Test cover page

Test description

Table of contents

Voci app Documentation

Introduction

Kotlin

Overview

This project was created as an assignment for the University of Milano Bicocca course "Dispositivi Mobili".

Idea

The idea is to create an app that would help catalog and manage requests and updates about homeless people in Milan. It was born because of a member of the team who is also a member of the VoCi ONLUS, an organization that aims to aid the homeless people of their city. Website: VoCi ONLUS (https://www.volontaricittadini.it)

Team

We are students of the University of Milano Bicocca (https://www.unimib.it/), and we are interested in making an app that not only will get us a good mark but that can be used and can be helpful to a good cause.

- Samuele "DreoX" - 904280
- Matthias "Inutiliax" - 894374
- Olcio "Tu padre" - 000000
- Enrico "Il magnifico" - 000000
- Gabriele "Fish" - 000000

Technologies

IDEs

We used Android Studio to write and test the code and Writerside to write the documentation. Both come from a well-established company in the coding world called "JetBrains". Android Studio is built in conjunction with Google.

Languages and Libraries

Our app is primarily written in Kotlin, a modern, concise, and type-safe programming language designed for Android development. It leverages various libraries to enhance functionality:

- **Jetpack**: A suite of libraries from Google that simplify common Android development tasks. This includes libraries for navigation, lifecycle management, UI components, and more.
- **Material Design 3**: The latest iteration of Google's Material Design system for Android. It provides a consistent and beautiful design language with pre-built components like buttons, text fields, and cards.
- **Gradle**: A build automation tool that manages dependencies, builds your app, and packages it for distribution.
- **Firebase**: Google's mobile app development platform. We utilize Firebase to:
  - **Store user data and app data**: Firebase Firestore, a NoSQL cloud database, provides flexible and scalable data storage for your app.
  - **Firebase Authentication (Auth)**: Simplifies user authentication and management, allowing users to sign in with various methods like email/password or social logins.
- **Git & GitHub**: Basic tools for version control and code sharing. We could have used GitLabs but we were more familiar with Github having used it for previous courses (Also we can't give all the data to Google, we have to share it with Microsoft as well).

Project

Folder structure

```
vociapp/
├── MainActivity.kt
├── ui/
│   ├── navigation/
```

```
│   │   ├── Screens.kt
│   │   └── NavGraph.kt
│   ├── screens/
│   │   ├── auth/
│   │   │   ├── SignInScreen.kt
│   │   │   └── SignUpScreen.kt
│   │   └── profile/
│   │       ├── UserProfileScreen.kt
│   │       └── UpdateProfileScreen.kt
│   ├── theme/
│   │   └── Theme.kt
│   └── viewmodels/
│       ├── AuthViewModel.kt
│       └── AuthState.kt
└── di/
    └── AppModule.kt
```

Screens and Navigation

Authentication Screens

Sign-In Screen

The `SignInScreen` allows users to sign in using their email and password. It integrates with `AuthViewModel` to handle the sign-in process. The screen layout includes input fields for email and password, and a button to submit the sign-in request.

Sign-Up Screen

The `SignUpScreen` allows users to create a new account. It includes input fields for email, password, and confirmation password. The `AuthViewModel` manages user registration and handles any errors during the process.

Profile Screens

User Profile Screen

The `UserProfileScreen` displays the profile information of the logged-in user. It retrieves data from `AuthViewModel` and shows details like display name and profile picture.

Update Profile Screen

The `UpdateProfileScreen` allows users to update their profile information, such as their display name and profile picture. Changes are processed and stored using the `AuthViewModel`.

Dependencies and Modules

Dependency Injection (DI)

We use Dagger (https://dagger.dev/) for dependency injection to provide the app's components with necessary dependencies. Configuration is done in `AppModule.kt` under the `di` directory.

Gradle Configuration

Our `build.gradle` files are configured to include necessary dependencies for Android development, Jetpack libraries, Firebase services, and other essential libraries.

Getting Started

Prerequisites

- Android Studio installed
- Kotlin plugin enabled
- Firebase project setup with Firestore and Authentication enabled

Installation

1. Clone the repository from GitHub.
2. Open the project in Android Studio.
3. Sync the Gradle files to download dependencies.
4. Configure Firebase by adding the `google-services.json` file to the `app` directory.
5. Run the project on an Android emulator or physical device.

Acknowledgements

We would like to thank:

- Our professor for the guidance and support throughout the course.
- VoCi ONLUS for inspiring the project concept and their ongoing work to aid the homeless community.

Screens (Screens.kt)

Overview

This file defines the screen routes and their associated data for navigation within the application. It uses a sealed class to represent different screens and provides a central place to manage navigation destinations.

