Aplikacje mobilne – TrailApp

Anna Ogorzałek 151805

Aplikacja mobilna TraillApp służy do przedstawienia szlaków górskich. Zaimplementowana została poprzez użycie dwóch głównych aktywności – do wyświetlania listy szlaków, a także do wyświetlania szczegółów danego szlaku. Poniżej zostaną opisane szczegóły co do implementacji wymagań minimalnych aplikacji.

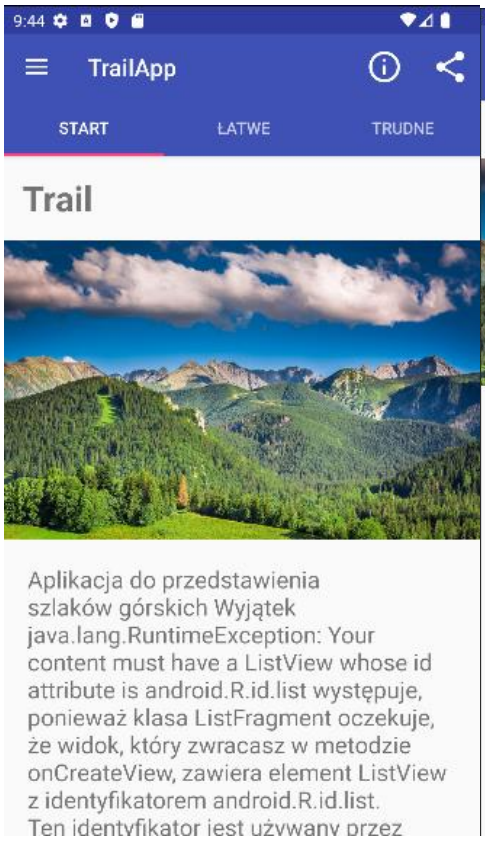
Aplikacja powinna mieć wersję układu dla smartfonów i osobną dla tabletów

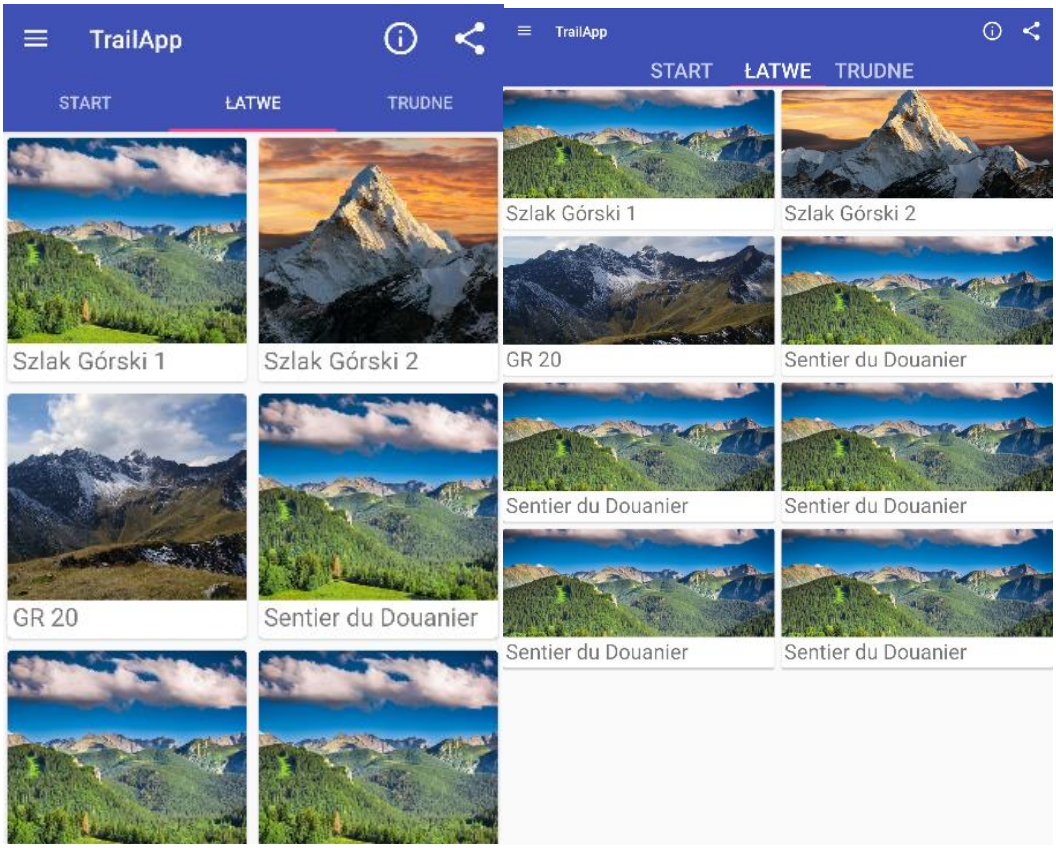
Różnice między układem dla smartfonów a tableta w aplikacji są głównie związane z dostosowaniem rozmiaru tekstu do różnych rozmiarów ekranu. Początkowo aplikacja miała założenie dotyczące wyświetlania dwóch fragmentów na tablecie, co było dobrym pomysłem na wstępnym etapie tworzenia projektu. Jednakże, dodanie kategorii do wyboru oraz opcji wyboru tras górskich za pomocą obrazków, spowodowało, że układ na tablecie zaczął lepiej wyglądać w formie podobnej do tej, przystosowanej do telefonów

