Sprawozdanie

Aplikacja typu lista-szczegóły z animacją i elementami biblioteki wsparcia

Uwagi początkowe:

- Wypisane zostały tylko spełnione wymagania.
- Aplikacja została stworzona za pomocą Jetpack Compose
- Do nawigacji zostały wykorzystane własnoręcznie napisane nawigatory opierające się na interfejsach Screen (interfejs obsługiwany przez Navigator) oraz Tab (interfejs obsługiwany przez TabNavigator)
- Baza danych stworzona za pomocą Room i Dao
- Zamiast trybu tabletowego aplikacja obsługuje składany telefon - wykrywany jest stan zawiasu a nie rozmiar ekranu tak więc na tablecie będzie widok „pojedynczy”
- (Obsługa ta jest możliwa tylko za pomocą dodatkowego wątku w tle sprawdzającego zmiany stanu zawiasu – okazuje się iż obrót urządzenia jest sprawą poważniejszą niż zmiana ekranu na którym wyświetla się treść jako że nie powoduje ona ponownego tworzenia aktywności (wywołania funkcji onCreate))
- Aplikacja korzysta z jednej aktywności (zmiana ekranów następuje poprzez zmianę wyświetlanej funkcji) i jednego ViewModel
- Kod na końcu sprawozdania (można się przenieść naciskając w [Kod] [X] [Y] gdzie X – nr pola z kodem, Y – wymagania do których odnosi się kod)
- Link do repozytorium: Szlaki

Spełnione wymagania:

Podstawowe:

1. Aplikacja ma korzystać z fragmentów:

~ wykorzystanie funkcji @Composable z Jetpack Compose

2. Aplikacja powinna mieć wersję układu dla smartfonów i osobną dla tabletów
3. Aplikacja powinna działać poprawnie po zmianie orientacji urządzenia

Użyty ViewModel oraz remember/rememberSaveable z Jetpack Compose, a także po prostu klasy (przykładowo stos ekranów w nawigatorze)

I. (dwa-układy-orientacja.mp4)
II. [Kod] [1] [1-3]

4. We fragmencie szczegółów należy zagnieździć fragment dynamiczny stopera / zegara / krokomierza
5. Stoper i zegar mają wyświetlać czas z dokładnością do sekundy:
6. Stoper /zegar / krokomierz powinien działać poprawnie na smartfonach i tabletach

7. Stoper /zegar / krokomierz powinien działać poprawnie po zmianie orientacji urządzenia
8. Stoper /zegar / krokomierz powinien mieć przyciski:

- start - uruchamiający odliczanie,
- stop - wyłączający odliczanie
- przerwij - wyłączający odliczanie w dowolnym momencie

9. Możliwość zapamiętania wyniku

Umieszczono stoper

I. (stoper.mp4)
II. [Kod] [2] [4-9]

10. Na ekranie szczegółów ma się pojawić przycisk FAB (floating action button), który będzie odpowiedzialny za uruchomienie aparatu fotograficznego, którym wykonamy sobie selfie ze szlaku (w uproszczonej wersji działanie przycisku może prowadzić jedynie do wyświetlenia odpowiedniego komunikatu).

~ Dodano przycisk na ekranie listy służący do uruchamiania Eksploratora plików w celu dodania szlaków

I. (dodanie-szlaku.mp4)
II. [Kod] [3] [10]

11. W aplikacji należy zastosować motywy.

Własne kolory z biblioteki Material3

12. Każda aktywność ma mieć pasek aplikacji w postaci paska narzędzi.
13. Do aplikacji należy dodać szufladę nawigacyjną

~ Każda karta/ekran posiada wspólną nawigację (pasek nawigacji u dołu lub po lewej) oraz pasek z nazwą ekranu

I. [Kod] [4] [12-13]

14. Przechodzenie pomiędzy kartami ma się odbywać także za pomocą gestu przeciągnięcia.

I. (animacje-gesty.mp4)
II. [Kod] [5] [14]

15. Animacja ma się opierać na systemie animacji właściwości, czyli korzystać z obiektu ObjectAnimator

~ Kod w Jetpack Compose – wykorzystane AnimatedVisibility, Transition oraz animateFloat

