail.xml`. Inside the `<menu>` tags the options are defined in `<item>` tags with the attributes assigning an id, the icon resource and a title externalised to `strings.xml`. Actions which should appear in the action bar have the `app:showAsAction` property set to `ifRoom,` while actions in the dropdown menu are set to `never`.

The menus are implemented into the main and detail activities by overwriting two functions. The `onCreateOptionsMenu()` function contains a `MenuInflater` which implements the options from the menu resource file into the activity. The `onOptionsItemSelected()` function is run when an option is selected and using a `when` block can run the correct action matching the option’s ID.

The add option in the main activity calls `launchAddNewTurntable()` which launches the edit detail activity with an `Intent` object including the `"method"` string extra `"add".` The edit option in the detail activity calls `launchEditTurntable()` which also starts the edit detail activity but with the `"method"` string extra `"edit"` and the `"turntable"` parcelable extra with the `TurntableEntity` object. A screenshot of the add and edit action bar options is included in *fig.8.*

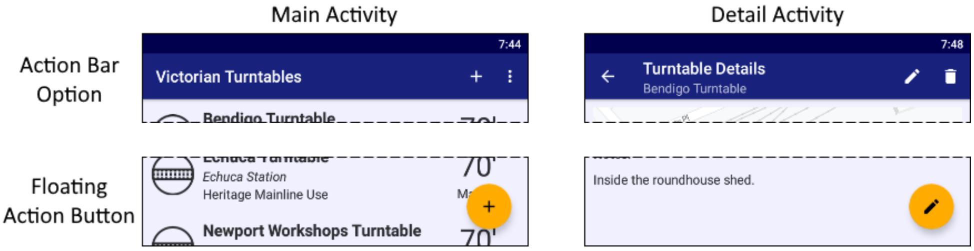
The “delete all” and “reset data” options in the main activity are found inside the kebab menu in the top right of the screen. These are irreversible and destructive actions so confirmation alert dialogs are shown to the user before the corresponding action is performed, which are implemented in *Gap 9*. The “reset data” option launches the `readDefaultTurntableData()` function in a coroutine to avoid blocking the UI thread, as explained in *Gap 5*. The “delete all” option calls `TurntableViewModel.deleteAll()` to clear the database.

To implement action bar back buttons in the top left corner of the screen the action bar had to be modified in the `onCreate()` function to call `supportActionBar.setDisplayHomeAsUpEnabled(true)`. Both the detail activity and edit detail activity implement these back buttons to go back to the previous activity. The `when` block in `onOptionsItemSelected()` has a case added for `android.R.id.home` which calls the function `onBackButtonPressed()`, which is the same function that is called when the device back button is pressed.

Gap 8 - Create floating action buttons for main activity and detail activity

To implement a floating action button (FAB) into the main activity and detail activity first the layout resources `/res/layout/activity_main.xml` and `/res/layout/activity_detail.xml` had to be modified. To make the FAB display on top of the other views the parent view of the layout must be a `CoordinatorLayout`, which is a view derived from `FrameLayout` that has additional support for child view behaviours and functionality. For the main activity this only required changing the `LinearLayout` parent view to a `CoordinatorLayout`, and for the detail activity I wrapped the whole existing layout code in a new `CoordinatorLayout` view. I added the FAB views to the bottoms of the layout XML files, making sure the FAB views are direct children of the `CoordinatorLayout`. The icons are the same add and edit icons previously used for the action bar options menu items. Styles for the floating action button were externalised to the file `/res/values/style_fab.xml` and applied to the FAB views using the `style="@style/style_fab"` attribute.

To give the action buttons functionality a click listener is created in the `onCreate()` function of the appropriate activity, using `findViewById()` to fetch the FAB view using its ID. The add FAB click listener in the main activity calls `launchAddNewTurntable()` and the edit FAB in the detail activity calls `launchEditTurntable()`, which both launch the edit detail activity with the appropriate `intent` extras as explained in *Gap 7*. A screenshot of the add and edit floating action buttons is included in *fig.8*.



Inside the roundhouse shed.

fig.8 - Action bar options and floating action buttons for add and edit functions in main and detail activities

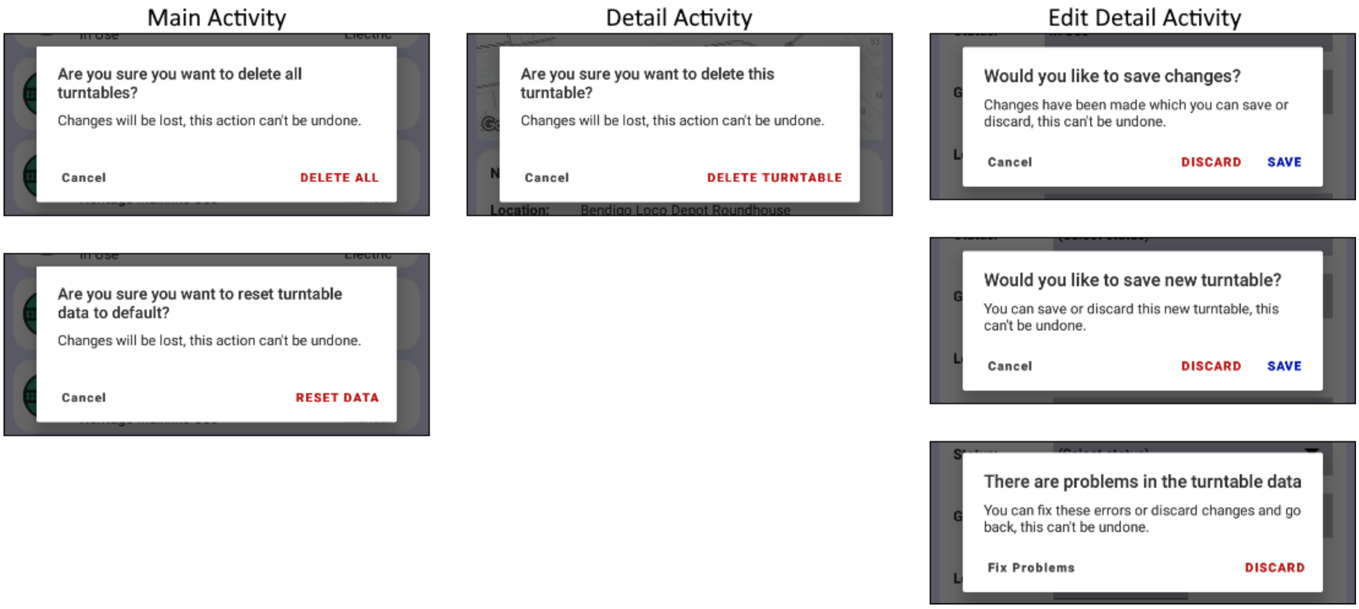
Gap 9 - Implement confirmation alert dialogs for irreversible user actions

For certain irreversible actions like deleting all turntables, resetting the database to the default values or deleting a turntable I've implemented confirmation dialogs to reduce accidental actions. The confirmation utilises an `AlertDialog` which asks the user if they're sure they want to perform the action, explains the options and reminds the user that the action can't be undone. The user can choose to cancel and go back, or choose to confirm the action. When the user presses the back button in the edit detail activity the alert dialog lets the user choose to save or discard changes or cancel the dialog and remain in the activity.

Alert dialogs buttons are defined as positive, negative or neutral rather than being defined positionally as left, center or right buttons. By defining the buttons this way they will appear in the correct place that the user expects as usual button order can vary between devices, OS versions and languages.

The alert dialog buttons are styled to easily differentiate the positive, negative and neutral buttons. I used red text for negative buttons following good UX design practice for destructive actions like deletion. Also following conventions from other apps I used blue text for the positive save buttons and grey text for the neutral cancel buttons. The alert dialog styles are externalised to the file `/res/values/style_alert_dialog.xml`, and are applied to the app using the `alertDialogTheme` attribute in the `themes.xml` files.

To implement the alert dialogs an `AlertDialog.Builder` object is created with the activity context as a parameter. To set the alert dialog title and message the `.setTitle()` and `.setMessage()` methods are called on the builder object. To set the buttons text and actions the `.setPositiveButton()`, `.setNegativeButton()` and `.setNeutralButton()` methods are used, which each take the button text string and a lambda callback for when the button is clicked. For the neutral cancel button a blank lambda is used, and for the positive and negative button the appropriate function for the action is called inside the lambda. Finally the `.show()` method is called on the builder which creates the `AlertDialog` itself and displays it to the user.



