SWINBURNE VIETNAM

HO CHI MINH CAMPUS

CLASS: COS30017

Assignment 3 Report

INSTRUCTOR: Dr. Tan Le

STUDENT NAMES: Huu Nhan Le - 1040711133

HO CHI MINH CITY – April, 4th 2025

1. Acknowledgement

No Generative AI tools were used for this task

2. Introduction

In assignment 3, I decided to design a password manager app for mobile devices. The app will store the list of password entries (including details like name, username, password, url, note) in a local SQLite database and display it in several activities. The user can also delete, edit, and add a new entry in the database. This report will go over UI/UX drafts for my user stories, some important app design choices, thorough description of the UI/UX, functionalities, and the Espresso tests. A reflection section will be included at the end of the report to provide a general overview of what I believe was done successfully on this project and what could be improved.

3. Time Logs

Date	Time	Description
26/03	3 hours	Create a UI/UX prototype based on user stories
26/03 - 27/03	8 hours	Design the layout for MainActivity
27/03	3 hours	Design the layout for ViewLoginActivity
28/03	3 hours	Design the layout for NewLoginActivity
28/03	4 hours	Add the Password data class
29/03 - 31/03	2 days	Implement the Room library to use SQLite for this app
31/03	5 hours	Implement the necessary functions in MainActivity to display the list of passwords
01/04	5 hours	Implement the necessary functions in ViewLoginActivity to display the details of each password entry
01/04 - 02/04	1 day	Implement the necessary functions in NewLoginActivity to enter and validate information of the new password entry
02/04	2 hours	Design the layout for EditLoginActivity
03/04	3 hours	Implement the necessary functions in EditLoginActivity to change the details of existing password entry
03/04	6 hours	Add 2 Espresso tests to test the UI and functionalities of the app
04/04	1 day	Fix any remaining bugs

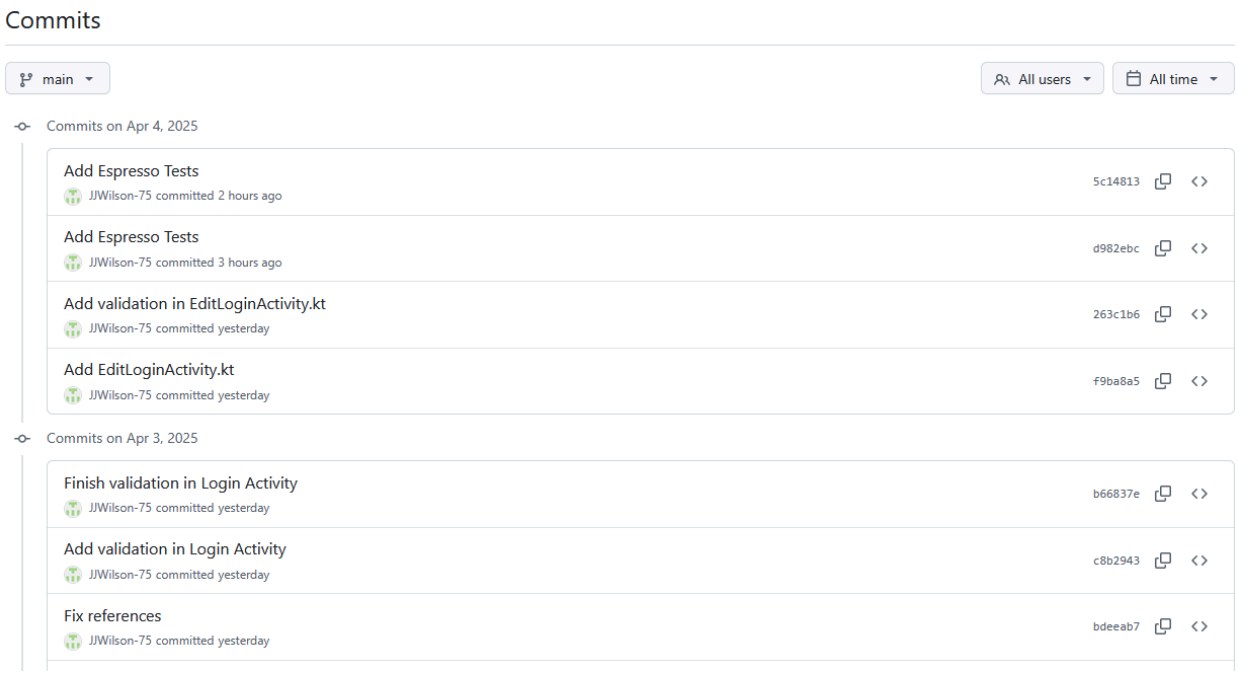

Figure 1 - Commit history on Github Classroom for Assignment 3

4. UI/UX

4.1. User stories

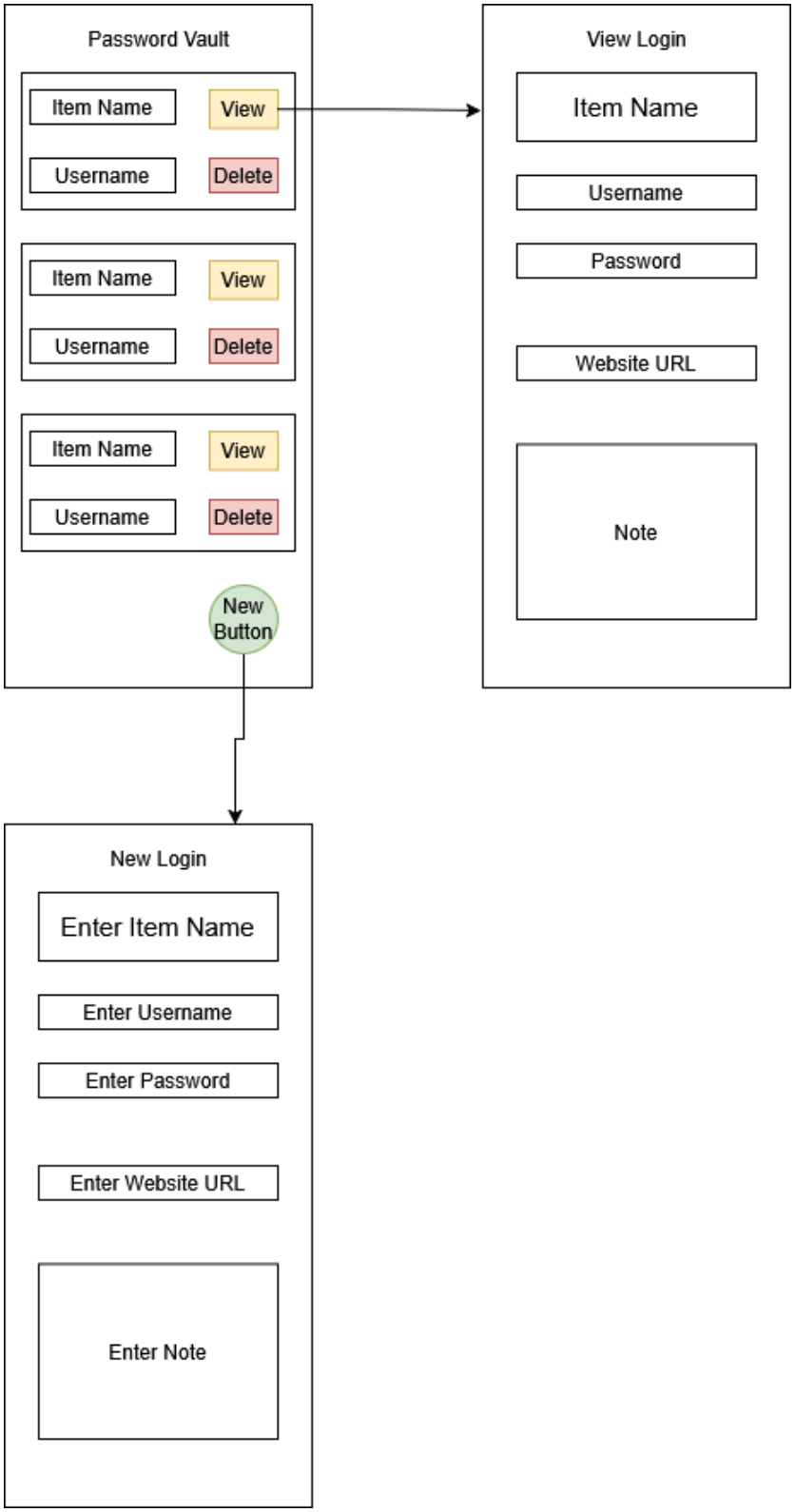

Figure 2 - Sketches of the layouts for 3 activities in the app

User story 1: As a old woman, I need to save the passwords somewhere because I can't possibly remember the details of all the passwords I use for different website

