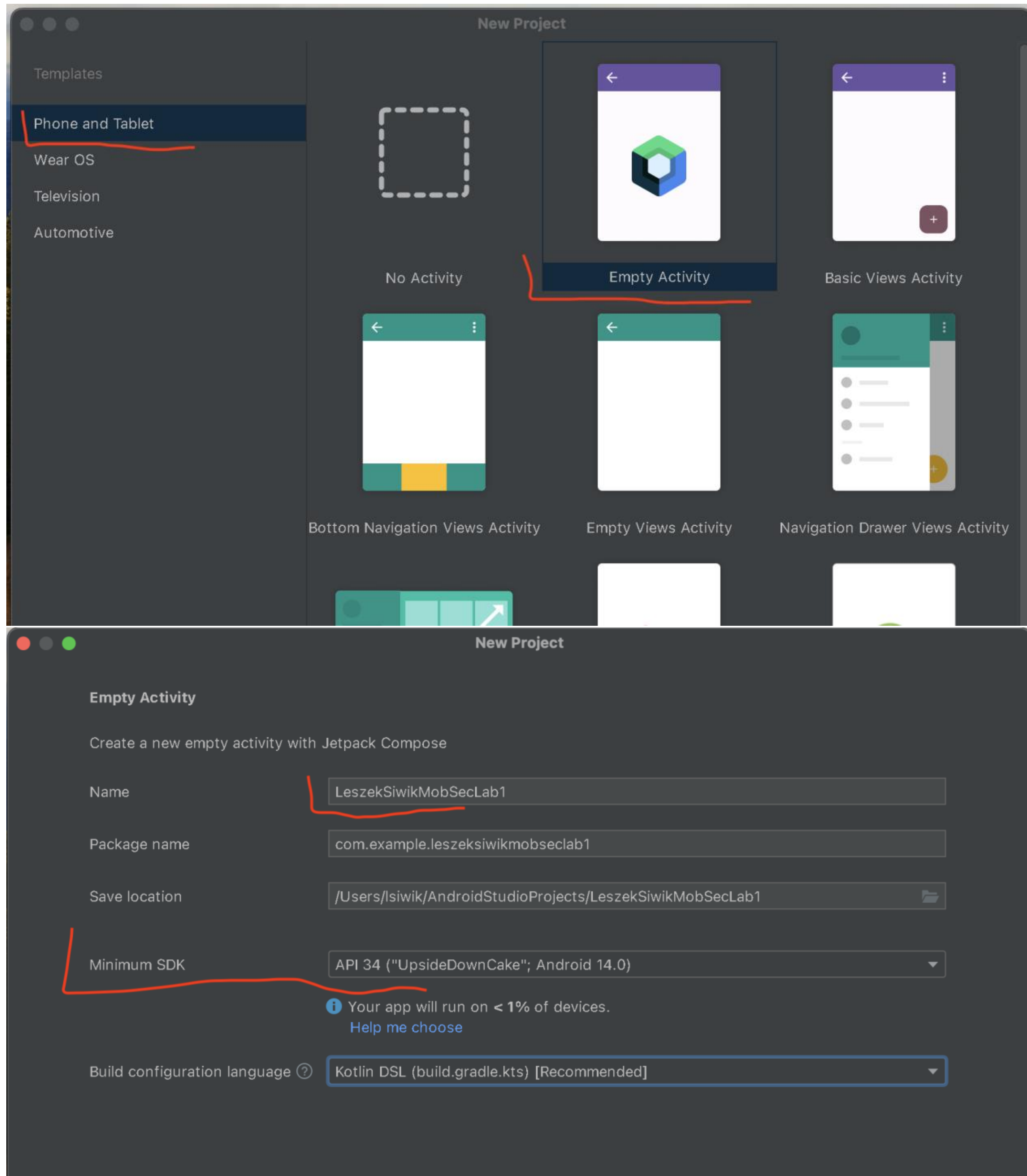
Bezpieczeństwo urządzeń mobilnych – Lab 1

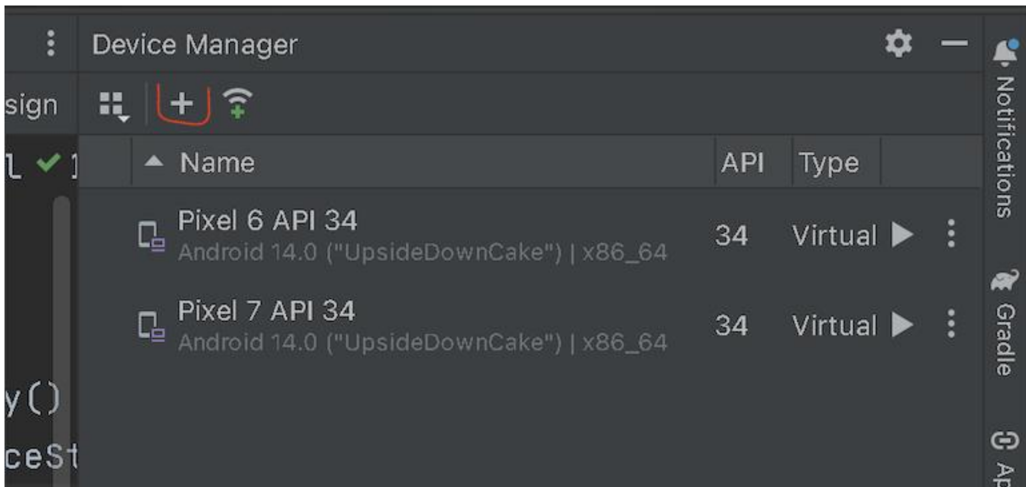
- Uruchom Android Studio
- Stworz nowy projekt File - > New Project -> Phone and Tablet
- Wybierz template Empty activity
- Nazwij project ImieNazwiskoMobSecLab1
- Minimum API Level: API 34



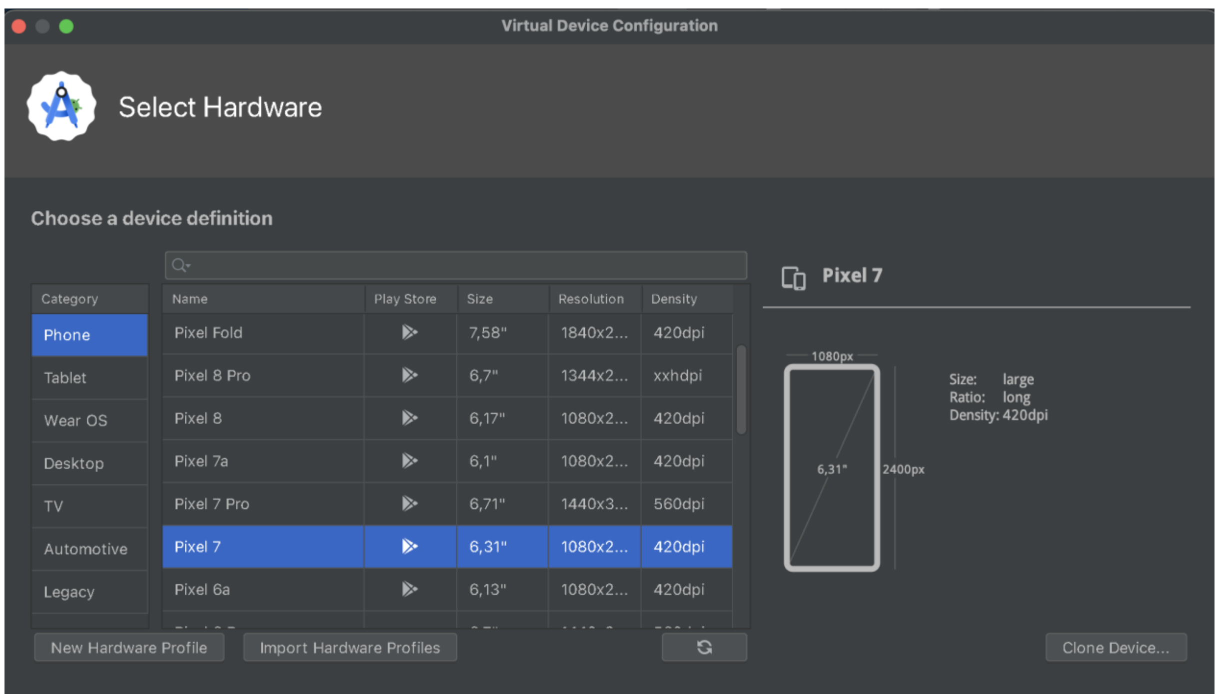
- Musimy poczekać (co niestety może zajac chwilke az Gradle wykona wszystkie czynności związane ze stworzeniem projektu

- A następnie powinnismy uzyskac srodowisko gotowe do pracy
- Zacznijmy moze od testowego uruchomienia projektu / aplikacji na emulatorze. Robimy to przez nacisniecie przycisku uruchm w gornej części studio
- (lub przez odpowiednie skroty: macOS Control + R, Windows i Linux Shift + F10. Przy czym ważne jest wskazanie emulatora na którym chcemy uruchomić naszą aplikację. Robimy to w zaznaczonym powyżej DropDown menu znajdującym się obok przycisku uruchamiania projektu. W moim przypadku wskazałem tam urządzenie Pixel 7 API 34.
- Może się zdarzyć, że nie mamy zdefiniowanego jeszcze żadnego urządzenia wirtualnego (tzw. AVD – Android Virtual Device), w takim przypadku musimy je stworzyć. Robimy to następująco.
- Klikamy w ikonkę DeviceManager:

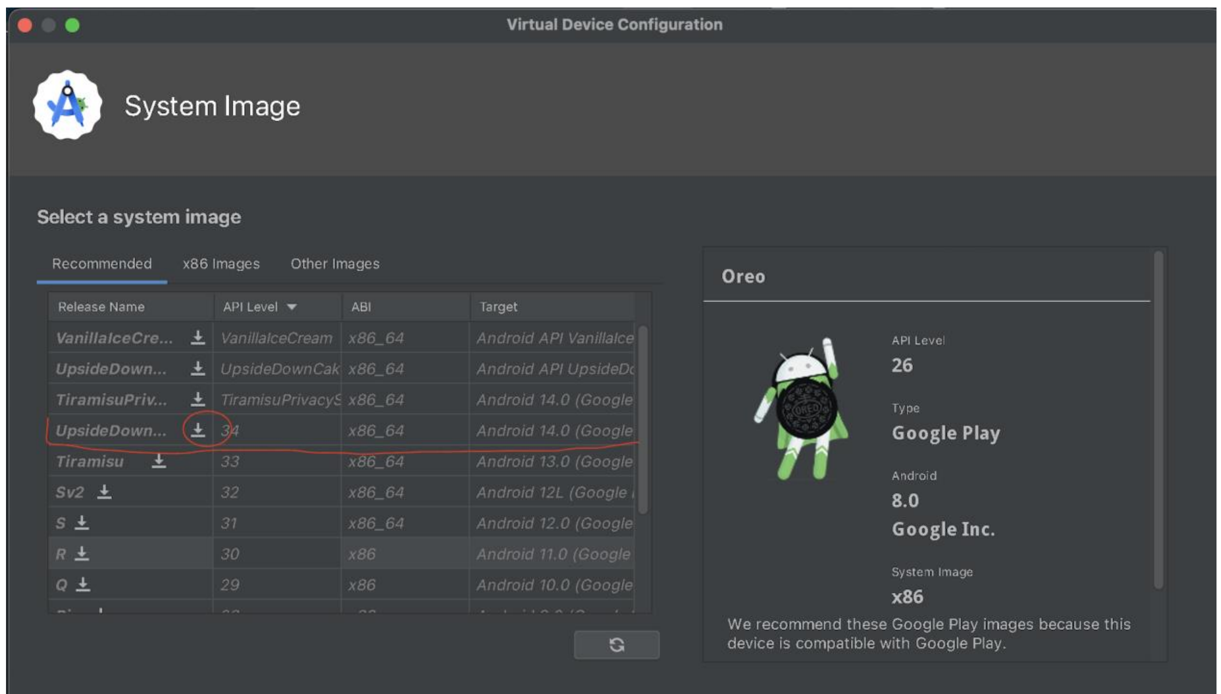
-
- Klikamy w plusik – Create Virtual Device