Main Activity

Are you sure you want to delete all turntables?

Changes will be lost, this action can't be undone.

Are you sure you want to reset turntable data to default?

Changes will be lost, this action can't be undone.

Detail Activity

Are you sure you want to delete this turntable?

Changes will be lost, this action can't be undone.

Edit Detail Activity

Would you like to save changes?

Changes have been made which you can save or discard, this can't be undone.

Would you like to save new turntable?

You can save or discard this new turntable, this can't be undone.

There are problems in the turntable data

You can fix these errors or discard changes and go back, this can't be undone.

fig.9 - Alert dialogs implemented for confirmation of irreversible and destructive actions

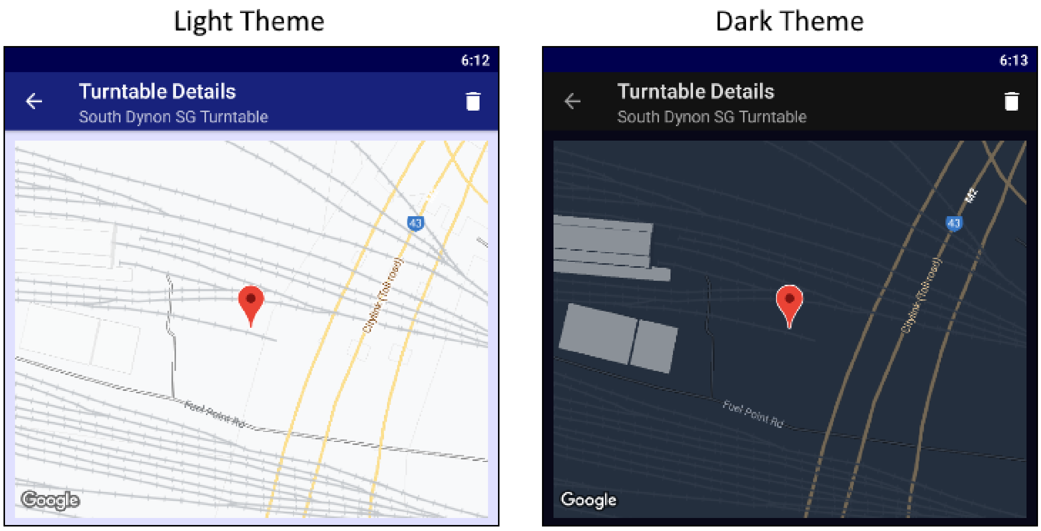
Gap 10 - Implement Google Maps fragment into detail activity

To implement the Google Maps fragment into the detail activity I followed the Google Maps SDK documentation. In the layout resource `/res/layout/activity_detail.xml` the `FragmentContainerView` for the map fragment must have the android:name attribute set to `com.google.android.gms.maps.SupportMapFragment`. This view is a container for the map fragment that will be generated by code.

The `DetailActivity` class was modified to extend the `OnMapReadyCallback` class which adds functionality for loading the map fragment asynchronously. A class scope lateinit variable `turntableMap` is created to store the GoogleMap object once it is created. In `onCreate()` the fragment is fetched by ID using `supportFragmentManager.findFragmentById()` which gets typecasted to a `SupportMapFragment`. The `.getMapAsync()` method is then called on the `SupportMapFragment` which sets up the map in a background thread and calls back the `onMapReady()` function with the `GoogleMap` object when it's done. The `onMapReady()` function is overridden to put the `GoogleMap` object into the `turntableMap` class variable, then create a marker at the latitude and longitude from the `TurntableEntity`, and finally move and zoom the camera.

To use the Google Maps SDK in an app an API key must be created in the Google Developer Console linked to a Google account. As this SDK is a paid service (though I'm using the 90 day free trial) it is important to keep API keys secret to avoid key theft. In Android this is done by putting the API key string as a variable `MAPS_API_KEY` into the `local.properties` file, which doesn't get committed to GitHub. The key can then be included in the `AndroidManifest.xml` file in a `<metadata>` tag with the name attribute `com.google.android.geo.API_KEY` and the value attribute using the string reference `${MAPS_API_KEY}`. This keeps API keys separate from the source code, while allowing them to be included in the compiled app. API keys can also be limited to specific application IDs in Google Developer Console so even if stolen by a malicious party they can't be used in a different app.

To style the map fragments I used the Google Maps JSON Styling Wizard tool to create two map style JSON files for light and dark themes. With the advanced settings I increased the weight of relevant features like railway lines and stations, while hiding irrelevant points of interest like businesses and restaurants. I added the JSON light style to the project in the string resource file `/res/values/google_maps_style_json.xml` which has a single string resource containing the JSON data. I then created another file with the same name in the `/res/values-night` folder containing the night theme JSON style. Putting the JSON data into a string resource rather than a `.json` file in `/res/raw` allows the correct map style to be selected based on the device theme with no additional code as resource selection is handled by the Material Design library.


Light Theme

Dark Theme

fig.10 - Google Maps fragment in the detail activity for light and dark themes

Gap 11 - Implement spinner dropdown inputs in edit detail activity

In the edit detail activity some of the user inputs should be dropdown menus with a list of options, which in android are called spinners. To implement the spinners into the activity layout the `<spinner>` tag is used and given an ID so its functionality can be handled later in code. To make the spinner's appearance match the `TextInputEditText` views in the activity a background drawable resource is created in the file `/res/drawable/bg_spinner.xml` and applied to the spinner `android:background` attribute. The drawable contains the input background color from the theme attributes and the down arrow showing the user that it can be expanded.

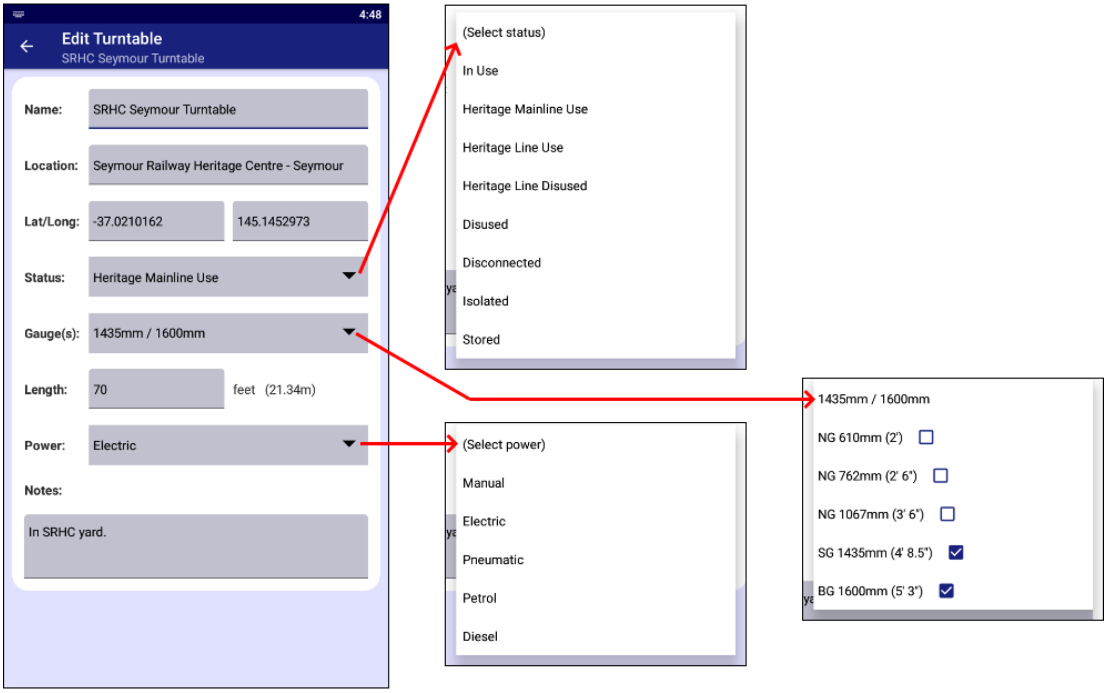
The options for the spinners are stored in string array resources in a new file `/res/values/spinner_string_arrays.xml`. The string arrays are defined using the `<string-array>` tag and the strings inside defined with `<item>` tags, which referer to strings from the `strings.xml` file in preparation for potential future localisation.

