Navegación de NimbusGuard

Este código implementa la navegación de la aplicación utilizando Jetpack Compose, permitiendo a los usuarios moverse entre diferentes pantallas de la app. Se utiliza `NavHost` para definir las rutas y `NavController` para gestionar la navegación.

Manejo de las Pantallas de la App

```
sealed class ManejoDeLasPantallasDeLaApp(val ruta: String) {
    object PantallaDeBienvenida : ManejoDeLasPantallasDeLaApp("pantalla_bienvenida")
    object PantallaDeRegistro : ManejoDeLasPantallasDeLaApp("pantalla_registro")
    object PantallaDeLogin : ManejoDeLasPantallasDeLaApp("pantalla_login")
    object PantallaDeInicio : ManejoDeLasPantallasDeLaApp("pantalla_inicio")
    object PantallaDePerfil : ManejoDeLasPantallasDeLaApp("pantalla_perfil")
    object PantallaDeNotificacion : ManejoDeLasPantallasDeLaApp("pantalla_notificaciones")
    object PantallaPrincipal : ManejoDeLasPantallasDeLaApp("pantalla_principal")
    object PantallaConInfoApp: ManejoDeLasPantallasDeLaApp("pantalla_con_info_app")
}
```

Este archivo contiene una clase sellada `ManejoDeLasPantallasDeLaApp` que define las rutas de navegación para las diferentes pantallas de la aplicación.

Estructura de la Clase

- **Clase Sellada**: `ManejoDeLasPantallasDeLaApp`
    - **Propiedad**: `ruta` (String)

Objetos de Navegación

- **PantallaDeBienvenida**: Ruta para la pantalla de bienvenida.
- **PantallaDeRegistro**: Ruta para la pantalla de registro de usuarios.
- **PantallaDeLogin**: Ruta para la pantalla de inicio de sesión.
- **PantallaDeInicio**: Ruta para la pantalla principal.
- **PantallaDePerfil**: Ruta para la pantalla de perfil del usuario.
- **PantallaDeNotificacion**: Ruta para la pantalla de notificaciones.
- **PantallaPrincipal**: Ruta para la pantalla principal de la aplicación.
- **PantallaConInfoApp**: Ruta para la pantalla que muestra información sobre la aplicación.

Estructura del Código de Navegación

Función de Navegación

```
@Composable
fun navegacionDeLaApp() {
    val navController = rememberNavController()
    val authService = UserAuthService()
    val context = navController.context
    val viewModel: BotonDeAlertaViewModel = viewModel(factory = BotonDeAlertaViewModelFactory(context))
    val buttonStates by viewModel.buttonStates.collectAsState(initial = emptyMap())

    NavHost(navController = navController, startDestination = ManejoDeLasPantallasDeLaApp.PantallaDeBienvenida.ruta) {
        composable(route = ManejoDeLasPantallasDeLaApp.PantallaPrincipal.ruta + "/{uid}",
            arguments = listOf(navArgument("uid") { type = NavType.StringType })) {
            PantallaPrincipal(navController, buttonStates, onButtonStatusChange = { buttonId, isEnabled ->
                viewModel.viewModelScope.launch {
                    viewModel.updateButtonState(buttonId, isEnabled)
                }
            })
        }
        composable(route = ManejoDeLasPantallasDeLaApp.PantallaDeInicio.ruta + "/{uid}",
            arguments = listOf(navArgument("uid") { type = NavType.StringType })) {
            PantallaDeInicio(navController, buttonStates, onButtonStatusChange = { buttonId, isEnabled ->
                viewModel.viewModelScope.launch {
                    viewModel.updateButtonState(buttonId, isEnabled)
                }
            })
        }
        composable(route = ManejoDeLasPantallasDeLaApp.PantallaDeRegistro.ruta) {
            PantallaDeRegistro(navController)
        }
        composable(route = ManejoDeLasPantallasDeLaApp.PantallaDeLogin.ruta) {
            PantallaDeLogin(navController = navController, users = listOf())
        }
        composable(route = ManejoDeLasPantallasDeLaApp.PantallaConInfoApp.ruta) {
            PantallaConInfoApp(navController)
        }
        composable(route = ManejoDeLasPantallasDeLaApp.PantallaDeNotificacion.ruta) {
            PantallaDeNotificacion(navController = navController)
        }
        composable(route = ManejoDeLasPantallasDeLaApp.PantallaDePerfil.ruta + "/{uid}",
            arguments = listOf(navArgument("uid") { type = NavType.StringType })) {
            val uid = it.arguments?.getString("uid") ?: ""
            PantallaDePerfil(navController = navController, uid = uid, authService = authService)
        }
        composable(route = ManejoDeLasPantallasDeLaApp.PantallaDeBienvenida.ruta) {
            PantallaDeBienvenida(navController)
        }
        composable(route = ManejoDeLasPantallasDeLaApp.PantallaConInfoApp.ruta) {
            PantallaConInfoApp(navController)
        }
    }
}
```

Flujo de Navegación

Inicio de la Aplicación

- La app comienza en la `PantallaDeBienvenida`.
- Los usuarios pueden navegar a otras pantallas, como:
    - Inicio de Sesión
    - Registro
    - Pantalla Principal
    - Pantalla de Inicio
    - Pantalla de Notificaciones
    - Pantalla de Perfil

Navegación entre Pantallas

- Se utilizan `NavController` y `NavHost` para manejar las rutas de la aplicación.
- Cada pantalla está definida como un `composable` que se muestra en función de la ruta activa.

