

### **Activities Admin UML Diagram**

UML Diagram provides a structured representation of the administrative subsystem and its interrelated components within the application. AdminDashboardActivity serves as the central control point, facilitating navigation to event management, profile oversight, organizer administration, image auditing, notification log review, and user report processing. Each activity is modeled with its essential methods, dependencies, and operational roles. Multiplicity indicators formally express one-to-one associations with Firebase services and adapter classes, as well as one-to-many relationships involving collections of Event, User, and Report objects. The diagram further demonstrates the reliance of administrative workflows on Firestore for persistent data operations, NotificationHelper for system-level communication, and RecyclerView adapters for structured list presentation. Supporting elements such as ActiveUsersCallback, UserActivityTracker, and the core domain models are represented through explicit dependency links, offering a comprehensive and precise overview of the administrative architecture.

### **Core Activities UML Diagram**

The Core Activities UML Diagram presents a structured overview of the application's primary authentication and entry-point workflow. MainActivity, LoginActivity, and CreateAcctActivity form the foundational navigation layer, each inheriting from AppCompatActivity to reflect Android's standard activity architecture. The diagram enumerates all relevant attributes and methods, including UI bindings, Firebase service references, SharedPreferences keys, and internal utility functions that support login validation, user state assessment, and account creation. Clear navigation pathways are depicted through explicit associations: MainActivity manages auto login logic and routes users to either AdminDashboardActivity or EventListActivity, while also initiating the NotificationListenerService when required. LoginActivity directs users according to their authentication status and assigned role, and CreateAcctActivity interacts with CreateAccountRepo to complete registration procedures. Multiplicity markers formally illustrate allowable interaction counts between activities and supporting components.

### **Activities Entrant UML Diagram**

The Entrant Activities UML Diagram provides a detailed structural representation of all user-facing workflows that support event discovery, participation, geolocation validation, notification management, QR scanning, and profile viewing. Core screens including EventListActivity, EventDetails, EntrantMapActivity, UserNotificationsActivity, MyEventsActivity, QRCodeScanner, ImageDetailsActivity, and ProfileActivity. All derive from AppCompatActivity, emphasizing their alignment with Android's activity lifecycle. Each class is presented with its complete set of attributes and methods, capturing UI bindings, Firestore access points, adapters, utility helpers, and internal logic for filtering, data loading, state updates, and user interaction handling. The diagram clearly models navigation pathways, illustrating how users move fluidly between event listings, detailed event information, map-based check-in

validation, personal event status tabs, and notification or profile screens. Operational dependencies are expressed through explicit “uses” relationships with components such as EventAdapter, PopularEventsAdapter, LotteryManager, LocationHelper, and the User model, reflecting the mechanisms that enable real-time waitlist updates, location-based verification, and personalized event recommendations. Multiplicity indicators show how activities interact with collections of events, notifications, and waitlist entries, while directional associations clarify the flow of control between components.

### **Activities Organizer & Related UML Diagram**

The Organizer & Related Activities UML Diagram presents a comprehensive structural overview of all organizer-focused features within the application, modeling how event creation, lottery management, participant oversight, and messaging functionalities interact across multiple coordinated activities. Core screens such as OrganizerActivity, CreateEvent, ManageDrawActivity, OrganizerMessagesActivity, and MyEventsActivity are shown with their complete sets of attributes and methods, reflecting their UI bindings, Firestore integrations, adapters, and operational helper classes. The diagram defines clear navigation pathways that illustrate how organizers transition between event setup, managing waiting-list draws, reviewing participant lists, sending targeted notifications, and monitoring event status. Shared screens including EventDetails, ProfileActivity, and QRCodeScanner. They are represented through precise associations that highlight reuse across both organizer and entrant workflows. Dependency relationships demonstrate how activities rely on key components such as Event, WaitingListEntry, User, NotificationMessage, NotificationHelper, LotteryManager, and EventRepository for retrieving event data, executing lottery logic, processing waitlists, and delivering system notifications. Multiplicity markers explicitly represent one-to-many relationships, such as organizers managing multiple events or each event containing numerous waiting-list entries.