For the status and power spinners the `ArrayAdapter` class was used for a standard spinner implementation. The `createSpinners()` function was created which is called from `onCreate()` and sets up the spinner adapter. The `ArrayAdapter.createFromResource()` constructor is used to create the spinner adapter from the string array resource and the row layout resource. After creation the adapters are assigned to lateinit class scope variables to be accessed again later, the adapter is assigned to the `Spinner.adapter` property, and the callback is set. To enable spinner callbacks the `EditDetailActivity` class must extend `AdapterView.OnItemSelectedListener` and override the abstract functions `onItemSelected()` and `onNothingSelected()`, though the latter is unused in this app it must still be overridden. The function `onItemSelected()` is called back whenever any spinner in that activity is selected and by using a `when` block the callbacks for the power and status spinners can be differentiated by the spinners' IDs. Inside the `when` case the selected value string is assigned to a class level variable which stores the currently selected value. The values in these class level variables stausSelected and powerSelected will be used when creating the new `TurntableEntity` object for saving changes.

For the gauges spinner I needed a multiselect dropdown input with check boxes to select one or multiple gauges. As there isn't an existing multi-select spinner adapter for Android I had to make a custom adapter for this purpose. I found a solution for a custom adapter on StackOverflow but it was written in Java, so I had to translate it into Kotlin and modify the functionality to my needs.

A custom layout was created in the file `/res/layout/spinner_checkbox_item.xml` which uses a horizontal `LinearLayout` that contains the `TextView` and `Checkbox` views that the spinner items will use. The file `GaugeSpinnerAdapter.kt` contains all of the Kotlin code that the custom spinner uses. A simple data class `GaugeState` is created which holds a string value for the gauge title and a boolean selected value. The adapter will use an `ArrayList<GaugeState>` in the class scope variable `listState` to keep track of the currently selected gauges.

The adapter class `GaugeSpinnerAdapter` was created in the file which extends the `ArrayAdapter` class. Inside the adapter class a ViewHolder is implemented which stores references to the TextView and Checkbox views. I created my own `createFromResource()` constructor function in a companion object so that creating the adapter in the activity can be done similarly to the other spinners that use the default adapter. This constructor takes the same parameters as the default `ArrayAdapter.createFromResource()` then constructs and returns the `GaugeSpinnerAdapter` object. The `getView()` and `getDropdownView()` functions are both overridden which are used to get the views of the selected item in the spinner and the items in the dropdown list respectively. Both of these functions call the function `getCustomView()` which implements the custom functionality and returns the views. When a checkbox's state changes an `OnCheckedChangeListener` detects this and updates the `listState` variable with the selected gauges. A human readable string of the selected gauges is also generated and put into the first option of the menu so it is visible when the spinner dropdown is closed. When the turntable data is saved the function `getGaugeList()` is used to convert the `ArrayList<GaugeState>` used by the adapter to a `MutableList<Int>` that the `TurntableEntity` uses.



4:48

Edit Turntable
SRHC Seymour Turntable

Name: SRHC Seymour Turntable

Location: Seymour Railway Heritage Centre - Seymour

Lat/Long: -37.0210162 145.1452973

Status: Heritage Mainline Use

Gauge(s): 1435mm / 1600mm

Length: 70 feet (21.34m)

Power: Electric

Notes:

In SRHC yard.

(Select status)
In Use
Heritage Mainline Use
Heritage Line Use
Heritage Line Disused
Disused
Disconnected
Isolated
Stored

1435mm / 1600mm
NG 610mm (2') □
NG 762mm (2' 6") □
NG 1067mm (3' 6") □
SG 1435mm (4' 8.5") ☑
BG 1600mm (5' 3") ☑

(Select power)
Manual
Electric
Pneumatic
Petrol
Diesel

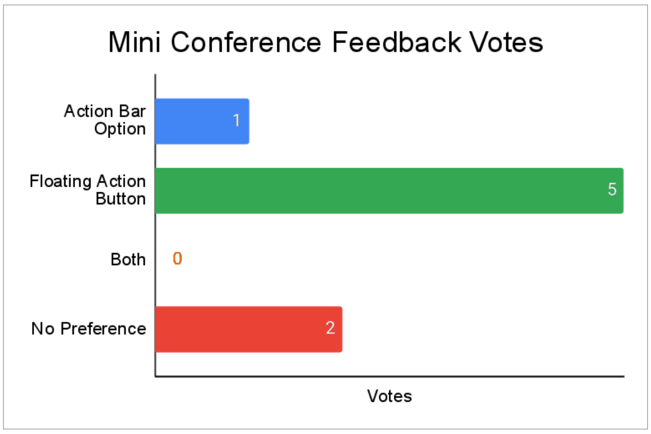
fig.11 - Edit detail activity spinner inputs for status, power and gauges

Mini Conference

Mini Conference Feedback Results

The main feedback I was seeking from the mini conference was the preferences of my peers between an action bar option and a floating action button (see *Gap 7* and *Gap 8*). For the purposes of development and demonstration I had implemented both in the main and detail activities for the add and edit functionality respectively (*fig.8*). These were showcased in my mini conference pitch video and all feedback comments were read and preferences totaled in a spreadsheet. Although the results were clear from the vote totals I created a graph (*fig.12*) to display the results.

The results of the feedback clearly showed a preference for the floating action button, which was also what I preferred. Given the location at the bottom of the screen a FAB is much easier to reach on a mobile phone held in the portrait orientation, making it preferable over an action bar option. Based on this feedback I then removed the add and edit action bar options from the app, leaving the floating action buttons as the user interfaces for adding and editing turntables.



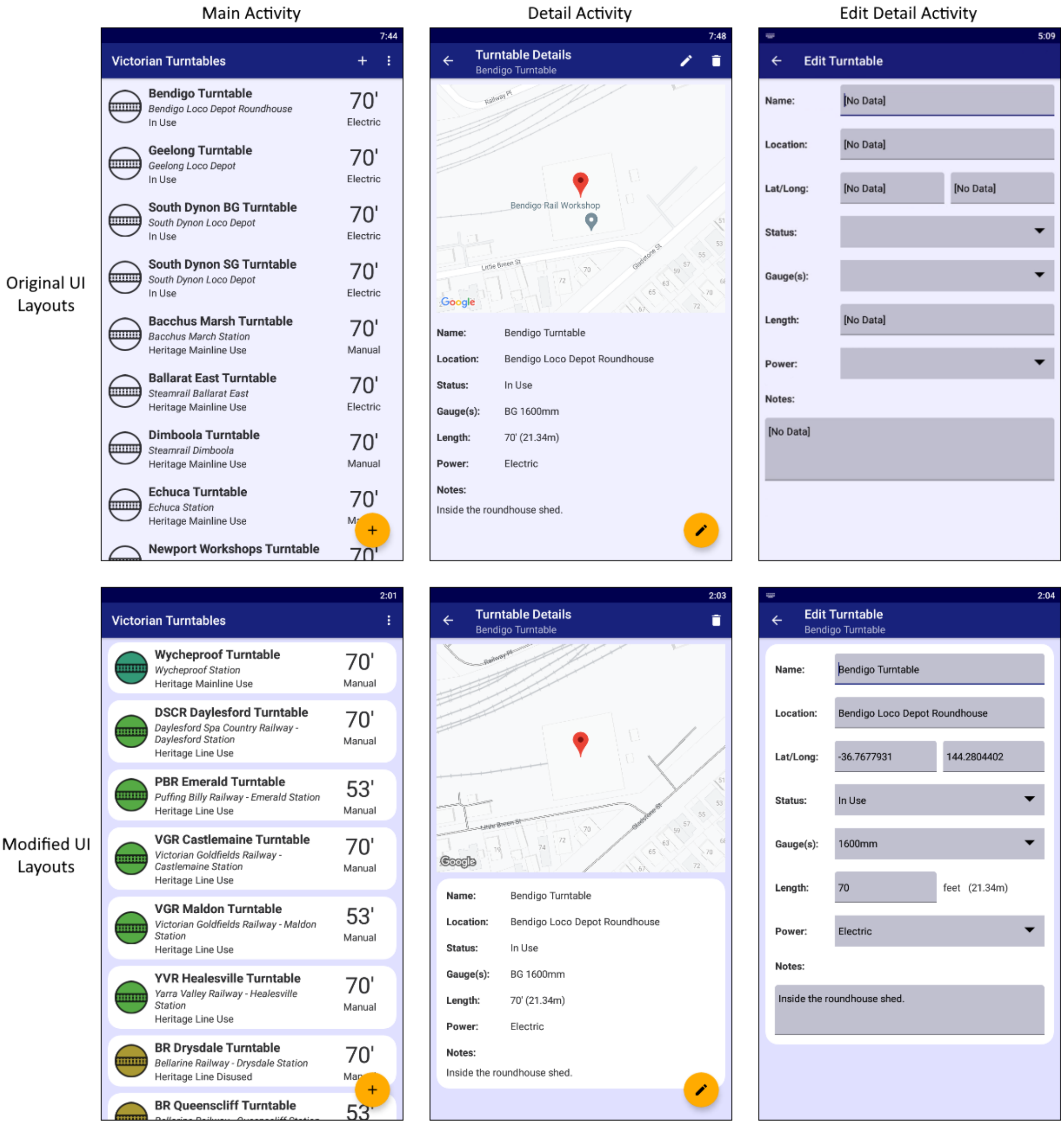
.