Use case: To satisfy the requirements of this user story, I have 2 activities in the design draft. The first one, MainActivity, will display all the passwords saved. If the user wants to see more information about the password entry (rather than just the name and username), she can click on the View button for that entry to get transitioned into ViewLoginActivity. In this activity, there are 4 main rows for item name, username, password, and website url and one large box for additional notes.

User story 2: As an office worker, I want to be able to delete some entries if I don't need to use them anymore

Use case: In the MainActivity, there is a Delete button for each password to allow the user to delete it from the database in the app. And although it isn't shown in the sketch, the user will see a confirmation dialog after clicking the Delete button to prevent accidental deletion, as the entries can't be restored after that.

4.2. MainActivity layouts

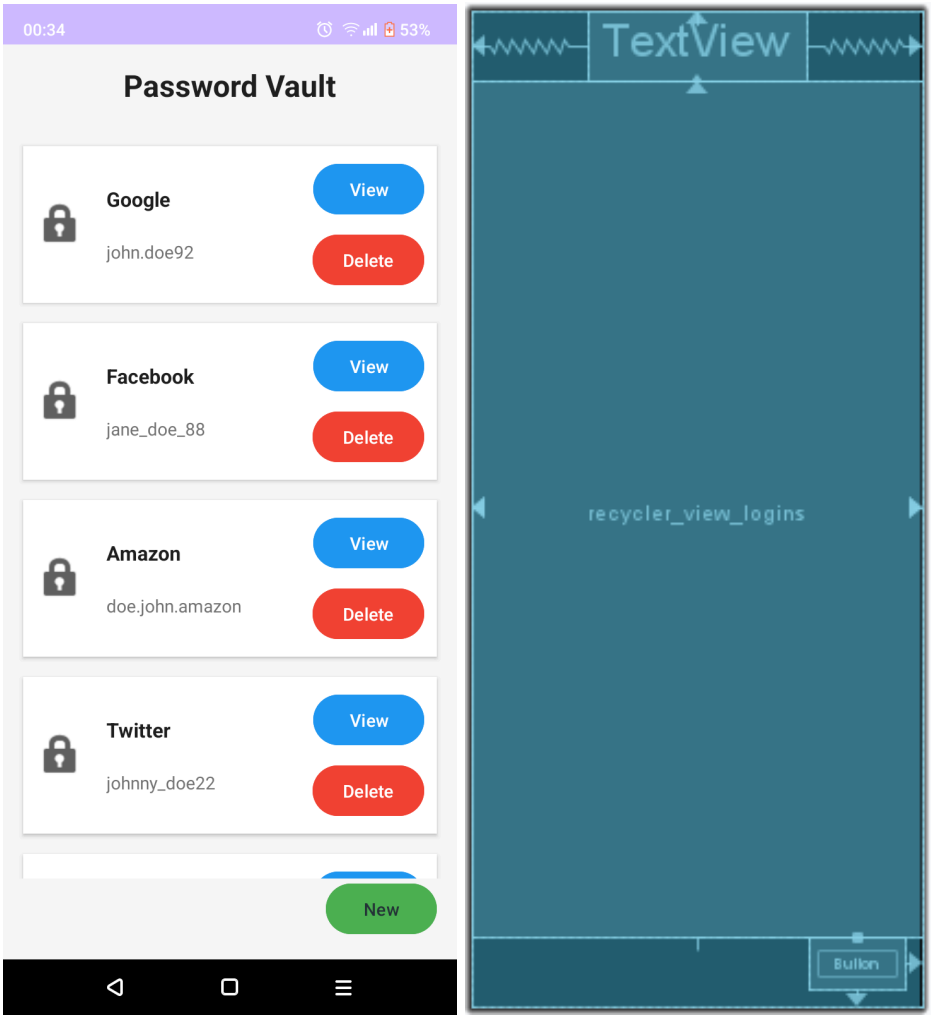

Figure 3 - Layout in Main Activity

In this layout design for the MainActivity, I applied a vertically structured user interface using ConstraintLayout to maintain both alignment and responsiveness across devices. At the top of the screen, there is a TextView element (title_text_view) that acts as the app's header, displaying the title "Password Vault" with bold styling and a relatively large font size (24sp) for visibility and emphasis. Directly beneath it lies a RecyclerView (recycler_view_logins), which serves as the main content area for displaying a scrollable list of saved login credentials. The layout uses constraints to ensure that the RecyclerView stretches horizontally across the screen and vertically occupies all space between the header and the "New" button at the bottom, while the clipToPadding and scrollbars attributes improve user interaction with the list. Finally, aligned to the bottom-end of the screen is a Button (new_button), clearly styled with a green background tint (#4CAF50) and labeled with the text defined in the btn_new_label string resource. This button is intended for adding new credentials, enhancing user engagement by offering an intuitive entry point for new data input.

