Raymond Bautista

Southern New Hampshire University (SNHU)

CS-499: Computer Science Capstone

March 22, 2026

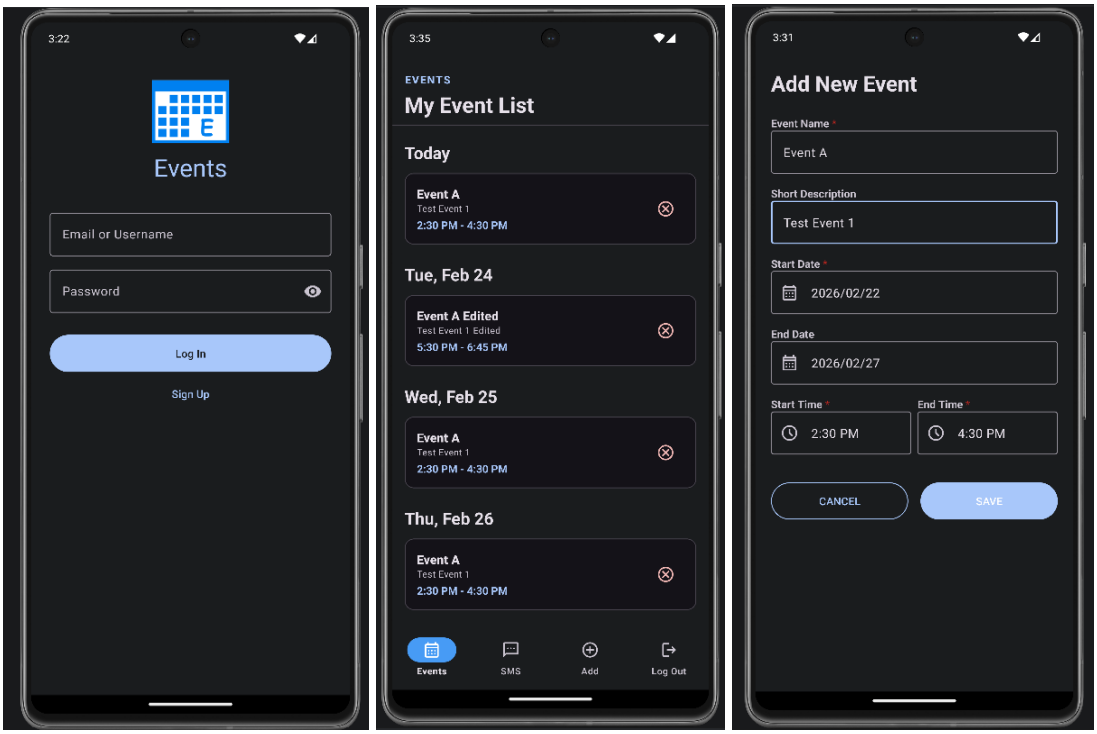
Milestone Two: Software Engineering and Design Enhancement

Artifact Overview

For this category, I selected the Events Android application developed in CS-360: Mobile Architecture and Programming (2026-C1). This project is a user-centered mobile application built using Android Studio, the Java programming language, and a local SQLite database. It emphasizes modern UI/UX best practices, lifecycle-aware architecture, secure authentication, and efficient performance, with the goal of reducing user cognitive load while delivering a smooth and distraction-free experience. The application functions as a personal scheduling hub, presenting events in an agenda-style list grouped by date and sorted in ascending chronological order, enabling users to easily track and manage multi-day commitments.

This project can be considered a full-stack mobile application, as it integrates both frontend and backend components. Key features include secure password storage using hashing and salting, a native database for managing users and events, and event creation and editing through an intuitive form with recurring event generation logic. It also includes a chronologically grouped event dashboard, daily SMS notifications, and the implementation of the Model-View-ViewModel (MVVM) architecture to ensure separation of concerns and maintainable code structure.

I selected this artifact because, in its initial implementation, it included foundational security features such as user authentication and password management using Bcrypt hashing. However, several industry-standard security practices remained unimplemented, including multi-factor authentication, login attempt limitations, password recovery mechanisms, and stateless authentication for secure session management. Additionally, the project presents opportunities to enhance the user experience by applying software design principles and UI improvements, such as introducing a calendar view and grouping past events under a collapsible section. These enhancements aim to further reduce cognitive load and maintain focus on relevant, upcoming information.



