SWINBURNE VIETNAM

HO CHI MINH CAMPUS

CLASS: COS30017

Assignment 2 Report

INSTRUCTOR: Dr. Tan Le

STUDENT NAMES: Huu Nhan Le - 1040711133

HO CHI MINH CITY – March, 12th 2025

1. Acknowledgement

No Generative AI tools were used for this task

2. Introduction

In assignment 2, I was required to design an app for a music studio to rent instruments and equipment monthly. In addition to several must-have features like display of instruments on the screen once at a time, a next button to navigate to the next item, or the attributes of instruments such as name, price, ratings, I also added multilingual support, screen rotation, logging and many others. This report will cover my development process including UI/UX draft for user stories, some key design decisions for the app, detailed explanation of UI/UX and functionalities, as well as the testing phase. At the end of the report, there will be a reflection section to roughly summarize what I feel like is done well in this project and what can be enhanced in future works.

3. Time Logs

Date	Time	Description
01/03	1 day	Review the knowledges required for developing this app, especially multi-activity, using intents through lectures and labs in week 5 and 6
02/03	2 hours	Design the draft for the layout of the app based on the user story
02/03	1 day	Design the MainActivity layout for both portrait and landscape mode
02/03	2 hours	Import necessary resources like instrument images, icons
03/03 - 04/03	2 hours	Implement the Instrument data class
04/03	7 hours	Implement the functions to progress through items in MainActivity
05/03	3 hours	Add customized styles and change MainActivity layout to use it
05/03	5 hours	Design the Booking Activity layout for both portrait and landscape mode
06/03	4 hours	Implement the function to toggle language in MainActivity and BookingActivity
06/03	4 hours	Implement the intents in MainActivity to pass selected instrument information, credit balance to BookingActivity

06/03	3 hours	Implement the dynamic change of slider in BookingActivity depending on the number of number instrument in stock
07/03	5 hours	Implement the input validation in BookingActivity such as email validation, credit balance check
07/03 - 08/03	8 hours	Implement the functions to get the result from BookingActivity back to MainActivity
08/03	6 hours	Implement a dialog in MainActivity to display the detailed information of rental instances
09/03	1 day	Design 3 Espresso tests for the app
10/03 - 12/03	2 days	Fix any remaining bugs in the app

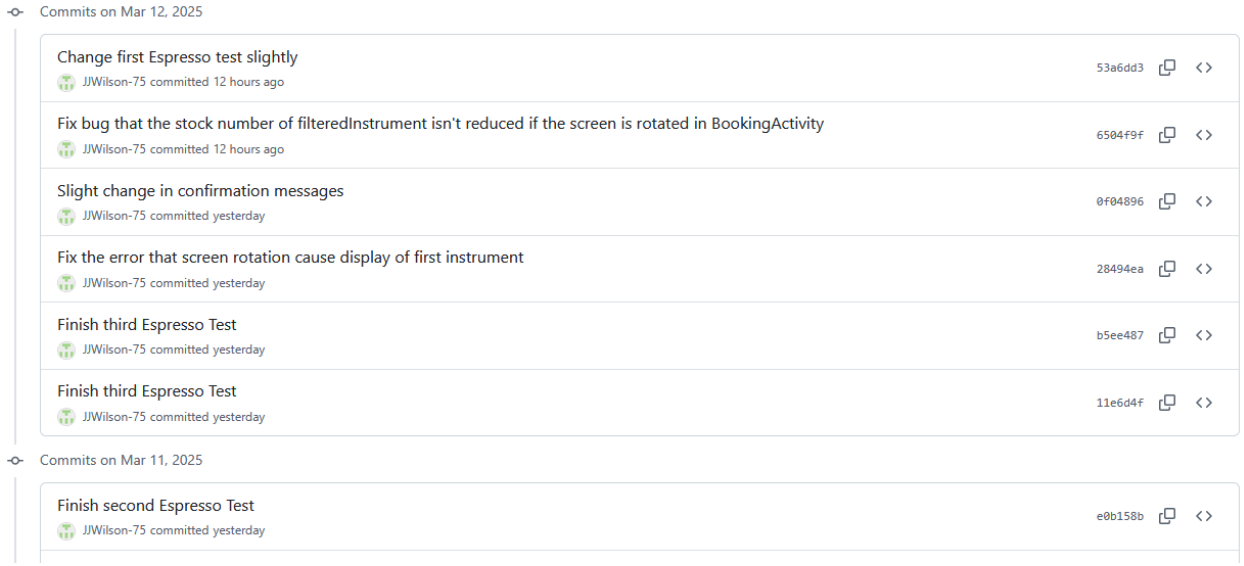

Figure 1 - Commit history on Github Classroom

4. UI/UX

4.1. User stories

User story 1: As a band manager, I want to rent several instruments so that the members in my band can play for an upcoming show.

User story 2: As a foreign guitarist, I want to rent some guitars so that I can try them before deciding on which to buy.

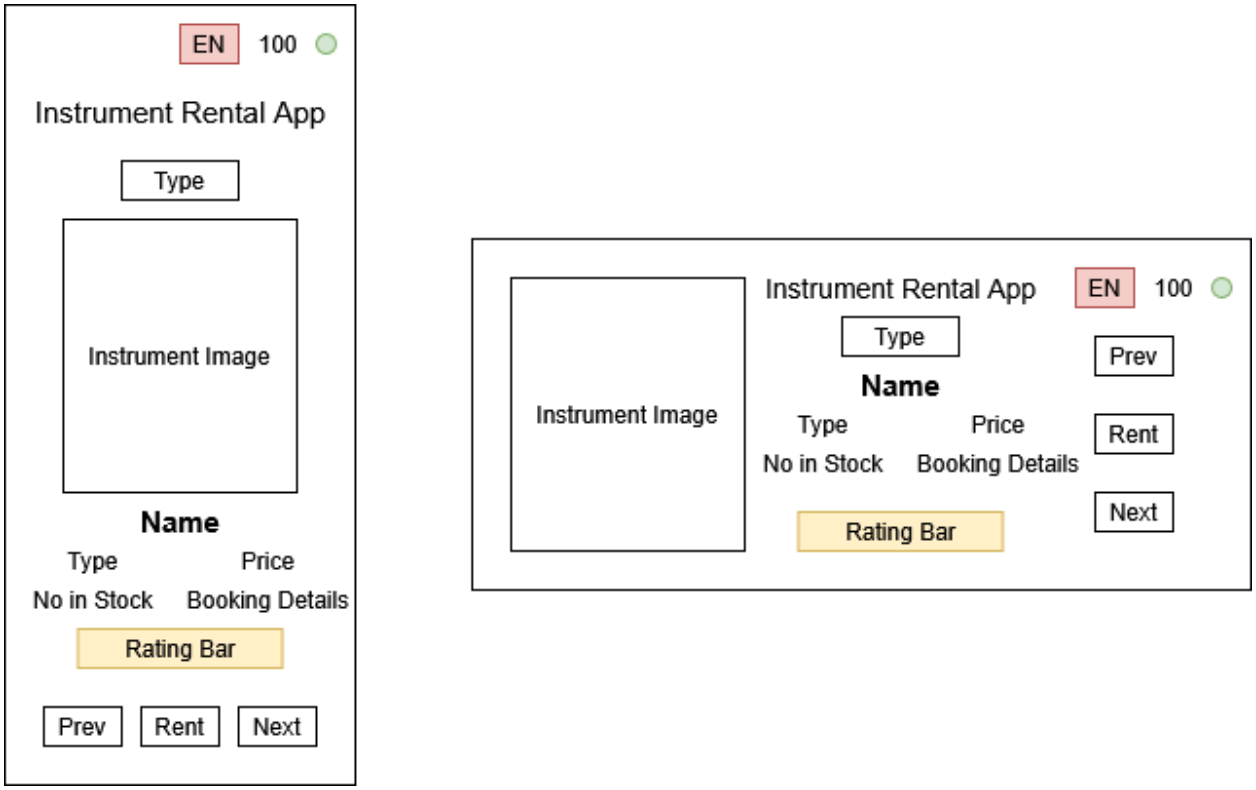

Figure 2 - Sketches of portrait and landscape layout

Use case: In the sketches, I decided to split the UI into 4 main parts. The first part contains a button to toggle between 2 languages, and a textview to display the account balance of the user. The toggle language button, with one of the languages being English, is part of the multilingual support implemented for this app, which will definitely be useful for the second user story as the user is a foreigner. The next part includes an app name and a chip to choose which filter out types of instrument. Initially, I planned this to be a button, upon being clicked would open a small dialog to show different chips of instrument types. However, when I reached the designing phase, I changed it to display all the chips right on the screen to simplify the process a little bit. Regardless, this widget can be applied for both user stories, because of many reasons. The band manager in user story 1 wants to rent many instruments for a show, which can imply different types of instruments, while the guitarist in the second user story can now filter out guitars only to view on the screen. The third part of the UI is dedicated to the information of the instrument, such as the image, name, type, price, number in stock, booking details. Most of them are attributes required directly by the assignment, but they are also useful for both user stories. The actor, the band manager and the guitarist, can see all the information of the instruments before making a decision. Finally, at the bottom of the screen, the app has 3 buttons, 2 (Prev and Next) for navigating through the instruments and 1 for transitioning into the second activity for rental.

4.2. MainActivity layouts

4.2.1. Portrait

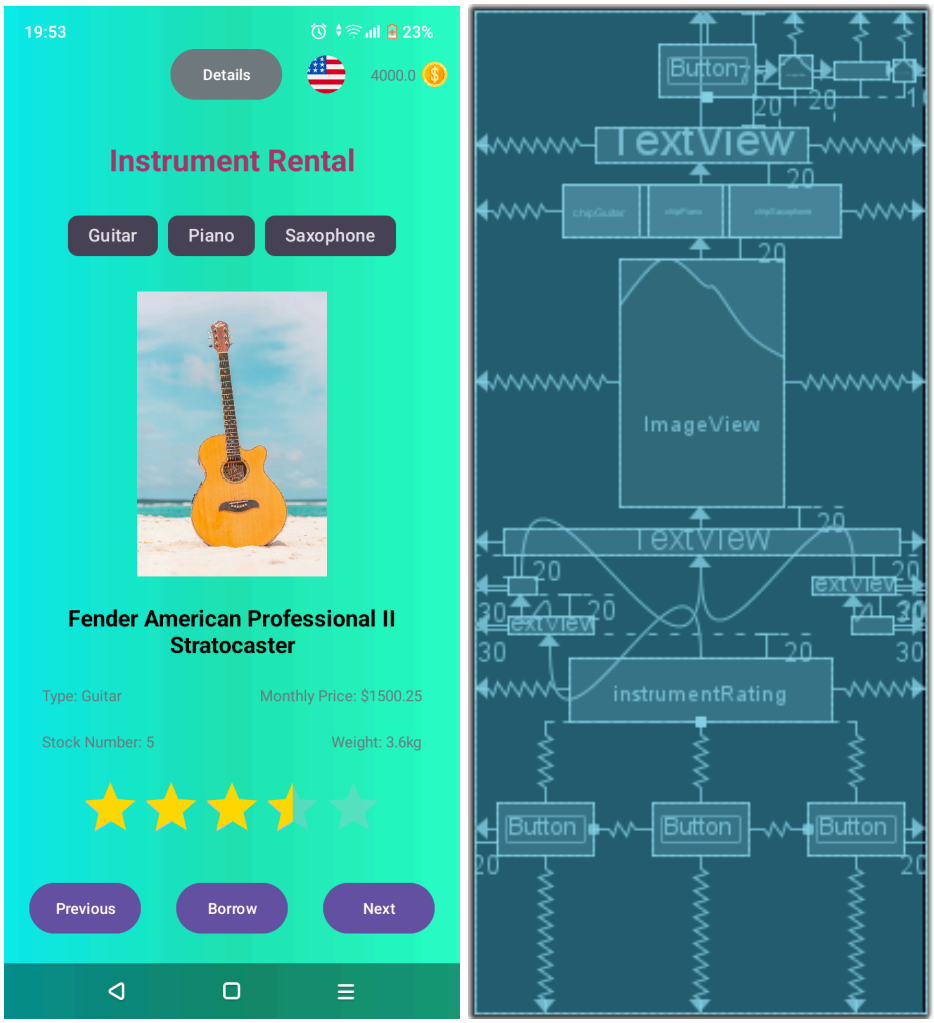

Figure 3 - Portrait layout in Main Activity

As you can see here, I have made some major changes from sketches to actual design. Firstly, at the top of the screen, from left to right, I included the button btnInstrumentBorrow with the label “Details” to show the rental details, an ImageView with the small icon of English or Vietnamese flags to toggle between those 2 languages, a TextView showing the credit balance of the user, and an ImageView of the coin icon. The “Details” button is normally set to Disabled state with the attribute android:enabled="false”, unless a rental attempt has been completed successfully.