Code Explanation

Breakdown

- **`sealed class Screens(val route: String, val title: String, val icon: ImageVector)`**
  - A sealed class is used to represent the different screens in the application. This ensures that all possible screens are defined within this class and prevents the creation of arbitrary screen types.
- **`object Home : Screens("home", title = "Home", icon = Icons.Filled.Home)`**
  - Each screen is defined as an object within the Screens sealed class.
  - Each screen object has the following properties:
    - `route`: A string representing the unique route for the screen. This is used by the navigation system to identify and navigate to the screen.
    - `title`: A string representing the title of the screen, used in the BottomBar (Bottom Navigation Bar (BottomBar.kt)).
    - `icon`: An `ImageVector` representing the icon associated with the screen, used in the BottomBar (Bottom Navigation Bar (BottomBar.kt)).
- **Screens list:**
  - `Home`
  - `UserProfile`
  - `UpdateUserProfile`

- `SignIn`
- `SignUp`

Usage

NavGraph

The screen routes defined in `Screens.kt` are used in the NavGraph (Navigation Graph (NavGraph.kt)) to define navigation destinations:

```
// In NavGraph.kt

composable(route = Screens.Home.route) {
    HomeScreen()
}

composable(route = Screens.UserProfile.route) {
    UserProfileScreen()
}
```

BottomBar

The screen `title` and `icon` are used in the BottomBar (Bottom Navigation Bar (BottomBar.kt)) component to render the items of the bar:

```
// In BottomBar.kt

fun BottomBar(navController: NavHostController) {
    val items = listOf(
        Screens.Home,
        Screens.UserProfile
    )

    //other parts of the component

    NavigationBar {
        items.forEach { screen ->
            NavigationBarItem(
```

```
                icon = { Icon(screen.icon, contentDescription =
screen.title) },
                label = { Text(screen.title) },
                selected = currentRoute == screen.route,
                onClick = { navController.navigate(screen.route) }
            )
        }
    }
}
```

Related Files

- **NavGraph (Navigation Graph (NavGraph.kt)):** Uses the screen `routes` defined to set up navigation destinations.
- **BottomBar (Bottom Navigation Bar (BottomBar.kt)):** Uses the Screens class to render the items in the bottom bar with the `icon` and `title`.

Additional Notes

- Using a sealed class for screen routes provides type safety and ensures all possible screens are defined in a central location.
- The `route` property is crucial for identifying and navigating to screens within the navigation graph.
- The `title` and `icon` properties are used in the UI components.

Navigation Graph (NavGraph.kt)

Overview

This file defines the navigation graph for the application, which controls the navigation flow between different screens. It uses Jetpack Compose's Navigation component to manage the navigation stack and transitions.

Code Explanation

Breakdown

- **`@Composable fun NavGraph(navController: NavHostController, paddingValues: PaddingValues)`**
  - This composable function sets up the navigation graph using `NavHost`.
  - Parameters:
    - `navController`: Receives the original NavHostController created in the Main Activity (Main Activity (MainActivity.kt)).
    - `paddingValues`: Receives the padding values that are default from the Scaffold (https://https://developer.android.com/develop/ui/compose/components/scaffold).
- **`val authViewModel = remember { AuthViewModel() }`**
  - Initializes an instance of `AuthViewModel` using the `remember` composable for state retention across recompositions.
- **`val authState by authViewModel.authState.collectAsState()`**
  - Collects the authentication state from the `AuthViewModel` as a state value using the `collectAsState` extension function.
- **`LaunchedEffect(authState)`**
  - A side effect that runs whenever `authState` changes. Useful for performing actions such as navigation based on the authentication state.

- **`when (authState)`**
  - Defines a list of actions that depend on the state of the user authentication:
    - `AuthState.Authenticated`: The user gets routed to the Home screen.
    - `AuthState.Unauthenticated`: The user gets routed to the SignIn screen.
- **`NavHost( navController = navController, startDestination = Screens.Home.route, modifier = Modifier.padding(paddingValues) )`**
  - This function sets up the navigation graph with all the possible routes and the according Screens (Screens (Screens.kt)).
  - Parameters:
    - `navController`: Receives the original NavHostController created in the Main Activity (Main Activity (MainActivity.kt)).
    - `modifier`: Applies the padding values.
    - `startDestination` parameter specifies the initial screen to be displayed when the app launches.
- **`composable(route = Screens.Example.route) { ExampleScreen(navController) }`:**
  - Inside the `NavHost` scope, `composable` functions define routes for each screen.
  - List:
    - `HomeScreen`
    - `SignInScreen`
    - `SignUpScreen`
    - `UserProfileScreen`
    - `UpdateUserProfileScreen`
- **`fun currentRoute(navController: NavHostController): String?` Helper function:**
  - This helper function retrieves the current route from anywhere inside the navigation graph.

- It uses the passed in `navController` and `currentBackStackEntryAsState()` to get the current route that can be used in components like the BottomBar (Bottom Navigation Bar (BottomBar.kt)).

Related Files

- **Screens (Screens.kt):** Defines the screen routes and related properties such as `route`, `title`, and `icon`.
- **Main Activity (MainActivity.kt):** The MainActivity is the entry point of the app and is responsible for the creation of the navController.

Usage

NavGraph

The `NavGraph` setup is called from the main activity's `setContent` block to establish the structure of navigation in your app:

```
import androidx.navigation.compose.rememberNavController
import com.example.vociapp.ui.navigation.NavGraph

val navController = rememberNavController()
NavGraph(navController = navController, paddingValues = innerPadding)
```

Rest of the app

Throughout the rest of the app we are going to use the navController parameter that is passed to every screen like this:

```
// example
navController.navigate("signIn")
```

Additional Notes

- The navigation graph's organization allows for a clear and scalable structure for handling multiple screens.

- Proper management of the authentication state enhances user experience by directing them to the appropriate screens based on their login status.
- Remember to handle deeper navigation features or customized transitions by referring to the official Jetpack Compose Navigation documentation.