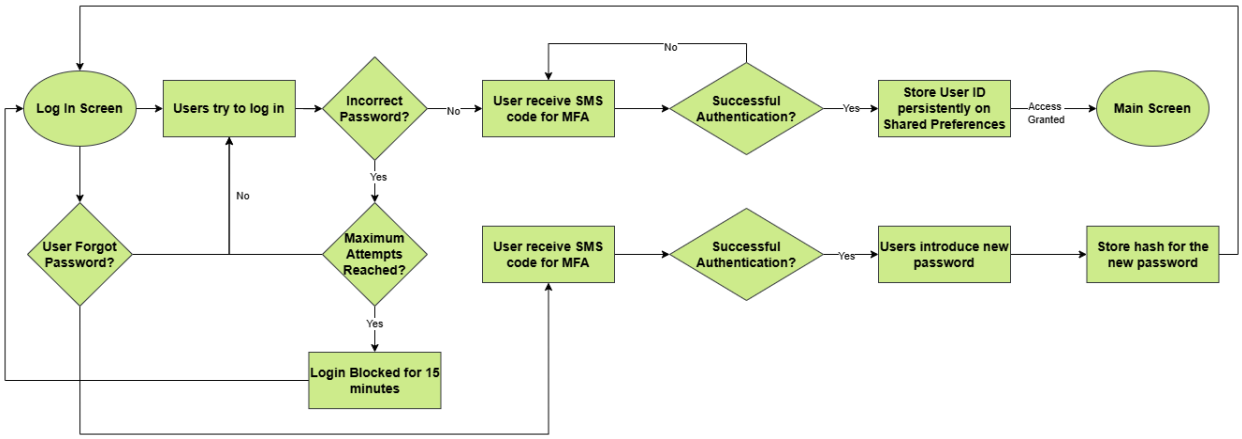
Proposed Enhancements

For this category, I have implemented the following improvements:

- Limiting login attempts to prevent brute-force attacks.

- Multifactor Authentication (MFA) using SMS codes with the user's registered phone number.
- Password change and recovery using MFA SMS verification codes.
- Stateless authentication to ensure persistent sessions that remain active until explicit logout.
- Grouping past events into a dropdown section to keep the main view focused on current and upcoming events.
- A calendar view to complement the chronological agenda list.

For this enhancement, I improved the software security and authentication architecture of the application from a software engineering and design perspective by implementing key industry best practices across the entire login and registration lifecycle. Specifically, I introduced a maximum of five consecutive login attempts, followed by a 15-minute cooldown period to mitigate brute-force attacks. Additionally, I implemented multi-factor authentication (MFA) by validating the user's identity through their registered credentials (username or email) and sending a one-time security code via SMS during the login process. This mechanism is also leveraged to support secure password recovery, allowing users to reset their credentials after successful verification. Finally, I explored different approaches to implement stateless authentication, enabling users to remain logged in until they explicitly log out. This was achieved by leveraging Android architecture components and persistent local storage solutions such as SharedPreferences to securely manage session data on the device. The updated login process incorporating these security enhancements is presented below.



Enhancements Implementation Process and Learning Reflections

- Limiting login attempts to prevent brute-force attacks.

Previously, the database schema for the user table only included basic user information fields such as email, username, password, and phone number. To support the new security enhancements, I recognized the need to implement security measures at the data layer, starting with modifications to the database schema and corresponding query operations. In this context, I extended the user data model to include additional fields for tracking security-related events, specifically the number of failed login attempts and a lockout timestamp. These fields enable the system to detect potential brute-force attacks and enforce temporary account lockouts based on elapsed time. At the Data Access Object (DAO) level, I implemented update queries to modify both the failed attempts counter and the lockout timestamp in a single operation, ensuring consistency and atomicity when enforcing security policies.

```
// Security Variables
6 usages
public int failedAttempts = 0;       // Tracks brute-force attempts
7 usages
public long lockoutTimestamp = 0;    // Time when lockout expires (ms)
8 usages
public String mfaCode;               // Temporary 6-digit SMS code
5 usages
public long mfaExpiry = 0;           // Time when MFA code expires
```