- 

- Na ekranie wyboru urządzenia wybierzmy Pixel 7

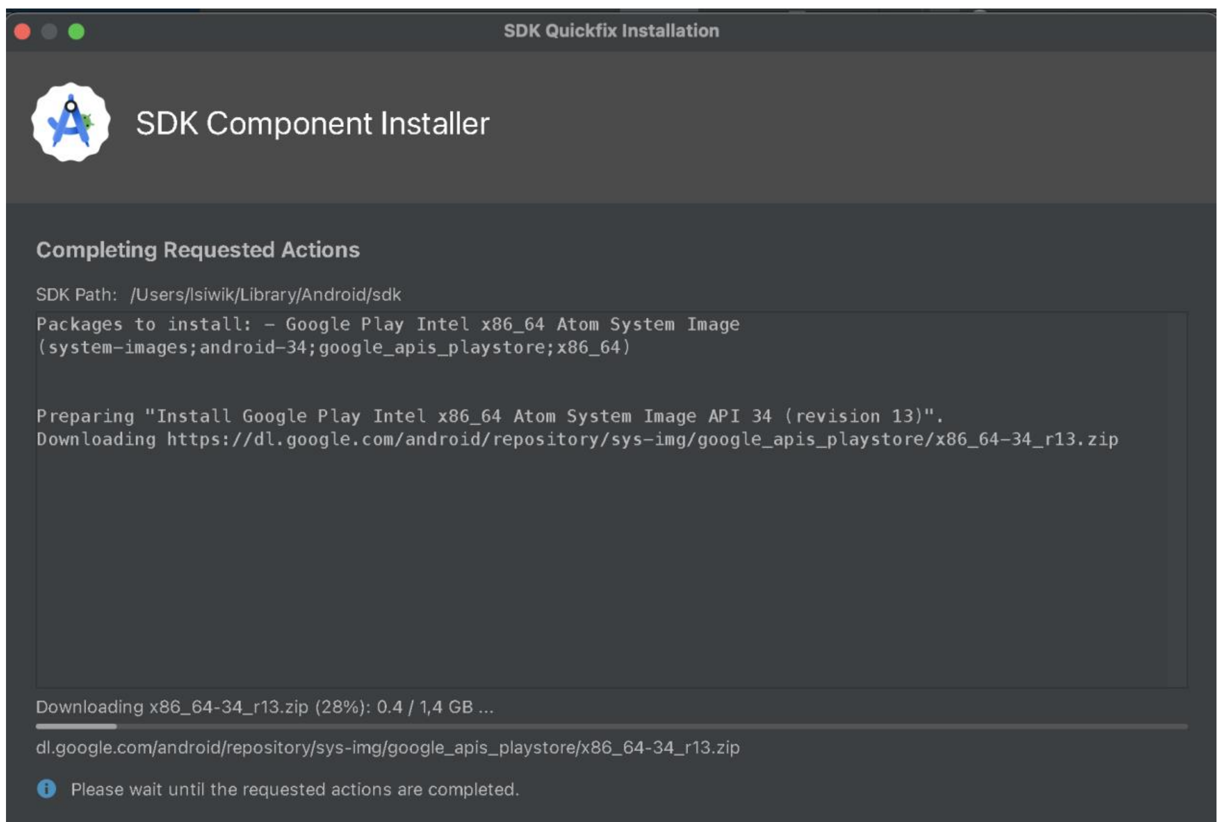
- 

- Na kolejnym ekranie (System Image) wybieramy UpsideDownCake (API Level 34)

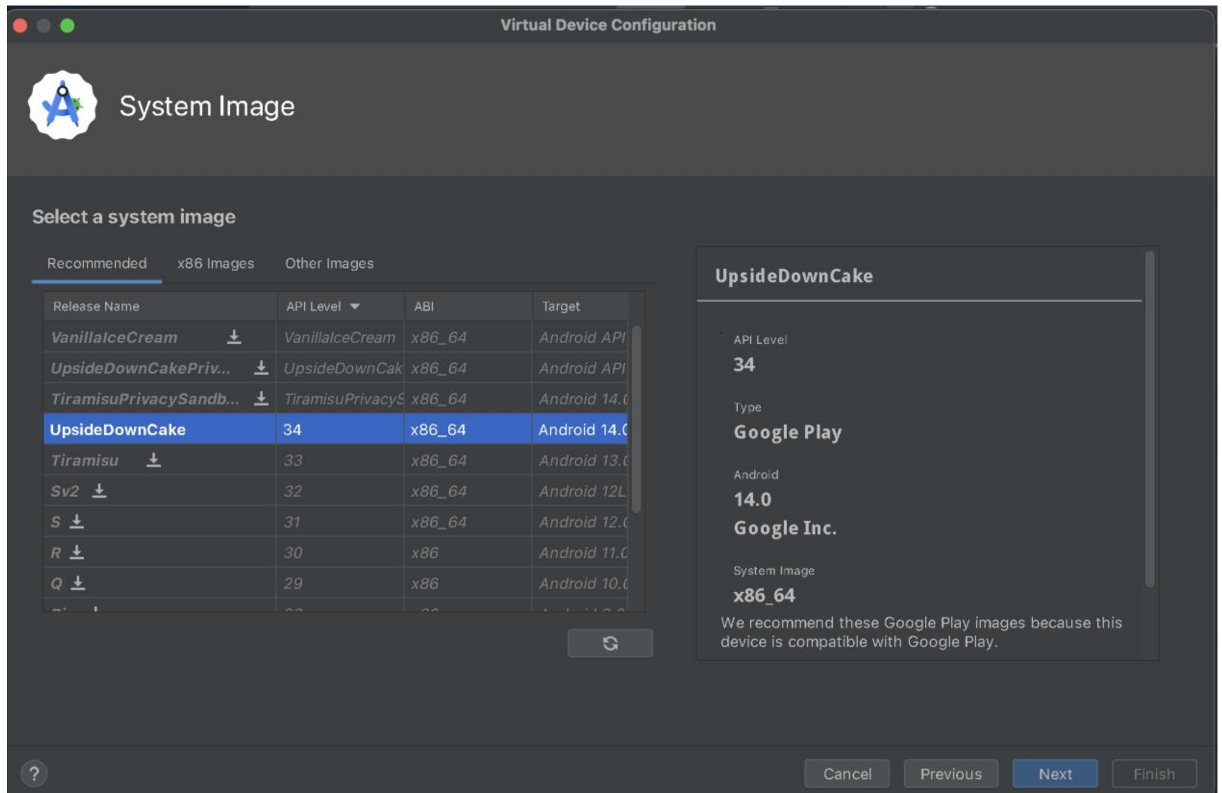
- 

- Przy czym jeśli obok nazwy widoczna jest strzalka pobierania, oznacza to ze nie mamy tego obrazu sciagnietego / dodanego do naszego środowiska i trzeba to zrobić, zatem klikamy w strzałkę pobierania.

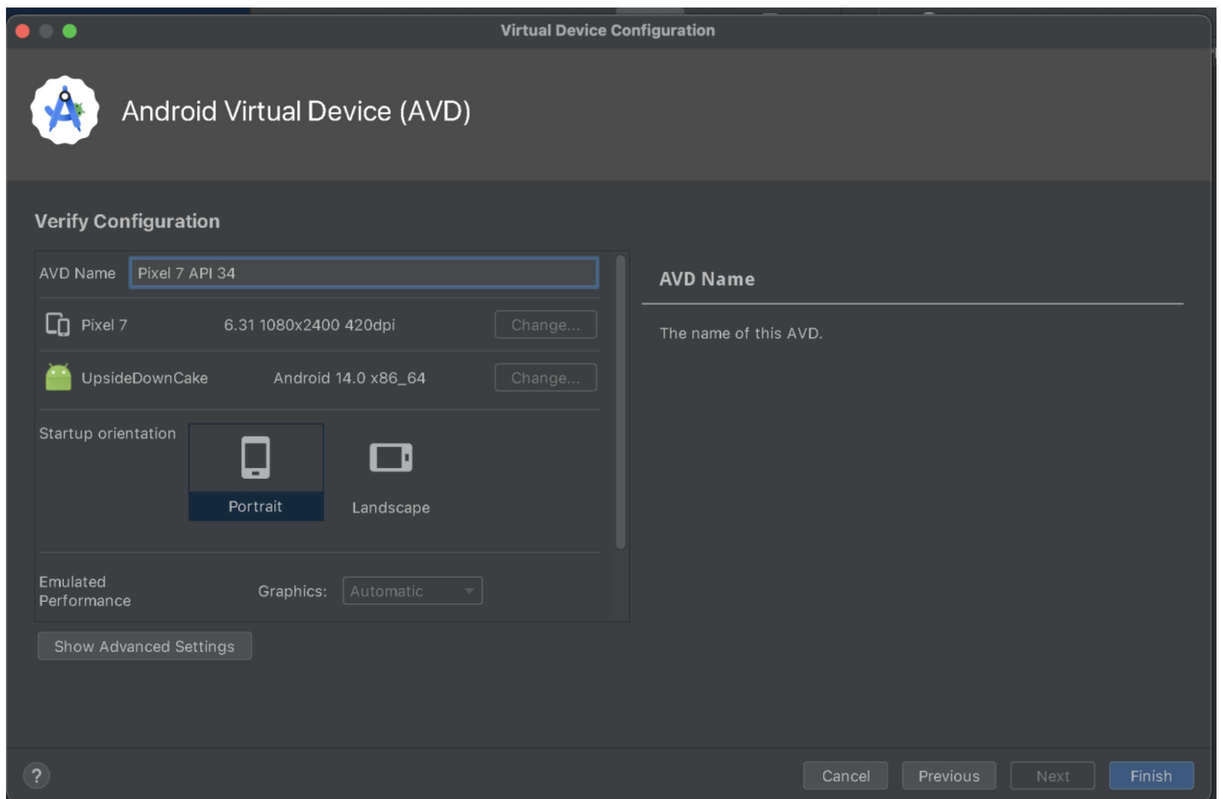
- Czekamy aż instalator wykona wszystkie swoje czynności (co może chwilke zając

- 

- Jak mu się już uda ze wszystkim uporać, klikamy w Finish i wrócimy do ekranu wyboru obrazu systemu, ale tym razem przy interesującym nas obrazie nie powinno już być strzałeczki do pobierania (tak jak u mnie poniżej), co oznacza, że interesujący nas obraz jest dostępny w środowisku i możemy go użyć:

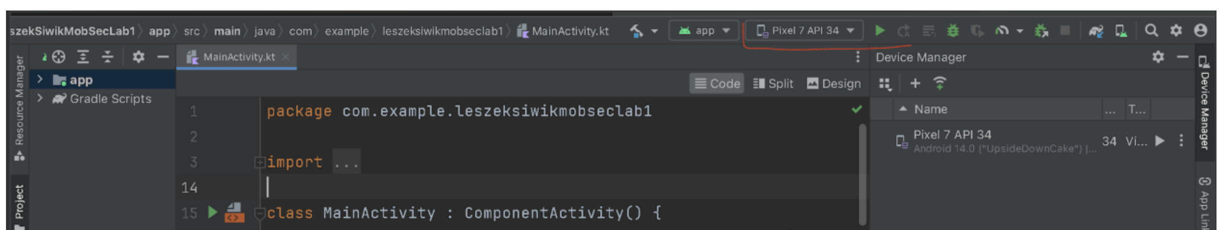
- 

- Zatem, klikamy Next. Na kolejnym ekranie możemy jeszcze dodatkowo zcustomizować nasze urządzenie wirtualne (nazwa, czy powinien domyślnie być uruchamiany w układzie poziomym czy pionowym etc.

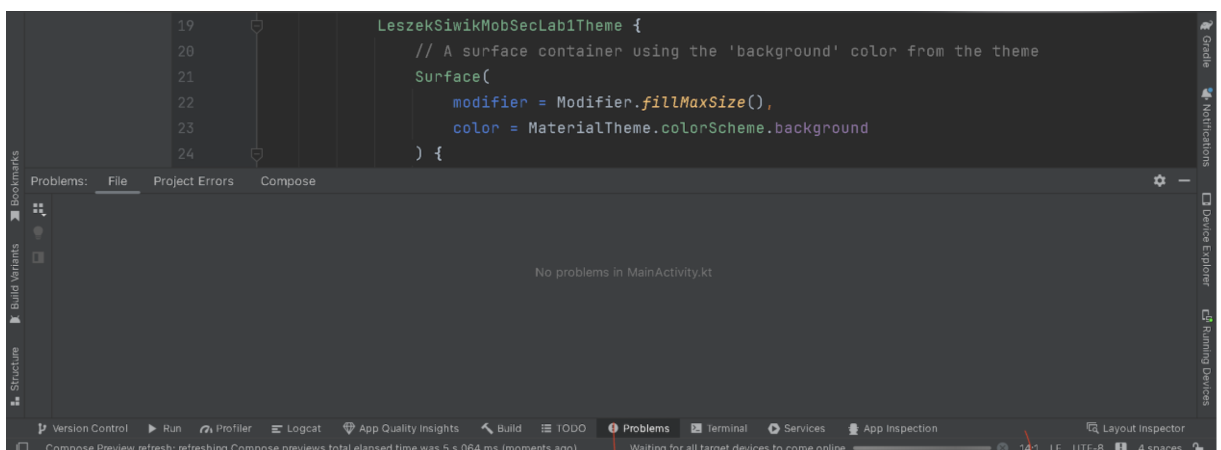
- 

- Możemy spokojnie zostawić wszystkie opcje z ustawieniami domyślnymi i klikamy Finish.

