Assignment 2 Project Report: "Rent With Intent" - Musical Instruments Rental App

Student Name: Dang Khoa Le
Student ID: 103844421
Tutorial: Wednesday 4.30-6.30 PM

A. Overview

This report reflects the development of a simple Android app named **"Rent With Intent"** for a music studio that rents musical instruments and equipment. The app allows clients to view available instruments, filter items based on attributes like category and brand, and "borrow" items for rental. The app consists of two activities and uses **Intents** to share data between them.

B. Key Features

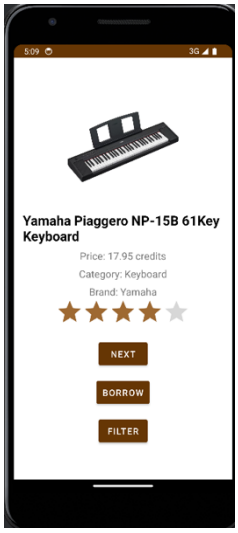
- **Two-Activity Structure**: The app includes a Main Activity where users can view and browse musical instruments, and a Booking Activity where they can confirm their rental bookings and set a borrowing period.
- **Data Sharing via Parcelable**: The instrument data is passed between activities using the Parcelable interface, which is more efficient than serializing objects in Android.
- **Multi-Choice Filtering**: A filtering feature allows users to filter instruments based on categories and brands using chip widgets.
- **DatePicker for Rental Period**: Users can set the end date for their rental using a DatePicker dialog, validating the dates and ensuring they are appropriate for booking.
- **Custom Styling**: The app uses custom themes and styles to ensure a consistent look and feel. This include customized styles/themes on the header and sub-header texts, buttons and nav-bar.
- **Error Handling and User Feedback**: When no instruments match the filter, a booking is made (or canceled), or invalid dates are selected, the app provides appropriate feedback using Toast messages to the user, as well as effective logs for error handling and debugging.

C. Planning

1. User Stories

- **Story 1**:
  Actor: User (parent looking for a musical instrument for their daughter)
  Goal: The user wants to browse and rent a musical instrument, particularly a piano or keyboard, that is affordable for their daughter's learning hobby.
  Preconditions: The app is installed and opened, and the user has access to browse the available instruments.
  Main Flow:
  - The user opens the app and is presented with a list of musical instruments.
  - The app displays each instrument's details, including the name, price, rating, category, brand, and a corresponding image.
  - The user scrolls through the instruments and finds a Yamaha keyboard that matches the desired price and quality.
  - The user decides to rent the keyboard for their daughter.

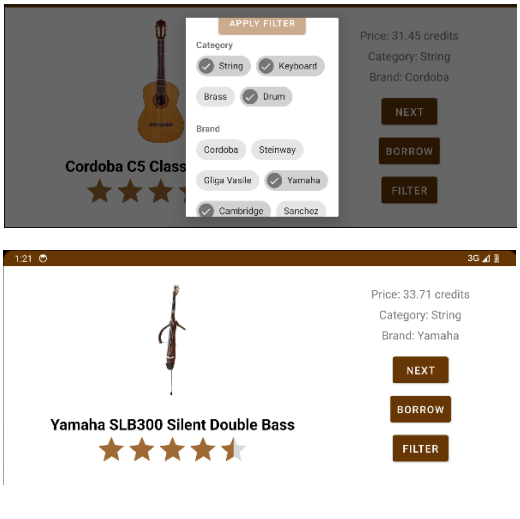
  Postconditions: The user is satisfied with the detailed display of the instrument's information and proceeds to rent it.

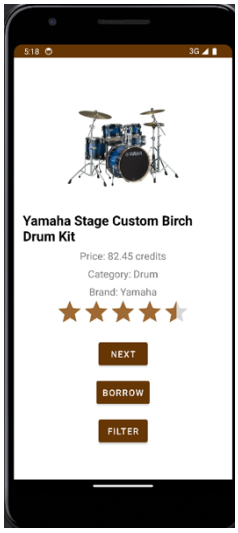


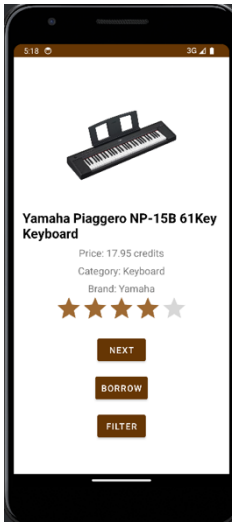
UI/UX Justification: The app effectively displays all necessary instrument details, allowing the user to make an informed decision based on the provided information. This design meets the user's needs for an affordable instrument for their daughter through displaying multiple available instruments with details listed rigorously for comparision and decision making.

- **Story 2**:
  Actor: User (musician preparing for a band's music concert)
  Goal: The user needs to filter and find specific musical instruments quickly for an upcoming concert.
  Preconditions: The app is installed and opened, and the user needs multiple instruments for a band performance.
  Main Flow:
  - The user opens the app and accesses the filter functionality using the chip widget dialog.
  - The user applies filters by selecting specific categories (e.g., keyboard, string, drums) and brands (e.g., Yamaha, Cambridge).
  - The app filters and displays instruments that match the selected criteria.
  - The user uses the "Next" button to cycle through the filtered instruments.
  - The user selects and prepares to rent the instruments needed for the concert.

  Postconditions: The user successfully finds and rents the required instruments efficiently using the filtering options.







These examples demonstrates the app's UI when user browsing the app in **Story 2**.

UI/UX Justification: The app effectively encompass the multi-choices filter function with the chip widget dialog allowing the user to quickly filter and navigate to the instrument that they need, shortcutting the browsing time. The app utilises the "Next" button allowing the user to go through available instruments from their filter choices.

2. Time Logs

- Day 1-4: Overview research and initial planning on deployments and set ups.
- Day 3-6: Design key UI elements in Android Studio layout editor, including the first 2 activities of Main Activity and Booking Activity.
- Day 5-8: Initialize backend components and functions for Main Activity and Booking Activity with Intent.
- Day 8-10: Initialize the Chip Dialog Filter function's backend, handling filtration functionalities, attempted testings on edge cases to avoid errors, using logs, and design their UI layout.
- Day 9: Implementing more logs to debug out error encountered.
- Day 10: Designing use cases applying on this app.
- Day 10-13: Design landscape layout, realizing the linear layout mode have multiple elements hidden or inappropriate visualized on landscape orientation. Main and Booking Activities' layout has been constructed during this time phase.
- Day 11: Initializing the DatePicker functionalities allowing users to set the period they would like to rent the instrument.
- Day 13: Designing the DatePicker on landscape mode (which lately removed).
- Day 14: Update, redesigning the Chip widget to have a scroller, allowing edge cases when the number of instrument brand and category exceed the dialog capacity, as well as ensuring no element will be covered when using landscape mode (only use 1 layout).
- Day 15: Add, deploy onSaveInstanceState method to save the current instrument index and filtered instrument list when changing the orientation in Main Activity.
- Day 16: Update, re-structure the DatePicker dialog, the start date is no longer required since the task specifies "Assume that the app allows for immediate pickup only; there is no concept of "future" bookings.". Add, use ActivityResultLauncher when launching the Booking activity, to allow data being transferred back from Booking Activity to Main Activity. Update, allow rating to be changed at Booking Activity, which transfers intent object to Main Activity to adapt new ratings and booking status. Update, update getParcelableArrayList method to the newest version. Update, a companion object is

created, the newInstance method is used to create instances of the ChipFilterDialog, ensuring correct setup and passing of required data through arguments.

D. Technical Usage

1. App Architecture

The app consists of two key activities:

- **Main Activity (MainActivity.kt)**: Displays musical instruments one by one, allows filtering, and has a "Next" button to cycle through available items. It passes the selected instrument to the Booking Activity via an intent[1]. It uses ActivityResultLauncher[12] to handle the result from the Booking Activity, feedback user based on the booking status and allow rating updates.
- **Booking Activity (Booking.kt)**: Allows users to confirm or cancel the booking for the selected instrument, displaying relevant details such as the name, price, and an image. It includes a DatePicker dialog to set the borrowing period with validation on date picking applied. Instrument's rating and booking status can be update and saved from Booking Activity and set as an intent object[1] transferred to Main Activity.

2. Model Design (Instrument.kt)

The app uses a simple **Instrument** data model to represent each musical instrument. This model implements the Parcelable[9] interface to facilitate efficient data transfer between the two activities (Main Activity and Booking Activity).

3. Chip Widget Dialog (ChipFilterDialog.kt):

The app uses a chip[13] widget, showcasing multi-choice options of available categories and brands on a modal dialog[2], allowing users to filter instruments based on their interest. The dialog has been include a scroll view for improved usability, especially in landscape mode and edge cases when the number of brand and category exceeding the capacity of the dialog.

