

Aplikacja mobilna typu lista-szczegóły

Paulina Guzior

02.04.2024

Spis treści

1 Wstęp	3
1.1 Opis aplikacji	3
1.2 Wykorzystane technologie	3
1.3 Przechowywanie danych	3
2 Konstrukcja aplikacji	4
2.1 Struktura projektu	4
2.2 Nawigacja	4
2.3 Pobieranie i przechowywanie danych	4
3 Implementacja	5
3.1 Układ Widoków	5
3.1.1 Fragmenty kodu	5
3.1.2 Zdjęcia z aplikacji	7
3.2 Przechowywanie danych o szlakach	10
3.2.1 Fragmenty kodu	10
3.2.2 Zdjęcia z aplikacji	11
3.3 Wybór trybu chodzenia	12
3.3.1 Fragmenty kodu	12
3.3.2 Zdjęcia z aplikacji	13
3.4 Ekran szczegółów szlaku	15
3.4.1 Zdjęcia z aplikacji	15
3.5 Stoper	16
3.5.1 Fragmenty kodu	16
3.5.2 Zdjęcia z aplikacji	17
3.6 Podział ekranu na karty	18
3.6.1 Fragmenty kodu	18
3.6.2 Zdjęcia z aplikacji	18
3.7 Lista szlaków	20
3.7.1 Fragmenty kodu	20
3.7.2 Zdjęcia z aplikacji	21
3.8 Przycisk FAB	22
3.8.1 Fragmenty kodu	22
3.8.2 Zdjęcia z aplikacji	23
3.9 Pasek narzędzi	24
3.9.1 Fragmenty kodu	24
3.9.2 Zdjęcia z aplikacji	26
3.10 Szufla nawigacyjna	29
3.10.1 Fragmenty kodu	29
3.10.2 Zdjęcia z aplikacji	29
3.11 Zdjęcia	30
3.11.1 Fragmenty kodu	30
3.11.2 Zdjęcia z aplikacji	32
3.12 Material Theme	33
3.12.1 Fragmenty kodu	33
3.13 Search field	34
3.13.1 Fragmenty kodu	34

3.13.2 Zdjęcia z aplikacji	35
3.14 Ikony akcji	36
3.14.1 Fragmenty kodu	36
3.14.2 Zdjęcia z aplikacji	37
3.15 Ikona aplikacji	38
3.15.1 Fragmenty kodu	38
3.15.2 Zdjęcia z aplikacji	38
3.16 Animacja	40
3.16.1 Fragmenty kodu	40
3.16.2 Zdjęcia z aplikacji	42
4 Podsumowanie	43

1 Wstęp

1.1 Opis aplikacji

Jest to aplikacja mobilna, która oferuje użytkownikom możliwość przeglądania różnych tras turystycznych. Użytkownicy mogą wybierać spośród różnych tras, które obejmują zarówno krótkie szlaki spacerowe, jak i długie trasy górskie. Każda trasa składa się z różnych etapów, z których każdy ma swoje unikalne ID, nazwę, opis i szacowany czas trwania.

1.2 Wykorzystane technologie

Aplikacja jest zbudowana na platformie Android, z użyciem języka Kotlin. Do zarządzania zależnościami i budowania projektu używam Gradle. Interfejs użytkownika jest tworzony z użyciem Jetpack Compose, nowoczesnej biblioteki do tworzenia interfejsów użytkownika na platformie Android.

1.3 Przechowywanie danych

Dane w aplikacji są przechowywane w Firebase Firestore, co pozwala na łatwe i efektywne zarządzanie danymi w chmurze. Dane są organizowane w kolekcje, gdzie każda kolekcja reprezentuje różne trasy. Każda trasa jest dokumentem w kolekcji, który zawiera pola takie jak ID, nazwa, opis, obraz i etapy. Etapy są przechowywane jako tablica map, gdzie każda mapa reprezentuje etap i zawiera pola takie jak ID, nazwa, opis i szacowany czas trwania. Dodatkowo zdjęcia przechowywane są w Firebase Cloud Storage w folderze `selfies`, reprezentowane jako plik z rozszerzeniem `'img'` o naziwie `'selfie(trail.id)'`.

2 Konstrukcja aplikacji

2.1 Struktura projektu

Projekt składa się z jednego modułu, ‘app’, który zawiera cały kod źródłowy, zasoby i konfiguracje specyficzne dla aplikacji. Kod źródłowy jest zorganizowany w pakietach według funkcjonalności, co ułatwia nawigację i utrzymanie kodu.

- **app/src/main**
 - **assets** - Zawiera pliki ikon.
 - **res** - Zasoby aplikacji, takie jak pliki graficzne, mipmapę z ikonami aplikacji.
 - **manifests** - Zawiera plik AndroidManifest.xml definiujący konfigurację aplikacji.
 - **java/com/example/projektlab** - Główny pakiet z kodem źródłowym aplikacji.
 - * **MainActivity.kt** - Główna aktywność aplikacji, która jest punktem wejścia do aplikacji.
 - * **Layouts.kt** - Definiuje różne układy używane w aplikacji, takie jak listy tras.
 - * **ui.theme** - Zawiera definicje motywów i stylów używanych w aplikacji.
 - **Color.kt** - Zawiera definicję poszczególnych kolorów dla danego trybu.
 - **Theme.kt** - Zawiera definicję stylu dla ProjektLab z użyciem MaterialTheme.
 - **Type.kt** - Zawiera definicję poszczególnych stylów typograficznych.
 - * **screens** - Zawiera różne ekranы aplikacji.
 - **ListScreen.kt** - Ekran wyświetlający listę tras.
 - **Main_Card.kt** - Ekran główny aplikacji.
 - **Settings_About_us.kt** - Ekran ustawień i informacji o nas.
 - **TimerScreen.kt** - Ekran z funkcją timera.
 - **TrailDetailsScreen.kt** - Ekran wyświetlający szczegóły wybranej trasy.
 - **TrailListScreen** - Ekran wyświetlający listę tras.
 - * **data** - Zawiera klasy danych i repozytorium.
 - **Stage** - Klasa reprezentująca etap trasy.
 - **Trail** - Klasa reprezentująca trasę.
 - **TrailRepository** - Repozytorium do zarządzania danymi tras.
 - * **additional** - Zawiera dodatkowe klasy i funkcje pomocnicze.
 - **Animation.kt** - Zawiera definicje animacji używanych w aplikacji.
 - **Drawer.kt** - Definiuje szufladę nawigacyjną aplikacji.
 - **Functions.kt** - Zawiera różne funkcje pomocnicze.
 - **Speed_selection.kt** - Zawiera funkcje do wyboru prędkości.
 - **Taking_picture.kt** - Zawiera funkcje do robienia zdjęć.
 - **res** - Zawiera zasoby aplikacji, takie jak pliki XML, obrazy i ciągi tekstowe.

2.2 Nawigacja

Nawigacja między różnymi ekranami jest zarządzana za pomocą biblioteki Navigation Compose. Definiuje ona różne "destynacje"(ekranы), które są dostępne w aplikacji, oraz ścieżki nawigacyjne między nimi.

2.3 Pobieranie i przechowywanie danych

Dane są pobierane z Firebase Firestore za pomocą asynchronicznych operacji. Po pobraniu, dane są przechowywane w stanie aplikacji za pomocą mechanizmu ‘remember’ dostarczanego przez Jetpack Compose. Stan aplikacji jest aktualizowany, gdy dane są pobierane, co powoduje automatyczne odświeżenie interfejsu użytkownika.

3 Implementacja

3.1 Układ Widoków

Plik MainActivity.kt pełni rolę punktu wejścia do aplikacji, gdzie inicjalizowany jest interfejs użytkownika poprzez funkcję MyScreen().

Plik Layouts.kt zawiera definicje różnych układów widoków dla różnych urządzeń. W pliku tym znajdują się implementacje PhoneLayout, MiddleLayout, oraz TabletLayout.

Ogólnie rzecz biorąc, sekcja ta odpowiada za organizację interfejsu użytkownika w zależności od rozmiaru ekranu urządzenia, zapewniając odpowiednią prezentację danych i nawigację dla użytkownika.

3.1.1 Fragmenty kodu

Opis: Plik ‘MainActivity.kt’ jest punktem wejścia do aplikacji, gdzie jest ustawiana treść widoku na podstawie funkcji ‘MyScreen()’.

Implementacja:

```
setContent {  
    MyScreen()  
}  
  
. . .  
  
@Composable  
fun MyScreen() {  
    if (configuration.screenWidthDp < 600) {  
        PhoneLayout(trails)  
    } else if (configuration.screenWidthDp < 1000) {  
        MiddleLayout(trails)  
    } else{  
        TabletLayout(trails)  
    }  
}
```

Plik ‘Layouts.kt’ definiuje różne układy widoków dla różnych urządzeń.

Opis: Układ przeznaczony dla urządzeń o szerokości ekranu mniejszej niż 600dp, zawierający listę szlaków oraz nawigację między nimi. Dla ekranów telefonów układ polega w 100% od nawigacji. Wyświetlany jest jeden ekran na raz.

Implementacja:

```
@Composable  
fun PhoneLayout(trails: List<Trail>) {  
    val navController = rememberNavController()  
  
    ProjektLabTheme {  
        NavHost(navController = navController, startDestination = "trailList") {  
            composable("trailList") {  
                ListScreen(navController, LocalContext.current, trails, 2)  
            }  
            composable("trailDetails/{trailId}") { backStackEntry ->  
                val trailId = backStackEntry.arguments?  
                    .getString("trailId")?.toIntOrNull()  
                if (trailId != null) {  
                    TrailDetailsScreen(trailId, context = LocalContext.current,  
                        navController, trails)  
                }  
            }  
            composable("settings_screen") {  
                SettingsScreen(navController)  
            }  
        }  
    }  
}
```

```
        }
    composable("about_us_screen") {
        AboutUsScreen(navController)
    }
}
}
```

Opis: Układ dla urządzeń o średniej szerokości ekranu, z dwiema częściami: lewą (z listą szlaków) i prawą (ze szczegółami szlaku). Różni się od TabletLayout tylko ilością gridcells podawaną do ListScreen.