Sign-In Screen (SignInScreen.kt)

Overview

This file defines the sign-in screen component for the application. The screen allows users to sign in using their email and password. It utilizes Jetpack Compose to create a responsive and interactive UI. The authentication process is managed by `AuthViewModel`.

Code Explanation

Breakdown

- **`@Composable fun SignInScreen(navController: NavHostController, authViewModel: AuthViewModel)`**
  - Defines the sign-in screen as a composable function.
  - Parameters:
    - `navController`: The navigation controller to manage app navigation.
    - `authViewModel`: The view model that handles authentication.
- **State Variables**
  - **`var email by remember { mutableStateOf("") }`**: Holds the email input value.
  - **`var password by remember { mutableStateOf("") }`**: Holds the password input value.
  - **`var showError by remember { mutableStateOf(false) }`**: Flag to show or hide error messages.
  - **`var errorMessage by remember { mutableStateOf("") }`**: Stores the error message text.
  - **`var isSigningIn by remember { mutableStateOf(false) }`**: Flag to indicate if sign-in process is ongoing.

- **`Box(modifier = Modifier.fillMaxSize().background(MaterialTheme.colorScheme.background))`**
  - Root container that fills the available space and sets the background color.
- **`Column(modifier = Modifier.fillMaxSize().padding(24.dp), ...)`**
  - Vertical layout container that holds the sign-in elements.
  - Centers its children both horizontally and vertically.
- **`Text("Sign In", style = MaterialTheme.typography.headlineLarge, ...)`**
  - Title text for the sign-in screen.
  - Uses the primary color and bold font weight.
- **`Spacer(modifier = Modifier.height(32.dp))`**
  - Adds vertical space between UI elements.
- **`Card(modifier = Modifier.fillMaxWidth().padding(horizontal = 16.dp), ...)`**
  - Displays the input fields for email and password.
  - Parameters:
    - `modifier`: Sets the width and horizontal padding of the card.
    - `elevation`: Sets the card elevation for shadow effect, using `CardDefaults.cardElevation`.
    - `shape`: Sets the shape of the card with rounded corners.
- **`Column(modifier = Modifier.fillMaxWidth().padding(24.dp), verticalArrangement = Arrangement.spacedBy(16.dp))`**
  - Inner vertical layout within the card for input fields and buttons.
  - Arranges its children with spaced margins.
- **`AuthTextField(value = email, onValueChange = { email = it }, label = "Email", icon = Icons.Default.Email)`**

    - Input field for email, utilizing the custom `AuthTextField` composable.
    - Displays an email icon.
- **`AuthTextField(value = password, onValueChange = { password = it }, label = "Password", icon = Icons.Default.Lock, isPassword = true)`**
    - Input field for password, utilizing the custom `AuthTextField` composable.
    - Sets `isPassword` to true for input masking.
- **`Button(onClick = { isSigningIn = true }, enabled = !isSigningIn)`**
    - Sign-in button.
    - Starts the sign-in process when clicked.
    - Disabled while `isSigningIn` is true to prevent multiple submissions.

Related Files

- **AuthViewModel.kt (Authentication ViewModel (AuthViewModel.kt)):** Manages user authentication and handles sign-in operations.
- **NavGraph.kt (Navigation Graph (NavGraph.kt)):** Defines the navigation routes, including the sign-in screen.
- **UserProfileScreen.kt (User Profile Screen (UserProfileScreen.kt)):** Navigates to the user profile screen upon successful sign-in.

Usage

The `SignInScreen` is used for user authentication within the app. Here is how it can be integrated and used:

Example Usage in NavGraph

Include the `SignInScreen` in the navigation graph to enable navigation to the sign-in screen:

```
composable(route = Screens.SignIn.route) {
    SignInScreen(navController = navController, authViewModel =
```

```
authViewModel)
}
```

Handling Sign-In

The sign-in button initiates the authentication process, managed by `AuthViewModel`:

```
Button(
    onClick = {
        isSigningIn = true
        // Authenticate user
        viewModelScope.launch {
            val result = authViewModel.signInWithEmailAndPassword(email,
password)
            isSigningIn = false
            if (result is AuthResult.Success) {
                // Navigate to user profile or home screen
                navController.navigate(Screens.UserProfile.route)
            } else if (result is AuthResult.Failure) {
                // Display error message
                showError = true
                errorMessage = result.message
            }
        }
    },
    enabled = !isSigningIn
) {
    Text("Sign In")
}
```

Additional Notes

- The screen provides a clean and user-friendly interface for user authentication.
- It ensures responsive design and optimal user experience across different devices.
- Proper state management and error handling enhance the reliability of the

authentication process.

Sign-Up Screen (SignUpScreen.kt)

Overview

This file defines the sign-up screen component for the application. The screen allows users to create a new account using their email and password. It utilizes Jetpack Compose to create a responsive and interactive UI. The authentication process is managed by `AuthViewModel`.

Code Explanation

Breakdown

- **`@Composable fun SignUpScreen(navController: NavHostController, authViewModel: AuthViewModel)`**
  - Defines the sign-up screen as a composable function.
  - Parameters:
    - `navController`: The navigation controller to manage app navigation.
    - `authViewModel`: The view model that handles authentication.
- **State Variables**
  - **`var email by remember { mutableStateOf("") }`**: Holds the email input value.
  - **`var password by remember { mutableStateOf("") }`**: Holds the password input value.
  - **`var confirmPassword by remember { mutableStateOf("") }`**: Holds the confirmation password input value.
  - **`var showError by remember { mutableStateOf(false) }`**: Flag to show or hide error messages.
  - **`var errorMessage by remember { mutableStateOf("") }`**: Stores the error message text.

