ADHD Task Manager

Programmer Document

ITC309 Team 1

Programmer Documentation

Table Of Contents

AndroidManifest

The application uses the following permissions:

- `android.permission.POST_NOTIFICATIONS`: This permission allows the app to post notifications to the user's device.
- `android.permission.RECEIVE_BOOT_COMPLETED`: This permission allows the app to receive a broadcast event when the device boots up. This allows the app to schedule alarms for any upcoming todos that the user has created.
- `android.permission.ACCESS_NOTIFICATION_POLICY`: This permission allows the app to access the user's notification policy. This is necessary for the app to be able to post notifications.
- `android.permission.SCHEDULE_EXACT_ALARM`: This permission allows the app to schedule alarms that will fire at exactly the specified time.
- `android.permission.USE_EXACT_ALARM`: This permission allows the app to use alarms that will fire at exactly the specified time.

The manifest file declares two activities:

- `ADHDTaskManager`: This is the main activity for the app. It is the activity that is launched when the user taps on the app's icon.
- `MainActivity`: This is another activity that is used by the app. It is not clear what this activity is used for from the manifest file alone.

The manifest file also declares a receiver:

- `AlarmReceiver`: This receiver is used to receive alarm events that have been scheduled by the app. When the receiver receives an alarm event,it will display a notification to the user.

MainActivity

The `MainActivity` class is responsible for setting up the application UI and handling user interactions.

The `MainActivity` class first creates a number of view models, including a `TodoViewModel`, a `RewardViewModel`, and a `LeaderboardViewModel`. These view models are responsible for managing the application's data and state.

The `MainActivity` class also creates a `NavHostController` object. This object is responsible for managing the application's navigation.

The `MainActivity` class then implements the `onCreate()` method. This method is called when the activity is first created.

In the `onCreate()` method, the `MainActivity` class performs the following steps:

1. Initializes the `AlarmSchedulerImpl` class. This class is responsible for scheduling alarms for the application.
2. Adds an `AuthStateListener` to the `googleAuthUiClient` object. This listener is called when the user's authentication state changes.
3. Fetches the default profile image from Firebase Storage.
4. Inflates the `custom_toast` layout and displays it as a Toast.
5. Sets the content of the activity to the `ADHDTaskManagerTheme` layout.

The `MainActivity` class creates the navigation drawer that contains all of the application screens as well as a button for signing out and handles user interactions, such as opening the drawer and navigating between screens.

The `ModalNavigationDrawer` takes the following arguments:

- `drawerState`: The `DrawerState` object that is used to control the drawer's state.
- `gesturesEnabled`: A Boolean value that indicates whether or not gestures are enabled for the drawer.
- `drawerContent`: The content of the drawer.

The `ModalNavigationDrawer` then creates a `ModalDrawerSheet` object. This object is responsible for rendering the drawer.

The `ModalDrawerSheet` object takes the following arguments:

- `drawerContainerColor`: The colour of the drawer's container.
- `drawerTonalElevation`: The elevation of the drawer.
- `modifier`: A `Modifier` object that can be used to modify the drawer's appearance.
- `content`: The content of the drawer.

The `ModalDrawerSheet` then creates a `LazyColumn` object. This object is used to display the list of drawer items.

The `LazyColumn` object takes the following arguments:

- `content`: The content of the list.

The `LazyColumn` then iterates over the list of screens and creates a `NavigationDrawerItem` object for each screen.

The `NavigationDrawerItem` object takes the following arguments:

- `colors`: A `NavigationDrawerItemDefaults.colors()` object that is used to specify the colours of the item.
- `icon`: The icon for the item.
- `label`: The label for the item.
- `selected`: A Boolean value that indicates whether or not the item is selected.
- `onClick`: A callback function that is called when the item is clicked.

The `NavigationDrawerItem` then creates a row that contains the item's icon, label, and any other desired content.

The `NavigationDrawerItem` also adds a click listener to the item. When the item is clicked, the `selectedItem.value` variable is updated to the selected screen. The `drawerState.close()` method is then called to close the drawer. Finally, the `navController.navigate()` method is called to navigate to the selected screen.

The `ModalNavigationDrawer` also includes a `NavigationDrawerItem` object for signing out. When this item is clicked, the `googleAuthUiClient.signOut()` method is called to sign the user out. A Toast is then displayed to inform the user that they have been signed out.

If the user is signed in anonymously, the `ModalNavigationDrawer` also displays a row that warns the user that they will lose all of their data if they sign out.

The `NavHost` section of the `MainActivity` class is responsible for handling the navigation between the different screens in the application.

The `NavHost` takes the following arguments:

- `navController`: The `NavController` object that is used to control the navigation.
- `startDestination`: The route of the screen that should be displayed initially.

The `NavHost` then defines a number of composable functions, one for each screen in the application.

Each composable function takes a number of arguments, including the state of the screen and any event handlers that are needed.

The composable functions then render the corresponding screens.

The `NavHost` also includes a `LaunchedEffect` block that is used to navigate to the `todo_screen` if the user is already signed in.

Here is a brief overview of the different screens in the application:

- `TodoScreen`: This screen displays a list of all of the user's to-dos.
- `SettingsScreen`: This screen allows the user to configure the application's settings.
- `LeaderboardScreen`: This screen displays a leaderboard of the top users in the application.
- `RewardsScreen`: This screen displays the user's rewards.
- `CompletedScreen`: This screen displays a list of all of the user's completed to-dos.
- `SignInScreen`: This screen allows the user to sign in to their account.
- `HelpScreen`: This screen displays the application's help documentation.
- `PomodoroTimerScreen`: This screen displays a Pomodoro timer.

The `getResponseUsingCallback()` function in the `MainActivity` class is responsible for retrieving the list of users from the Firebase database using a callback function.

The `getResponseUsingCallback()` function takes the following arguments:

- `callback`: A callback function that will be called when the response from the database is received.

The `getResponseUsingCallback()` function then calls the `getResponseUsingCallback()` method on the `leadViewModel` object. The `leadViewModel` object is responsible for handling all interactions with the Firebase database.

The `leadViewModel` object's `getResponseUsingCallback()` method takes the following arguments:

- `callback`: A callback function that will be called when the response from the database is received.

The `leadViewModel` object's `getResponseUsingCallback()` method then makes a request to the Firebase database to retrieve the list of users. When the response from the database is received, the callback function is called with the response.

The callback function passed to the `getResponseUsingCallback()` function in the `MainActivity` class is responsible for parsing the response from the database and updating the UI with the list of users.

ADHDTaskManager

The `ADHDTaskManager` class is an activity that displays a Lottie animation while the application is loading. It inherits from the `AppCompatActivity` class and overrides the `onCreate()` method.

The `onCreate()` method sets the content view of the activity to the `activity_video_splash` layout. This layout contains a `LottieAnimationView` widget.

The `onCreate()` method also adds an animator listener to the `LottieAnimationView` widget. This listener is called when the animation starts, ends, cancels or repeats.

When the animation ends, the `startActivity()` method is called to start the `MainActivity` activity. The `finish()` method is also called to finish the current activity.

Here is an example of how to use the `ADHDTaskManager` class:

```
// Create an intent to start the ADHDTaskManager activity.
val intent = Intent(this@MyApp, ADHDTaskManager::class.java)

// Start the ADHDTaskManager activity.
startActivity(intent)
```

This will start the `ADHDTaskManager` activity and display the Lottie animation while the application is loading. Once the animation ends, the `MainActivity` activity will be started.

UI

completed_screen

CompletedScreen

The `CompletedScreen` composable function is responsible for rendering the completed tasks screen.

The `CompletedScreen` composable function takes the following arguments:

- `state`: The state of the TodoViewModel.
- `scope`: A CoroutineScope object.
- `drawerState`: A DrawerState object.

The `CompletedScreen` composable function then creates a `Scaffold` object. The `Scaffold` object provides a default layout for the screen, including a top bar and a content area.

The `TopAppBar` object in the top bar provides a title for the screen and a button for opening the drawer.

The `content` section of the `Scaffold` object contains a `LazyColumn` object. The `LazyColumn` object is used to display a list of completed tasks.

The `LazyColumn` object iterates over the list of todos in the state and displays a `CompletedTaskCard` object for each completed task.

The `CompletedTaskCard` object is a composable function that is responsible for rendering a single completed task.

dialogs

AddEditTodoDialog

The `AddEditTodoDialog` composable function is responsible for rendering the add/edit todo dialog.

The `AddEditTodoDialog` composable function takes the following arguments:

- `state`: The state of the TodoViewModel.

- `onEvent`: A callback function that is called when the user interacts with the dialog.
- `scope`: A CoroutineScope object.
- `sheetState`: A SheetState object.
- `alarmScheduler`: An AlarmSchedulerImpl object.

The `AddEditTodoDialog` composable function first checks to see if the `showEditTodoDialog` property in the state is set to `true`. If it is, then the function sets the `thisTodo` variable to the todo that is currently being edited.