### **Adapters Admin UML Diagram**

The Admin Adapters UML Diagram provides a structured representation of the adapter layer responsible for rendering administrative data within the user interface. Each adapter corresponds to a specific administrative task such as managing events, moderating images, reviewing user profiles, processing reports, and displaying notification logs and encapsulates the logic required to translate domain objects (Event, User, Report, ImageItem) into interactive RecyclerView list items. The diagram shows that each adapter maintains a collection of domain entities, offers setter methods to update its dataset, and employs nested ViewHolder classes to bind UI components efficiently. Dependencies are modeled through clearly defined associations with listener interfaces such as OnImageDeleteListener, OnEventClickListener, OnReportActionListener, and OnProfileClickListener, which enable adapters to communicate user actions back to their hosting activities. Multiplicity indicators denote that each adapter

manages zero or more viewholders, while each listener represents a single functional callback relationship.

### **Adapters Entrant Event UML Diagram**

The Entrant Event Adapters UML Diagram provides a structured representation of the adapter components responsible for rendering event data to regular users across the application's interface. Each adapter, `EventAdapter`, `EventListAdapter`, `PopularEventsAdapter`, and `MyEventsAdapter`. They maintain a one-to-many relationship with the `Event` model, reflecting the presentation of multiple event items within list or card-based views. Corresponding `ViewHolder` classes encapsulate the UI elements required to display event information consistently and efficiently. Listener interfaces, including `EventListener` and `OnEventClickListener`, are modeled as explicit dependencies that enable adapters to communicate user interactions back to the hosting activities or fragments without introducing tight coupling. `MyEventsAdapter` additionally depends on `LotteryManager`, supporting user actions related to joining, accepting, or managing lottery-based event selections. Multiplicity markers and the "uses" relationships clarify how adapters coordinate with event models and helper components to deliver filtering, interaction handling, and dynamic UI updates.

### **Adapters Organizer Event UML Diagram**

The Organizer Event Adapter UML Diagram illustrates the structure responsible for rendering and managing organizer-owned events within the application's interface. The `OrganizerEventAdapter` maintains a one-to-many association with the `Event` model, enabling organizers to view and interact with multiple events simultaneously. All actionable behaviors such as editing an event, viewing details, managing lottery draws, exporting CSV files, and handling image updates. They are delegated through the `OrganizerEventListener` interface, ensuring a clean separation between UI presentation and event-management logic. The `OrganizerEventViewHolder` encapsulates the visual components of each event card, including images, titles, descriptions, status indicators, and action buttons, and provides a `bind` method that applies data and listener callbacks to the UI.

### **Adapters Waiting List & Notification UML Diagram**

The Waiting List & Notification Adapters UML Diagram models the two adapter components responsible for rendering entrant information and system notifications within the UI. The `WaitingListAdapter` manages a collection of `WaitingListEntry` objects and produces `WaitingListViewHolder` instances that display each entrant's name, join date, and status. Through the `OnSendNotificationClickListener` and `OnCancelEntrantClickListener` interfaces, it provides organizers with structured callbacks for sending custom messages or removing entrants from the waitlist. In parallel, the `NotificationMessageAdapter` handles a list of `NotificationMessage` objects, binding them to `MessageViewHolder` elements that present message titles, content, timestamps, and read status. Click interactions are delegated through the

OnMessageClickListener interface, maintaining separation between UI. Multiplicity notations emphasize that each adapter manages many model items while each ViewHolder corresponds to a single entry.

### **Constants UML Diagram**

The Constants UML Diagram represents the centralized repository of fixed configuration values used throughout the application. AppColors defines a consistent color palette, ensuring that UI components across different screens follow a unified visual style. AppConstants consolidates all global identifiers, including SharedPreferences keys, Firestore collection and field names, feature flags, intent extras, and fixed application rules such as minimum age limits and participant caps. By centralizing these immutable values, the constants package prevents hard-coded strings from being scattered across the codebase, reduces maintenance overhead, and helps ensure that structural.