Right below them is the TextView for app name, and then the ChipGroup (chipGroupType) with all the individual Chips (chipGuitar, chipPiano, chipSaxophone). The attribute app:selectionRequired of the ChipGroup is set to true to make sure that there is always at least one Chip selected, preventing the situation where there are no instruments displayed on screen. In addition, all 3 Chips have android:checkable and android:checked attributes set to true, making all 3 chips selected by default.

For the next part, I used 1 ImageView (instrumentPicture), 5 TextView (instrumentName, instrumentType, instrumentPrice, instrumentStockNumber, instrumentWeight), and 1 RatingBar (instrumentRating) elements to display the data of the instrument. The height and the width of the ImageView instrumentPicture is hardcoded to a 2:3 ratio (150dp in width and 225dp in height). Meanwhile, in the RatingBar instrumentRating widget, the android:numStar attribute is set to 5 to indicate the max stars/rating items are 5 and the android:stepSize attribute is set to 0.5 to indicate the step size of the rating bar is 0.5 (*RatingBar*, n.d.). I also set true to android:isIndicator to make the instrumentRating widget unchangeable by the user (RatingBar, n.d.). All 7 widgets dedicated to displaying information of the equipment can also be updated if the user navigates to the next or the previous instrument with the displayInstrument function in MainActivity.
And similar to the sketches, there are 3 buttons below the information widgets that are mainly used for moving around inside the app. The 2 buttons at the edge, btnPrev and btnNext, do exactly what their name suggests, go to the previous or next instruments when clicked. The center button, btnBorrow, is used for setting more details for the rental of the instrument in the second activity, BookingActivity.
The layout type I chose throughout the entire app is Layout as I have used it in the first assignment and it allows for a complex layout without nested view groups (*Build a Responsive UI with ConstraintLayout*, n.d.). Moreover, the position of the elements in the layout can be manipulated with a combination of constraints, margins, and constraint biases (*Build a Responsive UI with ConstraintLayout*, n.d.).

4.2.2. Landscape

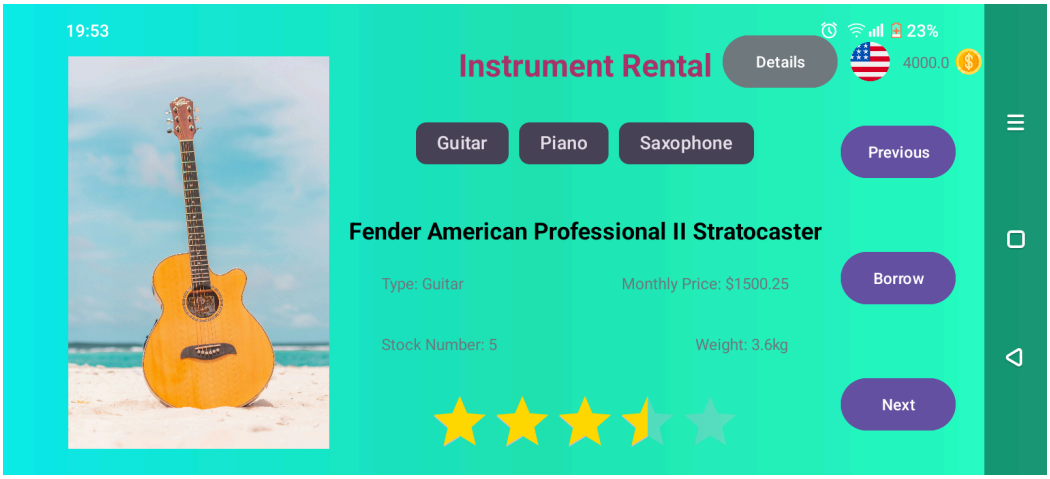

Figure 4 - Landscape layout in MainActivity

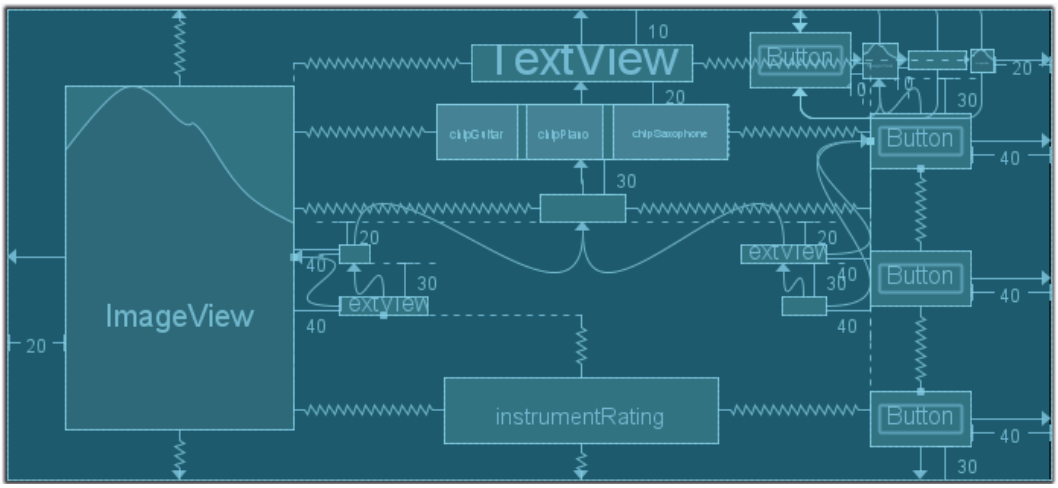

Figure 5 - Blueprint of landscape layout in MainActivity

The landscape layout of the MainActivity has all the elements from the portrait layout, except their position in the screen. To the left side of the phone, you can see the ImageView instrumentPicture occupying nearly one third of the screen. The android:layout_width and android:layout_height are now set to 200 and 300 dp respectively, so the user can see the image of the displayed instrument much clearer in this orientation.

On the top-right corner is where the btnInstrumentBorrow, btnChangeLang, creditAccount, and coinImage widgets are placed. Then right below it, the main 3 buttons in MainActivity, btnPrev, btnBorrow, and btnNext are lined up vertically.

In the center of the screen, the layout includes appName, chipGroupType, instrumentName, instrumentType, instrumentPrice, instrumentStockNumber, instrumentWeight, and instrumentRating. Their constraints are mostly set to the end of instrumentPicture and the start of btnPrev to make the widgets stay in the center.