I. (animacje-gesty.mp4)
II. [Kod] [6] [15]

Dodatkowe:

1. Kod aplikacji w Kotlinie
2. UI z wykorzystaniem Jetpack Compose zamiast XML
3. Źródło danych o szlakach inne niż tablica. Może to być na przykład baza danych lub usługa internetowa

Wykorzystane pliki gpx które można własnoręcznie dodać oraz baza danych

I. (dodanie-szlaku.mp4)
II. [Kod] [3] [3]
III. [Kod] [9] [3]

4. Dodanie do aktywności szczegółów:
    - informacji o orientacyjnym czasie przejścia poszczególnych odcinków szlaku
    - opcji wyboru stylu chodzenia (np. wolno, normalnie, szybko) i odpowiednie przeliczenie orientacyjnych czasów przejścia

Czas przejścia z plików gpx bądź z prędkości podawanej w ustawieniach

I. [Kod] [7] [4]

5. Stoper - Umieszczenie na przyciskach ikon zamiast napisów
6. Dodatkowe funkcje / dodatkowe przyciski

Możliwość ukrycia stopera

7. Zapamiętywanie wyników w bazie danych razem z datą pomiaru

I. [Kod] [2] [5-7]

8. Możliwość wglądu do zapamiętanych wyników

Ekran z wynikami

9. Wykorzystanie motywów z biblioteki wzornictwa (material design)
10. Dodanie do paska aplikacji opcji wyszukiwania szlaku zawierającego w nazwie i/lub opisie podany tekst.

Wyszukiwanie po nazwie – brak opisów

I. (wyszukiwanie.mp4)
II. [Kod] [8] [10]

11. Opracowanie własnych ikon związanych z akcją.

Strzałki do rozwijania tytułu, ikony do nawigacji, ikony stopera

12. Opracowanie własnej ikony dla aplikacji
13. Uruchomienie kilku animatorów równocześnie

AnimatedVisibility dla TabNavigator – dla każdego Tab
Transition i animateFloat na ekranie ładowania

I. (animacje-gesty.mp4)
II. [Kod] [6] [13]

Kod:

1. Układ i orientacja [Pod: 1-3]

```
object MainScreen: Screen {
    //…
    // Rozmiar ekranu/układ
    @Composable
    fun CheckForWindowSizeChanges() {
        val context = LocalContext.current
        val scope = rememberCoroutineScope()
        val windowInfoTracker = WindowInfoTracker.getOrCreate(context)
        // Use a state to hold the latest WindowLayoutInfo
        DisposableEffect(Unit) {
            val windowLayoutInfoFlow = windowInfoTracker.windowLayoutInfo(context)
            val job = scope.launch {
                windowLayoutInfoFlow.collect { newLayoutInfo ->
                    val foldingFeature =
                       newLayoutInfo.displayFeatures.filterIsInstance<FoldingFeature>()
                            .firstOrNull()
                    if (foldingFeature == null)
                        viewModel.FoldableStateUpdate(FoldableDeviceState.CLOSED)
                    else {
                        if (foldingFeature.state == FoldingFeature.State.FLAT)
                            viewModel.FoldableStateUpdate(FoldableDeviceState.OPENED)
                        else
                         viewModel.FoldableStateUpdate(FoldableDeviceState.HALF_OPENED)
                    }
                }
            }
            onDispose {
                job.cancel()
            }
        }

    }


    @Composable
    override fun Content() {
        CheckForWindowSizeChanges()
        viewModel = LocalViewModel.current

        Surface(
                modifier = Modifier.windowInsetsPadding(
                WindowInsets.navigationBars.only(WindowInsetsSides.Start +
WindowInsetsSides.End)
                ),
                color = MaterialTheme.colorScheme.background
        ) {
                val orientation = LocalConfiguration.current.orientation
                if (orientation == ORIENTATION_PORTRAIT ||
                viewModel.FoldableStateGet().collectAsState().value !=
FoldableDeviceState.CLOSED
                )
                NavigationPortraitOrOpened()
                else
                NavigationHorizontal()
```

```
        }
    }

    // Nawigacja po kartach zależna od orientacji i rozłożenia ekranu
    @OptIn(ExperimentalMaterial3Api::class)
    @Composable
    fun NavigationHorizontal() {
         // …
        // Drawer + column + topbar
                            tabNavigator.CurrentTab()

    }

    @OptIn(ExperimentalMaterial3Api::class)
    @Composable
    fun NavigationPortraitOrOpened() {
        // …
        // topbar+ bottombar
                            tabNavigator.CurrentTab()
    }


    @Composable
    override fun Content() {
        CheckForWindowSizeChanges()
        viewModel = LocalViewModel.current

        Surface(
            // modifier and colors
        ) {
            // …
            val orientation = LocalConfiguration.current.orientation
            if (orientation == ORIENTATION_PORTRAIT ||
                viewModel.FoldableStateGet().collectAsState().value !=
FoldableDeviceState.CLOSED
            )
                NavigationPortraitOrOpened()
            else
                NavigationHorizontal()
        }
    }
}

// Przykładowy ekran - obsługa rozłożonego ekranu

object ListTab: Tab {
    // …
    @Composable
    override fun Content() {
        viewModel = LocalViewModel.current

        navigator.HandleBackPress()

        if (viewModel.FoldableStateGet().collectAsState().value ==
FoldableDeviceState.CLOSED)
            SingleScreen()
        else
            DoubleScreen()
    }

    @Composable
    fun SingleScreen() {
        navigator.DisplayLast()
    }

    @Composable
    fun DoubleScreen() {
```

```
        Row {
            if (navigator.PossibleToEnableDoubleScreen())
                Box(modifier = Modifier.weight(1f)) {
                        navigator.DisplayPenultimate()
                }
            Box(
                modifier = Modifier
                    .weight(1f)
                    .fillMaxSize()
            ) {
                navigator.DisplayLast()
            }
        }
    }
}
```

