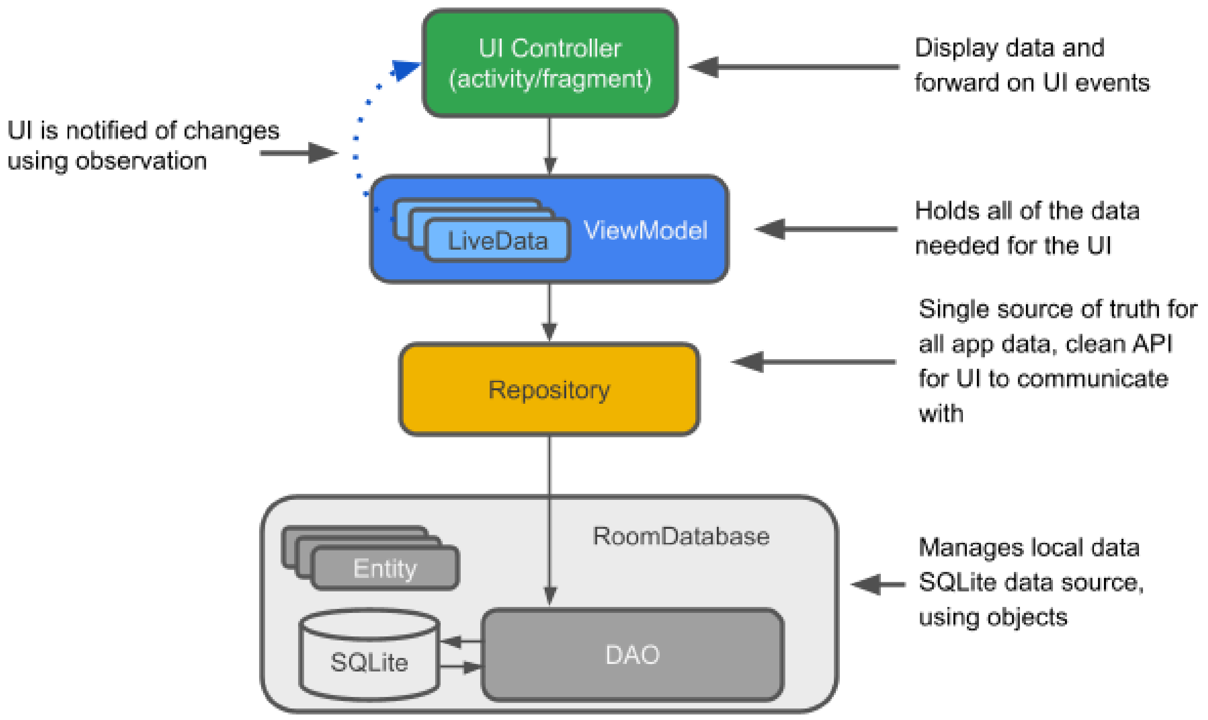
Name: Chhun Hong Lim

Student ID: 103527926

COS30017 Assignment 3

I. INTRODUCTION

For this assignment, we are given several options to create an app that would keep track of things that the user is interested in. In my case, I picked the option to create an app that would contain a list of books that I am interested in. While the idea of the app is different, all of them would follow the same concept as each other due to the requirements that have been specified in the task. For those requirements, it would include using RecycleView that has some complexity and the data of the app will be stored using either a file or a database such as Room or Firebase. Additionally, the app would need to include at least 3 novel activities that would have separate functionality from each other and the usage of view model, concurrency, and Live Data is expected in this assignment. For this assignment, I have decided to use code that is about the Room database from a tutorial that has been linked to us in previous weeks that would follow a similar concept as the requirements of this task and I have modified it to meet the requirements of the task. The link to the tutorial page and the GitHub Repository is provided in the reference section of the report. For my case, since I have picked the app that would list out the books that I am interested in, it would include 3 different activities in the app which would include an option to add a book in the list, an option to edit the name of the book, and an option to delete a specific book from the list. Those different activities can be accessed by using the buttons that are provided at the bottom of the screen and will communicate back to the main activity based on the activities the user has selected. To meet those given requirements, we would need to first understand the architecture of the components in the app. This architecture can be divided into four different sections which would include the Room database, the Repository, the ViewModel and Live Data, and the activities. For the Room database, it would contain the entity data class that would contain a string table used to stored the strings that are added, the DAO (data access object) which would allow use query methods that would allow use to interact with the database, and the SQLite database which is an on device storage. For the Repository, it is a class created to manage multiple data sources which would include functions for adding, deleting, and editing the strings. For the ViewModel and Live Data, it would hold all the information needed to communicate with the activities. More specifically, the ViewModel allows the Repository and the activities to communicate with each other by having it as the center of communication and the Live Data would always hold the latest version of the data and notifies the observer if there is any changes (Muntenescu, F. 2023). For the activities, as mentioned previously, there would be four activities in total in this app which would be the MainActivity, the NewWordActivity, the DeleteWordActivity, and the EditWordActivity. The MainActivity is where the user can see all the books that were added previously with three different buttons located at the bottom of the page so the user can choose which action they want to do next. Those three buttons would have each of their own activities which would be adding a new book into the list, delete a book that is in the list, and editing the book that is in the list respectively.


UI Controller
(activity/fragment)
Display data and forward on UI events
UI is notified of changes using observation
LiveData
ViewModel
Holds all of the data needed for the UI
Repository
Single source of truth for all app data, clean API for UI to communicate with
RoomDatabase
Entity
SQLite
DAO
Manages local data SQLite data source, using objects
Figure 1: The Architecture of the Components (Muntenescu, F. 2023)

II. USER STORIES

- The user wants to have a list app to store all the books that they are interested in. The app would be able to display the name of the books at the main page and the user would have the ability to add more books later on in the list if there is another book that they are interested in.

- The user finishes one of the books that they are interested in and wants the ability to remove that specific book from the list. In the app, this is possible by having a button located at the bottom of the screen with the bin icon that would bring up another activity for the user to select which book from a drop down list they would want to remove from the list.
- The user makes a mistake when typing the name of a book and would want the ability to edit the name in the app without deleting it and creating a new one. This can be achieved by the button at the bottom of the screen with the pencil icon which would allow the user to edit the name of the book.

III. DESIGN IMPLEMENTATION

The design that I am going for is a modified version of the tutorial code given by Muntenescu, F. which would have the RecycleView which is a list of books displayed with a orange background at the MainActivity with three buttons with their own respective icons at the bottom of the screen having their own functionality, which is adding a new book, editing a book, and deleting the book. The resulting activities can be seen in the figures in the table below.

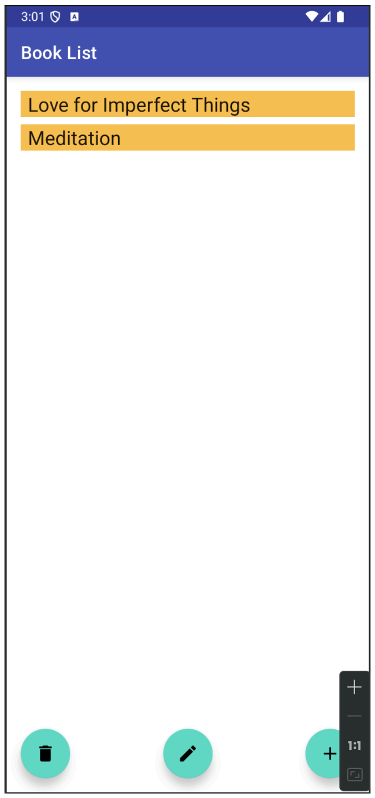


Figure 2: MainActivity

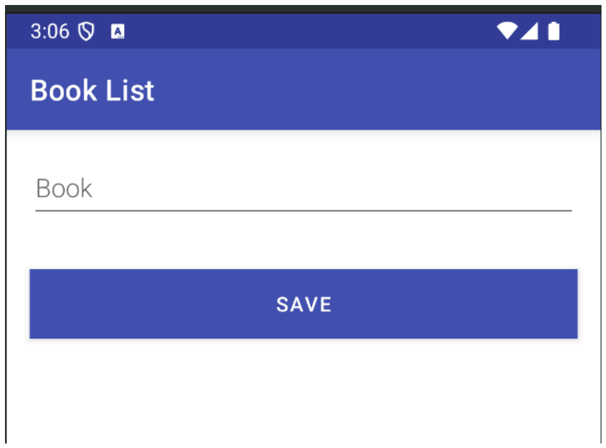


Figure 3: NewWordActivity

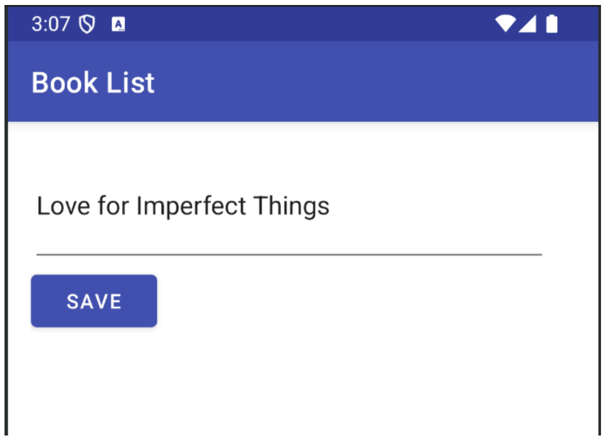


Figure 4: EditWordActivity

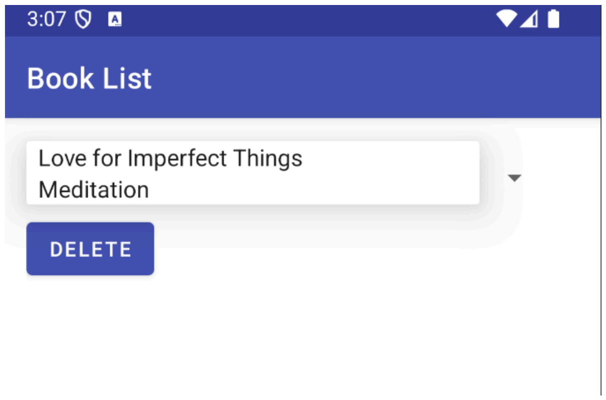
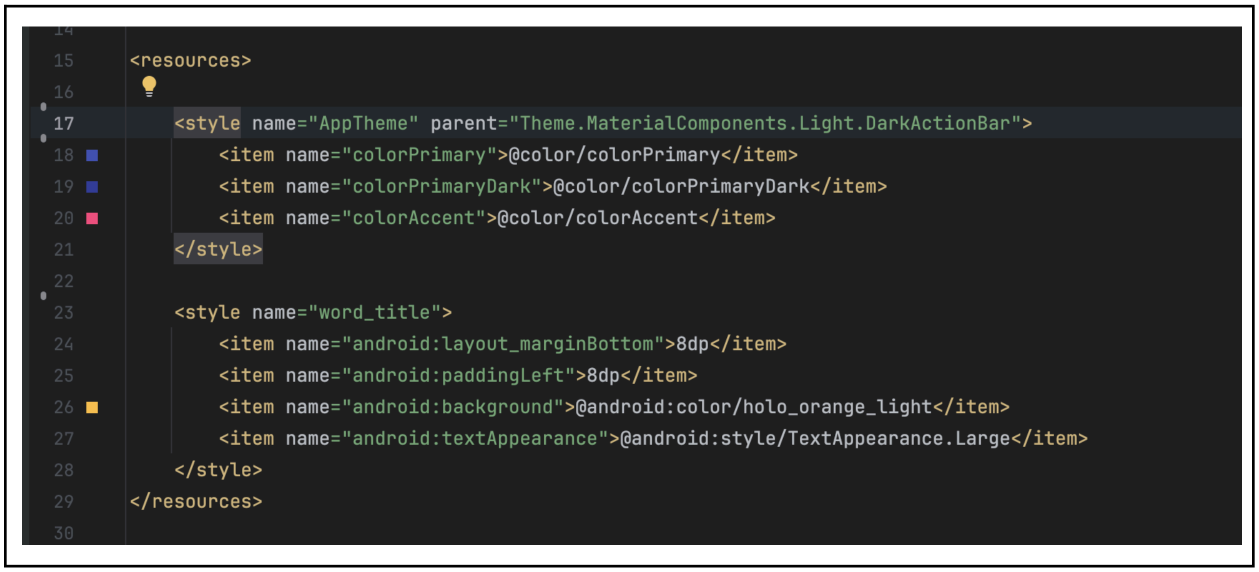