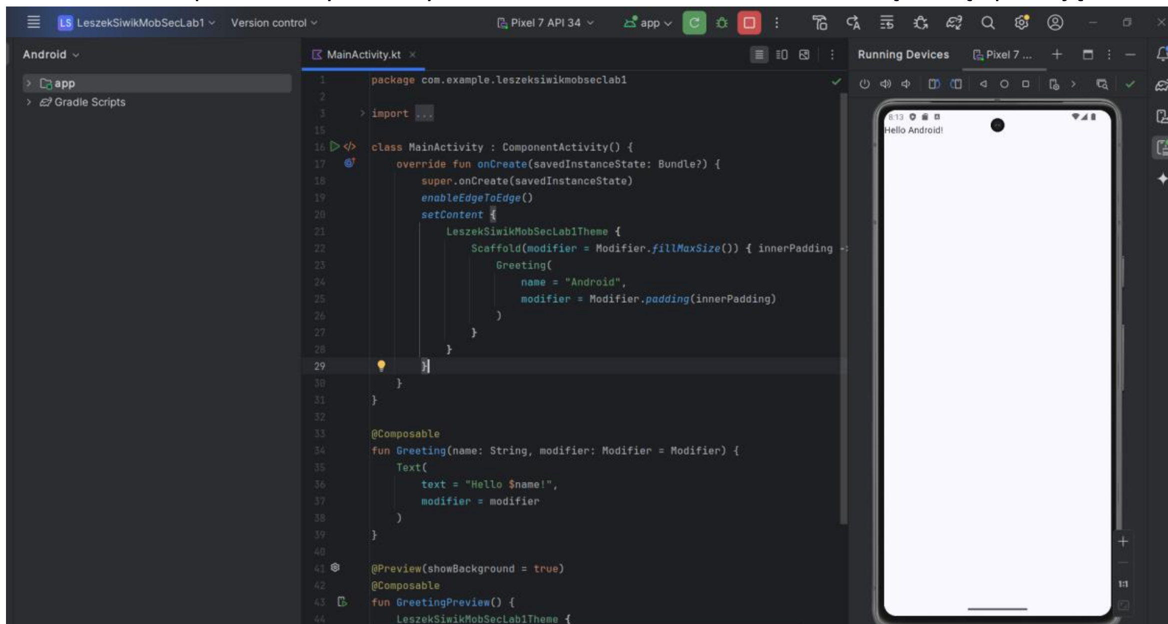
- Kiedy nam tutaj wszystko się powiedzie, nasz emulator powinien już być widoczny / dostępny drop down menu w gornej części Android Studio obok przycisku uruchamiania

- 

- No to klikamy zieloną strzałkę żeby spróbować uruchomić nasz projekt. Przy pierwszym uruchomieniu to może (i zwykle tak jest) chwilkę zająć

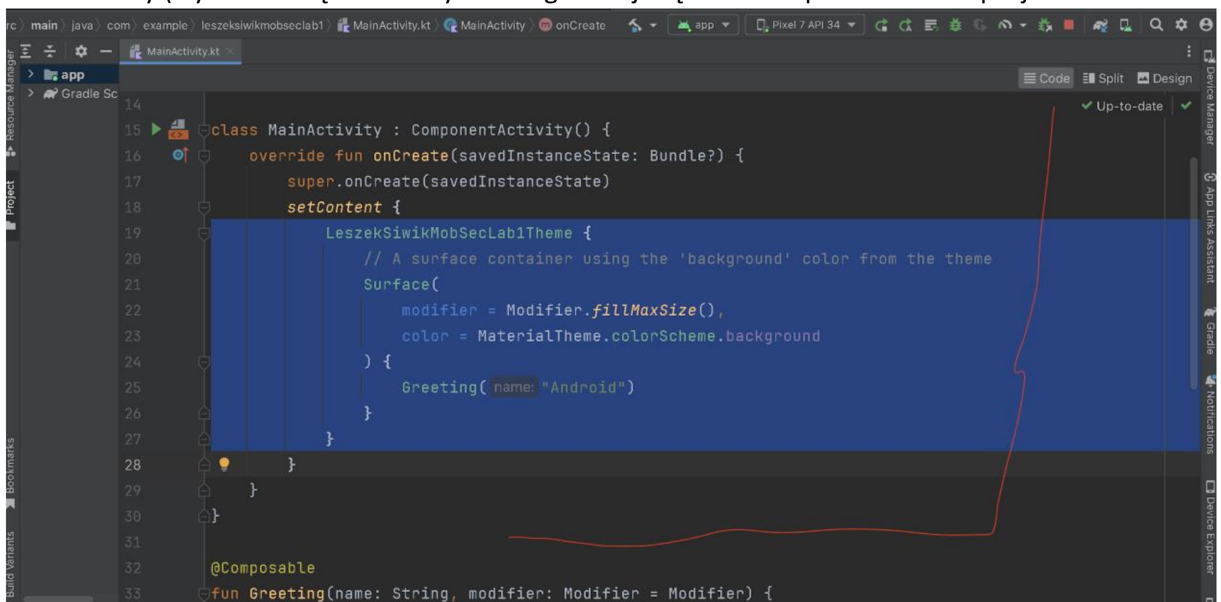
- 

- Ale ostatecznie powinniśmy zobaczyć nasz emulator a na nim uruchomioną naszą aplikację:



- Jeśli na tym etapie pojawiły się jakieś problemy / błędy – proszę o sygnał, postaram się pomóc je rozwiązać bo bez tego nie będziemy mogli iść dalej.

Praca z aplikacją - Pierwsze wprawki

- To spróbujmy popracować z naszą aplikacją. Przejdzmy do funkcji onCreate klasy MainActivity (wyświetla się ona domyślnie w głownej części Studio po stworzeniu projektu
- 

- I usuńmy zawartość funkcji setContent (czyli wszystko co podświetliłem na poprzednim zrzucie ekranu. Czyli aktualnie powinno to wyglądać następująco:

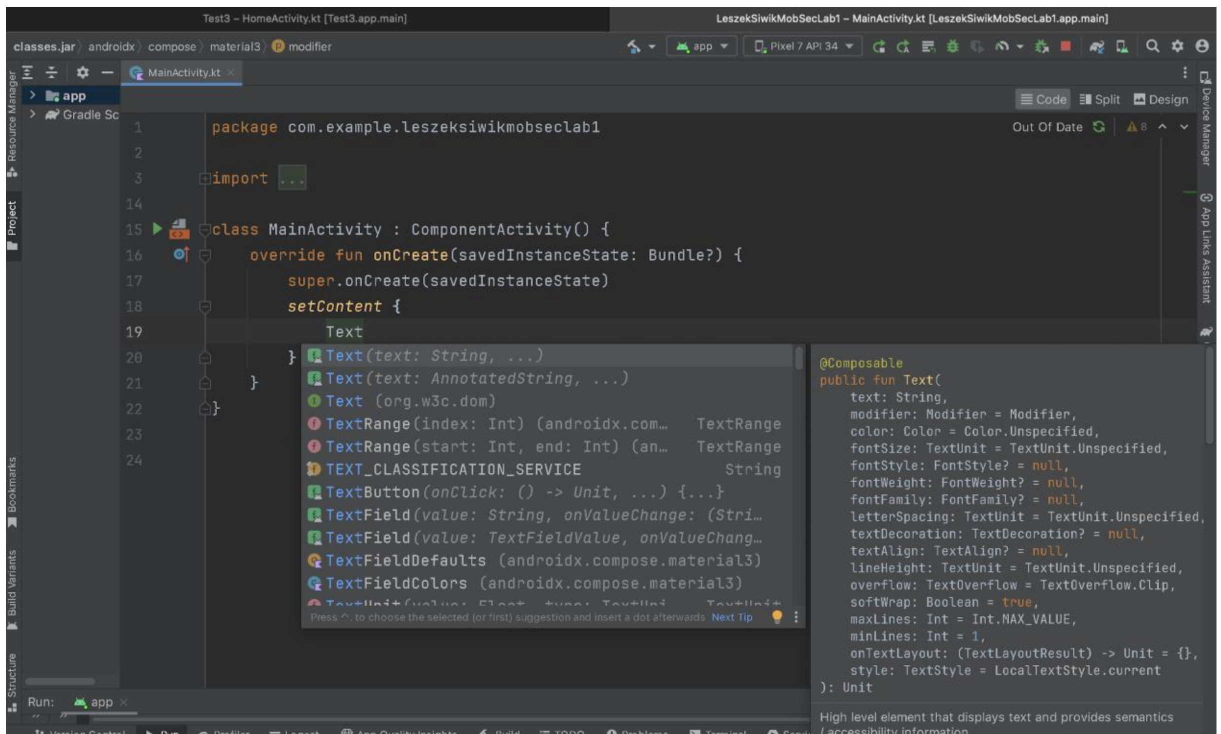
- ```
  class MainActivity : ComponentActivity() {
      override fun onCreate(savedInstanceState: Bundle?) {
  ```
  ```

```
        super.onCreate(savedInstanceState)
        setContent {

        }
    }
}
```

- Dodatkowo usuwamy (wygenerowane na potrzeby tworzonego przy zakladan iu projektu przykładu, a nam niepotrzebne funkcje Greeting oraz GreetingPreview (czyli to co podswietlilem na kolejnym zrzucie ekranu:
-
- Fiinalnie w pliku MainActivity powinnismy aktualnie mieć wyłącznie to co poniżej:
-
- I zacznijmy budować naszą własna aplikację.

- Zacznijmy może od dodania do aplikacji napisu Witaj Swiecie. Przejdzmy zatem do wnętrza funkcji setContent. Zaczynamy pisać Text Android Studio powinno podpowiedzieć nam dostępne opcje



- Chodzi nam o tę pierwsza opcję, naciskamy więc Enter i uzupełniamy zawartość atrybutu text o interesujący nas napis czyli w naszym przypadku Witaj Swiecie jak poniżej:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Text(text = "Witaj Swiecie")
        }
    }
}
```

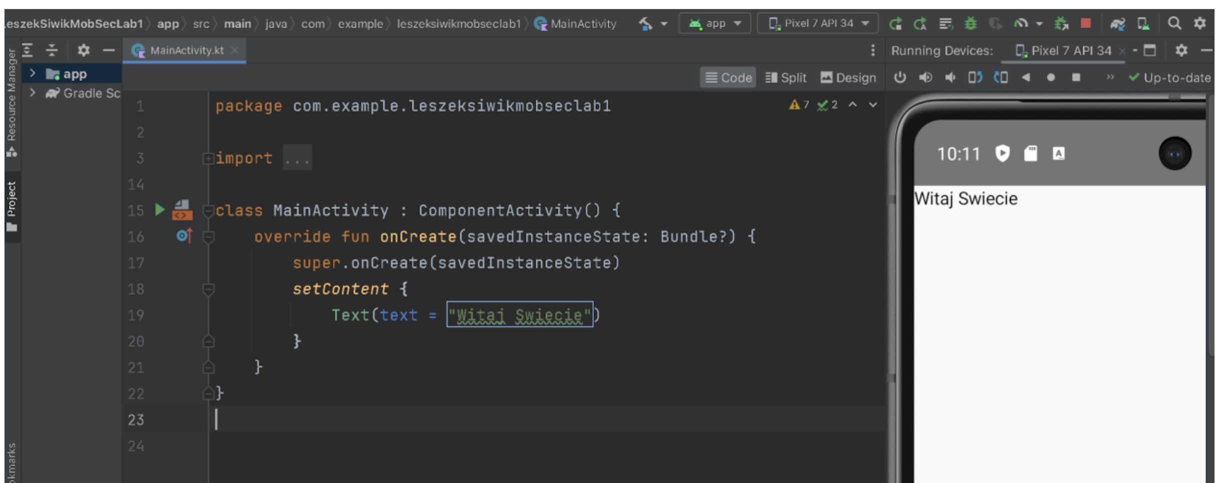
- Przebudujmy naszą aplikacje (klikamy w przycisk gdzie poprzednio była zielona strzalka do uruchamiania projektu

- ```
  package com.example.leszeksiwikmobseclab1

  import ...

  class MainActivity : ComponentActivity() {
      override fun onCreate(savedInstanceState: Bundle?) {
          super.onCreate(savedInstanceState)
          setContent {
              Text(text = "Witaj Swiecie")
          }
      }
  }
  ```

- I na emulatorze powinnismy zobaczyć na pierwszy napis:

- 
  ```

- Jak popatrzymy na parametrzy funkcji Text:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setCon
            Te
        }
    }
}
```

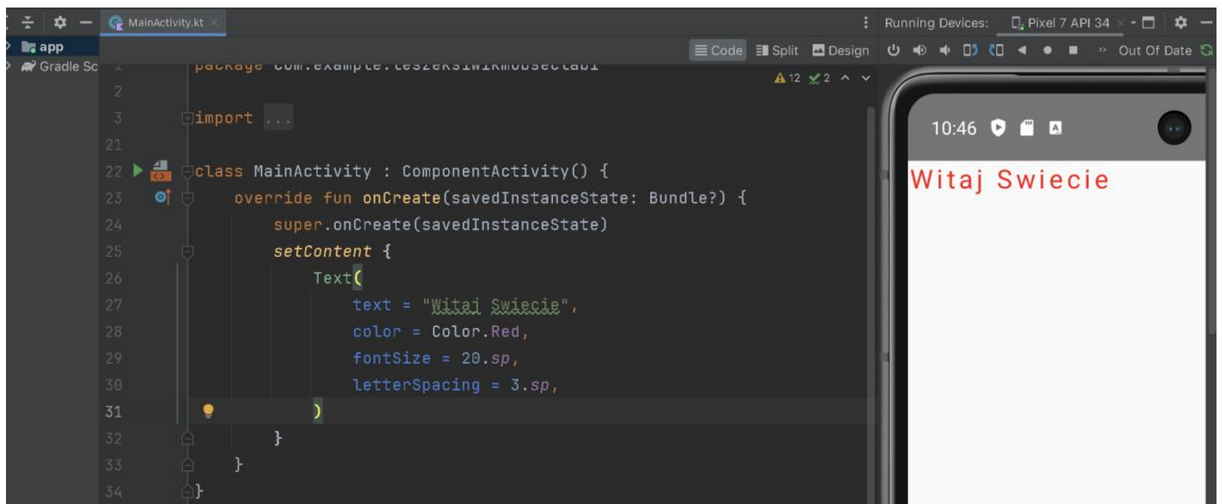
```
@Composable
public fun Text(
    text: String,
    modifier: Modifier = Modifier,
    color: Color = Color.Unspecified,
    fontSize: TextUnit = TextUnit.Unspecified,
    fontStyle: FontStyle? = null,
    fontWeight: FontWeight? = null,
    fontFamily: FontFamily? = null,
    letterSpacing: TextUnit = TextUnit.Unspecified,
    textDecoration: TextDecoration? = null,
    textAlign: TextAlign? = null,
    lineHeight: TextUnit = TextUnit.Unspecified,
    overflow: TextOverflow = TextOverflow.Clip,
    softWrap: Boolean = true,
    maxLines: Int = Int.MAX_VALUE,
    minLines: Int = 1,
    onTextLayout: (TextLayoutResult) -> Unit = {},
    style: TextStyle = LocalTextStyle.current
): Unit
```

High level element that displays text and provides semantics / accessibility information.
The default style uses the LocalTextStyle provided by the MaterialTheme / components. If you are setting your own style, you may want to consider first retrieving LocalTextStyle, and using TextStyle.copy to keep any theme defined attributes, only modifying the specific attributes you want to override.