2. Stoper [Pod: 4-9] [Dod: 5-7]

```
data class StopwatchFragment(val id:Long,val name:String) {
    private lateinit var timerList:MutableList<TimerEntity>
    private lateinit var localNavigator: Navigator
    private lateinit var viewModel: MainViewModel
    fun ChangeScreen() {
        localNavigator.AddScreen(TimerScreen(name,timerList))
    }

    private fun timerRun() {
        CoroutineScope(Dispatchers.Main).launch {
            while (!viewModel.timerIsStopped.value) {
                delay(1000L)
                if (viewModel.timerIsRunning.value)
                    viewModel.timerTime.value++
            }
        }
    }

    //Database
    private fun timerGet() {
        viewModel.viewModelScope.launch {
            timerList=viewModel.databaseHandling.TimerGetList(id).toMutableList()
        }
    }

    private fun timerToStringTime(constTime: Long): String {
        var time=constTime
        val hours=time/3600
        time-=(hours*3600)
        val minutes=time/60
        time-=(minutes*60)
        return "$hours:$minutes:$time"
    }

    private fun timerSave() {
        val time=TimerEntity(id,LocalDateTime.now().toString(),
            timerToStringTime(viewModel.timerTime.value))
        timerList.add(time)
        //Odświeżenie ekranu z wynikami (jeżeli są 2 i jest widoczny)
        if (viewModel.FoldableStateGet().value!= FoldableDeviceState.CLOSED &&
viewModel.timerScreenVisible.value)
            ChangeScreen()

        viewModel.TimerSave(time)
    }


    @Composable
```

```
    private fun VisibleStopwatch(timerShowTimer: MutableState<Boolean>) {
        //…
        Column(
            Modifier
                .fillMaxWidth()
                .padding(10.dp),
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Row (modifier= Modifier.fillMaxWidth(),
                horizontalArrangement = Arrangement.Center){

                Text(text =
timerToStringTime(viewModel.timerTime.collectAsState().value), fontSize = 32.sp)
            }
            Row {
                FloatingActionButton(onClick = {
                    if (!viewModel.timerIsRunning.value &&
viewModel.timerIsStopped.value) {
                        viewModel.timerIsRunning.value=true
                        viewModel.timerIsStopped.value=false
                        viewModel.timerTime.value=0L
                        timerRun()
                    }
                    else if (!viewModel.timerIsRunning.value) {
                        viewModel.timerIsRunning.value=true
                    }
                    else
                        viewModel.timerIsRunning.value=false}
                ) {
                    if (viewModel.timerIsRunning.collectAsState().value)
                        Icon(
                            painter = painterResource(R.drawable.pause),
                            contentDescription = "Pauza"
                        )
                    else
                        Icon(
                            painter = painterResource(R.drawable.start),
                            contentDescription = "Start"
                        )
                }
                FloatingActionButton(onClick = {
                    viewModel.timerIsRunning.value=false
                    viewModel.timerIsStopped.value=true
                    timerSave()
                    viewModel.timerTime.value=0L
                }) {
                    Icon(painterResource(R.drawable.stop),contentDescription = "Stop")
                }

                FloatingActionButton(onClick = { timerOpenListScreen.value = true }) {
                    Text(text = "Wyniki")
                }
                FloatingActionButton(onClick = { timerShowTimer.value = false }) {
                    Text(text = "Schowaj")
                }
            }
        }

    }

    @Composable
    fun Content() {
        viewModel= LocalViewModel.current
        localNavigator= LocalNavigator.current
        timerGet()

        val timerShowTimer = rememberSaveable { mutableStateOf(false) }
```

```
        if (timerShowTimer.value)
            VisibleStopwatch(timerShowTimer)
        else
            HiddenStopwatch(timerShowTimer)

    }

    @Composable
    private fun HiddenStopwatch(open: MutableState<Boolean>) {
        Column(
            Modifier
                .fillMaxWidth()
                .padding(10.dp),
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            FloatingActionButton(onClick = { open.value = true }) {
                Text(text = "Pokaż stoper")
            }
        }
    }
}

//Fragment umieszczony w:
data class TrailDetail(val trail:MutableState<Trail?>): Screen {
    //…
    @Composable
    override fun Content() {
            //…
            Scaffold(
                //Fragment ze stoperem
                bottomBar = { StopwatchFragment(trail.value!!.id,
trail.value!!.name).Content() }
            ) {
                //…
            }
        }
    }
}
```