Figure 5: DeleteWordActivity

For the design implementation of the app, I would follow the requirements given by the task that states that there needed to be at least 3 activities or fragments present in the app. As seen from Figure 2, For the MainActivity, the design of the activity would be a simple main page that would consist of RecyclerView which would display as a list of books that the user added as well as the buttons used to navigate to different activities as well. The display of the book list will depend on how many books the user wants to add but from Figure 1, there are only two books that are added to the list. For three buttons that can be seen at the bottom of the screen, it would be used to navigate to the three other activities that were implemented. The button with the plus icon is for the NewWordActivity which is used when the user wants to add another book that they are interested in to the list. The button with the pencil icon is for the EditWordActivity which is used for when the user wants to edit a name of a book when there is a mistake on the name. The button with the bin icon is for the DeleteWordActivity which is used for when the user wants to delete a certain book from the RecycleView list. The design layout of each activity is similar to each other with it containing a button for that certain interaction. In the NewWordActivity, there is an EditText field that the user can enter the name of the book and save it with the button below it so it would be displayed in the main activity. In the EditWordActivity, there is another EditText present which we can change the name of the book according to what we want and a button below it as well to save the changes. In the DeleteWordActivity, there is a drop down list where the user can select which word the user wants to delete from the RecycleView list in the main activity and the user can confirm the deletion by clicking on the button below the drop down list.

As for the styling of the app, I would follow the code given by Muntenescu, F. and change the RecycleView to have an orange background as well as changing the button color to a turquoise color. The figure below would show the styling choice that was made and applied to all the activities in the app.



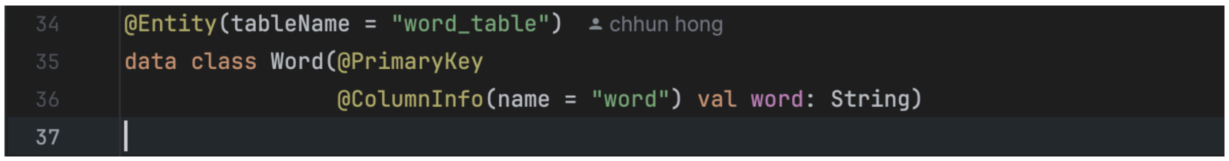
```
<resources>

    <style name="AppTheme" parent="Theme.MaterialComponents.Light.DarkActionBar">
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

    <style name="word_title">
        <item name="android:layout_marginBottom">8dp</item>
        <item name="android:paddingLeft">8dp</item>
        <item name="android:background">@android:color/holo_orange_light</item>
        <item name="android:textAppearance">@android:style/TextAppearance.Large</item>
    </style>
</resources>
```

Figure 6: Style Choices

IV. CODE

For this section of the report, I will divide it into four different sections talking about the Kotlin files that are involved in the app.

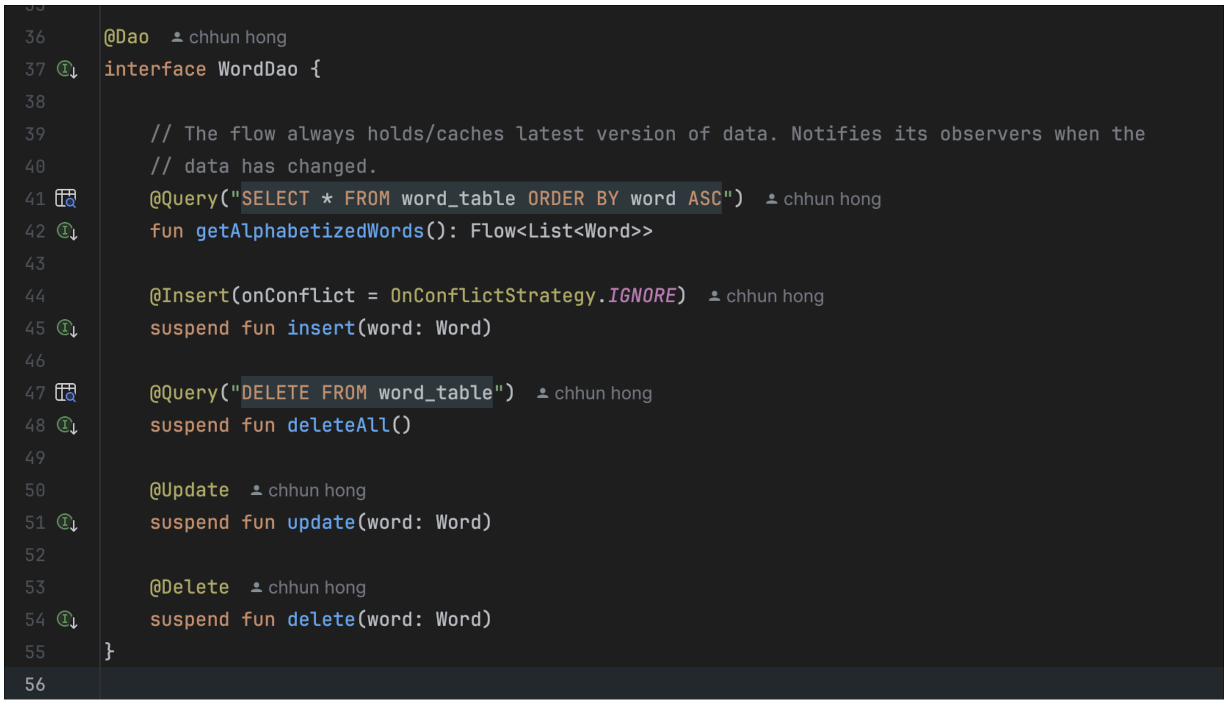
The first section is about the Room database and all the files that are responsible for the database. Those files would include **Word.kt**, **WordDao.kt**, and **WordRoomDatabase.kt.**



```kotlin
@Entity(tableName = "word_table")  chhun hong
data class Word(@PrimaryKey
                @ColumnInfo(name = "word") val word: String)
```

Figure 7: Word.kt

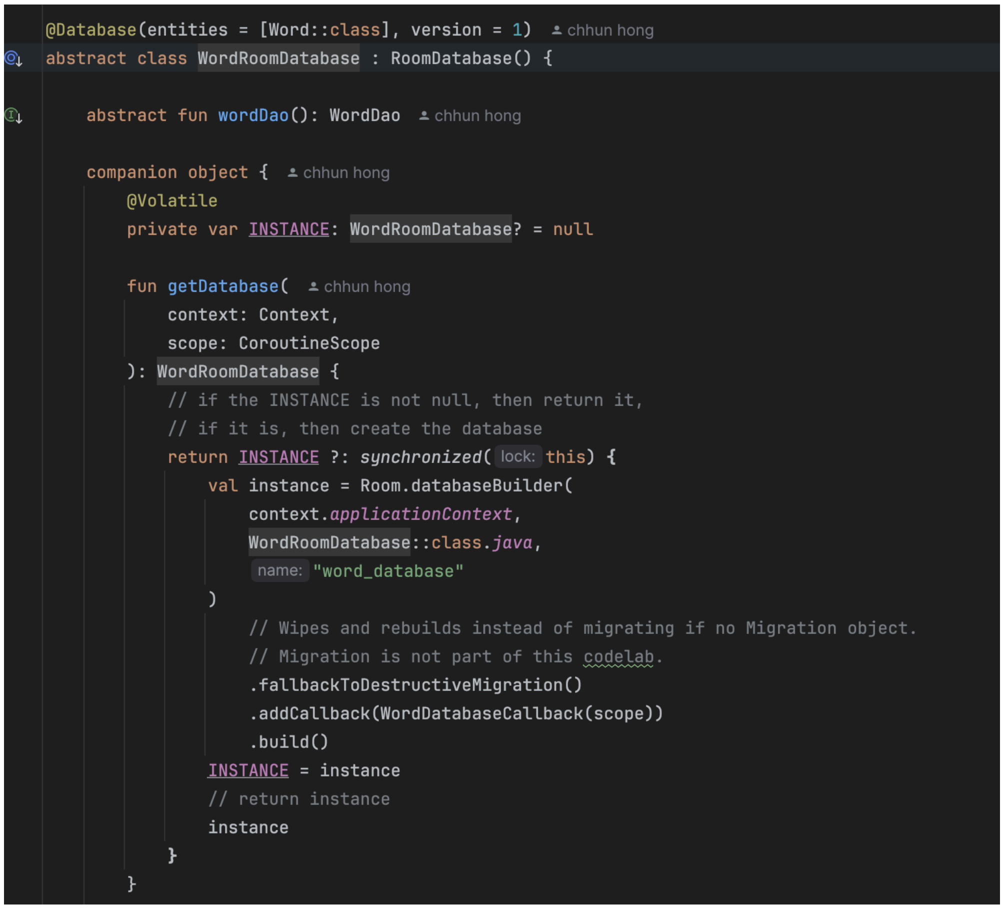
As seen from the figure above, this Kotlin file is responsible for creating an entity table in SQLite responsible for storing the book name that the user has entered. The property of the data class Word would include the @PrimaryKey, the @ColumnInfo, and the property of the class. The @PrimaryKey would act as an ID for the word but in this case we would use the word itself as the ID. This can be changed into an actual Integer ID depending on the implementation. The @ColumnInfo would decide the name of the column in which the data is placed in the table. In this case, the column name would be “word”. Finally, the property of the Word class would be a value called “word” which is a String.



```kotlin
@Dao  chhun hong
interface WordDao {

    // The flow always holds/caches latest version of data. Notifies its observers when the
    // data has changed.
    @Query("SELECT * FROM word_table ORDER BY word ASC")  chhun hong
    fun getAlphabetizedWords(): Flow<List<Word>>

    @Insert(onConflict = OnConflictStrategy.IGNORE)  chhun hong
    suspend fun insert(word: Word)

    @Query("DELETE FROM word_table")  chhun hong
    suspend fun deleteAll()

    @Update  chhun hong
    suspend fun update(word: Word)

    @Delete  chhun hong
    suspend fun delete(word: Word)
}
```

Figure 8: WordDao.kt

The next part of the first section is the WordDao.kt file which is a Dao class which allows us to specify the SQL queries and would associate them with functions to perform a certain action. For this Dao class, it would have five SQL queries which includes a query that would sort out the RecycleView list in the main activity so that it would appear in alphabetical order, an insert method that would insert the a book name into the main activity as well as ignoring duplicates which can be seen by the onConflict = OnConflictStrategy.IGNORE, a query that would delete all the words from the word_table that we created from Word.kt, an update method that would allow the user to edit the book names, and a delete method that would allow the user to delete a specific book name from the RecycleView list.