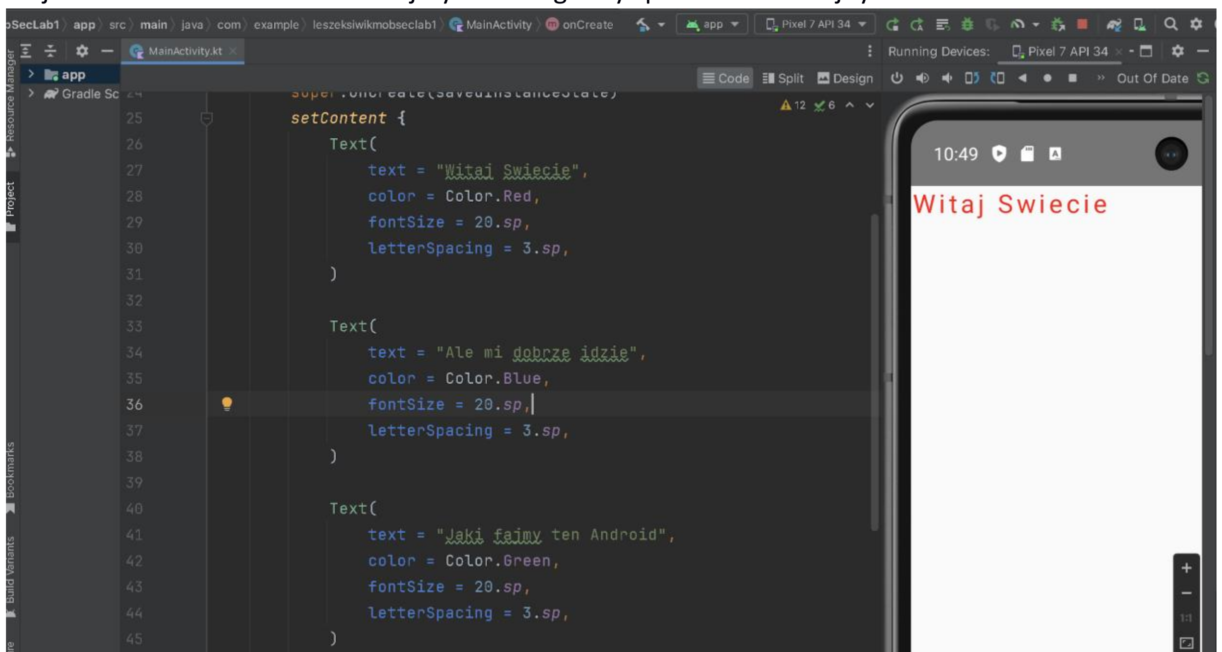
Profiler   Logcat
186 ms. (2 minutes ago)

- Poza parametrem text pozwalającym ustawić „zawartość”, jest tam sporo innych parametrów „formatujących” nasz napis. No to spróbujmy skorzystać z kilku. np. zmienmy kolor czcionki na czerwony (ustawiamy parametr color na Color.Red), zwiększmy rozmiar czcionki (ustawiamy parametr fontSize np. na 20.sp) i dodajmy odstępy pomiędzy znakami napisu czyli ustawiamy parametr letterSpacing np. na 3.sp. Czyli finalnie nasz Text konstruujemy aktualnie w następujący sposób:

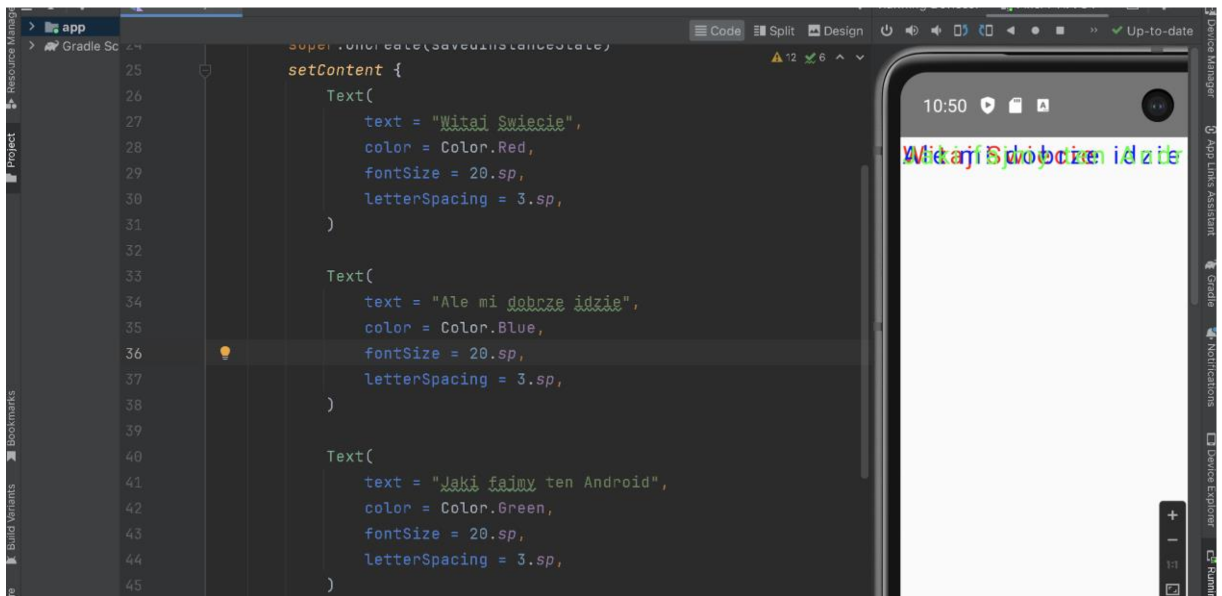
- ```
  Text(
      text = "Witaj Swiecie",
      color = Color.Red,
      fontSize = 20.sp,
      letterSpacing = 3.sp,
  )
  ```
- No to zobaczmy jaki to przynosi efekt:
  ```

- 

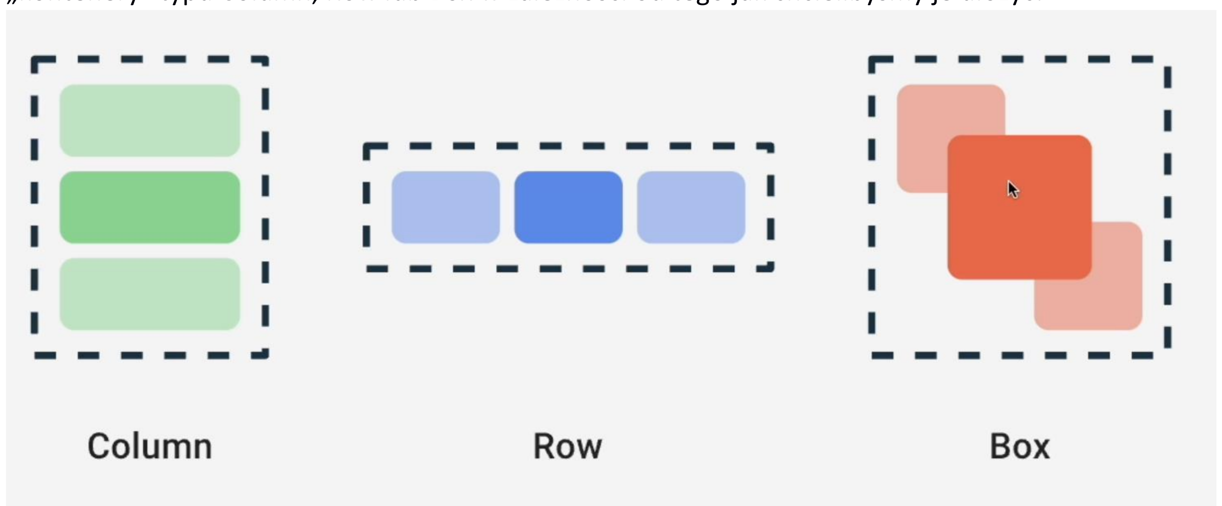
- No i pięknie
- To jeśli idzie nam tak dobrze dodajmy w analogiczny sposób kilka kolejnych elementów Text:



- Przebudujmy aplikacje i zobaczmy co mamy:



-
-
- Problem wynika z tego ze poniewaz nie dodalismy informacji jak te kolejne elementy maja być spozycjonowane na ekranie, no to wszystkie spozycjonowaly się domyślnie czyli w lewym gornym rogu aplikacji i wzajemnie na siebie nachodzą.
- Najprostszym sposobem rozwiązania problemu jest „opakowanie" naszych napisow w „kontenery" typu Column, Row lub Box w zależności od tego jak chcielibyśmy je ułozyc:



-
- Założmy ze chcielibyśmy umieścic je jeden pod drugim, użyjemy zatem kontenera Column.
- No to wracamy do naszej funkcji setContent, przed naszym pierwszym elementem tekstowym dodajemy pustą linijkę i zaczynamy pisać Column. Android Studio pokaże nam dostępne opcje

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column
            Column(...) {...}
            ColumnScope (androidx.compose.foundation.layout)
            Columns (android.provider.SyncStateContract)
            Columns (androidx.core.provider.FontsContractCo…
            Columns (android.provider.FontsContract)
            FlowColumn(...) {...}
            LazyColumn(...) {...}
            Text(
                text = "Ale mi dobrze idzie",
                color = Color.Blue,
                fontSize = 20.sp,
                letterSpacing = 3.sp,
            )

            Text(
                text = "Jaki fajny ten Android",
                color = Color.Green,
```

- Interesuje nas pierwsza opcja, zatem naciskamy enter i przenosimy nasze Teksty do wnętrza naszej kolumny:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column { this: ColumnScope
                Text(text = "Witaj Swiecie")
                Text(text = "Witajcie Wszyscy")
                Text(text = "Jaki fajny ten Android")
            }
        }
    }
}
```

- Przebudujmy nasza aplikacje i sprawdzmy jak to wygląda teraz:

- ```kotlin
  super.onCreate(savedInstanceState)
  setContent {
      Column {
          Text(
              text = "Witaj Swiecie",
              color = Color.Red,
              fontSize = 20.sp,
              letterSpacing = 3.sp,
          )

          Text(
              text = "Ale mi dobrze idzie",
              color = Color.Blue,
              fontSize = 20.sp,
              letterSpacing = 3.sp,
          )

          Text(
              text = "Jaki fajny ten Android",
              color = Color.Green,
              fontSize = 20.sp,
              letterSpacing = 3.sp,
          )
  ```


- Zdecydowanie lepiej
- A gdybyśmy chcieli „rozrzucić” nasze napisy równomiernie po ekranie jak poniżej:

-
  ```

- Jak popatrzymy na argumenty funkcji Column:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Col
            ) {
```

```
@Composable
public inline fun Column(
    modifier: Modifier = Modifier,
    verticalArrangement: Arrangement.Vertical = Arrangement.Top,
    horizontalAlignment: Alignment.Horizontal = Alignment.Start,
    content: @Composable() (ColumnScope.() -> Unit)
): Unit
```

A layout composable that places its children in a vertical sequence. For a layout composable that places its children in a horizontal sequence, see Row. Note that by default items do not scroll; see Modifier.verticalScroll to add this behavior. For a vertically scrollable list that only composes and lays out the currently visible items see LazyColumn.

The Column layout is able to assign children heights according to their weights provided using the ColumnScope.weight modifier. If a child is not provided a weight, it will be asked for its preferred height before the sizes of the children with weights are calculated proportionally to their weight based on the remaining available space. Note that if the Column is vertically scrollable or part of a vertically scrollable container, any provided weights will be disregarded as the remaining available space will be infinite.

When none of its children have weights, a Column will be as small as possible to fit its children one on top of the other. In order to change the height of the Column, use the Modifier.height modifiers; e.g. to make it fill the available height Modifier.fillMaxHeight can be used. If at least one child of a Column has a weight, the Column will fill the available height, so there is no need for Modifier.fillMaxHeight. However, if Column's size should be limited, the

- To widać tam m.in. parametr verticalArrangement odpowiedzialny pionowy układ elementów w kolumnie. No to spróbujmy go użyć:

```
import ...