3. Dodanie szlaku [Pod: 10] [Dod: 3]

```
class TrailList: Screen {
    //…
    @Composable
    override fun Content() {
        //…
        LazyColumn(
                //…
        ) {
            item { TopListBar() }
            //…
        }
    }

    @Composable
    fun TopListBar() {
        val addDialogOpened = remember { mutableStateOf(false) }
        if (addDialogOpened.value) {
            AddDialog(addDialogOpened, viewModel)
        }

        //…
            Button(
                modifier = Modifier
                    .padding(5.dp)
```

```
                    .fillMaxHeight(),
                onClick = {
                    addDialogOpened.value = true
                },
                //…
            ) {
                Icon(
                    painterResource(id = R.drawable.add),
                    "Dodaj szlak",
                )
            }
        }
    }

    //------------------------------Adding trails-------------------------------------

    @Composable
    private fun AddDialog(
        openAlertDialog: MutableState<Boolean>,
        viewModel: MainViewModel
    ) {

        val context = LocalContext.current

        val selectedFile = remember { mutableStateListOf<Uri?>(null) }

        val fileLauncher =
            rememberLauncherForActivityResult(contract =
ActivityResultContracts.GetMultipleContents()) { files ->
                selectedFile.apply {
                    clear()
                    addAll(files)
                }
                val strings = mutableListOf<String>()
                for (file in files) {

context.contentResolver.openInputStream(file)?.bufferedReader()?.readText()
                        ?.let { strings.add(it) }
                }
                viewModel.TrailAdd(strings)   //Zawiera zamianę pliku GPX na klasę
               //Wczytanie obliczenia i tym podobne
                openAlertDialog.value = false
            }

        AlertDialog(
            title = { Text(text = "Dodaj nowy szlak") },
            text = { Text(text = "Dodaj nowy szlak z pliku .gpx") },
            onDismissRequest = { openAlertDialog.value = false },
            confirmButton = {
                TextButton(
                    onClick = {
                        fileLauncher.launch("*/*")
                    }
                ) {
                    Text("Dodaj szlak")
                }

            },
            dismissButton = {
                TextButton(
                    onClick = {
                        openAlertDialog.value = false
                    }
                ) {
                    Text("Anuluj")
                }
            })
```

```
    }
}
```