4.3. ViewLoginActivity layouts

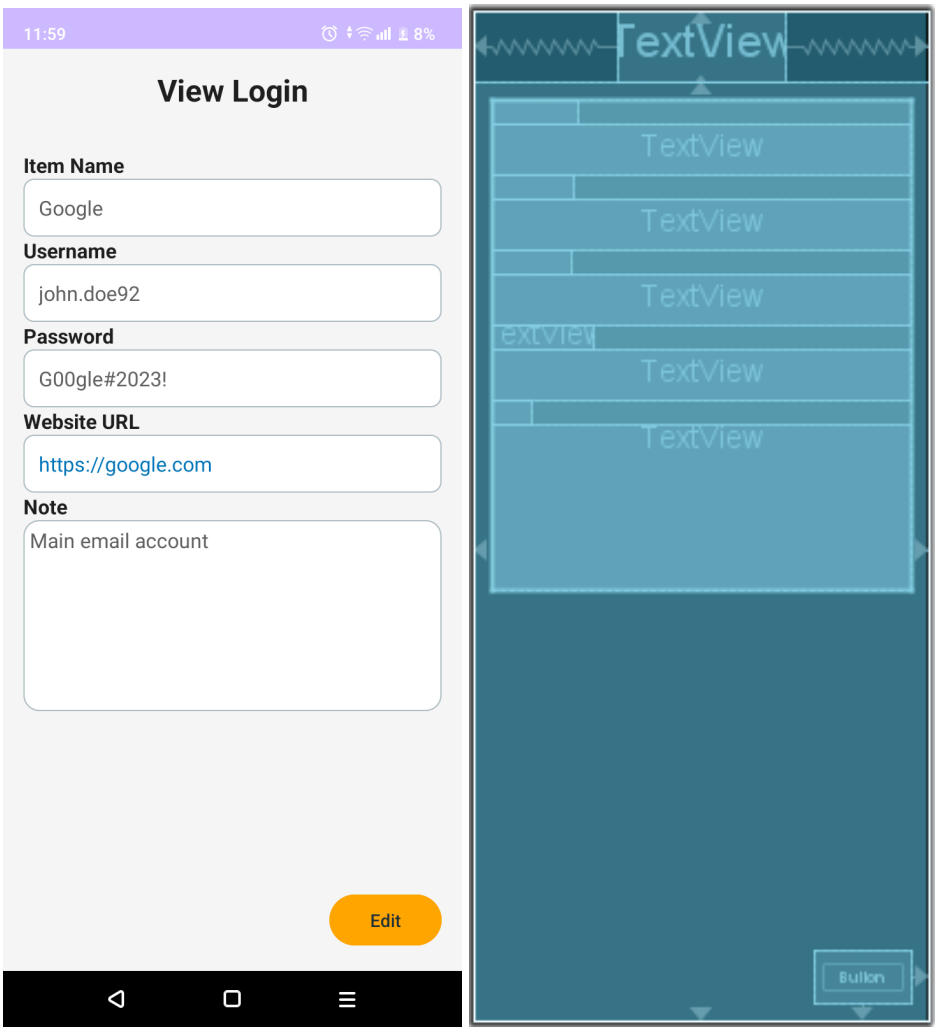

Figure 4 - Layout in ViewLoginActivity

At the top, a bold `TextView` (title_text_view) displays the title "View Login", centered with generous padding to emphasize its importance. Below it, a `ScrollView` wraps the core content, allowing the screen to remain scrollable in case of longer entries. Inside the `ScrollView`, another `ConstraintLayout` contains five neatly stacked sections—each comprising a bold label followed by a corresponding `TextView` for item name, username, password, website URL, and a larger note area—each styled with rounded borders and consistent padding. At the bottom-right corner of the screen, an orange-tinted `Button` (edit_button) allows the user to initiate edits, staying accessible while maintaining a distinct visual cue for interaction.

4.4. NewLoginActivity

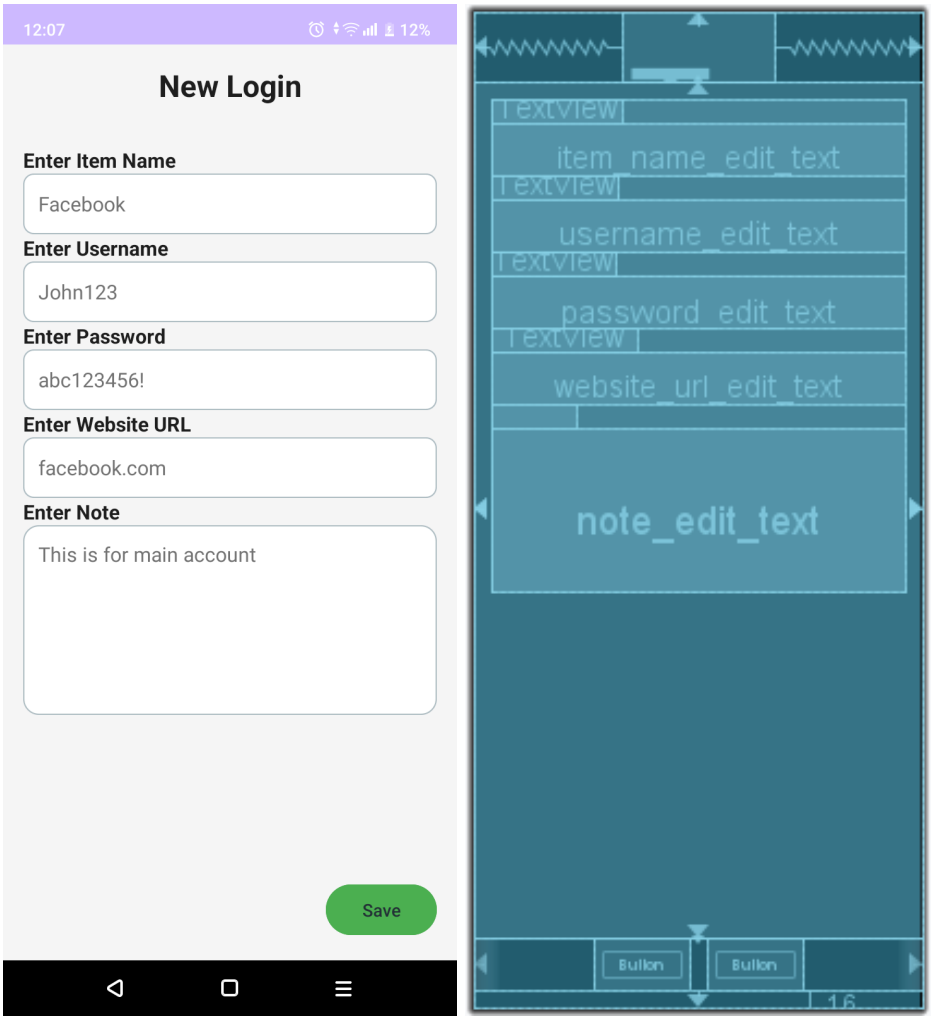

Figure 5 - Layout in EditLoginActivity

This layout for NewLoginActivity presents a clean, form-based interface for adding new login entries. At the top, a bold TextView (title_text_view) introduces the screen with the title “New Login,” centered and padded for prominence. Below it, a ScrollView ensures that all input fields remain accessible on smaller screens. Inside the scrollable area, a structured ConstraintLayout organizes five input sections—each consisting of a descriptive label followed by an EditText field for entering the item name, username, password, website URL, and an optional note. These fields are styled with rounded borders, soft padding, and intuitive hint text to guide the user. Finally, anchored at the bottom-right corner of the screen, a green-tinted Button (save_button) provides a clear action point to save the new login, making the interface functional and user-friendly.