4.3. MainActivity layouts

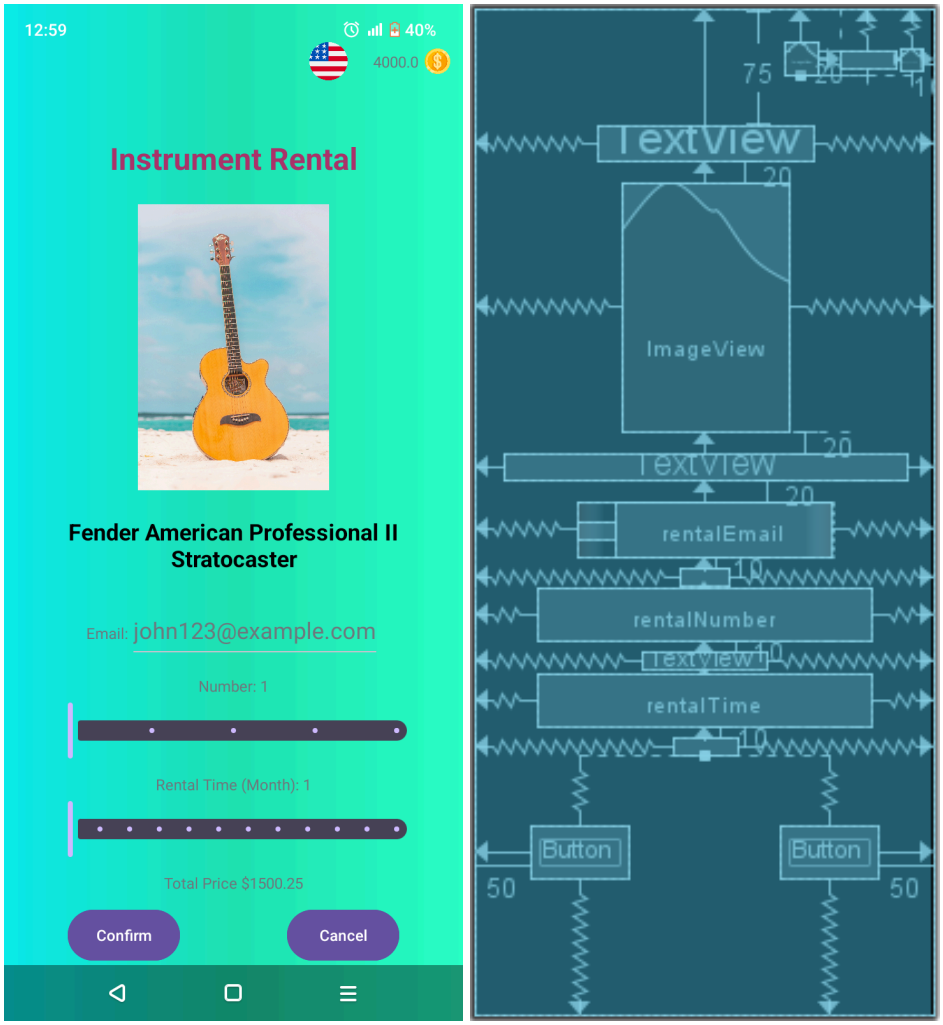

Figure 6 - Portrait layout in BookingActivity

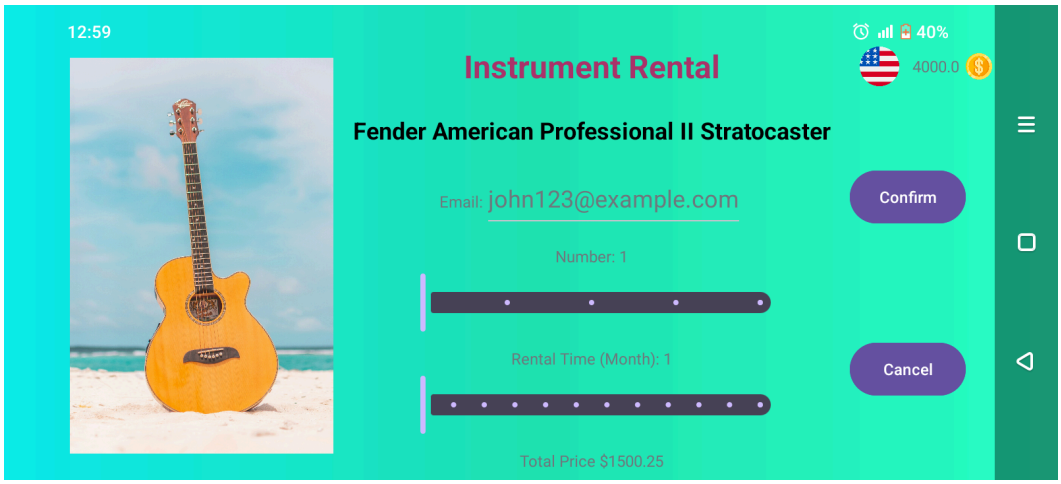

Figure 7 - Landscape layout in BookingActivity

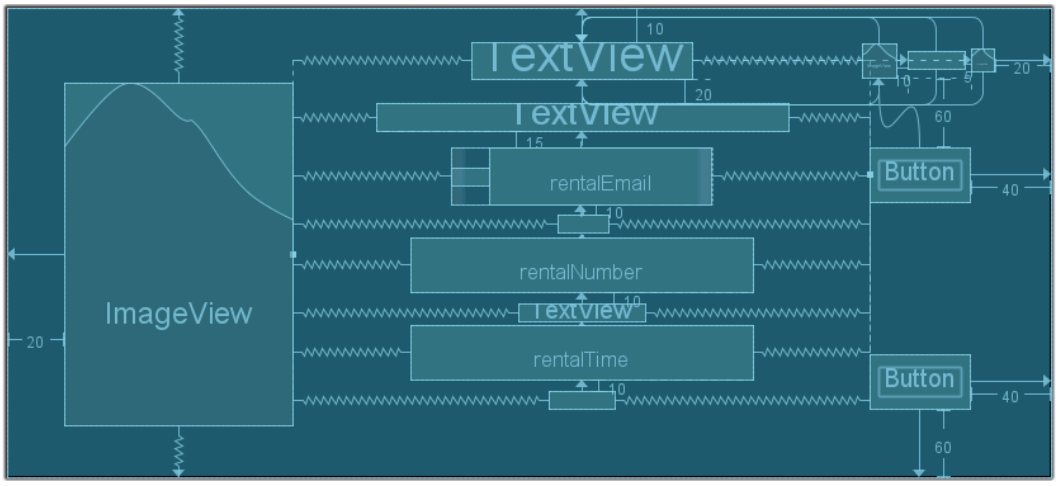

Figure 8 - Blueprint of landscape layout in BookingActivity

The portrait and landscape layouts in BookingActivity are mostly modified based on their counterparts in MainActivity to keep the thematic consistency in the app. Therefore, in this and the next section, I will just cover the differences between this and the MainActivity portrait layout. For instance, the btnInstrumentBorrow button does not exist in BookingActivity because I want the users to be able to view the rental details only after they completed the rental attempt. After all, this activity will display the necessary information required for the instrument rental below anyways.

Speaking of that, the instrumentType, instrumentPrice, instrumentStockNumber, instrumentWeight, and instrumentRating elements will be replaced. To begin with, to the bottom of instrument Picture in portrait mode or the right-hand side of instrumentPicture in landscape mode, I had a LinearLayout with horizontal orientation containing a TextView called labelRentalEmail with the label “Email:” and an EditText called rentalEmail to type in the email. After that, the app had 2 Slider components (rentalNumber and rentalTime) used for the number of instruments and the time rented. Both components have the android:stepSize and android:valueFrom attributes set to 1 to make the step size and the minimum value of the slider 1 respectively (Sliders – Material Design 3, n.d.). However, the max value of rentalTime slider is set to 12 to limit the rental time of a user to 12 months with “ android:valueTo="12" ”, while the rentalNumber slider can have a dynamic ranges of value with some functions in BookingActivity.kt (Sliders – Material Design 3, n.d.). In addition, both sliders also come with a label (labelRentalNumber and labelRentalTime) right above them to immediately update the text when the value of the sliders is changed by the user. The total price of each rental instance will also be recalculated and updated to the android:text attribute of rentalPrice TextView widget.

Finally, the 3 main buttons in MainActivity (btnPrev, btnBorrow, and btnNext) are replaced with 2 new buttons, btnConfirm and btnCancel. The functionalities of these 2 new buttons are exactly like their names, confirming or canceling the rental action.

4.4. Customized styles

As the assignment specifically requested the app must use at least 2 styles in 2 locations, I tried to combine styles for a variety of Views to create my own theme for the app (Styles and Themes, n.d.). This theme, which is declared in the themes.xml in the values folder, is used in both MainActivity and BookingActivity and in both portrait and landscape mode (Styles and Themes, n.d.).

4.4.1. AppBackground

```
<style name="AppBackground">
    <item name="android:background">@drawable/custom_app_gradient_background</item>
</style>
```

Figure 9 - AppBackground <style> element

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
    <item>
        <shape android:shape="rectangle">
            <gradient
                android:angle="0"
                android:startColor="#0cebeb"
                android:centerColor="#20e3b2"
                android:endColor="#29ffc6"
                android:type="linear" />
        </shape>
    </item>
</layer-list>
```

Figure 10 - custom_app_gradient_background.xml file in drawable folder

The first style, AppBackground, is extremely basic, as it just has one <item> element to set the android:background attribute to a xml resource file called custom_app_gradient_background in the drawable folder. I used it to set the background of both MainActivity and BookingActivity to a gradient color of green like you see in the layouts with the style="@style/AppBackground" attribute in the root ConstraintLayout widgets.

4.4.2. Title

```
<!-- Title Style for App Name -->
<style name="Title">
    <item name="android:textSize">24sp</item>
    <item name="android:textColor">@color/dark_pink</item>
    <item name="android:textStyle">bold</item>
    <item name="android:textAlignment">center</item>
</style>
```

Figure 11 - Title <style> element

Figure 12 - Example of Title style usage

For the appName TextView elements, a style called Title is used. From the figure of the xml code, the style will change the following attributes: the textSize to 24sp, textColor to a customized color called dark_pink declared in colors.xml file, textStyle to bold, and textAlignment to center. There is a figure example above demonstrating how the app name would look with this style.

4.4.3. Heading

```
<!-- Heading Style for Instrument Name -->
<style name="Heading">
    <item name="android:textSize">18sp</item>
    <item name="android:textColor">@color/black</item>
    <item name="android:textStyle">bold</item>
    <item name="android:textAlignment">center</item>
</style>
```

Figure 13 - Heading <style> element

Figure 14 - Example of Heading style usage

The Heading style uses the same attributes as the Title style (the textSize to 18sp, textColor to the default black color declared in colors.xml file, textStyle to bold, and textAlignment to center). It is intended as a lower level of the Title style, like how different levels of headings in Google Docs work. I applied this for all appName TextView widgets in the app, in order to make the name of the instruments more prominent than other details.

4.4.4. Normal

```
<!-- Normal Text Style for Instrument Attributes -->
<style name="Normal">
    <item name="android:textSize">12sp</item>
    <item name="android:textColor">@color/steel_gray</item>
</style>
```

Figure 15 - Normal <style> element

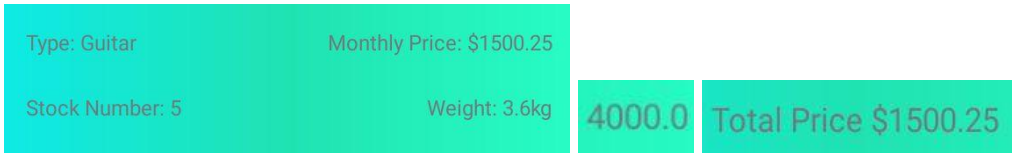

Figure 16 - Examples of Normal style usage

The Normal style, the most used <style> element in the entire app, set the android:textSize attribute to 12sp and android:textColor attribute a customized color called steel_gray declared in the colors folder. It is applied with the code line " style="@style/Normal" " to a variety of elements in the layouts, such as instrumentType, instrumentPrice, creditAccount, labelRentalNumber, etc.

4.4.5. RatingBarStyle

```
<style name="RatingBarStyle">
    <item name="android:progressTint">@color/gold</item>
</style>
```

Figure 17 - RatingBarStyle <style> element

Figure 18 - Example of RatingBarStyle style usage

The RatingBarStyle, as the name implies, is used to customize the RatingBar widget in the layouts. It contains one single <item> element to change the android:progressTint attribute of the RatingBar to the color gold declared in colors.xml file (Rana, 2020).

4.4.6. ButtonStyle

```
<!-- Button Style -->
<style name="ButtonStyle">
    <item name="android:textSize">12sp</item>
    <item name="android:padding">4dp</item>
    <item name="android:textAlignment">center</item>
    <item name="android:backgroundTint">@drawable/button_selector</item>
    <item name="android:textColor">@color/white</item>
</style>
```

Figure 19 - ButtonStyle <style> element

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Disabled state -->
    <item android:state_enabled="false" android:color="@color/steel_gray" />
    <!-- Default state -->
    <item  android:state_enabled="true" android:color="#ff6750a4" />
</selector>
```

Figure 20 - button_selector.xml file in drawable folder

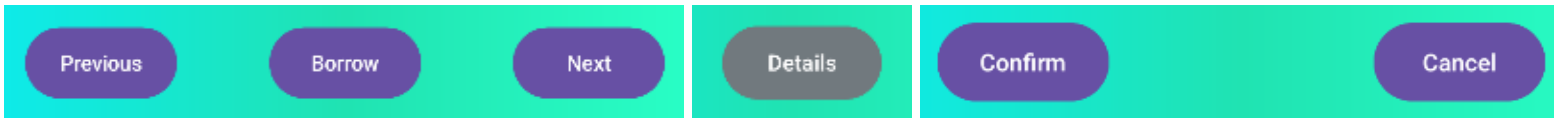

Figure 21 - Examples of ButtonStyle style usage

All the buttons in the Instrument Rental app look similar to each other due to the usage of the ButtonStyle. This style customized the following attributes of a button: the textSize to 12sp, textColor to the default white color declared in colors.xml file, textAlignment to center, padding to 4dp, backgroundTint to a specific button_selector.xml file in drawable. This button_selector.xml is a part I reused from assignment 1, which allowed me to choose the specific background color for the disabled and enabled states of a button.

4.5. Toast vs Snackbar

Toast and snackbar are mostly identical to one another (*Orange Design System*, 2020). Toast is more lightweight, while snackbar can include one action and be wiped off the screen (*Orange Design System*, 2020). However, for this project, I don't think I particularly need the snackbar for any scenarios in my app. I needed either toast or snackbar for 6 different messages in total, 3 for input validation error in BookingActivity, 1 for cancel and confirmation of instrument rental

each, and 1 for a specific condition in which there aren't any instruments available in stock. So I ultimately went with toast for the entire app to maintain consistency as well as keeping the app consuming less resources on my device.

5. Functionality

5.1. Instrument data class

```
data class Instrument(
    val name: String,
    val type: InstrumentType,
    val price: Float,
    var stockNumber: Int, //var to enable it being reduced when borrowed
    val weight: Float, // Weight in kg
    var rating: Float
) : Parcelable {
    constructor(parcel: Parcel) : this(
        parcel.readString() ?: "",  // For name string, if null return empty string
        InstrumentType.valueOf(parcel.readString() ?: InstrumentType.GUITAR.name),
        parcel.readFloat(),  //?: 1.0f,  // For price float, if null return 1.0f
        parcel.readInt(),  //?: 0,  // Read stockNumber integer, if null return 0
        parcel.readFloat(), // Read weight
        parcel.readFloat(),  //?: 1.0f    // Read rating float, if null returns 1.0f
    )

    override fun writeToParcel(parcel: Parcel, flags: Int) {
        parcel.writeString(name)
        parcel.writeString(type.name)
        parcel.writeFloat(price)
        parcel.writeInt(stockNumber)
        parcel.writeFloat(weight)
        parcel.writeFloat(rating)
    }

    override fun describeContents(): Int = 0
```

Figure 23 - Instrument data class code - Part 1

```
enum class InstrumentType(val key: String) {
    GUITAR("guitar"),
    PIANO("piano"),
    SAXOPHONE("saxophone");
}
```

Figure 24 - InstrumentType enum class

5.1.1. Properties

The Instrument data class is used as a model to store all the attributes of an instrument, except the image which is handled separately in MainActivity. It has 6 properties: name of String type; type of InstrumentType enum type; price of Float type; stockNumber of Int type; weight of Float type; and weight of Float type. The InstrumentType, containing 3 possible types (GUITAR, PIANO, and SAXOPHONE) is a special enum class designed by me to support multilingual features later on, as each type has a useful corresponding string. All properties are declared with the val keyword, to assign their values once, with the exception of stockNumber (Basic Syntax | Kotlin, n.d.). The stockNumber property is declared with the var keyword, so its value can be changed if the user successfully rent any instruments (Basic Syntax | Kotlin, n.d.).

