COMP4107 Event-Management-System Feature Implementation Document

Developer: YU Fengfei-21251215

Introduction

This document outlines the implementation details of key features in the Events Management System Android application. It serves to provide a clear understanding of how each component of the app functions and interacts with other modules.

Feature Implementation

KtorClient()

`KtorClient` is responsible for all network operations within the Events Management System app. It utilizes Ktor, a Kotlin-based client for making HTTP requests, and is configured to handle JSON data and manage authentication tokens.

Features

- **JSON Serialization**: Uses Kotlinx serialization for converting Kotlin objects to and from JSON.
- **Authentication Handling**: Manages an authentication token for secure API calls and provides a mechanism to clear tokens when logging out.
- **Error Handling**: Implements error handling to manage exceptions during network calls.
- **Dynamic Content Negotiation**: Configured to accept and content-type JSON for all requests.
- **Logging**: Includes a logging mechanism to debug HTTP requests and responses.

Implementation Details

- **Base Configuration**:
    - The client is configured to use content negotiation with JSON and a logging feature for debugging.
    - All requests include an `Authorization` header if a token is present.
- **API Endpoints**:
    - **getEvents(page: Int)**: Fetches a list of events based on the page number.
    - **getEventsByLocation(page: Int, location: Int)**: Retrieves events filtered by location.
    - **getEventById(_id: String)**: Fetches details for a single event by its ID.
    - **getRegisteredEvents()**: Retrieves a list of event IDs that the user has registered for.
    - **getEventBySearch(search: String, page: Int)**: Searches for events based on a query string.
    - **getUserEvents(page: Int)**: Fetches events related to the logged-in user.
    - **volunteerRegistration(information: Volunteer)**: Registers a user as a volunteer.
    - **JoinEvent(_id: String)**: Registers the user for a specific event.
    - **DeleteEvent(_id: String)**: Unregisters the user from an event.

- **getLoginToken(email: String, password: String)**: Authenticates the user and retrieves a token.
- **clearToken()**: Clears the stored authentication token, typically used during the logout process to ensure the session is completely terminated.

Error Handling

- All network requests are wrapped in try-catch blocks to handle exceptions, ensuring that the app can gracefully handle network errors or data parsing issues.

ScaffoldScreen()

`ScaffoldScreen` serves as the central navigation hub of the Events Management System Android application. It uses Jetpack Compose's `Scaffold` component to create a dynamic user interface that adapts based on the user's authentication status.

Features

- **Dynamic Navigation Bar**: The navigation bar updates its items and icons depending on whether the user is logged in. It supports different views like Home, Events, Search, User, and Login.
- **Conditional Content**: Displays different screen contents based on the current navigation item selected.
- **User Authentication State**: Manages the user's login state by introducting a variable `isLoggedIn` to toggle between authenticated and non-authenticated views.
- **Dynamic Screen Titles**: Updates the title in the `TopAppBar` dynamically based on the active screen.
- **Integration with Backend Services**: Utilizes `KtorClient` to fetch event data and user-specific details.
- **Snackbar Notifications**: Provides feedback to the user through snackbar messages for various operations like login success or errors.

Navigation Routes

Home Screen

- **Route**: "home"
- **Purpose**: Serves as the landing page or the main dashboard for the application, displaying all events.
- **Parameters**:
  - `navController`: For further navigation actions.
  - `feeds`: A list of events fetched asynchronously, which include events in the first page as `initial events`.
  - `isLoggedIn`: Indicates if the user is logged in, affecting the data presented (e.g., registration enabling).

Events Screen

- **Route**: "events"
- **Purpose**: Lists all events ordered by location number (`from 1 to 9`), allowing users to browse or search events based on location number.

- **Parameters**:
  - `navController`: Enables navigation to specific event details.
  - `isLoggedIn`: Used to determine if additional options like registration should be shown.

Search Screen

- **Route**: "search"
- **Purpose**: Allows users to search for customized events.
- **Parameters**:
  - `navController`: Needed for detailed event queries.
  - `feeds`: The results of the search, filtered by the user given query.
  - `isLoggedIn`: Used to determine if additional options like registration should be shown.