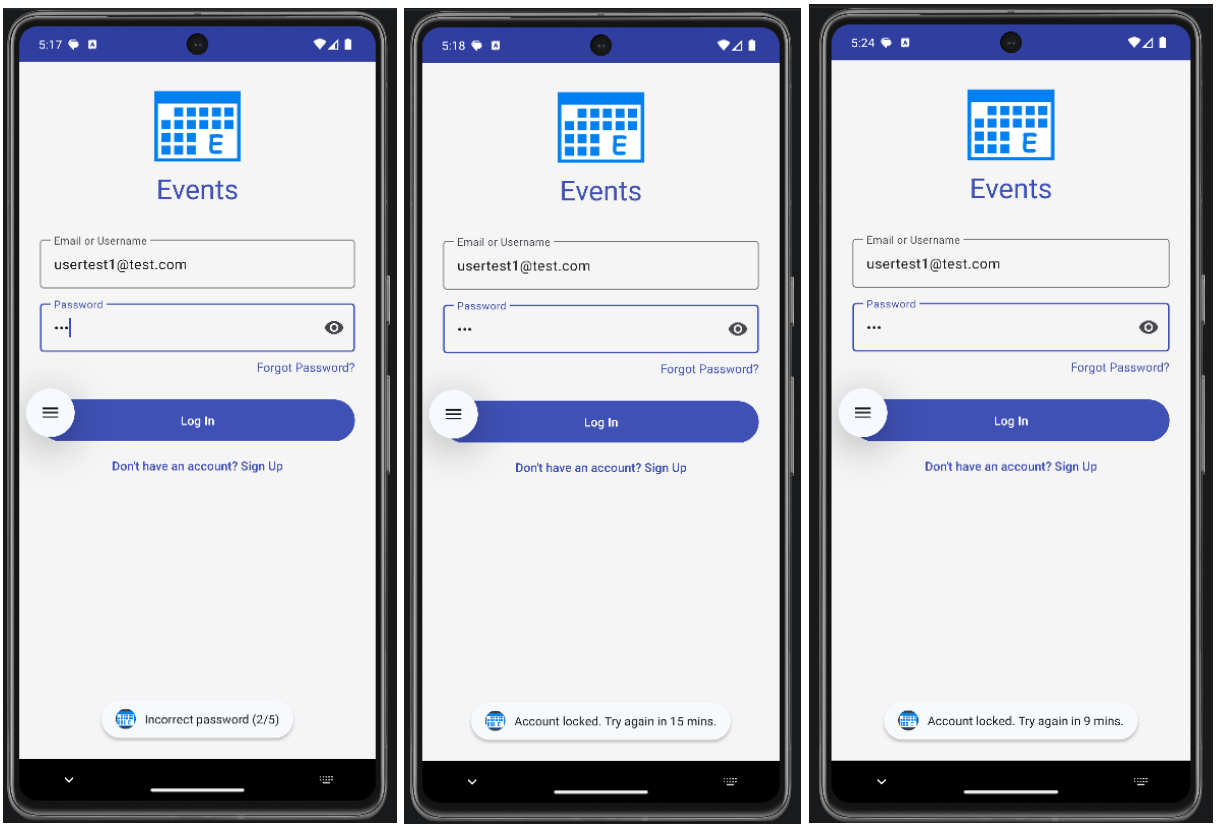
```
35      // Update login attempts and lockout status
        2 usages  1 implementation
36      @Query("UPDATE users SET failedAttempts = :attempts, lockoutTimestamp = :lockout WHERE id = :userId")
37      void updateLockoutStatus(int userId, int attempts, long lockout);
```

In the UserRepository class, where database interactions are abstracted and core data operations are implemented, I refactored the authentication method to incorporate failed login attempt tracking and mitigate brute-force attacks. Specifically, I implemented an account lockout mechanism that enforces a 15-minute cooldown period after five consecutive failed login attempts. The authentication logic first verifies whether a lockout is currently active by evaluating the stored lockout timestamp and the elapsed time using millisecond precision. If the account is still within the lockout period, the system prevents further authentication attempts and immediately notifies the user that the account is temporarily locked, avoiding unnecessary database operations. If no lockout is active, the method proceeds to validate the user-entered password against the stored hashed value. In cases where the credentials do not match, the system increments the failed attempts counter and enforces the lockout policy once the maximum threshold is reached. This implementation highlights the importance of carefully structuring the order of operations within authentication logic. By validating lockout conditions prior to password verification, the system reduces unnecessary database interactions and minimizes exposure to potential malicious activity, thereby improving both security and performance.

```
// Check Brute-Force Attacks and implements a cooldown
if (user.lockoutTimestamp > now) {
    long remaining = (user.lockoutTimestamp - now) / 60000;
    listener.onFinished( user: null,  statusMessage: "Account locked. Try again in " + (remaining + 1) + " mins.");
    return;
}
```

```
} else {
    // Failure - Increment attempts
    int attempts = user.failedAttempts + 1;
    long lockout = (attempts >= 5) ? now + (15 * 60000) : 0;

    userDao.updateLockoutStatus(user.id, attempts, lockout);

    String msg = (attempts >= 5) ? "Too many attempts. Account locked for 15m."
            : "Incorrect password (" + attempts + "/5)";
    listener.onFinished( user: null, msg);
}
```

The test screenshots below demonstrate how the system tracks consecutive failed login attempts and enforces the account lockout mechanism after the maximum threshold is exceeded. They also show that the lockout duration is accurately maintained over time, continuing to be enforced even if the application is closed and later reopened.



- Multifactor Authentication (MFA) using SMS codes with the user's registered phone number.

As previously mentioned, the user table in the application database was expanded to include additional security-related fields, specifically an MFA code and its corresponding expiration timestamp. These fields allow the system to track the validity window of the one-time verification code and ensure that expired codes cannot be reused. Similarly, update methods were added to the data layer to modify both fields within a single operation. This was implemented through the updateMfa query in the User DAO, ensuring consistency and atomicity when updating MFA-related data.

```
39      // Update MFA details
        2 usages  1 implementation
40      @Query("UPDATE users SET mfaCode = :code, mfaExpiry = :expiry WHERE id = :userId")
41      void updateMfa(int userId, String code, long expiry);
```