4. Pasek nawigacji [Pod: 12-13]

```
object MainScreen: Screen {
    lateinit var viewModel: MainViewModel
    private val tabNavigator=TabNavigator(
        tabs = listOf(ListTab, CategoriesTab, SettingsTab),
        initialIndex =  ListTab.index,
        true
    )

    @OptIn(ExperimentalMaterial3Api::class)
    @Composable
    fun NavigationHorizontal() {
        //…
        ModalNavigationDrawer(/*…*/) {
        Row {
            NavigationRail(containerColor = MaterialTheme.colorScheme.primaryContainer)
{
                ShowNavigationDrawer(drawerState = drawerState, scope = scope)
                TabNavigationRailItem(ListTab)
                TabNavigationRailItem(CategoriesTab)
                TabNavigationRailItem(SettingsTab)
            }
            Scaffold(
                topBar = {
                    CenterAlignedTopAppBar(/*…*/)
                },
                content = {
                    Box(/*…*/) {
                        tabNavigator.CurrentTab()
                    }
                }
            )
        }
    }
    }

    @OptIn(ExperimentalMaterial3Api::class)
    @Composable
    fun NavigationPortraitOrOpened() {
        Scaffold(
            topBar = {
                    CenterAlignedTopAppBar(/*…*/)
            },
            content = {
                Box(
                    modifier = Modifier
                        .padding(it)
                ) {
                    tabNavigator.CurrentTab()
                }
            },
            bottomBar = {
                NavigationBar(/*…*/) {
                    TabNavigationItem(ListTab)
                    TabNavigationItem(CategoriesTab)
                    TabNavigationItem(SettingsTab)
                }
            }
        )
    }

    @Composable
```

```
    fun RowScope.TabNavigationItem(tab: Tab) {
        NavigationBarItem(
            selected = tabNavigator.current.index == tab.index,
            onClick = { tabNavigator.ChangeTab(tab.index) },
            icon = { Icon(painter = tab.options.icon!!, contentDescription =
tab.options.title) }
        )
    }


    @Composable
    override fun Content() {
        //…
            val orientation = LocalConfiguration.current.orientation
            if (orientation == ORIENTATION_PORTRAIT ||
                viewModel.FoldableStateGet().collectAsState().value !=
FoldableDeviceState.CLOSED
            )
                NavigationPortraitOrOpened()
            else
                NavigationHorizontal()
        }
    }
}
```

5. Obsługa gestów pomiędzy kartami [Pod: 14]

```
class TabNavigator(
    private val tabs: List<Tab>,
    initialIndex: UShort,
    private val swipeEnabled: Boolean =false
) {
    //…
    @Composable
    fun CurrentTab() {
        var offset =0f
        val viewModel= LocalViewModel.current

        val swipeModifier= Modifier
            .fillMaxSize()
            .pointerInput(
                currentIndex,
                viewModel
                    .FoldableStateGet()
                    .collectAsState()
                    .value
            ) {
                detectHorizontalDragGestures(
                    onDragEnd = {
                        if (offset > 100) {
                            if (currentIndex > 0U) {
                                directionOfTabAnimation = -1
                                currentIndex--
                            }
                        } else if (offset < -100) {
                            if (currentIndex < (tabs.size - 1).toUShort()) {
                                directionOfTabAnimation = 1
                                currentIndex++
                            }
                        }
                    }
                ) { change, dragAmount ->
                    change.consume()
                    offset += dragAmount
                }
            }
```

```
        Box(
            modifier = if (swipeEnabled) swipeModifier else Modifier.fillMaxSize()
        ) {
            //…
            tab.Content()
        }
    }

   //…
}
```

6. Animacje [Pod: 15] [Dod: 13]

```
class TabNavigator(
    private val tabs: List<Tab>,
    initialIndex: UShort,
    private val swipeEnabled: Boolean =false
) {
    //…
    @Composable
    fun CurrentTab() {
        //…
        Box(/*…*/) {
            tabs.forEach { tab ->
                AnimatedVisibility(
                    visible = tab.index == currentIndex,
                    enter = slideInHorizontally(animationSpec = tween(300)) { fullWidth
->
                        if (directionOfTabAnimation == 1) fullWidth else -fullWidth
                    } + fadeIn(),
                    exit = slideOutHorizontally(animationSpec = tween(300)) { fullWidth
->
                        if (directionOfTabAnimation == 1) -fullWidth else fullWidth
                    } + fadeOut()
                ) {
                    tab.Content()
                }
            }
        }
    }

    fun ChangeTab(newIndex: UShort) {
        directionOfTabAnimation = if (currentIndex>newIndex)
            -1 else 1
        currentIndex = newIndex
    }

//Ekran ładowania
class LoadingScreen: Screen {
    lateinit var viewModel: MainViewModel

    //…

    @Composable
    override fun Content() {
        //Wczytanie bazy danych
        //…
        Visible()
    }

    //Visible Content
    @Composable
    private fun Visible() {
        val context = LocalContext.current
        val imageBitmap = ImageBitmap.imageResource(context.resources,
```

```
R.mipmap.ic_launcher_foreground)

        val imageWidth = imageBitmap.width
        val imageHeight = imageBitmap.height

        val screenWidth = LocalContext.current.resources.displayMetrics.widthPixels
        val screenHeight = LocalContext.current.resources.displayMetrics.heightPixels

        var state = remember { mutableStateOf("start")}

        // Create a transition
        val transition = updateTransition(targetState = state.value, label =
"imageTransition")

        val leftPartOffsetX by transition.animateFloat(
            transitionSpec = { tween(durationMillis = 3000, easing =
FastOutSlowInEasing) },
            label = "leftPartOffsetX"
        ) { state ->
            if (state == "start") -imageWidth / 3f
            else screenWidth / 2f - imageWidth / 2f + 1
        }

        val centerPartOffsetY by transition.animateFloat(
            transitionSpec = { tween(durationMillis = 3000, easing =
FastOutSlowInEasing) },
            label = "centerPartOffsetY"
        ) { state ->
            if (state == "start") -imageHeight.toFloat()
            else screenHeight / 2f - imageHeight / 2f + 1
        }

        val rightPartOffsetX by transition.animateFloat(
            transitionSpec = { tween(durationMillis = 3000, easing =
FastOutSlowInEasing) },
            label = "rightPartOffsetX"
        ) { state ->
            if (state == "start") screenWidth.toFloat()
            else screenWidth / 2f + imageWidth / 6f
        }

        LaunchedEffect(Unit) {
            state.value="stop"
        }

LaunchedEffect(viewModel.TrailListUpdateGet().collectAsState().value,transition.isRunni
ng) {
            if (viewModel.TrailListUpdateGet().value && !transition.isRunning) {
                LoadingAndMainScreens.ChangeScreen();
            }
        }

        Box(
            modifier = Modifier
                .fillMaxSize()
                .background(color = MaterialTheme.colorScheme.onPrimary),
            contentAlignment = Alignment.Center
        ) {
            Column(horizontalAlignment = Alignment.CenterHorizontally) {
                Canvas(modifier = Modifier
                    .fillMaxWidth()
                    .fillMaxHeight(0.5f)) {
                    drawIntoCanvas { canvas ->
                        val paint = Paint()

                        // Rysowanie lewej części obrazu
                        canvas.drawImageRect(
```

```
                            image = imageBitmap,
                            srcOffset = IntOffset(0, 0),
                            srcSize = IntSize(imageWidth / 3, imageHeight),
                            dstOffset = IntOffset(leftPartOffsetX.toInt(), screenHeight
/ 2 - imageHeight / 2),
                            dstSize = IntSize(imageWidth / 3, imageHeight),
                            paint = paint
                        )

                        // Rysowanie środkowej części obrazu
                        canvas.drawImageRect(
                            image = imageBitmap,
                            srcOffset = IntOffset(imageWidth / 3, 0),
                            srcSize = IntSize(imageWidth / 3, imageHeight),
                            dstOffset = IntOffset(screenWidth / 2-imageWidth/6,
centerPartOffsetY.toInt()),
                            dstSize = IntSize(imageWidth / 3, imageHeight),
                            paint = paint
                        )

                        // Rysowanie prawej części obrazu
                        canvas.drawImageRect(
                            image = imageBitmap,
                            srcOffset = IntOffset(2 * imageWidth / 3, 0),
                            srcSize = IntSize(imageWidth / 3, imageHeight),
                            dstOffset = IntOffset(rightPartOffsetX.toInt(),
screenHeight / 2 - imageHeight / 2),
                            dstSize = IntSize(imageWidth / 3, imageHeight),
                            paint = paint
                        )
                    }
                }
                Box(modifier = Modifier.fillMaxSize(), contentAlignment =
Alignment.Center ) {
                    CircularProgressIndicator()
                }
            }
        }
    }
}
```

7. Czas przejścia [Dod: 4]

```
//Opcja w ustawieniach umożliwiająca ustawienie prędkości
object SettingsScreen: Screen {
    private lateinit var viewModel: MainViewModel
    //…
    private val speed = MutableStateFlow("")

    @OptIn(ExperimentalMaterial3Api::class)
    @Composable
    private fun SpeedSettings(modifier: Modifier) {
        //…
        Box(/*…*/) {
            Column(/*…*/) {
                Text(text = "Ustaw szybkość chodu")
                Row(/*…*/) {
                    Button(enabled = SettingsTab.viewModel.SpeedSelectedButtonGet()
                        .collectAsState().value != 1.toShort(),
                        onClick = { SettingsTab.viewModel.SpeedSelectedButtonSet(1) })
{
                        Text(text = "Domyślna")
                    }
                    Button(enabled = SettingsTab.viewModel.SpeedSelectedButtonGet()
                        .collectAsState().value != 2.toShort(),
```

```
                        onClick = { SettingsTab.viewModel.SpeedSelectedButtonSet(2) })
{
                        Text(text = "Wolno")
                    }
                    Button(enabled = SettingsTab.viewModel.SpeedSelectedButtonGet()
                        .collectAsState().value != 3.toShort(),
                        onClick = { SettingsTab.viewModel.SpeedSelectedButtonSet(3) })
{
                        Text(text = "Szybko")
                    }
                }
                Row(/*…*/) {
                    val vmSpeed = viewModel.SpeedGet()
                    Button(
                        modifier = Modifier.weight(1f),
                        enabled = (vmSpeed !=
speed.collectAsState().value.toFloatOrNull() &&

viewModel.SpeedSelectedButtonGet().collectAsState().value > 3
                                ),
                        onClick = { speed.value = vmSpeed.toString() }) {
                        Text(text = (if (vmSpeed == -1f) "-" else "$vmSpeed"), maxLines
= 1)
                    }

                    Button(
                        modifier = Modifier.weight(1f),
                        enabled = ((SettingsTab.viewModel.SpeedSelectedButtonGet()
                            .collectAsState().value != 4.toShort()
                                || vmSpeed !=
speed.collectAsState().value.toFloatOrNull())
                                && speed.collectAsState().value.toFloatOrNull() !=
null),
                        onClick = {
                            SettingsTab.viewModel.SpeedSelectedButtonSet(
                                4,
                                speed.value.toFloat()
                            )
                        }) {
                        Text(text = "Dokładna")
                    }
                    //…
                    BasicTextField(
                        value = speed.collectAsState().value,
                        onValueChange = { newValue ->
                            if (newValue.matches(Regex("^\\d*\\.?\\d*\$"))) {
                                speed.value = newValue
                            }
                        },
                        keyboardOptions = KeyboardOptions(keyboardType =
KeyboardType.Number),
                        //…
                    )
                }
            }
        }
    }
   //…
}

//Obliczanie czasu     (-1f jako prędkość domyślna - z bazy danych)
val time: Duration = if (viewModel.SpeedGet() == -1f) {
    try {
        Duration.between(
            trinl.timeStart, trinl.timeEnd
        )
    } catch (_: Exception) {
```

```
        Duration.ZERO
    }
} else {
    Duration.ofSeconds(((trinl.length / viewModel.SpeedGet()) * 3600).toLong())
}
```