Login Screen

- **Route**: "login"
- **Purpose**: Manages user authentication.
- **Parameters**:
  - `navController`: Utilized for redirecting the user post-login.
  - `snackbarHostState`: Used to display messages about login success or issues.

User Screen

- **Route**: "user"
- **Purpose**: Displays user-specific events and interactions.
- **Parameters**:
  - `navController`: Allows navigation to detailed settings or event pages.
  - `userEvents`: A list of events that the user has participated in.

Become a Volunteer Screen

- **Route**: "becomeVolunteer"
- **Purpose**: Provides an interface for users to register as volunteers for events.
- **Parameters**:
  - `navController`: For navigation purposes post-registration.
  - `snackbarHostState`: To notify about registration status or errors.

Events by Location Screen

- **Route**: "eventsByLocation/{locationNumber}/{isLoggedIn}"
- **Purpose**: Shows events based on a specific location.
- **Parameters**:
  - `locationNumber`: An integer representing a specific location ID, used to fetch relevant events.

- `locationEvents`: A list of events at the specified location.
- `isLoggedIn`: Used to determine if additional options like registration should be shown.

Event Details Screen

- **Route**: "eventDetails/{_id}/{isLoggedIn}"
- **Purpose**: Provides detailed information about a specific event.
- **Parameters**:
  - `_id`: The unique identifier for an event, crucial for `fetching`, `joining` and `unregistering` specific event.
  - `isLoggedIn`: Used to determine if additional options like registration should be shown.

Logout

- **Route**: "logout"
- **Purpose**: Set the `isLoggedIn` variable to `false` and clear token.
- **Parameters**: None

Example Usage

```
@Composable
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Events_Management_SystemTheme(false) {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = Color.Black
                ) {
                    ScaffoldScreen()
                }
            }
        }
    }
}
```

BecomeVolunteerScreen()

`BecomeVolunteerScreen` is designed to allow users to register as volunteers within the Events Management System. It provides a comprehensive form that users can fill out to submit their details, which are then processed and recorded in the system.

Features

- **User Input Forms**: Collects essential information from users such as email, password, name, contact, age group, and additional remarks.
- **Age Group Selection**: Offers a dropdown menu for selecting age groups, ensuring that users can categorize themselves appropriately.
- **Password Visibility Toggle**: Allows users to toggle the visibility of their password for better security practices.
- **Terms and Conditions Agreement**: Includes a checkbox for users to agree to the terms and conditions before registering.
- **Snackbar Notifications**: Utilizes snackbar messages to provide feedback about form validations or registration status.

Implementation Details

- **Form Handling**:
  - Uses state variables (`remember`, `mutableStateOf`) to manage form inputs and UI component states.
  - Implements form validations to check for `empty fields`, `email format`, and `terms agreement` before allowing the registration process to proceed.
  - If successfully registered, the user will be navigated to the `Login` screen. Otherwise, error messages will be prompted.
- **Registration Process**:
  - Triggers a registration function from `KtorClient` upon successful form validation.
  - Navigates to the login screen post-registration, suggesting successful volunteer registration.
- **UI Components**:
  - `TextField` components for input fields with specific keyboard options for better usability (e.g., `ImeAction.Next` for smooth form navigation).
  - `ExposedDropdownMenuBox` for age group selection, enhancing the form with a modern, user-friendly dropdown selector.
  - `Button` to submit the registration form, with visual feedback adjustments (e.g., changing colors and displaying error messages via snackbar).

Example Usage

Below is an example snippet on how the `BecomeVolunteerScreen` can be used within the app's navigation:

```
@Composable
composable("becomeVolunteer") {
    BecomeVolunteer(navController, snackbarHostState)
    title = "Become a volunteer"
}
```

EventsByLocation() in `EventsBylocation.kt`

`EventsByLocation` provides a navigable list of events specific to a chosen geographical location. It dynamically fetches and displays events based on user selections, offering incremental data loading to enhance the user experience.