To implement this feature via SMS, I developed a dedicated helper component responsible for handling SMS delivery for MFA, following separation of concerns principles. Additionally, I applied the DRY (Don't Repeat Yourself) principle at the intersection of security and user experience design. Instead of creating multiple screens, I reused the existing login layout and dynamically updated the interface by enabling or disabling specific UI elements based on the current step of the authentication flow. This approach reduces code duplication while maintaining a consistent and streamlined user experience.

```
// Send MFA SMS to the user with an expiration of 5 minutes
try {
    SmsManager smsManager = getApplicationContext().getSystemService(SmsManager.class);
    if (smsManager != null) {
        String message = "Your Events App security code is: " + code + ". It expires in 5 minutes.";
        smsManager.sendTextMessage(phone,  scAddress: null, message,  sentIntent: null,  deliveryIntent: null);
        return Result.success();
    }
} catch (Exception e) {
    return Result.retry();
}
```

In the improved authentication workflow, instead of granting access immediately after verifying that the input password hash matches the stored value, the system generates a random MFA code within the UserRepository authenticate method. This code, along with its expiration timestamp, is then stored in the database, and the SMS helper component is triggered to deliver the code to the user. Additionally, I implemented a shared verification method used by both the MFA login flow and the password recovery process. This method compares the user-provided code against the stored value in the database and returns either a success or failure result. This result is propagated to the LoginViewModel, which validates the outcome and updates the UI accordingly through event listeners in the activity when the user submits the code. This process is only executed if the user exists in the database. Once identified, the system retrieves the registered phone number and masks it, displaying only the last four digits to provide a hint to the user while protecting personally identifiable information (PII).

```
// Check credentials against stored hash using BCrypt
if (BCrypt.checkpw(password, user.password)) {
    // Success - Reset attempts
    userDao.updateLockoutStatus(user.id, attempts: 0, lockout: 0);

    // Generate 6-digit random MFA Code
    String mfaCode = String.valueOf( i: (int)(Math.random() * 900000) + 100000);
    userDao.updateMfa(user.id, mfaCode, expiry: now + (5 * 60000)); // 5 min expiry

    // Trigger the SMS Worker
    triggerMfaWorker(user.phone, mfaCode);

    listener.onFinished(user, statusMessage: "MFA_SENT");
```

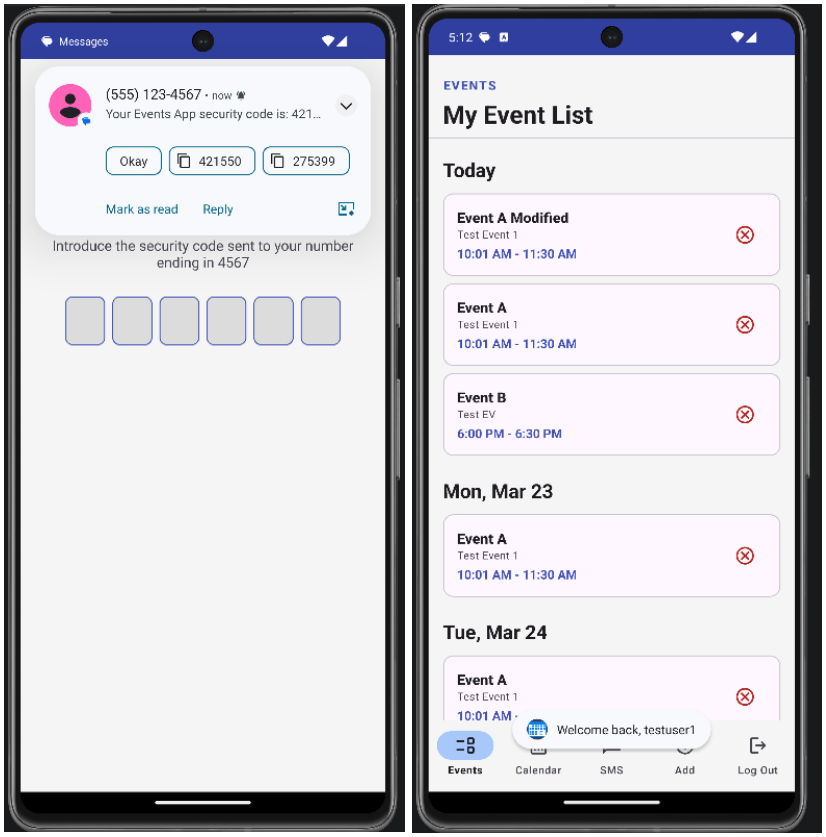
```
// Verify MFA code
1 usage
public void verifyMfa(int userId, String enteredCode, OnAuthListener listener) {
    AppDatabase.databaseWriteExecutor.execute(() -> {
        User user = userDao.getUserByIdSync(userId);
        long now = System.currentTimeMillis();

        // Check if the entered code is equal to the saved on the database and time has not expired
        if (user != null && enteredCode.equals(user.mfaCode) && now < user.mfaExpiry) {
            listener.onFinished(user, statusMessage: "MFA_SUCCESS");
        } else {
            listener.onFinished( user: null, statusMessage: "Incorrect or expired code");
        }
    });
}
```