Implementacja:

```
@Composable
fun MiddleLayout(trails: List<Trail>) {
    val navController = rememberNavController()
    var selectedTrailId by remember { mutableStateOf<Int?>(null) }

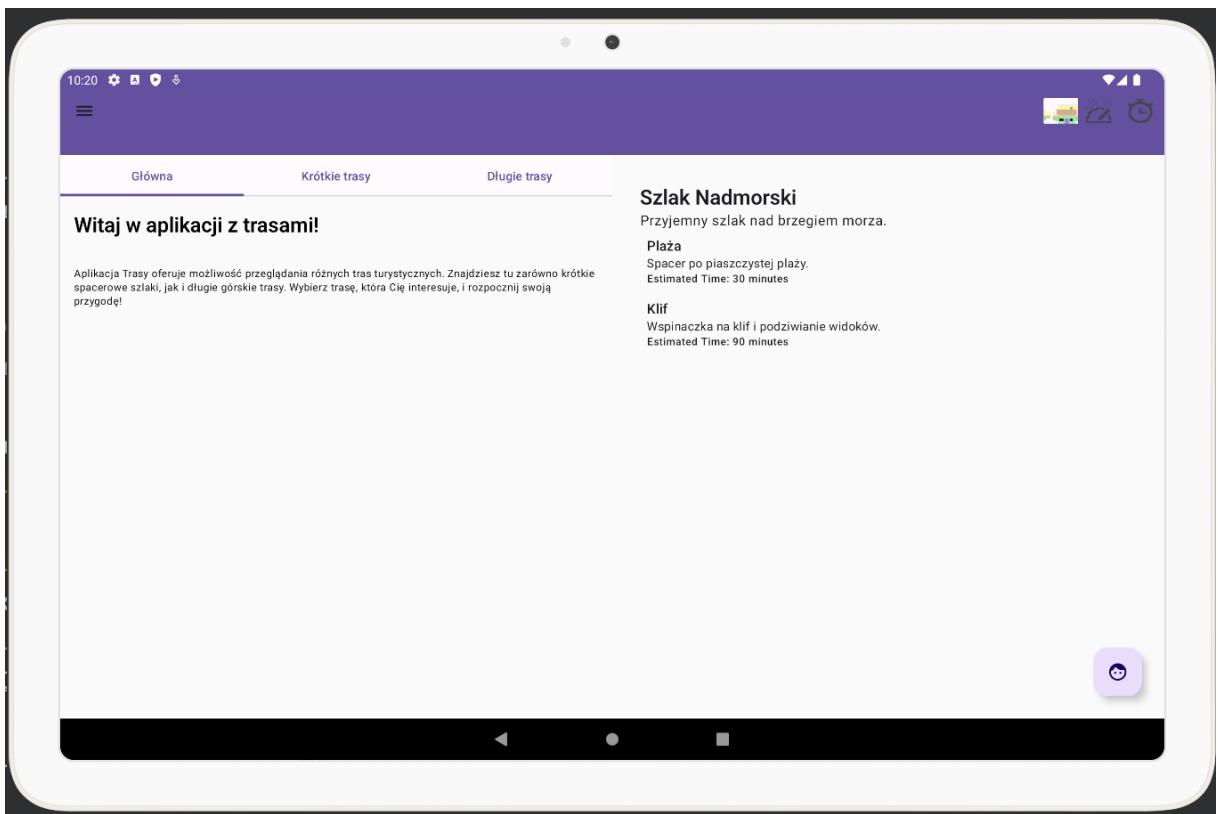
    Row {
        Box(modifier = Modifier.weight(1f)) {
            ListScreen(navController, LocalContext.current, trails, 3)
            ProjektLabTheme {
                NavHost(navController = navController, startDestination = "trailList") {
                    composable("trailList") {
                        }
                    composable("trailDetails/{trailId}") { backStackEntry ->
                        val trailId = backStackEntry.arguments?
                            .getString("trailId")?.toIntOrNull()
                        if (trailId != null) {
                            selectedTrailId = trailId
                        }
                        navController.navigate("trailList")
                    }
                    composable("settings_screen") {
                        SettingsScreen(navController)
                    }
                    composable("about_us_screen") {
                        AboutUsScreen(navController)
                    }
                }
            }
        }
    }

    Box(modifier = Modifier.weight(1f)) {
        if (selectedTrailId != null) {
            TrailDetailsScreen(selectedTrailId!!, context = LocalContext.current,
                navController, trails)
        }
        else{
            TopAppBar(
                title = { Text("") },
                actions = {
                },
                colors = TopAppBarDefaults.topAppBarColors(
                    containerColor = MaterialTheme.colorScheme.primary
                ),
                modifier = Modifier
                    .height(75.dp)
            )
        }
    }
}
```

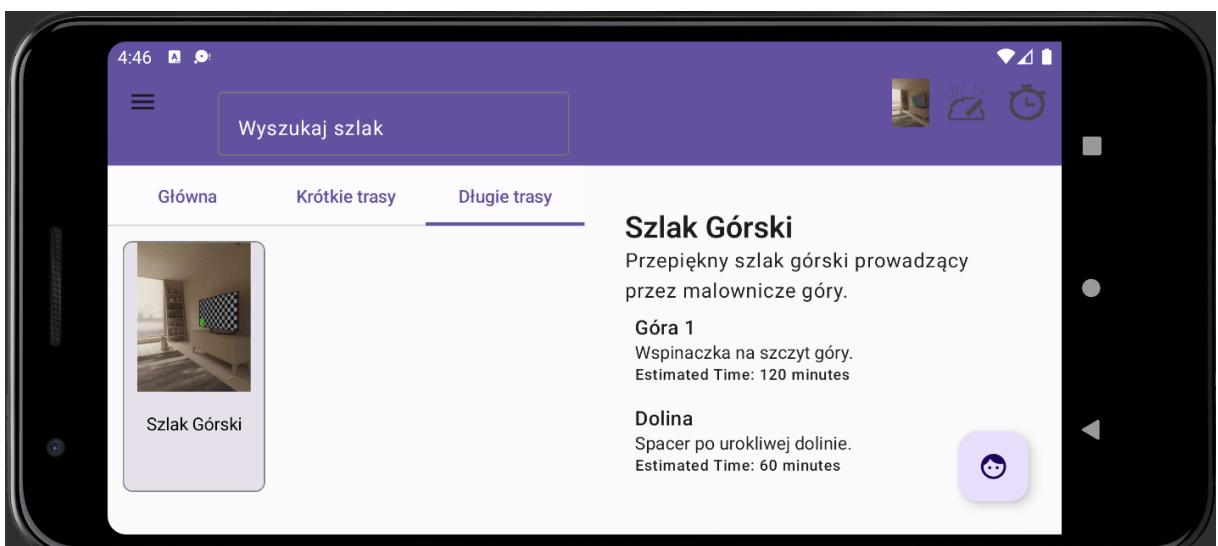
```
        }  
    }
```

3.1.2 Zdjęcia z aplikacji

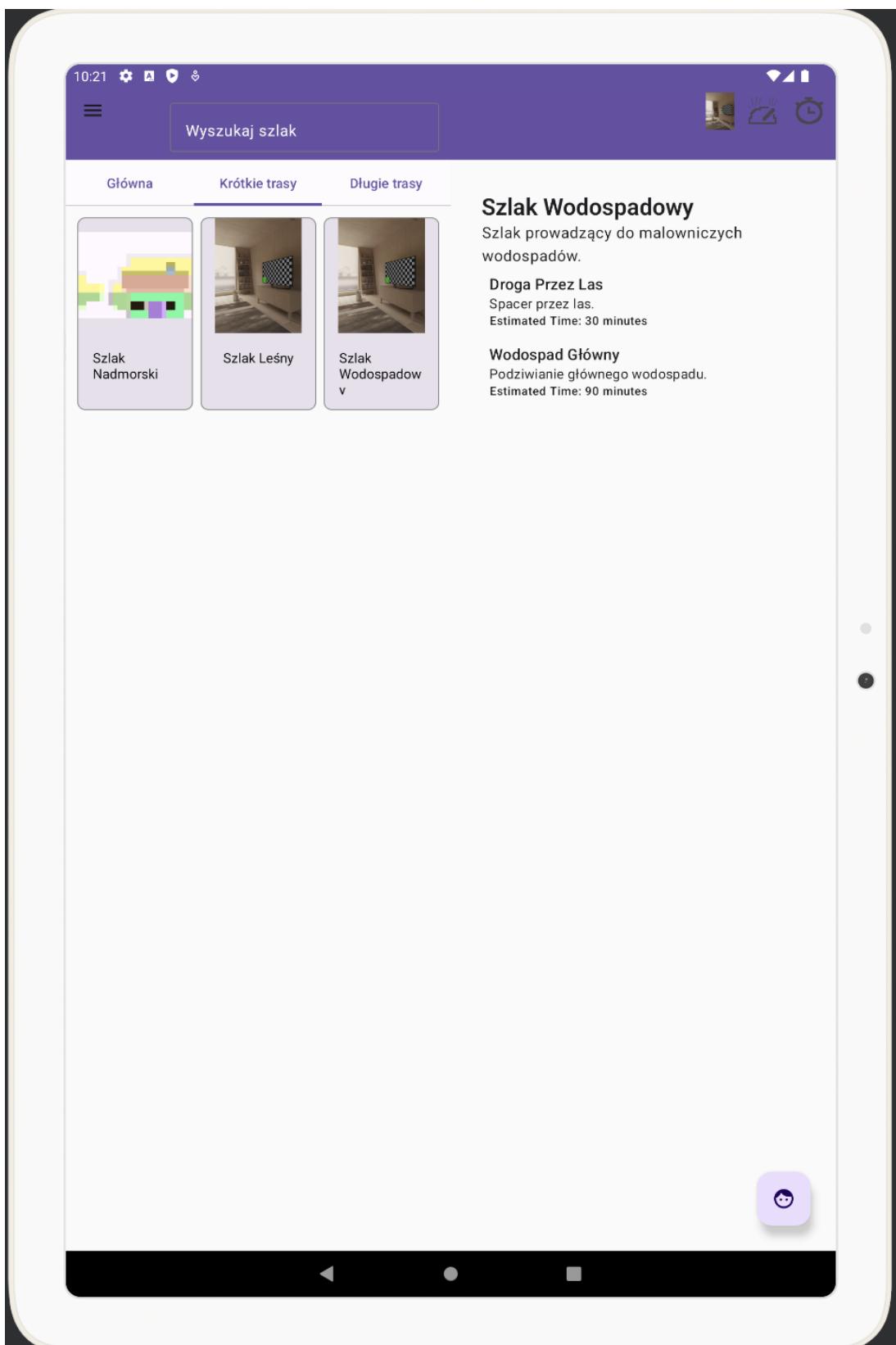
- Układ dla tabletu



- Układ dla telefonu obróconego



- Układ dla tabletu obróconego



- **Układ dla telefonu**



3.2 Przechowywanie danych o szlakach

Dane o szlakach są przechowywane w chmurze za pomocą Firebase Firestore. Każdy szlak jest reprezentowany jako dokument w kolekcji "trails". Dokument szlaku zawiera pola takie jak 'id', 'name', 'description', 'image' oraz 'stages'.

Pole 'stages' jest listą etapów szlaku. Każdy etap jest reprezentowany jako mapa z polami 'id', 'name', 'description' i 'estimatedTime'.

Do pobierania danych o szlakach służy klasa ['TrailRepository']. Metoda 'getTrails' tej klasy pobiera wszystkie szlaki z Firestore i zwraca je jako listę obiektów klasy ['Trail'].

Wszystkie dane są wczytywane do listy trails na początku ładowania aplikacji, następnie dane są przesyłane do kolejnych widoków.

3.2.1 Fragmenty kodu

```
@Composable
fun MyScreen() {
    var trails by remember { mutableStateOf<List<Trail>>(emptyList()) }
    var trailsLoaded by remember { mutableStateOf(false) }

    LaunchedEffect(Unit) {
        TrailRepository.getTrails(
            onSuccess = { fetchedTrails ->
                trails = fetchedTrails
                println("Pomyślnie pobrały szlaki do stage: $trails")
                trailsLoaded = true
            },
            onError = { exception ->
                println("Błąd podczas pobierania szlaków: $exception")
                trailsLoaded = false
            }
        )
    }
}
```

W pliku MainActivity.kt zostają wczytane dane i zapisane do listy trails.

```
data class Trail(
    val id: Int,
    val name: String,
    val description: String,
    var image: String,
    val stages: List<Stage>
)

data class Stage(
    val id: Int,
    val name: String,
    val description: String,
    val estimatedTime: Int
)
```

Definicje data class przetrzymujące obiekty.

```
object TrailRepository {
    private val db = Firebase.firestore

    fun getTrails(onSuccess: (List<Trail>) -> Unit, onError: (Exception) -> Unit) {
```

```

        db.collection("trails")
            .get()
            .addOnSuccessListener { result ->
                val trailsList = mutableListOf<Trail>()
                for (document in result) {
                    val trail = document.toObject(Trail::class.java)
                    trailsList.add(trail)
                }
                onSuccess(trailsList)
            }
            .addOnFailureListener { exception ->
                println("Error getting documents: $exception")
            }
        }
    }
}

```

Kod przedstawia obiekt ‘TrailRepository’, który korzysta z bazy danych Firebase Firestore do pobierania danych o szlakach. Funkcja ‘getTrails’ wykonuje zapytanie do kolekcji "trails" i zwraca listę szlaków w przypadku sukcesu lub obsługuje błąd w przypadku jego wystąpienia.