Aby osiągnąć różnicę w wielkości tekstu między smartfonem a tabletem, użyto mechanizmu zasobów (resources) w Androidzie. Zdefiniowaliśmy różne rozmiary tekstu w plikach dimens.xml dla wersji zwykłej i dla wersji large, dostosowując je odpowiednio do wielkości ekranu. Na przykład:

```
<resources>
    <dimen name="text_size_small">10sp</dimen>
    <dimen name="text_size_medium">20sp</dimen>
    <dimen name="text_size_large">30sp</dimen>
</resources>
```

Przykładowe ekrany (po lewej telefon, po prawej tablet)





Aplikacja powinna działać poprawnie po zmianie orientacji urządzenia

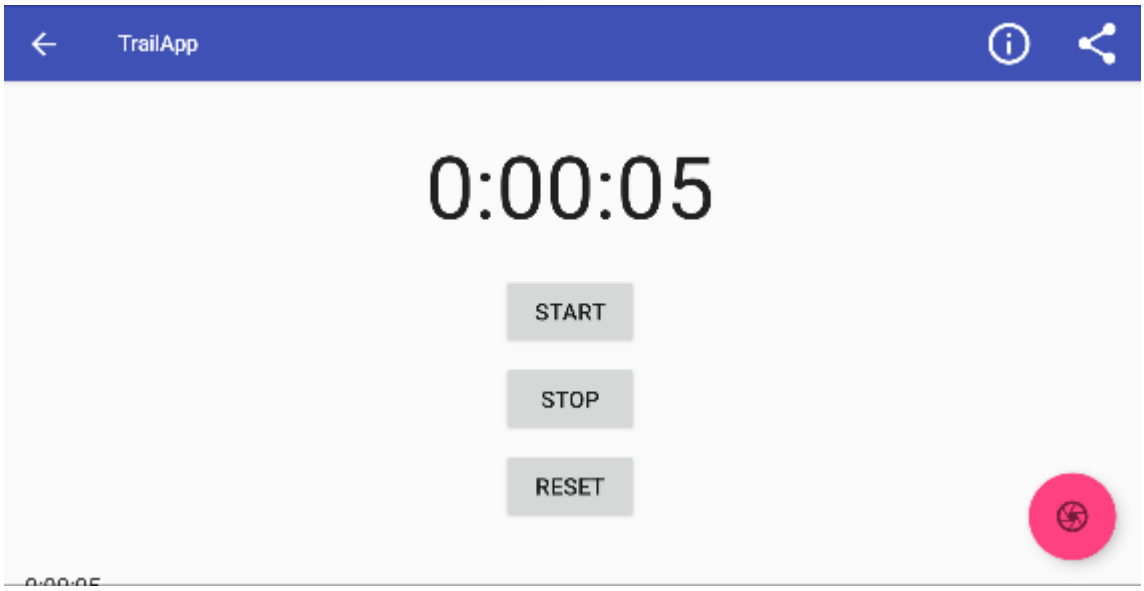
Działanie aplikacji po zmianie orientacji urządzenia zostało zapewnione poprzez prawidłową obsługę cyklu życia fragmentu oraz zachowanie stanu aplikacji. Główny wpływ na działanie aplikacji ma moment zmiany orientacji, szczególnie w przypadku działania np. stopera, który zaczyna odliczać czas. Aby zapewnić poprawne działanie aplikacji po zmianie orientacji, konieczne jest zachowanie stanu aplikacji i przypisanie poprzednich wartości do elementów interfejsu użytkownika.

W StopperFragment.kt zaimplementowano to w następujący sposób:Tutaj zapisujemy zmienne w Instancji

```
// Zapisujemy zmienne w instancji
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putInt("seconds", seconds)
    outState.putBoolean("running", running)
    outState.putBoolean("warRunning", wasRunning)
}
// Przy tworzeniu StopperFragment odczytujemy poprzednio zapisane zmienne,
aby je ustawić (jeśli instancja nie jest pusta)
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    if (savedInstanceState != null) {
        seconds = savedInstanceState.getInt("seconds")
        running = savedInstanceState.getBoolean("running")
        wasRunning = savedInstanceState.getBoolean("wasRunning")
    }
}
```

Dzięki tym operacjom, po zmianie orientacji urządzenia, poprzednie wartości zostaną przywrócone do stopera, zapewniając ciągłość działania aplikacji. Jest to ważne szczególnie w przypadku funkcji, które działają w tle, jak np. stoper, aby uniknąć utraty danych lub błędów w działaniu aplikacji po zmianie orientacji.

Płzykład działania:



We fragmencie szczegółów należy zagnieździć fragment dynamiczny stopera / zegara / krokomierza

```
<LinearLayout

…

    <FrameLayout
        android:id="@+id/stopper_container"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

…

</LinearLayout>
```

Stoper i zegar mają wyświetlać czas z dokładnością do sekundy

Aby zapewnić wyświetlanie czasu przez stoper z dokładnością do sekundy, zaimplementowana została funkcja, która aktualizuje widok czasu co sekundę. Stoper wyświetla czas w formacie godziny:minuty:sekundy, gdzie godziny są opcjonalne, a minuty i sekundy mają zawsze dwucyfrową reprezentację.

```
private fun runStoper(view: View) {
    val timeView: TextView = view.findViewById(R.id.time_view)
    val handler = Handler()
    handler.post(object : Runnable {
        override fun run() {
            val hours: Int = seconds / 3600
            val minutes: Int = (seconds % 3600) / 60
            val secs: Int = seconds % 60
            val time: String = String.format("%d:%02d:%02d", hours,
minutes, secs)
            timeView.text = time
            if (running) {
                seconds++
            }
            handler.postDelayed(this, 1000)
        }
    })
}
```