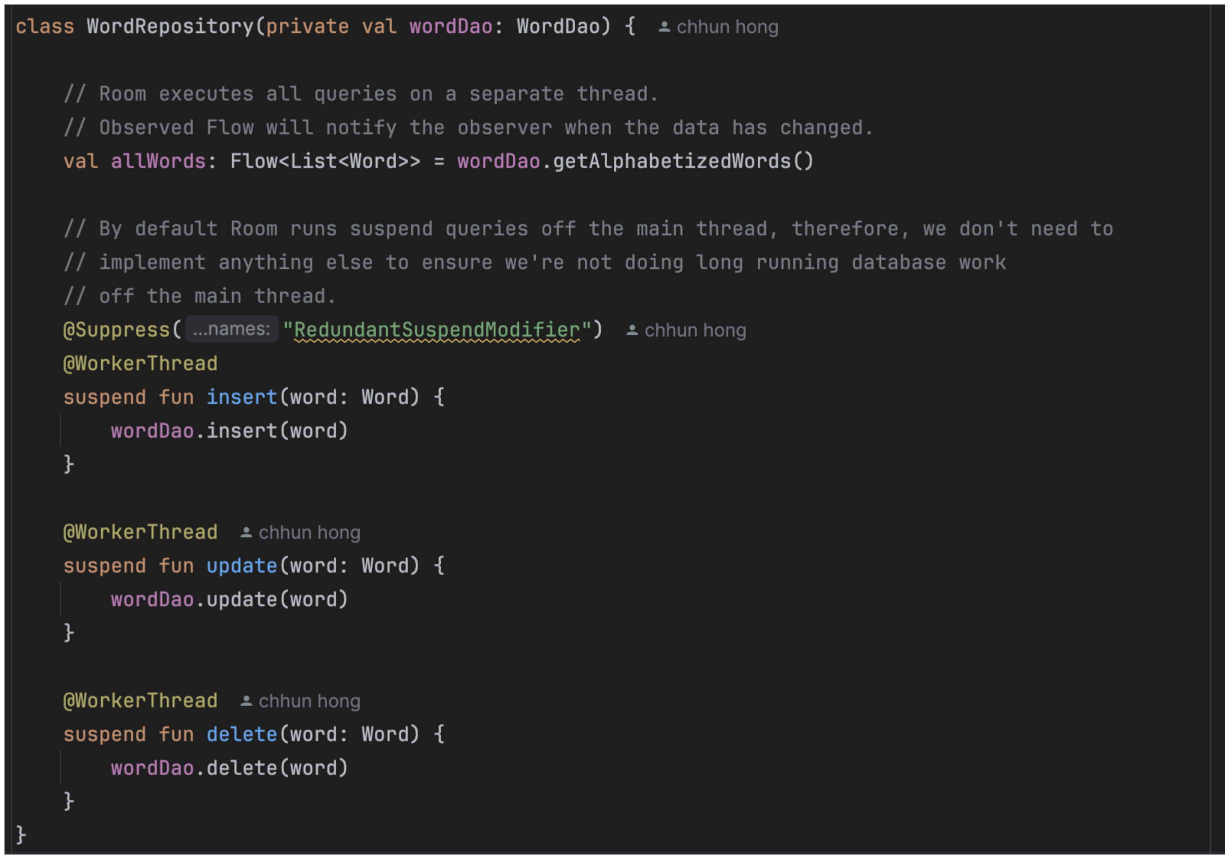


```
@Database(entities = [Word::class], version = 1)  chhun hong
abstract class WordRoomDatabase : RoomDatabase() {

    abstract fun wordDao(): WordDao  chhun hong

    companion object {  chhun hong
        @Volatile
        private var INSTANCE: WordRoomDatabase? = null

        fun getDatabase(  chhun hong
            context: Context,
            scope: CoroutineScope
        ): WordRoomDatabase {
            // if the INSTANCE is not null, then return it,
            // if it is, then create the database
            return INSTANCE ?: synchronized( lock: this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    WordRoomDatabase::class.java,
                     name: "word_database"
                )
                    // Wipes and rebuilds instead of migrating if no Migration object.
                    // Migration is not part of this codelab.
                    .fallbackToDestructiveMigration()
                    .addCallback(WordDatabaseCallback(scope))
                    .build()
                INSTANCE = instance
                // return instance
                instance
            }
        }
```

Figure 9: WordRoomDatabase.kt

The last part of the first section is about the WordRoomDatabase.kt file which is responsible for creating the database for our app. The class would need to be annotated with @Database and the parameter must include the entity that belongs in this database which in this case is Word and there is an abstract fun wordDao() would allow the database to get the methods for each Dao that is present. There would be a private variable created called INSTANCE and would be defined as a singleton which would prevent multiple instances of the database from launching at the same time. Afterwards, there would be a function called getDatabase that would return the instance if it is not null and if it is null, it would create a database with the name word_database.

The second section would be about the **WordRepository.kt** file and the responsibility of that file.

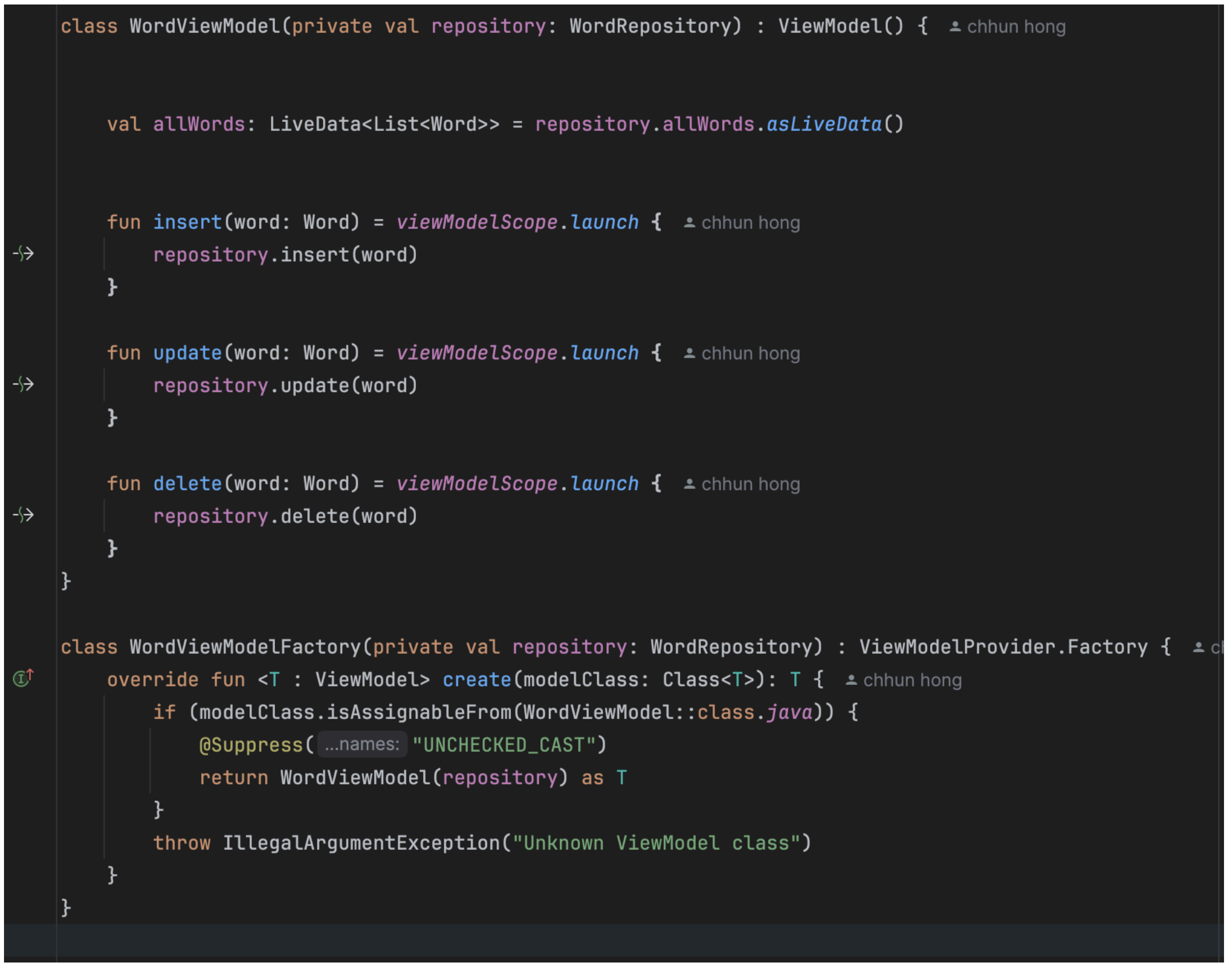


```
class WordRepository(private val wordDao: WordDao) {  chhun hong

    // Room executes all queries on a separate thread.
    // Observed Flow will notify the observer when the data has changed.
    val allWords: Flow<List<Word>> = wordDao.getAlphabetizedWords()

    // By default Room runs suspend queries off the main thread, therefore, we don't need to
    // implement anything else to ensure we're not doing long running database work
    // off the main thread.
    @Suppress( ...names: "RedundantSuspendModifier")  chhun hong
    @WorkerThread
    suspend fun insert(word: Word) {
        wordDao.insert(word)
    }

    @WorkerThread  chhun hong
    suspend fun update(word: Word) {
        wordDao.update(word)
    }

    @WorkerThread  chhun hong
    suspend fun delete(word: Word) {
        wordDao.delete(word)
    }
}
```

Figure 10: WordRepository.kt

The repository is responsible for managing the queries and allows us to use multiple backends (Muntenescu, F.). The WordRepository class would take in a private property of the WordDao since that file is responsible for the queries and methods that would allow us to add, edit, and delete words from the database. A value allWords is created to run all the queries in another thread and is used to monitor any data that has been changed. For all the suspend functions that are present in the last part of this file, it would allow the database to run those functions in the main thread.