The function then creates a `Column` object to display the contents of the dialog. The `Column` object contains the following components:

- A `Text` object to display the title of the dialog.
- A `TextField` object for the user to enter the title of the todo.
- A `TextField` object for the user to enter the description of the todo.
- A `ExposedDropdownMenuBox` object for the user to select the priority of the todo.
- A `DatePicker` object for the user to select the due date of the todo.
- A `TimePicker` object for the user to select the due time of the todo.

The `AddEditTodoDialog` composable function also creates a `Button` object for the user to save the todo. When the user clicks the button, the function calls the `onEvent()` callback function with a `TodoEvent` object that contains the updated todo information.

The `DatePickerDialog` is configured with the following properties:

- `onDismissRequest`: A callback function that is called when the user tries to dismiss the dialog.
- `confirmButton`: A `Button` object that the user can click to confirm the selected date.
- `dismissButton`: A `Button` object that the user can click to dismiss the dialog without confirming the selected date.
- `shape`: The shape of the dialog.
- `content`: The content of the dialog.

The `content` of the dialog is a `DatePicker` composable function that is configured with the following properties:

- `state`: The state of the date picker.

- `showModeToggle`: A boolean value that indicates whether the show mode toggle should be displayed.
- `title`: The title of the date picker.

The `DatePicker` composable function is configured with the `datePickerState` object if the `showDialog` property in the state is set to `true`, or the `editDatePickerState` object if the `showEditTodoDialog` property in the state is set to `true`. Otherwise, the `DatePicker` composable function is configured with the `datePickerState` object.

When the user clicks the "Confirm" button on the `DatePickerDialog`, the following code is executed:

```
if (state.showDialog) {
selectedDate = datePickerState.selectedDateMillis?.let {
Instant.ofEpochMilli(it).atOffset(ZoneOffset.UTC) }
Log.d("Date", selectedDate?.format(
DateTimeFormatter.ofPattern("dd-MM-yyyy")).toString()
) onEvent(TodoEvent.setDueDate(selectedDate?.format(
DateTimeFormatter.ofPattern("dd-MM-yyyy")).toString())
)
onEvent(TodoEvent.hideDateSelector) } else if
(state.showEditTodoDialog){
editedSelectedDate = editDatePickerState.selectedDateMillis?.let {
Instant.ofEpochMilli(it).atOffset(ZoneOffset.UTC)
} onEvent(TodoEvent.setDueDate(editedSelectedDate?.format(
DateTimeFormatter.ofPattern("dd-MM-yyyy")).toString())
)
thisTodo?.dueDate = editedSelectedDate?.format(
DateTimeFormatter.ofPattern("dd-MM-yyyy")).toString()
onEvent(TodoEvent.hideEditDateSelector)
}
```

This code gets the selected date from the `DatePickerDialog` and then calls the `onEvent()` callback function with a `TodoEvent` object that contains the selected date. The `onEvent()` callback function is responsible for updating the state of the TodoViewModel with the selected date.

The `DatePickerDialog` is configured with the following properties:

- `onDismissRequest`: A callback function that is called when the user tries to dismiss the dialog.
- `confirmButton`: A `Button` object that the user can click to confirm the selected time.
- `dismissButton`: A `Button` object that the user can click to dismiss the dialog without confirming the selected time.
- `shape`: The shape of the dialog.
- `content`: The content of the dialog.

The `content` of the dialog is a `Column` object that contains a `TimeInput` composable function. The `TimeInput` composable function is configured with the following properties:

- `colors`: The colors of the time input.
- `state`: The state of the time input.
- `layoutType`: The layout type of the time input.

The `TimeInput` composable function is configured with the `timePickerState` object if the `showDialog` property in the state is set to `true`, or the `editTimePickerState` object if the `showEditTodoDialog` property in the state is set to `true`. Otherwise, the `TimeInput` composable function is configured with the `timePickerState` object.

When the user clicks the "Confirm" button on the `DatePickerDialog`, the following code is executed:

```
if (state.showDialog) {
cal.isLenient = false cal.timeZone = TimeZone.getDefault()
cal.set(Calendar.HOUR_OF_DAY, timePickerState.hour)
cal.set(Calendar.MINUTE, timePickerState.minute)
onEvent(TodoEvent.setDueTime(timeFormatter.format(cal.time)))
onEvent(TodoEvent.hideTimeSelector)
} else { editCal.isLenient = false
editCal.timeZone = TimeZone.getDefault()
editCal.set(Calendar.HOUR_OF_DAY, editTimePickerState.hour)
editCal.set(Calendar.MINUTE, editTimePickerState.minute)
onEvent(TodoEvent.setDueTime(timeFormatter.format(editCal.time)))
thisTodo?.dueTime = timeFormatter.format(editCal.time)
onEvent(TodoEvent.hideEditTimeSelector)
}
```

This code gets the selected time from the `DatePickerDialog` and then calls the `onEvent()` callback function with a `TodoEvent` object that contains the selected time. The `onEvent()` callback function is responsible for updating the state of the TodoViewModel with the selected time.

The dialog contains a `Button` object that the user can click to cancel the add/edit todo dialog. If the user clicks the cancel button, the code calls the `onEvent()` callback function with a `TodoEvent.resetState` event. The code also calls the `sheetState.hide()` coroutine function to hide the sheet.

If the user clicks the "Add Task" or "Edit Task" button and the title and description fields are not empty, the code performs the following actions:

1. If the user is adding a new todo, the code calls the `onEvent()` callback function with a `TodoEvent.saveTodo` event.
2. If the user is editing an existing todo, the code calls the `onEvent()`callback function with a `TodoEvent.updateTodo()` event.
3. If the due date and due time fields are not empty, the code creates an `AlarmItem` object and schedules the alarm using the `alarmScheduler.schedule()` function.
4. The code calls the `sheetState.hide()` coroutine function to hide the sheet.

The dialog also contains code to handle title and description errors. If the title field is empty, the code calls the `onEvent()` callback function with a `TodoEvent.titleError()` event with the `true` parameter. If the description field is empty, the code calls the `onEvent()` callback function with a `TodoEvent.descriptionError()` event with the `true` parameter.

help_screen

HelpScreen

The `HelpScreen` composable function displays a help screen with three options: "View Manual", "View Architecture Diagram", and "View the Video". When the user clicks on one of the options, the code opens a WebViewActivity to display the relevant content.

Parameters:

- `scope`: A `CoroutineScope` object that is used to launch the WebViewActivity.
- `drawerState`: A `DrawerState` object that is used to control the state of the drawer.

WebViewActivity

This activity displays a WebView to display the content specified by the URL provided in the intent extras.

The WebViewActivity has the following methods:

- `onCreate()`: This method is called when the activity is first created. It initializes the WebView and loads the URL specified in the intent extras.
- `shouldOverrideUrlLoading()`: This method is called when the WebView is about to load a URL. It returns `true` if the activity should handle the URL loading, or `false` if the WebView should handle the URL loading.
- `onWebChromeClient()`: This method sets a custom WebChromeClient to handle video playback.

leaderboard_screen

Leaderboard

The `usersList()` function takes a `Response` object as input and parses it to extract the list of users. If the response contains a list of users, the function adds each user to the `Final.users` arraylist and logs the user's display name. If the response contains an exception, the function logs the exception message.

- `response.users?.let { users -> ... }`: This block of code is executed only if the `users` property of the `response` object is not null.
- `users.forEach{ user -> ... }`: This loop iterates over the list of users and executes the code block inside the loop for each user.
- `Final.addToList(user)`: This function adds the `user` object to the `Final.users` arraylist.
- `user.displayName?.let { Log.i(TAG, it) }`: This block of code is executed only if the `displayName` property of the `user` object is not null. The code block logs the user's display name to the logcat.
- `response.exception?.message?.let { Log.e(TAG, it) }`: This block of code is executed only if the `exception` property of the `response` object is not null. The code block logs the exception message to the logcat.

The `LeaderboardItem()` composable function takes a `Users` object and an `Int` rank as input and displays a leaderboard item for the user. The leaderboard item includes the user's rank, display name, country, and points.

- `Row()`: This composable function displays its children in a horizontal row.
- `Modifier.fillMaxWidth()`: This modifier makes the row fill the maximum available width.
- `Modifier.height(45.dp)`: This modifier sets the height of the row to 45 device-independent pixels.
- `Modifier.padding(start = 10.dp)`: This modifier adds 10 device-independent pixels of padding to the left side of the row.
- `Text()`: This composable function displays a text string.
- `LeaderboardBlue`: This is a custom color variable that is defined as `Color(0xFF045EA5)`.

LeaderboardScreen

The `LeaderboardScreen` composable function displays a leaderboard of users, sorted by their points. It takes the following parameters:

- `connectivityObserver`: A `ConnectivityObserverImpl` object that is used to observe the device's network connectivity status.
- `scope`: A `CoroutineScope` object that is used to launch coroutines.
- `drawerState`: A `DrawerState` object that is used to control the state of the app drawer.
- `defaultImageUrl:` This variable defines the default profile image URL that is used if the user does not have a profile image.
- `usersList`: This variable gets the list of users from the `Final` class.
- `sortedList:` This variable sorts the list of users by points.
- `Observer:` This variable observes the device's network connectivity status using the `ConnectivityObserver` object.
- `when(observer){ ... }`: This `when` statement displays the leaderboard of users if the device is connected to the internet, and displays a "No Internet

The function works as follows:

1. It gets the list of users from the `Final` class and sorts it by points.
2. It observes the device's network connectivity status using the `ConnectivityObserver` object.
3. It displays a leaderboard of users if the device is connected to the internet.
4. It displays a "No Internet" screen if the device is not connected to the internet.