Features

- **Dynamic Event Loading**: Fetches events specific to a chosen location with the ability to load more events on demand.
- **Interactive List**: Each event is clickable and directs the user to detailed event information.
- **Progress Indicators**: Displays a loading indicator while fetching additional events, providing a clear user feedback mechanism.
- **Pagination Support**: Implements pagination to manage and fetch additional events without overwhelming the client or server.

Implementation Details

- **State Management**:
  - Uses `remember` and `mutableStateOf` to manage the state of events and loading indicators.
  - Maintains pagination through a page state that triggers fetching additional events when incremented.
- **Event Fetching**:
  - Triggers an asynchronous fetch (`KtorClient.getEventsByLocation`) within a `LaunchedEffect` based on changes to `locationNumber` or `page`, ensuring that updates to these parameters efficiently reload data.
  - Conditionally accumulates events to the existing list or replaces them based on the pagination behavior.
- **User Interaction**:
  - Uses a `LazyColumn` for efficient list rendering, suitable for potentially long lists of event data.
  - Implements clickable events that route to a detailed view (`eventDetails`), passing necessary parameters like event ID and user login status.
- **UI Components**:
  - `CircularProgressIndicator` to inform users of ongoing background operations.
  - `Button` to load more events, enhancing usability by allowing manual control over data fetching.

Example Usage

Below is an example of how `EventsByLocation` can be integrated into an app's navigation setup:

```
composable(
    "eventsByLocation/{locationNumber}/{isLoggedIn}",
    arguments = listOf(
        navArgument("locationNumber") { type = NavType.IntType },
        navArgument("isLoggedIn") { type = NavType.BoolType })
```

```
) { backStackEntry ->
    val locationNumber = backStackEntry.arguments?.getInt("locationNumber") ?: 1
    val login = backStackEntry.arguments?.getBoolean("isLoggedIn") ?: false
    EventsByLocation(navController, locationNumber, locationEvents, login)
    title = "Address ${locationNumber}"
}
```

LocationList() in `EventsBylocation.kt`

`LocationList` is a simple yet functional component within the Events Management System that provides a navigable list of predefined location numbers, allowing users to select a location and view events associated with it.

Features

- **Navigable List of Locations**: Displays a `list of location numbers`, each `clickable` and leading to the `EventsByLocation` screen, showing events specific to the chosen location.
- **Simple and Efficient UI**: Utilizes a `LazyColumn` for efficient rendering of list items, suitable for displaying an expandable list of locations.

Implementation Details

- **User Interaction**:
  - Each list item is clickable and tied to navigation actions, using `navController` to navigate to the `EventsByLocation` screen with relevant parameters.
  - Incorporates the `Divider` component to visually separate each location item, enhancing the clarity and usability of the list.
- **Navigation Parameters**:
  - Passes the `isLoggedIn` status and location number to the `EventsByLocation` composable as parameters, ensuring that the user experience is consistent and personalized based on their authentication status.

Example Usage

Below is an example of how `LocationList` can be integrated into an app's navigation flow:

```
composable("events") {
    LocationList(navController, isLoggedIn)
    title = "Events"
}
```

HomeScreen

`HomeScreen` is the main dashboard of the Events Management System application. It prominently displays events, with a focus on highlighted events, and provides an engaging user interface complete with images and detailed event information.

Features

- **Dynamic Event Loading**: Automatically fetches new events as users scroll, enhancing user engagement with a `"Load More"` button for dynamic pagination.
- **Highlight Priority Display**: Events are sorted by their highlight status, ensuring that highlighted events appear first, making them more visible and accessible.
- **High-Quality Event Presentation**: Uses `Card` components to display events attractively, including images and key details.
- **Interactive List**: `Each event card is clickable`, leading to detailed event information, fostering deeper user interaction.
- **Visual and Informational Clarity**: Utilizes image loading with `AsyncImage` for visual appeal and `Text` components to convey event details clearly.