    - **`var isSigningUp by remember { mutableStateOf(false) }`**: Flag to indicate if sign-up process is ongoing.

- **`Box(modifier = Modifier.fillMaxSize().background(MaterialTheme.colorScheme.background))`**
    - Root container that fills the available space and sets the background color.
- **`Column(modifier = Modifier.fillMaxSize().padding(24.dp), ...)`**
    - Vertical layout container that holds the sign-up elements.
    - Centers its children both horizontally and vertically.
- **`Text("Create Account", style = MaterialTheme.typography.headlineLarge, ...)`**
    - Title text for the sign-up screen.
    - Uses the primary color and bold font weight.
- **`Spacer(modifier = Modifier.height(32.dp))`**
    - Adds vertical space between UI elements.
- **`Card(modifier = Modifier.fillMaxWidth().padding(horizontal = 16.dp), ...)`**
    - Displays the input fields for email, password, and confirmation password.
    - Parameters:
        - `modifier`: Sets the width and horizontal padding of the card.
        - `elevation`: Sets the card elevation for shadow effect, using `CardDefaults.cardElevation`.
        - `shape`: Sets the shape of the card with rounded corners.
- **`Column(modifier = Modifier.fillMaxWidth().padding(24.dp), verticalArrangement = Arrangement.spacedBy(16.dp))`**
    - Inner vertical layout within the card for input fields and buttons.
    - Arranges its children with spaced margins.

- **`AuthTextField(value = email, onValueChange = { email = it }, label = "Email", icon = Icons.Default.Email)`**
  - Input field for email, utilizing the custom `AuthTextField` composable.
  - Displays an email icon.
- **`AuthTextField(value = password, onValueChange = { password = it }, label = "Password", icon = Icons.Default.Lock, isPassword = true)`**
  - Input field for password, utilizing the custom `AuthTextField` composable.
  - Sets `isPassword` to true for input masking.
- **`AuthTextField(value = confirmPassword, onValueChange = { confirmPassword = it }, label = "Confirm Password", icon = Icons.Default.Lock, isPassword = true)`**
  - Input field for confirming the password.
  - Ensures the entered password matches the confirmation password.
- **`Button(onClick = { isSigningUp = true }, enabled = !isSigningUp)`**
  - Sign-up button.
  - Starts the sign-up process when clicked.
  - Disabled while `isSigningUp` is true to prevent multiple submissions.

Related Files

- **AuthViewModel.kt (Authentication ViewModel (AuthViewModel.kt)):** Manages user authentication and handles sign-up operations.
- **NavGraph.kt (Navigation Graph (NavGraph.kt)):** Defines the navigation routes, including the sign-up screen.
- **SignInScreen.kt (Sign-In Screen (SignInScreen.kt)):** Companion screen for user sign-in within the authentication flow.

Usage

The `SignUpScreen` is used for user registration within the app. Here is how it can be integrated and used:

Example Usage in NavGraph

Include the `SignUpScreen` in the navigation graph to enable navigation to the sign-up screen:

```
composable(route = Screens.SignUp.route) {
    SignUpScreen(navController = navController, authViewModel =
authViewModel)
}
```

Handling Sign-Up

The sign-up button initiates the registration process, managed by `AuthViewModel`:

```
Button(
    onClick = {
        isSigningUp = true
        // Register user
        viewModelScope.launch {
            if (password != confirmPassword) {
                showError = true
                errorMessage = "Passwords do not match"
                isSigningUp = false
                return@launch
            }

            val result =
authViewModel.createUserWithEmailAndPassword(email, password)
            isSigningUp = false
            if (result is AuthResult.Success) {
                // Navigate to sign-in or home screen
                navController.navigate(Screens.SignIn.route)
            } else if (result is AuthResult.Failure) {
                // Display error message
                showError = true
                errorMessage = result.message
```

```
            }
        }
    },
    enabled = !isSigningUp
) {
    Text("Create Account")
}
```

Additional Notes

- The screen provides a clean and user-friendly interface for user registration.
- It ensures responsive design and optimal user experience across different devices.
- Proper state management and error handling enhance the reliability of the registration process.
- Ensures password and confirmation password match to prevent user errors.

User Profile Screen (UserProfileScreen.kt)

Overview

This file defines the user profile screen component for the application. The screen displays the user's profile information and provides options for editing the profile and logging out. It utilizes Jetpack Compose to create a responsive and interactive UI.