LeaderboardViewModel

The `LeaderboardViewModel` class is a ViewModel that provides data for the leaderboard screen. It has a single function, `getResponseUsingCallback()`, which takes a `FirebaseCallback` as input and invokes it when the response from the Firebase database is received.

The `FirebaseCallback` interface is a simple interface that has a single method, `onSuccess()`, which is invoked when the response from the Firebase database is successful.

pomodoro_timer

PomodoroTimerScreen

The `PomodoroTimerScreen` composable function displays a Pomodoro timer screen. The screen consists of a progress bar, a timer text, and two buttons: "Start" / “Pause” / “Resume” and "Stop".

Parameters:

- `settingsViewModel`: A `SettingsViewModel` object that is used to get the values of the work timer and break timer.
- `initialWorkTime`: The initial work time in milliseconds.
- `initialBreakTime`: The initial break time in milliseconds.
- `handleColor`: The color of the handle of the progress bar.
- `inactiveBarColor`: The color of the inactive part of the progress bar.
- `activeBarColor`: The color of the active part of the progress bar.
- `modifier`: A `Modifier` object that is used to modify the appearance of the composable function.
- `initialValue`: The initial progress of the progress bar.
- `strokeWidth`: The stroke width of the progress bar in `Dp`.
- `context`: A `Context` object that is used to get the system settings.
- `activity`: An `Activity` object that is used to toggle the Do Not Disturb mode.
- `scope`: A `CoroutineScope` object that is used to launch the coroutine that updates the progress bar and timer text.
- `drawerState`: A `DrawerState` object that is used to control the state of the drawer.

Usage:

To use the `PomodoroTimerScreen()` composable function, simply call it from your code. The function will display a Pomodoro timer screen with the specified initial work time, break time, handle color, inactive bar color, and active bar color.

Additional notes:

- The `PomodoroTimerScreen()` composable function uses a coroutine to update the progress bar and timer text. This means that the function will continue to update the progress bar and timer text even if the user is interacting with other parts of the screen.
- The `PomodoroTimerScreen()` composable function also toggles the Do Not Disturb mode when the timer is running. This prevents the user from receiving notifications while the timer is running.

reward_screen

RewardRepo

This class is a repository for reward data. It provides methods to insert, update, delete, and read reward data from the database. It also provides a `LiveData` object that emits a list of all rewards in the database.

Methods:

- `allRewards`: A `LiveData` object that emits a list of all rewards in the database.
- `searchResults`: A `MutableLiveData` object that emits a list of rewards that match the specified search term.
- `insertReward(newReward: Reward)`: Inserts a new reward into the database.
- `updateReward(updatedReward: Reward)`: Updates an existing reward in the database.
- `deleteReward(deletedReward: Reward)`: Deletes an existing reward from the database.
- `findReward(name: String)`: Returns a `LiveData` object that emits a list of rewards that match the specified search term.

Example:

```
val rewardDao = RewardDao()
```

```
val rewardRepo = RewardRepo(rewardDao)

// Insert a new reward into the database
val newReward = Reward(name = "New Reward", points = 100)
rewardRepo.insertReward(newReward)

// Update an existing reward in the database
val updatedReward = rewardDao.getReward(1)
updatedReward.points = 200
rewardRepo.updateReward(updatedReward)

// Delete an existing reward from the database
val deletedReward = rewardDao.getReward(2)
rewardRepo.deleteReward(deletedReward)

// Get a list of all rewards in the database
val allRewards = rewardRepo.allRewards

// Get a list of rewards that match the search term "New Reward"
val searchResults = rewardRepo.findReward("New Reward")
```

RewardsScreen

The `RewardsScreen` composable function displays a screen that shows the user's current points, completed task rewards, and login rewards. The screen is only displayed if the device is connected to the internet. Otherwise, a `NoInternetScreen()` is displayed.

Parameters:

- `rewardViewModel`: A `RewardViewModel` object that is used to get the list of all rewards.
- `usersViewModel`: A `UsersViewModel` object that is used to get the user's current points and login num.
- `connectivityObserver`: A `ConnectivityObserverImpl` object that is used to observe the device's internet connectivity status.
- `scope`: A `CoroutineScope` object that is used to launch the coroutine that updates the user's points.
- `drawerState`: A `DrawerState` object that is used to control the state of the drawer.

RewardViewModel

This class is a ViewModel for rewards data. It provides methods to insert, update, delete, and read reward data from the database. It also provides a `LiveData` object that emits a list of all rewards in the database.

Properties:

- `allRewards`: A `LiveData` object that emits a list of all rewards in the database.
- `searchResults`: A `LiveData` object that emits a list of rewards that match the specified search term.

Methods:

- `insertReward(reward: Reward)`: Inserts a new reward into the database.
- `findReward(name: String)`: Returns a `LiveData` object that emits a list of rewards that match the specified search term.
- `updateReward(reward: Reward)`: Updates an existing reward in the database.
- `deleteReward(reward: Reward)`: Deletes an existing reward from the database.
- `updateState(rewardName: String)`: Updates the state of the ViewModel to reflect the specified reward name.

Usage:

```
val rewardViewModel = RewardViewModel(application)

// Get a list of all rewards in the database
val allRewards = rewardViewModel.allRewards

// Get a list of rewards that match the search term "New Reward"
val searchResults = rewardViewModel.findReward("New Reward")

// Update the state of the ViewModel to reflect the reward name
"Completed Task Reward"
rewardViewModel.updateState("Completed Task Reward")

// Insert a new reward into the database
val newReward = Reward(name = "New Reward", points = 100)
```

```
rewardViewModel.insertReward(newReward)

// Update an existing reward in the database
val updatedReward = rewardViewModel.allRewards.value?.first()
updatedReward.points = 200
rewardViewModel.updateReward(updatedReward)

// Delete an existing reward from the database
val deletedReward = rewardViewModel.allRewards.value?.last()
rewardViewModel.deleteReward(deletedReward)
```

settings_screen

SettingsMenu

The `SettingsMenu` composable function displays a row with a title, description, and switch.

Parameters:

- `title`: The title of the row.
- `desc`: The description of the row.
- `checked`: The checked state of the switch.
- `onCheckedChange`: A callback that is invoked when the checked state of the switch changes.
- `enabled`: Whether the row is enabled.
- `titleColor`: The color of the title.
- `titleFontWeight`: The font weight of the title.
- `descColor`: The color of the description.
- `titleFontSize`: The font size of the title.

SettingsScreen

The `SettingsScreen` composable function displays a settings screen with the following features:

- A toggle to enable/disable dark mode.
- A text field to update the user's username.
- A dropdown menu to select the user's country.
- A button to update the user's profile image.
- A section to adjust the Pomodoro timer values.
- An image picker to select a new profile image.

Parameters:

- `settingsViewModel`: A `SettingsViewModel` instance that provides data and methods for managing the settings.
- `currentUser`: A `AuthUiClient` instance that provides information about the current user.
- `context`: A `Context` instance that provides access to the Android system.
- `scope`: A `CoroutineScope` instance that is used to launch coroutines.
- `drawerState`: A `DrawerState` instance that provides information about the state of the app drawer.

Notes:

- The `SettingsScreen` composable function uses the `AnimatedVisibility()` composable function to show/hide the image picker section based on the value of the `showImagePicker` state variable.
- The `SettingsScreen` composable function uses the `IconButton()` composable function to display a close button in the image picker section. The button is used to close the image picker section.
- The `SettingsScreen` composable function uses the `LazyColumn()` composable function to display a list of profile images in the image picker section. The list is lazy, which means that only the images that are currently visible on the screen are loaded into memory.
- The `SettingsScreen` composable function uses the `chunked()` function to split the list of profile images into a list of lists, where each sublist contains two images. This is done so that the images can be displayed in a two-column grid.
- The `SettingsScreen` composable function uses the `Image()` composable function to display a profile image in the image picker section. The image is clickable, and when clicked, it updates the user's profile image.

SettingsViewModel

The `SettingsViewModel` class is a ViewModel that provides data and methods for managing the settings in the app. It uses SharedPreferences to store the theme preference and LiveData to represent the state of the settings.

Constructors:

- `SettingsViewModel(application: Application, usersViewModel: UsersViewModel)`: This constructor takes an `Application` instance and a `UsersViewModel` instance as parameters. The `Application` instance is used to access SharedPreferences and the `UsersViewModel` instance is used to update the user's country.

Properties:

- `isDarkTheme`: A `MutableLiveData` instance that represents the dark theme state.
- `workTimerValue`: A `MutableLiveData` instance that represents the Pomodoro work timer value.
- `breakTimerValue`: A `MutableLiveData` instance that represents the Pomodoro break timer value.
- `profileImages`: A `MutableLiveData` instance that holds a list of profile image URLs.
- `currentUserProfileImage`: A `MutableLiveData` instance that holds the current user's profile image URL.

