Captive Portal Analyzer App Documentation

Amr Elganainy

Contents

1 Introduction

1.1 Overview

The Captive Portal Analyzer is an Android application designed to streamline the analysis of captive portals through an intuitive user interface. This documentation provides comprehensive information about the application's features, architecture, and implementation details. The application facilitates direct interaction with captive portals through an integrated WebView and supports deeper inspection via packet capture by integrating with the PCAPdroid application.

1.2 Purpose

The primary purpose of this application is to facilitate the systematic analysis of captive portals by:

- Providing a secure and controlled environment for captive portal interaction using WebView and PCAPdroid.
- Collecting comprehensive data about portal behavior, characteristics, and network traffic (including raw packet data).
- Enabling AI-powered analysis of privacy implications using a multi-step, data-selectable process.
- Supporting research efforts through optional data sharing, including captured screenshots and PCAP file references.

1.3 Key Features

The application offers several key capabilities:

- Custom WebView implementation with JavaScript injection for application-layer data.
- Integration with PCAPdroid for raw network packet capture and TLS decryption.
- Comprehensive data collection (POST requests, headers, URLs, parameters, HTML/JS content, PCAP files).

- Automated screenshot capture with user-driven flagging for ToS/privacy relevance.
- Session-based interaction tracking, including data captured before and after authentication.
- Integrated, multi-step AI analysis with the Google Generative AI SDK, allowing selection of specific data (including PCAP summaries) for analysis.
- Secure local data storage and optional remote sharing via Firebase.
- Guided setup for PCAPdroid integration.
- Filtering capabilities for viewing collected session data.

1.4 Technical Stack

The application is built using modern Android development technologies:

- Primary Language: Kotlin
- UI Framework: Jetpack Compose
- Architecture Pattern: MVVM
- Navigation: Jetpack Navigation Compose with nested navigation graphs
- Key Libraries:
  - Room Database (for local persistence)
  - Coil (for image loading)
  - Google Generative AI SDK (for AI analysis)
  - Compose Markdown (for displaying AI results)
  - Android Request Inspector WebView (for WebView-based capture)
  - OkHttp and Kotlinx Serialization (for PCAP processing server interaction)
  - Kotlin Coroutines and Flow (for asynchronous operations and reactive data streams)
  - DataStore (for user preferences like theme, language, skip setup flags)
- Backend Services (for optional data upload and PCAP processing):
  - Firebase Firestore

  - Firebase Storage
  - Custom PCAP processing server (interacted via HTTP)
- External Application Integration:
  - PCAPdroid (for packet capture, interacted via Intents)

1.5 Getting Started

To begin using the application, follow these steps:

1.5.1 Repository Setup

Clone the repository using:

```
git clone https://github.com/ganainy/captive-portal-analyzer-kotlin.git
```

1.5.2 Configuration

1. Configure Google Generative AI API:
   - Obtain an API key from Google AI Studio or Google Cloud Console.
   - Add the API key to your project's `local.properties` file (create if it doesn't exist at the root of your project) as: `apiKey=YOUR_API_KEY_HERE`
   - Ensure `local.properties` is included in your `.gitignore` file.
2. Setup Firebase (Optional - for data upload functionality):
   - Create or select an existing project in the Firebase Console.
   - Register your Android app with Firebase (follow console instructions for package name, SHA-1 certificate fingerprint, etc.).
   - Download the `google-services.json` configuration file.
   - Place the `google-services.json` file in your app module's directory (usually `app/`).
   - Enable Firebase Firestore and Firebase Storage in the Firebase Console for your project.
3. (If using the PCAP processing server for AI analysis) Ensure the server URLs (`SERVER_UPLOAD_URL`, `SERVER_STATUS_URL_BASE`) in `AutomaticAnalysisViewModel.kt` are correctly configured for your deployment.

1.6 Architecture Overview

The application implements the MVVM (Model-View-ViewModel) architecture pattern. Each screen composable, e.g., `ManualConnectScreen`, typically has its own ViewModel, e.g., `ManualConnectViewModel`, which is responsible for the business logic. This ViewModel interacts with the Repository, e.g., `NetworkSessionRepository`, which serves as the single source of truth in the app. The repository manages both local (Room Database) and remote (Firebase) data storage. Screens observe UI state (often exposed as `StateFlow<UiState>`) from their ViewModels and delegate user actions to ViewModel functions. This structure promotes separation of concerns, testability, and maintainability by decoupling UI from business logic.
The main components of the MVVM architecture in this application include:

- **Screen Composables (Views)**: Responsible for UI presentation and user interaction. They observe state from ViewModels and delegate actions. Built with Jetpack Compose.
- **ViewModels**: Contain the presentation logic, manage and expose UI state, and interact with Repositories for data operations. ViewModels are often scoped to specific navigation graphs (e.g., `PrimaryAnalysisGraphRoute`, `AutomaticAnalysisGraphRoute`) or individual screens.
- **Repositories**: Act as an abstraction layer over data sources. They manage data fetching and storage logic, interacting with local databases (Room) and remote services (Firebase, PCAP processing server).
- **Data Layer**: Consists of data sources like Room DAOs, Firebase services, DataStore for preferences, and network clients (OkHttp for PCAP server).

The application utilizes a single Activity (`MainActivity`) which hosts the Jetpack Compose UI and manages global concerns such as PCAPdroid integration (intent handling, status broadcast receiver, SAF for file picking), permission handling, and theme/locale settings via a `MainViewModel`. Navigation is handled by Jetpack Navigation Compose, featuring nested navigation graphs for complex, sequential user flows like the primary analysis (WebView/PCAP capture to screenshot flagging) and the multi-step automatic AI analysis process.

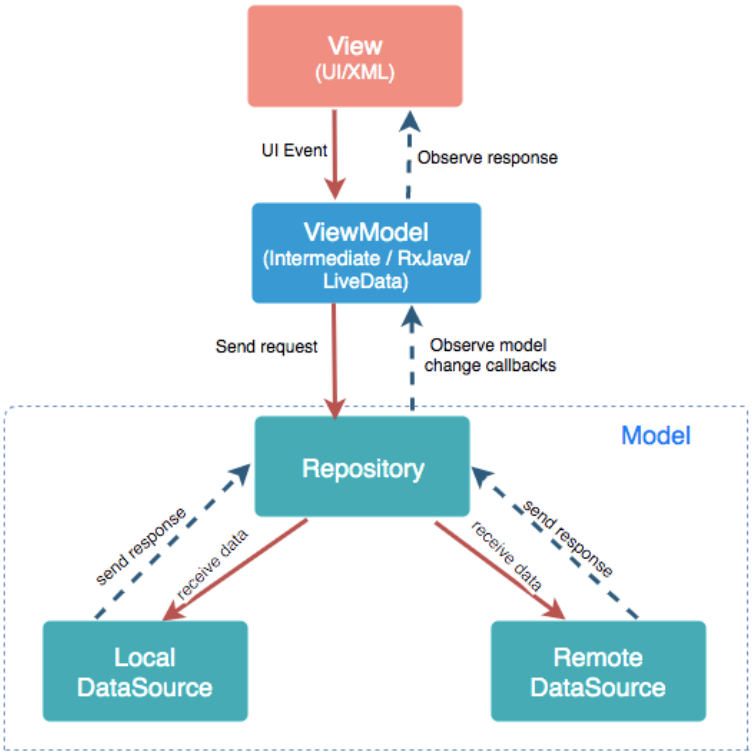

Figure 1: MVVM Architecture.