Code Explanation

Breakdown

- **`@Composable fun UserProfileScreen(navController: NavHostController, authViewModel: AuthViewModel)`**
  - Defines the user profile screen as a composable function.
  - Parameters:
    - `navController`: The navigation controller to manage app navigation.
    - `authViewModel`: The view model that handles authentication and user data.
- **`val userProfile = authViewModel.getCurrentUserProfile()`**
  - Retrieves the current user's profile information from the `AuthViewModel`.
- **`Box(modifier = Modifier.fillMaxSize().background(MaterialTheme.colorScheme.background))`**
  - Root container that fills the available space and sets the background color.
- **`Column(modifier = Modifier.fillMaxSize().padding(16.dp), ...)`**
  - Vertical layout container that holds the profile elements.
  - Centers its children both horizontally and vertically.
- **`Card(...)`**

    - Displays the user's profile information within a card.
    - Parameters:
        - `modifier`: Sets the width and padding of the card.
        - `elevation`: Sets the card elevation for shadow effect, using `CardDefaults.cardElevation`.
        - `shape`: Sets the shape of the card with rounded corners.
- **`IconButton(onClick = { ... }, modifier = Modifier.align(Alignment.TopStart))`**
    - Button for editing the profile, positioned at the top-left of the card.
    - Navigates to the `UpdateUserProfile` screen when clicked.
    - Uses `Icons.Default.Edit` for the edit icon.
- **`IconButton(onClick = { authViewModel.signOut() }, modifier = Modifier.align(Alignment.TopEnd))`**
    - Button for logging out, positioned at the top-right of the card.
    - Signs the user out when clicked.
    - Uses `Icons.AutoMirrored.Filled.ExitToApp` for the logout icon.
- **`Column(modifier = Modifier.fillMaxWidth().padding(24.dp), ...)`**
    - Inner vertical layout within the card for profile details.
    - Arranges its children with spaced margins.

Related Files

- **NavGraph.kt (Navigation Graph (NavGraph.kt)):** Defines the navigation graph and routes, including `UpdateUserProfile`.
- **AuthViewModel.kt (Authentication ViewModel (AuthViewModel.kt)):** Manages user authentication and provides profile information.

- **Screens.kt (Screens (Screens.kt)):** Defines the screen routes and properties, such as `UpdateUserProfile`.

Usage

The `UserProfileScreen` is used to display and manage the user's profile within the app. Here is how it integrates with other components:

Example Usage in NavGraph

The `UserProfileScreen` is included in the navigation graph to facilitate navigation:

```
composable(route = Screens.UserProfile.route) {
    UserProfileScreen(navController = navController, authViewModel =
authViewModel)
}
```

Handling User Actions

Edit and logout actions are handled via icon buttons within the `Card`:

```
// Edit button
IconButton(
    onClick = { navController.navigate(Screens.UpdateUserProfile.route)
},
    modifier = Modifier.align(Alignment.TopStart)
) {
    Icon(
        imageVector = Icons.Default.Edit,
        contentDescription = "Edit Profile",
        tint = MaterialTheme.colorScheme.primary
    )
}

// Logout button
IconButton(
    onClick = { authViewModel.signOut() },
    modifier = Modifier.align(Alignment.TopEnd)
) {
```

```
    Icon(
        imageVector = Icons.AutoMirrored.Filled.ExitToApp,
        contentDescription = "Logout",
        tint = MaterialTheme.colorScheme.error
    )
}
```

Additional Notes

- This screen provides a clean and user-friendly interface for managing user profiles.
- The use of `Card` and `IconButton` components ensures a consistent and modern design.
- Proper navigation and state management are facilitated through `navController` and `authViewModel`.
- Profile details and actions like editing and logging out follow the common Material Design guidelines for better user experience.

Update User Profile Screen (UpdateUserProfileScreen.kt)

Overview

This file defines the update user profile screen component for the application. The screen allows users to update their profile information such as their display name and profile picture URL. It utilizes Jetpack Compose to create a responsive and interactive UI. The update process is managed by `AuthViewModel`.

Code Explanation

Breakdown

- **`@Composable fun UpdateUserProfileScreen(navController: NavHostController, authViewModel: AuthViewModel)`**
  - Defines the update user profile screen as a composable function.
  - Parameters:
    - `navController`: The navigation controller to manage app navigation.
    - `authViewModel`: The view model that handles authentication and profile updates.
- **State Variables**
  - **`val currentProfile = authViewModel.getCurrentUserProfile()`**: Retrieves the current user's profile.
  - **`var displayName by remember { mutableStateOf(currentProfile?.displayName ?: "") }`**: Holds the display name input value.
  - **`var photoUrl by remember { mutableStateOf(currentProfile?.photoUrl ?: "") }`**: Holds the photo URL input value.
  - **`var showError by remember { mutableStateOf(false) }`**: Flag to show or hide error

messages.

- `var errorMessage by remember { mutableStateOf("") }`: Stores the error message text.
- `var isUpdating by remember { mutableStateOf(false) }`: Flag to indicate if the update process is ongoing.

- `Box(modifier = Modifier.fillMaxSize().background(MaterialTheme.colorScheme.background))`
  - Root container that fills the available space and sets the background color.
- `IconButton(onClick = { navController.popBackStack() }, modifier = Modifier.padding(16.dp).align(Alignment.TopStart))`
  - Back button to navigate back to the previous screen.
  - Displays an arrow icon.
- `Column(modifier = Modifier.fillMaxSize().padding(top = 56.dp, start = 16.dp, end = 16.dp, bottom = 16.dp), ...)`
  - Vertical layout container that holds the update profile elements.
  - Centers its children horizontally and arranges them with spaces.
- `Text("Update Profile", style = MaterialTheme.typography.headlineMedium, ...)`
  - Title text for the update profile screen.
  - Uses a medium headline style and bold font weight.
- `Card(modifier = Modifier.fillMaxWidth().padding(vertical = 16.dp), ...)`
  - Displays the input fields for display name and photo URL.
  - Parameters:
    - `modifier`: Sets the width and vertical padding of the card.
    - `elevation`: Sets the card elevation for shadow effect, using `CardDefaults.cardElevation`.