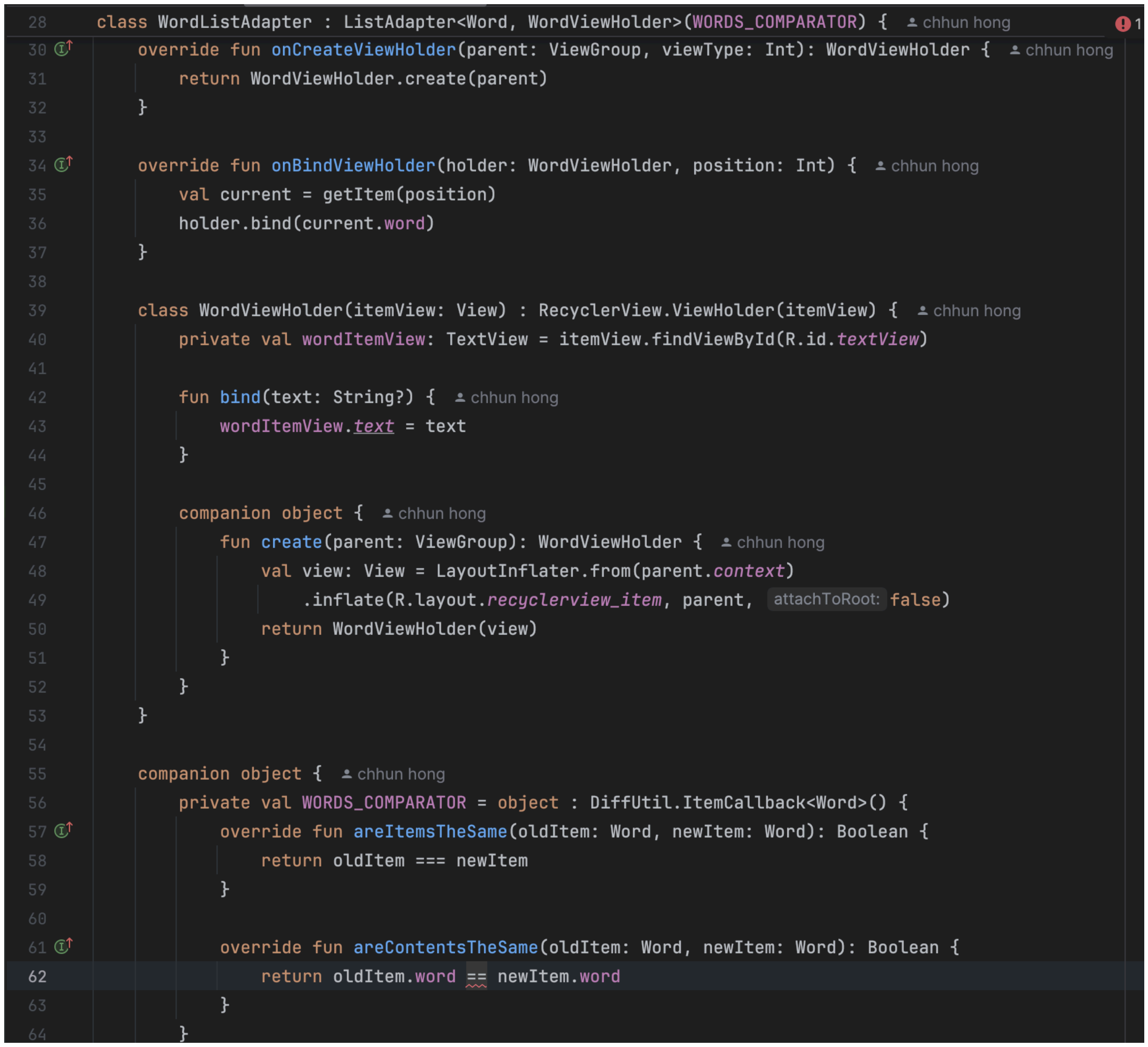
The third section of the code would be about the **WordViewModel.kt** and **WordListAdapter.kt** files and the explanation of those files.



```kotlin
class WordViewModel(private val repository: WordRepository) : ViewModel() {  chhun hong


    val allWords: LiveData<List<Word>> = repository.allWords.asLiveData()


    fun insert(word: Word) = viewModelScope.launch {  chhun hong
        repository.insert(word)
    }

    fun update(word: Word) = viewModelScope.launch {  chhun hong
        repository.update(word)
    }

    fun delete(word: Word) = viewModelScope.launch {  chhun hong
        repository.delete(word)
    }
}

class WordViewModelFactory(private val repository: WordRepository) : ViewModelProvider.Factory {  c
    override fun <T : ViewModel> create(modelClass: Class<T>): T {  chhun hong
        if (modelClass.isAssignableFrom(WordViewModel::class.java)) {
            @Suppress( ...names: "UNCHECKED_CAST")
            return WordViewModel(repository) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

Figure 11: WordViewModel.kt

The ViewModel is responsible for storing and managing data that is related to the UI in a lifecycle-conscious way and would also act as a way to communicate between the Repository and the UI. The class WordViewModel would take a private property from the WordRepository file as its single source of truth when dealing with the data. In the class, there would be a value called allWords which would use Live Data. By doing this it would only update the UI when the data is actually changed. For the functions insert(), update(), and delete(), it would be responsible for adding, updating, and deleting respectively and launching it in a new coroutine so that it is not in a non-blocking way. The WordViewModelFactory class would create the ViewModel and provide the necessary dependencies to the WordViewModel class when it is created. If the factory creates any other ViewModel that is not the WordViewModel, it will throw an exception and give the error “Unknown ViewModel class”.

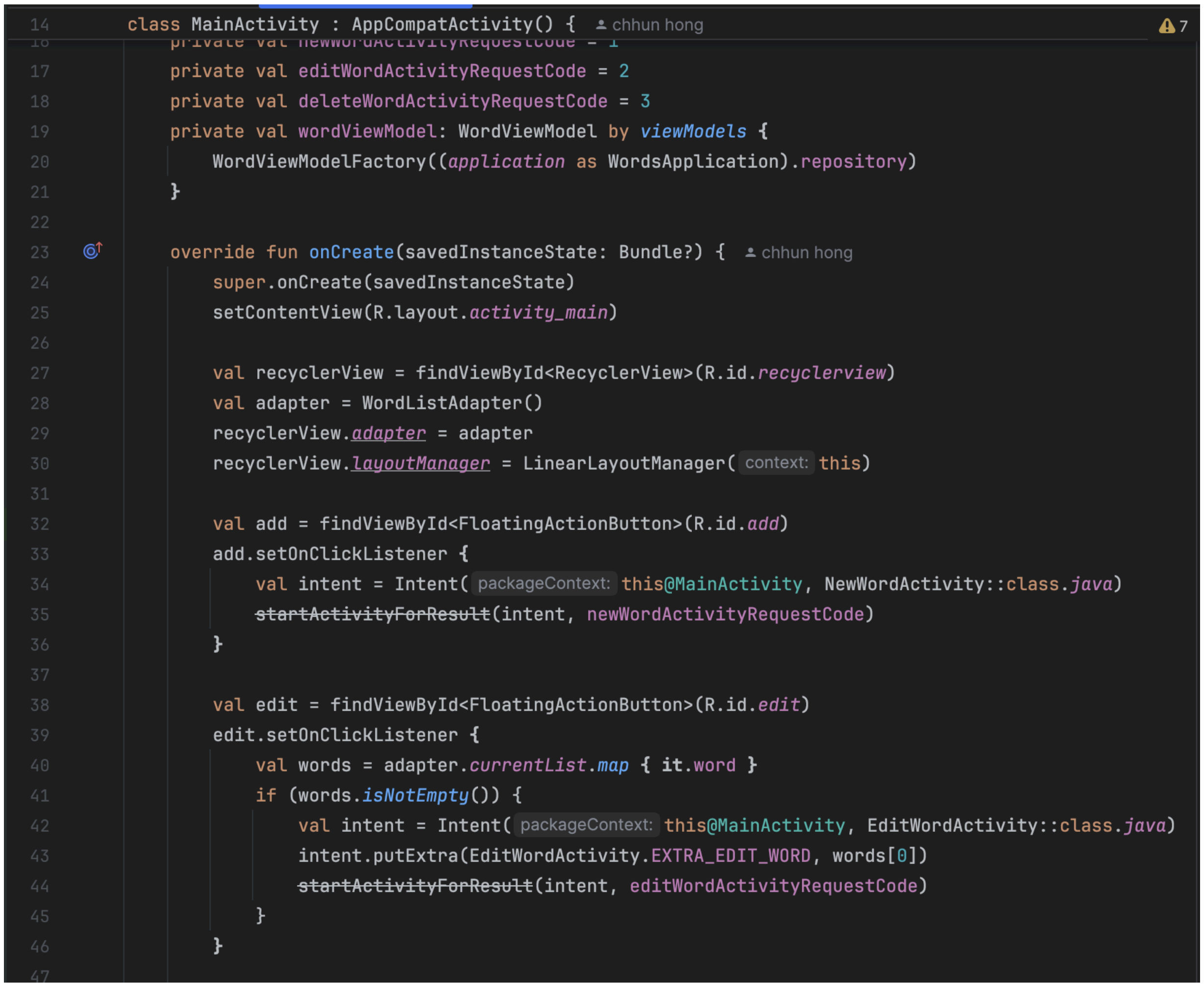


```kotlin
class WordListAdapter : ListAdapter<Word, WordViewHolder>(WORDS_COMPARATOR) {  chhun hong
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): WordViewHolder {  chhun hong
        return WordViewHolder.create(parent)
    }

    override fun onBindViewHolder(holder: WordViewHolder, position: Int) {  chhun hong
        val current = getItem(position)
        holder.bind(current.word)
    }

    class WordViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {  chhun hong
        private val wordItemView: TextView = itemView.findViewById(R.id.textView)

        fun bind(text: String?) {  chhun hong
            wordItemView.text = text
        }

        companion object {  chhun hong
            fun create(parent: ViewGroup): WordViewHolder {  chhun hong
                val view: View = LayoutInflater.from(parent.context)
                    .inflate(R.layout.recyclerview_item, parent,  attachToRoot: false)
                return WordViewHolder(view)
            }
        }
    }

    companion object {  chhun hong
        private val WORDS_COMPARATOR = object : DiffUtil.ItemCallback<Word>() {
            override fun areItemsTheSame(oldItem: Word, newItem: Word): Boolean {
                return oldItem === newItem
            }

            override fun areContentsTheSame(oldItem: Word, newItem: Word): Boolean {
                return oldItem.word == newItem.word
            }
        }
```

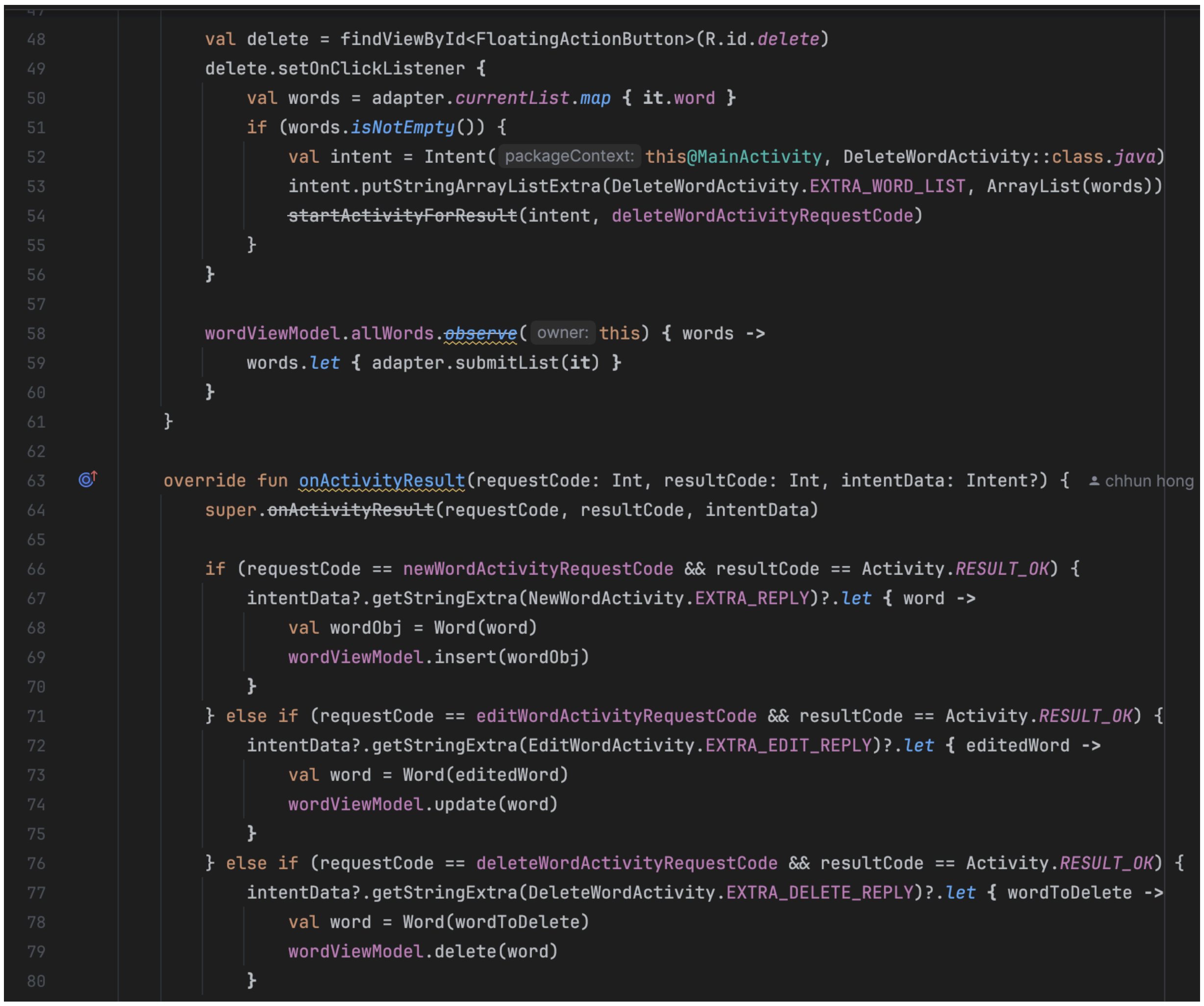
Figure 12: WordListAdapter.kt

The WordListAdapter.kt is a RecycleView that would display the lists of book names at the main activity. The override function onCreateViewHolder would create a new ViewModel for the book name to be displayed and it would return the created WordViewModel so that it would populate the list of book names. The override function onBindViewHolder would bind the data to a specific position in the list and pass it to the WordViewHolder class so that the function bind can be used to display the book name. The WordViewHolder class would initialize a TextView in the .xml file for the WordListAdapter and would bind the book name to the TextView using the function bind(). The companion object below it would create a new WordViewHolder which would inflate the RecycleView with all the book names. For the other companion object that is outside the WordViewHolder class it would be responsible for efficiently updating the list when there is a change in data. There are two functions in companion object as well which is the override function areItemTheSame and areContentsTheSame. For the first function, it would check whether the two Word objects are the same based on their reference and the second function would check whether the two Word object's strings are the same.

For the final section of the app, it would be the activities that are present in the app. This would include **MainActivity.kt, NewWordActivity.kt, DeleteWordActivity.kt, and EditWordActivity.kt**.