Methods:

- `updateUserCountry()`: This method updates the user's country.
- `toggleTheme()`: This method toggles the theme preference and saves it.
- `saveTimerValues()`: This method saves the Pomodoro work timer and break timer values.
- `getWorkTimerValue()`: This method returns the Pomodoro work timer value.
- `getBreakTimerValue()`: This method returns the Pomodoro break timer value.
- `fetchProfilePictures()`: This method fetches all profile pictures from Firebase Storage and updates the `profileImages` LiveData.
- `fetchCurrentUserProfileImage()`: This method fetches the current user's profile image from Firestore and updates the `currentUserProfileImage` LiveData.

- `updateProfileImage()`: This method updates the current user's profile image in Firestore.

The `SettingsViewModel` can be used in the following way:

```
val settingsViewModel = SettingsViewModel(application,
usersViewModel)

// Observe the isDarkTheme LiveData
settingsViewModel.isDarkTheme.observe(viewLifecycleOwner) { isDark ->
    // Update the UI based on the theme
}

// Update the user's country
settingsViewModel.updateUserCountry(userId, newCountry)

// Toggle the theme
settingsViewModel.toggleTheme(isDark)

// Save the Pomodoro timer values
settingsViewModel.saveTimerValues(workTime, breakTime)

// Get the Pomodoro work timer value
val workTimerValue = settingsViewModel.getWorkTimerValue()

// Get the Pomodoro break timer value
val breakTimerValue = settingsViewModel.getBreakTimerValue()

// Fetch all profile pictures
settingsViewModel.fetchProfilePictures()

// Fetch the current user's profile picture
settingsViewModel.fetchCurrentUserProfileImage(userId)

// Update the current user's profile image
settingsViewModel.updateProfileImage(userId, selectedImageUrl)
```

sign_in

SignInScreen

The `SignInScreen` composable function displays a sign-in screen with two buttons: "Sign in with Google" and "Sign in Anonymously".

Parameters:

- `state`: A `SignInState` object that contains the current state of the sign-in process.
- `onSignInClick`: A callback function that is invoked when the user clicks the "Sign in with Google" button.
- `onAnonymousSignIn`: A callback function that is invoked when the user clicks the "Sign in Anonymously" button.

Example:

```
val state = SignInState()

SignInScreen(
    state = state,
    onSignInClick = { /* Sign in with Google */ },
    onAnonymousSignIn = { /* Sign in Anonymously */ }
)
```

The `SignInScreen` composable function also uses the following `LaunchedEffect()`:

```
LaunchedEffect(key1 = state.signInError) {
    state.signInError?.let { error ->
        Toast.makeText(
            context,
            error,
            Toast.LENGTH_LONG
        ).show()
    }
}
```

This `LaunchedEffect()` is used to display a toast message to the user if there is an error with the sign-in process.

SignInViewModel

The `SignInViewModel` class is a ViewModel that provides data and methods for managing the sign-in process. It uses a `MutableStateFlow` to represent the current state of the sign-in process.

Constructors:

- `SignInViewModel()`: This constructor takes no parameters and creates a new `SignInViewModel` instance.

Properties:

- `state`: A `StateFlow` instance that represents the current state of the sign-in process.

Methods:

- `userIsAnonymous()`: This method sets the `userIsAnonymous` flag to `true`.
- `onSignInResult()`: This method updates the state of the sign-in process based on the result of the sign-in operation.
- `resetState()`: This method resets the state of the sign-in process.

The `SignInViewModel` can be used in the following way:

```
val viewModel = SignInViewModel()

// Observe the state of the sign-in process
viewModel.state.observe(viewLifecycleOwner) { state ->
    // Update the UI based on the state of the sign-in process
}

// Set the userIsAnonymous flag to true
```

```
viewModel.userIsAnonymous()

// Handle the sign-in result
viewModel.onSignInResult(result)

// Reset the state of the sign-in process
viewModel.resetState()
```

Here is an example of how to use the `SignInViewModel` in a `SignInScreen` composable function:

```
@Composable
fun SignInScreen(viewModel: SignInViewModel) {
    val state = viewModel.state.value

    // Display the sign-in buttons based on the state of the sign-in
process
    if (state.isSignInSuccessful) {
        // Show the signed-in UI
    } else if (state.userIsAnonymous) {
        // Show the anonymous sign-in UI
    } else {
        // Show the regular sign-in UI
    }
}
```

todo_screen

TodoScreen

The `TodoScreen` composable function displays a list of tasks, along with a floating action button to add new tasks. It uses a `BottomSheetScaffold` to display an add/edit task dialog when the user clicks the floating action button.

Parameters:

- `state`: A `TodoState` object that contains the current state of the to-do list.
- `onEvent`: A callback function that is invoked when the user interacts with the to-do list.

- `rewardViewModel`: A `RewardViewModel` instance.
- `usersViewModel`: A `UsersViewModel` instance.
- `alarmScheduler`: An `AlarmScheduler` instance.
- `navScope`: A `CoroutineScope` instance.
- `drawerState`: A `DrawerState` instance.

The `TodoScreen` composable function can be used in the following way:

```
val state = TodoState()

TodoScreen(
    state = state,
    onEvent = { event -> /* Handle the event */ },
    rewardViewModel = RewardViewModel(),
    usersViewModel = UsersViewModel(),
    alarmScheduler = AlarmSchedulerImpl(),
    navScope = CoroutineScope(),
    drawerState = DrawerState()
)
```

The `TodoScreen` composable function also uses the following `LaunchedEffect()`:

```
LaunchedEffect(key1 = showToast.value) {
    if (showToast.value) {
        delay(3000) // 3 seconds
        showToast.value = false
    }
}
```

This `LaunchedEffect()` is used to display a toast message to the user for 3 seconds when a task is completed.

TodoViewModel

The `TodoViewModel` class is a ViewModel that provides data and methods for managing the to-do list. It uses a `MutableStateFlow` to represent the current state of the to-do list.

Constructors:

- `TodoViewModel(todoDao: TodoDao)`: This constructor takes a `TodoDao`instance as a parameter and creates a new `TodoViewModel` instance.

Properties:

- `state`: A `StateFlow` instance that represents the current state of the to-do list.

Methods:

- `onEvent()`: This method handles events that are emitted by the to-do list.

Here is a brief overview of the events that are handled by the `TodoViewModel` class:

- `TodoEvent.titleError()`: This event is emitted when the user enters an invalid title for a to-do item.
- `TodoEvent.descriptionError()`: This event is emitted when the user enters an invalid description for a to-do item.
- `TodoEvent.updateTodo()`: This event is emitted when the user updates a to-do item.
- `TodoEvent.toggleIsClicked()`: This event is emitted when the user toggles the "is clicked" state of a to-do item.
- `TodoEvent.deleteTodo()`: This event is emitted when the user deletes a to-do item.
- `TodoEvent.showDialog()`: This event is emitted when the user wants to open the add/edit to-do dialog.
- `TodoEvent.hideDialog()`: This event is emitted when the user wants to close the add/edit to-do dialog.
- `TodoEvent.showEditDateSelector()`: This event is emitted when the user wants to open the date selector.
- `TodoEvent.hideEditDateSelector()`: This event is emitted when the user wants to close the date selector.
- `TodoEvent.showEditTimeSelector()`: This event is emitted when the user wants to open the time selector.

- `TodoEvent.hideEditTimeSelector()`: This event is emitted when the user wants to close the time selector.
- `TodoEvent.saveTodo()`: This event is emitted when the user wants to save a new or updated to-do item.
- `TodoEvent.setDescription()`: This event is emitted when the user changes the description of a to-do item.
- `TodoEvent.setDueDate()`: This event is emitted when the user changes the due date of a to-do item.
- `TodoEvent.setDueTime()`: This event is emitted when the user changes the due time of a to-do item.
- `TodoEvent.setPriority()`: This event is emitted when the user changes the priority of a to-do item.
- `TodoEvent.setTitle()`: This event is emitted when the user changes the title of a to-do item.
- `TodoEvent.sortBy()`: This event is emitted when the user wants to sort the to-do list.
- `TodoEvent.hideEditTodoDialog()`: This event is emitted when the user wants to close the edit to-do dialog.
- `TodoEvent.showEditTodoDialog()`: This event is emitted when the user wants to open the edit to-do dialog.
- `TodoEvent.hideDateSelector()`: This event is emitted when the user wants to close the date selector.
- `TodoEvent.showDateSelector()`: This event is emitted when the user wants to open the date selector.
- `TodoEvent.hideTimeSelector()`: This event is emitted when the user wants to close the time selector.
- `TodoEvent.showTimeSelector()`: This event is emitted when the user wants to open the time selector.
- `TodoEvent.toggleCompleted()`: This event is emitted when the user wants to toggle the completed state of a to-do item.
- `TodoEvent.setUserId()`: This event is emitted when the user wants to set the user ID of a to-do item.
- `TodoEvent.getTodoById()`: This event is emitted when the user wants to get a to-do item by its ID.
- `TodoEvent.resetTodos()`: This event is emitted when the user wants to reset the to-do list to the values from the database.
- `TodoEvent.resetState()`: This event is emitted when the user wants to reset the state of the to-do list.

ui_components

AppTopAppBar

The `SignInTopAppBar` composable function displays a top app bar for the sign in page. It has a title of "ADHD Task Manager" and the color scheme is set to the primary color palette.