fig.12 - Graph of votes from mini conference showing preferences between action bar options or FABs

UI Improvements Inspired by Other Custom Apps

Comparing my custom app to other students I noticed my UI was looking a bit flat and boring. A design feature I noticed in several custom apps was the use of rounded rectangle sections showing the content on top of a colored background. This design pattern makes the sections of the app easier to differentiate, gives the UI more dimensionality and maximises text contrast and readability while still incorporating colors.

To implement rounded rectangle sections into my app I created a drawable resource `/res/drawable/bg_rounded_rect.xml` which uses `<fill>` to set the appropriate white or black background color based on the `?attr/inputBackgroundColor` theme attribute. The `<corners>` tag is used to round the corners of the shape with the corner radius size externalised in `/res/values/dimens.xml`. This drawable resource could then be applied to the `android:background` attribute of the view styles that should have the background. For the main activity this was applied to the row layout with appropriate padding so each turntable would be in its own section in the `RecyclerView` list. For the detail and edit detail activities the rounded rectangle background was applied to the `TableLayouts` so that the entire turntable data is inside the section. This also required some adjustment of the padding values in `dimens.xml`, and the app background color was adjusted to be slightly more saturated to contrast with the rounded rectangle sections. Screenshots comparing the original UI layout before the mini conference to the updated layout with the rounded sections are included in *fig.13* on the next page.



Victorian Turntables

Bendigo Turntable *Bendigo Loco Depot Roundhouse* In Use	70'	Electric
Geelong Turntable *Geelong Loco Depot* In Use	70'	Electric
South Dynon BG Turntable *South Dynon Loco Depot* In Use	70'	Electric
South Dynon SG Turntable *South Dynon Loco Depot* In Use	70'	Electric
Bacchus Marsh Turntable *Bacchus March Station* Heritage Mainline Use	70'	Manual
Ballarat East Turntable *Steamrail Ballarat East* Heritage Mainline Use	70'	Electric
Dimboola Turntable *Steamrail Dimboola* Heritage Mainline Use	70'	Manual
Echuca Turntable *Echuca Station* Heritage Mainline Use	70'	M
Newport Workshops Turntable	70'	

Turntable Details
Bendigo Turntable

Name: Bendigo Turntable
Location: Bendigo Loco Depot Roundhouse
Status: In Use
Gauge(s): BG 1600mm
Length: 70' (21.34m)
Power: Electric
Notes:
Inside the roundhouse shed.

Edit Turntable

Name: [No Data]
Location: [No Data]
Lat/Long: [No Data] [No Data]
Status:
Gauge(s):
Length: [No Data]
Power:
Notes:
[No Data]

Victorian Turntables

Wycheproof Turntable *Wycheproof Station* Heritage Mainline Use	70'	Manual
DSCR Daylesford Turntable *Daylesford Spa Country Railway - Daylesford Station* Heritage Line Use	70'	Manual
PBR Emerald Turntable *Puffing Billy Railway - Emerald Station* Heritage Line Use	53'	Manual
VGR Castlemaine Turntable *Victorian Goldfields Railway - Castlemaine Station* Heritage Line Use	70'	Manual
VGR Maldon Turntable *Victorian Goldfields Railway - Maldon Station* Heritage Line Use	53'	Manual
YVR Healesville Turntable *Yarra Valley Railway - Healesville Station* Heritage Line Use	70'	Manual
BR Drysdale Turntable *Bellarine Railway - Drysdale Station* Heritage Line Disused	70'	Ma
BR Queenscliff Turntable	53'	

Turntable Details
Bendigo Turntable

Name: Bendigo Turntable
Location: Bendigo Loco Depot Roundhouse
Status: In Use
Gauge(s): BG 1600mm
Length: 70' (21.34m)
Power: Electric
Notes:
Inside the roundhouse shed.

Edit Turntable
Bendigo Turntable

Name: Bendigo Turntable
Location: Bendigo Loco Depot Roundhouse
Lat/Long: -36.7677931 144.2804402
Status: In Use
Gauge(s): 1600mm
Length: 70 feet (21.34m)
Power: Electric
Notes:
Inside the roundhouse shed.

fig.13 - Original UI layout before the mini conference versus the modified UI layout inspired by other students' custom app designs

Open Issues and Recommendations

Issue 1 - No other languages have been implemented

Due to my lack of speaking a second language my custom app has only been implemented with the default English language. I have demonstrated my ability to implement localisation in previous tasks, and given the nature of this app specific for Victoria I felt additional languages weren't necessary. Following android development conventions I have still externalised all strings in the app to the `/res/values/strings.xml` file, which means additional languages could be easily added with the Android Studio translation tool. If I do continue to develop this app in the future and expand the scope I might consider implementing other languages.

Issue 2 - No unit tests implemented

Due to time constraints I did not have the chance to implement any unit testing for my custom app. The complexity of my app would necessitate quite a lot of test cases to ensure all functionality can be tested. As I have demonstrated my abilities to design and implement unit tests in previous tasks, especially in the Core 3 Assignment, I felt it was not necessary to include unit tests for the scope of this project. If I do continue to work on this app outside of this unit I may consider adding unit tests to ensure everything works when adding features without manually testing every scenario.

Issue 3 - Google Maps API key is not shared, also free trial will expire

To protect Google Maps API keys from misuse they are defined in the `local.properties` project file that is not committed to GitHub, as explained in *Gap 10*. If the source code is downloaded and built without adding an API key to the `local.properties` file the map fragment in the detail activity will not load. Although all other functionality should still work there is no guarantee that this won't cause unexpected issues. Instructions for adding an API key to the project are included in the `README.md` file of the GitHub repository.

The Google Maps SDK is also a paid service, so although the 90 day free trial has lasted me through the development and assessment period I do not intend to start paying for the service. To use a map fragment in the future without paying I would need to implement a suitable substitute to Google Maps, for example OpenStreetMaps. This would require a decent amount of refactoring as the new SDKs would use very different syntax, however migration guides exist to assist developers in making the switch.

Issue 4 - Visual bug in Edit Detail Activity

There is a small visual glitch in the edit detail activity where the relative widths of the labels column and inputs column can slightly change as the spinner dropdown inputs are changed. This however does not affect the functionality in any way and is purely visual. I tried quite a few styling configurations to mitigate this issue but none of the solutions I tried fixed it. Whilst I could have hardcoded the width of the inputs as absolute values this would sacrifice the flexibility of the layout for different devices screen size and landscape orientation. If I continue to work on this app outside of the unit I may try again to fix this problem.

Issue 5 - Generated tinted icons are not cached

In the current version the tinted turntable icons for the main activity are generated when the row adapter `.bind()` function is run. This means that the tinted icons are only generated as needed when scrolling through the `RecyclerView` list. Whilst the results of the extension performance study showed that this method was more efficient than generating all icons at the app launch, it is still not the most optimised way to do it for this app.

As there is only a limited amount of status options that dictate the tint color, many rows are re-generating the same icons that could instead be reused. The naive approach would be to pre-generate icons for each required in `/res/drawable` and select the corresponding resource for each row based on the status. This method reduces the flexibility of the icon drawing and would make modifying colors or adding new statuses more difficult. A better solution would be to dynamically tint and generate the icon like currently implemented, but to cache the generated icons so that when they are requested again the cached version can be returned and the icon generation step skipped. I found a solution by GitHub user AnixPasBesoin which caches tinted icons [https://gist.github.com/AnixPasBesoin/1b29fc61c85698526ab902974f7c0aa1]. Although it is written in Java I could translate the code into Kotlin and adapt it for my purposes, which I may do if I continue to develop this app later.