```
class MainActivity : AppCompatActivity() {  chhun hong
    private val newWordActivityRequestCode = 1
    private val editWordActivityRequestCode = 2
    private val deleteWordActivityRequestCode = 3
    private val wordViewModel: WordViewModel by viewModels {
        WordViewModelFactory((application as WordsApplication).repository)
    }

    override fun onCreate(savedInstanceState: Bundle?) {  chhun hong
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val recyclerView = findViewById<RecyclerView>(R.id.recyclerview)
        val adapter = WordListAdapter()
        recyclerView.adapter = adapter
        recyclerView.layoutManager = LinearLayoutManager( context: this)

        val add = findViewById<FloatingActionButton>(R.id.add)
        add.setOnClickListener {
            val intent = Intent( packageContext: this@MainActivity, NewWordActivity::class.java)
            ~~startActivityForResult~~(intent, newWordActivityRequestCode)
        }

        val edit = findViewById<FloatingActionButton>(R.id.edit)
        edit.setOnClickListener {
            val words = adapter.currentList.map { it.word }
            if (words.isNotEmpty()) {
                val intent = Intent( packageContext: this@MainActivity, EditWordActivity::class.java)
                intent.putExtra(EditWordActivity.EXTRA_EDIT_WORD, words[0])
                ~~startActivityForResult~~(intent, editWordActivityRequestCode)
            }
        }
```

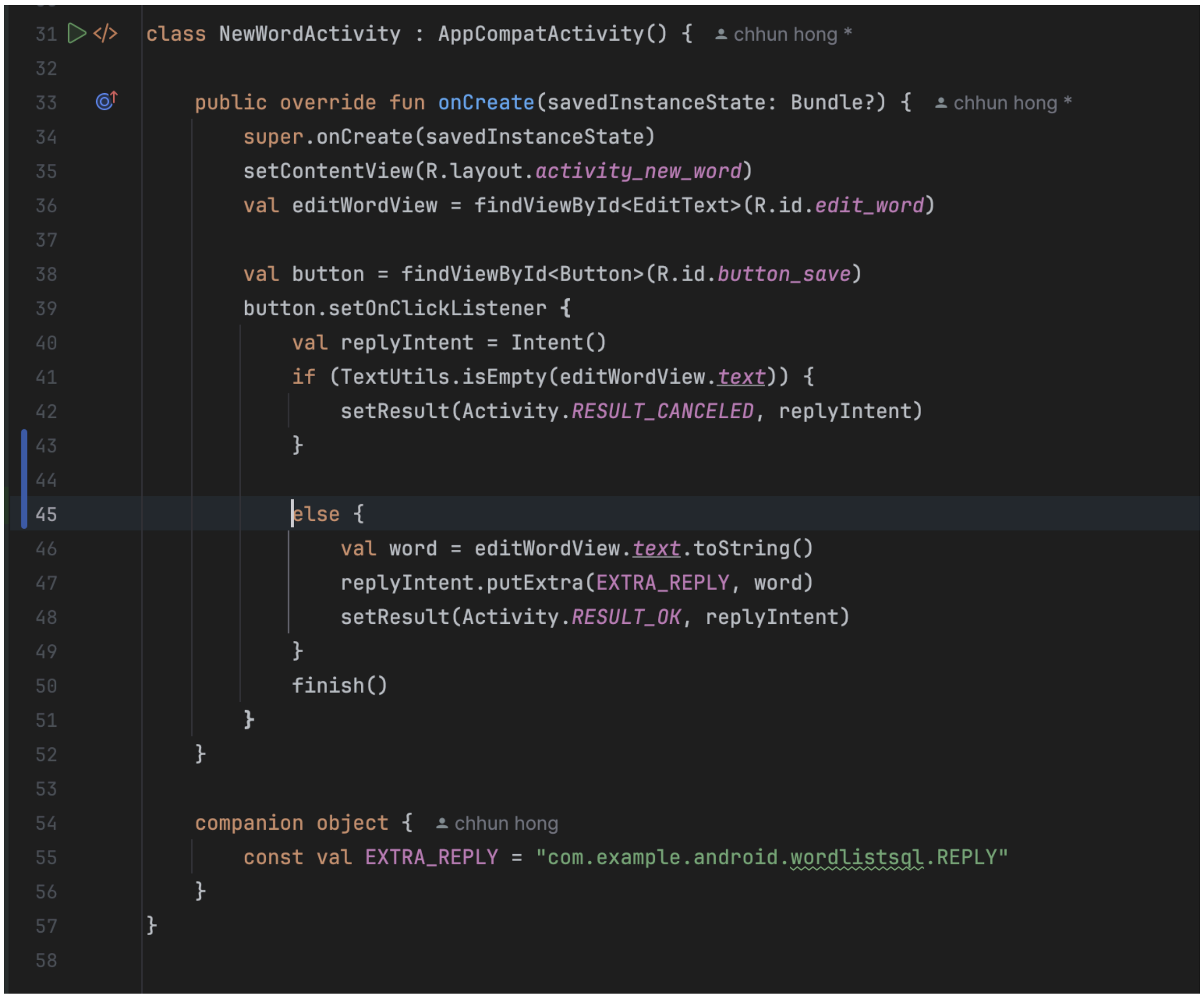


```kotlin
        val delete = findViewById<FloatingActionButton>(R.id.delete)
        delete.setOnClickListener {
            val words = adapter.currentList.map { it.word }
            if (words.isNotEmpty()) {
                val intent = Intent( packageContext: this@MainActivity, DeleteWordActivity::class.java)
                intent.putStringArrayListExtra(DeleteWordActivity.EXTRA_WORD_LIST, ArrayList(words))
                ~~startActivityForResult~~(intent, deleteWordActivityRequestCode)
            }
        }

        wordViewModel.allWords.~~observe~~( owner: this) { words ->
            words.let { adapter.submitList(it) }
        }
    }

    override fun onActivityResult(requestCode: Int, resultCode: Int, intentData: Intent?) {  chhun hong
        super.~~onActivityResult~~(requestCode, resultCode, intentData)