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column(
                verticalArrangement = Arrangement.SpaceBetween
            ) { this: ColumnScope

                Text(
                    text = "Witaj Swiecie",
                    color = Color.Red,
                    fontSize = 20.sp,
                    letterSpacing = 3.sp,
                )

                Text(
                    text = "Ale mi dobrze idzie",
                    color = Color.Blue,
```

- Przebudujmy aplikacje i zobaczmy co to dało:

```
import ...

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column(
                verticalArrangement = Arrangement.SpaceBetween
            ) { this: ColumnScope

                Text(
                    text = "Witaj Swiecie",
                    color = Color.Red,
                    fontSize = 20.sp,
                    letterSpacing = 3.sp,
                )

                Text(
                    text = "Ale mi dobrze idzie",
                    color = Color.Blue,
                    fontSize = 20.sp,
```

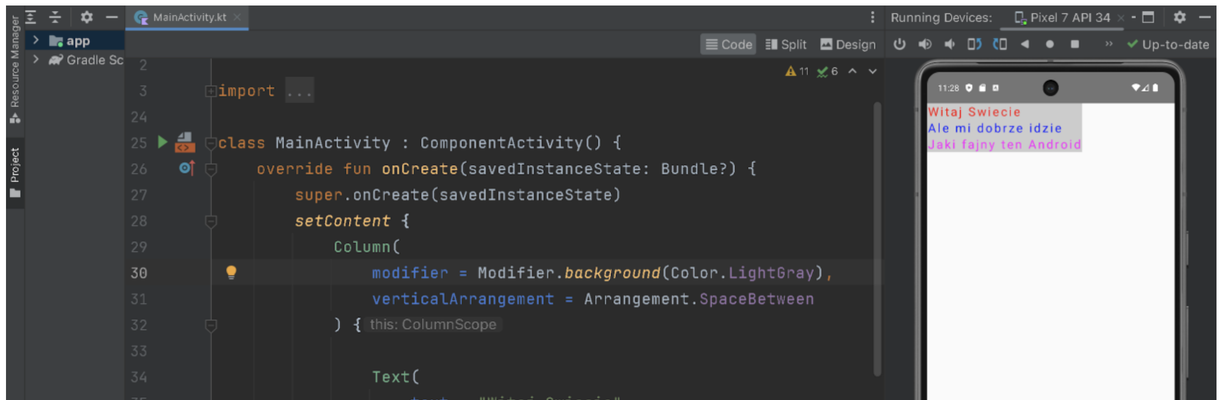
```
Witaj Swiecie
Ale mi dobrze idzie
Jaki fajny ten Android
```

- Hm. Mimo że opcja .SpaceBetween powinna załatwić sprawę, póki co niewiele nam to dało. No to przyjrzyjmy się jak aktualnie spozycjonowana jest na ekranie sama Kolumna. Najprościej będzie to zrobić ustawiając jej jakiś kolor tła i będziemy dokładnie widzieć co tam się dzieje.
- Funkcja Column nie posiada explicite parametry typu background czy backgroundColor ale posiada tzw. Modyfikator (parametr modifier) za pośrednictwem którego możemy modyfikować / ustawiać wiele parametrów danego elementu które nie zostały wyciągnięte jako wprost widoczne parametry wywołania danego elementu. I jest to dość typowa sytuacja. Zwykle dla danego elementu (graficznego) jako dostępne explicite wyciągnięte są najczęściej wykorzystywane parametry (jak np. rozmiar czy kolor czcionki dla napisu) a te rzadziej wykorzystywane dostępne są poprzez modyfikator.
- No to ustawmy tło dla naszej Kolumny:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column(
                modifier = Modifier.background(Color.LightGray),
                verticalArrangement = Arrangement.SpaceBetween
            ) { this: ColumnScope

                Text(
                    text = "Witaj Swiecie",
                    color = Color.Red,
```

- I zobaczmy jak spozycjonowana jest nasz kolumna na ekranie:

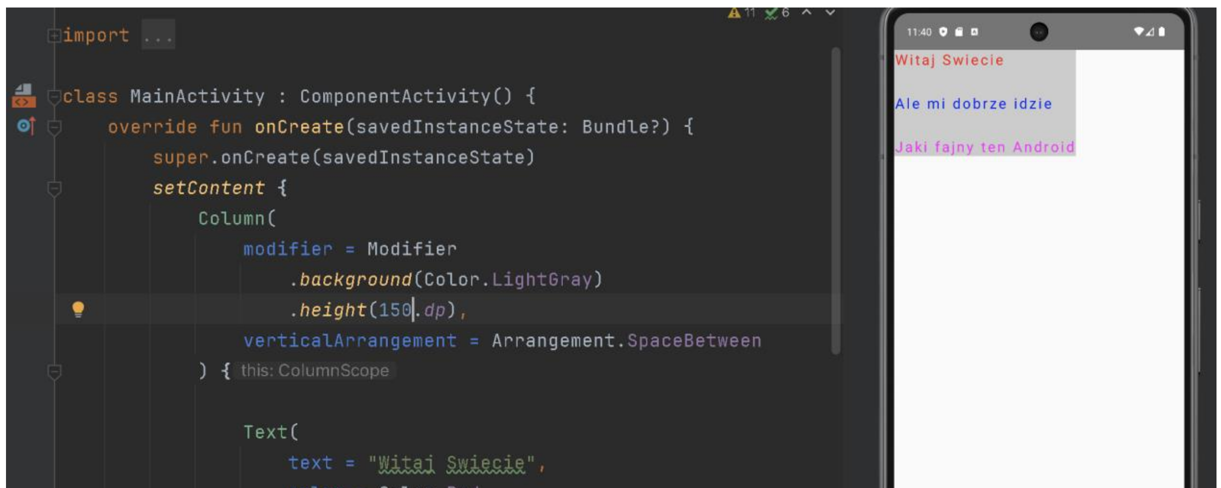
- 

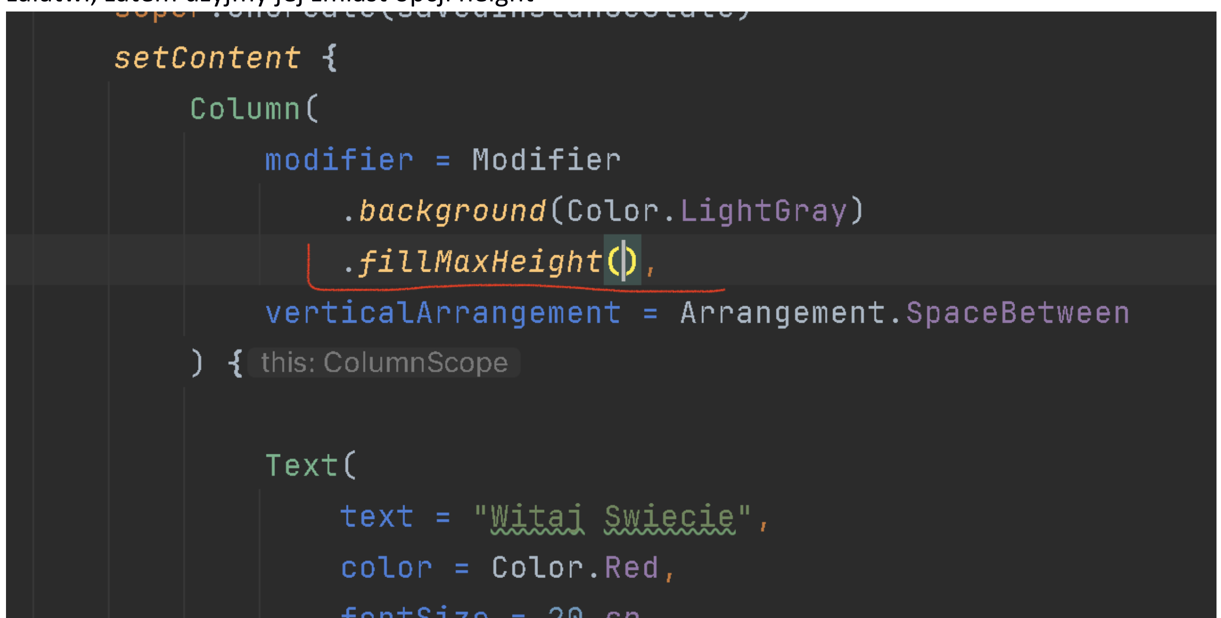
- Ano właśnie. Jak widać, jeśli nie zdefiniujemy tego inaczej domyślny rozmiar kolumny (zarówno wysokość jak i szerokość) to minimalne rozmiary pozwalające pomieścić umieszczone w niej elementy. A jeśli tak to opcja .SpaceBetween nie ma szans się wykazać bo aktualnie w kolumnie nie ma po prostu miejsca aby umieszczone w niej napisy jakos „rozrzucić".
- No to zwiększmy wysokość kolumny i zobaczmy czy Spacebetween będzie miał okazje się „popracować". Ponownie. Atrybutu height nie widać wprost wśród argumentów funkcji Column, wiec zrobimy to dodając odpowiiednią (kolejną) opcje dla parametru modifier np. jak poniżej:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column(
                modifier = Modifier
                    .background(Color.LightGray)
                    .height(150.dp),
                verticalArrangement = Arrangement.SpaceBetween
            ) { this: ColumnScope

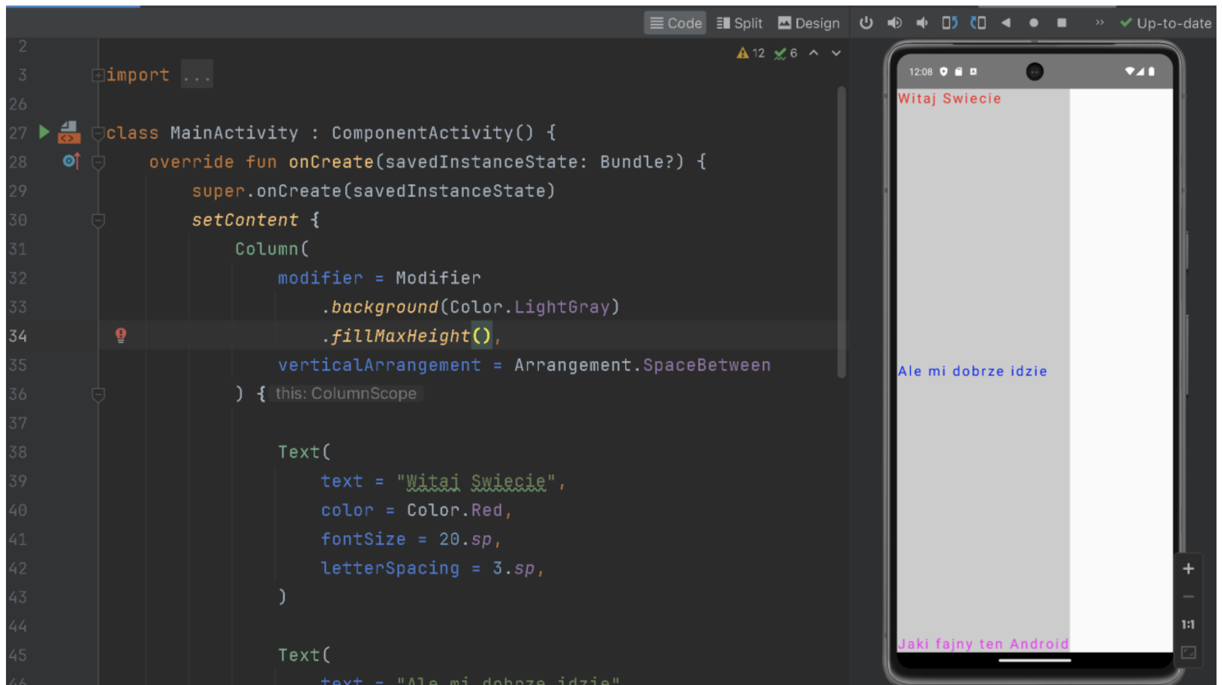
                Text(
                    text = "Witaj Swiecie",
                    color = Color.Red,
                    fontSize = 20.sp,
                    letterSpacing = 3.sp,
                )
```

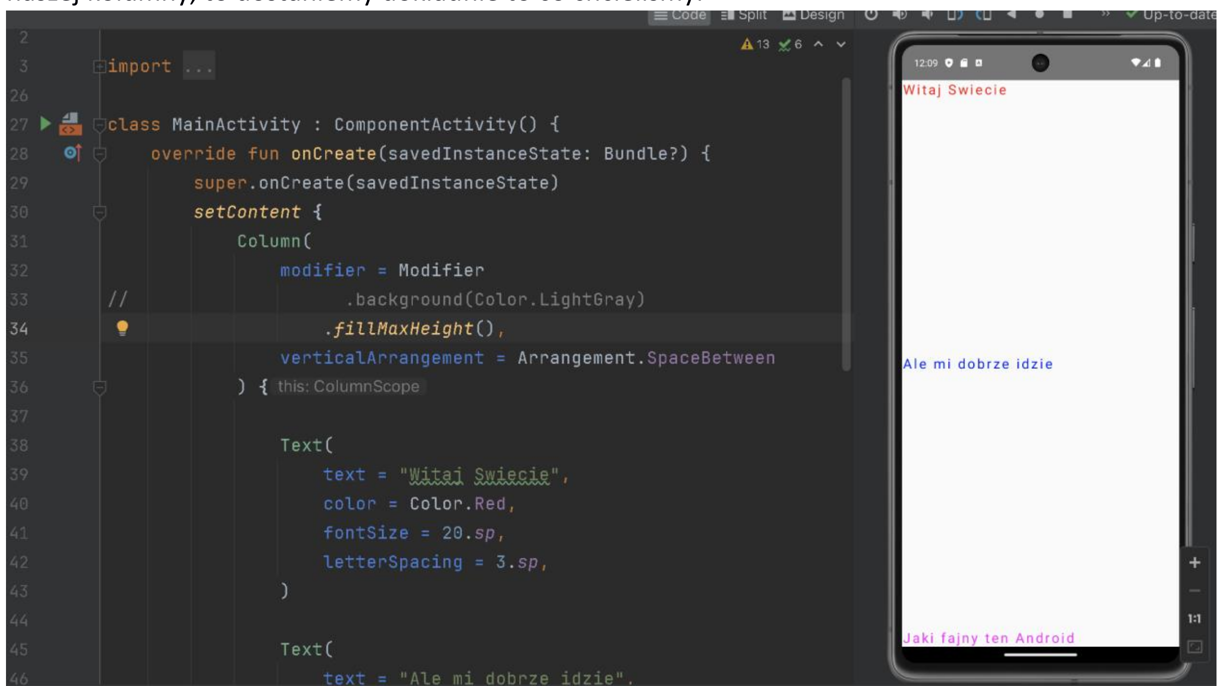
- I zobaczmy jak to teraz wygląda:

- 

- Ano właśnie, więc wygląda, że działa. No więc gdybyśmy ustawili wysokość kolumny rowna wysokości ekranu telefonu to powinnismy uzyskac to co chcemy. Podając jakąs konkretną wartość w parametrze .height modifiera potencjalnie nadziejemy się na dwa problemy: jaka przyjąć tam wartość a dwa czy będzie to tak samo wyglądać na telefonach o różnym rozmiarze ekranu. Na szczęście jedną z pcji modifiera jest .fillMaxHeight która to za nas załatwi, zatem użyjmy jej zmiast opcji height
- 

- I zobaczmy jak to aktualnie wygląda:

- 

- Ano prawie tak jak chcieliśmy. Jeśli dodfatkwo usuniemy nasz „debugowy” kolor tła dla naszej kolumny, to dostaniemy dokładnie to co chcieliśmy.
- 

- To w ramach wprawek, spróbuj uzyskać następujące ułożenie naszych napisów:

-

- Albo, żeby było ładniej z małym odsunięciem od krawędzi telefonu:
-

- Proszę o sygnał, jak będzie zrobione
- Zanim przejdziemy dalej, uporządkujmy sobie trochę nasz kod.
    - Po pierwsze poprawmy ewentualne niedoformatowania (Mac: Cmd + Option + L , Windows / Linux: Ctrl + Alt + L).
    - Dodatkowo wydzielmy sobie to co aktualnie zrobiliśmy do osobnej funkcji, dzięki czemu zrobimy sobie miejsce na kolejne eksperymenty w naszym setContent. Załóżmy że nasza funkcja będzie się nazywała MyTexts
    - Idziemy zatem w naszym edytorze za ostatni zamykający nawias klamrowy i piszemy: `fun MyTexts(){}`
    - Teraz do środka nawiasów klamrowych przenosimy z funkcji setContent definicję naszej kolumny z trzema textami. Po tym przeniesieniu zacznieMyTexts i Column zacznie nam się podkreślać na czerwono a to dlatego że funkcje korzystające z elementów @Composable (a takimi są zarówno Column jak i Text czy generalnie inne elementy interfejsu użytkownika z których będziemy korzystać) same muszą być oznaczone / adnotowane jako funkcje typu Composable – stąd bezpośrednio nad definicją naszej funkcji dodajemy adnotację @Composable