5.1.2. Parcelable interface

```
override fun describeContents(): Int = 0

companion object CREATOR : Parcelable.Creator<Instrument> {
    //This method reads the data from the Parcel and returns a new Instrument object.
    override fun createFromParcel(parcel: Parcel): Instrument {
        return Instrument(parcel)
    }

    //This is used when an array of Instrument objects needs to be created
    override fun newArray(size: Int): Array<Instrument?> {
        return arrayOfNulls(size)
    }
}
```

Figure 25 - Instrument data class code - Part 2

This data class also implements the Parcelable interface to enable the transfer of Instrument instances between MainActivity and BookingActivity through intents (*Solutions: Multiples (core)*, 2025). The second constructor in this class will read values from a Parcel object and create a new Instrument instance, utilizing methods like readFloat(), readInt(), readString(). The type

property is a special case, as it needs to read the string value first (default to guitar) before converting it to an InstrumentType enum value with the valueOf() method. The writeToParcel() method, on the other hand, serializes the Instrument object by writing its properties into a Parcel object with Parcel methods like writeString(), writeFloat(), writeInt().
The CREATOR companion object implements Parcelable.Creator<Instrument, with 2 overridden methods. The first one, createFromParcel, passes the Parcel object to the secondary constructor to create a new Instrument object. The second method, newArray, creates an array of Instrument objects with the size of size parameter.

5.1.3. Usage in MainActivity

```
private var instrumentsList = listOf(
    Instrument("Fender American Professional II Stratocaster", InstrumentType.GUITAR, 1500.25f, 5, 3.6f, 3.5f)
    Instrument("Gibson Les Paul Standard '50s", InstrumentType.GUITAR, 2500.5f, 3, 4.1f, 2.5f),
    Instrument("Steinway & Sons Model D Concert Grand", InstrumentType.PIANO, 170.5f, 2, 480.0f, 5.0f),
    Instrument("Yamaha FG800 Acoustic", InstrumentType.GUITAR, 220.75f, 7, 2.9f, 4.5f),
    Instrument("Yamaha P-125 Digital", InstrumentType.PIANO, 650.0f, 4, 110.8f, 2.5f),
    Instrument("Selmer Paris Series II Tenor", InstrumentType.SAXOPHONE, 3900.5f, 3, 3.2f, 4f)
)
```

Figure 26 - The variable list in MainActivity using the Instrument data class

At the start of MainActivity class, I initialized a list with the size of 6 to store all the information of all the instruments available in the music studio. The list uses the var keyword due to the same reason as the stockNumber property in Instrument data class, as the number of instruments fro a specific brand must be able to change during MainActivity.

5.1.4. Advantages of using Parcelable object on Android

Using Parcelable objects in Android provides a highly efficient way to serialize data for inter-process communication and transferring objects between activities or fragments. Unlike Serializable, which uses reflection and is slower, Parcelable is specifically optimized for Android, reducing memory overhead and execution time. This makes it ideal for passing complex data structures like lists or custom objects through Intent extras or Bundle, ensuring smoother app performance and responsiveness.

5.2. Chips Filter

```
private fun filterInstruments() {
    val selectedTypes = mutableSetOf<InstrumentType>()

    if (findViewById<Chip>(R.id.chipGuitar).isChecked) selectedTypes.add(InstrumentType.GUITAR)
    if (findViewById<Chip>(R.id.chipPiano).isChecked) selectedTypes.add(InstrumentType.PIANO)
    if (findViewById<Chip>(R.id.chipSaxophone).isChecked) selectedTypes.add(InstrumentType.SAXOPHONE)

    filteredInstruments = if (selectedTypes.isEmpty()) {
        instrumentsList // Show all if nothing is selected
    } else {
        instrumentsList.filter { it.type in selectedTypes }
    }

    currentInstrumentIndex = 0
    displayInstrument(currentInstrumentIndex)
}
```

Figure 27 - The filterInstruments() function

In the onCreate() function, I used this code line " chipGroupType.setOnCheckedStateChangeListener { _, _ -> filterInstruments() } " to set the listener for the chipGroupType ChipGroup function, which will call the filterInstrumetns() method when the chip's checked state changes (ChipGroup, 2019). In the filterInstruments() function, a mutableSet of InstrumentType, selectedTypes objects is initialized first to store what types of instruments the user has selected. Following that, a series of 3 conditions check whether all 3 chips have been checked with the isChecked method to add the proper instrument type in the selectedTypes list. Then, the selectedTypes list doesn't contain anything, then all instruments will be displayed in MainActivity (which acts as a safeguard only as the app:selectionRequired="true" of the chipGroupType has guaranteed at least one type must be selected). Otherwise, the filtered elements in the instrumentsList list will be included in the new filteredInstruments list. The currentInstrumentIndex will be reset to 0 and the displayInstrument with the currentInstrumentIndex argument will be called to show the first element of filteredInstruments list.

5.3. Display instruments

```
private fun displayInstrument(index: Int) {
    val instrument = filteredInstruments[index]
    instrumentName.text = instrument.name
    instrumentType.text = getString(R.string.label_text_type) + ": " + getLocalizedType(instrument.type)
    instrumentPrice.text = getString(R.string.label_text_price) + ": " + "$${instrument.price}"
    instrumentStockNumber.text = getString(R.string.label_text_stock_number) + ": " + "${instrument.stockNumber}"
    instrumentWeight.text = getString(R.string.label_text_weight) + ": ${instrument.weight}kg"
    instrumentRating.rating = instrument.rating

    // Set instrument image according to instrument name
    instrumentPicture.setImageResource(when (instrument.name) {
        "Fender American Professional II Stratocaster" -> R.drawable.guitar_picture_1
        "Gibson Les Paul Standard '50s" -> R.drawable.guitar_picture_2
        "Steinway & Sons Model D Concert Grand" -> R.drawable.piano_picture_1
        "Yamaha FG800 Acoustic" -> R.drawable.guitar_picture_3
        "Yamaha P-125 Digital" -> R.drawable.piano_picture_2
        else -> R.drawable.saxophone_picture_1
    })

    creditAccount.text = "$currentCreditAccountBalance"

    // Disable buttons if only one instrument is available
    if (filteredInstruments.size == 1) {
        btnNext.isEnabled = false
        btnPrev.isEnabled = false
    } else {
        btnNext.isEnabled = true
        btnPrev.isEnabled = true
    }
}
```

Figure 28 - The displayInstrument() function

This function will require a parameter of Int type. Firstly, the function initialized the local variable called instrument with the element at the index of filteredInstruments list. Then, the instrument's name, type, price, stock number, weight, and rating are displayed by modifying the attributes of the already initialized variables in onCreate() corresponding to instrumentName, instrumentPrice, instrumentType, instrumentStockNumber, instrumentWeight, and instrumentRating UI elements. To support 2 languages, some values are formatted with the getString() method and a string id. The getLocalizedType() method is used for translating the type of instruments, but it will be explained in other sections.
instrumentPicture is an exception here, as it dynamically updates the instrument image based on the instrument's name. A map instrument is applied to map specific instrument names to corresponding drawable resources. For example, if the instrument is a "Fender American Professional II Stratocaster", it assigns R.drawable.guitar_picture_1 as its image.
The text of creditAccount TextView is also updated with the value of the currentCreditAccountBalance variable. In the final step, the size of the filteredInstruments list is checked if it is equal to 1. If yes, then both btnNext and btnPrev buttons are disabled to prevent

the user from progressing to non-existing items. If not, then they are both enabled to allow the user to navigate to other instruments on the screen.

5.4. Items navigation

```
btnNext.setOnClickListener {
    navigateInstrument(1)
    Log.i("NAVIGATION", "NEXT - Instrument: ${filteredInstruments[currentInstrumentIndex].name}")
}

btnPrev.setOnClickListener {
    navigateInstrument(-1)
    Log.i("NAVIGATION", "PREVIOUS - Instrument: ${filteredInstruments[currentInstrumentIndex].name}")
}
```

Figure 29 - The listeners for btnNext and btnPrev buttons

```
private fun navigateInstrument(direction: Int) {
    if (filteredInstruments.isNotEmpty()) {
        currentInstrumentIndex += direction

        if (currentInstrumentIndex < 0) {
            currentInstrumentIndex = filteredInstr
        } else if (currentInstrumentIndex >= filte
            currentInstrumentIndex = 0  //Roll ba
        }

        displayInstrument(currentInstrumentIndex)
    }
}
```

Figure 30 - The navigateInstrument() function

The btnNext and btnPrev buttons allow users to navigate through the filtered list of instruments, leveraging the navigateInstrument() function to update the displayed instrument. When a user clicks btnNext, navigateInstrument(1) is called, incrementing the currentInstrumentIndex variable and logging the instrument name for debugging purposes. Similarly, clicking btnPrev calls navigateInstrument(-1), decrementing the currentInstrumentIndex variable while logging the corresponding instrument name. The navigateInstrument() function ensures smooth navigation by wrapping around the list. If the user goes beyond the last instrument, it loops back to the first, and if they go before the first, it cycles to the last. This circular navigation prevents out-of-bounds errors while maintaining an intuitive browsing experience. The function also calls

displayInstrument() each time the index is updated, ensuring the UI reflects the next or previous instrument in the filteredInstruments list.

5.5. Borrow button to pass intents to BookingActivity

5.5.1. MainActivity

```
findViewById<Button>(R.id.btnBorrow).setOnClickListener {
    if (filteredInstruments[currentInstrumentIndex].stockNumber < 1) {
        Toast.makeText(this, getString(R.string.out_of_instrument_message), Toast.LENGTH_SHORT).show()
        //Make sure the user can't rent if there isn't any instruments of that type anymore
        return@setOnClickListener
    }

    Log.i("BORROW", "Instrument: ${filteredInstruments[currentInstrumentIndex].name}")
    Log.i("BORROW", "Language - English: $isEnglish")
    Log.i("BORROW", "Credit Account: $currentCreditAccountBalance")
    val intent = Intent(this, BookingActivity::class.java).apply {
        putExtra("LANGUAGE", isEnglish)
        putExtra("selectedInstrument", filteredInstruments[currentInstrumentIndex])
        putExtra("currentCreditAccountBalance", currentCreditAccountBalance)
    }

    rentalActivityLauncher.launch(intent)
}
```

Figure 31 - btnConfirm in MainActivity pass intents to BookingActivity

Similar to the other 2 buttons, the setOnClickListener is also called for btnBorrow to handle the click event (*Buttons*, n.d.). To start with, it checks if the selected instrument, which would be the instrument at the currentInstrumentIndex index of the filteredInstruments list, has a value of the stockNumber greater than 1, because the user obviously can't rent an instrument unavailable at the music studio. Therefore, if the stockNumber value is 0, the app will just throw a toast with the message "Out of instruments to rent right now" or "Không có nhạc cụ để thuê lúc này" depending on the user's language preference, and exit out of the event handler early.

If the instrument is available, the function creates an explicit intent to transition to BookingActivity, carrying three key pieces of data: the boolean with the key "LANGUAGE" indicating if the current language preference is English or not, the Instrument object with the key "selectedInstrument" for the chosen instrument to rent, and the float currentCreditAccountBalance with the key "currentCreditAccountBalance" for the remaining amount of credit of the user (Intents and Intent Filters | Android Developers, 2019).