Implementation Details

- **Event Sorting and Loading**:
  - Events are automatically sorted so that `highlighted events are displayed at the top of the list`. This sorting is based on the `highlight` attribute of the events, enhancing the visibility of key events.
  - Incorporates a lazy loading mechanism with pagination handled by incrementing a `page` state. This triggers further fetching of events through `KtorClient.getEvents` when users interact with the `"Load More"` button.
- **User Interaction**:
  - Implements a `LazyColumn` to efficiently manage and render a potentially large list of events.
  - Uses `Card` for each event, making it visually distinct and easy to interact with, linking further navigation to detailed views via `NavController`.
- **Loading Indicators and Error Handling**:
  - Shows a `CircularProgressIndicator` while fetching data, providing feedback during loading operations.
  - Includes error handling mechanisms to address loading issues or failed data fetch attempts.

Example Usage

Below is an example of how `HomeScreen` can be utilized within an app's structure:

```
composable("home") {
    HomeScreen(navController, feeds, isLoggedIn)
    title = "Events Management System"
}
```

LoginScreen

`LoginPage` provides a secure and intuitive interface for users to log into the Events Management System. It handles `user authentication`, `displaying appropriate feedback` and `navigating the user` upon successful login.

Features

- **Secure Authentication**: Facilitates user login with email and password verification.
- **Dynamic UI Feedback**: Displays loading indicators and relevant messages during the authentication process.
- **Password Visibility Toggle**: Allows users to toggle the visibility of their password for enhanced security during input.
- **Navigation Post-Login**: Redirects users to the user profile screen upon successful authentication.

Implementation Details

- **Authentication Process**:
  - Utilizes `KtorClient.getLoginToken` to authenticate users against stored credentials.
  - Upon successful login, triggers a snackbar notification and a callback function `onLoginSuccess` to update the user's login status `isLoggedIn` Boolean variable in the `ScaffoldScreen`.
- **State Management**:
  - Manages the email, password, and loading state using `remember` and `mutableStateOf`.
  - `isLoading` flag controls UI components during the login process, disabling input fields and showing a progress indicator.
- **Interaction with ScaffoldScreen**:
  - After a successful login, `onLoginSuccess` is executed, which is typically used to set `isLoggedIn` to `true` within `ScaffoldScreen`. This state change triggers a UI update across the application, enabling access to user-specific data.
  - Navigates away from the login page to the `user` screen, effectively resetting the navigation stack to the start destination to prevent back navigation to the login screen.

Example Usage

Below is an example of how `LoginPage` can be used within an app's navigation setup:

```
composable("login") {
    LoginPage(navController, snackbarHostState) {
        isLoggedIn = true
        initialRegisteredEventsId = KtorClient.getRegisteredEvents()
        userEvents = KtorClient.getUserEvents(1)
    }
    title = "Log In"
}
```

UserScreen

`UserScreen` serves as the personalized dashboard for logged-in users within the Events Management System. It not only displays events that the user participated in but also provides a logout function, allowing users to securely exit their session.

Features

- **Personalized Event Display**: Lists events specifically associated with the logged-in user, enhancing the personalized experience.
- **Dynamic Event Loading**: Implements lazy loading of events with a `"Load More"` button, allowing for the dynamic fetching of additional events as the user scrolls.
- **Logout Functionality**: Provides a secure and straightforward way for users to log out of the application.

Implementation Details

- **Event Fetching and State Management**:
  - Uses `KtorClient.getUserEvents` to fetch user-specific events, managing pagination with a reactive `page` state that triggers further fetching when incremented.
  - Manages the list of events and loading state using `remember` and `mutableStateOf`, ensuring the UI reflects the current state accurately.
- **Logout Process**:
  - The logout button triggers a network call to `KtorClient.clearToken()` to securely end the user session on the backend.
  - Updates the frontend state to reflect the logout, showing a snackbar message for confirmation and navigating the user to a specified route (typically a login or landing page) while resetting the navigation stack to prevent back-navigation to authenticated screens.
- **User Interface**:
  - Displays a `logout button` at the top of the screen for easy access. The button is styled distinctly to draw attention and ensure that users are aware of the action.
  - Lists events in a `LazyColumn`, using `Card` components for each event to provide a clean and engaging interface.
  - Includes a `"Load More"` button at the end of the list to fetch additional events, with a `CircularProgressIndicator` shown during data fetching.