    - `shape`: Sets the shape of the card with rounded corners.
- **`Column(modifier = Modifier.fillMaxWidth().padding(24.dp), verticalArrangement = Arrangement.spacedBy(16.dp))`**
    - Inner vertical layout within the card for input fields and buttons.
    - Arranges its children with spaced margins.
- **`ProfileTextField(value = displayName, onValueChange = { displayName = it }, label = "Username", icon)`**
    - Input field for the display name, utilizing a custom `ProfileTextField` composable.
    - Displays a person icon.
- **`ProfileTextField(value = photoUrl, onValueChange = { photoUrl = it }, label = "Photo URL", icon)`**
    - Input field for the photo URL, utilizing a custom `ProfileTextField` composable.
    - Displays a face icon.
- **`Button(onClick = { isUpdating = true }, enabled = !isUpdating)`**
    - Update button.
    - Starts the update process when clicked.
    - Disabled while `isUpdating` is true to prevent multiple submissions.

Related Files

- **AuthViewModel.kt (Authentication ViewModel (AuthViewModel.kt)):** Manages user authentication and handles profile updates.
- **NavGraph.kt (Navigation Graph (NavGraph.kt)):** Defines the navigation routes, including the update profile screen.
- **UserProfileScreen.kt (User Profile Screen (UserProfileScreen.kt)):** Companion screen for viewing user profiles within the user management flow.

Usage

The `UpdateUserProfileScreen` is used for updating user profile information within the app. Here is how it can be integrated and used:

Example Usage in NavGraph

Include the `UpdateUserProfileScreen` in the navigation graph to enable navigation to the update profile screen:

```
composable(route = Screens.UpdateProfile.route) {
    UpdateUserProfileScreen(navController = navController, authViewModel
= authViewModel)
}
```

Handling Profile Update

The update button initiates the profile update process, managed by `AuthViewModel`:

```
Button(
    onClick = {
        isUpdating = true
        // Update user profile
        viewModelScope.launch {
            val result = authViewModel.updateUserProfile(displayName,
photoUrl)
            isUpdating = false
            if (result is AuthResult.Success) {
                // Navigate back to profile screen
                navController.navigate(Screens.UserProfile.route)
            } else if (result is AuthResult.Failure) {
                // Display error message
                showError = true
                errorMessage = result.message
            }
        }
    },
    enabled = !isUpdating
) {
```

```
    Text("Update Profile")
}
```

Additional Notes

- The screen provides a clean and user-friendly interface for updating user profiles.
- It ensures responsive design and optimal user experience across different devices.
- Proper state management and error handling enhance the reliability of the update process.

Typography (Type.kt)

Overview

This file defines the typography styles for the application. It uses Jetpack Compose's `Typography` to set up the text styles, including font families, weights, sizes, line heights, and letter spacing. These styles can be applied globally to the app's text components.

Code Explanation

Breakdown

- **`val Typography = Typography(...)`**
  - Instantiates a `Typography` object that holds various text styles used throughout the app.
  - Customizes the default set of Material typography styles.
- **`bodyLarge = TextStyle(...)`**
  - Defines the text style for the body text of the application.
  - Parameters:
    - `fontFamily`: Sets the font family to `FontFamily.Default`.
    - `fontWeight`: Sets the font weight to `FontWeight.Normal`.
    - `fontSize`: Sets the font size to `16.sp`.
    - `lineHeight`: Sets the line height to `24.sp`.
    - `letterSpacing`: Sets the letter spacing to `0.5.sp`.
- **Commented Styles (e.g., `titleLarge`, `labelSmall`)**
  - Shows examples of other default text styles that can be overridden.
  - Each style can be customized similarly to `bodyLarge`.

- Useful for defining consistent typography settings across different text components in the app.

Related Files

- **Theme.kt (Theme (Theme.kt)):** Applies the typography settings alongside the color schemes.
- **ColorPalette.kt (Color Palette):** Defines the color palette used in the theme.
- **MainActivity.kt (Main Activity (MainActivity.kt)):** Entry point of the app that applies the `VociAppTheme`.

Usage

The `Typography` instance defined in `Type.kt` is used to maintain consistent text styles across the app. Here's how it is typically integrated:

Applying Typography in Theme

Integrate the typography settings in your theme setup within `Theme.kt`:

```
MaterialTheme(
    colorScheme = colorScheme,
    typography = Typography, // Apply the typography settings
    content = content
)
```

Customizing Text Styles

You can define additional or override existing text styles by uncommenting and customizing the provided examples:

```
val Typography = Typography(
    bodyLarge = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 16.sp,
        lineHeight = 24.sp,
```

```
        letterSpacing = 0.5.sp
    ),
    // Uncomment and customize as needed
    titleLarge = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 22.sp,
        lineHeight = 28.sp,
        letterSpacing = 0.sp
    ),
    // Add more text styles here
)
```

Additional Notes

- Consistent typography enhances the readability and aesthetics of the app.
- Defining typography styles in a central file enables easy updates and ensures uniformity across different screens.
- Customize various text components, such as titles, body text, and labels, to fit the app's design requirements.