5.5.2. BookingActivity

```
// Restore language preference before setting content view
savedInstanceState?.let {
    isEnglish = it.getBoolean("LANGUAGE", true)
} ?: run {
    isEnglish = intent.getBooleanExtra("LANGUAGE", true)
}
```

```
// Retrieve the selected instrument
savedInstanceState?.let {
    selectedInstrument = it.getParcelable("selectedInstrument")
    currentCreditAccountBalance = it.getFloat("currentCreditAccountBalance", 0.0f)

    // Restore rentalEmail, rentalTime, and rentalNumber
    rentalEmail.setText(it.getString("rentalEmail", ""))
    rentalTimeSlider.value = it.getFloat("rentalTime", 1f)
    rentalNumberSlider.value = it.getFloat("rentalNumber", 1f)

    labelRentalNumber.text = getString(R.string.label_rental_number) + ": " + rentalNumberSlider.value.toInt()
    labelRentalTime.text = getString(R.string.label_rental_time) + ": " + rentalTimeSlider.value.toInt()
    updateRentalPrice()
} ?: run {
    selectedInstrument = intent.getParcelableExtra("selectedInstrument")
    currentCreditAccountBalance = intent.getFloatExtra("currentCreditAccountBalance", 0.0f)
}
```

Figure 32 - Code to handle intents passed from MainActivity in BookingActivity

In BookingActivity, for reasons explained later in the report, I had two different sections to restore the savedInstanceState. If the savedInstanceState is null, meaning this is likely the moment the user goes from MainActivity to the BookingActivity, the data embedded in the intent is extracted with getFloatExtra(), getParcelableExtra(), getBooleanExtra() methods and assigned to the variables in BookingActivity.

```
selectedInstrument?.let { instrument ->
    findViewById<TextView>(R.id.instrumentName).text = instrument.name
    findViewById<ImageView>(R.id.instrumentPicture).setImageResource(getInstrumentImage(instrument.name))

    //Specically to deal with the case the stock number of instrument is 1
    if (instrument.stockNumber == 1) {
        rentalNumberSlider.visibility = View.GONE
        rentalNumberSlider.value = 1f
        labelRentalNumber.text = getString(R.string.label_rental_number) + ": 1"
    } else {
        rentalNumberSlider.visibility = View.VISIBLE
        rentalNumberSlider.valueTo = instrument.stockNumber.toFloat()
        rentalNumberSlider.value = 1f
    }
}
```

Figure 33 - Code to retrieve selected instrument's information

```
private fun getInstrumentImage(instrumentName: String): Int {
    return when (instrumentName) {
        "Fender American Professional II Stratocaster" -> R.drawable.guitar_picture_1
        "Gibson Les Paul Standard '50s" -> R.drawable.guitar_picture_2
        "Steinway & Sons Model D Concert Grand" -> R.drawable.piano_picture_1
        "Yamaha FG800 Acoustic" -> R.drawable.guitar_picture_3
        "Yamaha P-125 Digital" -> R.drawable.piano_picture_2
        else -> R.drawable.saxophone_picture_1
    }
}
```

Figure 34 - getInstrumentImage() function

The selectedInstrument?.let { instrument -> … } code block first updates the UI by setting the instrument's name in a TextView and dynamically assigning the appropriate image using the getInstrumentImage() function. This function, similar to displayInstrument() function back in MainActivity, maps specific instrument names to their corresponding drawable resources, ensuring that each instrument is visually represented accurately.

A key feature of this implementation is how it handles the stock number of the instrument, particularly when only one unit remains. If the instrument's stock is exactly 1, the rentalNumber slider is hidden (rentalNumberSlider.visibility = View.GONE), its value is fixed at 1f, and the corresponding label labelRentalNumber is updated to reflect this. This prevents the user from selecting a rental quantity that exceeds the available stock. If more than one unit is available, the slider remains visible, its maximum value (valueTo) is set to the stock number, and it defaults to 1f to ensure a minimum rental of one unit.

Finally, the creditAccount TextView component in BookingActivity also updates its text with the value of the currentCreditAccountBalance variable.

5.6. User Input

```
// **Set initial values for sliders and rental price**
rentalNumberSlider.value = 1f
rentalTimeSlider.value = 1f
labelRentalNumber.text = getString(R.string.label_rental_number) + ": 1"
labelRentalTime.text = getString(R.string.label_rental_time) + ": 1"
updateRentalPrice() // Call function to calculate initial price
```

Figure 35 - Code to set default values for inputs

In the onCreate() function of BookingActivity, this block of code is used to set the default values for the rentalNumberSlider and the rentalTimeSlider sliders, and the 2 corresponding TextView, labelRentalNumber and labelRentalTime, so it is mainly used for when the user transitions from MainActivity to BookingActivity. The updateRentalPrice() function is also called here to display the default value of rentalPrice.

```
private fun updateRentalPrice() {
    selectedInstrument?.let { instrument ->
        rentalPrice.text = getString(R.string.label_rental_price) + " $" + calculateTotalPrice()
    }
}
```

Figure 36 - updateRentalPrice() function

```
rentalNumberSlider.addOnChangeListener { _, value, _ ->
    labelRentalNumber.text = getString(R.string.label_rental_number) + ": " + value.toInt()
    updateRentalPrice()
}

rentalTimeSlider.addOnChangeListener { _, value, _ ->
    labelRentalTime.text = getString(R.string.label_rental_time) + ": " + value.toInt()
    updateRentalPrice()
}
```

Figure 37 - The listeners for rentalNumberSlider and rentalTimeSlider sliders

The rentalNumberSlider listens for changes and updates the labelRentalNumber text to reflect the selected quantity, ensuring the displayed value is always an integer. Similarly, the rentalTimeSlider updates the labelRentalTime text to show the chosen rental duration. Both sliders also trigger updateRentalPrice(), to update the rentalPrice at the bottom.

```
private fun calculateTotalPrice(): Float {
    selectedInstrument?.let { instrument ->
        val rentalTime = rentalTimeSlider.value.toInt()
        val rentalNumber = rentalNumberSlider.value.toInt()
        return instrument.price * rentalTime * rentalNumber
    }

    return 0f // Return 0 only if no instrument is selected
}
```

Figure 38 - calculateTotalPrice() function

In the calculateTotalPrice() function, the selectedInstrument is first checked if it is null to return 0f as a safety measure. If it is not, the function returns a value calculated by multiplying the price property of the selectedInstrument, the value of rentalTimeSlider slider, and the value of rentalNumberSlider slider.

5.7. Confirm and Cancel buttons

```
btnConfirm.setOnClickListener {
    val email = rentalEmail.text.toString().trim()
    if (email == "") {
        Toast.makeText(this, getString(R.string.empty_email_message), Toast.LENGTH_SHORT).show()
    }
    else if (!Patterns.EMAIL_ADDRESS.matcher(email).matches()) {
        Toast.makeText(this, getString(R.string.invalid_email_message), Toast.LENGTH_SHORT).show()
    } else if (calculateTotalPrice() > currentCreditAccountBalance) {
        Toast.makeText(this, getString(R.string.insufficient_balance_message), Toast.LENGTH_SHORT).show()
    } else {
        // Proceed with booking logic
        currentCreditAccountBalance -= calculateTotalPrice()

        val resultIntent = Intent().apply {
            putExtra("selectedInstrument", selectedInstrument)
            putExtra("rentalEmail", email)
            putExtra("rentalNumber", rentalNumberSlider.value.toInt())
            putExtra("rentalTime", rentalTimeSlider.value.toInt())
            putExtra("rentalPrice", calculateTotalPrice())
            putExtra("currentCreditAccountBalance", currentCreditAccountBalance)

            putExtra("isEnglish", isEnglish)
        }

        setResult(Activity.RESULT_OK, resultIntent)
        finish()
    }
```

Figure 39 - Listener for btnConfirm

5.7.1. Input validation

When btnConfirm is clicked, it first retrieves the email input from rentalEmail, trims any extra spaces with the trim method, and checks if it is blank. If it indeed is, a Toast message appears, notifying the user to enter an email. Next, it verifies whether the input is a valid email format using Patterns.EMAIL_ADDRESS.matcher(email).matches(), displaying another Toast message if the format is incorrect (Julyus, 2016). The third validation step ensures the user has sufficient credits to cover the rental cost. If calculateTotalPrice() exceeds currentCreditAccountBalance, an insufficient balance message is shown. Only when all three conditions are met does the booking process proceed, ensuring that incomplete or invalid submissions are prevented, thereby maintaining a smooth and error-free rental experience.

5.7.2. Intent result from Confirm button

First, currentCreditAccountBalance is updated by subtracting the rentalPrice. Then, an Intent called resultIntent is created to store key rental details, including the selectedInstrument, the rentalEmail, rentalNumber, rentalTime, and rentalPrice with similarly named keys. Additionally, the updated currentCreditAccountBalance is included with the similarly named key to reflect the new balance after the transaction, and the isEnglish boolean value with the similarly named key is passed to maintain language settings. The function then sets the result of the activity with Activity.RESULT_OK, signaling that the booking was successful, and calls finish() to close the activity and return the result to the previous activity (*Solutions: Multiples (core)*, 2025).

5.7.3. Intent result from Cancel button

```
btnCancel.setOnClickListener {
    val resultIntent = Intent().apply {
        putExtra("isEnglish", isEnglish)
    }

    setResult(RESULT_CANCELED, resultIntent)
    finish()
}
```

Figure 40 - Listener for btnCancel

If the user clicks the btnCancel button, then the only data passed back to MainActivity is the isEnglish to keep language preference across different activities. The resultCode in setResult() method is set to RESULT_CANCELED, indicating the rental attempt has been canceled by the user before the finish() method is called to return to MainActivity (*Solutions: Multiples (core)*, 2025).

```
override fun onBackPressed() {
    // Trigger the Cancel button click logic
    btnCancel.performClick()

    super.onBackPressed()
}
```

Figure 41 - onBackPress() function

To fulfil a requirement of the assignment (pressing back should be considered cancellation), I overrode the onBackPressed() implementation by trigger a click event of the btnCancel with “btnCancel.performClick()” code line (Nick, 2011).

5.8. Data class RentalInfo

```
data class RentalInfo (
    val instrument: Instrument,
    val email: String,
    val number: Int,
    val time: Int,
    val price: Float
)
```

Figure 42 - RentalInfo data class

I also employed a second data class called RentalInfo to store all the details of a rental attempt. This class has 5 properties: instrument of Instrument type, email of string type, number of Int type, time of Int type, and price of Float type. This is used in MainActivity.kt with the code line “ private val rentalHistory = mutableListOf<RentalInfo>() ” to record all the rental instances into an unchangeable list.

5.9. Handle intent result from BookingActivity to MainActivity

5.9.1. RESULT_OK

```
rentalActivityLauncher = registerForActivityResult(ActivityResultContracts.StartActivityForResult()) {
    result ->
    if (result.resultCode == RESULT_OK) {
        result.data?.let {
            val selectedInstrument = it.getParcelableExtra<Instrument>("selectedInstrument") as Instrument
            val rentalEmail = it.getStringExtra("rentalEmail") ?: ""
            val rentalNumber = it.getIntExtra("rentalNumber", 0)
            val rentalTime = it.getIntExtra("rentalTime", 0)
            val rentalPrice = it.getFloatExtra("rentalPrice", 0.0f)

            //Should delete this part if it mess too much with other parts of the code
            val languagePreferenceFromRent = it.getBooleanExtra("isEnglish", isEnglish)
            if (languagePreferenceFromRent != isEnglish) {
                isEnglish = languagePreferenceFromRent
                setLocale(if (isEnglish) "en_US" else "vi")
                recreate() // Restart activity to apply language changes
            }

            currentCreditAccountBalance = it.getFloatExtra("currentCreditAccountBalance", currentCreditAccountBalance)
            creditAccount.text = "$currentCreditAccountBalance"

            // Add to rental history list
            val rentalAttempt = RentalInfo(selectedInstrument, rentalEmail, rentalNumber, rentalTime, rentalPrice)
            rentalHistory.add(rentalAttempt)
            updateBtnDetailsState()
            Log.i("RENTAL_HISTORY", "New rental attempt: ${selectedInstrument.name}")

            instrumentsList.find { it.name == filteredInstruments[currentInstrumentIndex].name }?.let {
                instrument -> instrument.stockNumber -= rentalNumber
                filteredInstruments[currentInstrumentIndex].stockNumber = instrument.stockNumber //change the stock number of displayed instruments as well
                Log.i("Reduce", "New stock number for default ${instrument.name}: ${instrument.stockNumber}")
                Log.i("Reduce", "New stock number for filtered ${filteredInstruments[currentInstrumentIndex].name}: ${filteredInstruments[currentInstrumentIndex].stockNumber}")
            } //Reduce the number of instruments in stock accordingly
            displayInstrument(currentInstrumentIndex)

            Toast.makeText(this, getString(R.string.rental_confirmation_message), Toast.LENGTH_SHORT).show()
```

Gradle build finished in 1 m 7 s 941 ms

Figure 43 - Code block to handle RESULT_OK

This part of the code defines and registers an ActivityResultLauncher called rentalActivityLauncher. It uses ActivityResultContracts.StartActivityForResult(), which allows MainActivity to receive data back from the BookingActivity once it finishes (*Solutions: Multiples (core)*, 2025).