4.5. EditLoginActivity

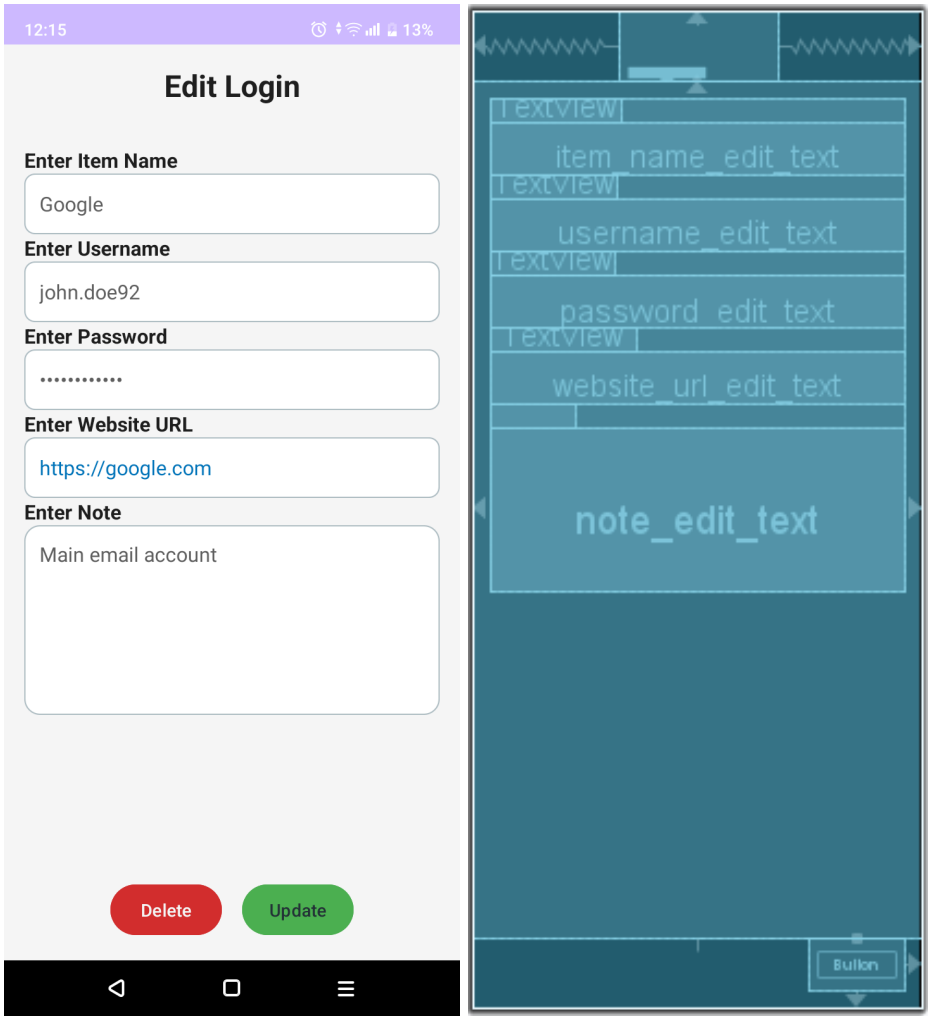

Figure 6 - Layout in EditLoginActivity

The activity_edit_login.xml layout is designed for editing existing login details, featuring a ScrollView containing labeled EditText fields for item name, username, password, website URL, and notes, all inside a ConstraintLayout for flexible alignment. It differs from activity_new_login.xml, which likely includes similar fields but is focused on creating new entries (thus missing the "Delete" button), and from activity_view_login.xml, which displays information in a non-editable format. This edit layout uniquely includes both Update and Delete buttons, enabling full control over modifying or removing login data.

4.6. Item_Login RecyclerView

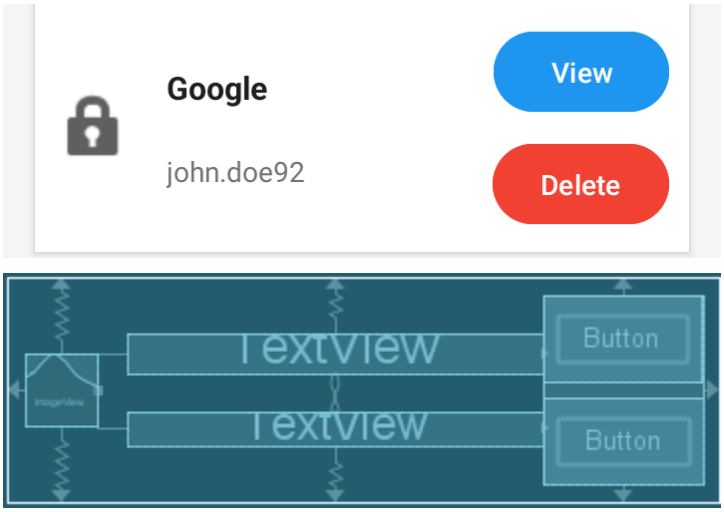

Figure 7 - item_login.xml RecyclerView

The item_login.xml layout defines a single item view used within a RecyclerView to display saved login entries in a scrollable list. Each item includes an icon, the item name, and username, along with two vertically stacked buttons—View and Delete—for interacting with that login. The RecyclerView efficiently reuses these item views using an adapter, which binds data (like login details) to the layout elements (TextViews and Buttons) as users scroll (*Create Dynamic Lists with RecyclerView*, n.d.). This reuse improves performance and memory usage, especially with large datasets (*Create Dynamic Lists with RecyclerView*, n.d.).

4.7. Architecture diagram

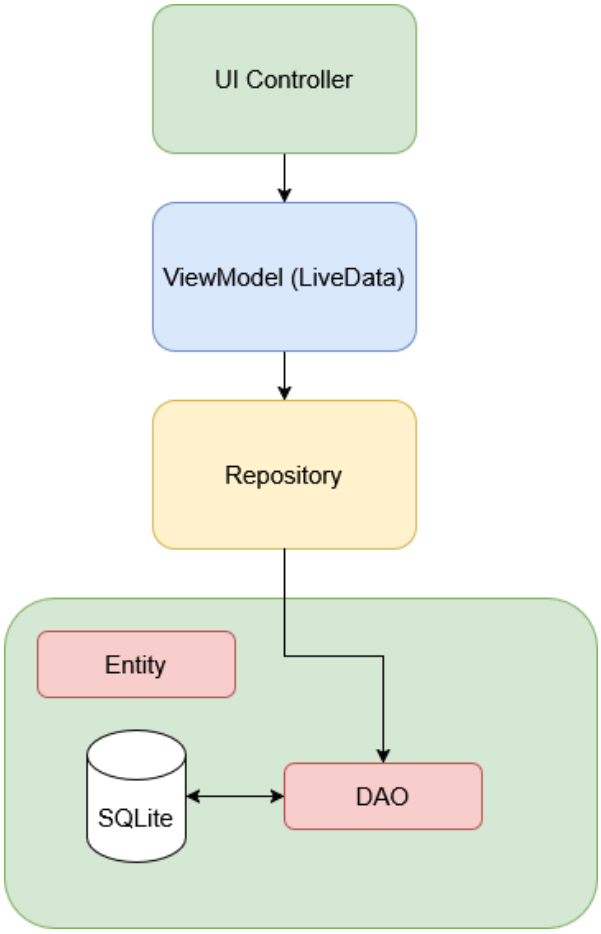

Figure 8 - Architecture diagram of this app

The architecture design of this app follows the Model-View-ViewModel (MVVM) pattern, as depicted in the diagram. At the top, the UI Controller (implemented by activities like NewLoginActivity and EditLoginActivity) handles user interactions and updates the UI through data binding. It communicates with the ViewModel (PasswordViewModel), which manages the app's live data and business logic, ensuring the UI remains reactive to data changes. The ViewModel interacts with the Repository, which acts as a single source of truth for data operations, abstracting the data layer. The Repository, in turn, uses a Data Access Object (DAO) to perform CRUD operations on the SQLite database, with the Entity representing the data model (e.g., the Password class). This layered architecture ensures separation of concerns, making the app modular, testable, and maintainable (*Room, LiveData, and ViewModel*, 2025).

5. SQLite database with Room

5.1. Instrument data class

```
@Entity(tableName = "password")
data class Password(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val itemName: String,
    val username: String,
    val password: String,
    val websiteUrl: String,
    val note: String
)
```

Figure 9 - Password data class

The Password data class is used as a model to represent a single login credential stored in the Room database, with five key properties: itemName, username, password, websiteUrl, and note, all of String type. It also includes a primary key id of Int type, annotated with @PrimaryKey(autoGenerate = true) to ensure each entry gets a unique, auto-incremented identifier upon insertion. This class is annotated with @Entity(tableName = "password") to indicate that it maps to a table named password in the local SQLite database managed by Room (*Defining Data Using Room Entities | Android Developers*, 2020). All properties are declared with val, ensuring immutability after creation, which aligns with Room's recommendation for defining data entities (*Defining Data Using Room Entities | Android Developers*, 2020).

5.2. DAO

```kotlin
@Dao
interface PasswordDao {
    @Query("SELECT * FROM password")
    fun getAllPasswords(): LiveData<List<Password>>

    @Query("SELECT * FROM password WHERE id = :id")
    fun getPasswordById(id: Int): LiveData<Password>

    @Insert
    suspend fun insert(passwordEntry: Password)

    @Update
    suspend fun update(passwordEntry: Password)

    @Delete
    suspend fun delete(passwordEntry: Password)
}
```

Figure 10 - Data Access Object for Password database

The PasswordDao interface defines the core database operations for managing login entries in the password table using Room (*Accessing Data Using Room DAOs | Android Developers*, 2019). It includes two @Query-annotated methods: getAllPasswords(), which returns all stored login entries as LiveData<List<Password>> for real-time UI observation, and getPasswordById(id: Int), which retrieves a single Password entry by its unique ID. The @Insert, @Update, and @Delete methods are marked as suspend functions, allowing them to perform database operations asynchronously within Kotlin coroutines to avoid blocking the main thread. These annotated methods automatically generate the underlying SQL queries, simplifying CRUD (Create, Read, Update, Delete) operations without manual implementation (*Accessing Data Using Room DAOs | Android Developers*, 2019).