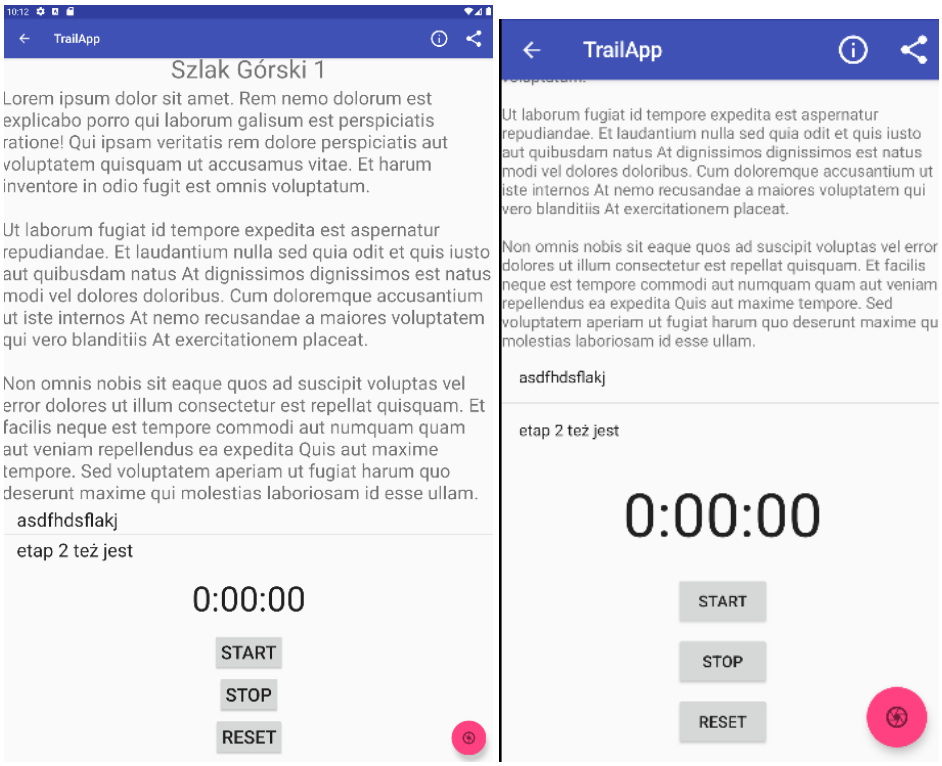
Funkcja runStoper jest odpowiedzialna za aktualizację widoku czasu co sekundę. Początkowo pobierany jest widok TextView, który będzie wyświetlał czas. Następnie korzystając z mechanizmu Handler, uruchamiana jest pętla, która będzie się wykonywać co sekundę.

Wewnątrz pętli obliczana jest aktualna wartość godzin, minut i sekund na podstawie zmiennej seconds, która reprezentuje łączną liczbę sekund, które upłynęły. Wartości formatowane są w odpowiedni sposób, aby uzyskać tekstową reprezentację czasu w formacie "godziny:minuty:sekundy" za pomocą metody String.format.

Następnie aktualizujemy tekstowy widok czasu (timeView) nowym czasem. Jeśli stoper jest włączony (running == true), inkrementujemy liczbę sekund. Na koniec, ponownie planujemy uruchomienie tej samej funkcji po upływie 1000 milisekund, co zapewnia cykliczne aktualizowanie czasu co sekundę.

Stoper powinien działać poprawnie na smartfonach i tabletach

Stoper wyświetla się poprawnie działa na obydwu urządzeniach



Stoper powinien działać poprawnie po zmianie orientacji urządzenia

Opisane wyżej

Stoper powinien mieć przyciski, możliwość zapamiętania wyniku

Stoper został rozszerzony o przyciski umożliwiające jego kontrolę oraz dodanie wyników do listy.

W pliku fragment_stopper.xml znajdują się przyciski start, stop i reset, które umożliwiają odpowiednio uruchomienie, zatrzymanie oraz resetowanie stopera.

```
<Button
    android:id="@+id/start_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="20dp"
    android:text="@string/start"
    android:textSize="@dimen/text_size_medium"
    />
<Button
    android:id="@+id/stop_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="8dp"
    android:text="@string/stop"
    android:textSize="@dimen/text_size_medium"
    />
```

```
<Button
    android:id="@+id/reset_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_marginTop="8dp"
    android:text="@string/reset"
    android:textSize="@dimen/text_size_medium"
    />
```

W klasie StopperFragment, w metodzie onCreateView(), przypisywane są przyciski do odpowiednich zmiennych oraz ustawiane są Listenery, które nasłuchują na ich naciśnięcie. Klasa StopperFragment implementuje interfejs View.OnClickListener, który pozwala na obsługę zdarzeń kliknięcia przycisków. W metodzie onClick(v: View), w zależności od id kliknięty przycisku, wywoływane są odpowiednie metody onClickStart(), onClickStop() lub onClickReset().