Appendix 1 - Tools and Resources Used

Appx. 1.1 - Software and Tools:

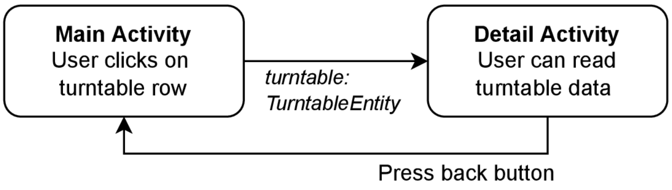
- Android Studio
- BlueStacks (Android emulator)
- Material.io Color Tool [https://material.io/resources/color/]
- InkScape (vector graphics editor)
- Google Maps Android SDK
- Google Maps JSON Styling Wizard [https://mapstyle.withgoogle.com/]

Appx. 1.2 - External Resources:

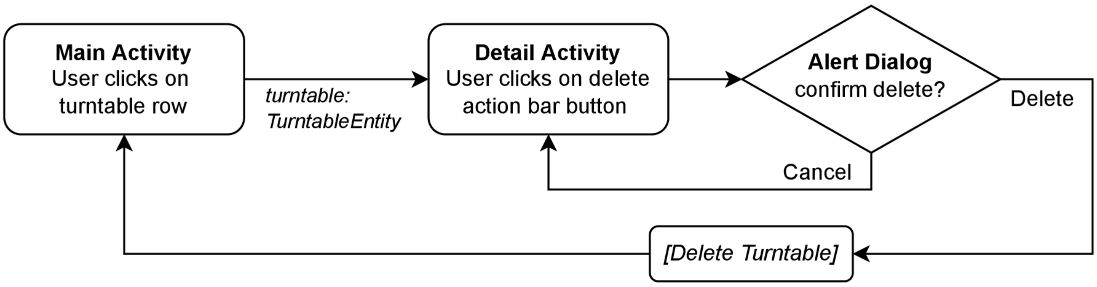
- *Android Developer:* Create app icons with Image Asset Studio [https://developer.android.com/studio/write/image-asset-studio]
- *Android Developer CodeLabs:* Room with a View [https://developer.android.com/codelabs/android-room-with-a-view-kotlin]
- *Android Developer:* RecyclerView [https://developer.android.com/develop/ui/views/layout/recyclerview]
- *Android Developer:* Dialogs [https://developer.android.com/develop/ui/views/components/dialogs]
- *Google Developers:* Google Maps SDK for Android [https://developers.google.com/maps/documentation/android-sdk/overview]
- *Google Developers:* Add a Styled Map [https://developers.google.com/maps/documentation/android-sdk/styling]
- *Google Developer:* Spinners [https://developer.android.com/develop/ui/views/components/spinner]
- Harshad Pansuriya; Answer to “Android spinner dropdown checkbox”, *Stack Overflow;* [https://stackoverflow.com/a/38418249]
- Canvas modules for weeks 1-10
- Previous weeks’ Android projects

Appendix 2 - UML Activity Diagrams

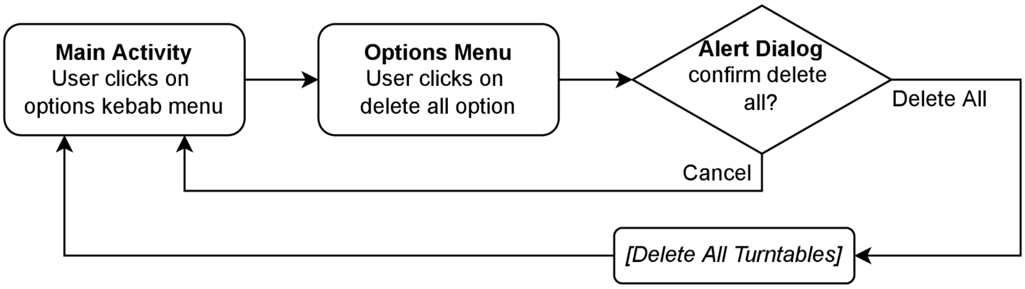
Appx. 2.1 - Read data from an existing turntable item in the database


Main Activity
User clicks on turntable row
turntable: TurntableEntity
Detail Activity
User can read turntable data
Press back button

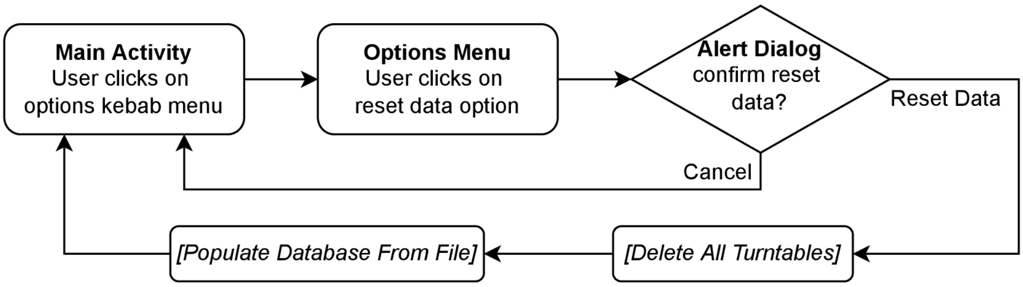
Appx. 2.2 - Delete an existing turntable item from the database


Main Activity
User clicks on turntable row
turntable: TurntableEntity
Detail Activity
User clicks on delete action bar button
Alert Dialog
confirm delete?
Delete
Cancel
[Delete Turntable]

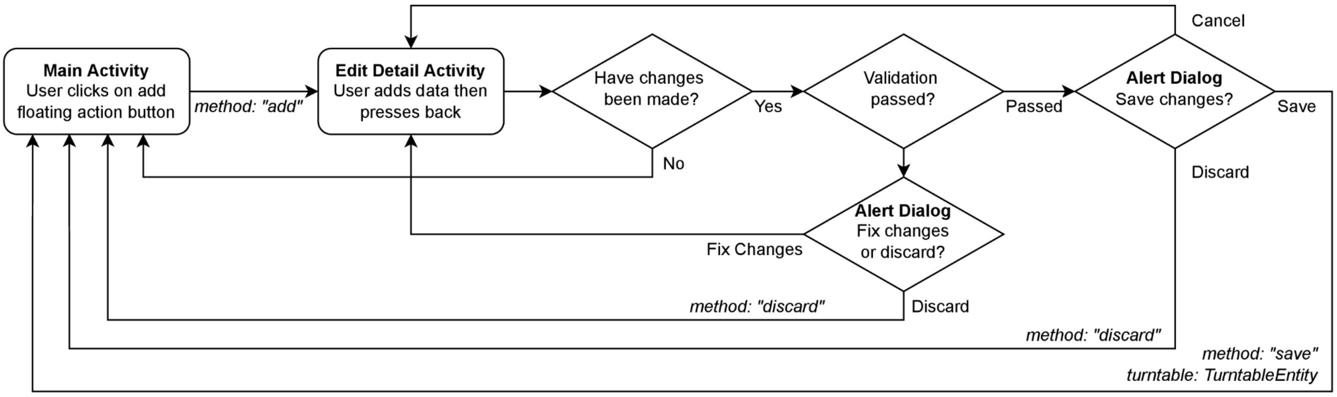
Appx. 2.3 - Delete all turntable items from the database



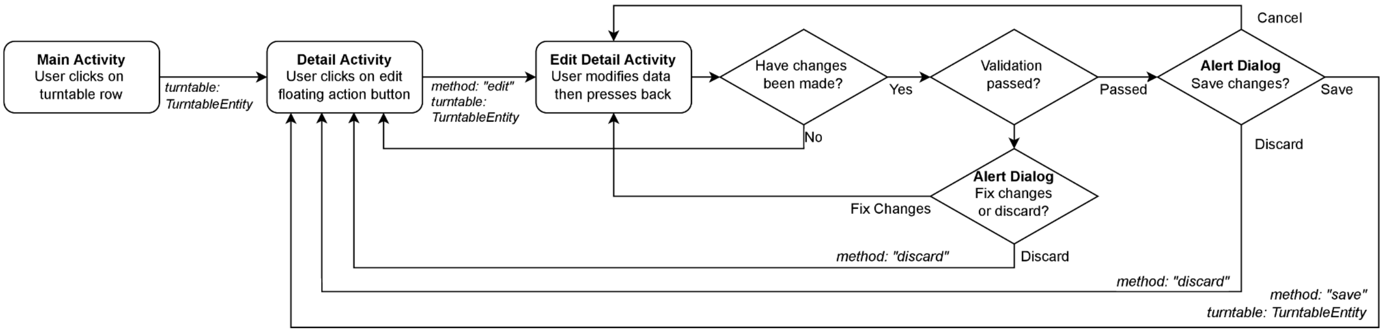
Appx. 2.4 - Reset the database to the default turntable items from file