3.2.2 Zdjęcia z aplikacji

Widok Firestore w Firebase console

(default)	trails	nm48hlvz5NzfI80n4hG4
+ Start collection	+ Add document	+ Start collection
times	CYkqe6nbEQinnL8nq82C	+ Add field
trails >	KAxEzNLlylzhP6pZURio	description: "Szlak prowadzący do malowniczych wodospadów."
	ek1V2mDD20S7b1E0w34Q	id: 4
	nm48hlvz5NzfI80n4hG4 >	image: "obraz.jpg"
		name: "Szlak Wodospadowy"
		+ stages
		0
		description: "Spacer przez las."
		estimatedTime: 30
		id: 1
		name: "Droga Przez Las"
		+ 1
		description: "Podziwianie głównego wodospadu."
		estimatedTime: 90

3.3 Wybór trybu chodzenia

Aplikacja umożliwia użytkownikowi wybór trybu chodzenia. Użytkownik może wybrać między trybem normalnym, szybkim chodzeniem, wolnym chodzeniem. Wybór trybu wpływa na szacowany czas trwania trasy, który jest wyświetlany na ekranie szczegółów trasy.

Wybór trybu chodzenia jest realizowany za pomocą funkcji ‘selectSpeed’ z pliku [‘Speed_selection.kt’]. Ta funkcja wyświetla dialog, w którym użytkownik może wybrać tryb chodzenia. Wybrany tryb jest następnie zapisywany w pamięci lokalnej urządzenia i używany do obliczania szacowanego czasu trwania trasy.

3.3.1 Fragmenty kodu

```
enum class Speed {
    SLOW,
    NORMAL,
    FAST
}

@Composable
fun SpeedSelection(selectedSpeed: Speed, onSpeedSelected: (Speed) -> Unit) {
    val speeds = Speed.values()

    Row {
        speeds.forEach { speed ->
            Text(
                text = speed.name,
                modifier = Modifier
                    .padding(vertical = 4.dp)
                    .clickable { onSpeedSelected(speed) },
                style = MaterialTheme.typography.bodyMedium,
                color = if (speed == selectedSpeed) {
                    MaterialTheme.colorScheme.primary
                } else {
                    MaterialTheme.colorScheme.onSurface.copy(alpha = 0.54f)
                }
            )
            Spacer(modifier = Modifier.width(8.dp))
        }
    }
}
```

Fragment kodu definiuje komponent Composable ‘SpeedSelection’, który pozwala na wybór prędkości. Komponent ten wyświetla dostępne prędkości jako teksty, a po ich kliknięciu wywołuje funkcję zwrotną ‘onSpeedSelected’, przekazując wybraną prędkość. Aktualnie wybrana prędkość jest podświetlana, a pozostałe są wyświetlane z innym kolorem tekstu.

```
@Composable
fun TrailDetailsScreen(trailId: Int, context: Context,
    navController: NavController, trails: List<Trail>) {
    .
    .
    if (showSpeedSelection) {
        Text(
            text = "Choose your way of walking: ",
            style = MaterialTheme.typography.bodyMedium,
            color = MaterialTheme.colorScheme.onSurface
        )
        SpeedSelection(selectedSpeed) { speed ->
            selectedSpeed = speed
        }
    }
}
```

```

    }

@Composable
fun StageItem(stage: Stage, speed: Speed) {
    Column(modifier = Modifier.padding(8.dp)) {

        ...

        val estimatedTime = when(speed) {
            Speed.SLOW -> stage.estimatedTime * 4 / 3
            Speed.NORMAL -> stage.estimatedTime
            Speed.FAST -> stage.estimatedTime * 3 / 4
        }
        Text(
            text = "Estimated Time: $estimatedTime minutes",
            style = MaterialTheme.typography.labelMedium,
            color = MaterialTheme.colorScheme.onSurface
        )
    }
}

```

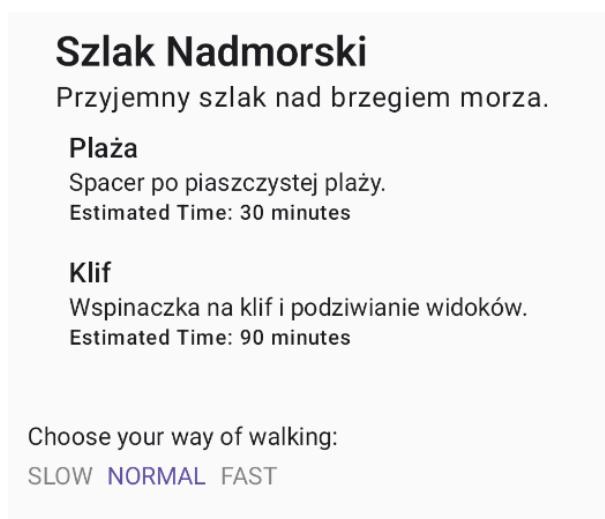
Te fragmenty dotyczą pliku 'TrailDetailsScreen.kt'. Tutaj znajdują się dwa komponenty Composable. Pierwszy z nich, 'TrailDetailsScreen', umożliwia użytkownikowi wybór prędkości chodzenia poprzez wyświetlenie odpowiedniego interfejsu użytkownika. Jeśli flaga 'showSpeedSelection' jest ustawiona na 'true', pojawia się tekst zachęcający do wyboru prędkości, a następnie użytkownik może dokonać wyboru za pomocą komponentu 'SpeedSelection'.

Drugi komponent, 'StageItem', jest odpowiedzialny za wyświetlanie szczegółów dotyczących poszczególnych etapów szlaku. W zależności od wybranej prędkości, obliczany jest przybliżony czas przejścia etapu, który jest następnie wyświetlany obok informacji o danym etapie.

Te dwa komponenty wspólnie odpowiadają za prezentację danych dotyczących szlaku oraz interakcję użytkownika związana z wyborem prędkości chodzenia.

3.3.2 Zdjęcia z aplikacji

- Tryb normalny chodzenia



- Tryb szybki chodzenia

Szlak Nadmorski

Przyjemny szlak nad brzegiem morza.

Plaża

Spacer po piaszczystej plaży.

Estimated Time: 22 minutes

Klif

Wspinaczka na klif i podziwianie widoków.

Estimated Time: 67 minutes

Choose your way of walking:

SLOW NORMAL **FAST**

3.4 Ekran szczegółów szlaku

Ekran szczegółów szlaku, zaimplementowany w pliku [‘TrailDetailsScreen.kt’], służy do wyświetlania szczegółowych informacji o wybranym szlaku.

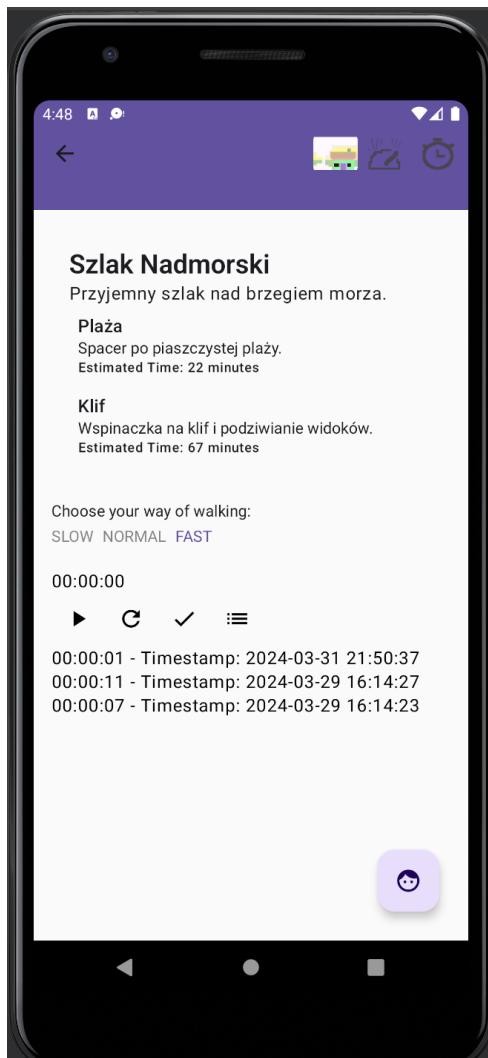
Na tym ekranie wyświetlane są takie informacje jak nazwa szlaku, opis, obraz ilustrujący szlak oraz lista etapów szlaku. Każdy etap jest wyświetlany jako osobna karta z nazwą, opisem i szacowanym czasem trwania.

Ekran szczegółów szlaku jest dostępny po kliknięciu na wybrany szlak na liście szlaków. Po wybraniu szlaku, aplikacja przechodzi do ekranu szczegółów szlaku, gdzie użytkownik może zapoznać się z pełnymi informacjami o szlaku.

Na tym widoku wyświetlona zostaje nazwa szlaku wraz ze swoim opisem, a także poszczególne etapy ze swoimi opisami oraz estymowanym czasem przejścia. Dodatkowo jest możliwość włączenia widoku stopera oraz wyboru trybu chodzenia.

3.4.1 Zdjęcia z aplikacji

Widok szczegółów szlaku



3.5 Stoper

Stoper jest zaimplementowany w pliku ['TimerScreen.kt']. Użytkownik może uruchomić, zatrzymać i zresetować stoper. Czas jest wyświetlany w formacie HH:MM:SS.