5.3. AppDatabase

```kotlin
@Database(entities = [Password::class], version = 2, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract fun passwordDao(): PasswordDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context, scope: CoroutineScope): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "password_database"
                )
                    .addCallback(PasswordDatabaseCallback(scope)) // Add this only if needed, may delete later
                    .build()
                INSTANCE = instance
                instance
            }
        }
    }
```

Figure 11 - AppDatabase class part 1

The AppDatabase.kt file defines a Room database class named AppDatabase, which serves as the main access point to the local SQLite database used in the application (Google, 2019). The abstract function passwordDao() provides access to the DAO (PasswordDao), enabling all database operations such as insertion, deletion, updates, and queries on the password table.

```kotlin
// Callback to pre-populate the database
private class PasswordDatabaseCallback(
    private val scope: CoroutineScope
) : RoomDatabase.Callback() {

    override fun onCreate(db: SupportSQLiteDatabase) {
        super.onCreate(db)
        INSTANCE?.let { database ->
            scope.launch(Dispatchers.IO) {
                populateDatabase(database.passwordDao())
            }
        }
    }

    // Method to insert initial data
    private suspend fun populateDatabase(passwordDao: PasswordDao) {
        // Insert 10 sample password entries
        passwordDao.insert(Password(itemName = "Google", username = "john.doe92", password = "G00gle#2023!", websiteUrl = "https://google.com", note = "Main email account"))
        passwordDao.insert(Password(itemName = "Facebook", username = "jane_doe_88", password = "F@ceB00k$88", websiteUrl = "https://facebook.com", note = "Social media account"))
        passwordDao.insert(Password(itemName = "Amazon", username = "doe.john.amazon", password = "Am@z0n_789!", websiteUrl = "https://amazon.com", note = "Shopping account"))
        passwordDao.insert(Password(itemName = "Twitter", username = "johnny_doe22", password = "Tw!tt3r#2022", websiteUrl = "https://twitter.com", note = "Microblogging account"))
        passwordDao.insert(Password(itemName = "LinkedIn", username = "jane.doe.prof", password = "L!nk3dIn@2023", websiteUrl = "https://linkedin.com", note = "Professional network"))
        passwordDao.insert(Password(itemName = "GitHub", username = "johndoe_dev", password = "G!tHub#303$", websiteUrl = "https://github.com", note = "Code repository"))
        passwordDao.insert(Password(itemName = "Netflix", username = "doe_family", password = "N3tfl!x_404#", websiteUrl = "https://netflix.com", note = "Streaming service"))
        passwordDao.insert(Password(itemName = "Spotify", username = "jane_musiclover", password = "Sp0t!fy$505", websiteUrl = "https://spotify.com", note = "Music streaming"))
        passwordDao.insert(Password(itemName = "PayPal", username = "john.doe.payment", password = "P@yP@l_606!", websiteUrl = "https://paypal.com", note = "Payment account"))
        passwordDao.insert(Password(itemName = "Dropbox", username = "doe.storage", password = "Dr0pB0x#707$", websiteUrl = "https://dropbox.com", note = "Cloud storage"))
    }
}
```

Figure 12 - AppDatabase class part 2

A key feature in this class is the PasswordDatabaseCallback, an inner class extending RoomDatabase.Callback (*Kotlin Coroutines on Android*, n.d.). It leverages coroutine support via CoroutineScope to insert default password entries when the database is first created. The onCreate() override launches a coroutine in the IO dispatcher and calls populateDatabase(), a suspend function that inserts ten pre-defined login credentials for popular platforms like Google, Facebook, Amazon, and others. This is particularly useful for demonstration, testing, or providing users with a pre-filled sample upon first installation.

5.4. Repository

```
class PasswordRepository(private val passwordDao: PasswordDao) {

    val allPasswords: LiveData<List<Password>> = passwordDao.getAllPasswords()

    fun getPasswordById(id: Int): LiveData<Password> {
        return passwordDao.getPasswordById(id)
    }

    suspend fun insert(passwordEntry: Password) {
        passwordDao.insert(passwordEntry)
    }

    suspend fun update(passwordEntry: Password) {
        passwordDao.update(passwordEntry)
    }

    suspend fun delete(passwordEntry: Password) {
        passwordDao.delete(passwordEntry)
    }
}
```

Figure 13 - PasswordRepository class

The PasswordRepository class acts as a mediator between the data access layer (via PasswordDao) and the rest of the application. It provides a clean API for accessing and manipulating password data, abstracting away the complexities of direct database operations (Caner Gures, 2020). The class exposes a LiveData list of all passwords (allPasswords) which allows the UI to observe changes to the data in real-time. It also provides methods for retrieving a password by its ID (getPasswordById), and suspending functions to insert, update, or delete password entries (insert, update, delete). As claimed by Caner Gures (2020), by encapsulating database operations in this repository, the class helps maintain separation of concerns and facilitates easier testing and maintenance of the codebase.

5.5. View Model

```
class PasswordViewModel(application: Application) : AndroidViewModel(application) {

    private val repository: PasswordRepository
    val allPasswords: LiveData<List<Password>>

    init {
        val passwordDao = AppDatabase.getDatabase(application, viewModelScope).passwordDao()
        repository = PasswordRepository(passwordDao)
        allPasswords = repository.allPasswords
    }

    fun getPasswordById(id: Int): LiveData<Password> {
        return repository.getPasswordById(id)
    }

    fun insert(passwordEntry: Password) = viewModelScope.launch {
        repository.insert(passwordEntry)
    }

    fun update(passwordEntry: Password) = viewModelScope.launch {
        repository.update(passwordEntry)
    }

    fun delete(passwordEntry: Password) = viewModelScope.launch {
        repository.delete(passwordEntry)
    }
}
```

Figure 14 - PasswordRepository class

The PasswordViewModel class is a part of the ViewModel architecture, providing a layer of abstraction between the UI and the repository, and it manages UI-related data in a lifecycle-conscious way (Caner Gures, 2020). It is initialized with an application context and uses the PasswordRepository to interact with the database. The allPasswords property exposes a LiveData list of passwords, allowing the UI to observe changes in real-time. The getPasswordById method retrieves a specific password by its ID. The insert, update, and delete methods are launched within the viewModelScope, ensuring that these operations run asynchronously on a background thread, thereby keeping the UI responsive.