The MainTopAppBar composable function displays a top app bar for the main page. It has a title of "ADHD Task Manager" and the color scheme is set to the primary color palette. It also has a navigation icon that opens and closes the app drawer.

The `SignInTopAppBar` composable function can be used in the following way:

```
@Composable
fun SignInScreen() {
    SignInTopAppBar()
}
```

CompletedTaskCard

The `CompletedTaskCard` composable function displays a card for a completed task. It takes a `Todo` object as a parameter and displays the following information:

- Title of the task
- Description of the task
- Priority of the task
- Due date of the task
- Due time of the task
- Date and time when the task was completed

The card is also styled to be visually appealing, with a light blue background and dark blue text. The title of the task is also displayed in a larger font size to make it stand out.

The `CompletedTaskCard` composable function can be used in the following way:

```
// Get the list of completed tasks
val completedTasks = viewModel.state.value.todos.filter { todo ->
todo.isCompleted }

// Display a card for each completed task
```

```
completedTasks.forEach { todo ->
    CompletedTaskCard(todo)
}
```

LeaderboardCard

The `LeaderboardCard` composable function displays a card for a user on the leaderboard. It takes a `Users` object and a `rank` integer as parameters, and displays the following information:

- Rank of the user on the leaderboard
- Profile image of the user
- Username of the user
- Country of the user
- Total points of the user (points + logins)

The card is also styled to be visually appealing, with a white background and blue text. The rank of the user is displayed in a larger and bolder font size to make it stand out.

Parameters:

- `user`: A `Users` object.
- `rank`: The rank of the user on the leaderboard.
- `defaultProfileImageUrl`: The default profile image URL to use if the user does not have a profile image.

The `LeaderboardCard` composable function can be used in the following way:

```
// Get the list of users on the leaderboard
val leaderboardUsers = viewModel.state.value.users

// Display a card for each user on the leaderboard
leaderboardUsers.forEachIndexed { rank, user ->
    LeaderboardCard(user, rank + 1, defaultProfileImageUrl)
}
```

LoginRewardCard

The `LoginRewardCard` composable function displays a card that shows the user's login reward progress. It takes a `Users` object as a parameter and displays the following information:

- The number of times the user has achieved their login reward
- The total number of points the user has earned for logging in

The card is also styled to be visually appealing, with a white background and blue text. The title of the card is displayed in a larger and bolder font size to make it stand out.

Parameters:

- `user`: A `Users` object.

The `LoginRewardCard` composable function can be used in the following way:

```
// Get the current user
val user = viewModel.state.value.user

// Display the login reward card
LoginRewardCard(user)
```

NoInternetScreen

The `NoInternetScreen` composable function displays a screen with a message and an icon indicating that there is no internet connection. It takes no parameters and returns nothing.

The `NoInternetScreen` composable function can be used in the following way:

```
// Check if the device has an internet connection
if (!isDeviceConnectedToTheInternet()) {
    // Display the no internet screen
    NoInternetScreen()
}
```

RewardCard

The `RewardCard` composable function displays a card that shows the user's progress towards a reward. It takes a `Reward` object, a `title` string, and a `Users` object as parameters and displays the following information:

- The title of the reward
- The number of times the user has achieved the reward
- The total number of points the user has earned

The card is also styled to be visually appealing, with a white background and blue text. The title of the card is displayed in a larger and bolder font size to make it stand out.

Parameters:

- `reward`: A `Reward` object.
- `title`: The title of the reward.
- `user`: A `Users` object.

The `RewardCard` composable function can be used in the following way:

```
// Get the list of rewards
val rewards = viewModel.state.value.rewards

// Get the current user
val user = viewModel.state.value.user

// Display a card for each reward
rewards.forEach { reward ->
    RewardCard(reward, title, user)
}
```

TodoCard

The `TodoCard` composable function displays a card for a todo item. It takes the following parameters:

- `todo`: A `Todo` object.
- `onEvent`: A callback function that is invoked when a user interacts with the card.
- `rewardViewModel`: A `RewardViewModel` object.

- `usersViewModel`: A `UsersViewModel` object.
- `showToast`: A mutable state variable that is used to display a toast message.
- `alarmScheduler`: An `AlarmScheduler` object.
- `user`: A `Users` object.

The card includes the following information:

- The title of the todo item
- A checkbox to mark the todo item as completed
- The description of the todo item
- The priority of the todo item
- The due date and time of the todo item
- A button to delete the todo item

The card also includes a Lottie animation that is displayed when the user marks the todo item as completed.

Utils

alarm_manager

AlarmItem

The `AlarmItem` data class has four properties:

- `id`: The unique identifier for the alarm item.
- `time`: The time at which the alarm should be triggered.
- `title`: The title of the alarm.
- `description`: A description of the alarm.

This data class is used to create alarm items to schedule notifications for tasks.

Here is an example of how to use the `AlarmItem` data class:

```
var alarmItem: AlarmItem? = null
alarmItem = AlarmItem(
id = state.id,
time = LocalDateTime.parse(
```

```
"${state.dueDate} ${state.dueTime}",
DateTimeFormatter.ofPattern("dd-MM-yyyy hh:mm a")
).atZone(ZoneId.systemDefault()).toLocalDateTime(),
title = state.title,
description = state.description
)
alarmItem?.let(alarmScheduler::schedule)
```

AlarmReciever

The `AlarmReceiver` class is a broadcast receiver used to receive and handle alarm events. It extends the `BroadcastReceiver` class and overrides the `onReceive()` method.

The `onReceive()` method is called when the receiver receives a broadcast intent. The intent contains information about the broadcast event, such as the action that was triggered and the data that was sent with the broadcast.

In the `onReceive()` method, the `AlarmReceiver` class extracts the title, description, and ID of the alarm item from the intent. It then calls the `showNotification()` method to display a notification for the alarm item.

The `showNotification()` method creates a new `NotificationCompat.Builder` object and configure it with the following information:

- `Content title`: The title of the alarm item.
- `Content text`: The description of the alarm item.
- `Auto cancel`: Whether the notification should be automatically cancelled when the user clicks on it.
- `Content intent`: A pending intent used to launch the main activity when the user clicks on the notification.
- `Small icon`: The icon that should be displayed for the notification.

The `showNotification()` method then uses the `NotificationManager` service to display the notification.

Here is an example of how the `AlarmReceiver` class could be used:

```
val intent = Intent(context, AlarmReceiver::class.java).apply {
putExtra("EXTRA_TITLE", alarmItem.title)
putExtra("EXTRA_DESCRIPTION", alarmItem.description)
putExtra("EXTRA_ID", alarmItem.id)
}
```

AlarmScheduler

The `AlarmScheduler` interface defines two methods:

- `schedule()`: Schedules an alarm item.
- `cancel()`: Cancels an alarm item.

The `schedule()` method takes an `AlarmItem` object as input and schedules the alarm for the specified time. The `cancel()` method takes an `AlarmItem` object as input and cancels the alarm.

The `AlarmScheduler` interface is implemented by the AlarmSchedulerImpl Class.

AlarmSchedulerImpl

The `AlarmSchedulerImpl` class is a concrete implementation of the `AlarmScheduler` interface. It schedules and cancels alarms using the system alarm service.

The `schedule()` method takes an `AlarmItem` object as input and schedules the alarm for the specified time. It uses the `setExactAndAllowWhileIdle()` method to schedule the alarm, which allows the alarm to be triggered even when the device is in idle mode.

The `cancel()` method takes an `AlarmItem` object as input and cancels the alarm. It uses the `cancel()` method to cancel the alarm, which removes the alarm from the system alarm service.

This implementation of the `AlarmScheduler` interface is more efficient than the previous implementation because it uses the `setExactAndAllowWhileIdle()` and `cancel()` methods directly. This avoids the need to create a new `PendingIntent` object each time an alarm is scheduled or cancelled.

Here is an example of how to use the `AlarmSchedulerImpl` class:

```
val alarmScheduler = AlarmSchedulerImpl(context)

// Schedule an alarm.
val alarmItem = AlarmItem(
    id = 1,
    time = LocalDateTime.now() + Duration.ofHours(1),
    title = "Wake up",
    description = "Get ready for work."
)
alarmScheduler.schedule(alarmItem)

// Cancel the alarm.
alarmScheduler.cancel(alarmItem)
```

connectivity

ConnectivityObserver

The `ConnectivityObserver` interface defines a single method:

- `observeConnectivity()`: Returns a `Flow<Status>` that emits the connectivity status of the device.

The `Status` enum has four possible values:

- `CONNECTED`: The device is connected to a network.
- `DISCONNECTED`: The device is not connected to a network.
- `LOSING`: The device is losing its connection to the network.
- `LOST`: The device has lost its connection to the network.