8. Wyszukiwanie [Dod: 10]

```
class CategoryList(val length: String):Screen {
    //…

    private lateinit var viewModel: MainViewModel

    @Composable
    override fun Content() {
        viewModel = LocalViewModel.current
        LazyColumn(/*…*/) {
            item { TopListBar() }
            items(filteredTrails) { trail ->
                if (trail.name.contains(viewModel.SearchTextGet(), true))
                    TrailListItem(trail).Content()
            }
        }
    }

    @Composable
    fun TopListBar() {
        val searchText = viewModel.SearchTextGet()

        Row(/*…*/) {
            TextField(
                //…
                value = searchText,
                onValueChange = { viewModel.SearchTextChange(it) },
                leadingIcon = {
                    Icon(
                        painter = painterResource(id = R.drawable.search),
                        contentDescription = "",
                        tint = MaterialTheme.colorScheme.primary,
                    )
                },
                placeholder = { Text(text = "Wyszukaj szlaki") },
            )
        }
    }
}
```

9. Baza danych [Dod: 3]

```
//Przykładowa tabela

@Entity(tableName = "segments",
    foreignKeys = [
        ForeignKey(
            entity = TrailEntity::class,
            parentColumns = ["id"],
            childColumns = ["trailId"],
            onDelete = ForeignKey.CASCADE
        ),
    ],
    primaryKeys = ["trailId","segmentId"],
    indices = [Index(value = ["trailId","segmentId"])])
data class SegmentEntity(
    val trailId:Long,
    val segmentId:Long,
    val meanElevation: Float,
```

```
    val upElevation: Float,
    val downElevation: Float,
    val timeStart: String?,
    val timeEnd: String?,
    val length: Double
)
//Przykładowe mapowanie obiektowo relacyjne
fun List<TrailEntity>.AsDomainModel(): List<TrailInList> {
    return map {
        TrailInList(id=it.id,it.name,it.length,
            Bounds(it.maxLatitude,it.maxLongitude,it.minLatitude,it.minLongitude),
            LocalDateTime.parse(it.timeStart,DateTimeFormatter.ISO_DATE_TIME),
            LocalDateTime.parse(it.timeEnd,DateTimeFormatter.ISO_DATE_TIME),
        )
    }
}
//Przykładowy dostęp do bazy danych (Dao)
@Dao
interface TrailDao {
    //Times
    @Query("SELECT * FROM timers WHERE trailId=:id")
    suspend fun TimerGetList(id:Long): List<TimerEntity>
    //…

    //Trails
    //…

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun waypointInsertList(waypoints: List<WaypointEntity>)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun segmentInsertList(segments: List<SegmentEntity>)

    @Transaction
    @Insert
    suspend fun insertTrailInfo(
        waypoints: List<WaypointEntity>,
        segments: List<SegmentEntity>,
    ) {
        waypointInsertList(waypoints)
        segmentInsertList(segments)
    }

    @Query("DELETE FROM trails WHERE id=:id")
    suspend fun TrailDelete(id:Long)

    //…
}

@Database(entities = [TrailEntity::class,WaypointEntity::class,SegmentEntity::class,
    TimerEntity::class],
    version = 496, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    abstract val trailDao: TrailDao
}

//Przykładowa obsługa żądań
class DatabaseHandling(
    private val appDatabase: AppDatabase,
) {
    //Timers
    @WorkerThread
    suspend fun TimerGetList(id:Long):List<TimerEntity> {
        return appDatabase.trailDao.TimerGetList(id)
    }
    //…
```

```
    @WorkerThread
    suspend fun TimerDeleteAll() {
        return appDatabase.trailDao.TimerDeleteAll()
    }
    //Trails

    @WorkerThread
    suspend fun TrailGetList():List<TrailInList> {
        return appDatabase.trailDao.TrailGetList().AsDomainModel()
    }
    //…
    @WorkerThread
    suspend fun TrailDelete(
        id: Long
    ) {
        appDatabase.trailDao.TrailDelete(id)
    }
}
```