Finally, as the complete authentication process is handled across the login activity, ViewModel, and layout, this implementation has evolved from a simple toggle between login and sign-up into a multi-step authentication workflow. This workflow now includes MFA verification, password recovery, and password reset processes, significantly increasing both security and functional complexity. To manage this complexity, I implemented a state machine to control the authentication flow. In this approach, specific user actions trigger transitions between defined authentication states, represented as enumerated constants. This design enables the system to determine the appropriate operations, including enabling or disabling UI elements, invoking the correct methods, and executing the corresponding data queries. By managing these states within a structured control flow (e.g., a switch-case construct), I established a clear, maintainable, and scalable mechanism for handling the different stages of authentication while ensuring consistency between the application logic and user interface.

```
// List of the authentication steps for the State Machine
12 usages
public enum AuthState { LOGIN, SIGNUP, MFA_LOGIN, MFA_RECOVERY, RESET_PASSWORD }
```

```
// Toggle Elements visibility based on the mode
1 usage
private void observeViewModel() {
    viewModel.getAuthState().observe( owner: this,  AuthState state -> {
        hideAllSections();
        switch (state) {
            case LOGIN:
                layoutEmailOrUser.setVisibility(View.VISIBLE);
                layoutPassword.setVisibility(View.VISIBLE);
                txtForgotPassword.setVisibility(View.VISIBLE);
                btnLogin.setVisibility(View.VISIBLE);
                btnSignUpToggle.setVisibility(View.VISIBLE);
                btnSignUpToggle.setText("Don't have an account? Sign Up");
                break;
            case SIGNUP:
                layoutEmailOnly.setVisibility(View.VISIBLE);
                layoutUsernameOnly.setVisibility(View.VISIBLE);
                layoutPhone.setVisibility(View.VISIBLE);
                layoutPassword.setVisibility(View.VISIBLE);
                btnRegister.setVisibility(View.VISIBLE);
                btnSignUpToggle.setVisibility(View.VISIBLE);
                btnSignUpToggle.setText("Already have an account? Log In");
                break;
            case MFA_LOGIN:
                btnSignUpToggle.setVisibility(View.GONE);
            case MFA_RECOVERY:
                btnSignUpToggle.setVisibility(View.GONE);
                layoutMfaContainer.setVisibility(View.VISIBLE);
                break;
            case RESET_PASSWORD:
                layoutMfaContainer.setVisibility(View.VISIBLE);
                layoutNewPassword.setVisibility(View.VISIBLE);
                btnSaveNewPassword.setVisibility(View.VISIBLE);
                break;
        }
    });
```

The following test screenshots illustrates the SMS security code sent during the MFA login authentication phase, and the user already authenticated with MFA interacting with the main screen.



- Password change and recovery using MFA SMS verification codes.

For this feature, I added an option on the initial login screen that allows users to initiate a password recovery process if they have forgotten their credentials. This implementation reuses the MFA methods and principles to send a verification code to the user's registered phone number, enabling identity validation before allowing access to password reset functionality. The system first verifies that the user exists by validating the provided identifier (username or email) against the database. Once the user is identified, an MFA security code is generated and sent via SMS. After the user enters and successfully validates the code, the interface dynamically