The `ConnectivityObserver` interface is implemented by the ConnectivityObserverImpl Class.

Connectivity ObserverImpl

The `ConnectivityObserverImpl` class is a more modern implementation of the `ConnectivityObserver` interface. It uses the `ConnectivityManager.NetworkCallback` API to monitor the connectivity status of the device.

The `ConnectivityManager.NetworkCallback` API is more efficient than the previous implementation because it allows you to register for callback notifications when the connectivity status changes. This avoids the need to poll the connectivity status periodically.

The `ConnectivityObserverImpl` class also uses the `callbackFlow()` operator to create a `Flow` that emits the connectivity status of the device. This allows you to use the `observeConnectivity()` method in a variety of ways, such as using the `collect()` method to collect the connectivity status or using the `distinctUntilChanged()` operator to filter out duplicate connectivity status values.

Here is an example of how to use the `ConnectivityObserverImpl` class:

```
val connectivityObserver = ConnectivityObserverImpl(context)

// Observe the connectivity status of the device.
connectivityObserver.observeConnectivity().collect { status ->
    when (status) {
        ConnectivityObserver.Status.CONNECTED -> {
            // The device is connected to a network.
        }
        ConnectivityObserver.Status.DISCONNECTED -> {
            // The device is not connected to a network.
        }
        ConnectivityObserver.Status.LOSING -> {
            // The device is losing its connection to the network.
        }
        ConnectivityObserver.Status.LOST -> {
            // The device has lost its connection to the network.
        }
    }
}
```

database_dao

RewardDao

The `RewardDao` interface defines a set of methods for interacting with a database table of rewards. The interface is annotated with the `@Dao` annotation, which indicates that it is a Data Access Object (DAO).

The DAO interface defines the following methods:

- `getAllRewards()`: Returns a `LiveData` object that emits a list of all rewards in the database.
- `updateReward()`: Updates the specified reward in the database.
- `insertReward()`: Inserts the specified reward into the database.
- `deleteReward()`: Deletes the specified reward from the database.
- `findReward()`: Returns a `LiveData` object that emits a list of rewards with the specified title.

TaskDao

The `TodoDao` interface defines a set of methods for interacting with a database table of todos. The interface is annotated with the `@Dao` annotation, which indicates that it is a Data Access Object (DAO).

The DAO interface defines the following methods:

- `insertTodo()`: Inserts the specified todo into the database.
- `deleteTodo()`: Deletes the specified todo from the database.
- `updateTodo()`: Updates the specified todo in the database.
- `getTodoById()`: Returns a `Flow<Todo>` that emits the todo with the specified ID.
- `getAllTodos()`: Returns a `Flow<List<Todo>>` that emits a list of all todos in the database.
- `sortByPriority()`: Returns a `Flow<List<Todo>>` that emits a list of todos sorted by priority in descending order.
- `sortByDueDateAndTime()`: Returns a `Flow<List<Todo>>` that emits a list of todos sorted by the due date in ascending order and due time in ascending order.
- `sortByCompleted()`: Returns a `Flow<List<Todo>>` that emits a list of completed todos sorted by completion date in ascending order.
- `sortByNotCompleted()`: Returns a `Flow<List<Todo>>` that emits a list of not completed todos.
- `updateTodoIsCompleted()`: Updates the `isCompleted` status of the specified todo to the opposite of its current value.

- `getCountOfCompletedTodos()`: Returns the number of completed todos in the database.
- `getAllCompletedTodos()`: Returns a `Flow<List<Todo>>` that emits a list of all completed todos in the database.

firebase

AuthUIClient

The `AuthUiClient` class is a client for managing user authentication and authorization using Firebase Auth and Google One Tap. It provides a number of public methods for signing in, signing out, and getting the current user's information.

Here is a summary of the public methods provided by the `AuthUiClient` class:

- `addAuthStateListener()`: Adds a listener that will be notified when the user's sign-in state changes.
- `getSignedIn()`: Returns `true` if the user is signed in, and `false` otherwise.
- `signIn()`: Starts the sign-in process with Google One Tap. Returns an `IntentSender` object that can be used to launch the sign-in activity.
- `signInWithIntent()`: Completes the sign-in process with Google One Tap. Returns a `SignInResult` object that contains the user's information or an error message.
- `signOut()`: Signs the user out.
- `getSignedInUser()`: Returns the current user's information, or `null` if no user is signed in.

The `AuthUiClient` class also provides a number of private methods for internal use. For example, the `buildSignInRequest()` method builds a `BeginSignInRequest` object that is used to start the sign-in process with Google One Tap.

FirebaseCallback

The `FirebaseCallback` interface defines a single method:

- `onResponse()`: Invoked when the Firebase operation is complete and a response is available.

The `onResponse()` method takes a `Response` object as its parameter. The `Response` object contains the data returned by the Firebase operation, or an error message if the operation failed.

The `FirebaseCallback` interface can be used to handle the results of Firebase operations in a flexible and asynchronous way. For example, the following code snippet shows how to use the `FirebaseCallback` interface to handle the results of a Firebase Realtime Database read operation:

```
val firebaseCallback: FirebaseCallback = object : FirebaseCallback {
    override fun onResponse(response: Response) {
        if (response.isSuccessful) {
            // The read operation was successful.
            // Get the data returned by the read operation.
            val data = response.data

            // Do something with the data.
        } else {
            // The read operation failed.
            // Show an error message to the user.
        }
    }
}

// Read the data from the Firebase Realtime Database.
val databaseReference =
FirebaseDatabase.getInstance().getReference("my-data")
databaseReference.get().addOnCompleteListener(firebaseCallback)
```

SignInResult

The `SignInResult` data class represents the result of a sign-in operation. It has two properties:

- `data`: The user's information, or `null` if the sign-in operation failed.
- `errorMessage`: An error message, or `null` if the sign-in operation was successful.

The `UserData` data class represents the user's information. It has three properties:

- `userId`: The user's unique identifier.
- `username`: The user's display name.
- `profilePictureUrl`: The URL of the user's profile picture.

These data classes can be used to represent the results of sign-in operations in a consistent and easy-to-use way.

Here is an example of how to use the `SignInResult` and `UserData` data classes:

```
val signInResult: SignInResult = authUiClient.signInWithIntent(data)

if (signInResult.data != null) {
    // The user signed in successfully.
    // Get the user's information.
    val user: UserData = signInResult.data

    // Do something with the user's information.
} else {
    // The user failed to sign in.
    // Show an error message to the user.
}
```

SignInState

The `SignInState` data class represents the state of the sign-in process. It has three properties:

- `isSignInSuccessful`: A Boolean value indicating whether the sign-in process was successful.
- `userIsAnonymous`: A Boolean value indicating whether the user signed in anonymously.
- `signInError`: An error message, or `null` if the sign-in process was successful.

This data class can be used to represent the state of the sign-in process in a consistent and easy-to-use way.

Here is an example of how to use the `SignInState` data class:

```
val signInState = SignInState(isSignInSuccessful = true,
userIsAnonymous = false)

when (signInState) {
    is SignInState.Success -> {
        // The user signed in successfully.
```

```
            // Get the user's information.
        }
        is SignInState.Anonymous -> {
            // The user signed in anonymously.
            // Update the UI to reflect this.
        }
        is SignInState.Failure -> {
            // The sign-in process failed.
            // Show an error message to the user.
        }
    }
```

firestore_utils

Users

The `Response` data class represents the response from the server. It has three properties:

- `users`: A list of users, or `null` if the request failed.
- `exception`: An exception, or `null` if the request was successful.
- `user`: A single user, or `null` if the request failed.

The `Users` data class represents a user. It has nine (9) properties:

- `displayName`: The user's display name.
- `points`: The user's points.
- `emailAddress`: The user's email address.
- `password`: The user's password.
- `username`: The user's username.
- `country`: The user's country.
- `userID`: The user's unique identifier.
- `profileImage`: The URL of the user's profile image.
- `loginNum`: The number of times the user has logged in.
- `totalPoints`: The user's total points.
- `lastLoginDate`: The date of the user's last login.

The `Final` class is a simple ArrayList that stores the final list of users ready for display. It has a companion object that provides a `finalDataList` property to store the final list of users. The `addToList()` function is used to add a user to the final list of users.

Here is an example of how to use the `Response`, `Users`, and `Final` classes:

```
val response: Response = getResponseFromServer()

// Check if the request was successful.
if (response.exception == null) {
    // Get the list of users.
    val users: List<Users>? = response.users

    // If the list of users is not empty, add it to the final list of
users.
    if (users != null) {
        for (user in users) {
            Final.addToList(user)
        }
    }
}

// Get the final list of users.
val finalListOfUsers: List<Users> = Final.finalDataList

// Display the final list of users.
```

UserRepo

The `UsersRepo` class is a repository for storing and retrieving user data from Firebase Firestore. It has four functions:

- `getResponse()`: This function retrieves a list of all users from Firestore and returns a `Response` object containing the list of users or an exception if the request fails.
- `updateCountry()`: This function updates the country of a user in Firestore.
- `getUserTwo()`: This function returns a `Flow` of the user with the given ID, or `null` if the user does not exist.
- `updatePoints()`: This function updates the points of a user in Firestore.