Theme (Theme.kt)

Overview

This file defines the theme for the application, including color schemes for both dark and light modes. It utilizes Jetpack Compose's `MaterialTheme` to apply these themes globally throughout the app. It also supports dynamic theming based on the system's dark mode settings and the Android version.

Code Explanation

Breakdown

- **`private val DarkColorScheme = darkColorScheme(...)`**
  - Defines the color scheme for dark mode.
  - Uses `darkColorScheme` to specify colors for various UI elements like primary, secondary, background, and error.
- **`private val LightColorScheme = lightColorScheme(...)`**
  - Defines the color scheme for light mode.
  - Uses `lightColorScheme` to specify colors for various UI elements similar to the dark mode setup.
- **`@Composable fun VociAppTheme(...) { ... }`**
  - This composable function sets up the theme using `MaterialTheme`.
  - Parameters:
    - `darkTheme`: Boolean flag to indicate whether dark theme is enabled. Defaults to system setting with `isSystemInDarkTheme()`.
    - `dynamicColor`: Boolean flag to indicate whether dynamic theming is enabled. Defaults to `false`.

    - `content`: Lambda that contains the UI content to which the theme will be applied.
- **`val colorScheme = when { ... }`**
    - Determines the appropriate color scheme based on the `darkTheme` and `dynamicColor` flags.
        - Uses `dynamicDarkColorScheme` or `dynamicLightColorScheme` for dynamic theming if supported (Android 12+).
        - Falls back to `DarkColorScheme` or `LightColorScheme` based on the `darkTheme` flag.
- **`MaterialTheme(...)`**
    - Applies the selected color scheme to the app's UI.
    - Parameters:
        - `colorScheme`: The chosen color scheme (either dark, light, or dynamic).
        - `typography`: Typography settings applied to the text throughout the app.
        - `content`: Lambda containing the composable content to which the theme is applied.

Related Files

- **ColorPalette.kt (Color Palette):** Defines the color palette used in the theme.
- **Type.kt (Typography (Type.kt)):** Defines the typography settings used in the theme.
- **MainActivity.kt (Main Activity (MainActivity.kt)):** Entry point of the app that applies the `VociAppTheme`.

Usage

The `VociAppTheme` function is used to wrap the entire content of the app to apply the theme consistently. Here's how it is typically used:

Applying the Theme

Wrap your composable content with the `VociAppTheme` to apply the theme:

```
setContent {
    VociAppTheme {
        // Composable content that uses the theme...
    }
}
```

Dynamic and Dark Mode Theming

The theme can automatically adjust based on the system's dark mode setting and support dynamic colors on Android 12+:

```
@Composable
fun MyApp() {
    VociAppTheme(
        darkTheme = isSystemInDarkTheme(), // Automatically adjust to
system's dark mode
        dynamicColor = true // Enable dynamic colors on supported
devices
    ) {
        // UI content with applied dynamic theme
    }
}
```

Additional Notes

- The dark and light color schemes are meticulously defined to ensure a cohesive and accessible UI.
- Dynamic theming allows the app to better integrate with the system themes available on newer Android devices.

- Usage of `MaterialTheme` ensures that the theme settings are propagated throughout the app, maintaining consistency.

Color Palette

This file defines the color palette used throughout the app. It provides color values for both light and dark themes, ensuring consistency and accessibility.

Color Definitions

Color Name	Light Theme Value	Dark Theme Value	Description
Primary	`0xFFFF9800`	`0xFFFFA726`	The main color of the app, used for primary elements.
On Primary	`0xFF000000`	`0xFF000000`	Color used for content on top of the primary color.
Primary Container	`0xFFFFE0B2`	`0xFFE65100`	Color used for containers with primary content.
On Primary Container	`0xFF613D00`	`0xFFFFE0B2`	Color used for content on top of primary containers.
Secondary	`0xFF2196F3`	`0xFF64B5F6`	A secondary color used for accents and highlights.
On Secondary	`0xFFFFFFFF`	`0xFF000000`	Color used for content on top of the secondary color.
Secondary Container	`0xFFBBDEFB`	`0xFF1976D2`	Color used for containers with secondary content.
On Secondary Container	`0xFF0D3C61`	`0xFFBBDEFB`	Color used for content on top of secondary containers.
Tertiary	`0xFF4CAF50`	`0xFF81C784`	A tertiary color used for additional accents.
On Tertiary	`0xFFFFFFFF`	`0xFF000000`	Color used for content on top of the tertiary color.

Tertiary Container	0xFFC8E6C9	0xFF2E7D32	Color used for containers with tertiary content.
On Tertiary Container	0xFF1B5E20	0xFFC8E6C9	Color used for content on top of tertiary containers.
Error	0xFFD32F2F	0xFFEF5350	Color used for error states.
On Error	0xFFFFFFFF	0xFF000000	Color used for content on top of error backgrounds.
Error Container	0xFFFFCDD2	0xFFB71C1C	Color used for containers with error content.
On Error Container	0xFF641414	0xFFFFCDD2	Color used for content on top of error containers.
Background	0xFFFAFAFA	0xFF212121	The background color of the app.
On Background	0xFF212121	0xFFFFFFFF	Color used for content on top of the background.
Surface	0xFFFFFFFF	0xFF424242	The surface color of UI elements.
On Surface	0xFF212121	0xFFFFFFFF	Color used for content on top of surfaces.
Surface Variant	0xFFEEEEEE	0xFF616161	A variant of the surface color.