Appx. 2.5 - Create a new turntable item in the database

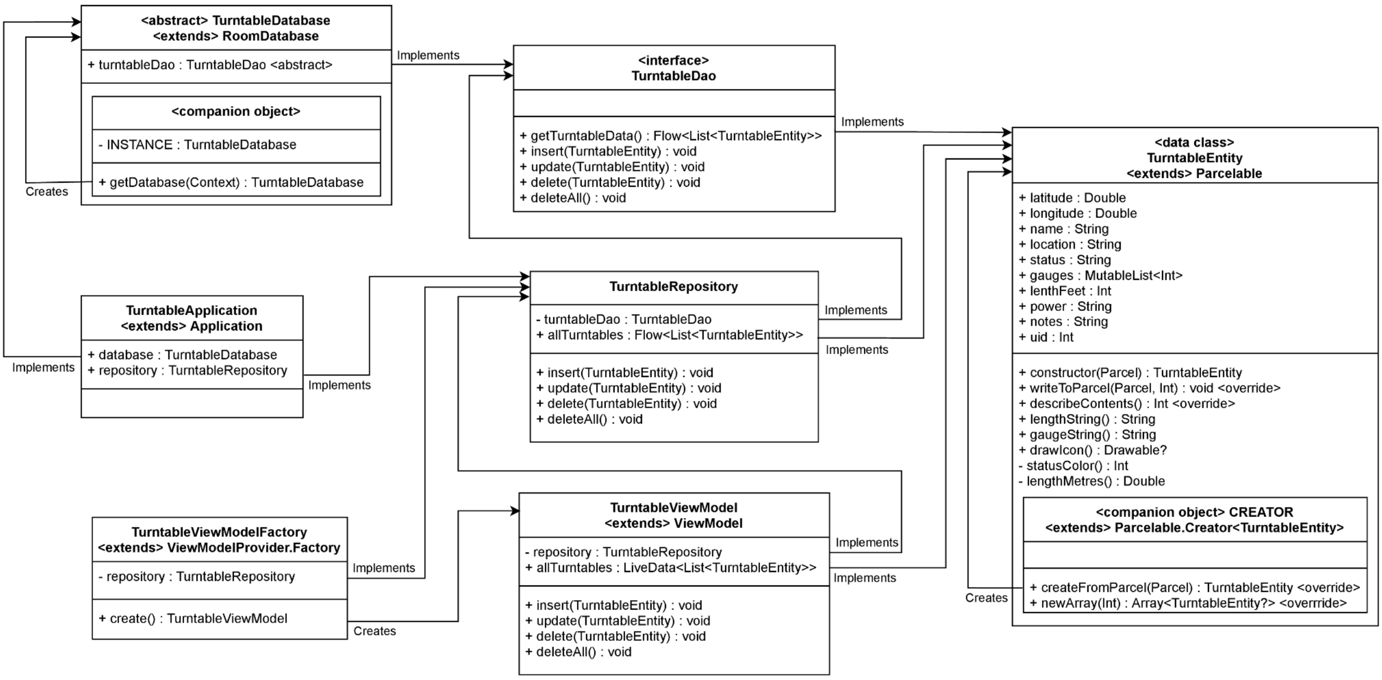


Appx. 2.6 - Update data in an existing turntable item from the database



Appendix 3 - UML Class Diagrams

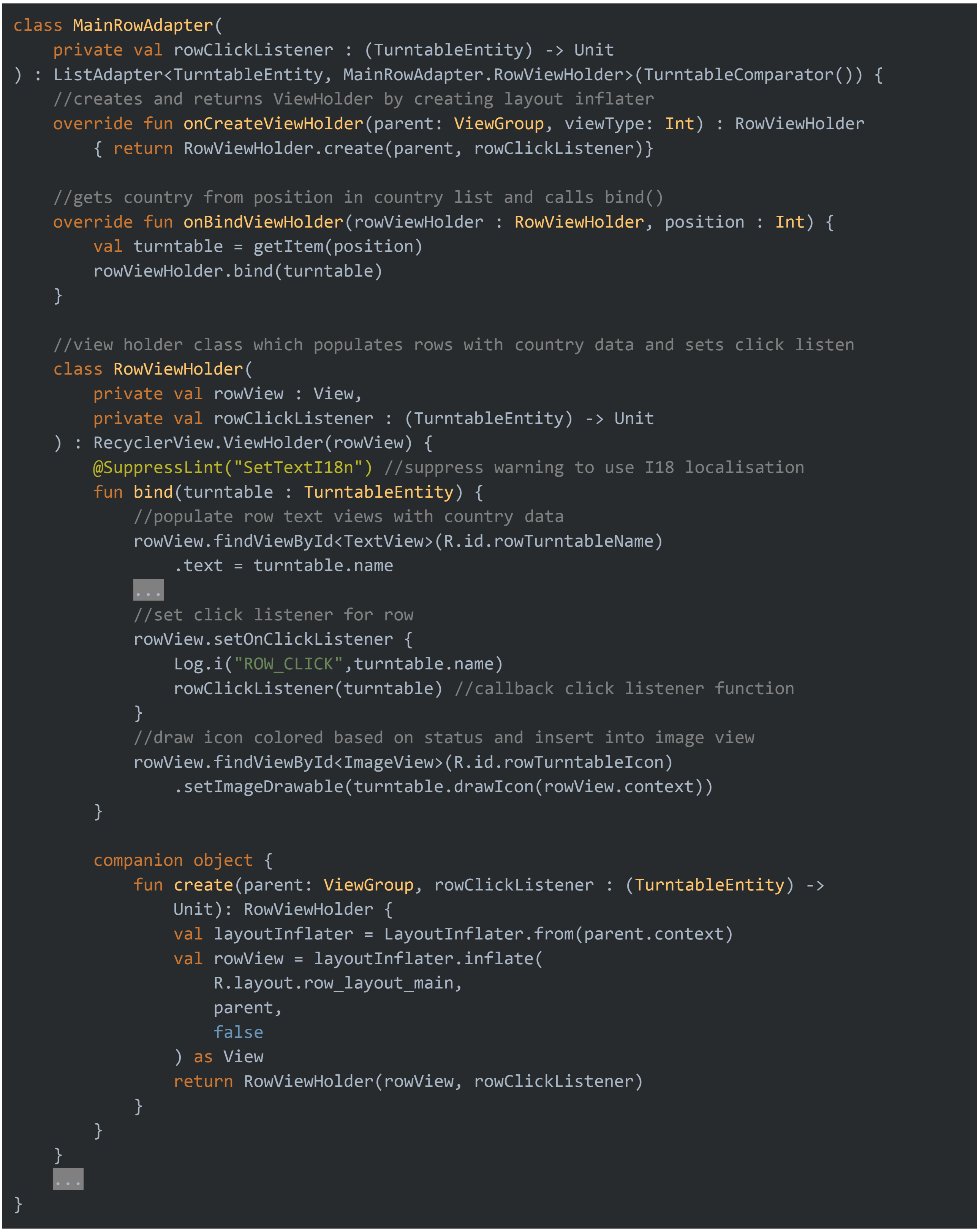
Appx. 3.1 - Room database classes and interfaces