Componentes Clave

- **NavHost**: Define la estructura de navegación de la aplicación.
- **NavController**: Controla la navegación entre las diferentes pantallas.
- **Composable**: Funciones que representan cada pantalla de la app.

Pantalla Principal

El código de esta sección maneja la estructura principal de navegación de la aplicación de NimbusGuard, implementando una barra de navegación inferior y una barra de información superior. La pantalla principal (`PantallaPrincipal`) permite a los usuarios moverse entre las secciones clave de la app: Inicio, Perfil y Notificaciones, utilizando un `NavController` para la navegación y proporcionando notificaciones dinámicas y actualizaciones de estado en tiempo real.

Estructura del Código

PantallaPrincipal define la estructura de la pantalla principal, configurando la barra de navegación inferior y el conteo de notificaciones para la app.

```
@Composable
fun PantallaPrincipal(
    navController: NavController,
    buttonStates: Map<String, Boolean>,
    onButtonStatusChange: (String, Boolean) -> Unit,
) {
    val uid = FirebaseAuth.getInstance().currentUser?.uid ?: ""
    val contadorDeNotificaciones by remember {
        derivedStateOf { NotificacionRepository.obtenerNotificaciones(uid).size }
    }
    barraDeNavegacionInferior(
        navController = navController,
        buttonStates = buttonStates,
        onButtonStatusChange = onButtonStatusChange,
        contadorDeNotificaciones = contadorDeNotificaciones
    )
}
```

Barra De Informacion Superior

`(barraDeInformacionSuperior)` Establece la barra de información en la parte superior, con un saludo y un ícono de la app, mostrando un mensaje toast al hacer clic en el ícono.

```
@OptIn(ExperimentalMaterial3Api::class, ExperimentalMaterial3Api::class)
@Composable
fun barraDeInformacionSuperior(){
    val context = LocalContext.current.applicationContext
    TopAppBar(
        title = {
            Text(text = "Bienvenido a Nimbus Guard",
                color = Color.Black,
                fontSize = 27.sp,
                textAlign = TextAlign.Center,
                modifier = Modifier.fillMaxWidth(),
                fontFamily = fontFamily,
                fontStyle = FontStyle.Italic
            )
        },
        navigationIcon = {
            IconButton(onClick = { Toast.makeText(context, "Icono", Toast.LENGTH_SHORT).show()}) {
                Image(painter = painterResource(id = R.drawable.logoappnimbusguard),
                    contentDescription = "Logo",
                    Modifier.size(80.dp)
                )
            }
        },
        colors = TopAppBarDefaults.topAppBarColors(
            containerColor = Color.White
        )
    )
}
```

Barra de Navegacion Inferior

`(barraDeNavegacionInferior)` Maneja la navegación entre pantallas utilizando NavigationBarItem para cada ítem, cada uno con su ícono correspondiente (Inicio, Perfil, Notificaciones). Implementa una BadgedBox para mostrar el conteo de notificaciones.

```
@Composable
fun barraDeNavegacionInferior(
    navController: NavController,
    buttonStates: Map<String, Boolean>,
    onButtonStatusChange: (String, Boolean) -> Unit,
    contadorDeNotificaciones: Int
) {
    val context = LocalContext.current
    val viewModel: BotonDeAlertaViewModel = viewModel(factory = BotonDeAlertaViewModelFactory(context))
    val _buttonStates by viewModel.buttonStates.collectAsState(initial = emptyMap())

    val listaDeItemsDeLaBarraDeNavegacion = listOf(
        ItemDeLaBarra("Inicio", Icons.Default.Home, 0),
        ItemDeLaBarra("Perfil", Icons.Default.Person, 0),
        ItemDeLaBarra("Notificaciones", Icons.Default.Notifications, contadorDeNotificaciones)
    )

    var selectedIndex by rememberSaveable { mutableStateOf(0) }

    Scaffold(
        modifier = Modifier.fillMaxSize(),
        bottomBar = {
            NavigationBar {
                listaDeItemsDeLaBarraDeNavegacion.forEachIndexed { index, itemDeLaBarra ->
                    NavigationBarItem(
                        selected = selectedIndex == index,
                        onClick = { selectedIndex = index },
                        icon = {
                            BadgedBox(badge = {
                                if (itemDeLaBarra.contadorDeNotificaciones > 0) {
                                    Badge {
                                        Text(text = itemDeLaBarra.contadorDeNotificaciones.toString())
                                    }
                                }
                            }) {
                                Icon(
                                    imageVector = itemDeLaBarra.icono,
                                    contentDescription = "Icono"
                                )
                            }
                        },
                        label = {
                            Text(text = itemDeLaBarra.texto)
                        }
                    )
                }
            }
        }
    ) { innerPadding ->
        contenidoDeLaBarraDeNavegacionInferior(
            modifier = Modifier.padding(innerPadding),
            selectedIndex = selectedIndex,
            navController = navController,
            buttonStates = _buttonStates,
            onButtonStatusChange = onButtonStatusChange
        )
        barraDeInformacionSuperior()
    }
}
```