### **Fragments UML Diagram**

The Fragments UML Diagram illustrates the modular reporting components used throughout the application. ReportDialogFragment encapsulates the user-facing interface for creating and submitting new reports, storing the associated event ID and handling user input such as descriptions and severity ratings before forwarding the data to the backend. In contrast, ReportDetailsDialogFragment supports administrative workflows by displaying the full details of an existing Report object and enabling resolution actions through a dedicated ReportResolveListener callback. Both fragments define their own lifecycle and view-inflation logic, ensuring UI behavior remains isolated and maintainable. Multiplicity relationships clarify that each fragment operates on a single report instance, while listener interfaces provide a clean communication channel back to the hosting activity. Together, these fragments deliver a structured, decoupled implementation of the reporting flow, balancing user interaction with administrative oversight.

### **Models UML Diagram**

The Models UML Diagram defines the core data entities underpinning the application's functionality. User and Event serve as the primary domain objects, while intermediary models like WaitingListEntry, NotificationMessage, and Report. They capture the interactions and state transitions that occur between them. WaitingListEntry represents a user's position and status within an event's waiting list, storing timestamps and optional geolocation data used for map-based features. NotificationMessage models all system and organizer generated communications, linking each message to its intended recipient and associated event. Report encapsulates user-submitted incident information, recording the reporter, event context, description, and administrative resolution status.

## **Network & Services UML Diagram**

The Network & Services UML Diagram outlines the foundational data-access layer that mediates all interaction between the application and Firebase. This layer abstracts authentication, data retrieval, and real-time updates away from the UI, ensuring a clean separation of concerns and enhancing maintainability. CreateAccountRepo encapsulates the full user-registration workflow by coordinating with Firebase Authentication and persisting user records to Firestore, returning success or failure states through the RegistrationCallback interface. EventRepository serves as the centralized gateway for event-related data, offering multiple query operations including full retrieval, name-based search, category filtering, location filtering, and date-range queries. While returning results asynchronously through the EventCallback interface. The NotificationListenerService operates as a background foreground-capable Android service, listening to Firestore for real-time notification changes and dispatching system notifications when new entries are detected.

## **Utils UML Diagram**

The Utils Layer encapsulates a collection of helper classes that provide supporting functionality to the application's core workflows, ensuring that complex operations remain modular, reusable, and cleanly separated from UI and network logic. The UML diagram highlights how each utility class delivers a focused responsibility: CsvUtils handles CSV file generation using the CsvRow model to represent individual exportable records; Validator leverages the Result class to standardize validation outcomes for draw capacity and input checks. The LotteryManager orchestrates the logic behind event lotteries, selecting entrants, updating statuses, and coordinating outcome notifications, while LotteryScheduler automates these processes by periodically scanning for events that require scheduled draws. NotificationHelper and NotificationActionsHelper encapsulate notification dispatch and action handling, ensuring consistent messaging behavior across the app. LocationHelper abstracts user location retrieval through callback-based operations, and UserActivityTracker manages user activity timestamps to support analytics and behavioral features.

## **Workers UML Diagram**

The Workers layer defines the background execution components responsible for running scheduled and asynchronous operations independently of the user interface. The UML diagram highlights two primary worker classes. `DailyNotificationWorker` automates the delivery of personalized event recommendations by retrieving user preferences from Firestore, filtering for relevant upcoming events, enforcing a 24-hour cooldown to prevent duplicate suggestions, and dispatching notifications through `NotificationHelper`. In parallel, `LotteryWorker` monitors events whose registration periods have ended and delegates the automated draw process to `LotteryScheduler`, ensuring timely and consistent lottery execution without organizer intervention. Both workers inherit from Android's `Worker` class, enabling them to operate reliably in the background using `WorkManager` scheduling.

## **ConnectApplication UML Diagram**

The `ConnectApplication` UML diagram captures the centralized lifecycle and background-task orchestration within the Zenith-Connect system. The `ConnectApplication` class functions as the global entry point for application-wide configuration, monitoring activity transitions to determine whether the user is active or inactive, and coordinating periodic background operations. Through `WorkManager`, it schedules two recurring worker tasks, `DailyNotificationWorker` for delivering personalized event recommendations and `LotteryWorker` for executing automated lottery draws once registration deadlines pass. Multiplicity indicators reflect the one-to-one scheduling relationship between the application and each worker, emphasizing that each automated process is globally managed rather than activity-specific. The diagram also highlights the application's use of `UserActivityTracker` to update user activity status as lifecycle callbacks occur.