3.5.1 Fragmenty kodu

Główna funkcja stopera, 'startTimer', jest funkcją korutyny, która zwiększa wartość czasu co sekundę i aktualizuje interfejs użytkownika za pomocą funkcji 'onTimeChanged'.

```
fun startTimer(startTime: Int, onTimeChanged: (Int) -> Unit): Job {
    return CoroutineScope(Dispatchers.Main).launch {
        var time = startTime
        while (true) {
            onTimeChanged(time)
            time++
            delay(1000L)
        }
    }
}
```

Funkcja 'formatTimestamp' jest używana do formatowania znacznika czasu na czytelny format daty i czasu.

```
fun formatTimestamp(timestamp: Long): String {
    val date = Date(timestamp)
    val formatter = SimpleDateFormat("yyyy-MM-dd HH:mm:ss", Locale.getDefault())
    return formatter.format(date)
}
```

Na ekranie stopera, użytkownik może również zobaczyć listę zapisanych czasów. Lista ta jest sortowana w kolejności od najnowszego do najstarszego czasu. Dane zapisywane oraz pobierane są w Firebase firestore z kolekcji times.

```
fun saveTimeToDatabase(time: Int) {
    val timestamp = System.currentTimeMillis()
    val docData = hashMapOf(
        "time" to time,
        "timestamp" to timestamp
    )

    db.collection("times")
        .add(docData)
        .addOnSuccessListener { documentReference ->
            println("DocumentSnapshot added with ID: ${documentReference.id}")
        }
        .addOnFailureListener { e ->
            println("Error adding document: $e")
        }
}

fun getAllTimes() {
    db.collection("times")
        .get()
        .addOnSuccessListener { result ->
            times = result.documents.mapNotNull { document ->
                val time = document.getLong("time")?.toInt()
                val timestamp = document.getLong("timestamp")
                if (time != null && timestamp != null) {
                    TimeEntry(time, timestamp)
                } else {

```

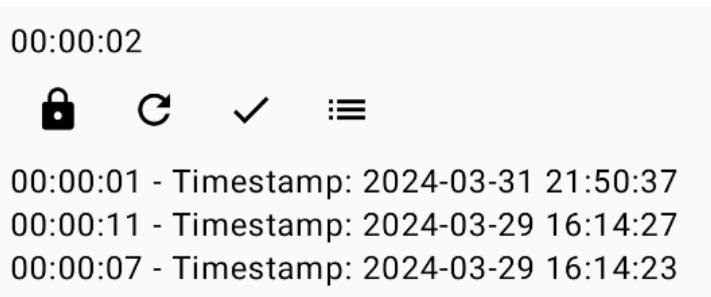
```

        null
    }
}
}
.addOnFailureListener { exception ->
    println("Error getting documents: $exception")
}
}

```

3.5.2 Zdjęcia z aplikacji

- Widok stopera



- Przechowywanie danych

Collection	Document ID	Fields
+ Start collection	+ Add document	+ Start collection
times	aKCXCnGOZrkbtTqJ2kO	+ Add field
trails	cAafsvYtuN31ZMCv0ghmY	time: 11
	t2iTriaG7Dpl5cr8Nl	timestamp: 1711725267933

3.6 Podział ekranu na karty

Ekran aplikacji jest podzielony na karty za pomocą komponentu ‘HorizontalPager‘ z biblioteki Accompanist. Każda karta reprezentuje inny ekran w aplikacji: główny, listę krótkich tras i listę długich tras. Użytkownik może przełączać się między kartami, przesuwając ekran w lewo lub w prawo.

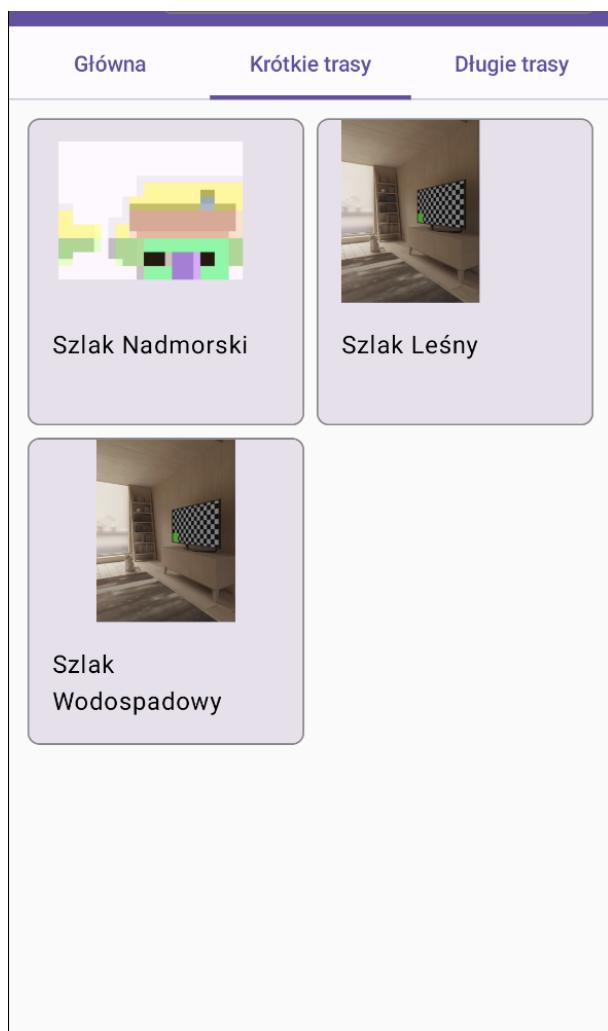
3.6.1 Fragmenty kodu

```
HorizontalPager(count = titles.size, state = pagerState) { page ->
    Column(Modifier.fillMaxHeight()) {
        when (page) {
            0 -> MainCard()
            1 -> TrailListScreen(navController, context, searchText,
                isShortRoute = true, trails, gridcells)
            2 -> TrailListScreen(navController, context, searchText,
                isShortRoute = false, trails, gridcells)
        }
    }
}
```

W powyższym fragmencie kodu, ‘HorizontalPager‘ tworzy kontener dla kart. Parametr ‘count‘ określa liczbę kart, a ‘state‘ przechowuje stan pagera, który kontroluje, która karta jest aktualnie wyświetlana. W bloku ‘when‘, dla każdej karty jest wywoływana odpowiednia funkcja Composable, która tworzy zawartość karty.

3.6.2 Zdjęcia z aplikacji

- **Widok z kartami**



- Przechodzenie między kartami gestem przeciągnięcia

Główna Krótkie trasy Długie trasy

trasami!

Przeglądania różnych tras o krótkie spacerowe. Wybierz trasę, która Cię zainteresuje!

Szlak Nadmorski

Szlak Wodospadowy

3.7 Lista szlaków

Lista szlaków jest wyświetlane na ekranie ‘TrailListScreen’. Szlaki były pobrane z bazy danych Firestore za pomocą klasy ‘TrailRepository’ i teraz wyświetlane zostają jako lista kart. Każda karta zawiera nazwę szlaku oraz obraz.

3.7.1 Fragmenty kodu

```
@Composable
fun TrailList(trails: List<Trail>, navController: NavController, gridcells: Int) {
    LazyVerticalGrid(
        columns = GridCells.Fixed(gridcells),
        modifier = Modifier.padding(8.dp)
    ) {
        items(trails.size) { index ->
            val trail = trails[index]
            var capturedImageUri by remember { mutableStateOf<Uri?>(null) }

            LaunchedEffect(trail) {
                getImage(trail) { uri ->
                    capturedImageUri = uri
                }
            }

            LaunchedEffect(trails) {
                getImage(trail) { uri ->
                    capturedImageUri = uri
                }
            }

            if (GlobalVariables.refreshImage && GlobalVariables.trailId == trail.id) {
                getImage(trail) { uri ->
                    capturedImageUri = uri
                    GlobalVariables.refreshImage = false
                    GlobalVariables.trailId = 0
                }
            }
        }

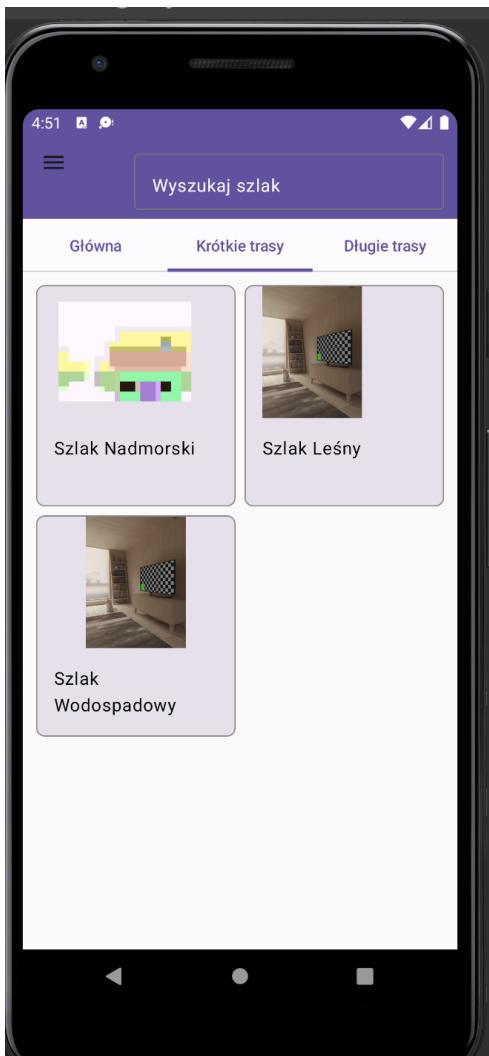
        Card(
            modifier = Modifier
                .padding(4.dp)
                .height(200.dp)
                .clickable {
                    navController.navigate("trailDetails/${trail.id}")
                },
            shape = RoundedCornerShape(8.dp),
            border = BorderStroke(1.dp, Color.Gray)
        ) {
            Column(horizontalAlignment = Alignment.CenterHorizontally,
                verticalArrangement = Arrangement.Center) {
                capturedImageUri?.let { uri ->
                    Image(
                        painter = rememberImagePainter(uri),
                        contentDescription = null,
                        modifier = Modifier
                            .size(120.dp)
                            .clip(shape = RoundedCornerShape(8.dp))
                    )
                }
            }
            Text(
```

```
        text = trail.name,  
        modifier = Modifier.padding(16.dp),  
        color = Color.Black  
    )  
}  
}  
}  
}  
}
```

Ten fragment kodu definiuje komponent Composable 'TrailList', który jest odpowiedzialny za wyświetlanie listy szlaków. Komponent ten korzysta z 'LazyVerticalGrid' zgodnie z danymi wejściowymi, aby zoptymalizować wydajność przewijania w przypadku dużej liczby elementów. Każdy szlak jest reprezentowany przez kartę ('Card'), która zawiera obrazek szlaku oraz jego nazwę. Po kliknięciu na kartę, użytkownik jest przenoszony do ekranu szczegółów szlaku. Dodatkowo, obrazek szlaku jest pobierany asynchronicznie za pomocą funkcji 'getImage', a także istnieje możliwość odświeżenia obrazka, jeśli zostanie ustawiona odpowiednia flaga.

3.7.2 Zdjęcia z aplikacji

Widok listy szlaków



3.8 Przycisk FAB

Przycisk FAB (Floating Action Button) jest używany w aplikacji do wykonywania głównej akcji na ekranie. Na przykład, na ekranie listy szlaków, przycisk FAB służy do wykonania zdjęcia.

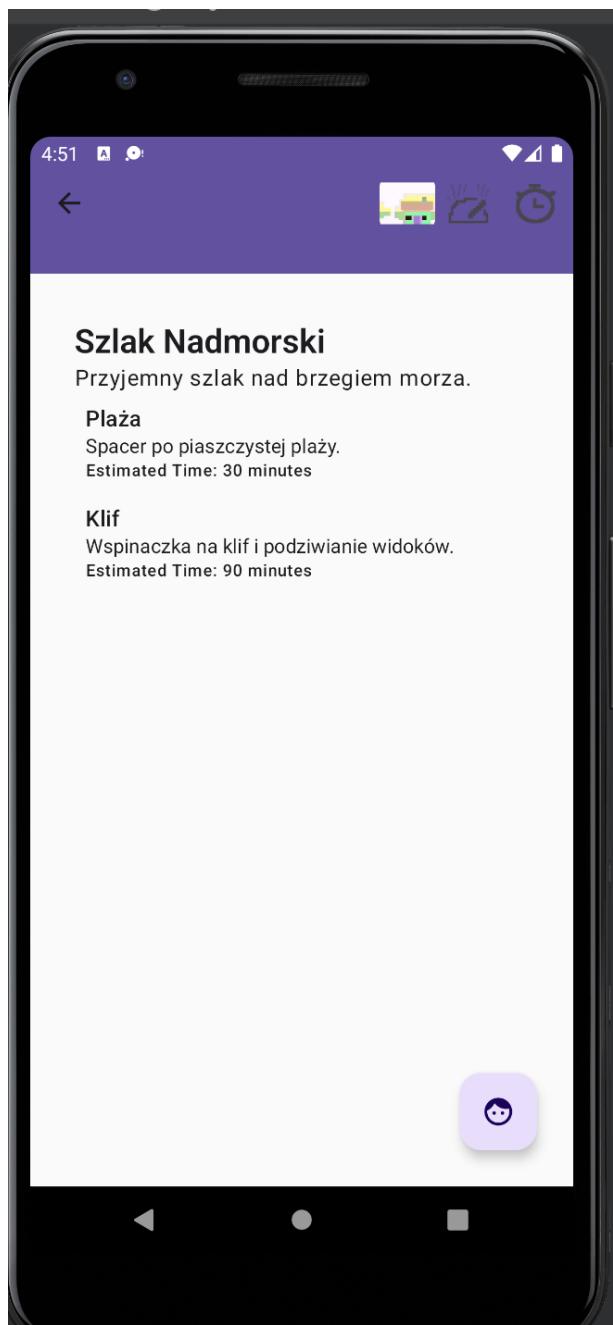
3.8.1 Fragmenty kodu

```
Box(  
    Modifier  
        .fillMaxSize()  
        .padding(10.dp)  
    ) {  
    FloatingActionButton(  
        onClick = {  
            val permissionCheckResult =  
                ContextCompat.checkSelfPermission(context,  
                    Manifest.permission.CAMERA)  
  
            if (permissionCheckResult == PackageManager.PERMISSION_GRANTED) {  
                cameraLauncher.launch(uri)  
            } else {  
                permissionLauncher.launch(Manifest.permission.CAMERA)  
            }  
        },  
        modifier = Modifier  
            .padding(16.dp)  
            .size(56.dp)  
            .align(Alignment.BottomEnd)  
    ) {  
        Icon(Icons.Filled.Face, contentDescription = "Take a selfie")  
    }  
}
```

Powyższy fragment kodu zawiera się w pliku 'Taking_picture.kt'. Przycisk FloatingActionButton wraz z ikoną, który służy do wywołania kamery w celu zrobienia selfie. Po kliknięciu na przycisk, sprawdzane są uprawnienia do korzystania z kamery. Jeśli uprawnienia są udzielone, wywoływane jest uruchomienie aparatu. W przeciwnym razie użytkownik jest poproszony o udzielenie uprawnień. Przycisk jest umieszczony w dolnym prawym rogu ekranu.

3.8.2 Zdjęcia z aplikacji

Widok szczegółów szlaku z przyciskiem FAB



3.9 Pasek narzędzi

Pasek narzędzi, zwany również TopBar, jest używany w aplikacji do wyświetlania ikon nawigacyjnych oraz różnych funkcji. W zależności od ekranu, pasek narzędzi może zawierać różne elementy, takie jak przycisk menu lub pole wyszukiwania.

3.9.1 Fragmenty kodu

```
@Composable
fun MainCardTopBar(onMenuClick: () -> Unit) {
    TopAppBar(
        title = { Text("") },
        navigationIcon = {
            IconButton(onClick = onMenuClick) {
                Icon(Icons.Default.Menu,
                    contentDescription = "Open navigation drawer")
            }
        },
        colors = TopAppBarDefaults.topAppBarColors(
            containerColor = MaterialTheme.colorScheme.primary
        ),
        modifier = Modifier
            .height(75.dp)
    )
}
```

W powyższym fragmencie kodu, ‘MainCardTopBar’ jest funkcją Composable, która tworzy pasek narzędzi dla ekranu ‘MainCard’. Pasek narzędzi zawiera ikonę menu, która otwiera szufladę nawigacyjną po kliknięciu.

```
@Composable
fun TrailListTopBar(searchText: String, onSearchTextChanged: (String) -> Unit, onMenuClick: () -> Unit) {
    TopAppBar(
        title = { Text("") },
        navigationIcon = {
            IconButton(onClick = onMenuClick) {
                Icon(Icons.Default.Menu,
                    contentDescription = "Open navigation drawer")
            }
        },
        actions = {
            SearchTextField(searchText, onSearchTextChanged)
        },
        colors = TopAppBarDefaults.topAppBarColors(
            containerColor = MaterialTheme.colorScheme.primary
        ),
        modifier = Modifier
            .height(75.dp)
    )
}
```

W powyższym fragmencie kodu, ‘TrailListTopBar’ jest funkcją Composable, która tworzy pasek narzędzi dla ekranu listy szlaków. Pasek narzędzi zawiera pole wyszukiwania, które pozwala użytkownikowi na filtrowanie listy szlaków oraz przejście do menu szuflady nawigacyjnej.

```
TopAppBar(
    title = { Text("") },
    navigationIcon = {
        if (LocalConfiguration.current.screenWidthDp < 600) {
            IconButton(onClick = { navController.navigateUp() })
        }
    }
)
```

```

        Icon(Icons.Filled.ArrowBack, contentDescription = "Back")
    }
}
),
actions = {
    capturedImageUri?.let { uri ->
        Image(
            painter = rememberImagePainter(uri),
            contentDescription = null,
            modifier = Modifier
                .size(40.dp)
                .clip(shape = RoundedCornerShape(8.dp))
        )
    }
    IconButton(onClick = { showSpeedSelection = !showSpeedSelection }) {
        val speedIcon = loadImageFromAssets(context, "icon_speed.png")
        speedIcon?.let { icon ->
            Icon(
                painter = BitmapPainter(icon),
                contentDescription = "Speed",
                modifier = Modifier.size(30.dp)
            )
        }
    }
    IconButton(onClick = { showTimer = !showTimer }) {
        val timerIcon = loadImageFromAssets(context, "icon_timer.png")
        timerIcon?.let { icon ->
            Icon(
                painter = BitmapPainter(icon),
                contentDescription = "Timer",
                modifier = Modifier.size(30.dp)
            )
        }
    }
},
colors = TopAppBarDefaults.topAppBarColors(
    containerColor = MaterialTheme.colorScheme.primary
),
modifier = Modifier
    .height(75.dp)
)
)

```

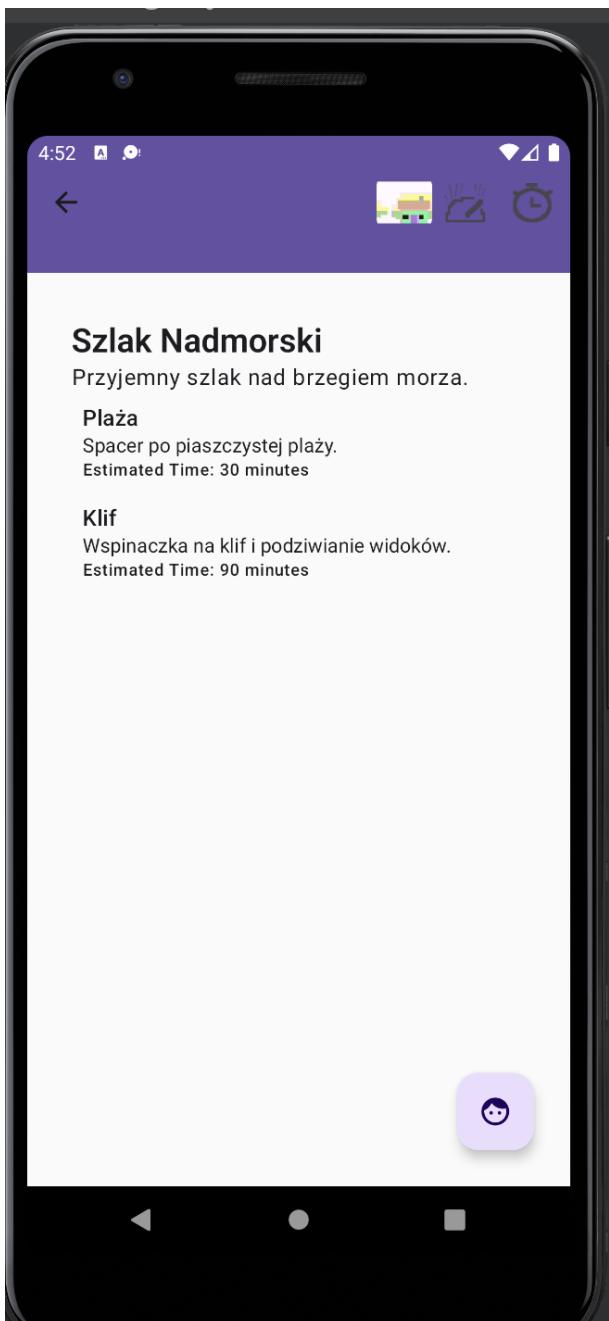
Ten fragment kodu definiuje pasek aplikacji (TopAppBar) w pliku TrailDetailsScreen.kt, zawierający różne elementy nawigacyjne oraz ikony akcji dla ekranu szczegółów szlaku. Elementy te obejmują:

- Przycisk nawigacji z ikoną strzałki wstecz, który umożliwia powrót do poprzedniego widoku. Ten przycisk jest widoczny tylko wtedy, gdy szerokość ekranu jest mniejsza niż 600 dp.
- Ikona reprezentująca zrobione zdjęcie, jeśli takie istnieje. Ikona ta jest wyświetlana w prawym górnym rogu.
- Ikona akcji do pokazania lub ukrycia wyboru prędkości chodzenia.
- Ikona akcji do pokazania lub ukrycia stopera.

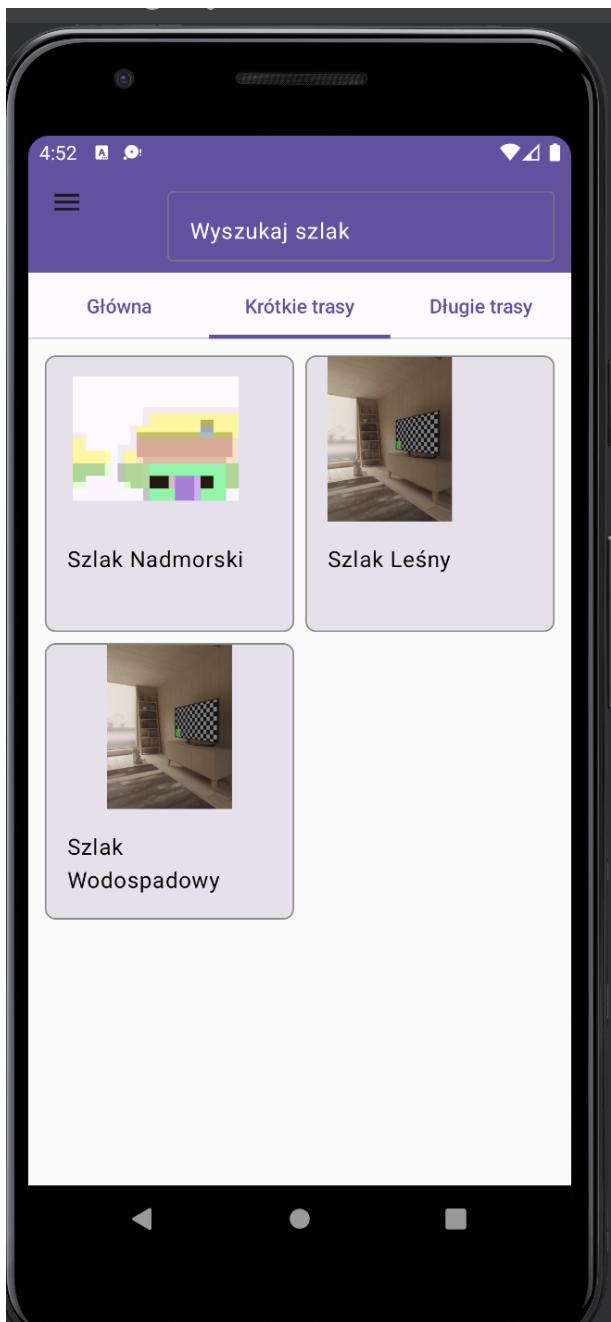
Pasek aplikacji ma również dostosowane kolory zgodnie z motywem aplikacji i jest ustawiony na wysokość 75 dp.

3.9.2 Zdjęcia z aplikacji

- Widok paska narzędzi na ekranie głównym



- Widok paska narzędzi na ekranie listy



- Widok paska narzędzi na ekranie szczegółów



3.10 Szuflada nawigacyjna

Szuflada nawigacyjna, zwana również Drawer, jest używana w aplikacji do nawigacji między różnymi ekranami. Szuflada zawiera listę pozycji menu, które odpowiadają różnym ekranom, takim jak główny ekran, ustawienia oraz o nas.

3.10.1 Fragmenty kodu

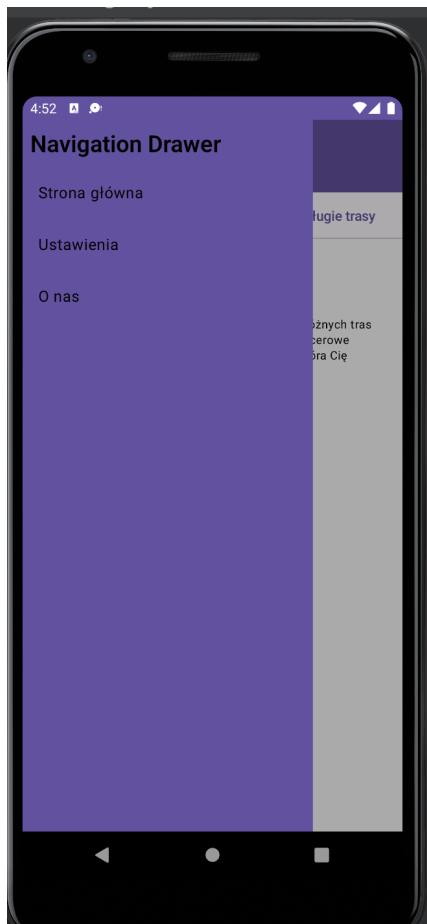
```
val drawerState = rememberDrawerState(DrawerValue.Closed)
ModalNavigationDrawer(
    drawerState = drawerState,
    drawerContent = {
        DrawerContent(navController = navController, drawerState = drawerState)
    }
)
```

Ten fragment kodu definiuje nawigacyjną szufladę modalną za pomocą komponentu 'ModalNavigationDrawer'. Szuflada ta jest otwierana i zamykana za pomocą stanu 'drawerState', który jest zarządzany przez funkcję 'rememberDrawerState'. Jako zawartość szuflady podawany jest komponent 'DrawerContent', który jest zdefiniowany w pliku 'Drawer.kt' i otrzymuje NavController oraz stan szuflady jako argumenty.

W pliku 'Drawer.kt' definiowane są komponenty Composable służące do budowania szuflady nawigacyjnej w aplikacji. 'DrawerContent' zawiera listę pozycji nawigacyjnych, takich jak "Strona główna", "Ustawienia" i "O nas", które użytkownik może wybrać. Każda pozycja jest klikalna i po jej kliknięciu odpowiada za nawigację do określonego celu za pomocą NavController. 'DrawerHeader' definiuje nagłówek dla szuflady, natomiast 'DrawerItem' definiuje pojedynczą pozycję w szufladzie, reprezentowaną jako tekst.

3.10.2 Zdjęcia z aplikacji

Szuflada nawigacyjna



3.11 Zdjęcia

Zdjęcia w aplikacji są zapisywane w usłudze Firebase Storage. Każde zdjęcie jest zapisywane w folderze "selfies" z unikalną dla danego szlaku nazwą pliku. Po zapisaniu zdjęcia, jego URL jest pobierany i zapisywany w bazie danych Firestore.

3.11.1 Fragmenty kodu

```
@Composable
fun imageCaptureFromCamera(context: Context, trail: Trail,
    onImageCaptured: (Uri) -> Unit) {
    val file = context.createImageFile(trail)

    val uri = FileProvider.getUriForFile(
        Objects.requireNonNull(context),
        context.packageName + ".provider", file
    )

    var capturedImageUri by remember {
        mutableStateOf<Uri>(Uri.EMPTY)
    }

    val cameraLauncher =
        rememberLauncherForActivityResult(ActivityResultContracts.TakePicture())
    { success ->
        capturedImageUri = uri
        if(success){
            updateTrailImage(trail, capturedImageUri) { uri ->
                onImageCaptured(uri)
                file.delete()
                GlobalVariables.refreshImage = true
                GlobalVariables.trailId = trail.id
            }
        }
    }
}

val permissionLauncher = rememberLauncherForActivityResult(
    ActivityResultContracts.RequestPermission()
) {
    if (it) {
        Toast.makeText(context, "Permission Granted", Toast.LENGTH_SHORT).show()
        cameraLauncher.launch(uri)
    } else {
        Toast.makeText(context, "Permission Denied", Toast.LENGTH_SHORT).show()
    }
}

Box(
    Modifier
        .fillMaxSize()
        .padding(10.dp)
) {
    FloatingActionButton( . . .
    ) {
        . . .
    }
}

fun Context.createImageFile(trail: Trail): File {
```

```

val timeStamp = SimpleDateFormat("yyyy_MM_dd_HH:mm:ss").format(Date())
val imageFileName = "JPEG_" + timeStamp + "_"
val image = File.createTempFile(
    imageFileName,
    ".jpg",
    externalCacheDir
)
return image
}

fun updateTrailImage(trail: Trail, capturedImageUri: Uri,
onImageCaptured: (Uri) -> Unit) {
    var filename = "selfie${trail.id}.jpg"

    val storage = Firebase.storage
    val storageRef = storage.reference

    var file = capturedImageUri
    val trailRef = storageRef.child("selfies/${filename}")
    val uploadTask = trailRef.putFile(file)
    uploadTask.addOnSuccessListener { _ ->
        getImage(trail) { uri ->
            onImageCaptured(uri)
        }
    }.addOnFailureListener { exception ->
        Log.e(ContentValues.TAG, "Failed to upload image", exception)
    }
}

fun getImage(trail: Trail, onImageLoaded: (Uri) -> Unit) {
    val filename = "selfie${trail.id}.jpg"

    val storage = Firebase.storage
    val storageRef = storage.reference

    val trailRef = storageRef.child("selfies/${filename}")

    trailRef.downloadUrl.addOnSuccessListener { uri ->
        onImageLoaded(uri)
    }.addOnFailureListener { exception ->
        Log.e(ContentValues.TAG, "getImage: Failed to download image", exception)
    }
}

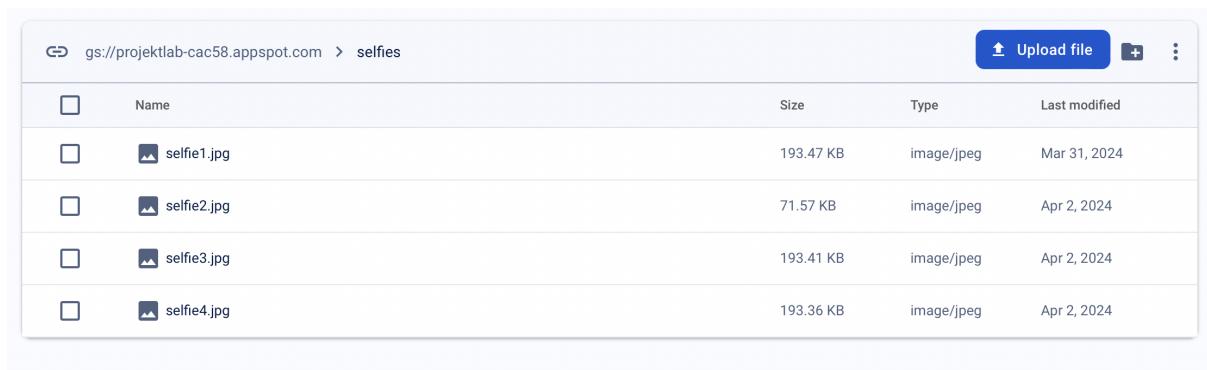
```

- **imageCaptureFromCamera:** Jest to funkcja Composable odpowiedzialna za przechwytywanie obrazu z aparatu. Najpierw tworzy plik obrazu o unikalnej nazwie na podstawie aktualnego czasu i daty za pomocą funkcji `createImageFile`. Następnie używa `rememberLauncherForActivityResult`, aby uzyskać dostęp do aparatu i pozwolić użytkownikowi zrobić zdjęcie. Po zrobieniu zdjęcia, plik obrazu jest aktualizowany na serwerze Firebase za pomocą funkcji `updateTrailImage` oraz zdjęcie stworzone za pomocą `createImageFile` zostaje usunięte.
- **createImageFile:** Jest to funkcja pomocnicza, która tworzy plik obrazu o unikalnej nazwie na podstawie aktualnego czasu i daty. Jest używana przez funkcję `imageCaptureFromCamera` do wygenerowania pliku, który będzie przechowywał przechwycony obraz.
- **updateTrailImage:** Funkcja odpowiedzialna za aktualizowanie obrazu na serwerze Firebase. Przyjmuje jako argumenty trasę (`trail`), przechwycony obraz (`capturedImageUri`) oraz funkcję `onImageCaptured`, która zostanie wywołana po zakończeniu aktualizacji obrazu. Wysyła przechwycony obraz na serwer Firebase i wykonuje działania po zakończeniu sukcesu lub niepowodzeniu.

- **getImage**: Funkcja służąca do pobierania obrazu z serwera Firebase na podstawie identyfikatora trasy. Przyjmuje jako argument trasę (trail) oraz funkcję `onImageLoaded`, która zostanie wywołana po pobraniu obrazu. Łączy się z serwerem Firebase, pobiera adres URL obrazu i wykonuje działania po zakończeniu sukcesu lub niepowodzeniu.

3.11.2 Zdjęcia z aplikacji

Widok z Firebase console



The screenshot shows the Firebase Storage console interface. At the top, there's a header with the URL "gs://projektlab-cac58.appspot.com" and a breadcrumb trail "selfies". On the right side of the header are three buttons: "Upload file" (blue), "+", and "...". Below the header is a table listing four files:

	Name	Size	Type	Last modified
<input type="checkbox"/>	selfie1.jpg	193.47 KB	image/jpeg	Mar 31, 2024
<input type="checkbox"/>	selfie2.jpg	71.57 KB	image/jpeg	Apr 2, 2024
<input type="checkbox"/>	selfie3.jpg	193.41 KB	image/jpeg	Apr 2, 2024
<input type="checkbox"/>	selfie4.jpg	193.36 KB	image/jpeg	Apr 2, 2024

3.12 Material Theme

Material Theme jest używany w aplikacji do zapewnienia spójnego wyglądu i stylu interfejsu użytkownika. Definiuje on kolory, typografię i inne elementy wyglądu, które są stosowane w całej aplikacji.

3.12.1 Fragmenty kodu

```
@Composable
fun ProjektLabTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable () -> Unit
) {
    val colorScheme = when {
        darkTheme -> DarkColorScheme
        else -> LightColorScheme
    }

    ..
    MaterialTheme(
        colorScheme = colorScheme,
        typography = Typography,
        content = content,
    )
}
```

W powyższym fragmencie kodu, ‘DarkColorScheme’ i ‘LightColorScheme’ są schematami kolorów dla ciemnego i jasnego motywów. ‘ProjektLabTheme’ jest funkcją Composable, która stosuje te schematy kolorów w zależności od tego, czy jest włączony ciemny motyw. Dodatkowo, definiuje ona typografię za pomocą zmiennej ‘Typography’. Definicje konkretnych kolorów oraz stylów typografii zapisane są odpowiednio w plikach Color.kt, Type.kt

3.13 Search field

Pole wyszukiwania jest używane na ekranie listy szlaków do filtrowania szlaków na podstawie wprowadzonego tekstu. Jest to pole tekstowe, które aktualizuje zmienną ‘searchText’ za każdym razem, gdy użytkownik wprowadza tekst.

3.13.1 Fragmenty kodu

```
@Composable
fun TrailListScreen(navController: NavController, context: Context,
searchText: String, isShortRoute: Boolean, trails: List<Trail>, gridcells: Int) {
    var selectedTrails = trails.filter { trail ->
        trail.name.contains(searchText, ignoreCase = true) ||
        trail.description.contains(searchText, ignoreCase = true)
    }.filter { trail ->
        if (isShortRoute) {
            trail.stages.sumBy { it.estimatedTime } < 180
        } else {
            trail.stages.sumBy { it.estimatedTime } >= 180
        }
    }

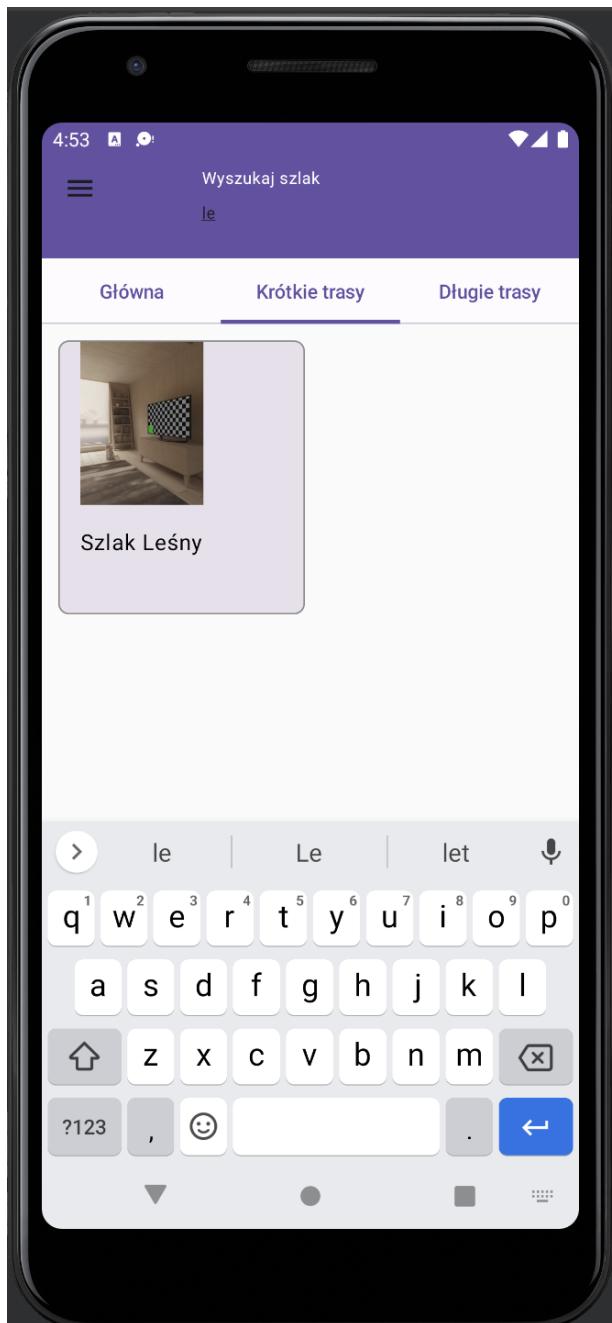
    Column {
        TrailList(trails = selectedTrails, navController = navController, gridcells)
    }
}

@Composable
fun SearchTextField(searchText: String, onSearchTextChanged: (String) -> Unit) {
    OutlinedTextField(
        value = searchText,
        onValueChange = onSearchTextChanged,
        label = { Text("Wyszukaj szlak", color = Color.White) },
        modifier = Modifier
            .padding(end = 8.dp, top = 8.dp, bottom = 8.dp)
            .height(60.dp),
        textStyle = MaterialTheme.typography.bodySmall
    )
}
```

- **TrailListScreen:** Jest to funkcja Composable odpowiedzialna za wyświetlanie listy szlaków na ekranie. Filtruje listę szlaków na podstawie wprowadzonego tekstu wyszukiwania i opcji krótkiego/niekrótkiego szlaku. Następnie renderuje listę szlaków za pomocą komponentu TrailList.
- **SearchTextField:** Jest to komponent Composable odpowiedzialny za wyświetlanie pola tekstowego do wyszukiwania szlaków. Pozwala użytkownikowi wprowadzać tekst wyszukiwania, który jest przekazywany do wywołanej funkcji onSearchTextChanged w komponencie nadziednym.

3.13.2 Zdjęcia z aplikacji

Widok podczas wyszukiwania



3.14 Ikony akcji

Ręcznie tworzone ikony akcji są używane na pasku narzędzi ekranu szczegółów szlaku do wykonania różnych akcji.

3.14.1 Fragmenty kodu

```
TopAppBar( . . .
    actions = {
        IconButton(onClick = { showSpeedSelection = !showSpeedSelection }) {
            val speedIcon = loadImageFromAssets(context, "icon_speed.png")
            speedIcon?.let { icon ->
                Icon(
                    painter = BitmapPainter(icon),
                    contentDescription = "Speed",
                    modifier = Modifier.size(30.dp)
                )
            }
        }
        IconButton(onClick = { showTimer = !showTimer }) {
            val timerIcon = loadImageFromAssets(context, "icon_timer.png")
            timerIcon?.let { icon ->
                Icon(
                    painter = BitmapPainter(icon),
                    contentDescription = "Timer",
                    modifier = Modifier.size(30.dp)
                )
            }
        }
    },
    . . .
)
```

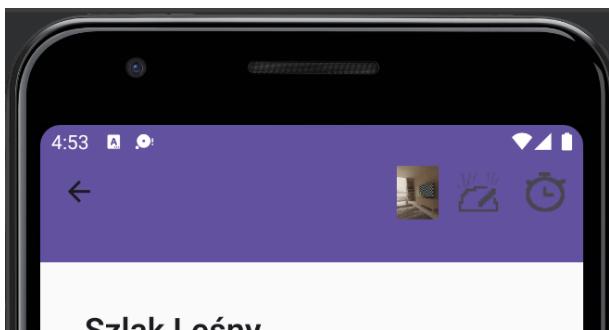
Ten fragment kodu definiuje pasek akcji ‘TopAppBar’ z dwiema ikonami akcji. Te ikony są wczytywane z plików obrazów ‘“icon_speed.png”’, i ‘“icon_timer.png”’, za pomocą funkcji ‘loadImageFromAssets’. Każda ikona jest renderowana jako przycisk ‘IconButton’ i wyświetlana w pasku akcji. Ikony te reprezentują akcje związane z prędkością i czasem.

```
@Composable
fun loadImageFromAssets(context: Context, fileName: String): ImageBitmap? {
    return try {
        val inputStream = context.assets.open(fileName)
        val drawable = Drawable.createFromStream(inputStream, null)
        drawable?.toBitmap()?.asImageBitmap()
    } catch (e: Exception) {
        e.printStackTrace()
        null
    }
}
```

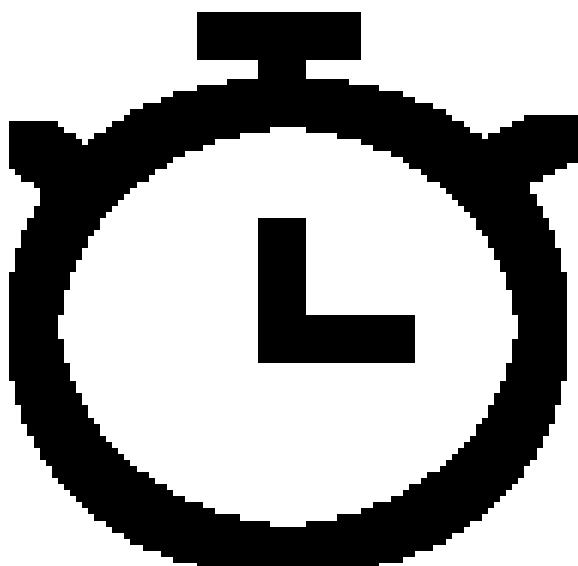
Ten fragment kodu definiuje funkcję ‘loadImageFromAssets’, która wczytuje obraz z zasobów aplikacji na podstawie nazwy pliku i kontekstu aplikacji. Funkcja próbuje otworzyć strumień do pliku obrazu z folderu zasobów aplikacji. Następnie konwertuje ten strumień na obiekt Bitmap, który jest następnie przekształcany w obiekt ImageBitmap. Jeśli wystąpi błąd podczas operacji, funkcja zwróci wartość null.

3.14.2 Zdjęcia z aplikacji

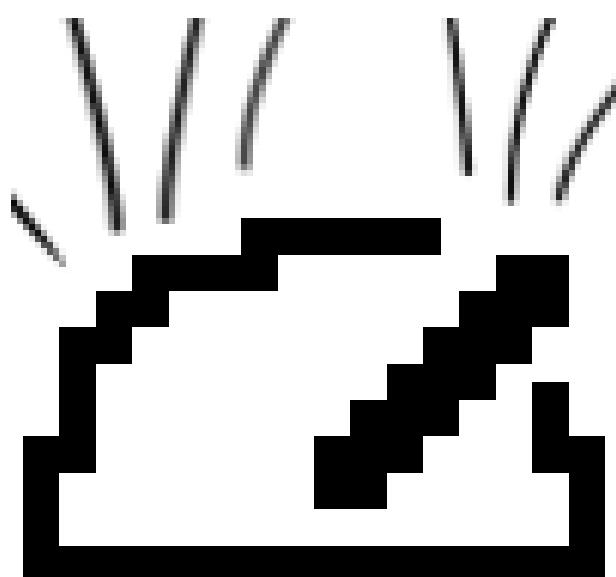
- Widok ikon w aplikacji



- Ikona stopera



- Ikona wyboru szybkości chodzenia



3.15 Ikona aplikacji

Ikona została przygotowana w programie do tworzenia obrazów pikselowych, podobnie jak ikony aplikacji. Po przygotowaniu ikony, została ona dodana do zasobów aplikacji jako plik mmap. Następnie, używając tego atrybutu w pliku manifestu, określono, którą z przygotowanych ikon należy użyć jako ikony aplikacji. Ta ikona jest wykorzystywana na urządzeniach, które wymagają zaokrąglonej wersji ikony aplikacji.