5.6. Adapter for RecyclerView

```
class PasswordAdapter(
    private val onViewClick: (Password) -> Unit,
    private val onDeleteClick: (Password) -> Unit
) : RecyclerView.Adapter<PasswordAdapter.PasswordViewHolder>() {

    private var passwordList: List<Password> = emptyList()

    class PasswordViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val itemNameTextView: TextView = itemView.findViewById(R.id.item_name_text_view)
        val usernameTextView: TextView = itemView.findViewById(R.id.username_text_view)
        val viewButton: Button = itemView.findViewById(R.id.view_button)
        val deleteButton: Button = itemView.findViewById(R.id.delete_button)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): PasswordViewHolder {
        val view = LayoutInflater.from(parent.context).inflate(R.layout.item_login, parent, false)
        return PasswordViewHolder(view)
    }

    override fun onBindViewHolder(holder: PasswordViewHolder, position: Int) {
        val passwordEntry = passwordList[position]
        holder.itemNameTextView.text = passwordEntry.itemName
        holder.usernameTextView.text = passwordEntry.username
        holder.viewButton.setOnClickListener { onViewClick(passwordEntry) }
        holder.deleteButton.setOnClickListener { onDeleteClick(passwordEntry) }
    }

    override fun getItemCount(): Int = passwordList.size

    fun submitList(newList: List<Password>) {
        passwordList = newList
        notifyDataSetChanged()
    }
}
```

Figure 15 - PasswordAdapter class

The PasswordAdapter class is a custom RecyclerView.Adapter that binds a list of Password objects to a RecyclerView (*Create Dynamic Lists with RecyclerView*, n.d.). It accepts two lambda functions as parameters: onViewClick and onDeleteClick, which are invoked when the corresponding buttons (view and delete) in each list item are clicked. The adapter uses a PasswordViewHolder class to reference the views within each item layout, such as the item name, username, and the action buttons. The onCreateViewHolder method inflates the

item_login layout, while the onBindViewHolder method binds the data from the Password object to the respective views, and sets up the click listeners for the buttons.
The submitList method is used to update the data in the adapter and notify the RecyclerView of the changes by calling notifyDataSetChanged(), ensuring the UI reflects any updates made to the list of passwords. Meanwhile, the getItemCount method returns the size of the list, ensuring that the RecyclerView knows how many items it needs to display.

6. Functionality

6.1. Data Binding

In the `MainActivity`, `ViewLoginActivity`, `NewLoginActivity`, and `EditLoginActivity` classes, data binding is utilized to directly link UI components in the XML layouts to corresponding variables in the code (*Data Binding Library | Android Developers*, 2019). For instance, in the `MainActivity`, the `ActivityMainBinding` object is used to access and manipulate the views defined in the `activity_main.xml` layout, such as `recyclerViewLogins` and `newButton`, without the need to use `findViewById`. This makes the code more concise and readable. Similarly, in other activities like `ViewLoginActivity` and `EditLoginActivity`, `ActivityViewLoginBinding` and `ActivityEditLoginBinding` provide a direct reference to the layout components, allowing easy manipulation of UI elements like text fields and buttons. This approach simplifies view access and enhances maintainability, as it reduces boilerplate code and eliminates the risk of `NullPointerExceptions` by ensuring that views are properly bound to the layout (*Data Binding Library | Android Developers*, 2019).

6.2. View details of password

```
val passwordId = intent.getIntExtra("PASSWORD_ID", -1)
if (passwordId != -1) {
    passwordViewModel.getPasswordById(passwordId).observe(this) { passwordEntry ->
        passwordEntry?.let {
            binding.itemNameTextView.text = it.itemName
            binding.usernameTextView.text = it.username
            binding.passwordTextView.text = it.password // Or mask it as "********"
            binding.websiteUrlTextView.text = it.websiteUrl
            binding.noteTextView.text = it.note
        }
    }
}
```

Figure 16 - Code block to display password entry's information

In the ViewLoginActivity, the details of a password entry are displayed by first retrieving the password's ID from the intent that started the activity. This ID is used to fetch the corresponding password entry from the PasswordViewModel via the getPasswordById() method. The passwordViewModel.getPasswordById(passwordId).observe() function is called, which

observes changes to the password entry. Once the password entry is fetched, the binding object is used to directly update the UI elements (like itemNameTextView, usernameTextView, passwordTextView, websiteUrlTextView, and noteTextView) with the values from the password entry object.

6.3. Edit password entries

```
if (passwordId != -1) {
    passwordViewModel.getPasswordById(passwordId).observe(this) { passwordEntry ->
        passwordEntry?.let {
            binding.itemNameEditText.setText(it.itemName)
            binding.usernameEditText.setText(it.username)
            binding.passwordEditText.setText(it.password)
            binding.websiteUrlEditText.setText(it.websiteUrl)
            binding.noteEditText.setText(it.note)
        }
    }
}

// Update password entry when clicking the update button
binding.updateButton.setOnClickListener {
    val updatedItemName = binding.itemNameEditText.text.toString()
    val updatedUsername = binding.usernameEditText.text.toString()
    val updatedPassword = binding.passwordEditText.text.toString()
    val updatedWebsiteUrl = binding.websiteUrlEditText.text.toString()
    val updatedNote = binding.noteEditText.text.toString()

    if (validateInputs(updatedItemName, updatedUsername, updatedPassword, updatedWebsiteUrl)) {
        val updatedPasswordEntry = Password(
            id = passwordId,
            itemName = updatedItemName,
            username = updatedUsername,
            password = updatedPassword,
            websiteUrl = updatedWebsiteUrl,
            note = updatedNote
        )

        passwordViewModel.update(updatedPasswordEntry)
        Toast.makeText(this, "Updated Successfully", Toast.LENGTH_SHORT).show()
        finish()
    }
}
```

Figure 17 - Code block to edit password entry's information

In EditLoginActivity, the user is allowed to edit an existing password entry by first retrieving the password ID passed via the intent (passwordId = intent.getIntExtra("PASSWORD_ID", -1)). Once the ID is obtained, the passwordViewModel.getPasswordById(passwordId) function is called to fetch the corresponding password entry from the database. The data binding mechanism updates the UI by setting the EditText fields (such as itemNameEditText, usernameEditText, passwordEditText, websiteUrlEditText, and noteEditText) with the existing values from the fetched password entry.
The user can then modify the values in these fields. When the "Update" button is clicked, the app gathers the new values from the EditText fields and checks them using the validateInputs() method, ensuring that the inputs meet the validation criteria. If the inputs are valid, a new Password object is created with the updated values, and the passwordViewModel.update(updatedPasswordEntry) method is called to save the changes to the database. A toast message is shown to confirm the update, and the activity is closed by calling finish().

6.4. Create a new password entry

```
binding.saveButton.setOnClickListener {
    val itemName = binding.itemNameEditText.text.toString()
    val username = binding.usernameEditText.text.toString()
    val password = binding.passwordEditText.text.toString()
    val websiteUrl = binding.websiteUrlEditText.text.toString()
    val note = binding.noteEditText.text.toString()

    if (validateInputs(itemName, username, password, websiteUrl)) {
        val passwordEntry = Password(
            itemName = itemName,
            username = username,
            password = password,
            websiteUrl = websiteUrl,
            note = note
        )
        passwordViewModel.insert(passwordEntry)
        finish() // Return to MainActivity
    }
}
```

Figure 18 - Code block to create new password entry

In NewLoginActivity, the user can add a new password entry by filling out fields such as item name, username, password, website URL, and note. When the "Save" button is clicked, the app collects the input values from these fields, then validates them using the validateInputs() method to ensure they meet the required criteria (e.g., non-empty fields, valid username, password, and website URL). If the inputs are valid, a new Password object is created with the collected data, and it is inserted into the database via passwordViewModel.insert(passwordEntry). After

successfully saving the new entry, the activity finishes, returning the user to the MainActivity where the updated password list is displayed.