<abstract> TurntableDatabase
<extends> RoomDatabase
+ turntableDao : TurntableDao <abstract>
<companion object>
- INSTANCE : TurntableDatabase
+ getDatabase(Context) : TurntableDatabase
Creates
Implements
<interface>
TurntableDao
+ getTurntableData() : Flow<List<TurntableEntity>>
+ insert(TurntableEntity) : void
+ update(TurntableEntity) : void
+ delete(TurntableEntity) : void
+ deleteAll() : void
Implements
<data class>
TurntableEntity
<extends> Parcelable
+ latitude : Double
+ longitude : Double
+ name : String
+ location : String
+ status : String
+ gauges : MutableList<Int>
+ lenthFeet : Int
+ power : String
+ notes : String
+ uid : Int
+ constructor(Parcel) : TurntableEntity
+ writeToParcel(Parcel, Int) : void <override>
+ describeContents() : Int <override>
+ lengthString() : String
+ gaugeString() : String
+ drawIcon() : Drawable?
- statusColor() : Int
- lengthMetres() : Double
<companion object> CREATOR
<extends> Parcelable.Creator<TurntableEntity>
+ createFromParcel(Parcel) : TurntableEntity <override>
+ newArray(Int) : Array<TurntableEntity?> <overrride>
Creates
TurntableApplication
<extends> Application
+ database : TurntableDatabase
+ repository : TurntableRepository
Implements
Implements
TurntableRepository
- turntableDao : TurntableDao
+ allTurntables : Flow<List<TurntableEntity>>
+ insert(TurntableEntity) : void
+ update(TurntableEntity) : void
+ delete(TurntableEntity) : void
+ deleteAll() : void
Implements
Implements
TurntableViewModelFactory
<extends> ViewModelProvider.Factory
- repository : TurntableRepository
+ create() : TurntableViewModel
Implements
Creates
TurntableViewModel
<extends> ViewModel
- repository : TurntableRepository
+ allTurntables : LiveData<List<TurntableEntity>>
+ insert(TurntableEntity) : void
+ update(TurntableEntity) : void
+ delete(TurntableEntity) : void
+ deleteAll() : void
Implements
Implements

Appendix 4 - Code Snippets

Appx. 4.1 - RecyclerView adapter code

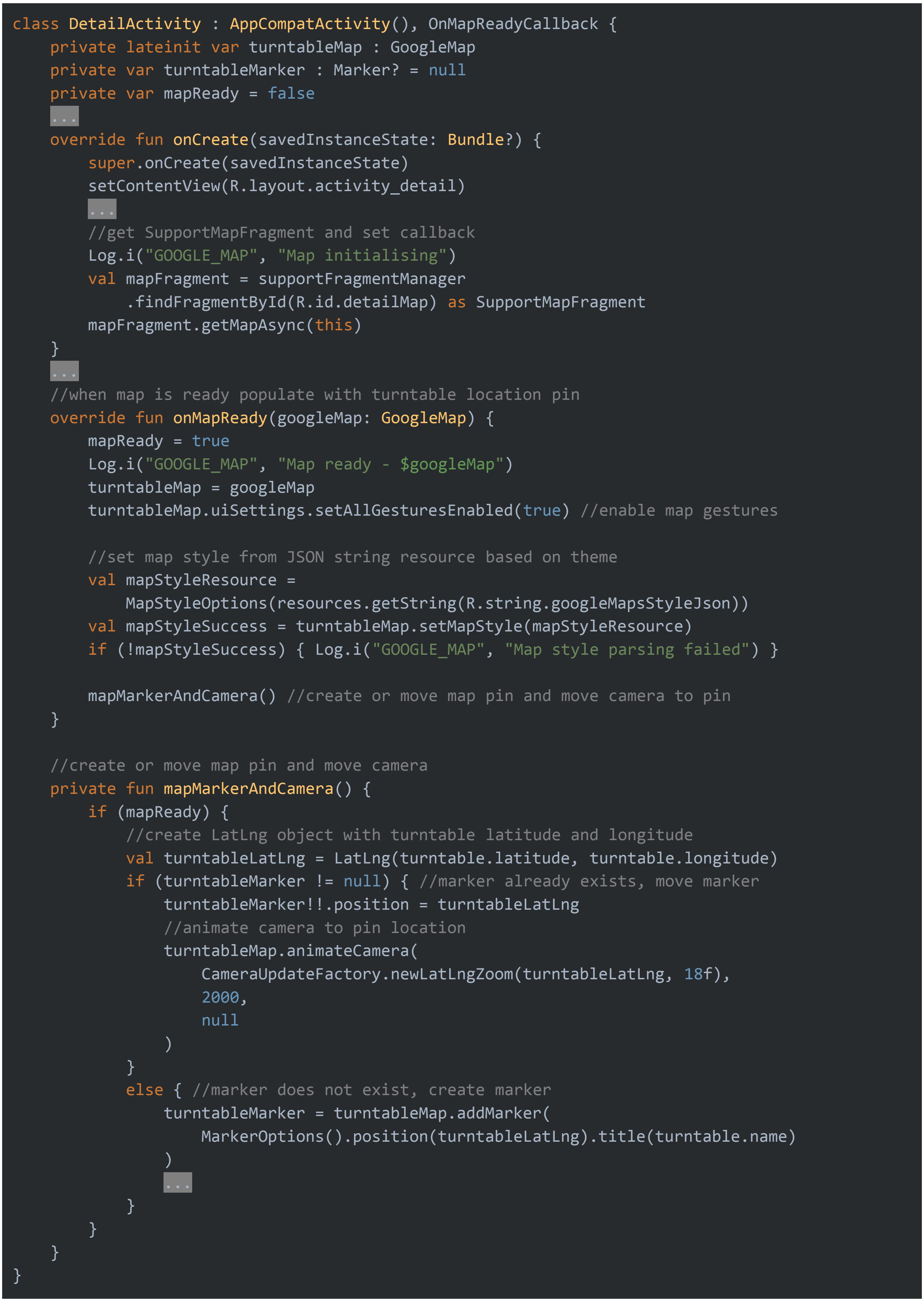
From MainRowAdapter.kt



```
class MainRowAdapter(
    private val rowClickListener : (TurntableEntity) -> Unit
) : ListAdapter<TurntableEntity, MainRowAdapter.RowViewHolder>(TurntableComparator()) {
    //creates and returns ViewHolder by creating layout inflater
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) : RowViewHolder
        { return RowViewHolder.create(parent, rowClickListener)}

    //gets country from position in country list and calls bind()
    override fun onBindViewHolder(rowViewHolder : RowViewHolder, position : Int) {
        val turntable = getItem(position)
        rowViewHolder.bind(turntable)
    }

    //view holder class which populates rows with country data and sets click listen
    class RowViewHolder(
        private val rowView : View,
        private val rowClickListener : (TurntableEntity) -> Unit
    ) : RecyclerView.ViewHolder(rowView) {
        @SuppressLint("SetTextI18n") //suppress warning to use I18 localisation
        fun bind(turntable : TurntableEntity) {
            //populate row text views with country data
            rowView.findViewById<TextView>(R.id.rowTurntableName)
                .text = turntable.name
            ...
            //set click listener for row
            rowView.setOnClickListener {
                Log.i("ROW_CLICK",turntable.name)
                rowClickListener(turntable) //callback click listener function
            }
            //draw icon colored based on status and insert into image view
            rowView.findViewById<ImageView>(R.id.rowTurntableIcon)
                .setImageDrawable(turntable.drawIcon(rowView.context))
        }

        companion object {
            fun create(parent: ViewGroup, rowClickListener : (TurntableEntity) ->
                Unit): RowViewHolder {
                val layoutInflater = LayoutInflater.from(parent.context)
                val rowView = layoutInflater.inflate(
                    R.layout.row_layout_main,
                    parent,
                    false
                ) as View
                return RowViewHolder(rowView, rowClickListener)
            }
        }
    }
    ...
}
```