```
@Composable
fun MyTexts() {
    Column(
        modifier = Modifier
```

- Jak widać to usunęło nam błędy. Możemy sobie teraz zwinąć definicję / zawartość naszej funkcji żeby nie zabierała nam miejsca w edytorze / nie utrudniała poruszania się po nim klikając w zaznaczony niżej element:

```
@Composable
fun MyTexts() {
    Column(
        modifier = Modifier
            .background(Color.LightGray)
            .fillMaxSize()
```

- Wreszcie, możemy sobie dodać wywołanie naszej funkcji MyTexts z funkcji setContent i mamy w miare fajnie uporządkowany nasz kod:

- Możemy sobie teraz zakomentować wywołanie funkcji MyTexts dzięki czemu nie będzie ona wykonywana / nie będzie rzutowała na to co mamy / widzimy w samej aplikacji, a w każdej chwili, w razie potrzeby będziemy mogli to wywołanie odkomentować i uruchomić
- P.S Wydzielenie, które właśnie zrobiliśmy można oczywiście zrobić automatycznie (zaznaczamy fragment kodu który chcemy wydzielić, prawy klik -> Refactor -> Function / Extract to Function nadajemy nazwę naszej nowej funkcji i zrobione. Ale ponieważ się uczymy chciałem żebyśmy zrobili to „ręcznie”.

- Kolejnym elementem UII którego użycie przećwiczymy, a który pozwoli nam na wykonywanie „akcji” jest Button. Idziemy zatem do naszego setContent, zaczynamy pisać Button, Studio pokaże nam dostępne opcje

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            // MyTexts()
            Butt
        }
    }
}
@Composab
fun MyTex
```

```
Button(onClick: () -> Unit, ...) {...}
ButtonDefaults (androidx.compose.material3)
Button (android.widget)
ButtonColors (androidx.compose.material3)
ButtonElevation (androidx.compose.material3)
ElevatedButton(onClick: () -> Unit, ...) {...}
ExtendedFloatingActionButton(onClick: () -> U…
ExtendedFloatingActionButton(text: () -> Unit…
FilledIconButton(onClick: () -> Unit, ...) {…
FilledIconToggleButton(checked: Boolean, onCh…
FilledTonalButton(onClick: () -> Unit, ...) {…
Press ⏎ to insert, ⇥ to replace Next Tip
```

```
@Composable
public fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    shape: Shape = ButtonDefaults.shape,
    colors: ButtonColors = ButtonDefaults.buttonCol
    elevation: ButtonElevation? = ButtonDefaults.bu
    border: BorderStroke? = null,
    contentPadding: PaddingValues = ButtonDefaults.
    interactionSource: MutableInteractionSource = r
    content: @Composable() (RowScope.() -> Unit)
): Unit
```

Material Design button.
Buttons help people initiate actions, from sending an email, to sharing a document, to liking a post.
!Filled button image

- Interesuje nas pierwsza opcja, naciskamy więc enter i zaczynamy pracować z naszym przyciskiem:

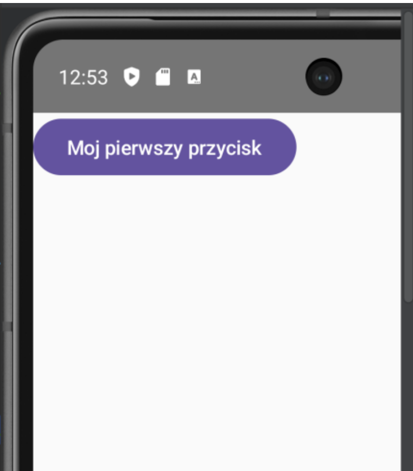
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Button(onClick = { /*TODO*/ }) { this: RowScope

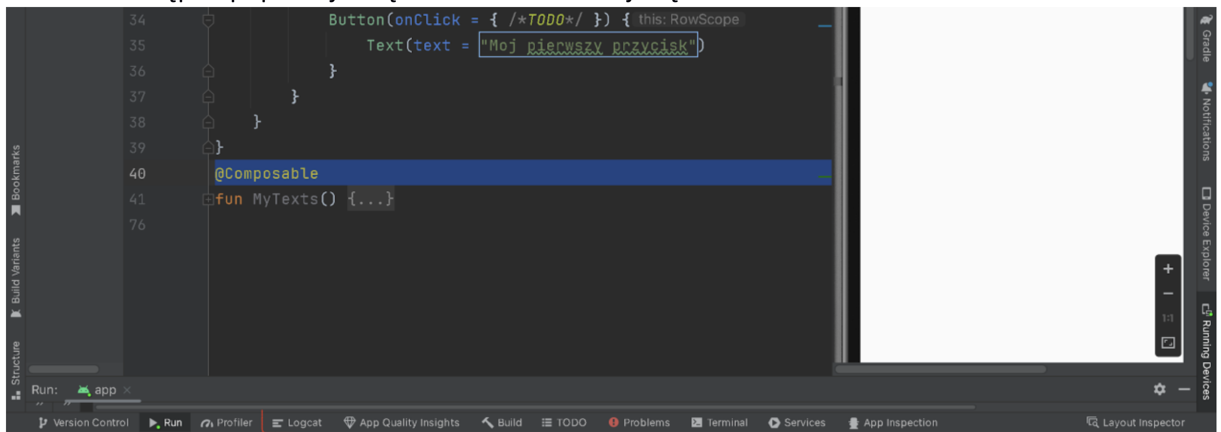
            }
        }
    }
}
@Composable
fun MyTexts() {...}
```

- Zacznujmy może od dodania na przycisku jakiego napisu (znanego nam już elementu Text) żebyśmy wiedzieli co to za przycisk / za co on odpowiada. No więc dodajmy na początek napis Mój pierwszy przycisk, i zobaczmy co w ogóle dostajemy. No więc powinniśmy dostać coś takiego:

- ```kotlin
  package com.example.leszeksiwikmobseclab1

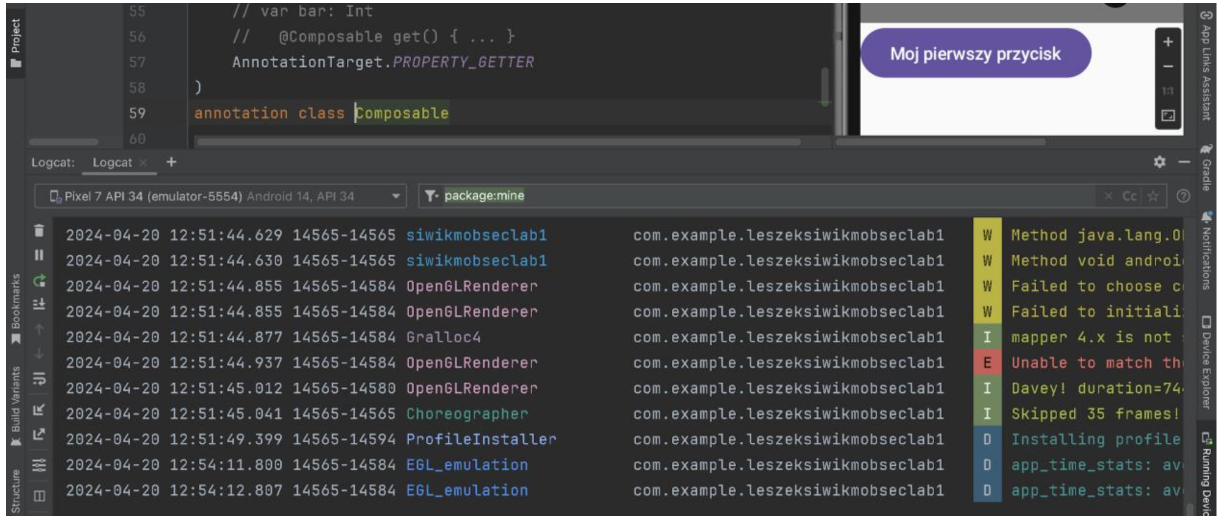
  import ...