3.15.1 Fragmenty kodu

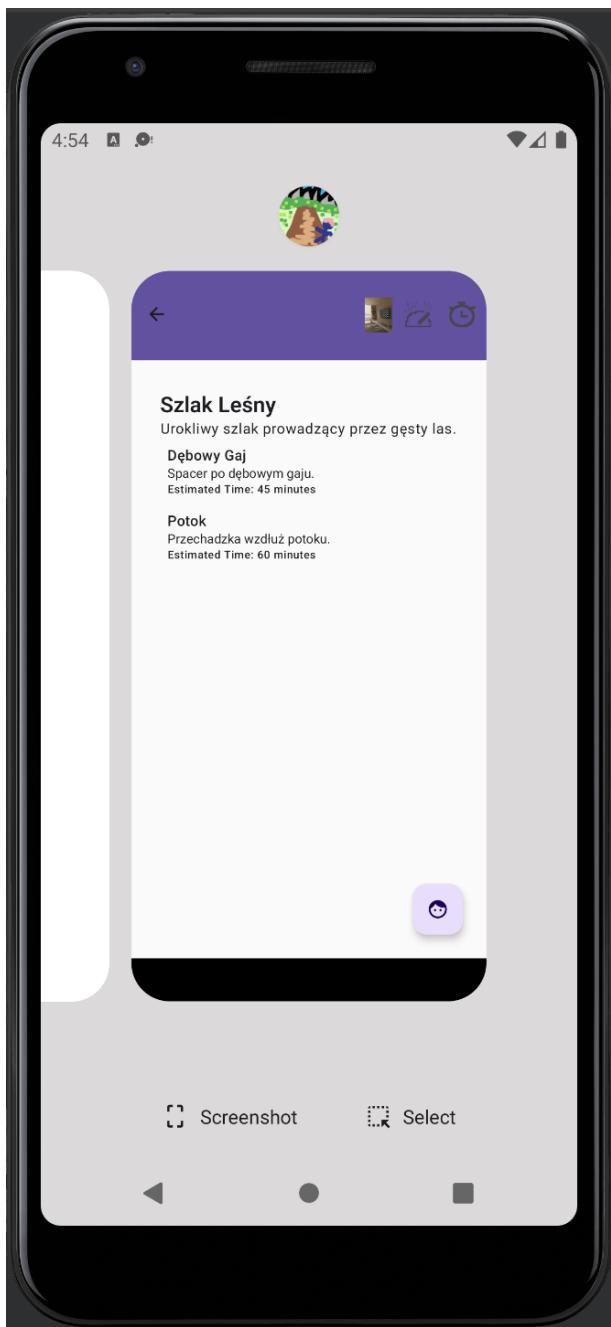
```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools">  
  
    <application  
        ...  
        android:icon="@mipmap/ic_launcher"  
        android:roundIcon="@mipmap/ic_launcher_round"  
        ...  
    </application>  
    ...  
</manifest>
```

Fragment kodu przedstawia plik manifestu aplikacji Android, gdzie określone są ustawienia aplikacji. Atrybut ‘android:icon’ określa ikonę aplikacji wyświetlaną na ekranie głównym urządzenia, natomiast ‘android:roundIcon’ określa ikonę aplikacji wyświetlaną na zaokrąglonym tle, na przykład w panelu powiadomień lub w menu aplikacji.

3.15.2 Zdjęcia z aplikacji



- Widok ikony aplikacji z ostatnio używanych aplikacji



3.16 Animacja

Animacja jest używana w aplikacji do tworzenia efektu opadającego deszczu. Jest to animacja, która jest wywoływana, gdy użytkownik kliknie na chmurę. Animacja tworzy 20 kropli deszczu, które spadają z chmury. Animacja trwa 10sekund przed wyłączeniem aplikacji.

3.16.1 Fragmenty kodu

```
@Composable
fun AnimatedScene() {
    val isRaining = remember { mutableStateOf(false) }
    val cloudSize = 200.dp
    val cloudOffset1 = remember { mutableStateOf(Offset.Zero) }
    val cloudOffset2 = remember { mutableStateOf(Offset.Zero) }

    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(Color(0xFFADD8E6))
    ) {
        Image(
            painter = painterResource(id = if (isRaining.value) R.drawable.cloud2
                else R.drawable.cloud1),
            contentDescription = "Cloud",
            modifier = Modifier
                .size(cloudSize)
                .align(Alignment.TopStart)
                .clickable { isRaining.value = !isRaining.value }
                .onGloballyPositioned { coordinates ->
                    cloudOffset1.value = coordinates.positionInRoot()
                }
        )
        Image(
            painter = painterResource(id = if (isRaining.value) R.drawable.cloud2
                else R.drawable.cloud1),
            contentDescription = "Cloud",
            modifier = Modifier
                .size(cloudSize)
                .align(Alignment.CenterEnd)
                .clickable { isRaining.value = !isRaining.value }
                .onGloballyPositioned { coordinates ->
                    cloudOffset2.value = coordinates.positionInRoot()
                }
        )
    }

    if (isRaining.value) {
        for(i in 0 until 10) {
            RainAnimation(cloudSize, cloudOffset1.value)
            RainAnimation(cloudSize, cloudOffset2.value)
        }
    }
}
```

Fragment kodu zawiera funkcję ‘AnimatedScene‘, która definiuje animowaną scenę przedstawiającą chmurę na niebie. Użytkownik może kliknąć na chmurę, co spowoduje zmianę jej wyglądu i uruchomienie symulacji opadów deszczu. Scena składa się z dwóch obrazków chmur, których wygląd zmienia się w zależności od stanu ‘isRaining‘. Gdy ‘isRaining‘ jest ustawione na ‘true‘, uruchamiana jest animacja opadów deszczu, symulująca spadanie kropli deszczu z chmur.

```

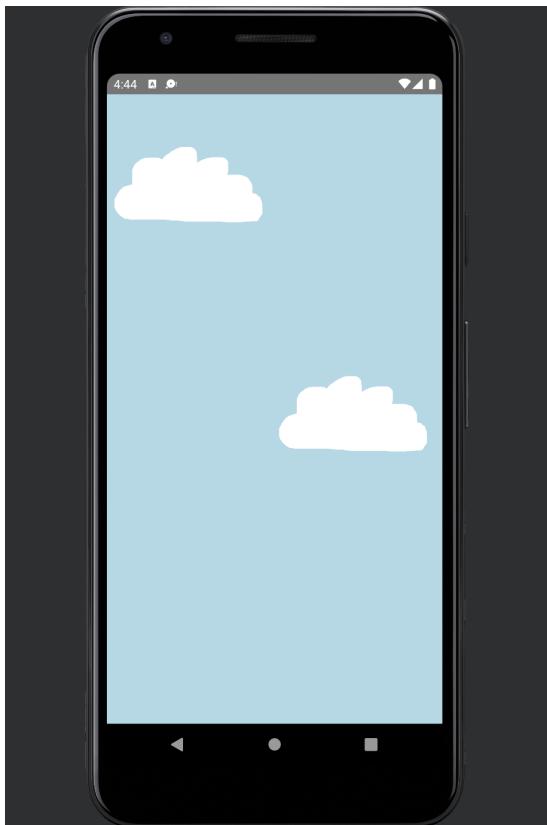
@Composable
fun RainAnimation(cloudSize: Dp, cloudOffset: Offset) {
    val raindropCount = 20
    val raindropOffsets = remember { List(raindropCount) { Animatable(0f) } }
    val coroutineScope = rememberCoroutineScope()
    Canvas(modifier = Modifier.fillMaxSize()) {
        val raindropWidth = 2.dp.toPx()
        val raindropHeight = 7.dp.toPx()
        for (i in 0 until raindropCount) {
            val duration = 100 + Random.nextInt(30000)
            val raindropOffset = raindropOffsets[i]
            coroutineScope.launch {
                raindropOffset.animateTo(
                    targetValue = 1f,
                    animationSpec = infiniteRepeatable(
                        animation = tween(duration, easing = LinearEasing),
                        repeatMode = RepeatMode.Restart
                    )
                )
            }
            val x = cloudOffset.x + 30 + i * (cloudSize.toPx() - 70) / raindropCount
            val y = cloudOffset.y + 400 + raindropOffset.value * size.height
            drawRect(
                color = Color.Blue,
                topLeft = Offset(x, y),
                size = Size(raindropWidth, raindropHeight)
            )
        }
    }
}

```

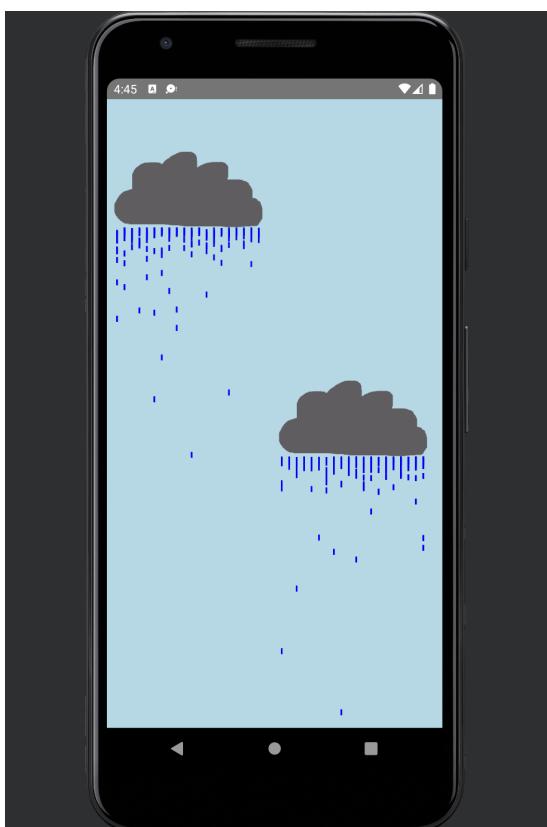
W powyższym fragmencie kodu, ‘RainAnimation’ jest funkcją Composable, która tworzy animację opadającego deszczu. Funkcja ta tworzy 20 kropli deszczu, które spadają z chmury. Każda kropla deszczu jest animowana tak, aby spadała z różną prędkością, co jest osiągane przez losowanie czasu trwania animacji dla każdej kropli.

3.16.2 Zdjęcia z aplikacji

- Widok animacji



- Widok animacji po naciśnięciu chmury



4 Podsumowanie

Projekt jest aplikacją mobilną napisaną w języku Kotlin, z wykorzystaniem platformy Android i nowoczesnej biblioteki do budowy interfejsu użytkownika - Jetpack Compose. Aplikacja służy do przeglądania i odkrywania tras turystycznych, dostarczając użytkownikowi szczegółowych informacji o każdej trasie.

Głównym ekranem aplikacji jest ‘MainCard’, który zawiera powitanie i krótki opis aplikacji. Użytkownik może przeglądać dostępne trasy, które są prezentowane w formie kart. Każda karta zawiera podstawowe informacje o trasie, takie jak jej nazwa i obraz. Po kliknięciu na kartę, użytkownik jest przenoszony do ekranu szczegółów trasy, gdzie może znaleźć więcej informacji, takich jak opis trasy, szacowany czas trwania i zdjęcia.

Aplikacja korzysta z Material Theme do zapewnienia spójnego i atrakcyjnego wyglądu interfejsu użytkownika. Wszystkie ekranы są zorganizowane za pomocą systemu nawigacji Jetpack Compose, który umożliwia łatwe przełączanie między różnymi ekranami.

Aplikacja zawiera również animacje, które dodają dynamiki i atrakcyjności interfejsu użytkownika. Na przykład, jest animacja opadającego deszczu, która jest wywoływana, gdy użytkownik kliknie na chmurę na ekranie szczegółów trasy.

Projekt jest zbudowany z wykorzystaniem systemu budowania Gradle, co umożliwia łatwe zarządzanie zależnościami i budowę aplikacji. Kod źródłowy projektu jest zarządzany za pomocą systemu kontroli wersji Git, co umożliwia efektywne śledzenie zmian.