Appx. 4.2 - Google Maps fragment code

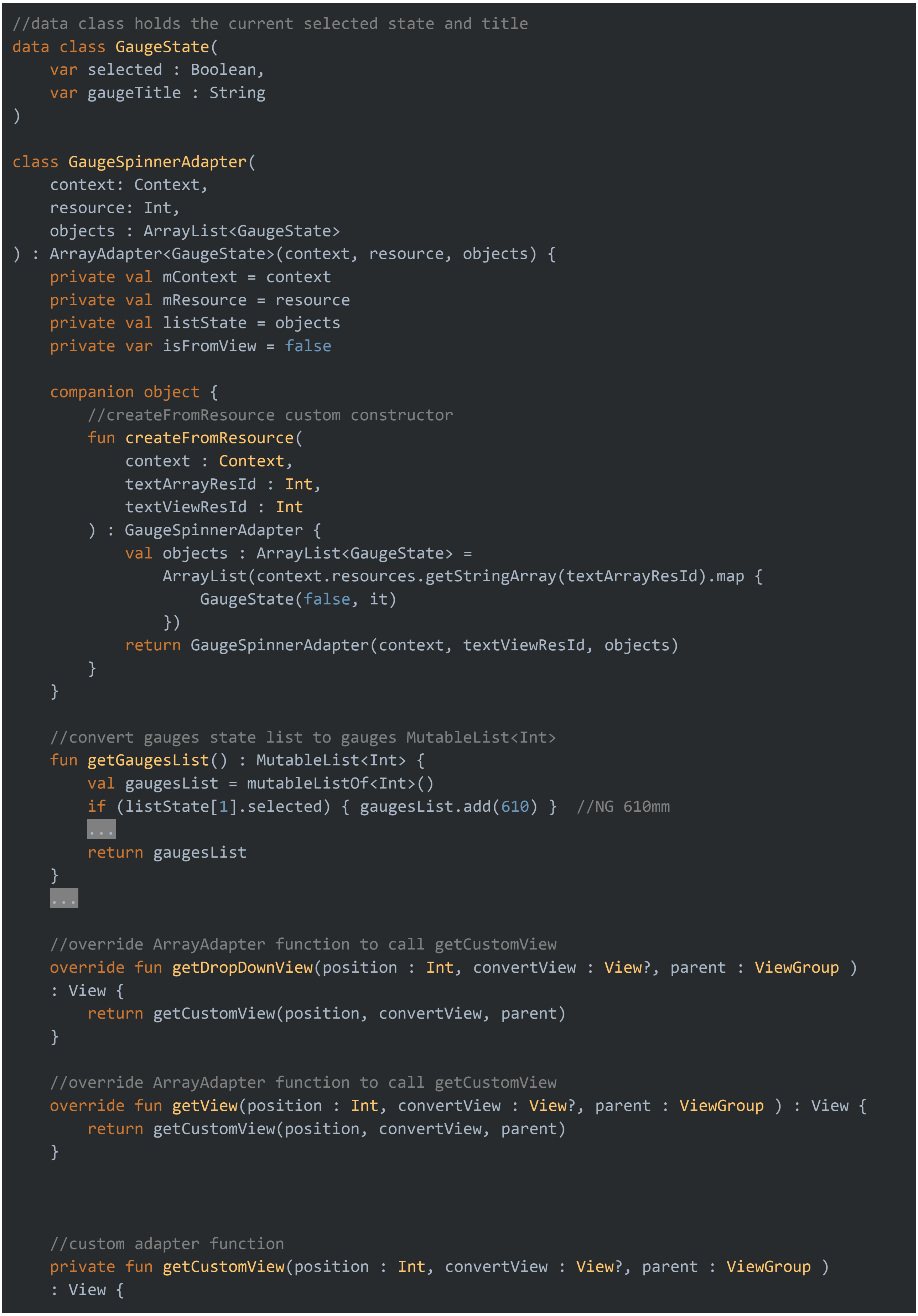
From DetailActivity.kt



```kotlin
class DetailActivity : AppCompatActivity(), OnMapReadyCallback {
    private lateinit var turntableMap : GoogleMap
    private var turntableMarker : Marker? = null
    private var mapReady = false
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_detail)
        ...
        //get SupportMapFragment and set callback
        Log.i("GOOGLE_MAP", "Map initialising")
        val mapFragment = supportFragmentManager
            .findFragmentById(R.id.detailMap) as SupportMapFragment
        mapFragment.getMapAsync(this)
    }
    ...
    //when map is ready populate with turntable location pin
    override fun onMapReady(googleMap: GoogleMap) {
        mapReady = true
        Log.i("GOOGLE_MAP", "Map ready - $googleMap")
        turntableMap = googleMap
        turntableMap.uiSettings.setAllGesturesEnabled(true) //enable map gestures

        //set map style from JSON string resource based on theme
        val mapStyleResource =
            MapStyleOptions(resources.getString(R.string.googleMapsStyleJson))
        val mapStyleSuccess = turntableMap.setMapStyle(mapStyleResource)
        if (!mapStyleSuccess) { Log.i("GOOGLE_MAP", "Map style parsing failed") }

        mapMarkerAndCamera() //create or move map pin and move camera to pin
    }

    //create or move map pin and move camera
    private fun mapMarkerAndCamera() {
        if (mapReady) {
            //create LatLng object with turntable latitude and longitude
            val turntableLatLng = LatLng(turntable.latitude, turntable.longitude)
            if (turntableMarker != null) { //marker already exists, move marker
                turntableMarker!!.position = turntableLatLng
                //animate camera to pin location
                turntableMap.animateCamera(
                    CameraUpdateFactory.newLatLngZoom(turntableLatLng, 18f),
                    2000,
                    null
                )
            }
            else { //marker does not exist, create marker
                turntableMarker = turntableMap.addMarker(
                    MarkerOptions().position(turntableLatLng).title(turntable.name)
                )
                ...
            }
        }
    }
}
```

Appx. 4.3 - Custom spinner adapter code

From GaugeSpinnerAdapter.kt



```
//data class holds the current selected state and title
data class GaugeState(
    var selected : Boolean,
    var gaugeTitle : String
)

class GaugeSpinnerAdapter(
    context: Context,
    resource: Int,
    objects : ArrayList<GaugeState>
) : ArrayAdapter<GaugeState>(context, resource, objects) {
    private val mContext = context
    private val mResource = resource
    private val listState = objects
    private var isFromView = false

    companion object {
        //createFromResource custom constructor
        fun createFromResource(
            context : Context,
            textArrayResId : Int,
            textViewResId : Int
        ) : GaugeSpinnerAdapter {
            val objects : ArrayList<GaugeState> =
                ArrayList(context.resources.getStringArray(textArrayResId).map {
                    GaugeState(false, it)
                })
            return GaugeSpinnerAdapter(context, textViewResId, objects)
        }
    }

    //convert gauges state list to gauges MutableList<Int>
    fun getGaugesList() : MutableList<Int> {
        val gaugesList = mutableListOf<Int>()
        if (listState[1].selected) { gaugesList.add(610) }  //NG 610mm
        ...
        return gaugesList
    }
    ...

    //override ArrayAdapter function to call getCustomView
    override fun getDropDownView(position : Int, convertView : View?, parent : ViewGroup )
    : View {
        return getCustomView(position, convertView, parent)
    }

    //override ArrayAdapter function to call getCustomView
    override fun getView(position : Int, convertView : View?, parent : ViewGroup ) : View {
        return getCustomView(position, convertView, parent)
    }



    //custom adapter function
    private fun getCustomView(position : Int, convertView : View?, parent : ViewGroup )
    : View {
        var itemView = convertView
        val viewHolder : SpinnerViewHolder?

        //create view holder if doesn't exist, else retrieve current view holder
        if (itemView == null) {
            //create view by inflating resource
            val layoutInflater = LayoutInflater.from(mContext)
            itemView = layoutInflater.inflate(mResource, parent, false) as View

            //create view holder and set as view tag
            viewHolder = SpinnerViewHolder()
            viewHolder.checkBox = itemView.findViewById<CheckBox>(R.id.spinnerCheckbox)
            viewHolder.textView = itemView.findViewById<TextView>(R.id.spinnerText)
            itemView.tag = viewHolder
        }
        else {
            //retrieve current view holder
            viewHolder = itemView.tag as SpinnerViewHolder
        }

        //set spinner item text
        viewHolder.textView!!.text = listState[position].gaugeTitle
        //set checkbox to checked state
        isFromView = true
        viewHolder.checkBox!!.isChecked = listState[position].selected
        isFromView = false

        if (position==0) {
            //first row checkbox should be invisible
            viewHolder.checkBox!!.visibility = View.INVISIBLE

            //first row should show selected gauges
            viewHolder.textView!!.text = gaugeString()
        }
        else { viewHolder.checkBox!!.visibility = View.VISIBLE }

        //set checked change listener to update listState
        viewHolder.checkBox!!.setOnCheckedChangeListener{ _, isChecked ->
            if (!isFromView) {
                listState[position].selected = isChecked //update listState bool from checkbox
                notifyDataSetChanged() //update list to show selected gauges
            }
        }
        return itemView //return spinner item view
    }

    //view holder class
    private class SpinnerViewHolder {
        var checkBox : CheckBox? = null
        var textView : TextView? = null
    }
}
```