When the BookingActivity returns a result, the lambda function checks if the resultCode is RESULT_OK. If so, the result.data bundle is accessed to retrieve the rental details. The retrieved data includes the selectedInstrument (as a Parcelable object), rentalEmail, rentalNumber, rentalTime, and rentalPrice. These values are extracted from the intent using getParcelableExtra and getXExtra methods, ensuring default values in case the data is missing.

If the rental activity has modified the language preference (through isEnglish), the app checks whether the new preference differs from the current one. If so, it updates the isEnglish flag, calls setLocale() to apply the new language ("en_US" for English or "vi" for Vietnamese), and restarts the activity using recreate() to ensure the UI updates accordingly.

The code then updates the credit account balance of the user (currentCreditAccountBalance) by retrieving it from the intent and updating the creditAccount UI element with the new value of currentCreditAccountBalance variable. Afterward, a new RentalInfo object is created using the retrieved rental details and added to the rentalHistory mutable list, which stores past rental

transactions. The updateBtnDetailsState() function is called, enabling the btnInstrumentBorrow as the rentalHistory list now has at least one element to show.
Stock management follows next. The code finds the rented instrument in the instrumentsList based on the currently displayed instrument's name. Once located, it reduces the stock number by the rented amount and ensures that the filteredInstruments list reflects the updated stock. The changes are logged using Log.i() for debugging purposes.
Finally, the UI is refreshed to display the updated instrument details using displayInstrument(currentInstrumentIndex function, and a confirmation Toast message is shown to inform the user that the rental was successful. This ensures a smooth user experience by providing immediate feedback on the transaction.

5.9.2. RESULT_CANCELED

```
} else if (result.resultCode == RESULT_CANCELED) {
    result.data?.let {
        val languagePreferenceFromRent = it.getBooleanExtra("isEnglish", isEnglish)
        if (languagePreferenceFromRent != isEnglish) {
            isEnglish = languagePreferenceFromRent
            setLocale(if (isEnglish) "en_US" else "vi")
            recreate() // Restart activity to apply language changes
        }

        Toast.makeText(this, getString(R.string.cancel_message), Toast.LENGTH_SHORT).show()
    }
```

Figure 44 - Code block to handle RESULT_CANCELED

If the resultCode is RESULT_CANCELED, then the MainActivity just retrieved isEnglish boolean value from the data bundle to change the language of the app if necessary. The app will also throw a toast to inform the user that he/she has canceled the rental attempt in BookingActivity.

5.10.Details Dialog

```
private fun showRentalHistoryDialog() {
    val builder = AlertDialog.Builder(this)
    builder.setTitle(getString(R.string.label_btn_instrument_borrow))

    if (rentalHistory.isNotEmpty()) {
        val historyDetails = rentalHistory.mapIndexed { index, rental ->
            "+ ${getString(R.string.label_rental_id)}: ${index + 1}\n" +  // Add the index (st
                    "+ ${getString(R.string.label_text_name)}: ${rental.instrument.name}\n" +
                    "+ ${getString(R.string.label_rental_email)} ${rental.email}\n" +
                    "+ ${getString(R.string.label_rental_number)}: ${rental.number}\n" +
                    "+ ${getString(R.string.label_rental_time)}: ${rental.time}\n" +
                    "+ ${getString(R.string.label_rental_price)}: $${rental.price}"
        }.joinToString("\n\n")

        builder.setMessage(historyDetails)
    }

    builder.setPositiveButton("Ok") { dialog, _ ->
        dialog.dismiss()
    }
```

Figure 45 - showRentalHistoryDialog() function

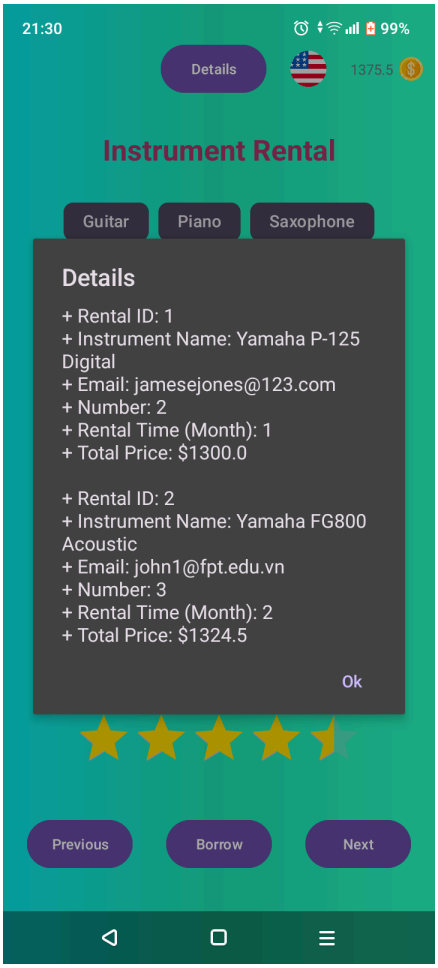

Figure 46 - Details dialog example

The showRentalHistoryDialog() function is responsible for displaying a dialog that presents the user's rental history. It uses an AlertDialog.Builder to construct a pop-up window, setting its title with getString(R.string.label_btn_instrument_borrow), utilizing multilingual support (Dialogs, 2024)

Within the function, the app first checks whether the rentalHistory list contains any past rentals. If there are recorded rentals, the function constructs a detailed rental history string. It iterates through the rentalHistory list using mapIndexed, which assigns an index to each rental entry, making it easier for the user to track their rental attempts. The details of each rental, such as the instrument name, renter's email, rental quantity, duration, and price, are formatted into a structured string, ensuring clarity and readability. Once the rental history string is constructed, it is set as the message of the dialog.

There will also be an "Ok" button at the end of the dialog so the users can dismiss it if they want, as I used the setPositiveButton() to execute the command dialog.dismiss() (Dialogs, 2024). Finally, the dialog is immediately shown to the user with the show() method of the AlertDialog class.

5.11. Multilingual Support

```xml
<resources>
    <string name="app_name">Instrument Rental</string>

    <string name="label_btn_instrument_borrow">Details</string>
    <string name="label_btn_prev">Previous</string>
    <string name="label_btn_borrow">Borrow</string>
    <string name="label_btn_next">Next</string>

    <string name="image_content_description">Instrument Image</string>
    <string name="btn_change_language_content_description">Country Icon</string>
    <string name="image_coin_content_description">Coin icons created by Freepik - Flaticon</string>

    <string name="guitar">Guitar</string>
    <string name="piano">Piano</string>
    <string name="saxophone">Saxophone</string>

    <string name="label_text_name">Instrument Name</string>
    <string name="label_text_type">Type</string>
    <string name="label_text_price">Monthly Price</string>
    <string name="label_text_stock_number">Stock Number</string>
    <string name="label_text_weight">Weight</string>

    <string name="label_btn_confirm">Confirm</string>
    <string name="label_btn_cancel">Cancel</string>

    <string name="label_rental_email">Email:</string>
    <string name="label_rental_number">Number</string>
    <string name="label_rental_time">Rental Time (Month)</string>
```

Figure 47 - strings.xml file in values folder

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Thuê Nhạc Cụ</string>

    <string name="label_btn_instrument_borrow">Chi Tiết</string>
    <string name="label_btn_prev">Trước</string>
    <string name="label_btn_borrow">Mượn</string>
    <string name="label_btn_next">Sau</string>

    <string name="image_content_description">Hình Ảnh Nhạc Cụ</string>
    <string name="btn_change_language_content_description">Biểu Tượng Quốc Gia</string>
    <string name="image_coin_content_description">Biểu tượng xu tạo bởi Freepik - Flaticon</string>

    <string name="guitar">Đàn Guitar</string>
    <string name="piano">Dương Cầm</string>
    <string name="saxophone">Kèn Saxophone</string>

    <string name="label_text_name">Tên Nhạc Cụ</string>
    <string name="label_text_type">Thể Loại</string>
    <string name="label_text_price">Giá Tháng</string>
    <string name="label_text_stock_number">Tồn Kho</string>
    <string name="label_text_weight">Khối Lượng</string>

    <string name="label_btn_confirm">Xác Nhận</string>
    <string name="label_btn_cancel">Hủy</string>

    <string name="label_rental_email">Email:</string>
    <string name="label_rental_number">Số Lượng</string>
```

Figure 48 - strings.xml file in values-vi folder

The 2 figures above are the screenshots of the 2 strings.xml file I used to translate all the text between 2 languages. Both files must have <string> elements with the same name attribute to use this in MainActivity, BookingActivity code, or activity_main, or activity_booking xml layouts.

```
btnLanguageToggle.setOnClickListener {
    isEnglish = !isEnglish
    setLocale(if (isEnglish) "en_US" else "vi")
    recreate() // Restart activity to apply changes
}
```

Figure 49 - Listener for btnChangeLang

When the button is clicked, it toggles the isEnglish boolean value (which determines the current language) by negating its current value. It then calls the setLocale function, passing either "en_US" (for English) or "vi" (for Vietnamese), depending on the value of isEnglish. After

updating the locale, the activity is restarted using recreate() to apply the language change immediately.

```
private fun setLocale(languageCode: String) {
    val locale = Locale(languageCode)
    Locale.setDefault(locale)
    val config = Configuration()
    config.setLocale(locale)
    resources.updateConfiguration(config, resources.displayMetrics)
}
```

Figure 50 - setLocale() function

This function sets the locale of the application based on the provided languageCode (e.g., "en_US" for English or "vi" for Vietnamese). It creates a Locale object using the given language code and sets it as the default locale. A Configuration object is created, and its locale is set to the new locale. Finally, the updateConfiguration method is called on the app's resources to apply the new locale settings

```
private fun updateButtonChangeLanguageIcon() {
    val iconRes = if (isEnglish) R.drawable.flag_us else R.drawable.flag_vietnam
    btnLanguageToggle.setImageResource(iconRes)
}
```

Figure 51 - updateButtonChangeLanguageIcon() function

TheupdateButtonChangeLanguageIcon() function updates the icon on the btnLanguageToggle button based on the current language preference. If isEnglish is true, it sets the button's icon to the US flag icon (flag_us icon image in drawable folder). Otherwise, it sets the icon to the Vietnamese flag (flag_vietnam icon image in drawable folder).