        if (requestCode == newWordActivityRequestCode && resultCode == Activity.RESULT_OK) {
            intentData?.getStringExtra(NewWordActivity.EXTRA_REPLY)?.let { word ->
                val wordObj = Word(word)
                wordViewModel.insert(wordObj)
            }
        } else if (requestCode == editWordActivityRequestCode && resultCode == Activity.RESULT_OK) {
            intentData?.getStringExtra(EditWordActivity.EXTRA_EDIT_REPLY)?.let { editedWord ->
                val word = Word(editedWord)
                wordViewModel.update(word)
            }
        } else if (requestCode == deleteWordActivityRequestCode && resultCode == Activity.RESULT_OK) {
            intentData?.getStringExtra(DeleteWordActivity.EXTRA_DELETE_REPLY)?.let { wordToDelete ->
                val word = Word(wordToDelete)
                wordViewModel.delete(word)
            }
```

Figure 13: MainActivity.kt

The MainActivity is responsible for displaying the RecycleView list which contains the name of the books that the user is interested in as well as having three buttons that would navigate the user into three different activities. As seen in the first image from the figure above, I would first initialize the request code of the NewWordActivity, EditWordActivity, and DeleteWordActivity and give them the value of 1, 2, and 3 respectively so those activities can be communicate back and forth with the MainActivity. The next value that is initialized is wordViewModel which would create a WordViewModel and use the viewModels delegate and pass in an instance of the WordViewModelFactory. The next after would be the initialization of the RecycleView by giving it the value of recycleView. This RecycleView would follow a Linear Layout format when the books are added to the list. The next part would be the initialization of the buttons used to navigate to the different activities. The first button is called “add” which is a Floating Action Button which is set with an onClickListener so that when the user clicks on the button, it would start an intent and would navigate the user to the NewWordActivity. This intent would also listen for the result back by using the method startActivityForResult() which would take in the intent and the newWordActivityRequestCode which was initialized at the beginning of the code. The next button is the “edit” button which is also a Floating Action Button and would follow a similar concept as the “add” button but it would get the list of names by creating a value “words” and check whether the list is empty or not. If the list is not empty, it would create an intent and navigate the user to the EditWordActivity. Additionally, it would communicate to the EditWordActivity by sending the data of the book that needs to be edited and would also include a startActivityForResult() as well with the created intent and its respective request code. The last button is the “delete” button which is a Floating Action Button which is similar to the EditWordActivity but instead it would navigate the user to the DeleteWordActivity and send the string array so it can be used for the drop down list in the DeleteWordActivity. It would also have a startActivityForResult() with the created intent for this button and its own request code. The next part is wordViewModel.allWords.observe() which would observe for any changes in the data and activity.

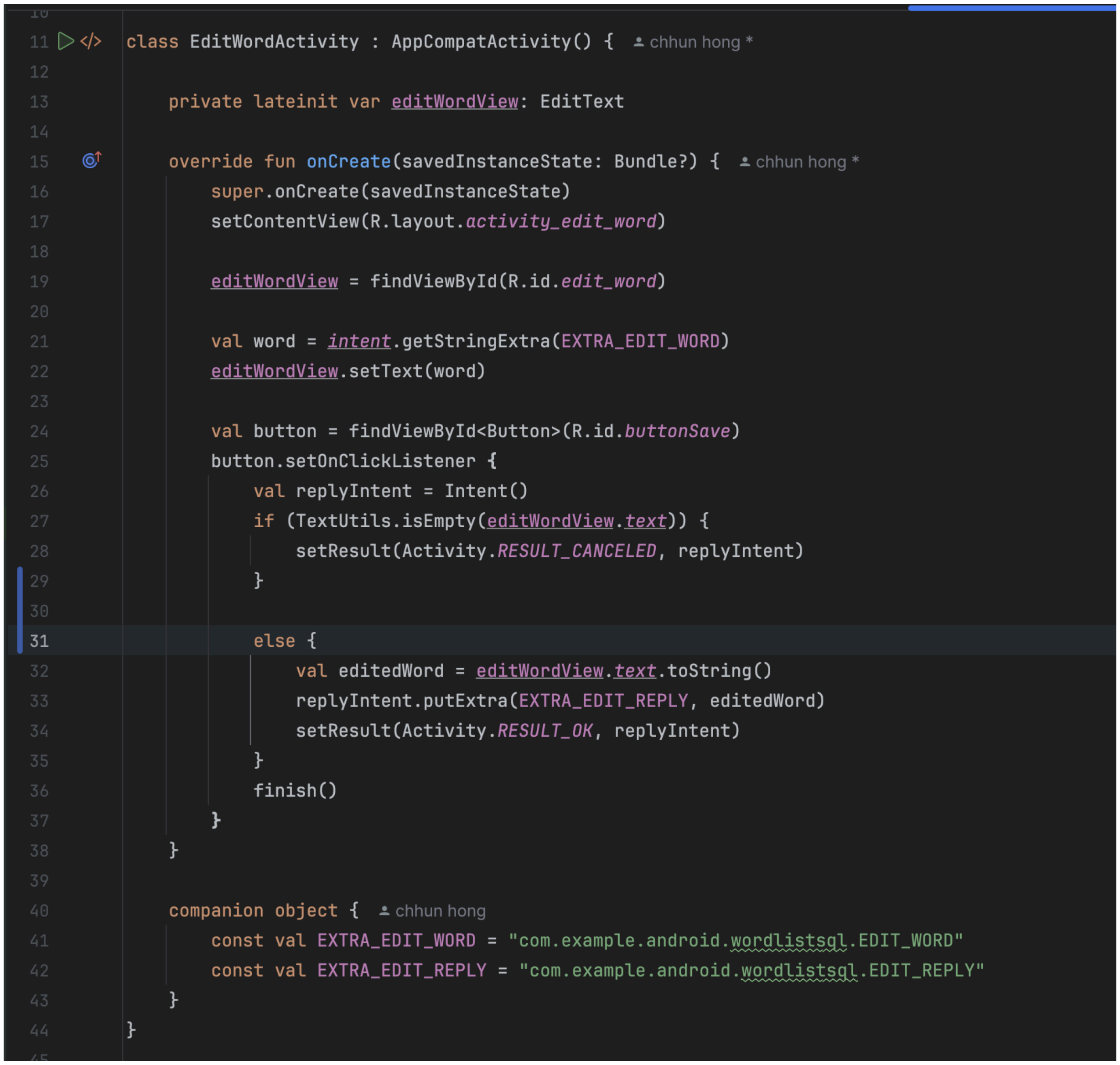
The next section is about when the activity is returned to the MainActivity. It would check the request code first to see which activity is it and would also check if the result code is equal to RESULT_OK. If the request code is from the NewWordActivity, it would get the string data that was entered in that activity and use the insert method from the wordViewModel to add the book to the RecycleView list in the MainActivity. If the request code is from the EditWordActivity, it would get the string data from that activity and would use the update method from the wordViewModel to update the word in the list. If the request code is from the DeleteWordActivity, it would get the string data that needs to be deleted and use the delete method from wordViewModel to delete the book from the RecycleView list.



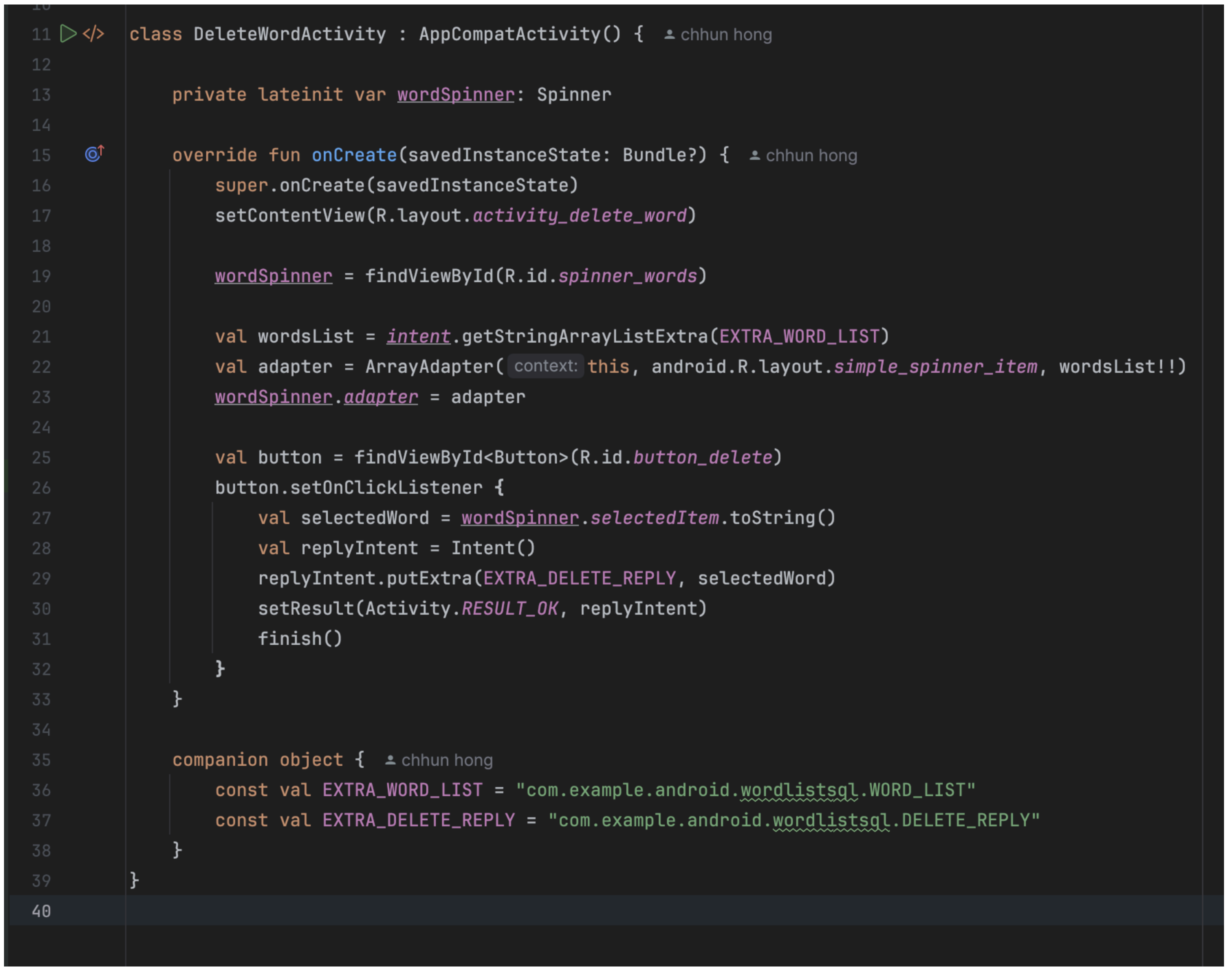
```kotlin
class NewWordActivity : AppCompatActivity() {  chhun hong *

    public override fun onCreate(savedInstanceState: Bundle?) {  chhun hong *
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_new_word)
        val editWordView = findViewById<EditText>(R.id.edit_word)

        val button = findViewById<Button>(R.id.button_save)
        button.setOnClickListener {
            val replyIntent = Intent()
            if (TextUtils.isEmpty(editWordView.text)) {
                setResult(Activity.RESULT_CANCELED, replyIntent)
            }

            else {
                val word = editWordView.text.toString()
                replyIntent.putExtra(EXTRA_REPLY, word)
                setResult(Activity.RESULT_OK, replyIntent)
            }
            finish()
        }
    }

    companion object {  chhun hong
        const val EXTRA_REPLY = "com.example.android.wordlistsql.REPLY"
    }
}
```

Figure 14: NewWordActivity.kt

The NewWordActivity file is an activity that can be navigated to when the user clicks on the add button from the MainActivity. In this file there is an EditText field that is initialized with the editWordView as well as a button used to save the book name. That button is set with an onClickListener so that when the user clicks on it, it will check the EditText field to see if it is empty or not. If it is empty, the result of that activity would equal to RESULT_CANCELED and the data would be returned back to the MainActivity with the setResult() method. If it is not empty, the text in the EditText field will be returned back to the MainActivity using the setResult() method with the value RESULT_OK and the reply intent. The finish() method will be called after the activity is finished and will return the user back to the MainActivity. The companion object at the bottom is used to communicate between the intents and activities.


```
class EditWordActivity : AppCompatActivity() {  chhun hong *

    private lateinit var editWordView: EditText

    override fun onCreate(savedInstanceState: Bundle?) {  chhun hong *
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_edit_word)

        editWordView = findViewById(R.id.edit_word)

        val word = intent.getStringExtra(EXTRA_EDIT_WORD)
        editWordView.setText(word)

        val button = findViewById<Button>(R.id.buttonSave)
        button.setOnClickListener {
            val replyIntent = Intent()
            if (TextUtils.isEmpty(editWordView.text)) {
                setResult(Activity.RESULT_CANCELED, replyIntent)
            }

            else {
                val editedWord = editWordView.text.toString()
                replyIntent.putExtra(EXTRA_EDIT_REPLY, editedWord)
                setResult(Activity.RESULT_OK, replyIntent)
            }
            finish()
        }
    }

    companion object {  chhun hong
        const val EXTRA_EDIT_WORD = "com.example.android.wordlistsql.EDIT_WORD"
        const val EXTRA_EDIT_REPLY = "com.example.android.wordlistsql.EDIT_REPLY"
    }
}
```
Figure 15: EditWordActivity.kt

The EditWordActivity would follow a similar concept as the NewWordActivity. It would also have an EditText field that would be already completed with the name of the book that the user wants to edit. Once the user is satisfied with the changes made, the user can click the save button and it would be communicated back to the MainActivity using the setResult() with the reply intent data.

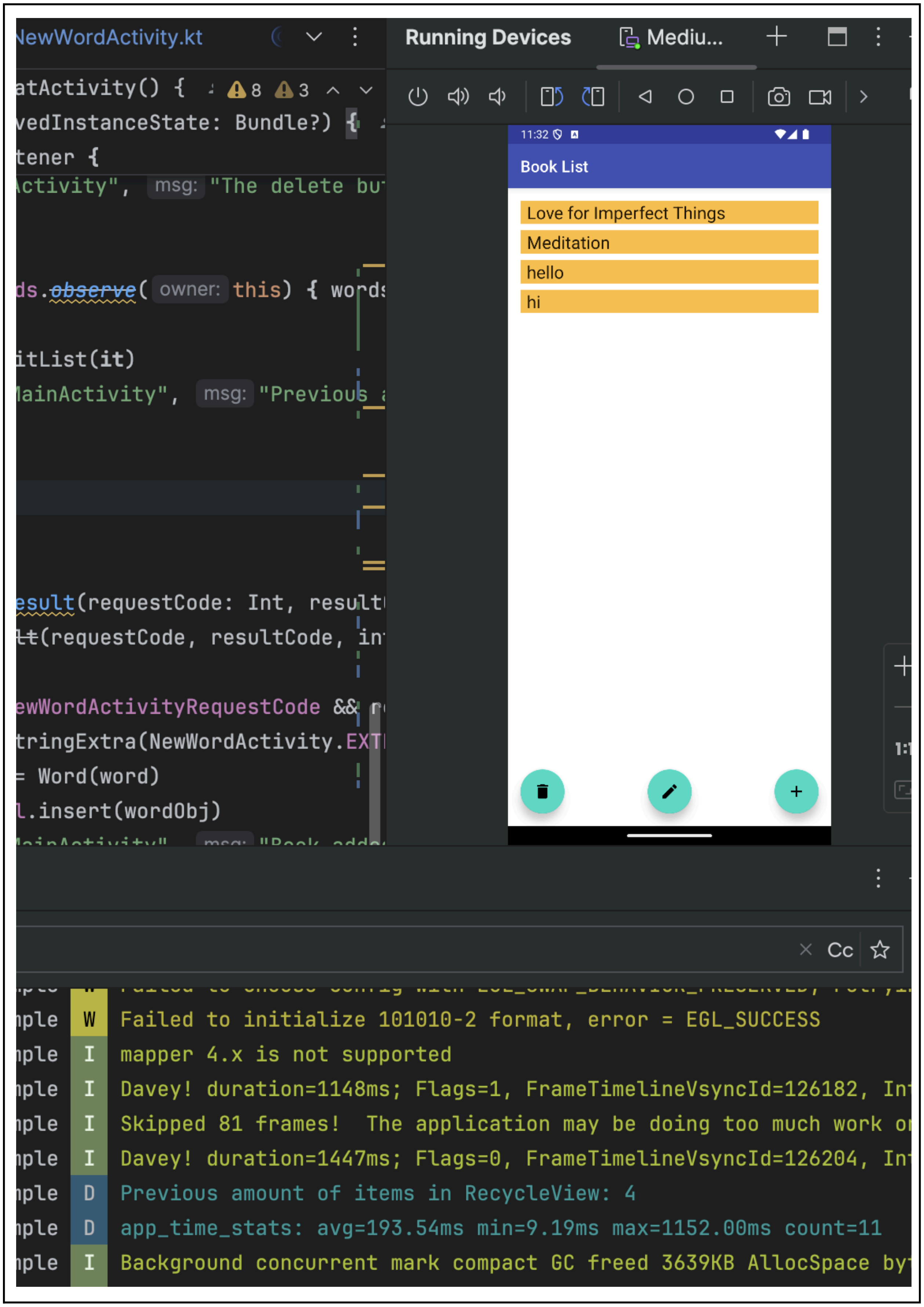


```kotlin
class DeleteWordActivity : AppCompatActivity() {  chhun hong

    private lateinit var wordSpinner: Spinner

    override fun onCreate(savedInstanceState: Bundle?) {  chhun hong
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_delete_word)

        wordSpinner = findViewById(R.id.spinner_words)

        val wordsList = intent.getStringArrayListExtra(EXTRA_WORD_LIST)
        val adapter = ArrayAdapter( context: this, android.R.layout.simple_spinner_item, wordsList!!)
        wordSpinner.adapter = adapter

        val button = findViewById<Button>(R.id.button_delete)
        button.setOnClickListener {
            val selectedWord = wordSpinner.selectedItem.toString()
            val replyIntent = Intent()
            replyIntent.putExtra(EXTRA_DELETE_REPLY, selectedWord)
            setResult(Activity.RESULT_OK, replyIntent)
            finish()
        }
    }

    companion object {  chhun hong
        const val EXTRA_WORD_LIST = "com.example.android.wordlistsql.WORD_LIST"
        const val EXTRA_DELETE_REPLY = "com.example.android.wordlistsql.DELETE_REPLY"
    }
}
```

Figure 16: DeleteWordActivity.kt

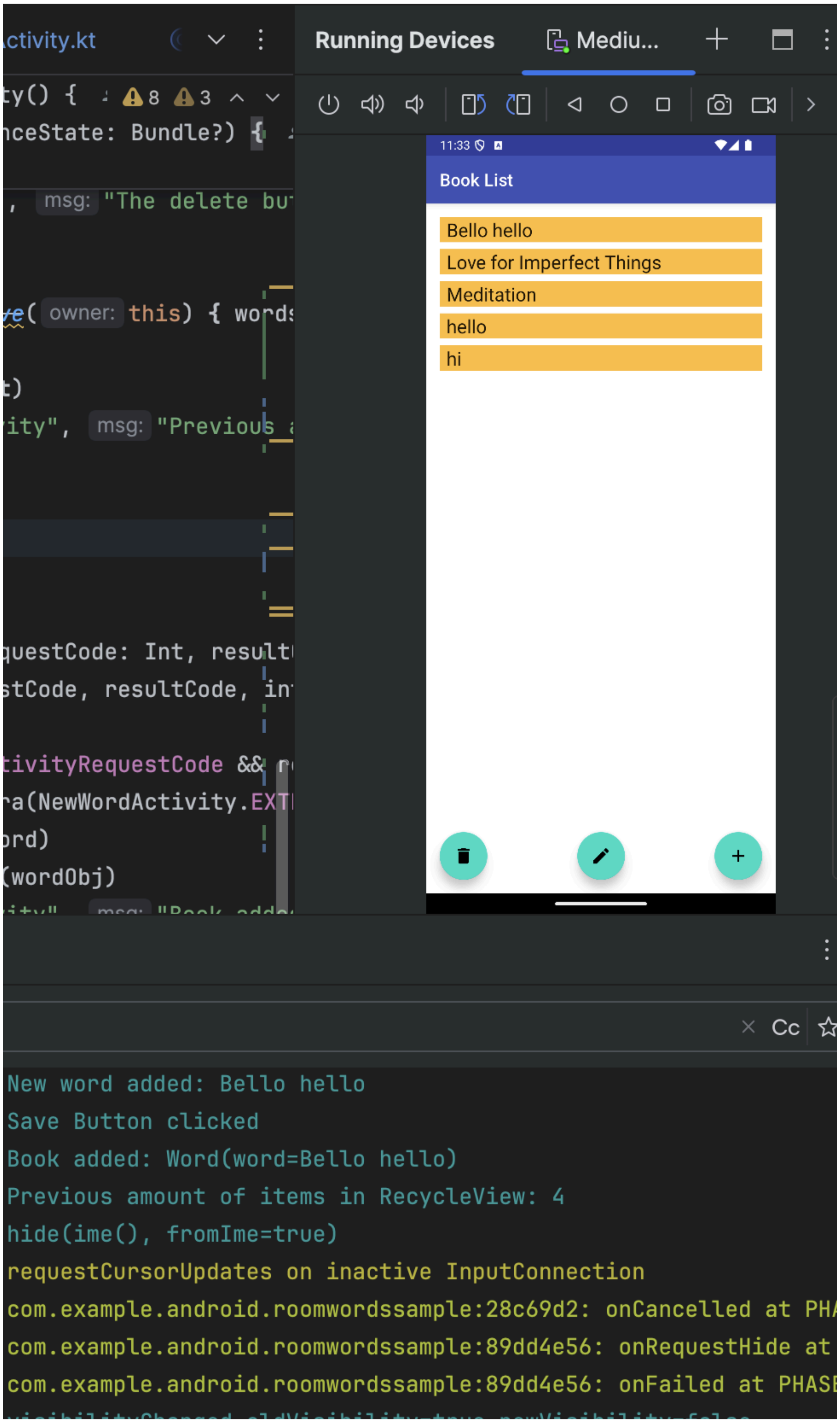
The DeleteWordActivity would follow a similar concept as the previous two activities but the only difference is that instead of an EditText field, this activity would have a drop down list that contains all the books that the user added to the RecycleView list and it would allow the user to pick which words the user want to delete. Once the user have selected the book that they want to delete, they can confirm the deletion by clicking on the button below the drop down list and it would return the user back to the MainActivity with the book deleted from the list.

V. DEBUG



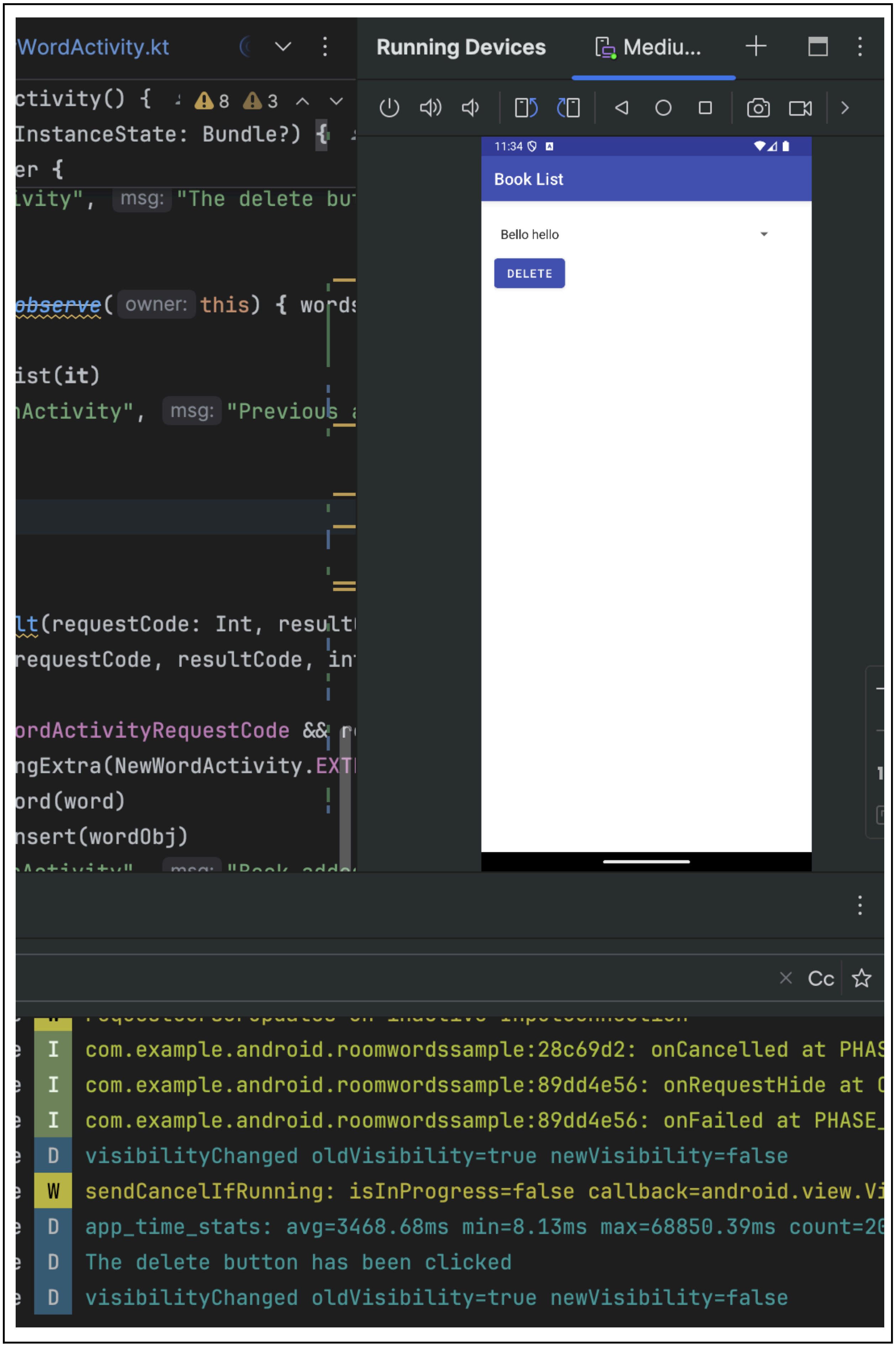
```
atActivity() {
vedInstanceState: Bundle?) {
tener {
Activity", msg: "The delete bu

ds.observe( owner: this) { words

itList(it)
MainActivity", msg: "Previous_a


esult(requestCode: Int, result
lt(requestCode, resultCode, in

ewWordActivityRequestCode && r
tringExtra(NewWordActivity.EXT
= Word(word)
l.insert(wordObj)
```

```
nple  W  Failed to initialize 101010-2 format, error = EGL_SUCCESS
nple  I  mapper 4.x is not supported
nple  I  Davey! duration=1148ms; Flags=1, FrameTimelineVsyncId=126182, In
nple  I  Skipped 81 frames!  The application may be doing too much work o
nple  I  Davey! duration=1447ms; Flags=0, FrameTimelineVsyncId=126204, In
nple  D  Previous amount of items in RecycleView: 4
nple  D  app_time_stats: avg=193.54ms min=9.19ms max=1152.00ms count=11
nple  I  Background concurrent mark compact GC freed 3639KB AllocSpace by
```



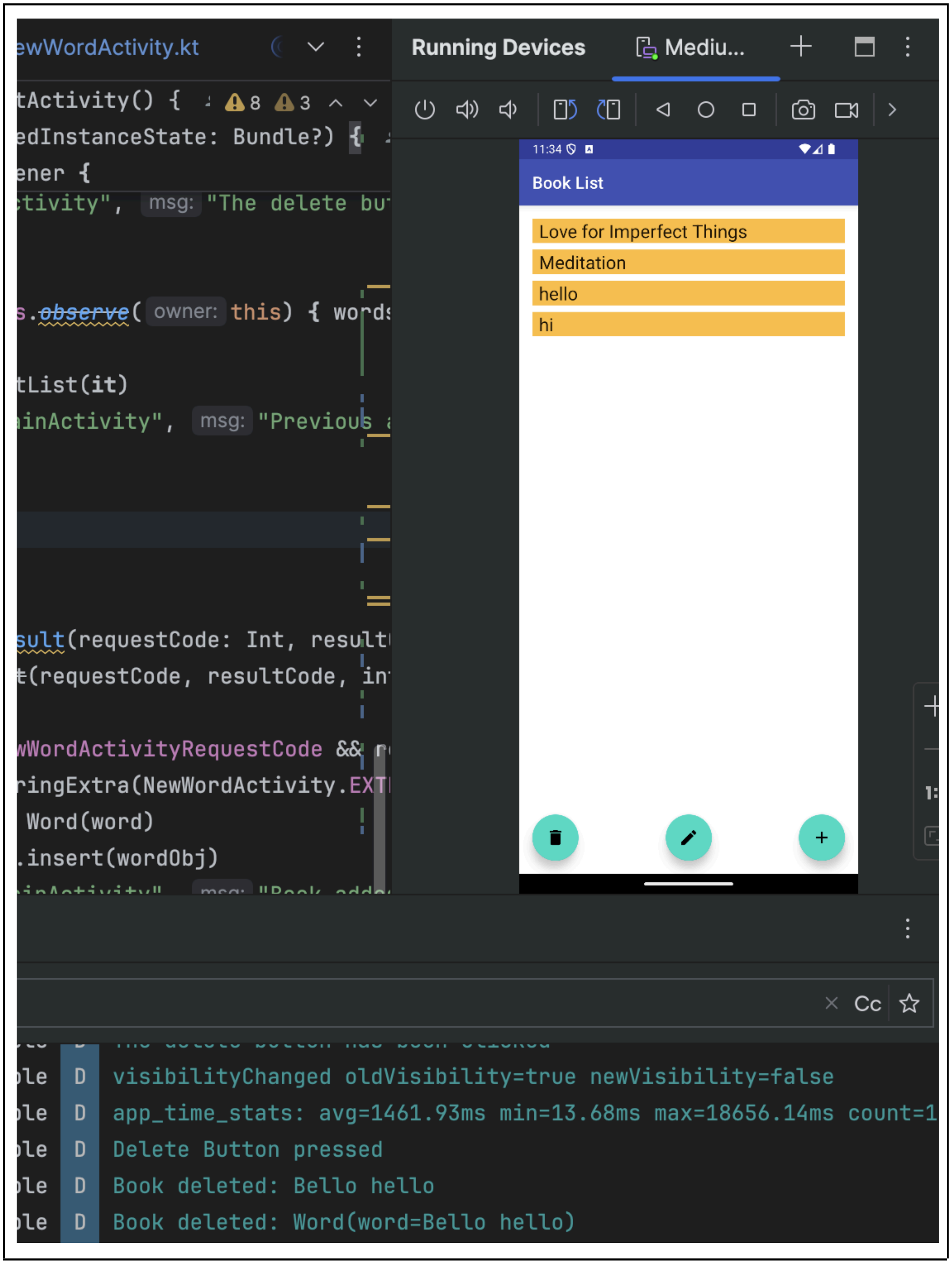
```
ty() {
nceState: Bundle?) {
, msg: "The delete bu
ve( owner: this) { words
t)
ity", msg: "Previous a

questCode: Int, result
stCode, resultCode, in

tivityRequestCode && r
ra(NewWordActivity.EXT
ord)
(wordObj)
```

```
New word added: Bello hello
Save Button clicked
Book added: Word(word=Bello hello)
Previous amount of items in RecycleView: 4
hide(ime(), fromIme=true)
requestCursorUpdates on inactive InputConnection
com.example.android.roomwordssample:28c69d2: onCancelled at PHA
com.example.android.roomwordssample:89dd4e56: onRequestHide at
com.example.android.roomwordssample:89dd4e56: onFailed at PHASE
```



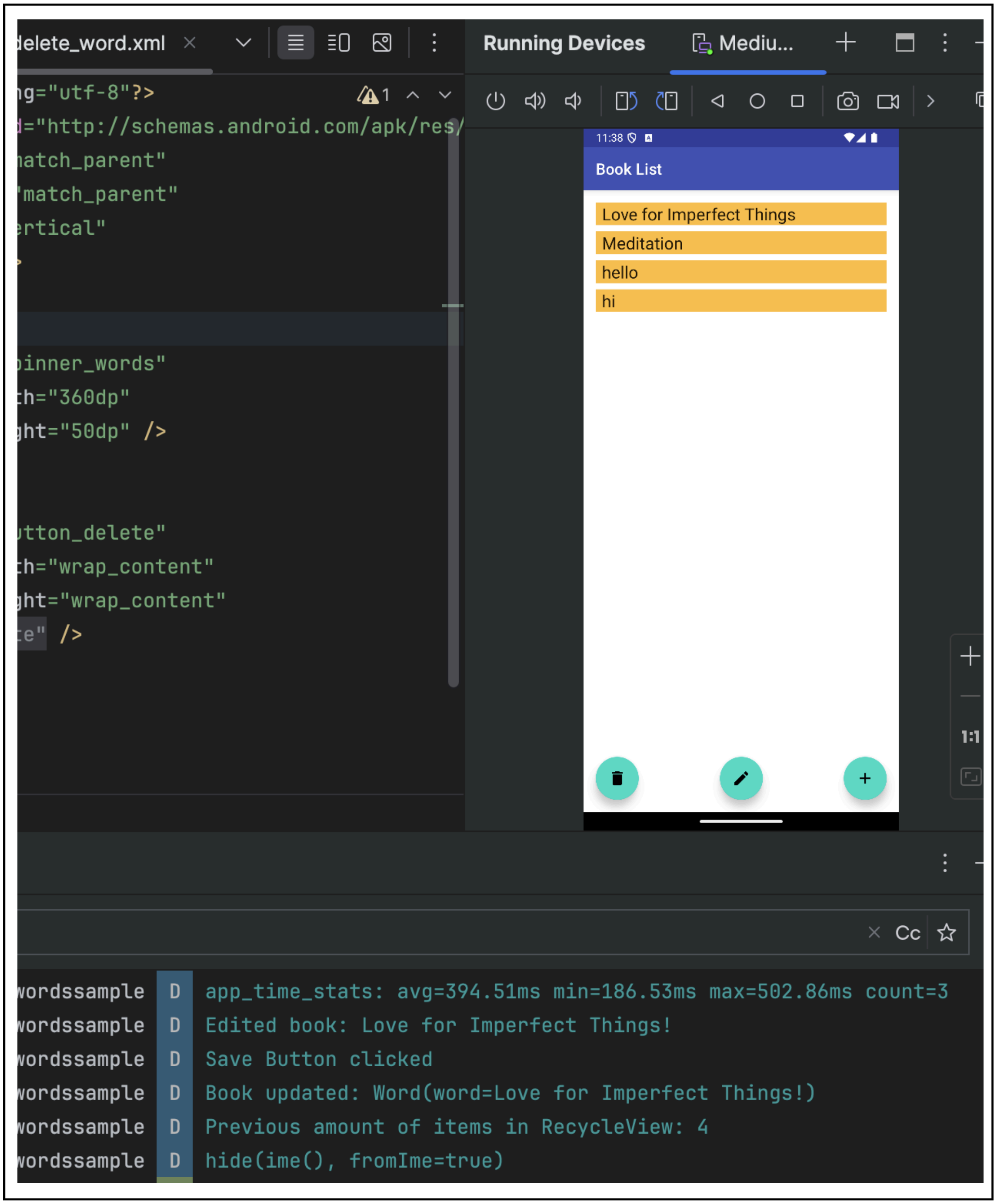
```
ctivity() {
InstanceState: Bundle?) {
er {
ivity",  msg: "The delete bu

observe( owner: this) { words

ist(it)
Activity",  msg: "Previous_

lt(requestCode: Int, result
requestCode, resultCode, in

ordActivityRequestCode && r
ngExtra(NewWordActivity.EXT
ord(word)
nsert(wordObj)
```

```
I  com.example.android.roomwordssample:28c69d2: onCancelled at PHAS
I  com.example.android.roomwordssample:89dd4e56: onRequestHide at
I  com.example.android.roomwordssample:89dd4e56: onFailed at PHASE_
D  visibilityChanged oldVisibility=true newVisibility=false
W  sendCancelIfRunning: isInProgress=false callback=android.view.Vi
D  app_time_stats: avg=3468.68ms min=8.13ms max=68850.39ms count=2
D  The delete button has been clicked
D  visibilityChanged oldVisibility=true newVisibility=false
```



```
tActivity() {
edInstanceState: Bundle?) {
ener {
tivity",  msg: "The delete bu

s.observe( owner: this) { words

tList(it)
inActivity",  msg: "Previous_a



sult(requestCode: Int, result
t(requestCode, resultCode, in

wWordActivityRequestCode && r
ringExtra(NewWordActivity.EXT
 Word(word)
.insert(wordObj)
```

```
ble  D  visibilityChanged oldVisibility=true newVisibility=false
ble  D  app_time_stats: avg=1461.93ms min=13.68ms max=18656.14ms count=1
ble  D  Delete Button pressed
ble  D  Book deleted: Bello hello
ble  D  Book deleted: Word(word=Bello hello)
```



```
ng="utf-8"?>
d="http://schemas.android.com/apk/res/
natch_parent"
'match_parent"
ertical"
>

pinner_words"
th="360dp"
ght="50dp" />


utton_delete"
th="wrap_content"
ght="wrap_content"
te" />
```

```
wordssample  D  app_time_stats: avg=394.51ms min=186.53ms max=502.86ms count=3
wordssample  D  Edited book: Love for Imperfect Things!
wordssample  D  Save Button clicked
wordssample  D  Book updated: Word(word=Love for Imperfect Things!)
wordssample  D  Previous amount of items in RecycleView: 4
wordssample  D  hide(ime(), fromIme=true)
```

VI. CONCLUSION

The resulting app would be a list app that would allow the user to keep track of the books that they are interested in. In the MainActivity, there will be a RecycleView that would display a list of books that the user found interesting. Additionally, there are three buttons located at the bottom of the MainActivity for the user to interact with. One of the buttons with the add symbol would navigate the user to a different activity where they can type the name of the book that they are interested in to add to the RecycleView list in the MainActivity. Once they typed out the name, there was a button in the activity to confirm the name so that it would be added to the RecycleView list in the MainActivity. The next button is the edit button which is denoted by a pencil icon and would navigate the user to another activity in which they can edit the name of the book in case they made a mistake while typing it or they got the wrong name. Once the changes have been made, they can click on the Save button and it will save the changes made and return the user back to the MainActivity. The last button is the delete button that is shown with the bin icon and would bring the user to a new activity in which they can choose which book name they want to delete from a drop down list. Once the user picks the book name that they want to delete, they can click on the Delete button and it will delete it from the RecycleView list in the MainActivity and return the user back there as well.

VII. REFERENCE

1. **GitHub Repo:** https://github.com/SoftDevMobDev-2024-Classrooms/assignment03-scherqw.git
2. **GitHub Repo for the Room tutorial:** https://github.com/android/codelab-android-room-with-a-view.git
3. Android Studio (2024, June). *Delete*. <https://developer.android.com/reference/androidx/room/Delete>
4. Android Studio (2024, June). *Insert*. <https://developer.android.com/reference/androidx/room/Insert>
5. Android Studio (2024, June). *Update*. <https://developer.android.com/reference/androidx/room/Update>
6. Android Studio (2024, October). *Save data in a local database using Room*. <https://developer.android.com/training/data-storage/room>

7. Android Studio (2024, June). *Entity*. <https://developer.android.com/reference/androidx/room/Entity>
8. Android Studio (2024, July). *ViewModel Overview*. <https://developer.android.com/topic/libraries/architecture/viewmodel>
9. Android Studio (2024, July). *LiveData Overview*. <https://developer.android.com/topic/libraries/architecture/livedata>
10. Android Studio (2024, April). *Dao*. <https://developer.android.com/reference/androidx/room/Dao>
11. Muntenescu, F. (2023, September). *Android Room with a View - Kotlin*. <https://developer.android.com/codelabs/android-room-with-a-view-kotlin#0>