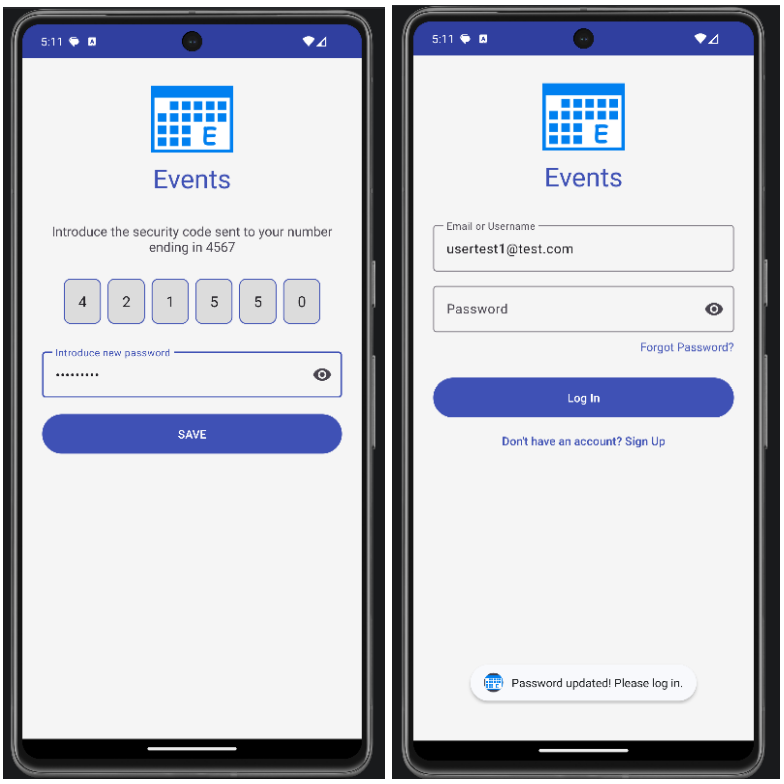
presents a field to enter a new password. Upon submission, the system updates the credentials using the respective SQL query and redirects the user back to the login screen. The logic for displaying the password reset field and handling navigation back to the login flow is managed through predefined state transition constants, leveraging the state machine approach previously described as shown below.

```
// Find the user and send MFA SMS for password recovery without password check
1 usage
public void initiatePasswordRecovery(String identifier, OnAuthListener listener) {
    AppDatabase.databaseWriteExecutor.execute(() -> {
        User user = userDao.findUserByIdentifier(identifier);
        if (user == null) {
            listener.onFinished( user: null,  statusMessage: "User not found.");
            return;
        }

        // Generate and save MFA code for recovery
        String code = String.valueOf( i: (int)(Math.random() * 900000) + 100000);
        userDao.updateMfa(user.id, code,  expiry: System.currentTimeMillis() + (5 * 60000));

        triggerMfaWorker(user.phone, code);
        listener.onFinished(user,  statusMessage: "RECOVERY_MFA_SENT");
    });
}

// Reset user password
1 usage
public void resetPassword(int userId, String newPassword, OnRegisterListener listener) {
    AppDatabase.databaseWriteExecutor.execute(() -> {
        String hashedPassword = BCrypt.hashpw(newPassword, BCrypt.gensalt());
        userDao.resetPassword(userId, hashedPassword);
        if (listener != null) listener.onFinished();
    });
}
```

```
// Initiate password recovery workflow based on user's identifier
1 usage
public void startPasswordRecovery(String identifier) {
    // Check if the user specified identity
    if (identifier == null || identifier.isEmpty()) {
        statusMessage.postValue("Please enter your Username or Email first.");
        return;
    }
    repository.initiatePasswordRecovery(identifier, ( User user,  String message) -> {
        if (user != null) { // Existing user
            pendingUser = user; // Create instance
            setMaskedPhone(user.phone); // Get phone for MFA hint
            authState.postValue(AuthState.MFA_RECOVERY);    // Change state
            statusMessage.postValue("Recovery code sent!");
        } else {
            statusMessage.postValue(message);
        }
    });
}
```

```
// If the MFA is used for recovery, set the state machine to RESET_PASSWORD
repository.verifyMfa(pendingUser.id, code, ( User success,  String message) -> {
    if ("MFA_SUCCESS".equals(message)) {
        if (authState.getValue() == AuthState.MFA_RECOVERY) {
            // Set State Machine to Password Recovery
            authState.postValue(AuthState.RESET_PASSWORD);
```

The following test screenshots illustrate the password recovery workflow, including the display of the required input field after successful MFA verification and the confirmation message shown once the password has been updated. They also demonstrate that, after a successful password reset, the user is redirected back to the login screen to continue the authentication process.



- Stateless authentication to ensure persistent sessions that remain active until explicit logout.

For this feature, I evaluated different design options and leveraged Android architecture components to implement a local session management strategy prior to cloud integration. Since each user session is associated with a unique user ID, I designed the system to persist this identifier using SharedPreferences. Before initiating the login authentication workflow, the application checks for an existing stored user ID and, if present, grants immediate access while

loading the corresponding user data. This approach enables session persistence across application restarts, maintaining the user's authenticated state until an explicit logout occurs.

```
/*
 * Implements stateless authentication by bypassing the login view
 * if a user ID is already stored in SharedPreferences.
 * This allows persistent sessions and automatically restores
 * the user's authenticated state across app launches.
 */
SharedPreferences prefs = getSharedPreferences( name: "EventPrefs", MODE_PRIVATE);
if (prefs.getInt( key: "USER_ID",  defValue: -1) != -1) {
    navigateToMain();
    return;
}
```

I adopted this solution because the application currently relies on a local Room database, meaning that both the stored session data and user information remain confined to the device. Within this context, storing the user ID locally provides a practical and efficient mechanism for session management while preserving user data privacy on the device. I recognize that a token-based session approach would offer stronger security guarantees, as it would involve generating expirable unique session tokens for each login, storing them securely, and validating them during authenticated operations. However, this approach introduces additional complexity and overhead that may not be justified in a fully local application environment. As I define the cloud migration strategy and integrate remote database capabilities, I plan to reevaluate this design and transition toward a more robust, token-based authentication model aligned with distributed system requirements.

- Grouping past events into a dropdown section to keep the main view focused on current and upcoming events.

For this feature, I modified the EventAdapter, ListItem, EventViewModel, and MainActivity to support three distinct item types within the main screen RecyclerView: date

headers, event entries, and collapsible sections for past events. Within the EventAdapter, I introduced two separate data structures to manage upcoming and past events independently. During data processing, each event is evaluated based on its date and position to determine its appropriate classification. Based on this classification, the system dynamically creates the corresponding view holders (e.g., cards and text views) and binds the data to the appropriate UI components on the ViewModel as it is loaded through the activity in real time. During implementation, I encountered null pointer exceptions when handling empty datasets for specific dates or categories. To address this, I incorporated validation checks and conditional control flows to safely handle null or empty collections, ensuring application stability and preventing runtime crashes.