The `UsersRepo` class also has a new function:

```
suspend fun updateUser(uid: String, userMap: HashMap<String, *>) {
    userRef.document(uid).update(userMap as Map<String, Any>).await()
```

```
}
```

This function updates the user with the given ID in Firestore with the given user map.

Here is an example of how to use the `UsersRepo` class:

```
val usersRepo = UsersRepo()

// Get a list of all users.
usersRepo.getResponse { response ->
    if (response.exception == null) {
        val users: List<Users>? = response.users

        // Do something with the list of users.
    } else {
        // Handle the exception.
    }
}

// Update the country of a user.
usersRepo.updateCountry("user-id", "United States")

// Get a Flow of the user with the given ID.
val userFlow: Flow<Users?> = usersRepo.getUserTwo("user-id")

// Observe the Flow and update the UI accordingly.
userFlow.collect { user ->
    // Update the UI with the user's information.
}

// Update the points of a user.
usersRepo.updatePoints(user, 100)

// Update the user with the given ID.
usersRepo.updateUser("user-id", hashMapOf("username" to
"new-username"))
```

UsersViewModel

The `UsersViewModel` class is a ViewModel for managing user data in a FirebaseFirestore database. It provides the following features:

- It exposes a `user` state flow that contains the current user.
- It provides a `checkUserInFirestore()` function that checks if the user exists in Firestore and adds it if it does not.
- It provides an `updateLastLoginDate()` function that updates the user's last login date in Firestore.
- It provides an `updateLoginNum()` function that increments the user's login number in Firestore.
- It provides an `addUserToFirestore()` function that adds a new user to Firestore.
- It provides an `initializeAuthUiClient()` function that initializes the AuthUiClient.
- It provides a `fetchAndUpdateUserPoints()` function that fetches the user's updated points from Firestore.
- It provides a `getUser()` function that fetches the user from Firestore.
- It provides an `updateUserCountry()` function that updates the user's country in Firestore.
- It provides an `updateTotalPoints()` function that updates the user's total points in Firestore.
- It provides a `completedTaskPoints()` function that updates the user's points after completing a task.
- It provides a `fetchImageList()` function that fetches a list of image URIs from a given directory in Firebase Storage.

Here is an example of how to use the `UsersViewModel` class:

```
val usersViewModel = UsersViewModel()

// Check if the user exists in Firestore.
usersViewModel.checkUserInFirestore(userId, authUiClient)

// Observe the user state flow.
usersViewModel.user.collect { user ->
    // Update the UI with the user's information.
}
```

```
// Update the user's country.
usersViewModel.updateUserCountry(userId, "United States")

// Fetch the user's updated points from Firestore.
usersViewModel.fetchAndUpdateUserPoints(userId)

// Get the user's total points.
val totalPoints = usersViewModel.user.value?.totalPoints

// Update the user's points after completing a task.
usersViewModel.completedTaskPoints()

// Fetch a list of image URIs from a given directory in Firebase
Storage.
val imageListFlow = usersViewModel.fetchImageList("images")

// Observe the image list flow.
imageListFlow.collect { imageList ->
    // Update the UI with the image list.
}
```

local_database

Reward

The `Reward` data class represents a reward in the Firebase Firestore database. It has the following properties:

- `title`: The title of the reward.
- `description`: The description of the reward.
- `pointsAwarded`: The number of points awarded for achieving the reward.
- `timesAchieved`: The number of times the reward has been achieved.
- `id`: The unique identifier of the reward.

The `@Entity` annotation indicates that the `Reward` data class is an entity in the Firebase Firestore database. The `tableName` attribute specifies the name of the table that the entity will be mapped to.

The `@ColumnInfo` annotation indicates that the `title`, `description`, `pointsAwarded`, and `timesAchieved` properties are columns in the `rewards` table.

The `@PrimaryKey` annotation indicates that the `id` property is the primary key of the `rewards` table.

Here is an example of how to use the `Reward` data class:

```
// Create a new reward.
val reward = Reward(
    title = "Complete 10 tasks",
    description = "Earn 100 points for completing 10 tasks.",
    pointsAwarded = 100,
    timesAchieved = 0
)

// Add the reward to the FirebaseFirestore database.
val db = FirebaseFirestore.getInstance()
val rewardRef = db.collection("rewards").document()
rewardRef.set(reward)

// Get a list of all rewards from the FirebaseFirestore database.
val rewards =
db.collection("rewards").get().await().toObjects(Reward::class.java)

// Update a reward in the FirebaseFirestore database.
val rewardToUpdate = rewards[0]
rewardToUpdate.timesAchieved++
rewardRef.update(rewardToUpdate)

// Delete a reward from the FirebaseFirestore database.
rewardRef.delete()
```

RewardDatabase

The `RewardDatabase` class is a Room database for storing and retrieving rewards. It has the following properties:

- `entities`: A list of entity classes that will be mapped to tables in the database.
- `version`: The version of the database schema.
- `exportSchema`: A Boolean value indicating whether to export the database schema to a file.

The `rewardDao` property is a DAO that provides access to the rewards table in the database.

The `getInstance()` function returns a singleton instance of the `RewardDatabase` class. It uses a synchronized block to ensure that only one instance of the database is created.

The `createFromAsset()` function creates a new database from the `reward.db` file in the assets folder.

Here is an example of how to use the `RewardDatabase` class:

```
// Get the database instance.
val database = RewardDatabase.getInstance(context)

// Get the DAO.
val rewardDao = database.rewardDao

// Insert a new reward.
rewardDao.insert(reward)

// Get all rewards from the database.
val rewards = rewardDao.getAll()

// Update a reward.
rewardDao.update(reward)

// Delete a reward.
rewardDao.delete(reward)
```

TodoDatabase

The `TodoDatabase` class is a Room database for storing and retrieving to-dos. It has the following properties:

- `entities`: A list of entity classes that will be mapped to tables in the database. In this case, the only entity class is `Todo`.
- `version`: The version of the database schema.
- `exportSchema`: A Boolean value indicating whether to export the database schema to a file.

The `todoDao` property is a DAO that provides access to the to-do table in the database.

Here is an example of how to use the `TodoDatabase` class:

```
// Get the database instance.
val database = TodoDatabase.getInstance(context)

// Get the DAO.
val todoDao = database.todoDao

// Insert a new to-do.
todoDao.insert(todo)

// Get all to-dos from the database.
val todos = todoDao.getAll()

// Update a to-do.
todoDao.update(todo)

// Delete a to-do.
todoDao.delete(todo)
```

nav_utils

Screen

The `Screen` sealed class represents the different screens that can be displayed in the navigation drawer menu. It has the following properties:

- `route`: The route of the screen.
- `icon`: The icon of the screen.
- `title`: The title of the screen.

To use the `Screen` sealed class, you can create a new screen object by calling the corresponding constructor. For example, to create a new `TodoScreen` object, you would call the following code:

```
val todoScreen = Screen.TodoScreen
```

You can then access the route, icon, and title of the screen using the corresponding

properties. For example, to get the route of the `TodoScreen`, you would use the following code:

```
val todoScreenRoute = todoScreen.route
```

You can also use the `Screen` Sealed class to check which screen is currently being displayed. For example, the following code will check if the current screen is the `TodoScreen`:

```
if (currentScreen is Screen.TodoScreen) {
    // The current screen is the TodoScreen.
}
```

notifications

NotificationsApp

The `NotificationsApp` class is a custom application class that creates a notification channel for the application. This is necessary in order to send notifications to the user on Android 8.0 (API level 26) and higher.

The `NotificationsApp` class overrides the `onCreate()` method of the `Application` class. In this overridden method, the class creates a new notification channel with the following properties:

- ID: "TodoNotification"
- Name: "Todo Reminder Notification"
- Importance: NotificationManager.IMPORTANCE_HIGH

The class then sets a description for the notification channel and creates the notification channel using the `notificationManager.createNotificationChannel()` method.

Here is an example of how to use the `NotificationsApp` class:

```
class MyApplication : NotificationsApp() {
    override fun onCreate() {
        super.onCreate()
```

```
        // Send a notification to the user.
        val notification = NotificationCompat.Builder(this,
"TodoNotification")
            .setContentTitle("Todo Reminder")
            .setContentText("You have a task due in 1 hour.")
            .setSmallIcon(R.drawable.ic_notification)
            .build()

        val notificationManager =
getSystemService(NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.notify(1, notification)
    }
}
```

The `NotificationsApp` class provides a convenient way to create and manage notification channels for your application.

permissions

AskForDoNotDisturbPermission

The `getDoNotDisturbPermission()`, `toggleDoNotDisturb()`, and `isDoNotDisturbEnabled()` functions are utility functions for managing Do Not Disturb functionality on Android.

The `getDoNotDisturbPermission()` function opens the system settings screen for granting the application permission to manage Do Not Disturb. This function is necessary because the application needs this permission to toggle Do Not Disturb on and off.

The `toggleDoNotDisturb()` function toggles Do Not Disturb on or off, depending on its current state. If the application does not have permission to manage Do Not Disturb, the function will open the system settings screen for granting the permission.