On Surface Variant	`0xFF757575`	`0xFFEEEEEE`	Color used for content on top of surface variants.
Outline	`0xFFBDBDBD`	`0xFF757575`	Color used for outlines and borders.

Usage

These color values can be accessed through the `ColorPalette` object in the `Colors.kt` file:

```
val primaryColor = ColorPalette.PrimaryLight // For light theme
val secondaryColor = ColorPalette.SecondaryDark // For dark theme
```

Authentication ViewModel (AuthViewModel.kt)

Overview

This file defines the `AuthViewModel` class, which handles user authentication and profile management. It interacts with Firebase Authentication to perform actions such as sign-in, sign-out, user creation, and profile updates. The class utilizes Kotlin Coroutines and StateFlow to manage and observe the authentication state.

Code Explanation

Breakdown

- **`class AuthViewModel : ViewModel()`**
  - Inherits from `ViewModel` to manage UI-related data in a lifecycle-conscious way.
- **`private val _authState = MutableStateFlow<AuthState>(AuthState.Uninitialized)`**
  - Holds the mutable state of the authentication status.
  - Initially set to `Uninitialized`.
- **`val authState: StateFlow<AuthState> = _authState.asStateFlow()`**
  - Exposes a read-only version of `_authState`.
- **`private val auth: FirebaseAuth = FirebaseAuth.getInstance()`**
  - Initializes an instance of Firebase Authentication.
- **`private val authStateListener = FirebaseAuth.AuthStateListener { ... }`**
  - Listener that updates the `_authState` based on the current user authentication state.
  - Sets the state to `Authenticated` if a `FirebaseUser` is present; otherwise, sets it to `Unauthenticated`.

- **`init { ... }`**
  - Adds the `authStateListener` to Firebase Authentication when the `AuthViewModel` is initialized.
- **`override fun onCleared() { ... }`**
  - Removes the `authStateListener` when the `ViewModel` is cleared to prevent memory leaks.
- **`suspend fun signInWithEmailAndPassword(email: String, password: String): AuthResult`**
  - Signs in a user with an email and password.
  - Returns `AuthResult.Success` if successful or `AuthResult.Failure` with an error message if unsuccessful.
- **`suspend fun createUserWithEmailAndPassword(email: String, password: String): AuthResult`**
  - Creates a new user account with an email and password.
  - Returns `AuthResult.Success` if successful or `AuthResult.Failure` with an error message if unsuccessful.
- **`fun signOut()`**
  - Signs out the current user.
- **`suspend fun updateUserProfile(displayName: String?, photoUrl: String?): AuthResult`**
  - Updates the user's profile information such as display name and profile photo.
  - Returns `AuthResult.Success` if successful or `AuthResult.Failure` with an error message if unsuccessful.
- **`fun getCurrentUserProfile(): UserProfile?`**
  - Retrieves the current user's profile information.
  - Returns a `UserProfile` object or null if no user is logged in.
- **`fun getCurrentUser(): FirebaseUser?`**

- Retrieves the currently authenticated `FirebaseUser`.

Helper Classes and Sealed Classes

- **`sealed class AuthResult`**
  - Represents the result of an authentication operation.
  - Contains two subclasses: `Success` and `Failure`.
- **`data class UserProfile(val displayName: String? = null, val photoUrl: String? = null)`**
  - Data class that stores user profile information such as display name and photo URL.

Related Files

- **UserProfileScreen.kt (User Profile Screen (UserProfileScreen.kt)):** Utilizes `AuthViewModel` to display and manage user profile information.
- **NavGraph.kt (Navigation Graph (NavGraph.kt)):** Defines the navigation routes including those for authentication-related screens (sign-in, sign-up).
- **MainActivity.kt (Main Activity (MainActivity.kt)):** Sets up the main structure and integrates the `AuthViewModel`.

Usage

The `AuthViewModel` is used to manage authentication and user profile operations within the app. Here's how it can be integrated and used:

Example Usage in Composable

Integrate the `AuthViewModel` in your composable screen to access and manipulate authentication data:

```
@Composable
fun UserProfileScreen(navController: NavHostController, authViewModel:
AuthViewModel) {
    val userProfile = authViewModel.getCurrentUserProfile()
```

```
    // UI elements to display user profile data
}
```

Handling Authentication

The `AuthViewModel` provides methods for sign-in, sign-out, user creation, and profile updates:

```
// Sign in with email and password
viewModelScope.launch {
    val result = authViewModel.signInWithEmailAndPassword(email,
password)
    if (result is AuthResult.Success) {
        // Handle successful sign-in
    } else if (result is AuthResult.Failure) {
        // Handle sign-in failure
    }
}

// Sign out the current user
authViewModel.signOut()

// Update user profile
viewModelScope.launch {
    val result = authViewModel.updateUserProfile(displayName, photoUrl)
    if (result is AuthResult.Success) {
        // Handle successful profile update
    } else if (result is AuthResult.Failure) {
        // Handle profile update failure
    }
}
```

Additional Notes

- Using `StateFlow` allows observing authentication state changes efficiently in a Compose UI.

- Incorporates best practices for handling authentication operations with Firebase in a Kotlin-based Android app.
- Proper usage and disposal of `AuthStateListener` prevent memory leaks and ensure the application remains responsive.