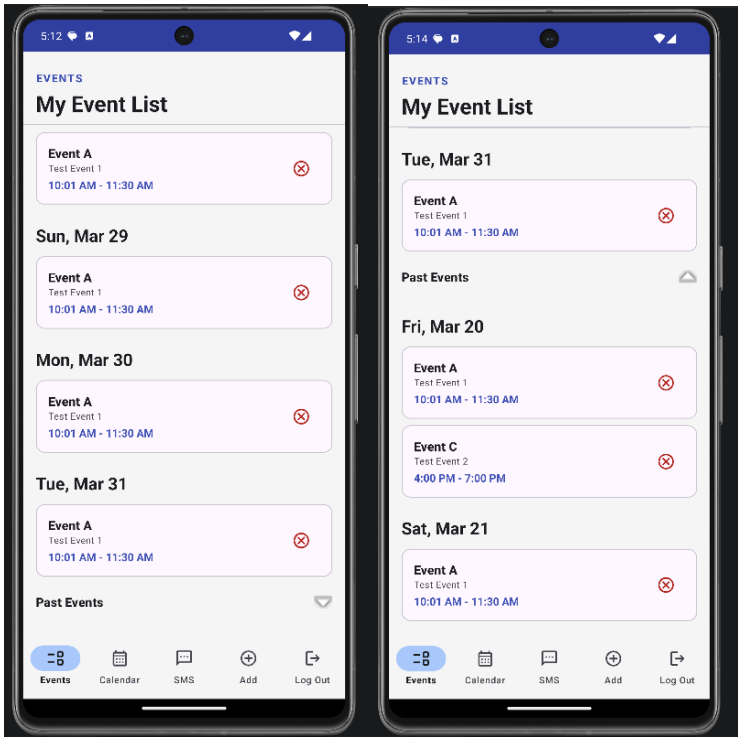
```
// Header type constants
3 usages
public static final int TYPE_HEADER = 0;        // Dates or today
2 usages
public static final int TYPE_EVENT = 1;         // Event header
2 usages
public static final int TYPE_COLLAPSIBLE = 2;   // Dropdown row
```

```
// Separated upcoming items from past items to leverage different lists
13 usages
private List<ListItem> upcomingItems = new ArrayList<>();
9 usages
private List<ListItem> pastItems = new ArrayList<>();
```

```
// Return the event type element int constant
@Override
public int getItemViewType(int position) {
    ListItem item;

    // Safety: If position is the toggle row
    if (!pastItems.isEmpty() && position == upcomingItems.size()) {
        return ListItem.TYPE_COLLAPSIBLE;
    }

    // Identify which list to pull from
    if (position < upcomingItems.size()) {
        item = upcomingItems.get(position);
    } else {
        // This only hits if isPastExpanded is true and position > upcomingItems.size()
        item = pastItems.get(position - upcomingItems.size() - 1);
    }

    return (item.event != null) ? ListItem.TYPE_EVENT : ListItem.TYPE_HEADER;
}
```

```java
// Create view holder based on element type
@NonNull
@Override
public RecyclerView.ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
    LayoutInflater inflater = LayoutInflater.from(parent.getContext());

    switch (viewType) {
        case ListItem.TYPE_HEADER:
            return new HeaderViewHolder(inflater.inflate(R.layout.item_date_header, parent, attachToRoot: false));
        case ListItem.TYPE_COLLAPSIBLE:
            return new CollapsibleViewHolder(inflater.inflate(R.layout.item_past_events_header, parent, attachToRoot: false));
        default:
            return new EventViewHolder(inflater.inflate(R.layout.item_event, parent, attachToRoot: false));
    }
}

@Override
public void onBindViewHolder(@NonNull RecyclerView.ViewHolder holder, int position) {
    ListItem item = null;

    // 1. Identify the Item
    if (pastItems.isEmpty() || position < upcomingItems.size()) {
        item = upcomingItems.get(position);
    } else if (position == upcomingItems.size()) {
        if (holder instanceof CollapsibleViewHolder) bindCollapsibleRow((CollapsibleViewHolder) holder);
        return;
    } else {
        int pastIndex = position - upcomingItems.size() - 1;
        if (pastIndex < pastItems.size()) item = pastItems.get(pastIndex);
    }

    // 2. Safe Binding
    if (item == null) return;

    if (holder instanceof EventViewHolder && item.event != null) {
        bindEventRow((EventViewHolder) holder, item.event);
    } else if (holder instanceof HeaderViewHolder && item.headerDate != null) {
        ((HeaderViewHolder) holder).txtHeader.setText(item.headerDate);
    }
}
```

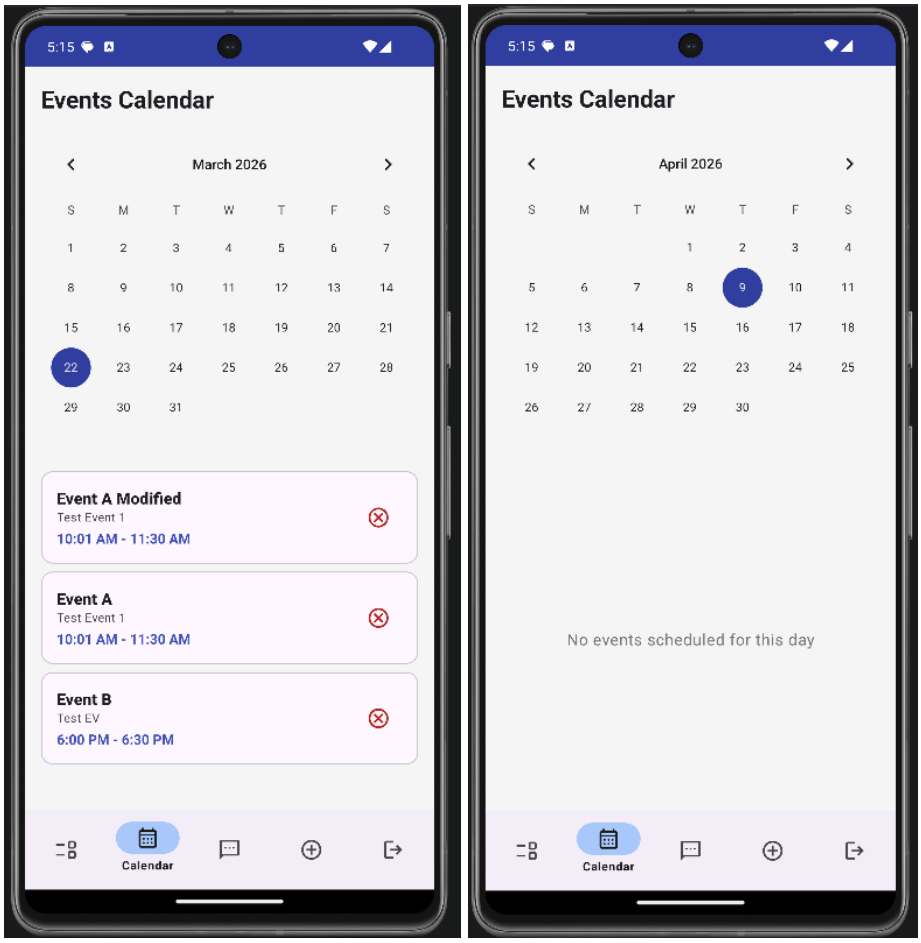
The following test screenshots illustrate past events of the sample list grouped under the past events' collapsable subsection.



- A calendar view to complement the chronological agenda list.

For this feature, I created a new screen that provides users with an alternative, broader view of their events through an intuitive monthly calendar interface. This screen allows users to view, edit, and delete events by selecting a specific day. It consists of a calendar component and a corresponding list below that displays all events associated with the selected date, including past events. By default, the application displays the current day's events; however, users can select any date on the calendar to retrieve the corresponding entries. If no events are registered for a selected date, the system displays an informative message in the list area, following UI/UX design best practices to maintain clarity and user feedback. To integrate this feature efficiently, I added a navigation icon to the bottom navigation bar and implemented the corresponding navigation logic within the activity. The calendar screen shares the same EventViewModel used by MainActivity, relying on a dedicated method to retrieve events by date. This approach applies the DRY (Don't Repeat Yourself) principle and separation of concerns by avoiding redundant logic and centralizing data retrieval within the ViewModel, while the activity focuses solely on UI rendering.

```java
// Filter events by a specific date
1 usage
public List<Event> filterEventsByDate(List<Event> allEvents, String targetDate) {
    List<Event> filteredResults = new ArrayList<>();

    // Safety check: if the list is null or no date is selected, return empty
    if (allEvents == null || targetDate == null) {
        return filteredResults;
    }

    for (Event event : allEvents) {
        if (event.date.equals(targetDate)) {
            filteredResults.add(event);
        }
    }
    return filteredResults;
}
```



Conclusion and Course Outcome Alignment

Through these enhancements, I strengthened the application's software design and engineering quality by focusing on security, architecture, and user experience while aligning with key course outcomes. I implemented industry-relevant security practices, including multi-factor authentication, account lockout policies, and secure password recovery, demonstrating a strong security mindset (Outcome 5). In addition, I evaluated design trade-offs when selecting approaches such as a state machine for managing authentication flow, shared UI layouts to reduce redundancy, and a local stateless session strategy appropriate for the current application

scope. These decisions reflect my ability to design and evaluate computing solutions using established computer science principles (Outcome 3).

Furthermore, I applied modern development practices and innovative techniques by leveraging Android architecture components to build scalable and maintainable features, such as dynamic list management and calendar-based event visualization, aligning with Outcome 4. I also supported collaborative development and clear technical communication (Outcomes 1 and 2) by managing my workflow through version control on GitHub, using a dedicated branch for enhancements while preserving the original implementation for traceability.