1.7 Design Patterns Used

The application implements several design patterns:

- **MVVM (Model-View-ViewModel)**
  - Separates UI logic (Composables) from business logic and state management (ViewModels).
  - Enhances testability, maintainability, and separation of concerns.
- **Repository Pattern**
  - Provides a single source of truth for all application data.
  - Abstracts data sources (local database, remote servers, DataStore) from ViewModels.
- **Observer Pattern (via Kotlin Flows)**
  - Utilizes Kotlin `Flow` and `StateFlow` for reactive programming and asynchronous data streams.

  - UI Composables observe `StateFlow` from ViewModels to reactively update based on state changes.

- **Factory Pattern (for ViewModel Creation)**
  - ViewModel factories (e.g., `AnalysisViewModelFactory`, `AutomaticAnalysisViewModelFactory`, `SessionListViewModelFactory`) are used to provide dependencies (like Application context, Repositories, specific session IDs) to ViewModels during their instantiation, particularly when constructor arguments are needed.
- **Sealed Classes/Interfaces (for UI State Management)**
  - Used to define restricted hierarchies for representing UI states (e.g., `AnalysisUiState`, `SessionState`, `PcapProcessingState`, `SessionListUiState`), enabling exhaustive `when` expressions for clear and type-safe state handling in Composables.
- **Dependency Injection (Manual/ViewModel Factories)**
  - While not using a dedicated DI framework like Hilt or Koin in the provided snippets, dependencies are manually provided to ViewModels via factories and passed down through composable functions. This follows DI principles.
- **Strategy Pattern (Implicitly in AI Model Selection)**
  - The ability to choose between different Gemini models (Pro, Flash) for analysis implies a strategy where the core analysis logic can operate with different underlying AI model implementations.

2 Overview Diagrams

2.1 Class Diagram

The application implements a structured data model centered around the `SessionData` entity (often represented by `NetworkSessionEntity` in the database and aggregated with related data counts and lists for UI/ViewModel usage). This entity serves as the primary container for all network-related information collected during a captive portal interaction. Key related entities include `CustomWebViewRequestEntity`, `ScreenshotEntity` (now with an `isPrivacyOrTosRelated` flag), and `WebpageContentEntity`. The `NetworkSessionEntity` itself now includes a `pcapFilePath` to link to captured packet data.

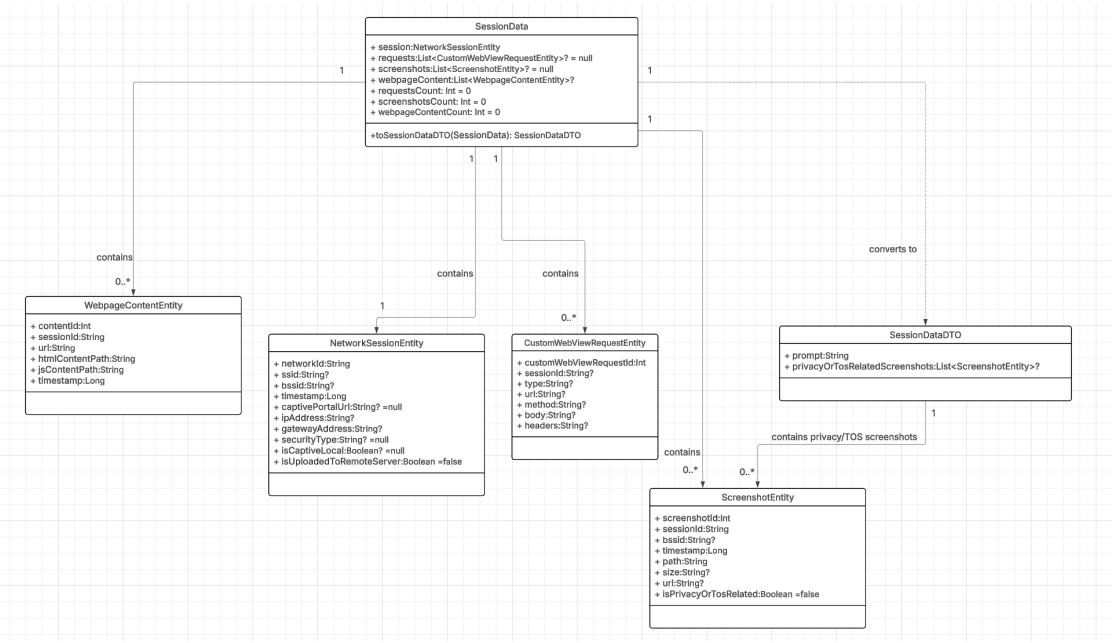

Figure 2: App class diagram.

2.2 App sketch

The diagram (Figure 3) illustrates the user's journey through the application. It begins with app launch, proceeding through initial setup choices (PCAPdroid setup or skip), manual network connection checks, and then into the primary analysis phase. During analysis, the user can interact via a WebView and/or enable PCAPdroid packet capture. After data collection, the user is guided to a screenshot flagging stage. Finally, the user can view collected session details, optionally upload data for further research, or initiate a multi-step AI-powered analysis on selected data components (including processed PCAP data if captured).