4. UI Components and Layouts

The app uses ImageView, TextView, RatingBar, ChipGroup, and DatePicker widgets to display the instrument details, allow for filtering, and set the rental period. The layout files define these components and apply custom styles.

Main Activity Layout (layouts/activity_main.xml and layouts-land/ activity_ main.xml):

- Displays the instrument image, name, price, category, and rating.
- Includes a "Next" button to browse instruments and a "Borrow" button to confirm the instrument that is wished to be proceeded to Booking activity page.
- Includes a "Filter" button to open the filter dialog with chip widgets to enable brands and categories filtering.
- Includes a RatingBar[4] showing the rating for each instrument. Note that rating in Main Activity is not changeable.
- Main Activity includes the UI layouts for both portrait (LinearLayout) and landscape modes (ConstraintLayout).
- Landscape mode layouts contain 2 LinearLayout splitting the screen to 2 side horizontally, restructure the app components to avoid app components being hidden from the portrait mode.

```xml
<!-- Block Left -->
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:gravity="center"
    android:layout_weight="1">
        <!-- ImageView for the instrument image -->
        <ImageView
            android:id="@+id/instrumentImage"
            android:layout_width="200dp"
            android:layout_height="200dp"
            android:contentDescription="Instrument Image" />
        <!-- TextView for the instrument name as header -->
        <TextView
            android:id="@+id/instrumentName"
            style="@style/HeadingStyle"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Instrument Name"
            android:paddingTop="16dp"/>
        <!-- RatingBar with adjust on star feedback, step-sizing 0.5 -->
        <RatingBar
            android:id="@+id/ratingBar"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:numStars="5"
            android:stepSize="0.5"
            android:rating="4.5" />
```

```xml
<!-- Block Right -->
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:gravity="center"
    android:layout_weight="1"
    android:paddingStart="4dp">
    <!-- TextView for the instrument's details -->
    <TextView
        android:id="@+id/instrumentPrice"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Price"
        android:textSize="18sp"
        android:paddingTop="8dp"
        app:layout_constraintTop_toBottomOf="@id/instrumentName"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />
    <TextView
        android:id="@+id/instrumentCategory"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Category"
        android:textSize="18sp"
        android:paddingTop="8dp"
        app:layout_constraintTop_toBottomOf="@id/instrumentPrice"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />
```

These images show how landscape mode are settled in ConstraintLayout, with 2 blocks of LinearLayout with the app's components allocated side-by-side in these 2 blocks.

Booking Activity Layout (layouts/activity_booking.xml and layouts-land/ activity_booking.xml):

- Displays the instrument image, rental details, a "Set Borrow Period" button for selecting rental dates, and "Confirm" and "Cancel" buttons for the booking.
- Displays a RatingBar[4] that can be modified and saved upon rental confirmation.
- Booking Activity includes the UI layouts for both portrait (LinearLayout) and landscape modes (ConstraintLayout).
- Landscape mode layouts contain 2 LinearLayout splitting the screen to 2 side horizontally, restructure the app components to avoid app components being hidden from the portrait mode.

Filter Dialog Layout (chip_dialog.xml):

- Provides a filter dialog[2] using ChipGroup[13] to filter instruments by category and brand. The chip groups are wrapped inside a ScrollView[10], enabling scrollable feature to handle large numbers of options.
- Displays the "Apply Filter" button to save and apply features (category and brand) to be filtered.

```xml
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Category"
    android:textStyle="bold" />

<!-- Chips for categories -->
<com.google.android.material.chip.ChipGroup
    android:id="@+id/categoryChipGroup"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
</com.google.android.material.chip.ChipGroup>

<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Brand"
    android:textStyle="bold"
    android:layout_marginTop="16dp" />

<!-- Chips for brands -->
<com.google.android.material.chip.ChipGroup
    android:id="@+id/brandChipGroup"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
</com.google.android.material.chip.ChipGroup>
```

The image presents the 'chip_dialog.xml' layout with the Category and Brand ChipGroup.

DatePicker Dialog Layout (datepicker_dialog.xml):

- Allows users to select the end date of their rental. The dialog[2] is designed to occupy the majority of the screen space[8] to enhance usability.
- Wrap the Sub-heading text and the DatePicker[3] calendar inside a ScrollView[10] enabling dynamic scrollable feature.

```
<!-- ScrollView wraps DatePicker to handle its size dynamically -->
<ScrollView
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1">
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical">
        <!-- End Date Picker -->
        <TextView
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:paddingBottom="5dp"
            android:text="Select End Date for Your Rental Period"
            style="@style/SubHeadingStyle"
            android:textStyle="italic" />
        <DatePicker
            android:id="@+id/endDatePicker"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:maxHeight="90dp" />
    </LinearLayout>
</ScrollView>
```

- Display the “Confirm Timeframe” button to save the end date that user wishes to rent the instrument.

5. Custom Styles

The app includes custom styles such as HeadingStyle, SubHeadingStyle, and ButtonStyle, which are applied to multiple UI components to maintain design consistency. These styles define font sizes, colors, and padding, contributing to a cohesive visual experience.

```
<style name="Theme.RentWithIntent" parent="Theme.MaterialComponents.Light.NoActionBar">
    <item name="colorPrimary">@color/lightbrown</item>
    <item name="colorPrimaryDark">@color/darkbrown</item>
    <item name="colorAccent">@color/mediumbrown</item>
</style>

<!-- Style for heading text -->
<style name="HeadingStyle">
    <item name="android:textSize">24sp</item>
    <item name="android:textColor">@color/black</item>
    <item name="android:textStyle">bold</item>
</style>

<!-- Style for heading text -->
<style name="SubHeadingStyle">
    <item name="android:textSize">14sp</item>
    <item name="android:textColor">@color/darkbrown</item>
    <item name="android:textStyle">italic</item>
</style>

<!-- Style for buttons -->
<style name="ButtonStyle">
    <item name="android:backgroundTint">@color/darkbrown</item>
    <item name="android:textColor">@color/white</item>
    <item name="android:padding">10dp</item>
    <item name="android:textSize">16sp</item>
</style>
```

This shows the custom styles defined in res/values/themes.xml.

The color of lightbrown, darkbrown and mediumbrown are set in res/values/colors.xml with intention to define the color-set matching the app components to the overall theme color of the app, which is brown.

6. Use of RatingBar and ChipGroup

The RatingBar[4] widget is used to display the instrument rating, and the ChipGroup[13] is used in the filter dialog to provide multi-choice filters for categories and brands.

The ChipGroup[13] is dynamically populated with unique categories and brands from the list of instruments.

```
<!-- RatingBar with adjust on star feedback, step-sizing 0.25 -->
<RatingBar
    android:id="@+id/ratingBar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:numStars="5"
    android:stepSize="0.25"
    android:rating="4.5" />
```

```
<!-- Chips for categories -->
<com.google.android.material.chip.ChipGroup
    android:id="@+id/categoryChipGroup"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
</com.google.android.material.chip.ChipGroup>
```

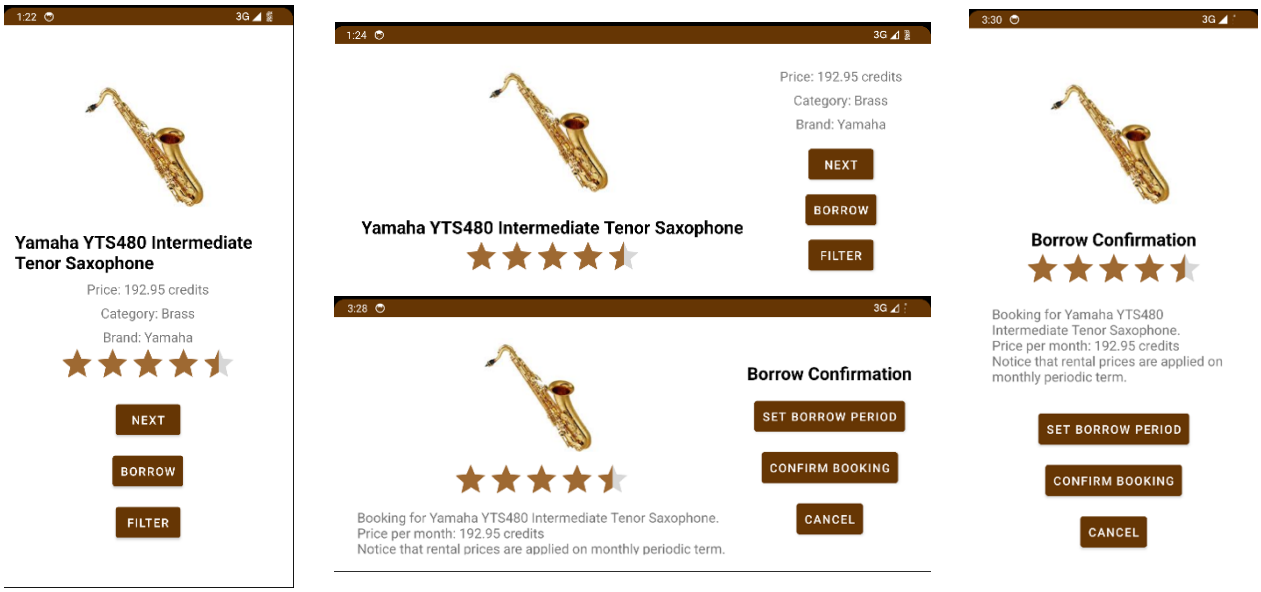
```
<!-- Chips for brands -->
<com.google.android.material.chip.ChipGroup
    android:id="@+id/brandChipGroup"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal">
</com.google.android.material.chip.ChipGroup>
```

7. Use of DatePicker and Validation Logic

The DatePicker dialog allows users to set a borrowing period with the end dates of their rental term. The app backend also validates the dates to ensure the selected timeframe is appropriate (e.g. end date must be after), prompting users to correct any invalid input.