```
//Return English or Vietnamese version of instrument type based on language preference
private fun getLocalizedType(type: InstrumentType): String {
    return when (type) {
        InstrumentType.GUITAR -> getString(R.string.guitar)
        InstrumentType.PIANO -> getString(R.string.piano)
        InstrumentType.SAXOPHONE -> getString(R.string.saxophone)
    }
}
```

Figure 52 - updateButtonChangeLanguageIcon() function

This function is used to return the localized string for an instrument type, based on the current language preference. It takes an InstrumentType (like InstrumentType.GUITAR,

InstrumentType.PIANO, or InstrumentType.SAXOPHONE) as a parameter. Depending on the value of the InstrumentType, it fetches the corresponding localized string using getString(R.string.instrument_name), where R.string.guitar, R.string.piano, and R.string.saxophone are predefined string resources for each instrument type in the app's resource files.

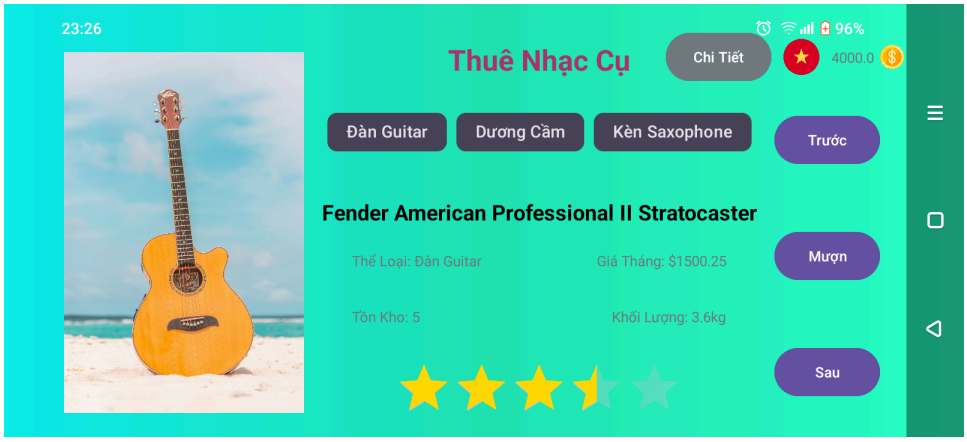

Figure 53 - MainActivity screen in Vietnamese

Figure 54 - BookingActivity screen in Vietnamese

5.12.savedInstanceState

```
//Save instance through bundle
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putBoolean("LANGUAGE", isEnglish) //save language preference in case of screen orientation
    Log.i("BUNDLE", "saveInstanceState: languageEnglish - $isEnglish")

    outState.putBoolean("guitarChip", findViewById<Chip>(R.id.chipGuitar).isChecked)
    outState.putBoolean("pianoChip", findViewById<Chip>(R.id.chipPiano).isChecked)
    outState.putBoolean("saxophoneChip", findViewById<Chip>(R.id.chipSaxophone).isChecked)
    Log.i("BUNDLE", "saveInstanceState: guitarChip - ${findViewById<Chip>(R.id.chipGuitar).isChecked}; pianoChip - ${findViewById<Chip>(R.id.chipPiano).isChecked}; saxophoneChip -
     ${findViewById<Chip>(R.id.chipSaxophone).isChecked}")

    outState.putInt("currentInstrumentIndex", currentInstrumentIndex)
    Log.i("BUNDLE", "saveInstanceState: currentInstrumentIndex - $currentInstrumentIndex")

    // Convert the filteredInstruments list to a JSON string
    val gson = Gson()
    val instrumentsJson = gson.toJson(filteredInstruments)
    outState.putString("filteredInstrumentsJson", instrumentsJson)
    Log.i("BUNDLE", "saveInstanceState: instrumentsJson - $instrumentsJson")

    val instrumentsListJson = gson.toJson(instrumentsList)
    outState.putString("instrumentsListJson", instrumentsListJson)
    Log.i("BUNDLE", "saveInstanceState: instrumentsListJson - $instrumentsListJson")

    outState.putFloat("currentCreditAccountBalance", currentCreditAccountBalance)
    Log.i("BUNDLE", "saveInstanceState: currentCreditAccountBalance - $currentCreditAccountBalance")

    val rentalHistoryJson = Gson().toJson(rentalHistory)
    outState.putString("rentalHistoryJson", rentalHistoryJson)
    Log.i("BUNDLE", "saveInstanceState: rentalHistory - $rentalHistoryJson")
}
```

Figure 55 - onSaveInstanceState function

The onSaveInstanceState() method is overridden to save important state information before the activity is destroyed (e.g., on orientation change). The method first saves the current language preference (isEnglish) in the Bundle. It then saves the checked states of the chips (guitarChip, pianoChip, saxophoneChip) by calling findViewById on each chip and using the isChecked property. The current instrument index (currentInstrumentIndex) is saved as well.
The method also serializes the filteredInstruments and instrumentsList lists into JSON strings using Gson and saves them in the Bundle. The currentCreditAccountBalance is saved, and the rentalHistory list is converted into a JSON string and stored as well. The log statements track the saved data for debugging purposes

.

```
//MUST restore instance state first before setContentView to force device to change language preference
savedInstanceState?.let {
    isEnglish = it.getBoolean("LANGUAGE") // Default to English if null
    currentCreditAccountBalance = it.getFloat("currentCreditAccountBalance", 4000.0f)
}
```

Figure 56 - restore savedInstanceState part 1

This block of code restores the instance state from the saved Bundle. It checks if the savedInstanceState is not null and retrieves the saved values for the language preference (isEnglish) and the current credit account balance (currentCreditAccountBalance). If savedInstanceState exists, the language preference is restored (defaulting to true for English if the saved value is null), and the current credit account balance is retrieved, with a default value of 4000.0f if it's not found. It is critical for this code block to be executed before the line “

setContentView(R.layout.activity_main) ” to force the device to set language preference first before rendering any UI elements. That’s the reason why I split the restoration of savedInstanceState into 2 different places

```
savedInstanceState?.let {
    val guitarChecked = it.getBoolean("guitarChip")
    val pianoChecked = it.getBoolean("pianoChip")
    val saxophoneChecked = it.getBoolean("saxophoneChip")

    // Restore the chip selections
    findViewById<Chip>(R.id.chipGuitar).isChecked = guitarChecked
    findViewById<Chip>(R.id.chipPiano).isChecked = pianoChecked
    findViewById<Chip>(R.id.chipSaxophone).isChecked = saxophoneChecked

    // Restore the current instrument index
    currentInstrumentIndex = it.getInt("currentInstrumentIndex")

    val gson = Gson()
    val type = object : TypeToken<List<Instrument>>() {}.type
    val instrumentsListJson = it.getString("instrumentsListJson")
    instrumentsList = instrumentsListJson?.let { json -> gson.fromJson(json, type) } ?: instrumentsList

    val filteredInstrumentsJson = it.getString("filteredInstrumentsJson")
    filteredInstruments = filteredInstrumentsJson?.let { json -> gson.fromJson(json, type) } ?: instrumentsList

      filterInstruments() //can't because it will reset currentInstrumentIndex
    displayInstrument(currentInstrumentIndex)

    //Nnecessary, can't delete this part
    val rentalHistoryJson = savedInstanceState.getString("rentalHistoryJson")
    if (!rentalHistoryJson.isNullOrEmpty()) {
        val type = object : TypeToken<List<RentalInfo>>() {}.type
        rentalHistory.clear()
        rentalHistory.addAll(Gson().fromJson(rentalHistoryJson, type))
        Log.i("BUNDLE", "restoreInstanceState: rentalHistory - $rentalHistoryJson")
    }
}
```

Figure 57 - restore savedInstanceState part 2

The second part retrieves the saved boolean values for each chip's checked state (guitarChecked, pianoChecked, saxophoneChecked) and updates the corresponding Chip views to reflect the correct state. It also restores the currentInstrumentIndex to remember which instrument was selected. The Gson library is used to deserialize the List<Instrument> objects from JSON format, such as instrumentsListJson and filteredInstrumentsJson.
Then, the app retrieves and deserializes the rentalHistoryJson string from the saved instance state. If this JSON is not null or empty, it is converted back into a List<RentalInfo> using Gson, and the rentalHistory list is updated. A log statement is included to print out the restored rental history for debugging purposes.

6. Espresso Testing

6.1. MainActivityTest1

```
@Test
fun mainActivityTest1() {
    //Part 1, where the activity is first loaded
    onView(allOf(withId(R.id.instrumentName))).check(matches(withText("Fender American Professional II Stratocaster")))

    onView(allOf(withId(R.id.instrumentStockNumber))).check(matches(withText("Stock Number: 5")))

    onView(allOf(withId(R.id.instrumentType))).check(matches(withText("Type: Guitar")))

    onView(allOf(withId(R.id.instrumentPrice))).check(matches(withText("Monthly Price: $1500.25")))

    onView(allOf(withId(R.id.instrumentWeight))).check(matches(withText("Weight: 3.6kg")))

    onView(allOf(withId(R.id.creditAccount))).check(matches(withText("4000.0")))

    onView(allOf(withId(R.id.btnInstrumentBorrow))).check(matches(not(isEnabled())))
    //End of part 1

    //Part 2, unselect the Guitar chip, then click Next twice
    onView(allOf(withId(R.id.chipGuitar))).perform(click())

    repeat(2) {
        onView(allOf(withId(R.id.btnNext))).perform(click())
    }

    onView(allOf(withId(R.id.instrumentName))).check(matches(withText("Selmer Paris Series II Tenor")))

    onView(allOf(withId(R.id.instrumentType))).check(matches(withText("Type: Saxophone")))

    onView(allOf(withId(R.id.instrumentPrice))).check(matches(withText("Monthly Price: $3900.5")))

    onView(allOf(withId(R.id.instrumentStockNumber))).check(matches(withText("Stock Number: 3")))

    onView(allOf(withId(R.id.instrumentWeight))).check(matches(withText("Weight: 3.2kg")))
    //End of part 2
```

Figure 58 - MainActivityTest1 part 1 and 2

In part 1 of the first Espresso test, I basically just checked the values of some UI components (instrumentName, instrumentStockNumber, instrumentType, instrumentPrice, instrumentWeight, and creditAccount) when the app started up. Most noticeably, the btnInstrumentBorrow must be disabled as there hadn't been any rental attempts yet.

Moving on to part 2, the test would unselect the chipGuitar chip, and click the btnNext button twice. The correct instrument to display on the screen at that moment would be Selmer Paris Series II Tenor, and a few next statements would make sure the values of relevant widgets like instrumentName, instrumentStockNumber, instrumentType, instrumentPrice, instrumentWeight match the correct information of Selmer Paris Series II Tenor.

```
//Part 3, change language
onView(allOf(withId(R.id.btnChangeLang))).perform(click())

onView(allOf(withId(R.id.chipGuitar))).check(matches(withText("Đàn Guitar")))

onView(allOf(withId(R.id.chipPiano))).check(matches(withText("Dương Cầm")))

onView(allOf(withId(R.id.chipSaxophone))).check(matches(withText("Kèn Saxophone")))

onView(allOf(withId(R.id.instrumentType))).check(matches(withText("Thể Loại: Kèn Saxophone")))

onView(allOf(withId(R.id.instrumentPrice))).check(matches(withText("Giá Tháng: $3900.5")))

onView(allOf(withId(R.id.instrumentStockNumber))).check(matches(withText("Tồn Kho: 3")))

onView(allOf(withId(R.id.instrumentWeight))).check(matches(withText("Khối Lượng: 3.2kg")))

onView(allOf(withId(R.id.creditAccount))).check(matches(withText("4000.0")))
//End of part 3

//Part 4, which only the saxophone is filtered, thus disabling Next and Previous buttons
onView(allOf(withId(R.id.chipPiano))).perform(click())

onView(allOf(withId(R.id.btnPrev))).check(matches((not(isEnabled()))))

onView(allOf(withId(R.id.btnNext))).check(matches((not(isEnabled()))))
//End of part 4
```

Figure 59 - MainActivityTest1 part 3 and 4

For part 3, the btnChangeLanguage is clicked to toggle the language to Vietnamese. Then, all of the widgets in part 1 (instrumentName, instrumentStockNumber, instrumentType, instrumentPrice, instrumentWeight, and creditAccount) are checked to see if their text has changed to Vietnamese properly. In addition, the text of the 3 chips (chipPiano, chipGuitar, chipSaxophone) are also checked.
And for part 4, the chipPiano chip is also clicked, thus unselecting it. Therefore, both the btnPrev and btnNext buttons should now be disabled as there is only one instrument filtered.

6.2. BookingActivityTest1