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View {
    val layout: View = inflater.inflate(R.layout.fragment_stopper,
container, false)
    runStoper(layout)
    val startButton = layout.findViewById<View>(R.id.start_button) as
Button
    startButton.setOnClickListener(this)
    val stopButton = layout.findViewById<View>(R.id.stop_button) as Button
    stopButton.setOnClickListener(this)
    val resetButton = layout.findViewById<View>(R.id.reset_button) as
Button
    resetButton.setOnClickListener(this)
    return layout
}
override fun onClick(v: View) {
    when (v.id) {
        R.id.start_button -> onClickStart()
        R.id.stop_button -> onClickStop()
        R.id.reset_button -> onClickReset()
    }
}
private fun onClickStart() {
    running = true
}

private fun onClickStop() {
    running = false
    listener?.onTimeStopped(getFormattedTime())
}

private fun onClickReset() {
    running = false
    seconds = 0
}
```

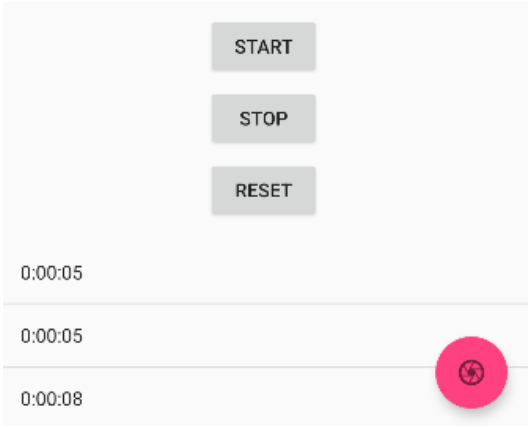
Metoda onClickStart() uruchamia stoper, onClickStop() zatrzymuje go oraz dodaje aktualny czas do listy wyników poprzez wywołanie metody listenera onTimeStopped(). Metoda onClickReset() resetuje stoper do stanu początkowego.

W klasie DetailActivity, w której znajduje się lista wyników, implementowana jest metoda onTimeStopped(time: String), która dodaje zapisany czas do listy i wyświetla ją poniżej stopera.

```
override fun onTimeStopped(time: String) {
    timeList.add(time)
    timeAdapter.notifyDataSetChanged()
    setListViewHeightBasedOnItems(timeAdapter, timeListView)
}
```

Dodatkowo, wykorzystywana jest metoda setListViewHeightBasedOnItems(), która zapewnia, że na ekranie wyświetlają się wszystkie zapisane czasy, poprzez dynamiczne zwiększanie wysokości widoku ListView.

Widok zapisanych wyników:



Karty kategorii zamiast listy nazw szlaków mają używać widoku RecyclerView z układem siatki (grid), w którym poszczególne pozycje (szlaki) będą prezentowane w postaci obrazka i nazwy, dla których użyto widoku CardView. Kliknięcie wybranej pozycji (szlaku) powoduje wyświetlenie szczegółów…

W aplikacji zostały użyte dwa fragmenty - jeden zawierający łatwe szlaki, a drugi trudne. Każdy z tych fragmentów filtruje potrzebne szlaki i korzysta z adaptera CaptionedImagesAdapter, który obsługuje widok RecyclerView. Ten adapter umożliwia wyświetlanie obrazków i nazw szlaków.

W metodzie onCreateView() każdego z fragmentów, następuje konfiguracja RecyclerView z wykorzystaniem adaptera oraz ustawienie GridLayoutManagera, który umożliwia wyświetlanie elementów w układzie siatki. W przypadku łatwych i trudnych szlaków, spanCount ustawiony jest na 2, co oznacza, że w jednym wierszu znajdują się dwie pozycje..

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    val trailRecycler =
        inflater.inflate(R.layout.fragment_trail_easy_list, container,
false) as RecyclerView

…
    val adapter = CaptionedImagesAdapter(trailsNames.toTypedArray(),
trailsImages.toIntArray())
    trailRecycler.setAdapter(adapter)

    val layoutManager = GridLayoutManager(activity, 2)
    trailRecycler.setLayoutManager(layoutManager)
```

```
    adapter.setListener(object : CaptionedImagesAdapter.Listener {
        override fun onClick(position: Int) {
            val intent = Intent(requireActivity(),
DetailActivity::class.java)
            intent.putExtra("TRAIL_ID", position)
            requireActivity().startActivity(intent)
        }
    })

    return trailRecycler
}
```

W pliku XML layoutu dla RecyclerView, należy pamiętać o ustawieniu odpowiednich właściwości, takich jak rozmiar oraz orientacja:

```
<androidx.recyclerview.widget.RecyclerView
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:id="@+id/tab1_recycler"
    tools:context=".fragments.TrailEasyListFragment">

</androidx.recyclerview.widget.RecyclerView>
```

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).