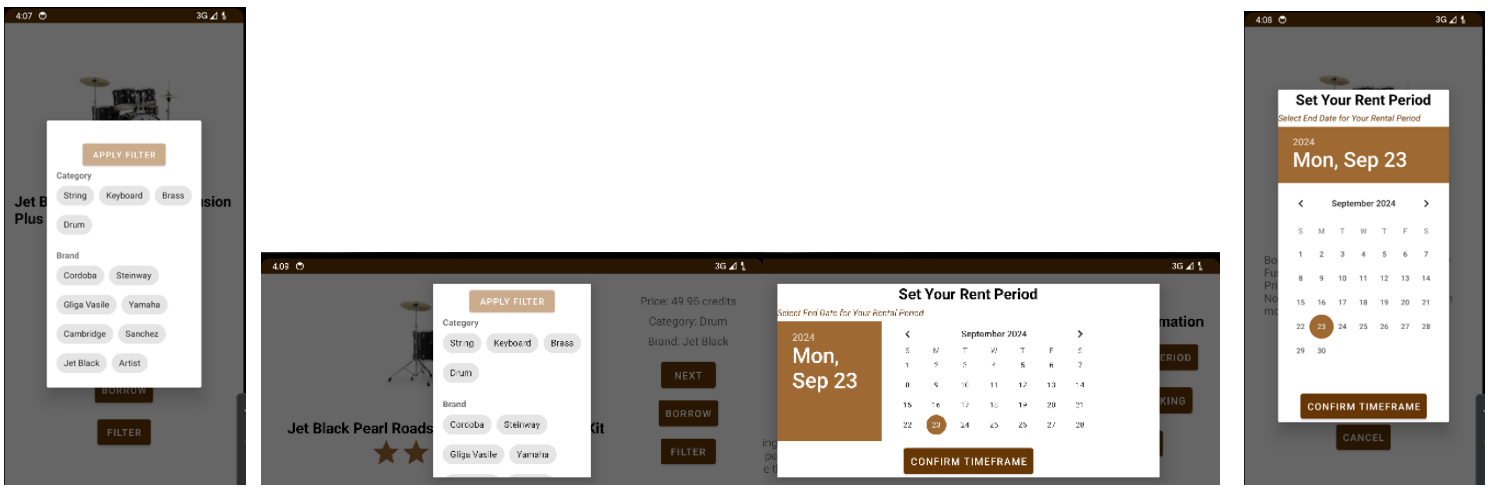
8. Landscape Mode Layouts

E. Landscape layouts were created for the Main Activity and Booking Activity. These layouts optimize the display and functionality for landscape orientation, ensuring a user-friendly



experience across different device orientations.

These images above show the difference in orientation between the landscape and portrait mode view elements in the 2 main activities, while these below shows UI orientations for chip widget and DatePicker dialogs.



9. Image Assets

The app's instrument images were downloaded and can be accessed at res/drawable.

The app's logo has been changed at res/drawable/logo.png. The logo matching the app's themes when illustrating multiple musical instruments with brown and white colorways. This logo image is powered by genAI tool DALL-E[22].

Figure. Rent With Intent app logo.

F. Technical Analysis

This section provides an analysis of the backend code for the **"Rent With Intent"** project, examining the implementation of core functionalities and the deployment of various methods and components to achieve the desired functionality.

1. MainActivity.kt

a) Instrument List and Filtering

The **instrument list** in Main Activity is a hardcoded collection of Instrument objects. Each instrument has attributes such as name, rating, price, category, and brand. This data is stored in memory and is not persisted, satisfying the requirement to maintain temporary in-memory data.

```
// Listing instruments with name (string), rating (float), price (float), category (string), and brand (string)
private val instruments = listOf(
    Instrument( name: "Cordoba C5 Classical Guitar", rating: 4.0f, price: 31.45f, category: "String", brand: "Cordoba"),
    Instrument( name: "Steinway Model B Grand Piano - Satin Ebony", rating: 4.9f, price: 3429.75f, category: "Keyboard", brand: "Steinway"),
    Instrument( name: "Gliga Vasile Violin", rating: 4.7f, price: 78.75f, category: "String", brand: "Gliga Vasile"),
    Instrument( name: "Yamaha Piaggero NP-15B 61Key Keyboard", rating: 4.1f, price: 17.95f, category: "Keyboard", brand: "Yamaha"),
    Instrument( name: "Cambridge TR620L Trumpet", rating: 3.9f, price: 49.75f, category: "Brass", brand: "Cambridge"),
    Instrument( name: "Yamaha YTS480 Intermediate Tenor Saxophone", rating: 4.4f, price: 192.95f, category: "Brass", brand: "Yamaha"),
    Instrument( name: "Sanchez Soprano Ukulele - Natural Satin", rating: 3.7f, price: 3.50f, category: "String", brand: "Sanchez"),
    Instrument( name: "Jet Black Pearl Roadshow Fusion Plus Drum Kit", rating: 3.5f, price: 49.95f, category: "Drum", brand: "Jet Black"),
    Instrument( name: "Artist ST62 Fiesta Red Electric Guitar", rating: 4.3f, price: 16.45f, category: "String", brand: "Artist"),
    Instrument( name: "Cambridge Double Key Tenor Flute", rating: 4.2f, price: 24.80f, category: "Brass", brand: "Cambridge"),
    Instrument( name: "Yamaha SLB300 Silent Double Bass", rating: 4.6f, price: 33.71f, category: "String", brand: "Yamaha"),
    Instrument( name: "Yamaha Stage Custom Birch Drum Kit", rating: 4.4f, price: 82.45f, category: "Drum", brand: "Yamaha")
)
```

Filtered Instrument List: The app applies a filtering mechanism through a modal dialog that allows users to filter instruments by categories and brands. This list is updated in real time based on user interaction with the filtering chips[13].

Indexing and Cycling Through Instruments: The app maintains an index (currentInstrumentIndex) to track which instrument is currently being displayed. This index is updated by the "Next" button, which cycles through the list of instruments using the modulo operation:

```
currentInstrumentIndex = (currentInstrumentIndex + 1) % filteredInstruments.size
```

This ensures that the instrument display loops back to the first instrument after the last one has been shown.

Display Instruments: The function displayInstrument plays a crucial role in updating the UI with the details of the currently selected instrument. It is responsible for displaying the relevant information of an instrument from the filtered list (filteredInstruments) based on the current index (currentInstrumentIndex). This is how it operates:

- Retrieving the instrument to display with the current index.
- Debugging with Logs
- Updating TextViews: Each TextView component (nameTextView, priceTextView, brandTextView, categoryTextView, and brandTextView) is updated with the relevant attributes of the selected instrument. By updating these TextViews dynamically, the UI

reflects the instrument's name, price, brand, and category seamlessly based on the current instrument in focus.