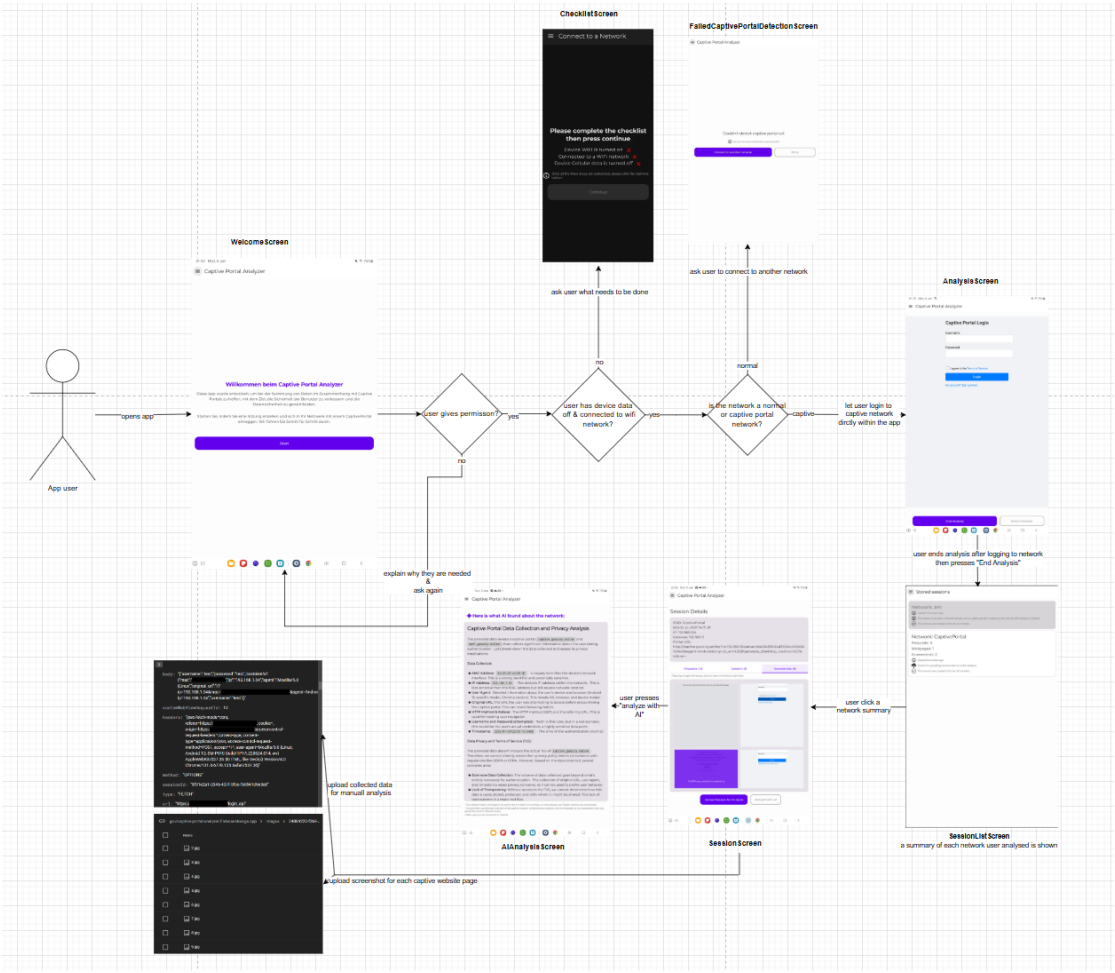

Figure 3: App sketch illustrating key user flows and decision points.

3 MainActivity

The application utilizes a single activity, `MainActivity.kt`, which serves as the app's entry point. It is responsible for initializing critical classes and components used throughout the application, such as the `AppDatabase`, `NetworkSessionManager`, and the global `MainViewModel`. `MainActivity` hosts multiple Jetpack Compose screens, with navigation managed by a `NavController` through actions defined in `AppNavGraph.kt`. It also handles Android system interactions, including:

- Setting up and managing ActivityResultLaunchers for PCAPdroid control intents (start, stop, status).

- Setting up and managing an ActivityResultLauncher for the Storage Access Framework (SAF) to allow users to select PCAP files.
- Registering and unregistering a `BroadcastReceiver` to monitor PCAPdroid's capture status externally.
- Managing app-wide settings like theme and locale changes, often by observing state from `MainViewModel` and triggering UI recompositions or activity recreation.

3.1 AppDatabase

Responsible for local data persistence using the Room library. It provides Data Access Objects (DAOs) for various entities. Key methods provided through its DAOs include:

- `insertSession(session: NetworkSessionEntity)`
  - Purpose: Stores a new network session in the local database.
  - Use case: When a user connects to a new network and initiates analysis, this method creates a persistent record for the session.
  - Key parameters:
    - `session`: Contains all metadata about the network session (SSID, BSSID, timestamp, captive portal URL, IP address, gateway address, and importantly, `pcapFilePath` if applicable).
- `getSessionByNetworkId(networkId: String): NetworkSessionEntity?`
  - Purpose: Retrieves a specific network session from local storage by its unique network identifier.
  - Use case: Checking if a network has been previously encountered or loading details for an existing session.
  - Returns: The `NetworkSessionEntity` details or null if not found.
- `updateSession(session: NetworkSessionEntity)`
  - Purpose: Modifies an existing network session record.
  - Use case: Updating session information as more data is collected, such as adding a `pcapFilePath` or marking the session as uploaded.
- `updatePortalUrl(sessionId: String, portalUrl: String)`

  - Purpose: Updates the captive portal URL for a specific session.
  - Use case: Capturing and storing the detected captive portal URL during the analysis phase.

- `updateIsCaptiveLocal(sessionId: String, isLocal: Boolean)`
  - Purpose: Updates the captive portal status as either **local** (suitable for small businesses and DIY networks[1]) or **remote-hosted** (commonly used by medium and large businesses).

- `getSessionRequests(sessionId: String): List<CustomWebViewRequestEntity>`
  - Purpose: Retrieves all web requests (captured via WebView) made during a specific session.
  - Returns: A reactive stream (Kotlin Flow) of network requests.
  - Use case: Loading network traffic for display in the session details screen or for AI analysis.

- `insertWebpageContent(content: WebpageContentEntity)`
  - Purpose: Stores webpage content (HTML and JS file paths) associated with a session.
  - Use case: Saving HTML and JavaScript web resources captured from the WebView.

- `getSessionWebpageContent(sessionId: String): List<WebpageContentEntity>`
  - Purpose: Retrieves all webpage content entries for a specific session.
  - Returns: A reactive stream (Kotlin Flow) of webpage content entities.

- `getCompleteSessionData(sessionId: String): Flow<SessionData>`
  - Purpose: Provides a comprehensive data retrieval method for a single session.
  - Aggregates:
    * The `NetworkSessionEntity` itself.
    * Associated `CustomWebViewRequestEntity` list.
    * Associated `ScreenshotEntity` list.
    * Associated `WebpageContentEntity` list.

[1] DIY (Do It Yourself) refers to self-managed or custom-built network solutions.

    - Counts for requests, screenshots, and webpage content.
  - Use case: Obtaining a complete overview of a network session for detailed display or AI analysis input.

- Screenshot-related methods:
  - `insertScreenshot(screenshot: ScreenshotEntity)`
    - Purpose: Inserts metadata for a new screenshot, including its storage path on the device, the session and webpage it belongs to, and its `isPrivacyOrTosRelated` status.
  - `getSessionScreenshots(sessionId: String): Flow<List<ScreenshotEntity>>`
    - Purpose: Retrieves all screenshot metadata for a specific session.
    - Returns: A reactive stream (Kotlin Flow) of screenshots.
  - `updateScreenshot(screenshot: ScreenshotEntity)` (Implicitly available via DAO)
    - Purpose: Updates existing screenshot metadata, e.g., to change the `isPrivacyOrTosRelated` flag.

- Remote upload method (handled by `NetworkSessionRepository` using `SessionUploader`):
  - `uploadSessionData(sessionId: String): Result<Unit>` (Conceptual method within Repository)
    - The `SessionUploader` class, instantiated within the repository, handles the comprehensive upload process to Firebase, which now includes:
      1. Loading complete session data (including PCAP path) from the local database.
      2. Uploading screenshot image files to Firebase Storage.
      3. Uploading HTML/JS content files to Firebase Storage.
      4. Potentially uploading the PCAP file or its summary to Firebase Storage.
      5. Creating/updating a session document in Firebase Firestore with metadata and download URLs for the stored files.

- `deleteSessionAndRelatedData(networkId: String)` (Method in Repository, uses DAOs)
  - Purpose: Deletes a session and all its associated data (requests, screenshots, webpage content) from the local database. Associated file deletion from device storage is handled separately in the ViewModel.

3.2 NetworkSessionManager

Responsible for creating and managing network sessions:

- `getCurrentSession(): NetworkSessionEntity?` (Or similar method to get current session details)
    1. Checks app permissions (e.g., location for Wi-Fi scanning).
    2. Retrieves current network identifiers (SSID, BSSID, gateway address, DHCP server address).
    3. Creates a unique network identifier based on this data.
    4. Checks if a session for this network already exists in the database.
    5. Loads the existing session or creates a new `NetworkSessionEntity` if it's a new network or if a new analysis is started. This entity will now include a placeholder or null for `pcapFilePath` initially.
- Uses `ConnectivityManager.NetworkCallback()` to:
    - Detect Wi-Fi connection changes (connect/disconnect).
    - Call a method like `updateCurrentSession()` or `getCurrentSession()` to ensure session information is accurate and can be updated or a new session started if the network changes during app use.
- Provides methods like `savePortalUrl(portalUrl: String, sessionId: String)` and `saveIsCaptiveLocal(value: Boolean, sessionId: String)` to update the current session details in the database via the repository.

3.3 NetworkConnectivityObserver

Monitors device network connectivity to:

- Observe overall network connection status (e.g., connected to any network, internet access available).
- Provide listeners (typically ViewModels) with updates on connection loss or restoration.
- This information is used by ViewModels to enable/disable features like data upload or AI analysis, and to inform the user (e.g., via an internet banner).

3.4 MainViewModel

Manages cross-screen and app-wide state and logic:

- Shares data and state between composable screens and `MainActivity` (e.g., `clickedSessionId` for viewing session details, `PcapDroidCaptureState`, `statusMessage` from PCAPdroid, `copiedPcapFileUri`).
- Manages global UI interactions like dialogs (`DialogState`) and toast messages (`ToastState`).
- Handles app-wide settings (language, theme mode) by interacting with DataStore and exposing them as `StateFlow`.
- Encapsulates logic for interacting with PCAPdroid:
  - Creating intents for starting, stopping, and getting the status of PCAPdroid capture.
  - Handling results from PCAPdroid intents (e.g., `handleStartResult`, `handleStopResult`).
  - Managing the `PcapDroidCaptureState` (e.g., IDLE, RUNNING, FILE_READY).
  - Storing the `targetPcapName` and the URI of the `copiedPcapFileUri` selected by the user via SAF.
- Manages the `skipSetup` preference for the PCAPdroid setup tutorial.
- Provides utility functions like `hasFullInternetAccess(context: Context)`.

3.5 AppNavGraph.kt

The `AppNavGraph.kt` file is central to the application's navigation structure, utilizing Jetpack Navigation Compose to define and manage transitions between different screens and features.

Key responsibilities and features include:

- **Screen Definitions:** A `sealed class Screen` defines all navigable destinations within the app, each associated with a unique route string and a string resource for its title (e.g., `Screen.Welcome`, `Screen.Analysis`, `Screen.PCAPDroidSetup`, `Screen.ScreenshotFlagging`, `Screen.SessionList`, `Screen.Session`, `Screen.AutomaticAnalysisInputRoute`, etc.).

- **Navigation Actions:** A `NavigationActions` class encapsulates all navigation logic. It takes a `NavController` instance and provides lambda functions for navigating to specific screens or graphs (e.g., `actions.navigateToManualConnectScreen`, `actions.navigateToPrimaryAnalysisGraph`, `actions.navigateToAutomaticAnalysisGraph`). This promotes cleaner and more organized navigation calls from individual screen composables.
- **Dependency Management for Screens:** The `AppNavGraph` composable is responsible for setting up the `NavHost` and defining each composable destination. When a screen requires a ViewModel or other dependencies (like `NetworkSessionRepository` or `MainViewModel`), these are typically provided at this level. For ViewModels intended to be shared across multiple screens within a specific flow, they are scoped to a nested navigation graph.
- **Nested Navigation Graphs:** To manage complex user flows and enable ViewModel scoping, the app employs nested navigation graphs:
  - `PrimaryAnalysisGraphRoute`: This graph encapsulates the main data collection and initial processing flow. It starts with the `AnalysisScreen` (where users interact with the captive portal and/or manage PCAPdroid capture) and proceeds to the `ScreenshotFlaggingScreen`. ViewModels such as `AnalysisViewModel` and `ScreenshotFlaggingViewModel` are scoped to this graph's `NavBackStackEntry`, allowing them to share state and data seamlessly throughout this specific part of the user journey.
  - `AutomaticAnalysisGraphRoute`: This graph manages the multi-step AI analysis process. It includes screens like `AutomaticAnalysisInputScreen` (for prompt and data selection), `PcapInclusionScreen` (for PCAP data handling), `AutomaticAnalysisPromptPreviewRoute` (to review the AI prompt), and `AutomaticAnalysisOutputRoute` (to display AI results). The `AutomaticAnalysisViewModel` is scoped to this graph, maintaining its state across these distinct but related steps.
- **ViewModel Scoping:** By retrieving a `NavBackStackEntry` for a specific navigation graph (e.g., `navController.getBackStackEntry(PrimaryAnalysisGraphRoute)`) and using it as the `viewModelStoreOwner` when creating a ViewModel, the ViewModel's lifecycle becomes tied to that graph. This means the ViewModel instance persists as long as any screen within that graph is on the back stack, facilitating state sharing and continuity within that flow. ViewModel factories (e.g., `AnalysisViewModelFactory`, `AutomaticAnalysisViewModelFactory`) are used to pass necessary dependencies to these graph-scoped ViewModels.
- **Global Component Integration:** The `AppNavGraph` also integrates global UI components like `AppToast` (for displaying toast messages managed by

`MainViewModel`) and `ActionAlertDialog` (for dialogs managed by `MainViewModel`), ensuring they are available across all screens.

Example of defining a screen within a nested graph in `AppNavGraph.kt`:

```
// --- Nested Navigation Graph for Primary Analysis Flow ---
navigation(
    startDestination = Screen.Analysis.route,
    route = PrimaryAnalysisGraphRoute // Unique route for this nested graph
) {
    composable(route = Screen.Analysis.route) { backStackEntry ->
        // Obtain the NavBackStackEntry for the graph to scope the ViewModel
        val graphEntry = remember(backStackEntry) {
            navController.getBackStackEntry(PrimaryAnalysisGraphRoute)
        }
        // Instantiate AnalysisViewModel using its factory and scoped to the graph
        val analysisViewModel: AnalysisViewModel = viewModel(
            viewModelStoreOwner = graphEntry,
            factory = AnalysisViewModelFactory(
                application = LocalContext.current.applicationContext
                as Application,
                repository = repository, // Passed down to AppNavGraph
                sessionManager = sessionManager, // Passed down
                showToast = mainViewModel::showToast, //from MainViewModel
                mainViewModel = mainViewModel // Passed down
            )
        )
        // Pass necessary configurations and callbacks to the AnalysisScreen
        AnalysisScreen(
            screenConfig = AnalysisScreenConfig(
                repository = repository,
                sessionManager = sessionManager,
                mainViewModel = mainViewModel,
            ),
            navigationConfig = NavigationConfig(
                onNavigateToScreenshotFlagging =
                actions.navigateToScreenshotFlaggingScreen,
                // Other navigation actions
            ),
            intentLaunchConfig = IntentLaunchConfig(
                onStartIntent = onStartIntentLaunchRequested,
                // Callbacks to MainActivity
```

```
                // Other intent launch callbacks
            )
            // The AnalysisScreen will internally use the analysisViewModel
            // or it can be passed explicitly if needed by its direct composables.
        )
    }
    // Definition for ScreenshotFlaggingScreen, also using graphEntry
    // for its ViewModel
    composable(route = Screen.ScreenshotFlagging.route) { /* ... */ }
}
```

In this structure:

- `screenConfig`, `navigationConfig`, and `intentLaunchConfig` are data classes that group related parameters passed to the `AnalysisScreen`, promoting cleaner function signatures.
- The `analysisViewModel` is explicitly scoped to the `PrimaryAnalysisGraphRoute`. This ensures that when the user navigates from `AnalysisScreen` to `ScreenshotFlaggingScreen` (both within this graph), they can potentially access the same ViewModel instance or easily pass data prepared by `AnalysisViewModel`.
- Navigation actions, like `actions.navigateToScreenshotFlaggingScreen`, are used to move between screens, respecting the defined navigation graph structure.

4 App screens

This section outlines the functions and flow of each screen, explaining how the app operates behind the scenes. Typically, each screen is divided into a `ViewModel` which handels the logic and a `ScreenCompose` that manages the UI. However, for simple screens like `AboutScreen`, `SettingsScreen`, or `WelcomeScreen`, a `ViewModel` is not necessary.

Notes:

- Users are advised to use placeholder data during the signup and login processes, as this data, with user consent, will be uploaded to the server for further analysis.
- For a more thorough understanding of the app's functionality, a video preview is available in the app's GitHub repository: [**ganainy˙captive˙portal˙analyzer˙kotlin**].

4.1 Welcome Screen

Nothing much is going on here since this is a static welcome page. It only has a button that uses the `navigateToNetworkList: () -> Unit` function to navigate to the `Manual Connect` screen when the user clicks it.

4.2 Manual Connect Screen

The `ManualConnectViewModel` is responsible for the following tasks:

1. **Permission Check:** Upon its creation, which is tied to the `ManualConnectScreen`, the `checkPermissions()` function is called. This function checks if the app has the required permissions to access the device's Wi-Fi state. Depending on the permission state, the ViewModel will update the `_uiState`:
   - If `PermissionState.Denied`, the app will ask the user for permission.
   - If `PermissionState.NeedsRequest`, the app will request the necessary permissions.
   - If `PermissionState.Granted`, the app will update the UI to show the checklist to the user.
2. **Wi-Fi State Monitoring:** The ViewModel uses a `BroadcastReceiver` called `wifiStateReceiver` to check if Wi-Fi is turned on, saving the result in the `_isWifiOn` variable.
3. **Network Connection Monitoring:** The ViewModel checks every two seconds whether cellular data is on (`_isCellularOn`) and if Wi-Fi is on. This information is stored in `_isConnectedToWifiNetwork`. It calls the `startCheckingNetworkConnection()` function to monitor the connection status.
4. **Requirements Fulfillment:** Each time any of the variables `_isCellularOn`, `_isConnectedToWifiNetwork`, or `_isWifiOn` changes, the ViewModel checks if all three conditions for starting the analysis are met. If so, it sets `_areAllRequirementsFulfilled` to `true`. When `_areAllRequirementsFulfilled` is true, it is passed to the UI to re-enable the button, allowing the user to start the analysis.

The `ManualConnectScreen` compose is responsible for the following tasks:

1. **Permission Handling:** The screen is responsible for requesting the necessary permissions required for the app's functionality. It uses the `ManualConnectViewModel` to check and handle permissions by observing the `permissionState` and updating the UI accordingly.

2. **Permission Request and Rationale:** If permissions are needed, the screen displays either a permission request screen or a rationale screen, depending on the state of the permissions. It requests permissions using the `ActivityResultContracts.RequestMultiplePermissions` API when needed.

3. **UI State Management:** The screen observes the `uiState` from the ViewModel to decide which content to display to the user. Depending on the state, the screen may show a loading indicator, an error message with retry functionality, or a UI that checks if all requirements (Wi-Fi, cellular, etc.) are met for analysis.

4. **Monitoring Device State:** The screen listens for updates in device states like Wi-Fi status, cellular data status, and network connection using the `ManualConnectViewModel` variables `_areAllRequirementsFulfilled`, `_isCellularOn`, `_isWifiOn`, and `_isConnectedToWifiNetwork`, and updates the UI checklist with a green icon if the prerequisite is fulfilled and a red error icon otherwise.

5. **Navigation:** Once all the conditions are met and the user grants the necessary permissions, the user can navigate to the analysis screen by invoking the `navigateToAnalysis` callback on the Continue button.

4.3 Analysis Screen

The `AnalysisScreen` is the primary interface for interacting with captive portals and initiating data capture. It operates within the `PrimaryAnalysisGraphRoute` and its logic is managed by the graph-scoped `AnalysisViewModel`. This ViewModel handles captive portal detection, WebView interactions, PCAPdroid control (via `MainViewModel`), and data saving.

Key responsibilities and functions of the `AnalysisViewModel`:

1. **Captive Portal Detection (`getCaptivePortalAddress()`)**

   - Invoked when the user proceeds after the initial preference setup in the WebView tab.
   - Utilizes the `CaptivePortalDetector` helper class, which checks standard connectivity check URLs (e.g., `http://clients3.google.com/generate_204`).
   - If a redirection to a captive portal URL occurs (e.g., from a connectivity check URL to `captive.example.com`), the ViewModel updates the UI

state to `AnalysisUiState.CaptiveUrlDetected` and stores the portal URL.

- If no portal is detected (direct internet access), it sets an error state (e.g., `AnalysisUiState.Error(ErrorType.CannotDetectCaptiveUrl)`).
- Handles various error conditions during detection (no internet, timeout, etc.).

2. **UI State and Data Management (`AnalysisUiState`, `AnalysisUiData`):**
   - Manages the overall UI state: `PreferenceSetup` (initial state), `Loading`, `CaptiveUrlDetected`, `AnalysisCompleteNavigateToNextScreen`, and various `Error` states.
   - Stores UI-related data in `AnalysisUiData`, such as the detected `portalUrl`, current `webViewType` (Custom or Normal), and whether initial hints were shown.
   - Interacts with `NetworkSessionManager` to get the current session ID and save the detected portal URL and local/remote status to the current session via the repository.
3. **Local vs. Remote Captive Portal Detection:**
   - After detecting a portal URL, `detectLocalOrRemoteCaptivePortal()` is called. This uses `LocalOrRemoteCaptiveChecker` to perform a DNS lookup on the portal's domain to infer if it's likely hosted locally or remotely, updating the session entity accordingly.
4. **Internet Access Verification (via `MainViewModel.hasFullInternetAccess()`)**
   - This check is used to determine if captured requests (`CustomWebViewRequestEntity.hasFullInternetAccess`) occurred before or after successful authentication, aiding in understanding the portal's behavior at different stages.
5. **WebView Request Handling:**
   - The `saveWebViewRequest(request: WebViewRequest)` and `saveWebResourceRequest(request: WebResourceRequest?)` methods capture details from the WebView.
   - They convert these platform-specific request objects into a standardized `CustomWebViewRequestEntity`, which includes URL, method, headers, body (for `WebViewRequest`), type, and the `hasFullInternetAccess` flag.

- These entities are then saved to the local database via the `NetworkSessionRepository`.

6. **Switching WebView Client (`switchWebViewType()`)**
   - Allows toggling between `WebViewType.CustomWebView` (with JavaScript injection via Android Request Inspector WebView for POST body capture) and `WebViewType.NormalWebView` (standard, no injection).
   - This provides a fallback if the JS injection causes issues with a particular captive portal.
7. **Saving Analysis Data (Screenshots, Web Content):**
   - `takeScreenshot(webView: WebView, url: String)`: Captures the WebView's current content as a bitmap, saves it as a PNG file in the app's internal storage (within a session-specific directory), and creates a `ScreenshotEntity` record in the database with the file path, URL, and timestamp.
   - `saveWebpageContent(webView: WebView, url: String, showToast: ...)`: Extracts HTML and JavaScript content from the current WebView page. This content is saved to separate files (e.g., `webpage_<sanitized_url>.html`, `webpage_<sanitized_url>.js`) in internal storage, and a `WebpageContentEntity` record is created with paths to these files. The `saveContentToFile()` private helper manages file I/O.
8. **PCAPdroid Interaction Management:**
   - Does not directly launch PCAPdroid intents but relies on callbacks from the `AnalysisScreen` (which in turn calls methods on `MainViewModel`) to initiate PCAPdroid start, stop, or status check actions.
   - The `storePcapFilePathInTheSession()` method is called when the user successfully selects a PCAP file in the "Packet Capture" tab (after capture is stopped). It updates the current `NetworkSessionEntity` in the database with the path to the copied PCAP file.
   - The `isPcapDroidAppInstalled()` method checks if PCAPdroid is installed on the device.
9. **Concluding Analysis:**
   - The `stopAnalysis(navigateToScreenshotFlagging: () -> Unit)` method is called when the "End Analysis" button (WebView tab, packet capture disabled) is pressed.

- The `markAnalysisAsComplete()` method is called when the "Save PCAP File" button (Packet Capture tab, packet capture enabled and file selected) is pressed.
- Both methods ultimately set the UI state to `AnalysisUiState.AnalysisCompleteNavigateToNextScreen`.
- `resetViewModelState()` is called to clear data before navigating, ensuring a clean state for subsequent analyses.

The `AnalysisScreen` composable orchestrates the UI based on `AnalysisUiState` and `PcapDroidPacketCaptureStatus` (from `MainViewModel`).

- **Initial State (`AnalysisUiState.PreferenceSetup`):**
  Displays the `PreferenceSetupContent` within the "WebView" tab, prompting the user to choose between WebView-only analysis or PCAPdroid-assisted analysis.
  - If "Continue with Packet Capture" is chosen and PCAPdroid is installed and configured (or after setup), it triggers PCAPdroid start via `MainViewModel` and updates status to `PcapDroidPacketCaptureStatus.PCAPDROID_CAPTURE_ENABLED`. Then, `getCaptivePortalAddress()` is called.
  - If "Continue without Packet Capture" is chosen, status is set to `PcapDroidPacketCaptureStatus.PCAPDROID_CAPTURE_DISABLED`, and `getCaptivePortalAddress()` is called.
- **Loading State:** Shows a loading indicator while detecting the captive portal.
- **Captive Portal Detected (`AnalysisUiState.CaptiveUrlDetected`):**
  - **WebView Tab:** Displays the captive portal URL in the selected WebView (`CustomWebView` or `NormalWebView`). Provides "End Analysis" and "Switch Browser" buttons.
  - **Packet Capture Tab:** Content depends on `PcapDroidPacketCaptureStatus`:
    - If `PCAPDROID_CAPTURE_ENABLED`: Shows `PacketCaptureEnabledContent` for managing PCAPdroid capture (start/stop), selecting the captured PCAP file via SAF, and saving its path.
    - If `PCAPDROID_CAPTURE_DISABLED`: Shows `PacketCaptureDisabledContent`, indicating analysis will proceed without packet capture.
- **Error States:** Displays appropriate error messages (e.g., no portal detected, no internet) with options to retry or connect to another network.

- **Navigation:** Upon `AnalysisUiState.AnalysisCompleteNavigateToNextScreen`, it calls `navigationConfig.onNavigateToScreenshotFlagging` to proceed to the next step in the `PrimaryAnalysisGraph`.

The choice of WebView (`CustomWebView` with JS injection vs. `NormalWebView`) is managed by `AnalysisUiData.webViewType` and can be toggled by the user. Both WebViews trigger data saving methods (`takeScreenshot`, `saveWebpageContent`, `saveWebViewRequest`/`saveWebResourceRequest`) on page load completion (`onPageFinished`).

4.4 Session List Screen

The `SessionListScreen` displays a list of all previously analyzed network sessions, allowing users to filter them and select one for detailed viewing or deletion.

Frontend

- A `TabRow` at the top features two `FilterTab` composables: "Captive" and "Normal", allowing users to switch between viewing sessions classified as captive portals versus regular networks.
- Below the tabs, a list of `NetworkSessionItem` composables is displayed. Each item shows:
  - SSID (Network Name).
  - IP Address and Gateway Address.
  - For captive portal sessions: counts of captured Requests, Webpages, and Screenshots.
  - Timestamp of when the session was created (formatted as "time ago").
- Captive portal items are visually distinct (e.g., primary color for title) and are clickable to navigate to the `SessionScreen` for details. Normal network items are typically greyed out or styled less prominently and are not interactive for navigation.
- Each item includes a delete icon (`IconButton` with `Icons.Filled.Delete`), which, when clicked, triggers a confirmation dialog managed by `MainViewModel`.
- Appropriate empty state UIs are shown:
  - `EmptyStateIndicator`: If no sessions exist in the database at all.
  - `EmptyFilteredListIndicator`: If sessions exist but none match the currently selected filter (e.g., "No captive sessions found").

- A loading indicator (`LoadingIndicator`) is displayed while fetching session data.
- An error component (`ErrorComponent`) is shown if data retrieval fails.

Backend (`SessionListViewModel`)

The `SessionListViewModel` is responsible for fetching, filtering, and managing the list of network sessions.

- **Initialization and Data Retrieval:**
  - Upon creation, the ViewModel launches a coroutine to fetch all `SessionData` objects (which aggregate `NetworkSessionEntity` with counts of related data like requests, screenshots, etc.) from the `NetworkSessionRepository` using `repository.getCompleteSessionDataList().collect`.
  - The fetched list is stored in a private `_sessionsFlow`.
  - The overall UI state (`_uiState: MutableStateFlow<SessionListUiState>`) is updated based on this initial fetch: `Loading` initially, then `Empty` if no sessions are found, or `Success` if sessions are present. If an exception occurs, it's set to `Error`.
- **Filtering Logic:**
  - Manages the currently selected filter via `_selectedFilter: MutableStateFlow<SessionFilterType>` (defaulting to `CAPTIVE`).
  - The `selectFilter(filterType: SessionFilterType)` function updates this state.
  - A public `filteredSessionDataList: StateFlow<List<SessionData>?>` combines `_sessionsFlow` and `_selectedFilter`. It filters the full list of sessions based on whether they are captive portals (i.e., have associated requests, web content, or screenshots, as determined by the `SessionData.isCaptivePortal()` extension function) or normal networks.
- **Session Deletion (`deleteSession(networkId: String)`)**
  - This function is called when the user confirms deletion.
  - It launches a coroutine on `Dispatchers.IO`.
  - It first attempts to retrieve the `NetworkSessionEntity` to get file paths for associated data (PCAP file, screenshots, HTML/JS content).

- It then calls a private helper `deleteAssociatedFiles(session: NetworkSessionEntity)` to delete these files from the device's internal storage. This helper itself uses `File(path).delete()` and logs the outcome.
- Finally, it calls `repository.deleteSessionAndRelatedData(networkId)` to remove the session and its related entries from the Room database.
- The `_sessionsFlow` (and consequently `filteredSessionDataList`) automatically reflects the deletion due to the ongoing collection from the repository.
- Handles and logs exceptions during the deletion process, potentially updating `_uiState` to an error state.

4.5 Session Details Screen

The `SessionScreen` provides an in-depth view of a selected captive portal session, allowing users to inspect collected data across various categories, manage data upload, and initiate AI-powered analysis. Its logic is primarily handled by the `SessionViewModel`.

Backend (`SessionViewModel`)

The `SessionViewModel` is responsible for managing the state and data related to a specific network session selected by the user.

- **Session State Management (`SessionState`):**
  - The ViewModel manages the overall state of the session interaction using a `SessionState` sealed class. This class represents various states such as:
    - `Loading`: Initial state while the session data is being fetched.
    - `NeverUploaded`: Indicates the session has not yet been uploaded to the remote server. The "Upload" button is enabled.
    - `Uploading`: State during the process of uploading session data to Firebase. The UI shows an upload log (`SessionUploadHistoryLog`).
    - `AlreadyUploaded`: The session has been previously uploaded successfully. The "Upload" button is disabled.
    - `Success`: Indicates a recent upload operation completed successfully.
    - `ReUploading` (Primarily for testing): Allows re-uploading an already uploaded session.
    - Error states:

  - `ErrorLoading(message: String)`: An error occurred while loading session data from the local database.
  - `ErrorUploading(message: String)`: An error occurred during the upload process to the remote server.

- **Session Data Loading (`loadSessionData()`):**
  - Triggered on ViewModel initialization, using the `clickedSessionId` passed from `MainViewModel`.
  - Fetches the complete `SessionData` (including the session entity, requests, screenshots, web content, and PCAP file path) for the selected session from the `NetworkSessionRepository`.
  - Updates the `sessionUiData: StateFlow<SessionUiData>` which holds the loaded data and filtering states.
  - Sets the initial `SessionState` based on the `isUploadedToRemoteServer` flag of the fetched session.
  - Handles and logs errors during data loading, updating the `SessionState` accordingly.
- **Screenshot Privacy/TOS Toggle:**
  - Although screenshots are initially flagged in the `ScreenshotFlaggingScreen`, the `SessionViewModel` would be responsible for any further modifications to the `isPrivacyOrTosRelated` state of a `ScreenshotEntity` if such functionality were to be added to this screen (currently, toggling is done prior to reaching this screen).
- **Session Upload (`uploadSession(sessionData: SessionData, showToast: ...)`):**
  - Handles the upload of the complete `sessionData` (including requests, HTML/JS file paths, screenshot file paths, and PCAP file path) to Firebase.
  - Delegates the actual upload mechanism to the `NetworkSessionRepository` (which uses `SessionUploader`).
  - Updates the `SessionState` to `Uploading` at the start, and transitions to `Success` or `ErrorUploading` based on the outcome.
  - Populates the `uploadHistory: StateFlow<List<Pair<String, MessageType>>>` with step-by-step messages during the upload process (e.g., "Initializing upload...", "Uploading screenshots...", "Upload complete.").

  - Uses the `showToast` callback (from `MainViewModel`) to display success or error messages to the user.

- **Session Requests Filtering:**
  - Manages filtering options for the "Requests" tab through `SessionUiData` state, including:
    * `showFilteringBottomSheet: Boolean` to control the visibility of the filter controls.
    * `isBodyEmptyChecked: Boolean` to filter requests based on whether their body is empty or not.
    * `selectedMethods: List<Map<RequestMethod, Boolean>>` to filter requests by HTTP method (GET, POST, etc.).
  - Provides functions like `toggleShowBottomSheet()`, `toggleIsBodyEmpty()`, `modifySelectedMethods(method: RequestMethod)`, and `resetFilters()` to update these filter states.
  - The actual filtering of the request list displayed in the UI is typically done reactively in the composable based on these state values from `sessionUiData`.

Frontend (`SessionScreen`)

The `SessionScreen` composable displays the detailed information of the selected session and provides actions for uploading or further analysis.

- **UI State Management:**
  - Dynamically updates the UI based on the current `SessionState` from `SessionViewModel`.
  - When `SessionState.Uploading`, it displays the `SessionUploadHistoryLog` composable, showing a list of upload progress messages with icons and a loading indicator.
  - For `Loading`, `ErrorLoading`, or `ErrorUploading` states, it shows appropriate loading indicators or error components (with retry options for upload errors).
  - For states like `NeverUploaded`, `AlreadyUploaded`, or `Success`, it displays the main session details content.
- **Session Details Display (`SessionDetail` composable):**
  - Uses a `LazyColumn` for efficient display of potentially long lists of data.

  - `SessionGeneralDetails`: Displays general information at the top (SSID, BSSID, IP, Gateway, Portal URL, and PCAP file name if present). This section is expandable/collapsible.
  - A `stickyHeader` contains a `TabRow` allowing users to switch between:
    * **Requests Tab**: Lists `CustomWebViewRequestEntity` items using `RequestListItem`, separated by `ListSectionHeader` into "Before Authentication" and "After Authentication" (based on `hasFullInternetAccess`). When this tab is active, a `FiltersHeader` with a filter icon is shown in the sticky header, which toggles the `FilterBottomSheet`.
    * **Content Tab**: Lists `WebpageContentEntity` items using `ContentItem`, showing URL and paths to HTML/JS files.
    * **Images Tab**: Displays `ScreenshotEntity` items using `ScreenshotDisplayCard` within a `FlowRow`, grouped by "Privacy-Related Screenshots" and "Other Screenshots" based on the `isPrivacyOrTosRelated` flag.
  - Each list item (`RequestListItem`, `ContentItem`) is clickable, navigating to `RequestDetailsScreen` or `WebPageContentScreen` respectively, passing the relevant entity ID via `MainViewModel`.

- **User Interaction Handling:**
  - `SessionActionButtons`: A horizontally scrollable row of buttons at the bottom:
    * "Upload Session for Analysis": Enabled if `SessionState` is `NeverUploaded` (or `ReUploading`). Triggers `sessionViewModel.uploadSession(...)`.
    * Status texts like "Already Uploaded" or "Thanks for Uploading" are shown if the upload state reflects these.
    * "Automatic Analysis": Navigates to the `AutomaticAnalysisGraphRoute` to start the AI analysis flow.
  - Internet connectivity is monitored via `MainViewModel.isConnected`, and an `AnimatedNoInternetBanner` is shown if offline.
  - A `HintInfoBox` (a `NeverSeeAgainAlertDialog`) provides initial guidance to the user.