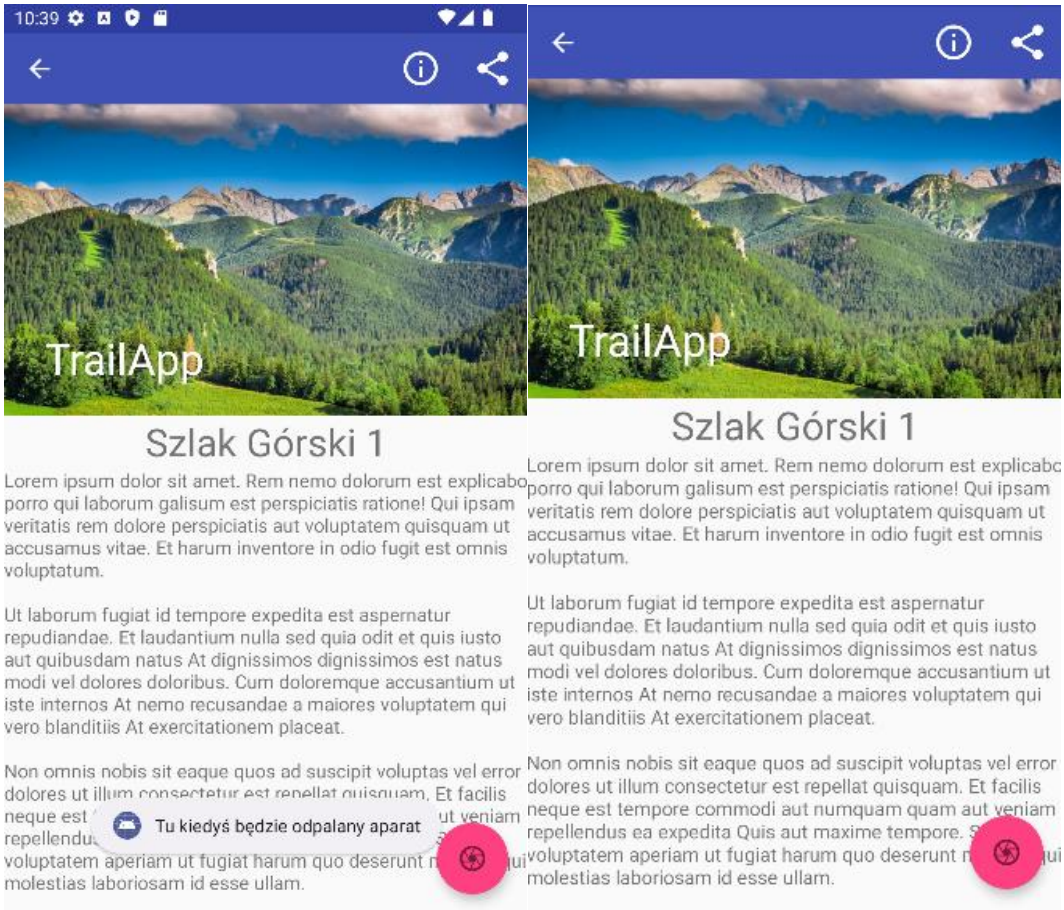
W celu dodania przycisku FAB (floating action button) na ekranie szczegółów, został użyty widok FloatingActionButton w pliku XML. Po kliknięciu przycisku, wywoływana jest funkcja onClickDone(), która wyświetla komunikat Toast.

W pliku XML dodany został widok FloatingActionButton:

```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="end|bottom"
    android:layout_margin="16dp"
    android:contentDescription="@string/ftb_desc"
    android:onClick="onClickDone"
    android:src="@drawable/camera" />
```

Następnie, w odpowiedniej aktywności, dodana została obsługa kliknięcia przycisku. Funkcja onClickDone() wyświetla krótki komunikat Toast, informując użytkownika, że funkcjonalność uruchomienia aparatu będzie dostępna w przyszłości.

```
fun onClickDone(view: View) {
    val context = view.context
    val toast = Toast.makeText(context, "Tu kiedyś będzie odpalany aparat",
Toast.LENGTH_SHORT)
    toast.show()
}
```



Każda aktywność ma mieć pasek aplikacji w postaci paska narzędzi.

Na ekranie szczegółów szlaków został zaimplementowany pasek aplikacji w postaci paska narzędzi, który zawiera tytuł oraz przycisk powrotu do poprzedniej aktywności. Pasek narzędzi został wydzielony do osobnego pliku XML dla łatwiejszego użycia w różnych aktywnościach. W głównej aktywności, MainActivity.kt, pasek narzędzi oraz kategorie są ustawiane przy użyciu kodu:

```
val toolbar: Toolbar = findViewById<View>(R.id.toolbar) as Toolbar
setSupportActionBar(toolbar)

val tabLayout = findViewById<View>(R.id.tabs) as TabLayout
tabLayout.setupWithViewPager(pager)
```

Toolbar_main.xml:

```
<androidx.appcompat.widget.Toolbar
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" />
```

Część kodu z activity_main.xml

```
<com.google.android.material.appbar.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar">

    <include
```

```
        android:id="@+id/toolbar"
        layout="@layout/toolbar_main"
        android:baselineAligned="false" />

    <com.google.android.material.tabs.TabLayout
        android:id="@+id/tabs"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

</com.google.android.material.appbar.AppBarLayout>
```

Druga aktywność, DetailActivity, korzysta z tego samego paska narzędzi. W tym przypadku dodatkowo ustawiamy actionBar w formie strzałki, która cofa użytkownika do nadrzędnej aktywności (w tym wypadku do aktywności głównej, MainActivity). Ustawienie to znajduje się w AndroidManifest.xml:

```
<activity
    android:name=".activity.DetailActivity"
    android:exported="false"
    android:parentActivityName=".activity.MainActivity" />
```

Aby umożliwić przewijanie ekranu szczegółów w pionie wraz z paskiem aplikacji, zastosowaliśmy CollapsingToolbarLayout wraz z tekstem, obrazkiem i paskiem narzędzi. Następnie, poza tym widokiem, umieściliśmy NestedScrollView, który umożliwia przewijanie treści ekranu. Osiągnięty efekt jest zgodny z oczekiwaniami.