- Updating the RatingBar: The RatingBar[4] is updated with the instrument's rating, which is a floating-point value between 0 and 5.
- Setting the Instrument Image: mapping the ImageView (each instruments has a unique image corresponding to it), to the name of the instrument, using the when expression.

```
// Instrument to be displayed are the one filtered
val instrument : Instrument = filteredInstruments[currentInstrumentIndex]
Log.d( tag: "MainActivity", msg: "Displaying instrument: $instrument")

// Set instrument details in the corresponding views
nameTextView.text = instrument.name
priceTextView.text = "Price: ${instrument.price} credits"
categoryTextView.text = "Category: ${instrument.category}"
brandTextView.text = "Brand: ${instrument.brand}"
ratingBar.rating = instrument.rating

// Set a corresponding image for each instrument
when (instrument.name) {
    "Cordoba C5 Classical Guitar" -> imageView.setImageResource(R.drawable.guitar)
    "Steinway Model B Grand Piano - Satin Ebony" -> imageView.setImageResource(R.drawable.piano)
    "Gliga Vasile Violin" -> imageView.setImageResource(R.drawable.violin)
    "Yamaha Piaggero NP-15B 61Key Keyboard" -> imageView.setImageResource(R.drawable.keyboard)
    "Cambridge TR620L Trumpet" -> imageView.setImageResource(R.drawable.trumpet)
    "Yamaha YTS480 Intermediate Tenor Saxophone" -> imageView.setImageResource(R.drawable.saxophone)
    "Sanchez Soprano Ukulele - Natural Satin" -> imageView.setImageResource(R.drawable.ukulele)
    "Jet Black Pearl Roadshow Fusion Plus Drum Kit" -> imageView.setImageResource(R.drawable.drum)
    "Artist ST62 Fiesta Red Electric Guitar" -> imageView.setImageResource(R.drawable.eguitar)
    "Cambridge Double Key Tenor Flute" -> imageView.setImageResource(R.drawable.flute)
    "Yamaha SLB300 Silent Double Bass" -> imageView.setImageResource(R.drawable.doublebass)
    "Yamaha Stage Custom Birch Drum Kit" -> imageView.setImageResource(R.drawable.drumkit)
}
```

Handling No Result from Chip Filter: When no instruments match the user's filter from the chip modal window, function handleNoResults is called, which update the nameTextView with text "No matching result, please try again.", with error Toast message displayed to the user and error logs saved to LogCat.

```
Toast.makeText( context: this, text: "Filter returns no matching result!", Toast.LENGTH_SHORT).show()
Log.d( tag: "MainActivity", msg: "Filter returns no matching result!")
nameTextView.text = "No matching result, please try again."
```

Saving Instance State upon Device Orientation: onSaveInstanceState method used to save and restore the state of currentInstrumentIndex and filteredInstruments. This allows the app to retain the state when the device orientation changes. The filteredInstruments are saved as an ArrayList[14] of Parcelable objects[9], which is necessary since Parcelable is the efficient way Android passes complex data types between activities and states.

```
// Utilize savedInstanceState to remain the currentInstrumentIndex and filteredInstruments upon changing orientation
// Restore saved state if exists
if (savedInstanceState != null) {
    currentInstrumentIndex = savedInstanceState.getInt( key: "currentInstrumentIndex",  defaultValue: 0)
    filteredInstruments = savedInstanceState.getParcelableArrayList( key: "filteredInstruments", Instrument::class.java) ?: instruments
}
```

```
// Using onSaveInstanceState to save data upon changing orientation
Khoa Le
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putInt("currentInstrumentIndex", currentInstrumentIndex)
    outState.putParcelableArrayList("filteredInstruments", ArrayList<Parcelable>(filteredInstruments))
}
```

Initially, upon creating the Main Activity, it will check if any saved instance state exist, if it exist, it will use the current instrument index integer and filtered instrument array listing saved upon the last instance state saving.

b) Intents and Navigation

The Intent mechanism[1] is utilized for navigation between the Main Activity and Booking Activity. When the "Borrow" button is pressed, the selected instrument (including any applied

filters) is passed to the Booking Activity via an Intent, using the Parcelable interface[9] to efficiently serialize complex data between activities.

```
// "Borrow" button listener to navigate to the booking page/screen with the current instrument
borrowButton.setOnClickListener { it: View!
    val instrument : Instrument = filteredInstruments[currentInstrumentIndex]
    Log.d( tag: "MainActivity", msg: "Borrow button clicked for instrument: $instrument")
    // Pass the instrument details to the Booking activity using Intent
    val intent : Intent = Intent( packageContext: this, Booking::class.java).apply { this: Intent
        putExtra( name: "instrument", filteredInstruments[currentInstrumentIndex])
    }
    bookingActivityLauncher.launch(intent) // Launch Booking activity
}
```

The use of Parcelable[9] to pass the instrument data is a more efficient way to serialize complex data between activities then using Serializable.

```
// Define the ActivityResultLauncher
private lateinit var bookingActivityLauncher: ActivityResultLauncher<Intent>
```

The ActivityResultLauncher[12] is used to handle the feedback from the Booking Activity. Upon confirming a booking, the Main Activity receives the updated rating of the instrument from the Booking Activity. If the user changes the rating during booking, the updated rating is reflected in the Main Activity. The display is updated accordingly, providing immediate feedback to the user about the successful booking and any changes made to the instrument's rating. This dynamic interaction between activities ensures seamless data transfer and consistent user experience, enhancing the app's responsiveness and usability.

```
// Initialize the ActivityResultLauncher
bookingActivityLauncher = registerForActivityResult(ActivityResultContracts.StartActivityForResult()) { result ->
    if (result.resultCode == RESULT_OK) { // Booking activity passes the result confirming booking is successful.
        Toast.makeText( context: this, text: "Rental booking confirmed!", Toast.LENGTH_SHORT).show()
        Log.d( tag: "MainActivity", msg: "Rental booking confirmed!")
        val updatedRating : Float? = result.data?.getFloatExtra( name: "updatedRating", ratingBar.rating)
        // If user changed the rating at Booking page, alert changes
        if (updatedRating != filteredInstruments[currentInstrumentIndex].rating) {
            filteredInstruments[currentInstrumentIndex].rating =
                updatedRating ?: ratingBar.rating
            displayInstrument(nameTextView, priceTextView, categoryTextView, brandTextView, ratingBar, imageView) // Update display
            Toast.makeText( context: this, text: "Rating updated!", Toast.LENGTH_SHORT).show()
            Log.d( tag: "MainActivity", msg: "Rating updated to $updatedRating")
        }
    }
}
```

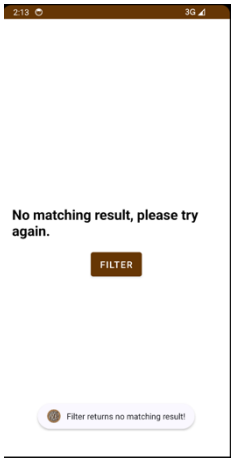
Upon successful booking, a Toast message and log confirm the booking, and if the rating has been modified, another message alerts the user of the rating update. This approach ensures that changes in the Booking Activity are promptly reflected back in the Main Activity, maintaining the integrity of the displayed information.

c) Dynamic UI Updates

The method displayInstrumentOrhandleNoResults dynamically updates the visibility of UI components based on whether there are filtered instruments available, as well as trigger the displayInstrument or handleNoResults function appropriately.

```
if (filteredInstruments.isNotEmpty()) {
    displayInstrument(nameTextView, priceTextView, categoryTextView, brandTextView, ratingBar, imageView)
    priceTextView.visibility = View.VISIBLE
    categoryTextView.visibility = View.VISIBLE
    brandTextView.visibility = View.VISIBLE
    ratingBar.visibility = View.VISIBLE
    imageView.visibility = View.VISIBLE
    nextButton.visibility = View.VISIBLE
    borrowButton.visibility = View.VISIBLE
} else { // Case no result, disable visibility
    handleNoResults(nameTextView)
    priceTextView.visibility = View.GONE
    categoryTextView.visibility = View.GONE
    brandTextView.visibility = View.GONE
    ratingBar.visibility = View.GONE
    imageView.visibility = View.GONE
    nextButton.visibility = View.GONE
    borrowButton.visibility = View.GONE
}
```

This enhances the user experience by hiding irrelevant UI elements and indicates errors when there are no matching results after applying filters.



This image shows the app UI handling for no result.

2. Instrument.kt

The Instrument class represents each musical instrument as a **data class**. This is appropriate since data classes in Kotlin automatically provide functionalities such as equals(), hashCode(), and toString() methods, simplifying the handling of such objects.

```
data class Instrument(
    val name: String,
    var rating: Float,
    val price: Float,
    val category: String,
    val brand: String
) : Parcelable {
```

a) Parcelable Implementation

The Parcelable[9] interface is implemented in the Instrument class to serialize the instrument object so that it can be passed between activities. The Parcel object is used to read and write the instrument's attributes efficiently.

```
constructor(parcel: Parcel) : this(
    name: parcel.readString() ?: "",    // Read name string, if null, returns empty string
    parcel.readFloat(),                 // Read rating float
    parcel.readFloat(),                 // Read price float
    category: parcel.readString() ?: "", // Read category string, if null, returns empty string
    brand: parcel.readString() ?: ""    // Read brand string, if null, returns empty string
)
```

```
override fun writeToParcel(parcel: Parcel, flags: Int) {
    parcel.writeString(name)
    parcel.writeFloat(rating)
    parcel.writeFloat(price)
    parcel.writeString(category)
    parcel.writeString(brand)
}
```

This approach minimizes the overhead associated with data serialization in Android.