- **Filtering Handling (`FilterBottomSheet`):**
  - Displayed when the filter icon in the "Requests" tab's header is clicked (`sessionUiData.showFilteringBottomSheet` is true).
  - Allows users to toggle a checkbox for "Hide requests with empty body" (`onToggleIsBodyEmpty`) and select/deselect request methods using `CustomChips` (`onModifySelectedMethods`).

– A "Remove Filters" button calls `onResetFilters`.

4.6 Automatic Analysis Flow (Graph-Scoped `AutomaticAnalysisViewModel`)

The automatic analysis feature is implemented as a multi-screen flow, managed by the `AutomaticAnalysisGraphRoute` and its graph-scoped `AutomaticAnalysisViewModel`. This ViewModel orchestrates data loading, user selections, PCAP processing, prompt generation, and interaction with the generative AI model (Google Generative AI SDK).

Key responsibilities and functions of the `AutomaticAnalysisViewModel` include:

- **Initialization & Session Data Loading:**
  - Upon creation (scoped to the graph), it receives the `clickedSessionId`.
  - The `loadSessionData()` function fetches the complete `SessionData` (including session entity, requests, screenshots, web content, and PCAP file path) from the `NetworkSessionRepository`.
  - It initializes the UI state with counts of available data items and checks if a PCAP file is associated and accessible (`isPcapIncludable`, `pcapFilePath`).
  - Defaults to selecting all available requests and relevant (ToS/privacy-flagged) screenshots for analysis.
- **User Input and Data Selection Management:**
  - Manages the custom prompt entered by the user (`updatePromptEditText(text: String)`).
  - Handles user selections for including specific network requests, screenshots (those flagged as ToS/privacy related), and webpage content items in the analysis (`toggleRequestSelection`, `toggleScreenshotSelection`, `toggleWebpageContentSelection`, and 'setAll...Selected' methods).
  - Manages the selection state for including PCAP data (`togglePcapSelection(isSelected Boolean)`).
- **PCAP File Processing (Client-Server Interaction):**
  - If PCAP data is selected and available (`isPcapIncludable` is true and `isPcapSelected` is true):

    * The `convertPcapToJsonAndWait(pcapFilePath: String)` suspend function is invoked. This function:
        1. Uploads the local PCAP file (specified by `pcapFilePath`) to a remote processing server (e.g., `http://your-server-ip/upload`) using OkHttp.
        2. Receives a `job_id` from the server.
        3. Polls a status endpoint (e.g., `http://your-server-ip/status/<job_id>`) for completion.
        4. Once completed, downloads the processed JSON summary from a URL provided by the server.
        5. Handles potential truncation of the JSON summary if it exceeds `PCAP_JSON_MAX_LENGTH`.
    * Manages the `pcapProcessingState` (`Idle`, `Uploading`, `Processing`, `Polling`, `DownloadingJson`, `Success`, `Error`) to provide UI feedback.
    * Stores the resulting JSON content (and truncation status) in the UI state.
    * Includes a `retryPcapConversion()` function if the server-side processing fails.
    * Provides `cancelPcapConversion(reason: String)` to stop ongoing PCAP processing.

- **Prompt Generation for Preview and Analysis:**
    - The `generatePromptForPreview()` function (and implicitly within `analyzeWithAi`) dynamically constructs the prompt sent to the AI model.
    - The prompt includes:
        * A base instruction set for the AI regarding its role and the expected output format (default provided in `AutomaticAnalysisUiState.inputText`).
        * The user's custom input text (if modified from default).
        * Formatted representations of the selected network requests (prioritizing POST requests and truncating if necessary to fit within `REQUESTS_LIMIT_MAX`).
        * A note indicating the number of selected relevant screenshots (which are passed as image content to the AI).
        * A note about selected webpage content (content itself not directly in prompt, but noted).

    - The processed PCAP JSON summary (if successfully converted and included), or a status message if PCAP processing failed or was skipped.
  - The `AutomaticAnalysisPromptPreviewScreen` displays this generated prompt.

- **AI Analysis Interaction:**
  - The `analyzeWithAi(modelName: String?)` function:
    - Takes the selected AI model name (e.g., "gemini-1.5-pro-latest", "gemini-1.5-flash-latest") as a parameter.
    - Prepares the input content for the GenerativeModel, which includes the final text prompt and any selected screenshot bitmaps (decoded via `pathToBitmap(path: String)`).
    - Instantiates the `GenerativeModel` with the API key and specified model name.
    - Calls `generateContentStream(inputContent)` to get a streaming response from the AI.
    - Updates the `outputText` in the UI state incrementally as text chunks are received, providing a real-time feel.
    - Manages `isLoading` and `error` states specific to the AI interaction.

The automatic analysis flow is presented to the user through a sequence of screens, all observing and interacting with the graph-scoped `AutomaticAnalysisViewModel`:

- `AutomaticAnalysisInputScreen`: Allows users to input a custom prompt and select which categories of data (requests, screenshots, web content) from the session to include in the analysis. Presents data counts and selection checkboxes.

- `PcapInclusionScreen`: If a PCAP file is associated with the session, this screen allows the user to choose whether to include its processed summary in the AI analysis. It displays the status of the PCAP processing (upload, conversion, download of summary) and provides options to analyze with different AI models (Pro or Fast). Users can also navigate to the prompt preview from here.

- `AutomaticAnalysisPromptPreviewScreen`: Shows the user the exact prompt that will be sent to the AI model, including their custom text and a representation/status of the selected data (including PCAP summary status). UI includes status cards for each data type and the generated prompt text.

- `AutomaticAnalysisOutputScreen:` Displays the results of the AI-based analysis. It shows a collapsible header with the prompt used and a summary of included data, followed by the AI-generated report (streamed in and rendered as Markdown). Includes a collapsible hints section.

Throughout this flow, the UI dynamically updates based on the `AutomaticAnalysisUiState`, showing loading indicators, error messages, or the relevant content for each step. Internet connectivity is also monitored and displayed.

4.7 Settings Screen

The `SettingsScreen` logic is relatively simple and does not require its own ViewModel. Adding support for more languages is simple—just supply an XML file containing the desired translations (e.g., `strings.es.xml` for Spanish). The screen receives two callback functions and two arguments from the `MainActivity`:

- `onLocalChanged: (Locale) -> Unit`: Notifies the `MainActivity` when the user changes the app language which with help of `SharedViewModel` is saved to the user persistent preference using `DataStore` so that the next time the app is launched the last used settings are loaded.
- `onThemeChange: (ThemeMode) -> Unit`: Similar to `onLocalChanged`, it notifies the `MainActivity` when the user changes the app theme mode which with help of `SharedViewModel` is saved to the user persistent preference using `DataStore`.
- `themeMode: ThemeMode`: Represents the currently selected theme mode set by the user.
- `currentLanguage: String`: Stores the currently selected language set by the user.