Odpowiednio pasek narzędzi strony głównej i ekranu szczegółów:

Ekran szczegółów ma być przewijany w pionie razem z paskiem aplikacji. Na ekranie szczegółów obrazek ma się pojawić na pasku aplikacji, ale ma się razem z nim zwijać.

Na ekranie szczegółów szlaków została zaimplementowana możliwość przewijania w pionie razem z paskiem aplikacji. Aby osiągnąć ten efekt, w pliku XML activity_detail.xml zastosowaliśmy strukturę opartą na CoordinatorLayout.

Wewnątrz CoordinatorLayout umieszczony został AppBarLayout, który zawiera CollapsingToolbarLayout. W CollapsingToolbarLayout umieszczone zostały TextView oraz ImageView, które wyświetlają odpowiednie dane na pasku aplikacji. Dodatkowo, zaimportowaliśmy toolbar z pliku toolbar_main.xml.

Poza AppBarLayout umieszczony został NestedScrollView, który umożliwia przewijanie treści ekranu. Wewnątrz NestedScrollView znajduje się LinearLayout zawierający FragmentContainerView oraz ListView, w którym wyświetlane są szczegóły szlaku.

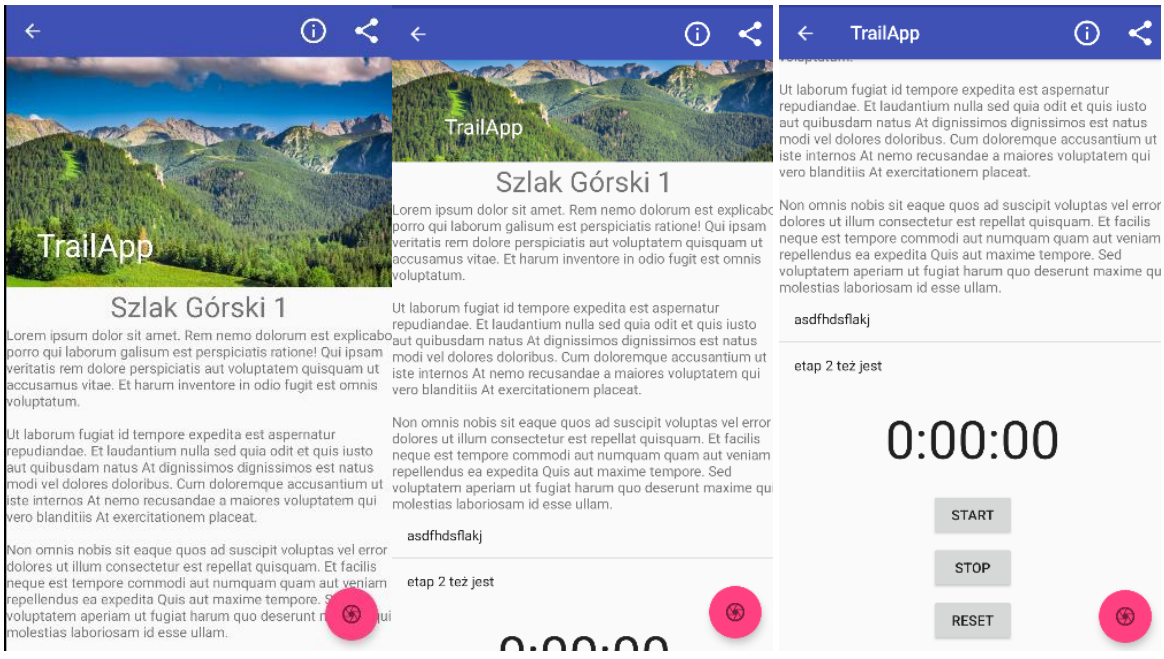
Na samym dole pliku XML znajduje się FloatingActionButton, który umożliwia dodanie interaktywnego przycisku na ekranie.

Ostateczna struktura pliku activity_detail.xml wygląda następująco:

```
<androidx.coordinatorlayout.widget.CoordinatorLayout

    <com.google.android.material.appbar.AppBarLayout>

        <com.google.android.material.appbar.CollapsingToolbarLayout>

            <TextView/>

            <ImageView/>

            <include layout="@layout/toolbar_main""/>

        </com.google.android.material.appbar.CollapsingToolbarLayout>
    </com.google.android.material.appbar.AppBarLayout>

    <androidx.core.widget.NestedScrollView>

        <LinearLayout>
        <androidx.fragment.app.FragmentContainerView />

        <ListView/>
        </LinearLayout>

    </androidx.core.widget.NestedScrollView>


<com.google.android.material.floatingactionbutton.FloatingActionButton/>

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

Dzięki takiemu układowi, użytkownik może swobodnie przewijać zawartość ekranu wraz z paskiem aplikacji, a obrazek na pasku aplikacji będzie się zwijał wraz z przewijaniem.

Osiągnięty efekt:



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

Aby umożliwić przechodzenie pomiędzy kartami za pomocą gestu przeciągnięcia, zastosowaliśmy ViewPager na całym ekranie w pliku activity_main.xml. ViewPager jest komponentem, który umożliwia przewijanie pomiędzy różnymi fragmentami lub widokami.

W pliku activity_main.xml dodano ViewPager:

```
<androidx.viewpager.widget.ViewPager
    android:id="@+id/pager"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

W MainActivity definiujemy pager z zastosowaniem stworzonego adaptera:

```
val pagerAdapter = SectionsPagerAdapter(supportFragmentManager, resources)
val pager = findViewById<View>(R.id.pager) as ViewPager
pager.setAdapter(pagerAdapter)
```

Adapter SectionsPagerAdapter jest odpowiedzialny za dostarczenie odpowiednich fragmentów do ViewPager w zależności od ich pozycji. W zależności od pozycji, adapter przekierowuje do odpowiedniego fragmentu:

```
override fun getItem(position: Int): Fragment {
    when (position) {
        0 -> return TrailListFragment()
        1 -> return TrailEasyListFragment()
        2 -> return TrailHardListFragment()
    }
    return TrailListFragment()
}
```

Dzięki temu, użytkownik może przemieszczać się między kartami za pomocą gestu przeciągnięcia (swipe), co zapewnia płynne przejścia między różnymi widokami.