3. Booking.kt

The Booking Activity manages the display of instrument details and provides options for the user to set a rental period, confirm, or cancel the booking. Recent updates have introduced a DatePicker[3] dialog[2] for setting the rental end date, refining the booking process and enhancing user interaction.

a) Intent and Parcelable Data Retrieval

The Booking Activity retrieves the passed Instrument object using the updated getParcelableExtra method[6].

```
val instrument : Instrument? = intent.getParcelableExtra( name: "instrument", Instrument::class.java)
```

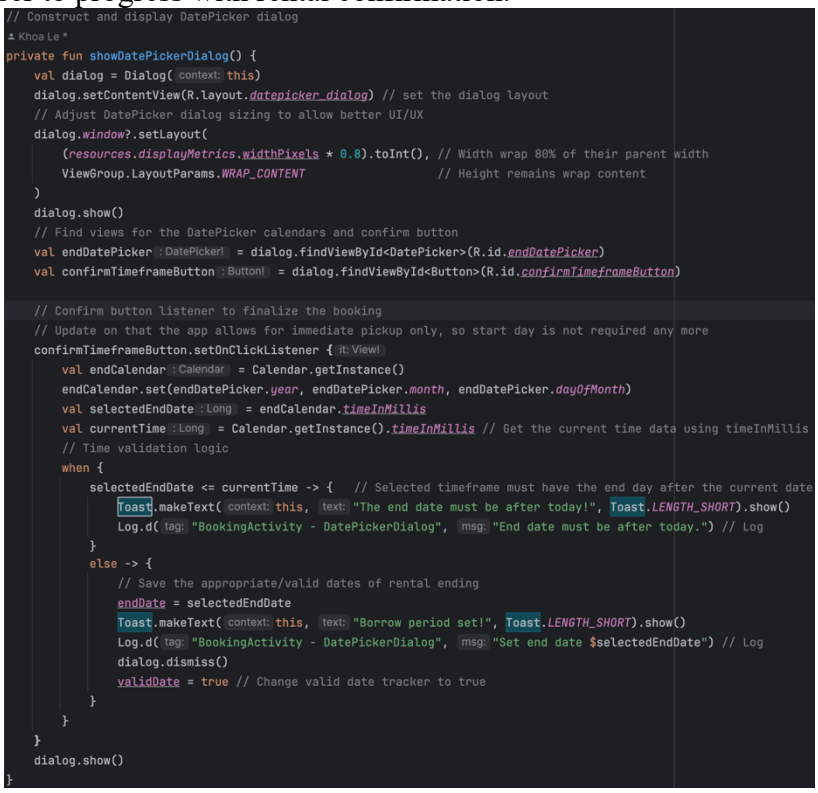
This ensures the correct details of the selected instrument are displayed to the user.

b) Rental Period Selection

- The updated Booking Activity introduces a **Set Borrow Period** button that triggers a DatePicker dialog. This allows users to select the end date of their rental period, meeting the app's requirement of setting a specific rental timeframe.
- The showDatePickerDialog method constructs and displays the DatePicker[3] dialog[2]. It dynamically adjusts the dialog's size to improve usability, ensuring that the dialog doesn't occupy the entire screen width (using displayMetrics to scale 80% width[11], balancing visibility and accessibility) but remains visible and accessible.

c) Timeframe Validation Logic

- The showDatePickerDialog method includes validation to ensure the selected end date is after the current date. If the user attempts to choose an invalid date (e.g., today or a past date), a Toast message alerts the user: “The end date must be after today!” This validation logic ensures that rental dates are realistic and prevent errors in the booking process.
- Upon timeframe is validated, the method will return validDate boolean value to true, allowing user to progress with rental confirmation.



```
// Construct and display DatePicker dialog
Khoa Le *
private fun showDatePickerDialog() {
    val dialog = Dialog( context: this)
    dialog.setContentView(R.layout.datepicker_dialog) // set the dialog layout
    // Adjust DatePicker dialog sizing to allow better UI/UX
    dialog.window?.setLayout(
        (resources.displayMetrics.widthPixels * 0.8).toInt(), // Width wrap 80% of their parent width
        ViewGroup.LayoutParams.WRAP_CONTENT                   // Height remains wrap content
    )
    dialog.show()
    // Find views for the DatePicker calendars and confirm button
    val endDatePicker : DatePicker! = dialog.findViewById<DatePicker>(R.id.endDatePicker)
    val confirmTimeframeButton : Button! = dialog.findViewById<Button>(R.id.confirmTimeframeButton)

    // Confirm button listener to finalize the booking
    // Update on that the app allows for immediate pickup only, so start day is not required any more
    confirmTimeframeButton.setOnClickListener { it: View!
        val endCalendar : Calendar = Calendar.getInstance()
        endCalendar.set(endDatePicker.year, endDatePicker.month, endDatePicker.dayOfMonth)
        val selectedEndDate : Long = endCalendar.timeInMillis
        val currentTime : Long = Calendar.getInstance().timeInMillis // Get the current time data using timeInMillis
        // Time validation logic
        when {
            selectedEndDate <= currentTime -> {   // Selected timeframe must have the end day after the current date
                Toast.makeText( context: this, text: "The end date must be after today!", Toast.LENGTH_SHORT).show()
                Log.d( tag: "BookingActivity - DatePickerDialog", msg: "End date must be after today.") // Log
            }
            else -> {
                // Save the appropriate/valid dates of rental ending
                endDate = selectedEndDate
                Toast.makeText( context: this, text: "Borrow period set!", Toast.LENGTH_SHORT).show()
                Log.d( tag: "BookingActivity - DatePickerDialog", msg: "Set end date $selectedEndDate") // Log
                dialog.dismiss()
                validDate = true // Change valid date tracker to true
            }
        }
    }
    dialog.show()
}
```

This image shows the showDatePickerDialog method, initialising the DatePicker dialog for timeframe selection and validating the timeframe selected.

d) UI Display

Define bookingDetailsTextView setting up the details of the instrument displayed at the Booking page, details settled from the instrument intent object get from Main Activity.

The setImage function dynamically updates the displayed image based on the instrument name, providing a tailored and engaging booking screen. This personalization adds to the app’s appeal by matching images to the corresponding instrument, making the booking experience more interactive and visually pleasing.

Displays ratingBar with default rating settled from the instrument intent object get from Main Activity.

e) Confirmation and Cancellation Logic

- Upon setting a valid rental period (validDate is true), users can finalize their booking using the "Confirm Booking" button. The app calculates the borrowing period in days by subtracting the current time from the selected end date.
- If the rental period is not set, users are prompted with a Toast message: "Please pick your borrowing period!" This feedback ensures users follow the correct steps before confirming their booking.
- When the booking is confirmed, a Toast provides feedback, stating the instrument's name and the duration of the rental period, enhancing the user's understanding of their booking details. Log at this stage also recorded.
- Similarly, when the "Cancel" button is pressed, Toast will message the user: "Booking Cancelled" and return to the Main Activity. Log at this stage also recorded.
- The Booking Activity sends back the result of the booking action (confirmed or cancelled) to the Main Activity using the ActivityResultLauncher[12], as well as any changes made with the RatingBar. This feedback loop ensures that users receive immediate updates about their booking actions, with appropriate Toast messages reflecting the outcome.

```
// Confirm button listener to finalize the booking
confirmButton.setOnClickListener { it: View!
    if (validDate == false) {
        Toast.makeText( context: this, text: "Please pick your borrowing period!", Toast.LENGTH_SHORT).show()
        Log.d( tag: "BookingActivity", msg: "Borrow period not defined.")
    }
    else {
        val currentTime : Long = Calendar.getInstance().timeInMillis // Get the current time data using timeInMillis
        val borrowPeriod : Int = TimeUnit.MILLISECONDS.toDays( duration: endDate!! - currentTime).toInt()
        Toast.makeText( context: this, text: "Booking Confirmed! You are borrowing ${instrument?.name} for $borrowPeriod days.", Toast.LENGTH_SHORT).show()
        Log.d( tag: "BookingActivity", msg: "Booking confirmed for ${instrument?.name} for $borrowPeriod days.")
        val intent : Intent = Intent().apply { this: Intent // Pass intent object back to Main activity page upon changes on booking status and rating
            putExtra( name: "isBooked", value: true)
            putExtra( name: "updatedRating", ratingBar.rating)
        }
        setResult(RESULT_OK, intent)
        finish() // Finish the activity and return to the previous (main) page/screen
    }
}
// Cancel button listener to cancel the booking
cancelButton.setOnClickListener { it: View!
    Toast.makeText( context: this, text: "Booking Cancelled", Toast.LENGTH_SHORT).show()
    Log.d( tag: "BookingActivity", msg: "Booking cancelled for ${instrument?.name}")
    setResult(RESULT_CANCELED) // Cancel booking
    finish() // Finish the activity and return to the previous (main) page/screen
}
```

4. ChipFilterDialog.kt

ChipFilterDialog is a fragment class[7] that manages the filtering functionality through a custom dialog. It uses ChipGroup components for multi-choice selection of categories and brands, making the filter dialog intuitive and flexible for users.

a) Dynamic Population of ChipGroup

The dialog dynamically populates the ChipGroup components[13] with chips representing unique categories and brands extracted from the list of instruments. Using the distinct function, only unique categories and brands are added, ensuring a clean and user-friendly filtering experience. This approach allows the filter dialog to adapt to any set of instruments dynamically, making it versatile and scalable.

```
// Dynamically create chips for each category and add them to the category chip group.
categories.forEach { category ->
    val chip : Chip = Chip(requireContext()).apply { this: Chip
        text = category // Set chip text to category name.
        isCheckable = true // Make the chip selectable.
    }
    categoryChipGroup.addView(chip) // Add the chip to the category chip group.
}

// Dynamically create chips for each brand and add them to the brand chip group.
brands.forEach { brand ->
    val chip : Chip = Chip(requireContext()).apply { this: Chip
        text = brand // Set chip text to brand name.
        isCheckable = true // Make the chip selectable.
    }
    brandChipGroup.addView(chip) // Add the chip to the brand chip group.
}
```

- **Category Chips**: Chips are created and added to the category chip group based on the distinct categories in the instrument list.
- **Brand Chips**: Similarly, brand chips are dynamically added to the brand chip group, providing users with selectable options to refine their search.

b) Filter Application Logic
When the user applies a filter by selecting chips, the filtering logic is executed by checking if the instrument matches the selected categories and brands. The selected chips' text values are retrieved and used to filter the instruments accordingly:

```
    // Retrieve the selected categories and brands from the checked chips.
    val selectedCategories : List<String> = categoryChipGroup.checkedChipIds.map { it: Int!
        dialog.findViewById<Chip>(it).text.toString()
    }
    val selectedBrands : List<String> = brandChipGroup.checkedChipIds.map { it: Int!
        dialog.findViewById<Chip>(it).text.toString()
    }

    Log.d( tag: "ChipFilterDialog",  msg: "Selected Categories: $selectedCategories, Selected Brands: $selectedBrands")

    // Filter the list of instruments based on the selected categories and brands
    val filtered : List<Instrument> = instruments.filter { it: Instrument
        (selectedCategories.isEmpty() || it.category in selectedCategories) &&
                (selectedBrands.isEmpty() || it.brand in selectedBrands)
    }

    Log.d( tag: "ChipFilterDialog",  msg: "Filtered Instruments: $filtered")

    // Apply the filtered list using the callback function and dismiss the dialog
    onFilterApplied(filtered)
    dismiss()
}

return dialog // Return the constructed dialog
```

- **Selected Categories and Brands**: The dialog captures the selected categories and brands using checkedChipIds[13] and converts the selected chip IDs to their corresponding text values.
- **Filtering Execution**: The instruments are filtered based on the selected values. If no chips are selected, the filter condition defaults to allowing all categories or brands, ensuring a comprehensive and accurate filtering process.
- **Callback Mechanism**: The filtered list is then passed back to the MainActivity using the callback function onFilterApplied, updating the UI with the filtered results.

c) Fragment Initialization: A companion object[15] is created. The newInstance method[15] is used to create instances of the ChipFilterDialog, ensuring correct setup and passing of required data through arguments. The instruments list and callback function are set via this method, promoting better encapsulation and reuse of the fragment.

```
companion object {
    // newInstance method creates a new instance of ChipFilterDialog and sets up the necessary data
    new *
    fun newInstance(
        instruments: List<Instrument>, // List of instruments to display in the filter dialog
        onFilterApplied: (List<Instrument>) -> Unit // Callback function to apply filter results
    ): ChipFilterDialog {
        val fragment = ChipFilterDialog() // Create a new instance of the dialog fragment
        val args : Bundle = Bundle().apply { this: Bundle
            // Save the instruments list in the arguments Bundle using Parcelable
            putParcelableArrayList("instruments", ArrayList(instruments))
        }
        fragment.arguments = args // Set the arguments to the fragment
        fragment.onFilterApplied = onFilterApplied // Set the callback function
        return fragment // Return the configured fragment instance
    }
}
```

d) Scrollable Chip Widgets
The ChipGroup components are placed inside a ScrollView[10], allowing better navigation when there are many options, particularly in landscape mode.

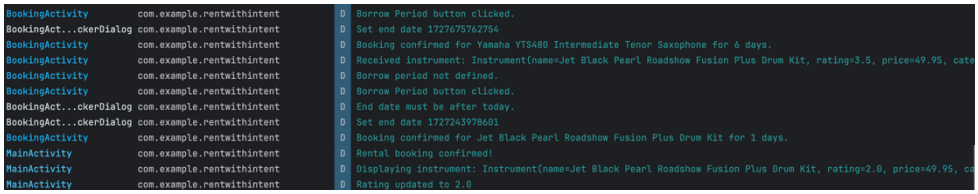
5. Logs and User Feedbacks

Log Statements

The app leverages **log statements** throughout the code for debugging and tracking user interactions, such as clicking buttons or displaying instruments.

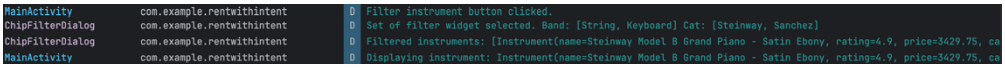
Logs also enable developers to verify and validate the content to be displayed, empower advantages on early debugging processes.

These logs are useful during the development phase to trace the flow of execution and identify any potential issues.



Example log statements with action of borrowing "Jet Black Pearl Roadshow Fusion Plus Drum Kit" instrument, when user confirming their borrowing timeframe and proceed to rental confirmation.

Example log statements of action browsing between different filtered instruments before selecting a particular instrument.



Example log statements when filtering instrument.

Toasts for User Feedback

The app uses **Toast messages** to provide feedback to the user, such as when instrument rental booking is confirm or cancelled, when user hasn't set their renting period, renting period (timeframe) is not valid, or when no instruments match the filter criteria.

The decision to utilise Toast over Snackbar for this project is due to the UI/UX visual impact, as Toast is visually more relatable to the overall theme of the app (brown and white), more impactful upon user recognition when using the app, and most importantly, doesn't hide any app content upon their alert.

G. Design Choices

- **Parcelable for Data Sharing**: Using Parcelable[9] instead of Serializable offers better performance for passing objects between activities, as it avoids the overhead of reflection used by Serializable.
- **Dynamic UI Updates**: Hiding and showing elements like buttons and text views based on conditions (e.g., no instruments match the filter) ensures a clean, user-friendly experience.
- **Use of Chips for Filtering**: The use of ChipGroup for filtering by categories and brands provides an intuitive and visually appealing way for users to interact with the app.
- **Use of DatePicker for Timeframe selection:** The need to have a time period – setting functionality is compulsory for the rental service provider to acquire the desired time user

rent their instrument. The use of DatePicker is visually approachable and easy-to-use for any generic customer group to start using it at the first stage.

- **Custom Layouts for Landscape:** Separate layouts for landscape mode ensure a consistent and effective UI/UX across device orientations.
- **Error Handling with State Restoration:** Leveraging onSaveInstanceState ensures that the app maintains a consistent state across orientation changes

H. Challenges and Improvements

I. Filtration Method Returns No Result

One challenge was ensuring the app could handle cases where no instruments matched the filter. This was resolved by dynamically adjusting the visibility of UI elements based on the filtered results (show View.GONE for buttons and elements not used when there are no result, create better visual impact for the user as well as avoid ambiguity when having redundant elements shown up).

- **Filtration Method Deprecation**

The other one was getParcelableExtra and getParcelableArrayList method get deprecated[5], this is as it uses a generic type (T) which could cause runtime error.

```
val instrument : Instrument? = intent.getParcelableExtra<Instrument>( name: "instrument")
```

The solution was to use a newer version[6,14] of the method with the class type specified. *TIRAMISU* annotation[6] (in Java) is also added upon setting this change.

```
val instrument : Instrument? = intent.getParcelableExtra( name: "instrument", Instrument::class.java)
```

- **DatePicker Usability**: The DatePicker dialog initially had usability issues due to its small size. The dialog was adjusted to cover the majority of the screen[8] to improve user interaction.
- **Handling Landscape Mode**: Additional landscape layouts were created for all main screens, ensuring a smooth user experience in different orientations. ChipGroup widget and DatePicker dialog are set with bigger screen area, and wrapped with ScrollView[10] to ensure no element to be hidden.

J. Espresso UI Testing

Espresso UI testing[17] conducted on the Main Activity and Booking activities of the "Rent With Intent" app. The tests utilizes AndroidJUnitRunner[18] and were designed to validate the functionality and correctness of UI components, ensuring that user interactions with the app behave as expected. Each test scenario was executed successfully, confirming the reliability of the user interface.

1. MainActivityTest.kt

The MainActivityTest class tests the core functionality of the Main Activity, focusing on ensuring all UI elements are present, navigation between instruments works as intended, navigation to Booking Activity, and dialogs open correctly.

testAllUIElementExist: This test verifies that all essential UI components of Main Activity are visible when the activity starts. Elements tested include the instrument name, price, category, brand, rating bar, image, and buttons (Borrow, Next, and Filter).

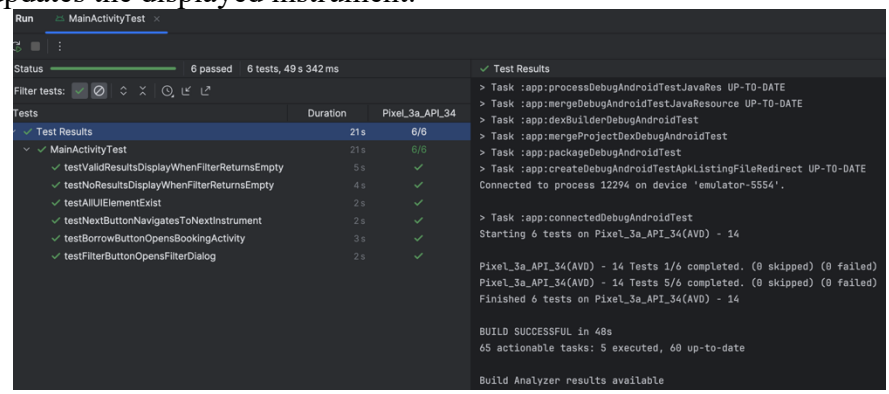
testNextButtonNavigatesToNextInstrument: This test validates the "Next" button functionality, ensuring that it navigates to the subsequent instrument in the list when clicked. The test confirms the initial instrument's name and checks that it changes correctly to the next instrument when the button is pressed.

testBorrowButtonOpensBookingActivity: Verified that clicking the "Borrow" button successfully opens the Booking Activity.

testFilterButtonOpensFilterDialog: Confirmed that the "Filter" button opens the filter dialog with appropriate UI elements.

testNoResultsDisplayWhenFilterReturnsEmpty: Ensured that applying an invalid filter selection displays a "No matching result" message and hides unrelated UI components.

testValidResultsDisplayWhenFilterReturnsEmpty: Confirmed that applying a valid filter correctly updates the displayed instrument.



All 6 test cases passed successfully.

2. BookingTest.kt

The BookingTest class focuses on verifying the functionality of the Booking Activity, including date selection through the DatePicker, navigation to Main Activity, confirming bookings, and cancellation correctly.

testValidDatePicker:
This test simulates a user selecting a valid future date using the DatePicker. It checks that after setting a valid date (select DatePicker using org.hamcrest.Matcher[19]), the "Confirm" button is displayed, allowing the user to proceed with booking. After confirming the booking, the test ensures that the app correctly returns to Main Activity.

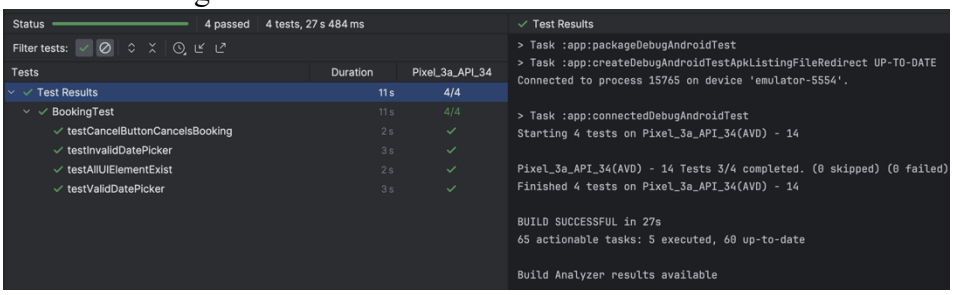
testInvalidDatePicker:
This test checks the behaviour when an invalid past date is selected in the DatePicker. After setting the past date, it ensures that the "Confirm" button is not displayed, indicating that the selection is invalid.

testCancelButtonCancelsBooking:
This test verifies that clicking the "Cancel" button in the Booking activity ends the booking process and correctly transitions back to Main Activity. The test checks the visibility of an element unique to Main Activity to confirm the transition.

setDate Function:

- getConstraints(): defines constraints to ensure the action is only applied to DatePicker. By returning isAssignableFrom(DatePicker::class.java)[19], it uses Matcher[21] to restrict the action to only be performed on DatePicker views.
- getDescription(): provides a human-readable description of what the action does, particularly useful for logging and debugging purposes.
- perform():
  - uiController parameter: Control the UI during the execution
  - view parameter: The target view on action, DatePicker

- The view is cast to a DatePicker to access its specific methods.
- The updateDate(year, month - 1, day) method is called on the DatePicker.
- The month value is adjusted by subtracting 1 because the DatePicker uses zero-based[20] indexing for months.



All 4 test cases passed successfully.

K. Reflection on Assignment 1

This section elaborates on the key reflections and the changes made between Assignment 1 (Climber app) to Assignment 2 (Rent With Intent).

Enhanced UI Components:

- **Assignment 1:** Used basic UI components such as buttons and spinners with limited dynamic interaction. The main focus was on scoring logic and maintaining the app state across orientation changes.
- **Assignment 2:** Introduced advanced UI elements like ChipGroup for filtering, DatePicker to select date, RatingBar for displaying ratings, and ScrollView for scrollable view, enhancing user interactivity. Custom styles and landscape-specific layouts improved the overall visual appeal and usability.

Improved Data Management:

- **Assignment 1:** Managed simple data with minimal inter-component communication. Used onSaveInstanceState for preserving score and state on orientation changes.
- **Assignment 2:** Implemented Parcelable for efficient data transfer between activities, facilitating complex interactions and improving performance. The ActivityResultLauncher ensured robust feedback and state updates between activities.

Enhanced User Feedback:

- **Assignment 1:** Relied on color changes and ripple effects for interaction feedback, similarly, using Toast and Log.d statements for debugging and user interactions.
- **Assignment 2:** Similarly, also utilises Toast and Log.d, accompanied with dynamic UI to hide unused elements and handle errors.

Advanced Orientation and State Management:

- **Assignment 1:** Managed state effectively with separate layouts and state-saving methods but was limited to basic orientation handling.
- **Assignment 2:** Expanded orientation handling with dynamic state management, preserving filter results and instrument indices, ensuring a seamless experience across configuration changes.

UI Testing:

- **Assignment 1:** Only test UI elements manually by user and developers upon feedbacks.
- **Assignment 2:** Combine both manual and automated UI testing with Espresso.

L. Quality Assurance

The Quality Assurance (QA) process for the development of the "Rent With Intent" app was designed to ensure a seamless user experience, optimal performance, and minimal bugs. This

section outlines the strategies employed to verify the quality of the app, focusing on testing approaches, challenges, and outcomes.

1. Testing Strategy

A combination of manual and automated testing methods was utilized to ensure comprehensive coverage of the app's functionality across different scenarios. The key aspects of the testing strategy[16] included:

- **Unit Testing**: Focused on verifying the functionality of individual components, particularly the ChipFilterDialog, Booking, and Main Activity. Unit tests ensured that the logic behind filtering instruments, managing intent data, and updating ratings was functioning as expected.
- **UI/UX Testing**: Verified that all UI components, such as ChipGroup, RatingBar, and DatePicker, behaved correctly across different screen sizes, orientations, and usage patterns. Special attention was given to landscape mode layouts to ensure proper rendering and positioning of elements.
- **Integration Testing**: Tested the interaction between activities, particularly ensuring that data passed between Main Activity and Booking Activity was correctly handled using Parcelable and ActivityResultLauncher. Integration tests also ensured the proper functioning of the Chip filter dialog in filtering instruments.
- **Device Compatibility**: The app was tested on multiple Android versions, ensuring compatibility across a variety of devices. Specifically, testing was performed on Android 12 (Tiramisu) to ensure the new APIs (e.g., getParcelableArrayList(String, Class<T>)) were used correctly.
- **Espresso UI Testing:**
  - **MainActivityTest.kt:** Automated UI tests were conducted on Main Activity using Espresso to validate the presence and interaction of key UI elements, such as navigating between instruments using the next button and verifying the filter dialog's functionality.
  - **BookingTest.kt:** Automated UI tests were implemented to ensure the proper functioning of booking actions, including validating date selections using DatePicker and testing cancelation flows. The tests confirmed the correct transition between Main Activity and Booking Activity while ensuring the appropriate feedback for valid and invalid date inputs.

2. Error Handling and Recovery

To ensure a robust user experience, the app was tested for various edge cases to validate proper error handling:

- **Filter Returns No Results**: The app dynamically handles cases where the filter returns no matching instruments. It ensures that the UI components are updated correctly (e.g., hiding buttons) and an appropriate message is displayed to the user.
- **Invalid Date Selection**: The DatePicker dialog implements validation logic to prevent users from selecting an invalid rental timeframe. Toast messages provide immediate feedback when errors occur, such as selecting a past date.
- **Orientation Change**: Special care was taken to ensure that the app could handle orientation changes without losing the current state of filters or selected instruments. The onSaveInstanceState() method ensures that the current instrument index and filtered instrument list are preserved during screen rotation.
- **Espresso Testing on Error States:** Tests like testInvalidDatePicker in BookingTest.kt and testNoResultsDisplayWhenFilterReturnsEmpty in MainActivityTest.kt were

specifically designed to validate that the UI responds correctly when users attempt invalid actions (selecting a past date or non-valid chip selection and combination).

3. Usability Testing

- **User Feedback**: The app was tested by users representing different personas (e.g., a parent renting for their child, a musician renting for a concert). This testing focused on verifying the ease of use of the filtering function (ChipFilterDialog) and the DatePicker for selecting rental periods.
- **Intuitive Design**: Custom styles, error messages, and dynamic UI updates were tested to ensure the app's visual elements were consistent, readable, and easy to interact with. The color scheme, toast messages, and logs were verified for optimal user feedback during interaction.

4. Challenges and Solutions

- **Fragment Initialization Error**: During development, a common issue was encountered when switching device orientation while the ChipFilterDialog was open. This was resolved by updating the fragment handling with the use of a companion object and the newInstance() method to ensure the correct passing of arguments.
- **Handling Deprecation**: Android API deprecations, such as getParcelableArrayList, required updates to newer methods (e.g., getParcelableArrayList(String, Class<T>)). This ensured compatibility with the latest Android versions and provided better type safety.

5. Logs and Debugging

- **Extensive Logging**: Log statements were strategically placed throughout the app, especially in the ChipFilterDialog and MainActivity, to track user interactions, filter selections, and rental booking status. This helped in early detection of bugs and provided a clear view of the app's workflow.
- **Error Tracking**: Logs and crash reports were monitored during testing to detect any unhandled exceptions. Most issues were identified and resolved, particularly those related to fragment lifecycle management and state restoration during orientation changes.

M. Conclusion

The "Rent With Intent" app effectively meets the needs of a music studio by allowing clients to browse, filter, and rent musical instruments efficiently. Key elements such as advanced UI components, robust data handling with Parcelable, and intuitive filtering through ChipGroup and renting period confirmation through DatePicker enhance user experience.

The implementation of custom styles, dynamic UI updates, and landscape-specific layouts ensures a visually appealing and responsive design. The use of ActivityResultLauncher for handling data transfer between activities further improves app interactivity and feedback.

All Espresso UI tests for the Main and Booking activities were executed successfully, demonstrating the accuracy and reliability of the app's UI components. The tests confirmed that essential functions such as navigation, filtering, date selection, and activity transitions behave as expected, providing a reliable user experience

Overall, the project demonstrates a solid understanding of Android development practices, showcasing the ability to create an engaging and functional application that aligns with

business requirements. The app's thoughtful design choices and effective handling of user interactions and feedback mark a significant improvement from previous assignments, highlighting the development of advanced skills in mobile app development.

N. Link to GitHub code repository

https://github.com/SoftDevMobDev-2024-Classrooms/assignment02-Lelekhoa1812

O. Acknowledgement

Technical References

1. Android Developers. Intent and intent filters. Retrieved from https://developer.android.com/guide/components/intents-filters?hl=vi

2. Android Developers. Dialog. Retrieved from https://developer.android.com/develop/ui/views/components/dialogs?hl=vi

3. GeeksforGeeks. (2022). DatePicker in Kotlin. Retrieved from https://www.geeksforgeeks.org/datepicker-in-kotlin/

4. GeeksforGeeks. (2022). Rating bar in Kotlin. Retrieved from https://www.geeksforgeeks.org/ratingbar-in-kotlin/

5. Stack Overflow. (2022). The getParcelableExtra method is deprecated. Retrieved from https://stackoverflow.com/questions/73019160/the-getparcelableextra-method-is-deprecated

6. Microsoft Learn. GetParcelableExtra method and usage. Retrieved from https://learn.microsoft.com/en-us/dotnet/api/android.content.intent.getparcelableextra?view=net-android-34.0

7. Assem, I. (2021). Custom dialog using dialog fragments. Medium. Retrieved from https://islamassem.medium.com/custom-dialog-using-dialog-fragments-de6a0874b6a4

8. GeeksforGeeks. (2022). How to make an alert dialog fill majority of the screen size in Android. Retrieved from https://www.geeksforgeeks.org/how-to-make-an-alert-dialog-fill-majority-of-the-screen-size-in-android/

9. Estefania Cassingena Navone. (2018). How to implement and use a Parcelable class in Android: Part 1. Medium. Retrieved from https://medium.com/techmacademy/how-to-implement-and-use-a-parcelable-class-in-android-part-1-28cca73fc2d1

10. GeeksforGeeks. (2024). ScrollView in Android. Retrieved from https://www.geeksforgeeks.org/scrollview-in-android/

11. Android Developers. DisplayMetrics. Retrieved from https://developer.android.com/reference/android/util/DisplayMetrics

12. Android Developers. ActivityResultLauncher. Retrieved from https://developer.android.com/reference/androidx/activity/result/ActivityResultLauncher

13. Material Designs. Chips. Retrieved from https://m2.material.io/components/chips/android#using-chips

14. Android Developers. android.os.Bundle. Retrieved from https://developer.android.com/sdk/api_diff/33/changes/android.os.Bundle

15. Kotlin. (2024). Object declarations and expressions. Retrieved from https://kotlinlang.org/docs/object-declarations.html

16. Shake. (2023). How to write great QA documentation for your mobile app. Retrieved from https://www.shakebugs.com/blog/how-to-write-qa-documentation/

17. Luis Marchani. (2022). Get started with Espresso & Kotlin!. Medium. Retrieved from https://medium.com/globant/get-started-with-espresso-kotlin-8aa5ca935489

18. Android Developers. AndroidJUnitRunner. Retrieved from https://developer.android.com/reference/androidx/test/runner/AndroidJUnitRunner

19. Gradle User Manual. isAssignableFrom. Retrieved from https://docs.gradle.org/current/kotlin-dsl/gradle/org.gradle.api.reflect/-type-of/is-assignable-from.html

20. Stack Overflow. (2021). DatePicker shows wrong value of month. Retrieved from https://stackoverflow.com/questions/4467816/datepicker-shows-wrong-value-of-month

21. Android Developers. ViewMatchers. Retrieved from https://developer.android.com/reference/androidx/test/espresso/matcher/ViewMatchers

22. GenAI was used to create the app logo, DALL-E prompt used and output.
Prompt: “generate an image logo for my musical instrument selling app, which may have to include a vast range of different musical instrument within the logo. Try to make the logo look classical.”
Output:

Here is the classical-style logo for your musical instrument selling app, featuring a variety of instruments in a timeless and elegant design. Let me know if you'd like any adjustments or further enhancements!

Instrument Images References

World of Music. Cordoba C5 Classical Guitar. Retrieved from https://www.worldofmusic.com.au/guitars/classical-guitars/cordoba-c5-classical-guitar/

DC Pianos. Steinway Model B Grand Piano - Satin Ebony. Retrieved from https://dcpianos.com/products/steinway-model-b-grand-piano-in-satin-ebony-236077/

Vivace Violins. Gliga Vasile Violin. Retrieved from https://vivaceviolins.com.au/shop/string-instruments/violins/mid-range-violins/gliga-vasile-violin/

Piano City. Yamaha Piaggero NP-15B 61Key Keyboard. Retrieved from https://pianocity.com.au/product/yamaha-piaggero-np-15b/

Sax & Woodwind. Cambridge TR620L Trumpet. Retrieved from https://www.saxandwoodwind.com.au/Product/cambridge-advanced-student-trumpet-gold-lacquer

Google Shopping. Yamaha YTS480 Intermediate Tenor Saxophone. Retrieved from https://www.google.com/shopping/product/3434598287014914385?sca_esv=8e8ead8949b91364

Musocity. Sanchez Soprano Ukulele - Natural Satin. Retrieved from https://www.musocity.com.au/products/sanchez-c20-soprano-ukulele-natural-satin

World of Music. Jet Black Pearl Roadshow Fusion Plus Drum Kit. Retrieved from https://www.worldofmusic.com.au/drums/acoustic-drum-kits/pearl-roadshow-22-fusion-kit-jet-black/

Artist Guitars. Artist ST62 Fiesta Red Electric Guitar. Retrieved from https://www.artistguitars.com.au/buy/Artist-ST62FR-Electric-Guitar-Fiesta-Red

eBay. Cambridge Double Key Tenor Flute. Retrieved from https://il.ebay.com/b/Cambridge-Band-Orchestral-Woodwind-Instruments/181267/bn_7437886

Violins.com.au. Yamaha SLB300 Silent Double Bass. Retrieved from https://www.violins.com.au/products/silent-bass-yamaha-slb300-outfit

Wollongong Music. Yamaha Stage Custom Birch Drum Kit. Retrieved from https://www.wollongongmusic.com.au/drums-percussion/drum-kits/yamaha-stage-custom-birch-drum-kit