6.5. Delete password entries

```
onDeleteClick = { passwordEntry ->
    showDeleteConfirmationDialog(passwordEntry)
}
```

Figure 19 - Delete button implementation in MainActivity

```
private fun showDeleteConfirmationDialog(passwordEntry: Password) {
    AlertDialog.Builder(this)
        .setTitle("Confirm Deletion")
        .setMessage("Are you sure you want to delete this password?")
        .setPositiveButton("Delete") { _, _ ->
            passwordViewModel.delete(passwordEntry) // Proceed with deletion
        }
        .setNegativeButton("Cancel", null) // Dismiss dialog
        .show()
}
```

Figure 20 - Delete confirmation dialog in MainActivity

In MainActivity, when the user clicks the delete button associated with a password entry (handled by onDeleteClick in the PasswordAdapter), the method showDeleteConfirmationDialog(passwordEntry) is triggered. This method creates an AlertDialog to ask the user for confirmation before proceeding with the deletion. The dialog contains two buttons: a "Delete" button and a "Cancel" button.
If the user confirms the deletion by clicking "Delete," the password entry is passed to the passwordViewModel.delete(passwordEntry) function, which handles removing the entry from the database. Once the password is deleted, the RecyclerView (which displays the list of passwords) is updated, and the password entry is removed from the view.

```
// Delete password entry when clicking the delete button
binding.deleteButton.setOnClickListener {
    val passwordToDelete = Password(
        id = passwordId,
        itemName = binding.itemNameEditText.text.toString(),
        username = binding.usernameEditText.text.toString(),
        password = binding.passwordEditText.text.toString(),
        websiteUrl = binding.websiteUrlEditText.text.toString(),
        note = binding.noteEditText.text.toString()
    )

    showDeleteConfirmationDialog(passwordToDelete)
}
```

Figure 21 - Listener for Delete button in ViewLoginActivity

```
private fun showDeleteConfirmationDialog(passwordEntry: Password) {
    AlertDialog.Builder(this)
        .setTitle("Confirm Deletion")
        .setMessage("Are you sure you want to delete this password?")
        .setPositiveButton("Delete") { _, _ ->
            passwordViewModel.delete(passwordEntry) // Proceed with deletion

            // Redirect to MainActivity after deletion
            val intent = Intent(this, MainActivity::class.java)
            intent.flags = Intent.FLAG_ACTIVITY_CLEAR_TOP or Intent.FLAG_ACTIVITY_NEW_TASK
            startActivity(intent)
            finish()
        }
        .setNegativeButton("Cancel", null) // Dismiss dialog
        .show()
}
```

Figure 22 - Delete confirmation dialog in ViewLoginActivity

In EditLoginActivity, the process is similar, but it involves the user deciding whether to delete the password while viewing or editing the details of a specific entry. If the user clicks the "Delete" button, an AlertDialog is shown to confirm their intention to delete the password. This confirmation dialog works the same way as in MainActivity: the dialog asks the user if they are sure about deleting the entry.

Upon confirmation, the password entry is passed to the passwordViewModel.delete(passwordEntry) function, which deletes the entry from the database. After the deletion, the app redirects the user back to MainActivity using an Intent with the flags Intent.FLAG_ACTIVITY_CLEAR_TOP and Intent.FLAG_ACTIVITY_NEW_TASK, ensuring that MainActivity is refreshed, and the password entry is removed from the list.

6.6. Data validation

```
private fun validateInputs(itemName: String, username: String, password: String, websiteUrl: String): Boolean {
    if (itemName.isEmpty()) {
        showToast("Item name cannot be empty")
        return false
    }

    if (!isValidUsername(username)) {
        showToast("Username must be alphanumeric and max 10 characters")
        return false
    }

    if (!isValidPassword(password)) {
        showToast("Password must be 8-20 characters, include a letter, a number, and a special character")
        return false
    }

    if (!isValidWebsite(websiteUrl)) {
        showToast("Enter a valid website URL")
        return false
    }

    return true
}

private fun isValidUsername(username: String): Boolean {
    return username.matches(Regex("^[a-zA-Z0-9]{1,10}$"))
}

private fun isValidPassword(password: String): Boolean {
    return password.length in 8 ≤ .. ≤ 20 &&
            password.any { it.isLetter() } &&
            password.any { it.isDigit() } &&
            password.any { "!@#$%^&*()-_=+[]{}|;:'\",.<>?/`~".contains(it) }
}

private fun isValidWebsite(url: String): Boolean {
```

Figure 23 - Code block to validate the data in NewLoginActivity

In NewLoginActivity, data validation is performed before saving a new password entry, ensuring that all inputs are correct and meet specific requirements. When the saveButton is clicked, the values entered by the user are retrieved from the respective input fields (itemNameEditText, usernameEditText, passwordEditText, websiteUrlEditText, and noteEditText). Each of these values is then validated one by one. First, the itemName is checked to ensure it is not empty. If it is, a Toast message is displayed, indicating that the item name cannot be left blank. If the item name passes this check, the validation then proceeds to the username. The username must match a regex pattern (^[a-zA-Z0-9]{1,10}$), which ensures that the username is alphanumeric

and no longer than 10 characters. If the username does not meet this criterion, a Toast message is shown, alerting the user to input a valid username.
Next, the password is validated to confirm it adheres to a set of rules: it must be between 8 and 20 characters long, include at least one letter, one number, and one special character. If the password does not meet these conditions, a Toast message is displayed, informing the user of the password requirements. Additionally, the websiteUrl is validated to ensure it matches a valid URL format using the Patterns.WEB_URL.matcher(url).matches() method. If the URL is invalid, the user is shown a Toast message prompting them to enter a valid URL. Only when all these checks are passed does the app proceed to create a Password object and save the entry using the passwordViewModel.insert(passwordEntry) method. This ensures that invalid or incomplete entries are not saved, thus preserving the integrity of the password data.

```
private fun validateInputs(itemName: String, username: String, password: String, websiteUrl: String): Boolean {
    if (itemName.isEmpty()) {
        showToast("Item name cannot be empty")
        return false
    }

    if (!isValidUsername(username)) {
        showToast("Username must be alphanumeric and max 10 characters")
        return false
    }

    if (!isValidPassword(password)) {
        showToast("Password must be 8-20 characters, include a letter, a number, and a special character")
        return false
    }

    if (!isValidWebsite(websiteUrl)) {
        showToast("Enter a valid website URL")
        return false
    }

    return true
}

private fun isValidUsername(username: String): Boolean {
    return username.matches(Regex("^[a-zA-Z0-9]{1,10}$"))
}

private fun isValidPassword(password: String): Boolean {
    return password.length in 8 ≤ .. ≤ 20 &&
            password.any { it.isLetter() } &&
            password.any { it.isDigit() } &&
            password.any { "!@#$%^&*()-_=+[]{}|;:'\",.<>?/`~".contains(it) }
}

private fun isValidWebsite(url: String): Boolean {
    return Patterns.WEB_URL.matcher(url).matches()
```

Figure 24 - Code block to validate the data in EditLoginActivity

In EditLoginActivity, a similar validation process is applied when the user attempts to update an existing password entry. Upon clicking the updateButton, the app retrieves the updated values from the input fields and validates them. First, the itemName is checked to ensure it is not empty. If empty, a Toast message is displayed to prompt the user to fill it in. Next, the username is validated using the same regex pattern as in NewLoginActivity, ensuring it is alphanumeric and no longer than 10 characters. If the username does not meet the requirement, an appropriate error message is shown to the user. Following the username validation, the password is verified to meet the required conditions (8-20 characters, a combination of letters, numbers, and special characters). If it does not comply, a Toast message is displayed detailing the password rules. The final validation step checks if the websiteUrl is a valid URL format. If any validation step fails, a relevant Toast message is shown, and the password entry is not updated until all conditions are met.

7. Espresso Testing

6.1. MainActivityAndViewLoginActivityTest

```
fun mainActivityAndViewLoginActivityTest() {
    onView(allOf(withId(R.id.item_name_text_view), withText("Google"))).check(matches(isDisplayed()))

    onView(allOf(withId(R.id.username_text_view), withText("john.doe92"))).check(matches(isDisplayed()))

    onView(allOf(withId(R.id.item_name_text_view), withText("Twitter"))).check(matches(isDisplayed()))

    onView(allOf(withId(R.id.username_text_view), withText("johnny_doe22"))).check(matches(isDisplayed()))

    //Special usage to select the button in item_login with the item_name_text_view Facebook
    onView(allOf(
        withId(R.id.view_button),
        isDescendantOfA(withItemContainingText(R.id.item_name_text_view, "Facebook"))
    )).perform(click())

    //Part 2 Check the details when click View
    onView(withId(R.id.item_name_text_view)).check(matches(withText("Facebook")))

    onView(withId(R.id.username_text_view)).check(matches(withText("jane_doe_88")))

    onView(withId(R.id.password_text_view)).check(matches(withText("F@ce800k$88")))

    onView(withId(R.id.website_url_text_view)).check(matches(withText("https://facebook.com")))

    onView(withId(R.id.note_text_view)).check(matches(withText("Social media account")))
}

private fun withItemContainingText(textViewId: Int, text: String): Matcher<View> {
    return object : BoundedMatcher<View, View>(View::class.java) {
        override fun describeTo(description: Description?) {
            description?.appendText("is an item view containing a TextView with text: $text")
        }

        override fun matchesSafely(item: View?): Boolean {
            if (item == null) return false
            val textView = item.findViewById<View>(textViewId) as? TextView
```

Figure 25 - MainActivityAndViewLoginActivityTest Espresso Test

In the first Espresso test, I decided to just check if all values of the password database are correctly displayed in MainActivity and ViewLoginActivity. Firstly I just checked the item name and the user name of 2 different password entries when MainActivity started up. Next I need to click the View button of the “Facebook” password entry. However, testing the UI/UX of RecyclerView is not supported by default here, so I must design some custom function at the

bottom to achieve this. Then, in ViewLoginActivity, all 5 attributes (item name, username, password, website url, note) of a password item in the SQLite database are tested to see if they match what displayed on the device

6.2. NewLoginActivityTest

```
@Test
fun newLoginActivityTest() {
    // Click "New" button to start creating a new login
    onView(withId(R.id.new_button)).perform(click())

    // Enter details in the form
    onView(withId(R.id.item_name_edit_text)).perform(replaceText("Fanfiction"), closeSoftKeyboard())
    onView(withId(R.id.item_name_edit_text)).perform(pressImeActionButton())

    onView(withId(R.id.username_edit_text)).perform(replaceText("JohnCal1"), closeSoftKeyboard())
    onView(withId(R.id.username_edit_text)).perform(pressImeActionButton())

    onView(withId(R.id.password_edit_text)).perform(replaceText("abc123456!"), closeSoftKeyboard())
    onView(withId(R.id.password_edit_text)).perform(pressImeActionButton())

    onView(withId(R.id.website_url_edit_text)).perform(replaceText("fanfiction.net"), closeSoftKeyboard())
    onView(withId(R.id.website_url_edit_text)).perform(pressImeActionButton())

    onView(withId(R.id.note_edit_text)).perform(replaceText("Post fictional stories"), closeSoftKeyboard())

    // Save the new login
    onView(withId(R.id.save_button)).perform(click())

    // Scroll to the "Fanfiction" item in the RecyclerView
    onView(withId(R.id.recycler_view_logins)).perform(
        RecyclerViewActions.scrollTo<androidx.recyclerview.widget.RecyclerView.ViewHolder>(
            hasDescendant(allOf(withId(R.id.item_name_text_view), withText("Fanfiction")))
        )
    )
```

Figure 26 - NewLoginActivityTest Espresso Test part 1

```
    //Special usage to select the button in item_login with the item_name_text_view Facebook
    onView(allOf(
        withId(R.id.view_button),
        isDescendantOfA(withItemContainingText(R.id.item_name_text_view, "Fanfiction"))
    )).perform(click())

    onView(withId(R.id.item_name_text_view)).check(matches(withText("Fanfiction")))

    onView(withId(R.id.username_text_view)).check(matches(withText("JohnCal1")))

    onView(withId(R.id.password_text_view)).check(matches(withText("abc123456!")))

    onView(withId(R.id.website_url_text_view)).check(matches(withText("fanfiction.net")))

    onView(withId(R.id.note_text_view)).check(matches(withText("Post fictional stories")))
}

private fun withItemContainingText(textViewId: Int, text: String): Matcher<View> {
    return object : BoundedMatcher<View, View>(View::class.java) {
        override fun describeTo(description: Description?) {
            description?.appendText("is an item view containing a TextView with text: $text")
        }

        override fun matchesSafely(item: View?): Boolean {
            if (item == null) return false
            val textView = item.findViewById<View>(textViewId) as? TextView
            return textView != null && textView.text.toString() == text
        }
    }
}
```

Figure 27 - NewLoginActivityTest Espresso Test part 2

First, the test simulates the user clicking the "New" button in MainActivity, which navigates to NewLoginActivity. Afterward, the test proceeds to enter values into the form fields. It starts by entering the itemName ("Fanfiction") into the item_name_edit_text field, followed by the username ("JohnCal1") in the username_edit_text. The password ("abc123456!") is entered next in the password_edit_text, and then the websiteUrl ("fanfiction.net") and note ("Post fictional stories") are filled in. After filling all the fields, the test simulates clicking the "Save" button to save the new login entry.

Next, the test checks if the new entry appears in the RecyclerView that displays saved passwords. Since testing RecyclerView items directly is not supported by default, the test uses the RecyclerViewActions.scrollTo method to scroll to the item with the item_name_text_view that contains the text "Fanfiction." This is where the custom matcher function withItemContainingText comes into play. This function is defined to match any item in the RecyclerView that contains a TextView with the specified text, which in this case is the item name "Fanfiction."

Once the test has scrolled to the correct item, it clicks the button within the item to view its details. The test then verifies that all five attributes of the password entry (item name, username, password, website URL, and note) are correctly displayed in the TextViews on the subsequent screen. If the test passes, it confirms that the new login entry was saved correctly and that the details are displayed properly in the RecyclerView.

8. Reflection

8.1. What worked well

For assignment 3, I think there are a lot of similarities with assignment 2. For instance, I also reused some concepts from assignment 2 for the password manager app like Toast, Intent, Dialog, like how I also utilized many concepts from assignment 1 in assignment 3. And once again, I also needed to research outside a bit to successfully implement the functionality of the app, such as Room library and RecyclerView.

8.2. What could be improved

If I have the chance to improve this app in the future, I want to add some additional features in the app. In Bitwarden, for example, the user can randomize the passwords with some custom limits like at least 10 characters or including special characters. Another feature is the ability to auto-fill the username and password fields in a browser or a mobile app, although this would be immensely challenging to say the least.

9. Github Repository Link

https://github.com/2025-HX01-COS30017-HCM/assignment-3-JJWilson-75

References

1. *Accessing data using Room DAOs | Android Developers*. (2019). Android Developers. https://developer.android.com/training/data-storage/room/accessing-data
2. Caner Gures. (2020, December 6). *Basic Implementation of Room Database With Repository and ViewModel | Android Jetpack*. Medium; The Startup. https://medium.com/swlh/basic-implementation-of-room-database-with-repository-and-viewmodel-android-jetpack-8945b364d322
3. *Create dynamic lists with RecyclerView*. (n.d.). Android Developers. https://developer.android.com/develop/ui/views/layout/recyclerview
4. *Data Binding Library | Android Developers*. (2019). Android Developers. https://developer.android.com/topic/libraries/data-binding
5. *Defining data using Room entities | Android Developers*. (2020). Android Developers. https://developer.android.com/training/data-storage/room/defining-data
6. Google. (2019). *Save data in a local database using Room | Android Developers*. Android Developers. https://developer.android.com/training/data-storage/room
7. *Kotlin coroutines on Android*. (n.d.). Android Developers. https://developer.android.com/kotlin/coroutines
8. *Room, LiveData, and ViewModel*. (2025). Github.io. https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-4-saving-user-data/lesson-10-storing-data-with-room/10-1-c-room-livedata-viewmodel/10-1-c-room-livedata-viewmodel.html