Screenshot z przesuwania kategorii:

Do aplikacji należy dodać szufladę nawigacyjną

Dodano szufladę nawigacyjną do głównej aktywności (activity_main.xml) poprzez NavigationView. Szuflada nawigacyjna zawiera widok składający się ze zdjęcia aplikacji oraz informacji o użytkowniku, a także menu z dwoma przykładowymi elementami.

W górnej części szuflady nawigacyjnej wykorzystano stworzony samodzielnie widok, który zawiera zdjęcie aplikacji oraz dane użytkownika, takie jak login i adres e-mail. Niższa część wykorzystuje drawer_menu, w którym dodano dwa przykładowe elementy: "Import" i "Galeria".

```
<com.google.android.material.navigation.NavigationView
    android:id="@+id/nav_view"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    app:headerLayout="@layout/drawer_header"
    app:menu="@menu/drawer_menu" />
```

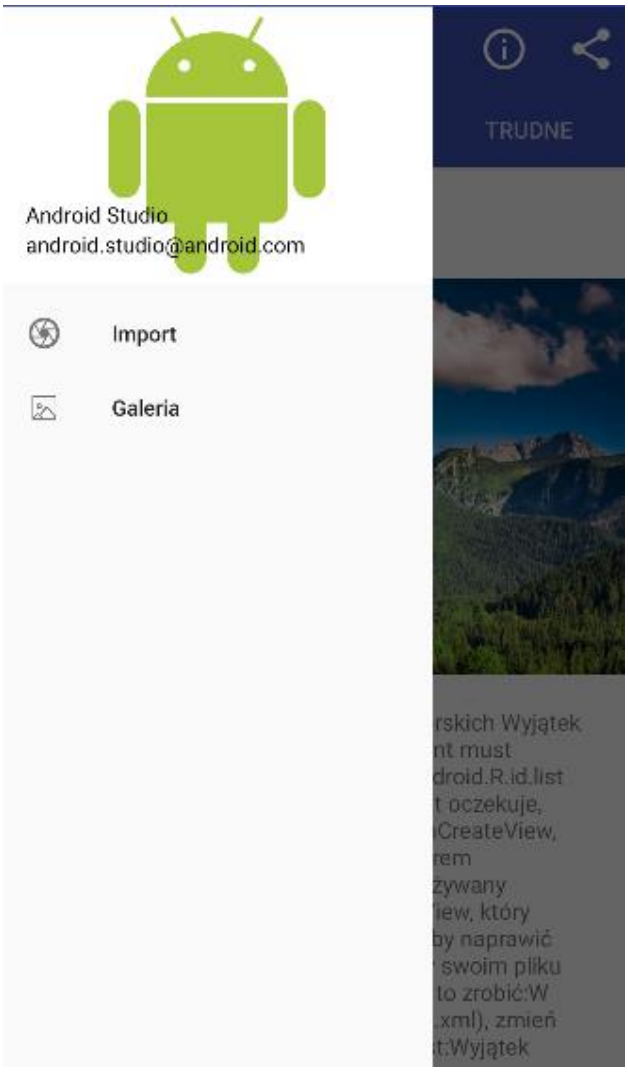
Plik drawer_menu.xml zawiera definicje elementów menu:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/drawer_import"
        android:icon="@drawable/camera"
        android:title="@string/drawer_import" />

    <item
        android:id="@+id/drawer_gallery"
        android:icon="@drawable/gallery"
        android:title="@string/drawer_gallery"/>
</menu>
```

W implementacji w MainActivity.kt, został utworzony listener oczekujący na włączenie menu oraz listener reagujący na naciśnięcie przycisku w menu. W zależności od wybranego elementu menu, użytkownik zostaje przeniesiony do odpowiedniej aktywności.

```
val drawer = findViewById<View>(R.id.list_frag) as DrawerLayout
val toggle = ActionBarDrawerToggle(
    this,
    drawer,
    toolbar,
    R.string.open_drawer,
    R.string.close_drawer
)
drawer.addDrawerListener(toggle)
toggle.syncState()