Example Usage

Below is an example of how `UserScreen` can be integrated within an app:

```
composable("user") {
    UserScreen(navController, userEvents)
    title = "User"
}
```

EventDetail

`EventDetail` provides a detailed view of individual events within the Events Management System, enabling logged-in users to `join` or `unregister` from events based on their registration status and the event's availability.

Features

- **Comprehensive Event Information**: Displays detailed information about an event, including images, title, organizer, description, date, location, and quota.
- **Dynamic Interaction**: Offers `"Join Event"` or `"Unregister"` buttons dynamically based on the user's registration status and the event's remaining quota.
- **Real-Time Data Synchronization**: Ensures that the registration status is always current by updating the `registeredEventsId` list upon each component invocation.

Implementation Details

- **Event Registration Operations**:
    - **Join Event**: Allows a user who is not already registered and where quota is available, to join the event. This action verifies the quota and, upon success, updates the user's registration status, displays a confirmation message, and navigates back to the home screen.
    - **Unregister Event**: Permits a user to cancel their registration. Successful unregistration triggers a status update, a confirmation message, and navigation back to the home screen.
- **State and Data Management**:
    - **Event and Registration Data Fetching**: Initially fetches detailed event data from `KtorClient.getEventById` and checks registration status through `KtorClient.getRegisteredEvents` to synchronize the user's current event registrations.
    - **State Updates**: Uses `remember` and `mutableStateOf` to handle state such as event details, registration changes, and loading states. Ensures that `registeredEventsId` is refreshed each time the component is rendered to prevent data discrepancies and errors.
- **User Interface**:
    - Displays event details in a `LazyColumn` for efficient scrolling and data handling.
    - Conditional rendering of "Join Event" or "Unregister" buttons based on the user's registration status and event quota, with the visibility and actions driven by the latest registration data.

Example Usage

Below is an example of how `EventDetail` might be integrated within an app:

```
composable(
    "eventDetails/{_id}/{isLoggedIn}",
    arguments = listOf(
        navArgument("_id") { type = NavType.StringType },
        navArgument("isLoggedIn") { type = NavType.BoolType }
    )
) { backStackEntry ->
    val _id = backStackEntry.arguments?.getString("_id") ?: ""
    val login = backStackEntry.arguments?.getBoolean("isLoggedIn") ?: false
    EventDetail(
        navController = navController,
        snackbarHostState = snackbarHostState,
        _id = _id,
        login = login,  // 假设已登录
        initialRegisteredEventsId = listOf(),
        selectedItem = selectedItem,
        setSelectedItem = { newItem -> selectedItem = newItem }
    )
    title = "Event Detail"
}
```

Limitations

Lack of In-App Navigation Controls

Currently, the app lacks dedicated back navigation controls within its interface, relying solely on the device's operating system back button. This may lead to a less intuitive user experience, as users may not have an obvious way to navigate back through the app's screens without using their device's hardware or system gestures.

User Interface Aesthetics

While functional, the user interface of the app can be further enhanced to improve visual appeal and user engagement. An example improvement could be the addition of a dedicated user information page, which would not only beautify the interface but also provide users with a more personalized experience.

Response Times for User Actions

The response times for critical user interactions such as `login`, `logout`, `joining an event`, `deleting an event`, and `registering as a volunteer` are slightly longer than ideal. This can negatively impact the user experience, making the app feel slower and less responsive than desired.

Navigation Bar Bugs

There is a bug in the navigation bar concerning the updating of the selected item state. This issue can lead to incorrect display of which page or section is active, potentially confusing users and detracting from the navigational experience.

These limitations highlight areas for potential improvement and should be addressed in future updates to enhance functionality, aesthetics, and overall user satisfaction.