The `isDoNotDisturbEnabled()` function checks if Do Not Disturb is currently enabled.

Here is an example of how to use these functions:

```
// Check if Do Not Disturb is enabled.
val isDoNotDisturbEnabled = isDoNotDisturbEnabled(context)

// Toggle Do Not Disturb on or off.
toggleDoNotDisturb(context, activity)

// Get the permission to manage Do Not Disturb.
getDoNotDisturbPermission(activity)
```

states

TodoState

The `TodoState` data class represents the state of the to-do list in the application. It has the following properties:

- `todos`: A list of to-do items.
- `title`: The title of the to-do item.
- `description`: The description of the to-do item.
- `priority`: The priority of the to-do item.
- `dueDate`: The due date of the to-do item.
- `dueTime`: The due time of the to-do item.
- `userId`: The user ID of the user who created the to-do item.
- `isClicked`: A Boolean value indicating whether the to-do item has been clicked.
- `id`: The unique identifier of the to-do item.
- `sortType`: The sort type of the to-do list.
- `titleError`: A Boolean value indicating whether the to-do item has a title error.
- `descriptionError`: A Boolean value indicating whether the to-do item has a description error.
- `showDialog`: A Boolean value indicating whether the to-do list dialog is open.
- `showEditTodoDialog`: A Boolean value indicating whether the edit to-do dialog is open.
- `showDateSelector`: A Boolean value indicating whether the date selector is open.
- `showTimeSelector`: A Boolean value indicating whether the time selector is open.
- `showEditDateSelector`: A Boolean value indicating whether the edit date selector is open.
- `showEditTimeSelector`: A Boolean value indicating whether the edit time selector is open.
- `completedTodos`: A list of completed to-do items.

- `showLottieAnimation`: A Boolean value indicating whether the Lottie animation is visible.

This data class can be used to store the state of the to-do list and to update the UI accordingly. It can also be used to validate the to-do item data and to prevent the user from adding or updating to-do items with invalid data.

Here is an example of how to use the `TodoState` data class:

```
// Create a new TodoState object.
val todoState = TodoState()

// Get the list of to-do items.
val todos = todoState.todos

// Add a new to-do item to the list.
val newTodo = Todo(
    title = "Complete 10 tasks",
    description = "Earn 100 points for completing 10 tasks.",
    priority = Priority.HIGH,
    dueDate = "2023-10-29",
    dueTime = "12:00 PM"
)
todoState.todos = todos + newTodo

// Validate the to-do item data.
val isTodoItemValid = todoState.validateTodoItem(newTodo)

// Update the UI based on the TodoState object.
```

todo_utils

Priority

The `Priority` enum class represents the priority of a to-do item. It has the following values:

- `LOW`: The to-do item is low priority.
- `MEDIUM`: The to-do item is a medium priority.
- `HIGH`: The to-do item is a high priority.

This enum class can be used to store the priority of a to-do item and to sort the to-do list based on priority.

SortType

The `SortType` enum class represents the sort type of the to-do list. It has the following values:

- `BY_PRIORITY`: The to-do list is sorted by priority, with high-priority to-do items appearing first.
- `BY_DATE_TIME`: The to-do list is sorted by due date and time, with to-do items due sooner appearing first.
- `BY_COMPLETED`: The to-do list is sorted by completion status, with completed to-do items appearing last.
- `BY_NOT_COMPLETED`: The to-do list is sorted by completion status, with not completed to-do items appearing first.

This enum class can be used to store the sort type of the to-do list and to sort the to-do list accordingly.

Here is an example of how to use the `SortType` enum class:

```
// Set the sort type of the to-do list to BY_PRIORITY.
todoState.sortType = SortType.BY_PRIORITY

// Sort the to-do list based on the sort type.
todoState.todos = todoState.todos.sortedBy { it.priority }
```

Todo

The `Todo` data class represents a to-do item in the application. It has the following properties:

- `title`: The title of the to-do item.
- `description`: The description of the to-do item.
- `priority`: The priority of the to-do item.
- `dueDate`: The due date of the to-do item.
- `dueTime`: The due time of the to-do item.
- `isCompleted`: A Boolean value indicating whether the to-do item has been completed.
- `completionDate`: The date on which the to-do item was completed.
- `isClicked`: A Boolean value indicating whether the to-do item has been clicked.
- `id`: The unique identifier of the to-do item.
- `userID`: The user ID of the user who created the to-do item.

The `@Entity` annotation indicates that the `Todo` data class is an entity in the Room database. The `@PrimaryKey` annotation indicates that the `id` property is the primary key of the `Todo` table.

Here is an example of how to use the `Todo` data class:

```
// Create a new to-do item.
val todo = Todo(
    title = "Complete 10 tasks",
    description = "Earn 100 points for completing 10 tasks.",
    priority = Priority.HIGH,
    dueDate = "2023-10-29",
    dueTime = "12:00 PM",
    userID = "1234567890"
)

// Add the to-do item to the Room database.
val todoDao = database.todoDao
todoDao.insert(todo)

// Get the to-do item from the Room database.
val todoById = todoDao.getTodoById(todo.id)

// Update the to-do item.
```

```
todo.isCompleted = true

// Update the to-do item in the Room database.
todoDao.update(todo)

// Delete the to-do item from the Room database.
todoDao.delete(todo)
```

TodoEvent

The `TodoEvent` sealed interface represents the different events that can occur in the application. It has the following sealed classes:

- `titleError(val error: Boolean)`: A sealed class that represents a title error.
- `descriptionError(val error: Boolean)`: A sealed class that represents a description error.
- `updateTodo(val todo: Todo)`: A sealed class that represents a todo update event.
- `getTodoById(val id: Int)`: A sealed class that represents a get todo by id event.
- `toggleIsClicked(val todo: Todo)`: A sealed class that represents a toggle is clicked event.
- `saveTodo`: A sealed class that represents a save todo event.
- `setTitle(val title: String)`: A sealed class that represents a set title event.
- `setDescription(val description: String)`: A sealed class that represents a set description event.
- `setPriority(val priority: Priority)`: A sealed class that represents a set priority event.
- `setDueDate(val dueDate: String)`: A sealed class that represents a set due date event.
- `setDueTime(val dueTime: String)`: A sealed class that represents a set due time event.
- `deleteTodo(val todo: Todo)`: A sealed class that represents a delete todo event.
- `setCompletedDate(val todo: Todo)`: A sealed class that represents a set completed date event.
- `setUserId(val userId: String)`: A sealed class that represents a set user ID event.
- `toggleCompleted(val todo: Todo)`: A sealed class that represents a toggle completed event.

- `sortBy(val sortType: SortType)`: A sealed class that represents a sort by event.
- `showDialog`: A sealed class that represents a show dialog event.
- `hideDialog`: A sealed class that represents a hide dialog event.
- `showEditTodoDialog`: A sealed class that represents a show edit todo dialog event.
- `hideEditTodoDialog`: A sealed class that represents a hide edit todo dialog event.
- `showDateSelector`: A sealed class that represents a show date selector event.
- `hideDateSelector`: A sealed class that represents a hide date selector event.
- `showTimeSelector`: A sealed class that represents a show-time selector event.
- `hideTimeSelector`: A sealed class that represents a hide time selector event.
- `showEditDateSelector`: A sealed class that represents a show edit date selector event.
- `hideEditDateSelector`: A sealed class that represents a hide edit date selector event.
- `showEditTimeSelector`: A sealed class that represents a show edit time selector event.
- `hideEditTimeSelector`: A sealed class that represents a hide edit time selector event.
- `ShowCompletedTasks`: A sealed class that represents a show-completed tasks event.
- `resetTodos`: A sealed class that represents a reset todos event.
- `resetState`: A sealed class that represents a reset state event.

These sealed classes can be used to represent the different events that can occur in the application and to handle them accordingly.

BitmapUtils

The `blurBitmap()`, `takeScreenshot()`, and `captureScreenshotWhenReady()` functions are utility functions for blurring and capturing screenshots in Android.

The `blurBitmap()` function blurs a given bitmap using the RenderScript API. It takes the bitmap, the application context, and the blur radius as input and returns a blurred bitmap if successful, or null if an error occurred.

The `takeScreenshot()` function takes a screenshot of the given view and returns a bitmap.

The `captureScreenshotWhenReady()` function captures a screenshot of the given view when it is ready to be drawn. It takes the view and a callback function as input. The callback function is called with the captured bitmap when it is available.

Here is an example of how to use the `blurBitmap()` and `captureScreenshotWhenReady()` functions:

```
// Capture a screenshot of the view when it is ready to be drawn.
captureScreenshotWhenReady(view) { bitmap ->
    // Blur the screenshot.
    val blurredBitmap = blurBitmap(bitmap, applicationContext)

    // Display the blurred screenshot.
    imageView.setImageBitmap(blurredBitmap)
}
```

These functions can be used to blur and capture screenshots in a variety of different ways. For example, you could use the `blurBitmap()` function to blur a screenshot of the entire screen before displaying it to the user, or you could use the `captureScreenshotWhenReady()` function to capture a screenshot of a specific view and then blur it.