Contenido de la Barra de Navegacion Inferior

`(contenidoDeLaBarraDeNavegacionInferior)` Establece el contenido que se muestra en la pantalla de acuerdo con el índice de seleccionado, controlando la

navegación entre: `PantallaDeInicio, PantallaDePerfil, y PantallaDeNotificacion.`

```
@Composable
fun contenidoDeLaBarraDeNavegacionInferior(
    modifier: Modifier,
    selectedIndex: Int,
    navController: NavController,
    buttonStates: Map<String, Boolean>,
    onButtonStatusChange: (String, Boolean) -> Unit
) {
    when (selectedIndex) {
        0 -> PantallaDeInicio(
            navController = navController,
            buttonStates = buttonStates,
            onButtonStatusChange = onButtonStatusChange
        )
        1 -> {
            val uid = FirebaseAuth.getInstance().currentUser?.uid ?: ""
            val authService = UserAuthService()
            PantallaDePerfil(navController = navController, uid = uid, authService = authService)
        }
        2 -> PantallaDeNotificacion(navController = navController)
    }
}
```

Clase ItemDeLaBarra

La clase **ItemDeLaBarra** representa un elemento individual en la barra de navegación de la pantalla principal de la aplicación. Cada elemento de la barra de navegación puede contener un texto, un ícono y un contador de notificaciones asociado.

```
class ItemDeLaBarra(
    val texto: String,
    val icono: ImageVector,
    val contadorDeNotificaciones: Int
)
```

Propiedades

- **texto: String**
  Descripción: Texto que se muestra en el elemento de la barra de navegación. Este texto suele representar la funcionalidad del ítem, como "Inicio", "Perfil", o "Notificaciones".
- **icono: ImageVector**
  Descripción: Ícono que se asocia con el texto del elemento de la barra de navegación. Se utiliza un objeto de tipo `ImageVector` para representar gráficamente la función del ítem.
- **contadorDeNotificaciones: Int**
  Descripción: Contador que muestra la cantidad de notificaciones asociadas a este ítem. Por ejemplo, en el caso del ítem "Notificaciones", este contador indica cuántas notificaciones pendientes tiene el usuario.

Funcionamiento

La clase **ItemDeLaBarra** se utiliza para crear y gestionar los elementos visuales que aparecen en la barra de navegación de la pantalla principal. Cada instancia de esta clase contiene toda la información necesaria para representar un ítem de la barra, permitiendo que la interfaz de usuario muestre tanto texto como un ícono y un contador de notificaciones.

Flujo de Navegación

Inicio de la Aplicación

La aplicación inicia en PantallaPrincipal, desde donde el usuario puede navegar a las siguientes pantallas:

- Pantalla de Inicio
- Pantalla de Notificaciones
- Pantalla de Perfil

Navegación entre Pantallas

La navegación es controlada por `NavController` y `NavHost,` utilizando `NavigationBar` para manejar las rutas de la aplicación en la parte inferior de la pantalla. selectedIndex se actualiza según la pestaña seleccionada.

Componentes Clave

- **NavController**: Controla la navegación entre las diferentes pantallas.
- **NavHost**: Define la estructura de navegación de la aplicación.
- **Composable**: Funciones que representan cada pantalla de la app `(PantallaPrincipal, PantallaDeInicio, PantallaDePerfil, PantallaDeNotificacion)` .

Pantalla de Inicio

Este archivo contiene la función `PantallaDeInicio` , que se encarga de renderizar la pantalla principal de la aplicación donde los usuarios pueden enviar alertas. Utiliza un `NavController` para la navegación y un `ViewModel` para gestionar el estado de los botones.

Funciones

PantallaDeInicio

```
@Composable
fun PantallaDeInicio(
    navController: NavController,
    buttonStates: Map<String, Boolean>,
    onButtonStatusChange: (String, Boolean) -> Unit
)
```

Parámetros

- navController: Controlador de navegación que gestiona la navegación entre las pantallas.
- buttonStates: Mapa que representa el estado de los botones (habilitado/deshabilitado).
- onButtonStatusChange: Función que se llama para actualizar el estado de un botón.

Descripción

- Obtiene el contexto actual y el `ViewModel BotonDeAlertaViewModel` .
- Llama a la función `contenidoPantallaDeInicio` para renderizar la UI.

contenidoPantallaDeInicio

```
@Composable
fun contenidoPantallaDeInicio(
    navController: NavController,
    buttonStates: Map<String, Boolean>,
    onButtonStatusChange: (String, Boolean) -> Unit,
    viewModel: BotonDeAlertaViewModel
)
```

Parámetros

- `navController:` Controlador de navegación que gestiona la navegación entre las pantallas.
- `buttonStates:` Mapa que representa el estado de los botones (habilitado/deshabilitado).
- `onButtonStatusChange:` Función que se llama para actualizar el estado de un botón.
- `viewModel:` Instancia del BotonDeAlertaViewModel para gestionar el estado de los botones.

Descripción

- Obtiene el `uid` y `alertId` de los argumentos de la navegación.
- Solicita permisos de ubicación.
- Define la configuración de varios botones utilizando `ButtonBuilder` .
- Renderiza un Column que incluye un título y una lista de botones.
- Cada botón envía una alerta al presionar, utilizando la clase `BotonDeAlerta` .

Estructura de los Botones

Cada botón se configura con las siguientes propiedades:

- **Color del Contenedor**: Color de fondo del botón.
- **Color del Contenido**: Color del texto y de los iconos.
- **Dimensiones:** Ancho, alto y padding del botón.
- **Imagen:** Icono asociado al botón.
- **Título:** Texto que aparece en el botón.
- **Color y Tamaño del Título:** Configuración visual del texto del botón.

```
ButtonBuilder()
    .setContainerColor(Color.Red)
    .setContentColor(Color.White)
    .setWidthButton(300.dp)
    .setHeightButton(100.dp)
    .setPaddingButton(10.dp)
    .setImageId(R.drawable.ambulance)
    .setSizeImage(60.dp)
    .setTitle("Urgencia Médica")
    .setColorTitle(Color.White)
    .setFontSizeTitle(20.sp)
    .setPaddingTitle(10.dp)
    .build()
```

ButtonBuilder y ButtonConfig

Este archivo contiene la clase `ButtonBuilder` y la clase de datos `ButtonConfig`, que se utilizan para configurar y construir botones en la aplicación, específicamente para las alertas en la pantalla de inicio.

Clase: ButtonConfig

La clase `ButtonConfig` define la configuración de un botón. Utiliza propiedades predeterminadas y realiza validaciones en el inicializador.

```
data class ButtonConfig(
    var containerColor: Color = Color.Red,
    var contentColor: Color = Color.White,
    var widthButton: Dp = 300.dp,
    var heightButton: Dp = 100.dp,
    var paddingButton: Dp = 10.dp,
    var imageId: Int = 1,
    var sizeImage: Dp = 60.dp,
    var title: String = "Emergencia",
    var colorTitle: Color = Color.White,
    var fontSizeTitle: TextUnit = 20.sp,
    var paddingTitle: Dp = 10.dp
) {
    init {
        require(imageId != 0) { "imageId no puede estar vacío o nulo" }
        require(title.isNotEmpty()) { "title no puede estar vacío o nulo" }
    }
}
```

Propiedades

- **containerColor**: Color de fondo del botón (predeterminado: `Color.Red`).
- **contentColor**: Color del texto y de los iconos (predeterminado: `Color.White`).
- **widthButton**: Ancho del botón en dp (predeterminado: `300.dp`).
- **heightButton**: Alto del botón en dp (predeterminado: `100.dp`).
- **paddingButton**: Padding interno del botón en dp (predeterminado: `10.dp`).
- **imageId**: ID del recurso de imagen (predeterminado: `1`). Debe ser distinto de `0`.
- **sizeImage**: Tamaño de la imagen en dp (predeterminado: `60.dp`).
- **title**: Título del botón (predeterminado: `"Emergencia"`). No puede estar vacío.
- **colorTitle**: Color del título (predeterminado: `Color.White`).
- **fontSizeTitle**: Tamaño de fuente del título (predeterminado: `20.sp`).
- **paddingTitle**: Padding del título en dp (predeterminado: `10.dp`).

Clase: ButtonBuilder

La clase `ButtonBuilder` permite construir una instancia de `ButtonConfig` utilizando el patrón Builder.

```
class ButtonBuilder {
    private var config = ButtonConfig()
    fun setContainerColor(color: Color) = apply { config = config.copy(containerColor = color) }
    fun setContentColor(color: Color) = apply { config = config.copy(contentColor = color) }
    fun setWidthButton(width: Dp) = apply { config = config.copy(widthButton = width) }
    fun setHeightButton(height: Dp) = apply { config = config.copy(heightButton = height) }
    fun setPaddingButton(padding: Dp) = apply { config = config.copy(paddingButton = padding) }
    fun setImageId(imageId: Int) = apply { config = config.copy(imageId = imageId) }
    fun setSizeImage(size: Dp) = apply { config = config.copy(sizeImage = size) }
    fun setTitle(title: String) = apply { config = config.copy(title = title) }
    fun setColorTitle(color: Color) = apply { config = config.copy(colorTitle = color) }
    fun setFontSizeTitle(size: TextUnit) = apply { config = config.copy(fontSizeTitle = size) }
    fun setPaddingTitle(padding: Dp) = apply { config = config.copy(paddingTitle = padding) }
    fun build() = config
}
```

Métodos

- `setContainerColor(color: Color):` Establece el color del contenedor del botón.
- `setContentColor(color: Color):` Establece el color del contenido del botón.
- `setWidthButton(width: Dp):` Establece el ancho del botón.
- `setHeightButton(height: Dp):` Establece el alto del botón.
- `setPaddingButton(padding: Dp):` Establece el padding del botón.
- `setImageId(imageId: Int):` Establece el ID de la imagen del botón.
- `setSizeImage(size: Dp):` Establece el tamaño de la imagen del botón.
- `setTitle(title: String):` Establece el título del botón.
- `setColorTitle(color: Color):` Establece el color del título del botón.
- `setFontSizeTitle(size: TextUnit):` Establece el tamaño de fuente del título del botón.
- `setPaddingTitle(padding: Dp):` Establece el padding del título del botón.
- `build():` Devuelve la configuración del botón como una instancia de ButtonConfig.

Componente BotonDeAlerta

El componente **BotonDeAlerta** es una implementación de un botón de alerta en Jetpack Compose que permite a los usuarios enviar alertas junto con su ubicación actual. Este botón tiene varias configuraciones personalizables y se integra con servicios de localización y Firestore.

```
@Composable
fun BotonDeAlerta(
    uid: String,
    idAlert: String,
    buttonId: String,
    buttonConfig: ButtonConfig,
    fusedLocationClient: FusedLocationProviderClient,
    permissionsState: MultiplePermissionsState,
    context: Context,
    onButtonStatusChange: (Boolean) -> Unit
)
```

Parámetros

1. **uid: String**
   Descripción: Identificador único del usuario que está enviando la alerta.
   Requerido: Sí

2. **idAlert: String**
   Descripción: Identificador de la alerta que se está enviando.
   Requerido: Sí

3. **buttonId: String**
   Descripción: Identificador único del botón de alerta.
   Requerido: Sí

4. **buttonConfig: ButtonConfig**
   Descripción: Configuración del botón que define propiedades como color, tamaño y texto.

Requerido: Sí

5. **fusedLocationClient: FusedLocationProviderClient**
   Descripción: Cliente de ubicación utilizado para obtener la ubicación actual del usuario.
   Requerido: Sí

6. **permissionsState: MultiplePermissionsState**
   Descripción: Estado de permisos que se utiliza para gestionar los permisos de ubicación.
   Requerido: Sí

7. **context: Context**
   Descripción: Contexto de la aplicación, necesario para acceder a recursos y servicios.
   Requerido: Sí

8. **onButtonStatusChange: (Boolean) -> Unit**
   Descripción: Callback que se invoca para notificar el cambio del estado del botón (habilitado/deshabilitado).
   Requerido: Sí

Funcionamiento

```
 @RequiresApi(Build.VERSION_CODES.O)
@OptIn(ExperimentalPermissionsApi::class)
@Composable
fun BotonDeAlerta(
    uid: String,
    idAlert: String,
    buttonId: String,
    navController: NavController,
    buttonConfig: ButtonConfig,
    fusedLocationClient: FusedLocationProviderClient,
    permissionsState: MultiplePermissionsState,
    context: Context,
    onButtonStatusChange: (Boolean) -> Unit
) {
    var showAlertDialog by remember { mutableStateOf(false) }
    var showConfirmationDialog by remember { mutableStateOf(false) }
    var latitud by remember { mutableStateOf(0.0) }
    var longitud by remember { mutableStateOf(0.0) }
    var buttonTitle by remember { mutableStateOf("") }
    var timeRemaining by remember { mutableStateOf(0) }
    var isButtonEnabled = remember { mutableStateOf(true) }

    LaunchedEffect(Unit) {
        val (buttonState, remainingTime) = getButtonState(context, uid, buttonId)
        isButtonEnabled.value = buttonState
        timeRemaining = remainingTime

        if (!isButtonEnabled.value && timeRemaining > 0) {
            startTimer(context, uid, buttonId, timeRemaining, isButtonEnabled) { updatedTimeRemaining ->
                timeRemaining = updatedTimeRemaining
                if (updatedTimeRemaining == 0) {
                    isButtonEnabled.value = true
                }
            }
        }
    }

    val requestPermissionLauncher = rememberLauncherForActivityResult(
        ActivityResultContracts.RequestMultiplePermissions()
    ) { permissions ->
        permissions.entries.forEach {
            if (!it.value) {
                Toast.makeText(context, "Permiso de ubicación no concedido.", Toast.LENGTH_SHORT).show()
            }
        }
    }

    Button(
        onClick = {
            if (isButtonEnabled.value) {
```

```
            if (isButtonEnabled.value) {
                if (permissionsState.allPermissionsGranted) {
                    getCurrentLocation(fusedLocationClient) { lat, lon ->
                        latitud = lat
                        longitud = lon
                        buttonTitle = buttonConfig.title
                        showAlertDialog = true
                    }
                } else {
                    requestPermissionLauncher.launch(arrayOf(
                        Manifest.permission.ACCESS_FINE_LOCATION,
                        Manifest.permission.ACCESS_COARSE_LOCATION
                    ))
                }
            } else {
                Toast.makeText(context, "Este botón está deshabilitado temporalmente.", Toast.LENGTH_SHORT).show()
            }
        },
        enabled = isButtonEnabled.value,
        colors = ButtonDefaults.buttonColors(
            containerColor = buttonConfig.containerColor,
            contentColor = buttonConfig.contentColor
        ),
        modifier = Modifier
            .size(buttonConfig.widthButton, buttonConfig.heightButton)
            .padding(buttonConfig.paddingButton)
    ) {
        Image(
            painter = painterResource(id = buttonConfig.imageId),
            contentDescription = null,
            modifier = Modifier.size(buttonConfig.sizeImage)
        )
        Text(
            textAlign = TextAlign.Center,
            text = buttonConfig.title,
            color = buttonConfig.colorTitle,
            fontSize = buttonConfig.fontSizeTitle,
            modifier = Modifier.padding(buttonConfig.paddingTitle)
        )
    }

    if (!isButtonEnabled.value && timeRemaining > 0) {
        Text(
            text = "Reactivación en: ${timeRemaining} segundos",
            modifier = Modifier.padding(8.dp)
        )
    }

    if (showAlertDialog) {
        showAlertInputDialog(
            state = "Pendiente",
            latitud = latitud,
            longitud = longitud,
            uid = uid,
            typeAlert = buttonTitle,
            idButton = buttonId,
            idAlert = idAlert,
            onSend = { alertId ->
                showAlertDialog = false
                showConfirmationDialog = true
                isButtonEnabled.value = false
                timeRemaining = 30
                onButtonStatusChange(false)
                saveButtonState(context, uid, buttonId, isButtonEnabled.value, timeRemaining)
                startTimer(context = context, uid = uid, buttonId =  buttonId, initialTime = timeRemaining, isButtonEnabledState = i
                    timeRemaining = updatedTimeRemaining
```

```
                }
            },
            onDismiss = { showAlertDialog = false }
        )
    }

    if (showConfirmationDialog) {
        showConfirmationDialog(onAccept = { showConfirmationDialog = false })
    }
}
```

- **Interacción con el usuario:** Al hacer clic en el botón de alerta, se comprueba si el botón está habilitado y si se han concedido los permisos de ubicación. Si ambos son válidos, se obtiene la ubicación actual del usuario y se muestra un diálogo para que el usuario ingrese detalles sobre la alerta.
- **Temporizador:** Si se envía una alerta, el botón se deshabilita temporalmente durante 30 segundos para evitar envíos múltiples. Este estado se guarda en `SharedPreferences` para que persista incluso si la aplicación se cierra.
- **Diálogo de alerta:** Se muestra un diálogo que permite al usuario ingresar más detalles sobre la alerta. Una vez enviada, se presenta un diálogo de confirmación que informa al usuario que la alerta ha sido enviada con éxito.

Funciones Internas

- **startTimer(...)**
  Descripción: Inicia un temporizador que deshabilita el botón durante un tiempo específico, actualizando el estado en `SharedPreferences`.

```
private fun startTimer(
    context: Context,
    uid: String,
    buttonId: String,
    initialTime: Int,
    isButtonEnabledState: MutableState<Boolean>,
    onTimeUpdate: (Int) -> Unit
) {
    val scope = CoroutineScope(Dispatchers.Main)
    var remainingTime = initialTime

    scope.launch {
        while (remainingTime > 0) {
            delay(1000L)
            remainingTime -= 1
            saveButtonState(context = context, uid = uid, buttonId = buttonId, isEnabled = false, timeRemaining = remainingTime)
            onTimeUpdate(remainingTime)
        }
        saveButtonState(context = context, uid = uid, buttonId = buttonId, isEnabled = true, timeRemaining = 0)
        isButtonEnabledState.value = true
        onTimeUpdate(0)
    }
}
```

- **saveButtonState(...)**
  Descripción: Guarda el estado del botón y el tiempo restante en `SharedPreferences`.

```
fun saveButtonState(context: Context, uid: String, buttonId: String, isEnabled: Boolean, timeRemaining: Int) {
    val sharedPreferences = context.getSharedPreferences("button_states", Context.MODE_PRIVATE)
    with(sharedPreferences.edit()) {
        putBoolean("$uid $buttonId", isEnabled)
        putInt("$uid ${buttonId}_timeRemaining", timeRemaining)
        apply()
    }
}
```

- **getButtonState(...)**
  Descripción: Recupera el estado del botón y el tiempo restante desde `SharedPreferences`.

```
fun getButtonState(context: Context, uid: String, buttonId: String): Pair<Boolean, Int> {
    val sharedPreferences = context.getSharedPreferences("button_states", Context.MODE_PRIVATE)
    val isEnabled = sharedPreferences.getBoolean("$uid $buttonId", true)
    val timeRemaining = sharedPreferences.getInt("$uid ${buttonId}_timeRemaining", 30)
    return Pair(isEnabled, timeRemaining)
}
```

- **showAlertInputDialog(...)**
  Descripción: Muestra un diálogo para que el usuario ingrese detalles sobre la alerta y maneja la lógica para enviar la alerta.

```
@RequiresApi(Build.VERSION_CODES.O)
@Composable
fun showAlertInputDialog(
    state: String,
    idAlert: String,
    latitud: Double,
    longitud: Double,
    uid: String,
    typeAlert: String,
    idButton: String,
    onSend: (String) -> Unit,
    onDismiss: () -> Unit
) {
    var alertDetails by remember { mutableStateOf("") }

    val context = LocalContext.current

    AlertDialog(
        onDismissRequest = { onDismiss() },
        title = { Text(text = "Alerta Recibida!") },
        text = {
            Column {
                Text("Latitud: $latitud, Longitud: $longitud")
                Text("¿Puedes darnos más detalles sobre la situación?")
                Spacer(modifier = Modifier.height(8.dp))
                androidx.compose.material3.OutlinedTextField(
                    value = alertDetails,
                    onValueChange = { alertDetails = it },
                    label = { Text("Detalles") },
                    modifier = Modifier.fillMaxWidth()
                )
            }
        },
        confirmButton = {
            Button(onClick = {
                handleSendAlert(
                    context =  context,
                    idAlert =  idAlert,
                    state =  state,
                    latitud =  latitud,
                    longitud =  longitud,
                    uid =  uid,
                    alertDetails =  alertDetails,
                    typeAlert =  typeAlert,
                    idButton =  idButton
                ) { alertId ->
                    onSend(alertId)
                }
            }) {
                Text("Enviar")
            }
        },
        dismissButton = {
            Button(onClick = { onDismiss() }) {
                Text("Cancelar")
            }
        }
    )
}
```

- **handleSendAlert(...)**
  Descripción: Maneja el envío de la alerta a Firestore, creando una notificación correspondiente.

```
@RequiresApi(Build.VERSION_CODES.O)
fun handleSendAlert(
    context: Context,
    idAlert: String,
    state: String,
    latitud: Double,
    longitud: Double,
    uid: String,
    alertDetails: String,
    typeAlert: String,
    idButton: String,
    onSuccess: (String) -> Unit
) {
    val fireStoreServiceAlert = FireStoreServiceAlert()
    val userFireStoreService = UserFireStoreService()
    var alert = Alert()
    alert.apply {
        this.idAlert = ""
        this.state = state
        this.latitude = latitud
        this.longitude = longitud
        this.detail = alertDetails
        this.typeAlert = typeAlert
        this.idButton = idButton
        this.userId = uid
        this.date = LocalDateTime.now().toString()
    }

    userFireStoreService.getUserbyUid(uid) { user ->
        if (user != null) {
            alert.user = user
            fireStoreServiceAlert.saveAlertInFireStore(alert, onSuccess = { alertId ->
                alert.apply {
                    this.idAlert = alertId
                }
                val db = FirebaseFirestore.getInstance()
                db.collection("alerts")
                    .document(alertId)
                    .set(alert)
                val alertService = AlertService(context)
                alertService.createNotification(
                    alertId =  alertId,
                    estado =  state,
                    message =  alertDetails,
                    typeAlert =  typeAlert,
                    latitud =  latitud.toString(),
                    longitud =  longitud.toString()
                )
                onSuccess(alertId)
            }, onFailure = {
                Log.e("DEBUG", "Error al guardar la alerta")
            })
        } else {
            Log.d("DEBUG", "No se encontró el usuario con UID: $uid")
        }
    }
}
```

- **showConfirmationDialog(...)**
  Descripción: Muestra un diálogo de confirmación tras el envío exitoso de la alerta.

```
@Composable
fun showConfirmationDialog(onAccept: () -> Unit) {
    AlertDialog(
        onDismissRequest = { },
        title = { Text(text = "Información Enviada") },
        text = { Text("Información enviada con éxito, revisa el apartado de notificaciones para tener información sobre el estado de
                "\nNOTA: No podrás volver a enviar esta misma alerta hasta que pase al menos 30 segundos, esperamos tu comprension."
        confirmButton = {
            Button(onClick = { onAccept() }) {
                Text("Aceptar")
            }
        }
    )
}
```

- **getCurrentLocation(...)**
  Descripción: Define la lógica para obtener la ubicación en tiempo real del usuario cuando le da click a un botón de alerta.

```
@SuppressLint("MissingPermission")
private fun getCurrentLocation(fusedLocationClient: FusedLocationProviderClient, onLocationResult: (Double, Double) -> Unit) {
    fusedLocationClient.lastLocation.addOnSuccessListener { location ->
        if (location != null) {
            onLocationResult(location.latitude, location.longitude)
        } else {
            onLocationResult(0.0, 0.0)
        }
    }
}
```

Clase BotonDeAlertaViewModel

La clase **BotonDeAlertaViewModel** gestiona el estado de los botones de alerta en la aplicación, utilizando `SharedPreferences` para mantener la persistencia del estado incluso después de que la aplicación se cierre. Esta clase está diseñada para trabajar en conjunto con el componente `BotonDeAlerta`.

```
class BotonDeAlertaViewModel(context: Context) : ViewModel()
```

Propiedades

- **preferences: SharedPreferences**
  Descripción: Almacena los estados de los botones de alerta de forma persistente.

```
private val preferences: SharedPreferences =
        context.getSharedPreferences("button_states", Context.MODE_PRIVATE)
```

- **_buttonStates: MutableStateFlow<Map<String, Boolean>>**
  Descripción: Flujo que mantiene el estado actual de los botones. Utiliza `loadButtonStates()` para inicializar su valor.

```
private val _buttonStates = MutableStateFlow<Map<String, Boolean>>(loadButtonStates())
```

- **buttonStates: StateFlow<Map<String, Boolean>>**
  Descripción: Proporciona una versión inmutable del estado de los botones para ser observada por las UI.

```
val buttonStates = _buttonStates.asStateFlow()
```

- **activeTimers: MutableMap<String, Boolean>**
  Descripción: Mantiene un registro de los botones que tienen un temporizador activo, para evitar que se deshabiliten múltiples veces simultáneamente.

```
private val activeTimers = mutableMapOf<String, Boolean>()
```

Funciones

disableButton(...)

```
fun disableButton(buttonId: String, duration: Long = 30000L) {
        if (activeTimers[buttonId] == true) return

        viewModelScope.launch {
            activeTimers[buttonId] = true
            updateButtonState(buttonId, false)

            delay(duration)

            updateButtonState(buttonId, true)
            activeTimers.remove(buttonId)
        }
    }
```

- **Descripción:** Deshabilita un botón de alerta durante un tiempo específico (por defecto, 30 segundos).
- **Parámetros:**
  - **buttonId: String:** Identificador del botón que se desea deshabilitar.
  - **duration: Long:** Tiempo en milisegundos durante el cual el botón permanecerá deshabilitado (opcional, por defecto es 30,000 ms).
- **Funcionamiento:** Si el botón ya está deshabilitado, no se hace nada. Si no, se inicia un temporizador y se actualiza el estado del botón a `false`. Tras el tiempo especificado, se vuelve a habilitar el botón.

updateButtonState(...)

```
public suspend fun updateButtonState(buttonId: String, isActive: Boolean) {
        withContext(Dispatchers.Main) {
            val currentState = _buttonStates.value
            val newState = currentState.toMutableMap().apply { this[buttonId] = isActive }
            _buttonStates.emit(newState)
            saveButtonStates(newState)
        }
    }
```

- **Descripción:** Actualiza el estado del botón y lo guarda en `SharedPreferences`.
- **Parámetros:**
  - **buttonId: String:** Identificador del botón a actualizar.
  - **isActive: Boolean:** Nuevo estado del botón (habilitado o deshabilitado).
- **Funcionamiento:** Se emite un nuevo estado a `_buttonStates` y se guardan los cambios en `SharedPreferences`.

loadButtonStates(): Map<String, Boolean>

```
private fun loadButtonStates(): Map<String, Boolean> {
        val buttonStates = mutableMapOf<String, Boolean>()
        val buttonIds = listOf("alertButton_0", "alertButton_1", "alertButton_2", "alertButton_3", "alertButton_4")
        for (buttonId in buttonIds) {
            buttonStates[buttonId] = preferences.getBoolean(buttonId, true)
        }
        return buttonStates
    }
```

- **Descripción:** Carga el estado de los botones de alerta desde `SharedPreferences`.
- **Retorno:** Mapa de identificadores de botones a sus respectivos estados (`true` o `false`).

saveButtonStates(states: Map<String, Boolean>)

- **Descripción:** Guarda el estado de los botones en `SharedPreferences`.
- **Parámetros:**
  - **states: Map<String, Boolean>:** Mapa de estados de botones a guardar.

Integración con BotonDeAlerta

La clase **BotonDeAlertaViewModel** es esencial para el funcionamiento del componente **BotonDeAlerta**. Permite gestionar el estado de cada botón de alerta, asegurando que se mantenga deshabilitado durante un periodo específico tras su uso y que su estado persista entre sesiones de la aplicación. El uso de `StateFlow` facilita la reactividad, permitiendo a la interfaz de usuario actualizarse automáticamente cuando el estado de los botones cambia.

Clases Alert y FireStoreServiceAlert

Clase Alert

La clase **Alert** representa una alerta que se puede enviar a través del componente `BotonDeAlerta`. Esta clase contiene toda la información necesaria para describir una alerta, incluyendo la ubicación del usuario y los detalles de la alerta.

```
class Alert {
    var state: String = ""
    var typeAlert:String = ""
    var idButton:String = ""
    var idAlert:String = ""
    var user: User = User()
    var userId:String = ""
    var latitude: Double = 0.0
    var longitude: Double = 0.0
    var date:String = ""
    var detail:String = ""
}
```

Propiedades

- **state: String**
  Descripción: Estado actual de la alerta (por ejemplo, "Pendiente", "En atención", "Resuelta").
- **typeAlert: String**
  Descripción: Tipo de alerta que se está enviando (por ejemplo, "Emergencia", "Advertencia").
- **idButton: String**
  Descripción: Identificador del botón de alerta que envió esta alerta.
- **idAlert: String**
  Descripción: Identificador único de la alerta en Firestore.
- **user: User**
  Descripción: Objeto que representa al usuario que envió la alerta. Se espera que sea una instancia de la clase `User`.
- **userId: String**
  Descripción: Identificador único del usuario que envió la alerta.
- **latitude: Double**
  Descripción: Latitud de la ubicación desde la que se envió la alerta.
- **longitude: Double**
  Descripción: Longitud de la ubicación desde la que se envió la alerta.
- **date: String**
  Descripción: Fecha en que se envió la alerta.
- **detail: String**
  Descripción: Detalles adicionales sobre la alerta proporcionados por el usuario.

Clase FireStoreServiceAlert

La clase **FireStoreServiceAlert** es responsable de manejar las interacciones con Firestore para almacenar alertas. Proporciona un método para guardar una alerta en la base de datos y manejar la respuesta.

```
class FireStoreServiceAlert {
    fun saveAlertInFireStore(alert: Alert, onSuccess: (String) -> Unit, onFailure: () -> Unit) {
        val db = FirebaseFirestore.getInstance()
        db.collection("alerts")
            .add(alert)
            .addOnSuccessListener { documentReference ->
                val alertId = documentReference.id
                onSuccess(alertId)
            }
            .addOnFailureListener {
                onFailure()
            }
    }
}
```

Funciones

saveAlertInFireStore(...)

- **Descripción:** Guarda una alerta en Firestore y gestiona el éxito o fallo de la operación.
- **Parámetros:**
  - **alert: Alert:** Instancia de la clase `Alert` que se desea guardar.
  - **onSuccess: (String) -> Unit:** Callback que se invoca si la alerta se guarda con éxito, devolviendo el ID de la alerta creada.
  - **onFailure: () -> Unit:** Callback que se invoca si hay un error al guardar la alerta.
- **Funcionamiento:** Utiliza la instancia de Firestore para agregar la alerta a la colección "alerts". En caso de éxito, devuelve el ID de la alerta a través del callback `onSuccess`. Si falla, invoca el callback `onFailure`.

Integración con BotonDeAlerta

Las clases **Alert** y **FireStoreServiceAlert** son fundamentales para el funcionamiento del componente **BotonDeAlerta**. Cuando un usuario envía una alerta a través de `BotonDeAlerta`, se crea una instancia de `Alert` que se llena con la información relevante (como estado, tipo, ubicación, etc.). Luego, `FireStoreServiceAlert` se encarga de guardar esta alerta en Firestore, permitiendo que el sistema mantenga un registro de las alertas enviadas y sus estados.