```
fun bookingActivityTest1() {
    //Part 1, first unselect the Saxophone chip, click the Prev button twice, then the Borrow butt
     Booking Activity
    onView(allOf(withId(R.id.chipSaxophone))).perform(click())

    repeat(2) {
        onView(allOf(withId(R.id.btnPrev))).perform(click())
    }

    onView(allOf(withId(R.id.btnBorrow))).perform(click())

    onView(allOf(withId(R.id.instrumentName))).check(matches(withText("Yamaha FG800 Acoustic")))

    onView(allOf(withId(R.id.labelRentalNumber))).check(matches(withText("Number: 1")))

    onView(allOf(withId(R.id.labelRentalTime))).check(matches(withText("Rental Time (Month): 1")))

    onView(allOf(withId(R.id.rentalPrice))).check(matches(withText("Total Price $220.75")))

    onView(allOf(withId(R.id.creditAccount))).check(matches(withText("4000.0")))

    onView(allOf(withId(R.id.appName))).check(matches(withText("Instrument Rental")))

    onView(allOf(withId(R.id.btnConfirm))).check(matches(withText("Confirm")))

    onView(allOf(withId(R.id.btnCancel))).check(matches(withText("Cancel")))
    //End of part 1
```

Figure 60 - BookingTestActivity1 part 1

The second Espresso test is mainly used for testing the UI of BookingActivity. For part 1, the chipSaxophone chip will be unselected, followed by the btnNext being clicked twice. The btnBorrow will then be clicked to go to the BorrowActivity. The selected instrument should be Yamaha FG800 Acoustic, and this part aims to check that and the default value for labelRentalTime, labelRentalNumber, and rentalPrice

```
//Part 2, change the language
onView(allOf(withId(R.id.btnChangeLang))).perform(click())

onView(allOf(withId(R.id.creditAccount))).check(matches(withText("4000.0")))

onView(allOf(withId(R.id.appName))).check(matches(withText("Thuê Nhạc Cụ")))

onView(allOf(withId(R.id.instrumentName))).check(matches(withText("Yamaha FG800 Acoustic")))

onView(allOf(withId(R.id.labelRentalNumber))).check(matches(withText("Số Lượng: 1")))

onView(allOf(withId(R.id.labelRentalTime))).check(matches(withText("Thời Gian Thuê (Tháng): 1")))

onView(allOf(withId(R.id.rentalPrice))).check(matches(withText("Tổng Tiền $220.75")))

onView(allOf(withId(R.id.btnConfirm))).check(matches(withText("Xác Nhận")))

onView(allOf(withId(R.id.btnCancel))).check(matches(withText("Hủy")))
//End of part 2

//Part 3, type in the email, drag both sliders with customized functions
onView(allOf(withId(R.id.rentalEmail))).perform(replaceText("johnjames1@gmail.com"), closeSoftKeyboard())

onView(allOf(withId(R.id.rentalEmail))).perform(pressImeActionButton())

onView(allOf(withId(R.id.rentalEmail))).check(matches(withText("johnjames1@gmail.com")))

onView(withId(R.id.rentalNumber)).perform(setValue(3.0F))

onView(withId(R.id.rentalNumber)).check(matches(withValue(3.0F)))

onView(allOf(withId(R.id.labelRentalNumber))).check(matches(withText("Số Lượng: 3")))

onView(withId(R.id.rentalTime)).perform(setValue(2.0F))

onView(withId(R.id.rentalTime)).check(matches(withValue(2.0F)))

onView(allOf(withId(R.id.labelRentalTime))).check(matches(withText("Thời Gian Thuê (Tháng): 2")))

onView(allOf(withId(R.id.rentalPrice))).check(matches(withText("Tổng Tiền $1324.5")))
```

Figure 61 - BookingTestActivity1 part 2 and part 3

In part 2, the language is toggled to Vietnamese with the click of the btnChangeLang. Then the test checks a variety of elements like appName, instrumentName, labelRentalNumber, labelRentalTime,etc. to see if they have been set to Vietnamese values.

For part 3, I must include some functions to set and check the values of sliders as Espresso doesn't support this natively. With the help of this (Loutremaline, 2020) webpage on StackOverflow, I was able to achieve this. First, the test inputs an email address ("johnjames1@gmail.com") into the rental email field and verifies that the text was entered correctly. The rentalNumber is changed to "3", and the rentalTime is set to "2 months", with

corresponding TextView labels, labelRentalNumber and labelRentalTime updating to reflect these changes. Finally, the rentalPrice's text should update dynamically based on these new values.

```
fun withValue(expectedValue: Float): Matcher<View?> {
    return object : BoundedMatcher<View?, Slider>(Slider::class.java) {
        override fun describeTo(description: Description) {
            description.appendText("expected: $expectedValue")
        }

        override fun matchesSafely(slider: Slider?): Boolean {
            return slider?.value == expectedValue
        }
    }
}

fun setValue(value: Float): ViewAction {
    return object : ViewAction {
        override fun getDescription(): String {
            return "Set Slider value to $value"
        }

        override fun getConstraints(): Matcher<View> {
            return ViewMatchers.isAssignableFrom(Slider::class.java)
        }

        override fun perform(uiController: UiController?, view: View) {
            val seekBar = view as Slider
            seekBar.value = value
        }
    }
}
```

Figure 62 - 2 functions for set and check values of sliders

6.3. MainActivityTest2

```
@Test
fun mainActivityTest2() {
    //Select an instrument to borrow, then check if the details in MainActivity matches
    repeat(2) {
        onView(allOf(withId(R.id.btnNext))).perform(click())
    }

    onView(allOf(withId(R.id.creditAccount))).check(matches(withText("4000.0")))

    onView(allOf(withId(R.id.btnInstrumentBorrow))).check(matches(not(isEnabled())))

    onView(allOf(withId(R.id.btnBorrow))).perform(click())

    onView(allOf(withId(R.id.rentalEmail))).perform(replaceText("johnjames1@gmail.com"), closeSoftKeyboard())

    onView(allOf(withId(R.id.rentalEmail))).perform(pressImeActionButton())

    onView(allOf(withId(R.id.instrumentName))).check(matches(withText("Steinway & Sons Model D Concert Grand")))

    onView(allOf(withId(R.id.rentalEmail))).check(matches(withText("johnjames1@gmail.com")))

    onView(withId(R.id.rentalNumber)).perform(setValue(2.0F))

    onView(withId(R.id.rentalNumber)).check(matches(withValue(2.0F)))

    onView(allOf(withId(R.id.labelRentalNumber))).check(matches(withText("Number: 2")))

    onView(withId(R.id.rentalTime)).perform(setValue(4.0F))
onView(allOf(withId(R.id.labelRentalTime))).check(matches(withText("Rental Time (Month): 4")))

onView(allOf(withId(R.id.rentalPrice))).check(matches(withText("Total Price $1364.0")))

onView(allOf(withId(R.id.btnConfirm))).perform(click())

onView(allOf(withId(R.id.instrumentName))).check(matches(withText("Steinway & Sons Model D Concert Grand")))

onView(allOf(withId(R.id.creditAccount))).check(matches(withText("2636.0")))

onView(allOf(withId(R.id.btnInstrumentBorrow))).check(matches(isEnabled()))

onView(allOf(withId(R.id.btnInstrumentBorrow))).perform(click())

onView(allOf(withId(android.R.id.message))).check(matches(withText("+ Rental ID: 1\n+ Instrument Name: Steinway & Sons Model D Concert
 Grand\n+ Email: johnjames1@gmail.com\n+ Number: 2\n+ Rental Time (Month): 4\n+ Total Price: $1364.0")))
```

Figure 63 - MainActivityTest2

The final Espresso test is used to rent one item successfully then checked if the rental details are displayed in the dialog in MainActivity. This test also used the 2 functions line BookingActivityTest1 to deal with the sliders. To start with, the "btnNext" button is clicked twice to navigate through the available instruments, and the credit account balance is checked to be "4000.0". Before any rental action, the "btnInstrumentBorrow" button must be disabled. Then,

the "btnBorrow" button is clicked to proceed to the booking screen, where the user inputs "johnjames1@gmail.com" as the rental email and presses the IME action button to confirm. The test verifies that the instrument displayed is "Steinway & Sons Model D Concert Grand" and that the entered email appears correctly. Next, the rentalNumber slider is set to "2", and the label updates to "Number: 2", while the rentalTime slider is adjusted to "4 months", with the corresponding label changing to "Rental Time (Month): 4". The total price should be updated to "$1364.0". After clicking "btnConfirm", the credit account balance should decrease to "2636.0", and the "btnInstrumentBorrow" button must now be enabled. Clicking this button then triggers a confirmation dialog displaying the rental details, including Rental ID, instrument name, email, number of instruments, rental time, and total price.

7. Reflection

7.1. What worked well

In this assignment, I feel like I have both reused a lot of the concepts from assignment 1 like logging, savedInstanceState and applied some new concepts learned in labs 5 and 6 very well. The UI/UX is definitely a major improvement compared to the first assignment as I used every bit of the screen for the app, leaving very little empty space. Furthermore, due to the increase in complexity of the app, I had to be more proficient in detecting and fixing all possible bugs in the app

7.2. What could be improved

I think my OnCreate functions in both MainActivity and BookingActivity should be shortened significantly as some parts of the code could be made into separate functions to be reused. In the future, one of the features I want to apply for my app can be the ability to change between two themes, light and dark, or even allow the user to customize his/her own theme, as the color of the UI components can vary wildly based on devices.

8. Github Repository Link

https://github.com/2025-HX01-COS30017-HCM/assignment-2-JJWilson-75

References

1. *Basic syntax | Kotlin*. (n.d.). Kotlin Help. https://kotlinlang.org/docs/basic-syntax.html#variables
2. *Build a Responsive UI with ConstraintLayout*. (n.d.). Android Developers. https://developer.android.com/develop/ui/views/layout/constraint-layout
3. *Buttons*. (n.d.). Android Developers. https://developer.android.com/develop/ui/views/components/button
4. *ChipGroup* (2019). *ChipGroup | Android Developers*. Android Developers. https://developer.android.com/reference/com/google/android/material/chip/ChipGroup
5. *Dialogs*. (2024). Android Developers. https://developer.android.com/develop/ui/views/components/dialogs#kotlin
6. *Intents and Intent Filters | Android Developers*. (2019). Android Developers. https://developer.android.com/guide/components/intents-filters
7. Julyus, M. (2016, March 16). *Email validation on EditText - Android*. Stack Overflow. https://stackoverflow.com/questions/36040154/email-validation-on-edittext-android
8. Loutremaline. (2020, December 21). *Androidx : How to test Slider in UI tests (Espresso)?* Stack Overflow. https://stackoverflow.com/questions/65390086/androidx-how-to-test-slider-in-ui-tests-espresso
9. Nguyen Huy Quyet. (2017, August 28). Serializable và Parcelable trong Android. Viblo. https://viblo.asia/p/serializable-va-parcelable-trong-android-GrLZDbe35k0
10. Nick. (2011, June 20). *Android: Proper Way to use onBackPressed() with Toast*. Stack Overflow. https://stackoverflow.com/questions/6413700/android-proper-way-to-use-onbackpressed-with-toast
11. *Orange Design System*. (2020). Orange.com. https://system.design.orange.com/0c1af118d/p/887440-toast-and-snackbars
12. Rana, A. (2020, July 20). *How to Customize RatingBar Component in Android*. Hackernoon.com. https://hackernoon.com/how-to-customize-ratingbar-component-in-android-gf1h3u0d
13. *RatingBar*. (n.d.). Android Developers. https://developer.android.com/reference/android/widget/RatingBar

14. *Sliders – Material Design 3*. (n.d.). Material Design. https://m3.material.io/components/sliders/overview
15. *Solutions: Multiples (core)*. (2025). Swinburne. https://swinburne.instructure.com/courses/64110/pages/solutions-multiples-core?module_item_id=4563943
16. *Styles and Themes*. (n.d.). Android Developers. https://developer.android.com/develop/ui/views/theming/themes