val navigationView = findViewById<View>(R.id.nav_view) as NavigationView
navigationView.setNavigationItemSelectedListener(this)
override fun onNavigationItemSelected(item: MenuItem): Boolean {
    val id = item.itemId
    val intent: Intent = when (id) {
        R.id.drawer_import -> Intent(this, GalleryActivity::class.java)
        R.id.drawer_gallery -> Intent(this, ImportActivity::class.java)
        else -> Intent(this, MainActivity::class.java)
    }
    startActivity(intent)
    val drawer = findViewById<DrawerLayout>(R.id.list_frag)
    drawer.closeDrawer(GravityCompat.START)
    return true
}
```

Dzięki tym implementacjom użytkownik może korzystać z szuflady nawigacyjnej do przechodzenia między różnymi aktywnościami w aplikacji.



Dodanie animacji w trakcie uruchamiania aplikacji, która ma się opierać na systemie animacji właściwości, czyli korzystać z obiektu ObjectAnimator

Dodano animację wschodzącego słońca podczas uruchamiania aplikacji. Implementacja oparta jest na systemie animacji właściwości, korzystając z obiektu ObjectAnimator. W pliku fragment_sunset.xml wyświetlone są elementy, takie jak niebo, słońce i morze.

W SunsetFragment.kt, w funkcji onCreateView, wyciągane są widoki zdefiniowane w pliku XML, a także określane są kolory, które zostaną wykorzystane w animacji.

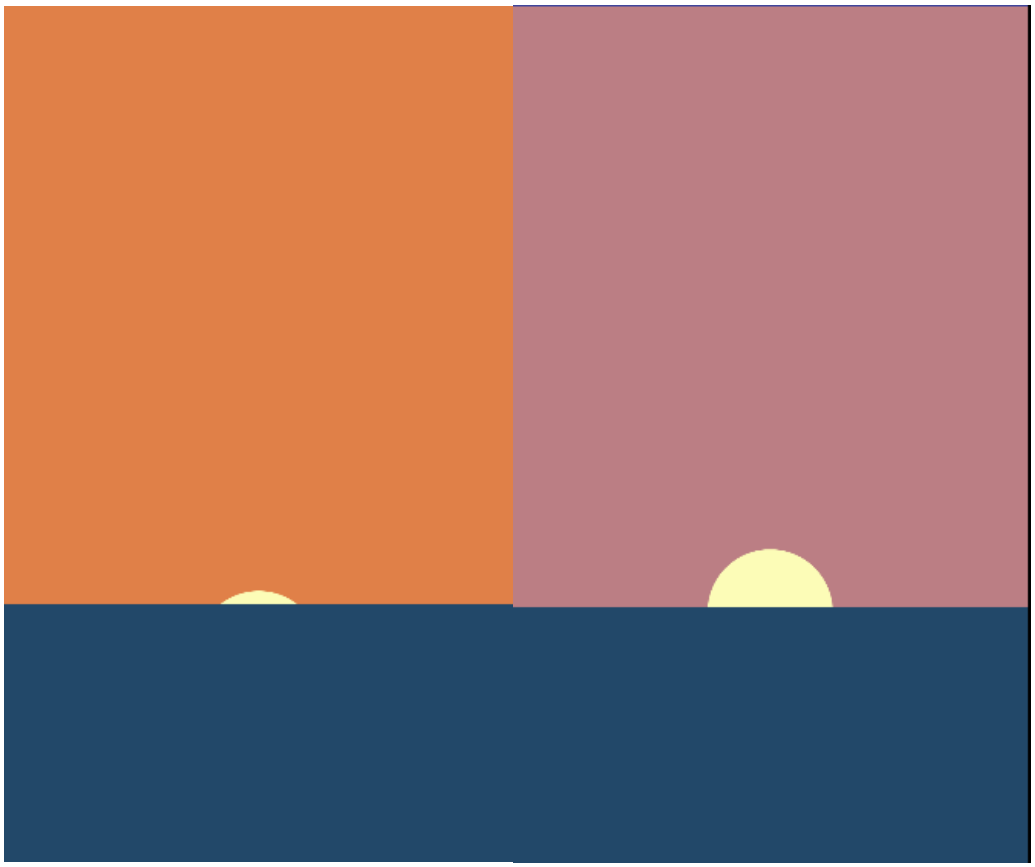
Następnie, w funkcji startAnimation(), z widoku nieba pobierana jest jego wysokość, aby określić punkt startowy i końcowy dla słońca. Wykorzystując ObjectAnimator, ustawiana jest animacja przesuwania słońca wzdłuż osi y. Dodatkowo, tworzona jest animacja zmiany koloru nieba. Obydwie animacje są uruchamiane równocześnie za pomocą AnimatorSet.

Dodano również listener nasłuchujący na zakończenie animacji. Po zakończeniu animacji, użytkownik jest przenoszony do głównej aktywności.

```
private fun startAnimation() {
    mSkyView?.post {
        val skyHeight = mSkyView?.height ?: 0
        val sunYStart = skyHeight.toFloat()
        val sunYEnd = skyHeight / 2.toFloat()

        val heightAnimator = ObjectAnimator.ofFloat(mSunView!!, "y",
sunYStart, sunYEnd)
            .setDuration(3000)
        heightAnimator.interpolator = AccelerateInterpolator()

        val sunriseSkyAnimator =
            ObjectAnimator.ofArgb(mSkyView!!, "backgroundColor",
mSunsetSkyColor, mBlueSkyColor)
                .setDuration(3000)

        val animatorSet = AnimatorSet()
        animatorSet
            .play(heightAnimator)
            .with(sunriseSkyAnimator)

        animatorSet.addListener(object : AnimatorListenerAdapter() {
            override fun onAnimationEnd(animation: Animator) {
                val intent = Intent(context, MainActivity::class.java)
                startActivity(intent)
            }
        })

        animatorSet.start()
    }
}
```

Dzięki tej implementacji, podczas uruchamiania aplikacji użytkownik widzi animację wschodzącego słońca, która następnie przenosi go do głównej aktywności.

Przykładowe zdjęcia dla telefonu:



Tablet: