Sprawozdanie Aplikacje Mobilne Laboratorium

Kamil Grzymkowski (151908)
Jacek Młynarczyk (151747)

1. Obsługa zmiany orientacji urządzenia

Aplikacja została zaprojektowana tak, aby działała poprawnie niezależnie od zmiany orientacji urządzenia. Android domyślnie restartuje aktywności po zmianie orientacji, co może powodować utratę stanu. W aplikacji OpenSzlaki implementacja zarządzania stanem odbywa się za pomocą rememberSaveable dostępnego w jetpack Compose oraz własnego savera oraz convertera do bazy danych napisanych dla klasy Trail (zawierającej informacje o szlaku).
Saver klasy Trail:

```
val trailMutableSaver: Saver<MutableState<Trail>, List<String>> = Saver(
    save = { state ->
        listOf(
            state.value.uid.toString(),
            state.value.name,
            state.value.description,
            GlobalConverter.fromUriList(state.value.imagePaths) ?: "",
            GlobalConverter.fromMeasurementList(state.value.measurements) ?: ""
        )
    },
    restore = { list ->
        mutableStateOf(
            Trail(
                uid = list[0].toInt(),
                name = list[1],
                description = list[2],
                imagePaths = GlobalConverter.fromUriString(list[3]) ?: mutableListOf(),
                measurements = GlobalConverter.fromMeasurementString(list[4]) ?: mutableListOf()
            )
        )
    }
)
```

Converter List<Uri> → String

```kotlin
val GlobalConverter = Converters()

class Converters {

    @TypeConverter
    fun fromUriString(value: String?): MutableList<Uri>? {
        if (value == null) return null
        val listType = object : TypeToken<MutableList<String>>() {}.type
        return try {
            val stringList: MutableList<String> = Gson().fromJson(value, listType)
            stringList.map { Uri.parse(it) }.toMutableList()
        } catch (e: NullPointerException) {
            println("Error parsing JSON: $value")
            null
        }
    }

    @TypeConverter
    fun fromUriList(list: MutableList<Uri>?): String? {
        if (list == null) return null
        val stringList = list.map { it.toString() }.toMutableList()
        return Gson().toJson(stringList)
    }
```

Użycie convertera w bazie danych:

```
@Database(entities = [Trail::class], version = 17)
@TypeConverters(Converters::class)
abstract class AppDatabase : RoomDatabase() {
    abstract fun trailDao(): TrailDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase {
            if (INSTANCE == null) {
                synchronized( lock: this) {
                    INSTANCE = Room.databaseBuilder(
                        context.applicationContext,
                        AppDatabase::class.java,
                         name: "myDatabase.db"
                    ).fallbackToDestructiveMigration().build()
                }
            }
            return INSTANCE!!
        }
    }
}
```

Przykład użycia w kodzie:

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun NewTrail(dao: TrailDao, navController: NavHostController, drawerState: DrawerState) {
    val scope = rememberCoroutineScope()
    val trail = rememberSaveable(saver = trailSaver) { Trail(
        name="", description = "", imagePaths = mutableListOf(), measurements = mutableListOf()) }
    var trailName by remember { mutableStateOf( value: "") }
    var isValidName by remember { mutableStateOf( value: true) }
```

2. Zapamiętywanie wyników w bazie danych razem z datą pomiaru

Dzięki użyciu RoomDB oraz implementacji saverów, kod z wewnątrz aplikacji może bezpośrednio oddziaływać na obiektach w bazie danych, a ich format jest jasny I łatwo czytelny.

```
@Entity(tableName = "trails")
data class Trail(
    @PrimaryKey(autoGenerate = true) var uid: Int = 0,
    @ColumnInfo(name = "name") var name: String,
    @ColumnInfo(name = "description") var description: String,
    @ColumnInfo(name = "imagePaths") var imagePaths: MutableList<Uri>,
    @ColumnInfo(name = "measurements") var measurements: MutableList<Measurement>
) {
```

3. Floating Action Button (FAB) do uruchamiania aparatu

Uruchamianie aparatu odbywa się przy pomocy rememberLauncherForActivityResult()

```
    val takePhoto = rememberLauncherForActivityResult(ActivityResultContracts.TakePicture()) { succ ->
        Log.d( tag: "Photo taken?",  msg: "$succ")
        if(!succ) return@rememberLauncherForActivityResult
        trail.value.imagePaths.add(tmpPhotoUri)
        trail.value = trail.value.copy()
        showConfirm.value = true;
    }
```

Uruchomienie takePhoto.launch() z argumentem URI otrzymanym od FileProvider, powoduje, że po wykonaniu zdjęcia, znajdzie się ono w folderze plików użytkownika aplikacji.

```
val getTmpUri = {
    val imageName = "IMG_" + System.currentTimeMillis().toString() + ".jpg"
    val file = File(ctx.filesDir, imageName)
    FileProvider.getUriForFile(ctx,  authority: ctx.packageName + ".provider", file)
}
```

4. Przewijanie ekranu szczegółów z obrazkiem na pasku aplikacji

Przewijanie zostało zrealizowane przy pomocy CollapsingToolbarScaffold() dostępnego z biblioteki: *implementation*("me.onebone:toolbar-compose:2.3.5")
Composable automatycznie konsumuje nested scroll events.

```
val state = rememberCollapsingToolbarScaffoldState()
CollapsingToolbarScaffold(
    modifier = Modifier.fillMaxSize(),
    state = state,
    scrollStrategy = ScrollStrategy.EnterAlwaysCollapsed,
    toolbar = {
        if(trail.value.imagePaths.isNotEmpty())
            AsyncImage(
                model = ImageRequest.Builder(LocalContext.current)
                    .data(trail.value.imagePaths.first())
                    .build(),
                contentDescription = "${trail.value.name} thumbnail",
                modifier = Modifier
                    .sizeIn(maxHeight = LocalConfiguration.current.screenHeightDp.dp / 2)
                    .fillMaxWidth(),
                contentScale = ContentScale.Crop
            )
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .wrapContentHeight()
                .pin()
                .background(MaterialTheme.colorScheme.onBackground.copy(alpha = 0.5F))
        ){
            Column (modifier = Modifier.align(Alignment.CenterVertically)){
                IconButton(modifier = Modifier.padding(16.dp), onClick = {scope.launch {drawerState.open()}}) {
                    Icon(Icons.Filled.Menu, contentDescription = "Open Drawer")
                }
            }
```

5. Przechodzenie pomiędzy kartami za pomocą gestu przeciągnięcia

Przechodzenie pomiędzy kartami za pomocą gestu przeciągnięcia zostało zrealizowane za pomocą detectHorizontalDragGestures użytego w Modifier.pointerInput() Elementu Box, który enkapsuuje widok TrailDetails.

```
Box(
    modifier = Modifier
        .fillMaxSize()
        .pointerInput(Unit) {
            detectHorizontalDragGestures { change, dragAmount ->
                change.consume()
                if (dragAmount > 0) {
                    val nextUid =
                        trails.findBefore { it.uid == trailId }?.uid ?: trail.uid
                    navController.navigate( route: "trailDetails/$nextUid") {
                        popUpTo( route: "trailList") { inclusive = false }
                    }
                    scope.launch { drawerState.close() }
                    return@detectHorizontalDragGestures
                }
                val prevUid =
                    trails.findAfter { it.uid == trailId }?.uid ?: trail.uid
                navController.navigate( route: "trailDetails/$prevUid") {
                    popUpTo( route: "trailList") { inclusive = false }
                }
                scope.launch { drawerState.close() }
            }
        },
    content = {
        TrailDetails(dao, trail, drawerState)
    }
)
```

6. Szuflada nawigacyjna

Wykonana jako Composable enkapsuujący każdy widok dostępny w aplikacji.

```
@Composable
fun drawer(state: DrawerState, navController: NavHostController, trails: List<Trail>, inside: @Composable () -> Unit){
    val scope = rememberCoroutineScope()
    ModalNavigationDrawer(
        drawerState = state,
        drawerContent = {
            ModalDrawerSheet {
                Row(modifier = Modifier.padding(16.dp), horizontalArrangement = Arrangement.Center){
                    Button(onClick = {
                        navController.navigate( route: "trailList")
                        {scope.launch {state.close()}}
                    }) { Text( text: "Trails List")}
                    Spacer(modifier = Modifier.padding(16.dp))
                    Button(onClick = {
                        navController.navigate( route: "trailList/new")
                        {scope.launch {state.close()}}
                    }) { Text( text: "New Trail")}
                }
                // Trail details entries
                trails.forEach { trail ->
                    NavigationDrawerItem(
                        label = { Text(trail.name) },
                        selected = false,
                        onClick = {
                            navController.navigate( route: "trailDetails/${trail.uid}")
                            {scope.launch {state.close()}}
                        }
                    )
                }
            }
        }
```

Zarządzanie stanu szuflady odbywa się w dziedzinie coroutine.