  class MainActivity : ComponentActivity() {
      override fun onCreate(savedInstanceState: Bundle?) {
          super.onCreate(savedInstanceState)
          setContent {
              Button(onClick = { /*TODO*/ }) {
                  Text(text = "Moj pierwszy przycisk")
              }
          }
      }
  }
  @Composable
  fun MyTexts() {...}
  ```
  

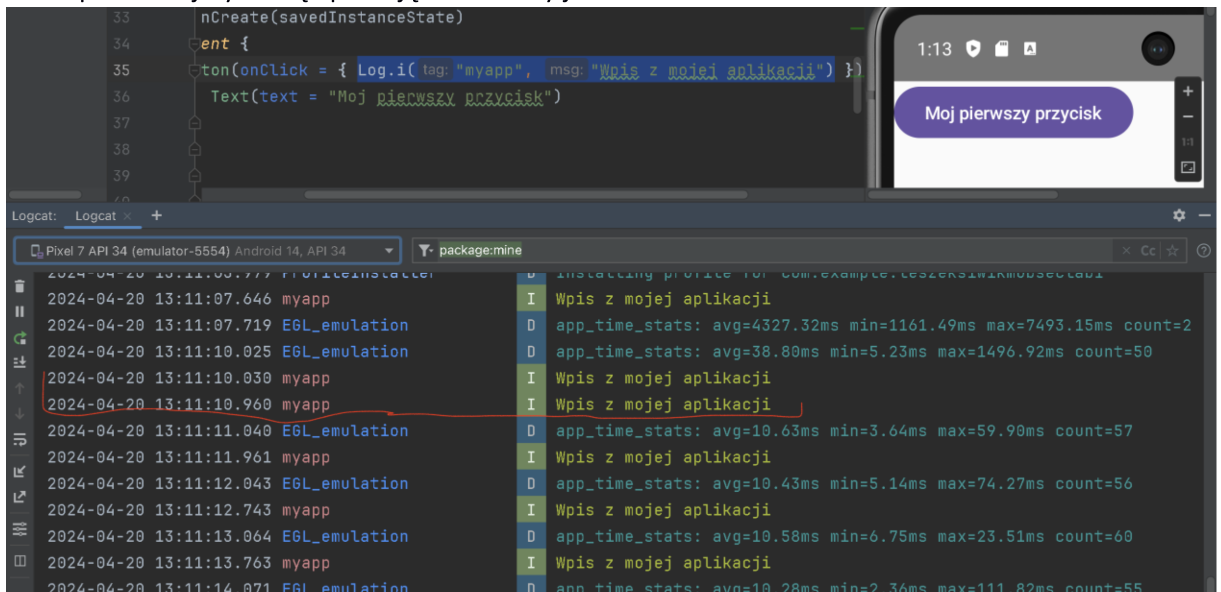
- Przycisk oczywiście jest „klikalny" tylko na razie nie robi (a wlasciwie to aplikacja po nacisnieciu na niego nie robi nic bo jak widać zawartość atrybutu onClick jest pusta i zawiera jedynie komentarz jest to ciągle dopiero do zaimplementowania. No to spróbujmy wykonać jakąś akcje.
- Spróbujmy może na początek zapisać jakąś wiadomość/treść w Logach naszego urządzenia / emulatora. Pierwsza istotna sprawa to jak w ogóle dostać się do logów urządzenia i móc je przeglądać. Otóż w Studio dostępne mamy narzędzie LogCat które dokładnie do tego służy. Jest ono dostępne poprzez jedną z zakładek w dolnej części Android Studio:
  

- Otwórzmy je i po chwili potrzebnej na nawiązanie połączenia z urządzeniem / emulatorem i zainicjalizowanie całego mechanizmu / narzędzia powinniśmy zobaczyć logi z naszego
  ```

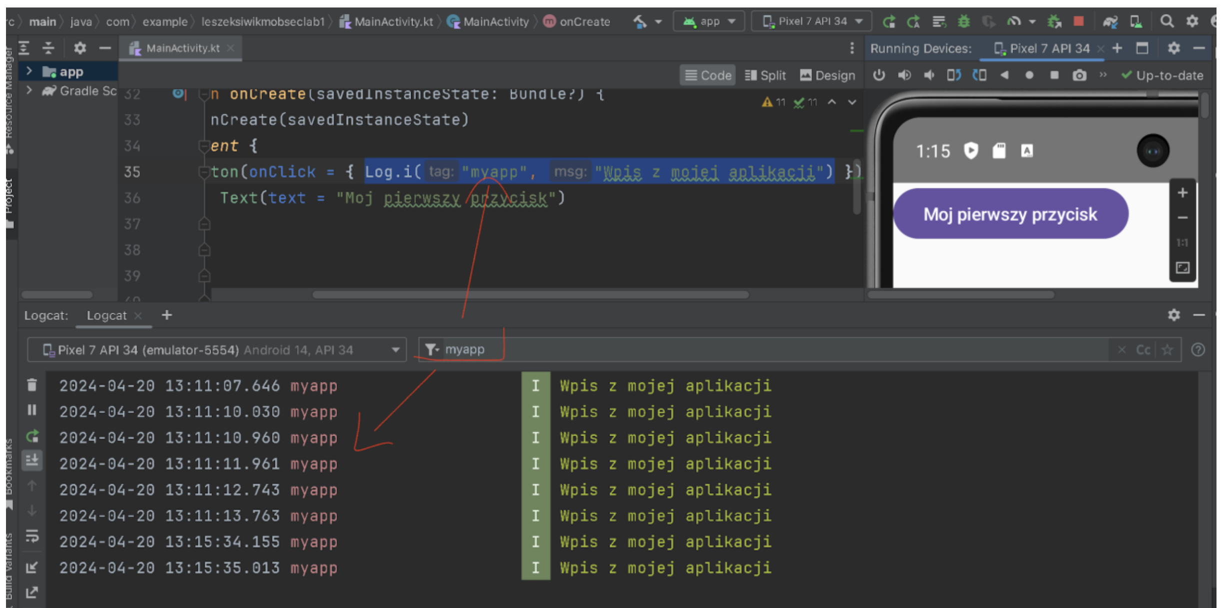
emulatora:

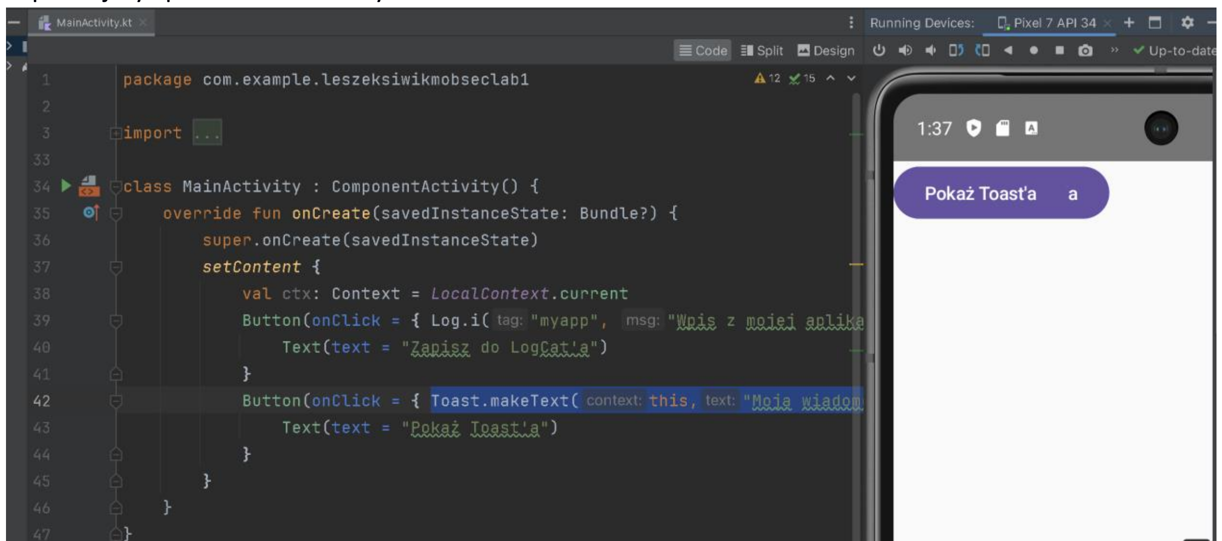


- To jak wiemy gdzie szukać ewentualnych wpisów do logów, to spróbujmy rzeczywiście coś tam wpisać. Wracamy zatem do definicji naszego przycisku, idziemy do definicji parametru onClick i komentarz /*TODO*/ zastępujemy przez: Log.i("myapp", "Wpis z mojej aplikacji")
- Ten kawałek kodu powinien dodać nam w logach wpis „Wpis z mojej aplikacji” otagowany tagiem „myapp” (dzięki czemu łatwiej jest wyszukać w LogCat’cie interesujące nas Logi.
- No to przebudujmy naszą aplikację i zobaczmy jak to działa:

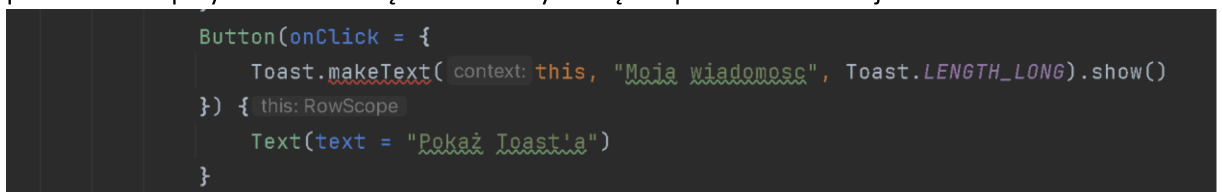


- No i wygląda że „coś” działa i po każdym kliknięciu w przycisk pojawia się w logach nowy wpis.
- Możemy dla ułatwienia przefiltrować wyjście LogCata po naszym tagu:

- 

- Ok, umiemy zatem zapisać cos do logów. To spróbujmy wyświetlić cos w samej aplikacji. Zanim przejdziemy jednak dalej zmieńmy może napis na naszym pierwszym przycisku na „Zapisz do LogCat'a), żeby było wiadomo co on robi / który jest który. Dodajmy teraz drugi przycisk, ustawmy mu napis „Pokaz Toast'a" a jako obsługę atrybutu onClick dodajmy następujący kawałek kodu:
- Toast.makeText(this,"Moja wiadomosc", Toast.*LENGTH_LONG*).show()
- I sprobujmy sprawdzic co mamy
- 

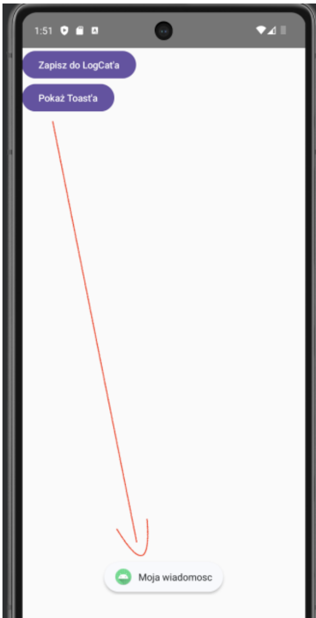
- No tak. Mamy na razie drobny bałagan, bo przyciski nałozyły się na siebie. Ale to już wiemy jak sobie z tym poradzić. Zatem opakujmy je kolumną I sprawdzmy. Hm, u mnie po przeniesieniu przycisków do wnętrza kolumny zaczął mi podkreślać funkcje makeText



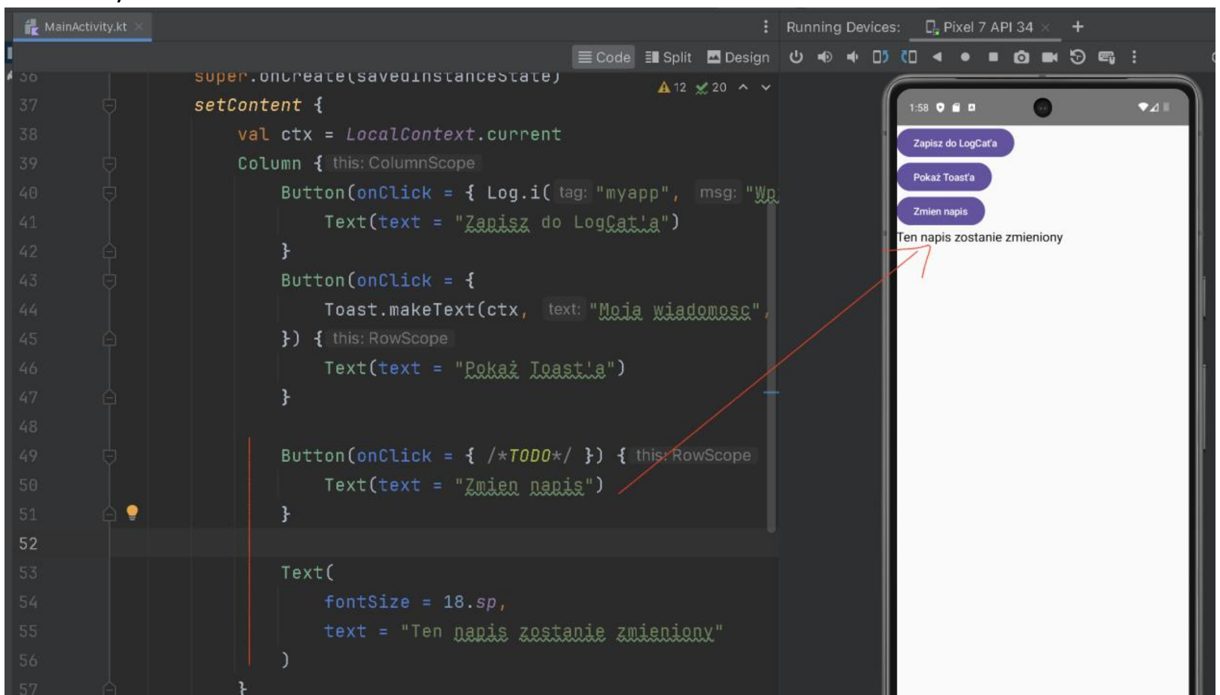
- To dlatego że funkcja ta wymaga podania jako pierwszy argument aktywność w kontekście której (czy też na której) toast będzie wyświetlany i o ile przed obudowaniem kolumną mogliśmy podać ten argument jako „this" to po wstawieniu do kolumny this nie oznacza już

kontekstu aktywności tylko kontekst Columny a Toast nie może się wyświetlić w kontekście Columny. Zapamiętajmy sobie zatem kontekst aktywności gdzieś przed rozpoczęciem columny jako:

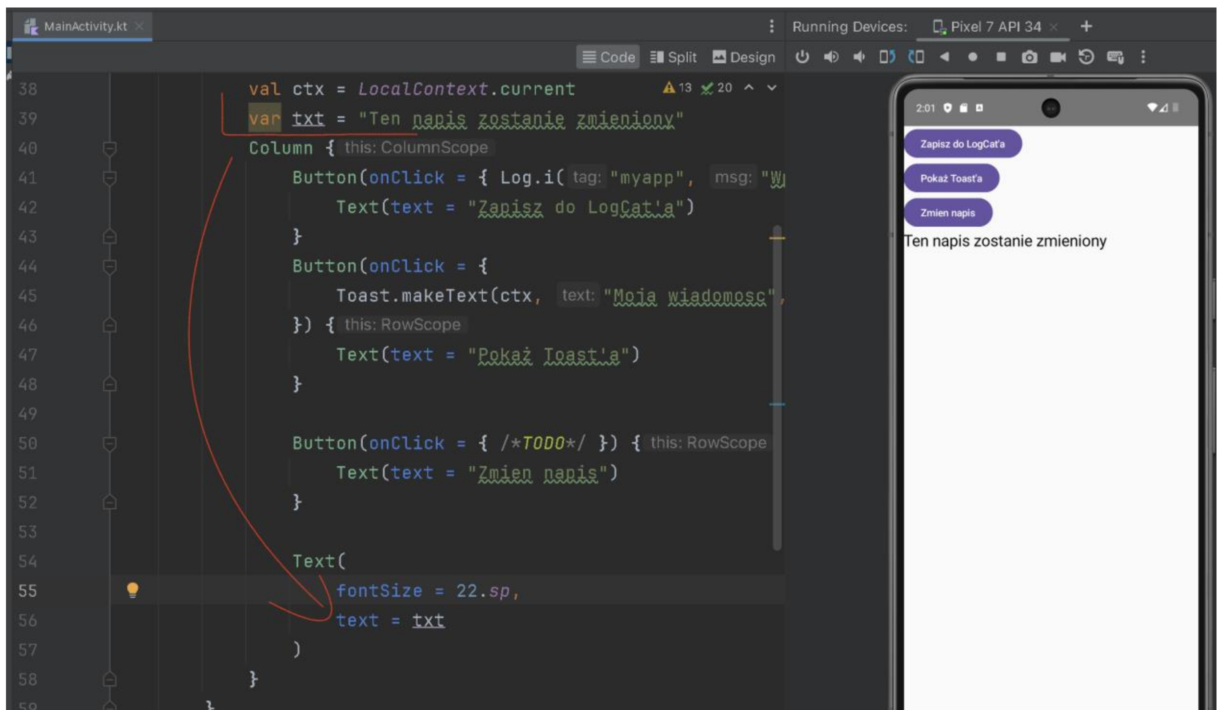
- val ctx = *LocalContext*.current
- i użyjmy go zamiast this w funkcji makeText. To powinno usunąć problem. Sprawdźmy zatem finalnie co dostajemy:

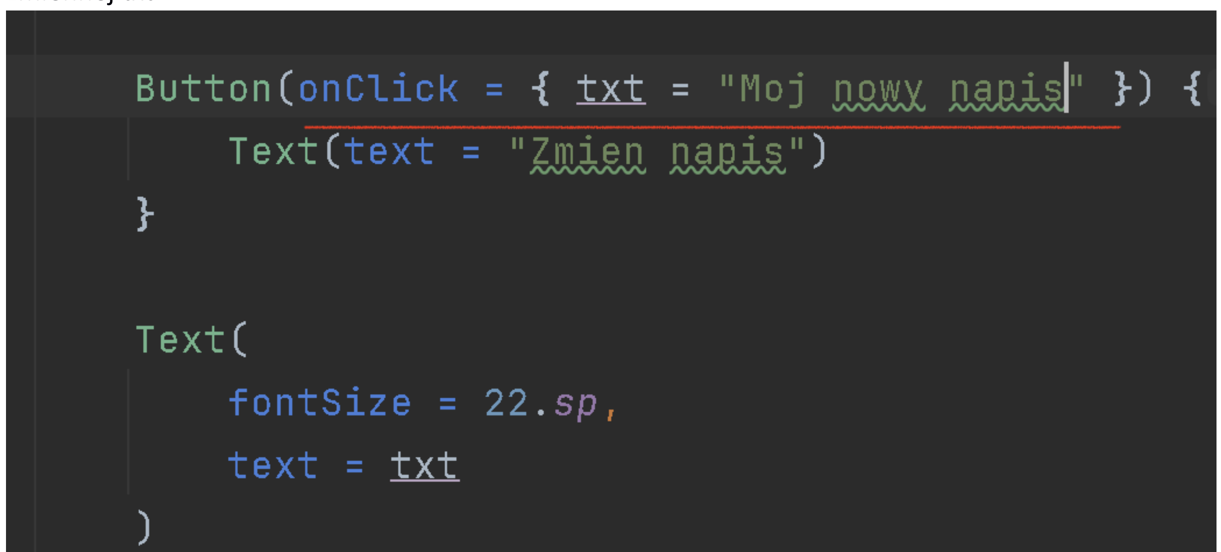


- No to idźmy dalej i spróbujmy „pisać” bezpsrednio po interfejsie użytkownika. Dodajmy zatem kolejny przycisk z napisem „Zmien napis” oraz Tekst z napisem „Ten napis zostanie zmieniony”

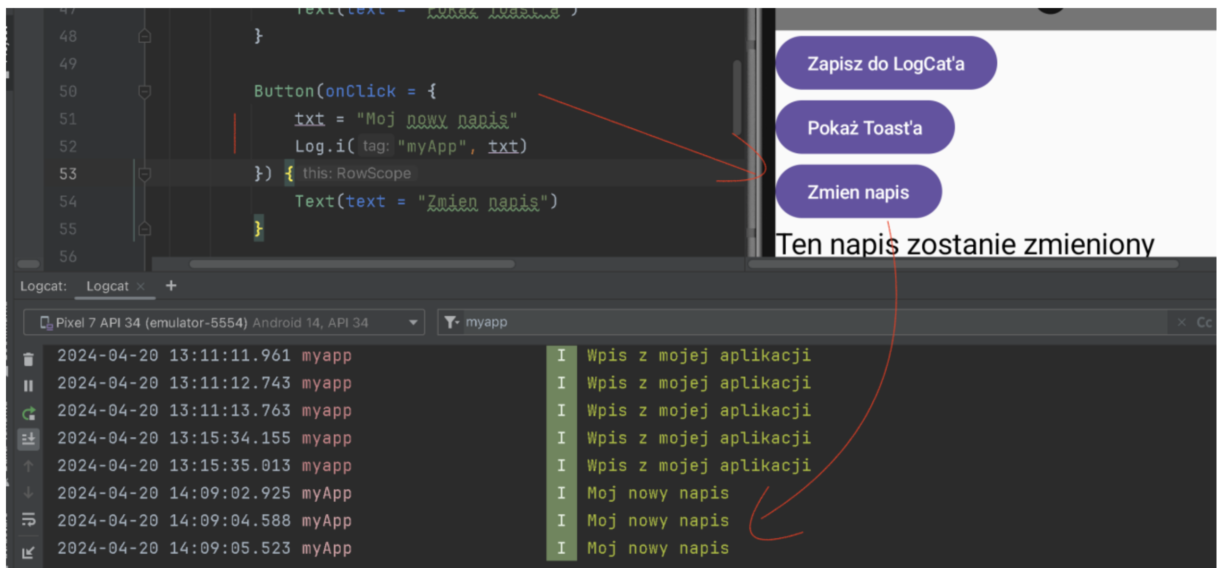


- No i zgodnie z przypuszczeniami, po kliknieciu w nasz nowy przycisk chcielibyśmy zmienić treść/zawartość w naszym Tekscie.
- Pierwszym krokiem byłoby zdefiniowanie zmiennej w której przechowywać będziemy zawartość naszego tekstu:

- 

- No i teoretycznie teraz, nic prostszego tylko przy kliknięciu w przycisk zmienić wartość naszej zmiennej txt
  

- I wszystko wygląda ok…. tylko że nie działa. Tzn ile razy byśmy nie kliknęli w nasz nowy przycisk, tresc napisu pozostaje caly czas taka sama. No wiec pytanie „co poszło nie tak” . Zacznijmy może od tego, że dodamy do naszego onClicka (po zaktualizowaniu wartości zmiennej txt jej zalogowanie do logow urządzenia, i sprawdzimy czy tak akcja w ogole się wykonuje i czy zawartość tej zmiennej się aktualizuje, i zobaczmy co tam się dzieje:

- 

- No więc akcja się wykonuje i zawartość zmiennej się aktualizuje, ale w samej aplikacji sam napis pozostaje niezmieniony…. Z czego to wynika, ano właśnie z tego że zawartość zmiennej txt się co prawda aktualizuje, ale nasz Text „wyrysowany" został na ekranie przy „starej" zawartościi tej zmiennej i później już nikt (i nic) nie wymusza jego odświeżenia.
- Aby to wymusić, zmienna przechowująca określony atrybut elementu UI (w tym przypadku atrybut textelementu Text) musi zostać zadeklarowana w „specjalny" sposób tzn jako zmienna przechowująca (modyfikowalny) stan naszej aplikacji / naszego interfejsu czyli jako MutableState<>. W naszym przypadku będzie to zatem:
- var txt: MutableState<String>
- i dalej przypisujemy jej wartość początkową jako:
- remember { *mutableStateOf*("Ten napis zostanie zmieniony") }
- a zatem, definicja zmiennej txt wygląda następująco:

```
var txt: MutableState<String> =
    remember { mutableStateOf( value: "Ten napis zostanie zmieniony") }
```

- I teraz, w przycisku zmieniamy nie tyle samą zmienną txt ile jej własność .value:

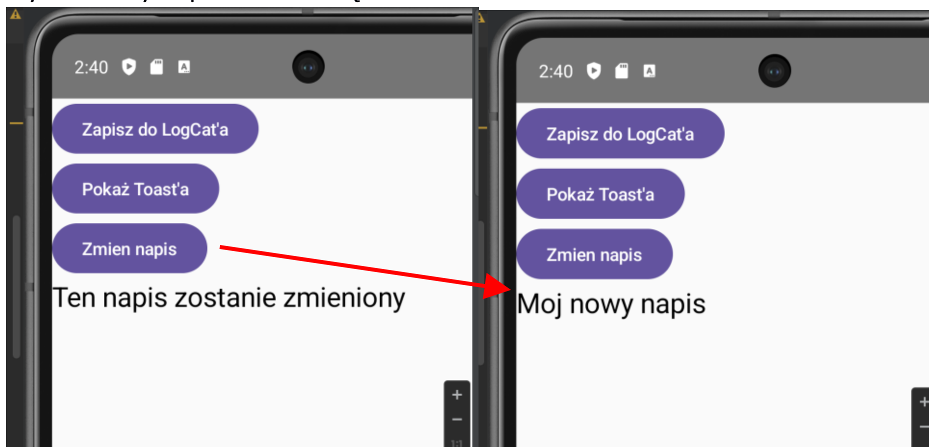
```
Button(onClick = {
    txt.value = "Moj nowy napis"
    Log.i( tag: "myApp", txt.value)
}) { this: RowScope
    Text(text = "Zmien napis")
}
```

- I podobnie, wyświetlając czy przypisując sam napis, nie odwołujemy się do zmiennej txt jako takiej tylko do jej własności .value

```
Button(onClick = {
    txt.value = "Moj nowy napis"
    Log.i( tag: "myApp", txt.value)
}) { this: RowScope
    Text(text = "Zmien napis")
}

Text(
    fontSize = 22.sp,
    text = txt.value
)
```

- No i teraz, powinno nam to zadziałać tak, jakbyśmy chcieli, tzn po kliknieciu w przycisk wyświetlany napis zmienia się.



- Przykład o tyle warto zapamiętać że musimy postąpić w analogiczny sposób za każdym razem kiedy chcemy definiować jakiś atrybut elementu UI poprzez zmienną i móc ją modyfikować wymuszając odświeżenie stanu tego elementu / atrybutu.
- No to spróbujmy iść dalej. Dodajmy nowy przycisk z napisem „Wyswietl kolorowe napisy" i po nacisnieciu tego przycisku chcielibyśmy zgodnie z opisem wyswietlic na ekranie kilka kolorowych napisów.
- I teraz, skoro zaimplementowana przez nas na początku zajęć funkcja MyTexts robi właśnie to co byśmy chcieli (wyświetla kolorowe napisy), to az się prosi aby z niej skorzystać.
- Pierwszy pomysł byłby taki aby po prostu dodać wywołanie funkcji MyTexts jako obsługę kliknięcia naszego nowego przycisku czyli w ten sposób:

```
Button(onClick = { MyTexts() }) { this: RowScope
    Text(text = "Wyswietl kolorowe napisy")
}
```

- Niestety, jak widać nie jest to możliwe, bowiem funkcje Composable (czyli dodające/korzystające z elementów UI (a taką jest nasza funkcja MyTexts) mogą być wywoływane wyłącznie z innych funkcji typu Composable (ew z funkcji setContent) a onClick takiego warunku nie spełnia
- No to wywołujmy nasze MyTexts bezpośrednio z funkcji setContent (tak jak robiliśmy to poprzednio), tylko opakujmy to wywołanie jakims warunkiem typu if(isButtonClicked)
- No to zadeklarujmy sobie zmienną isButtonClicked która na początku byłaby ustawiona na false:
- var isButtonClicked = false
- i która warunkowałaby wywolanie naszej funkcji MyTexts

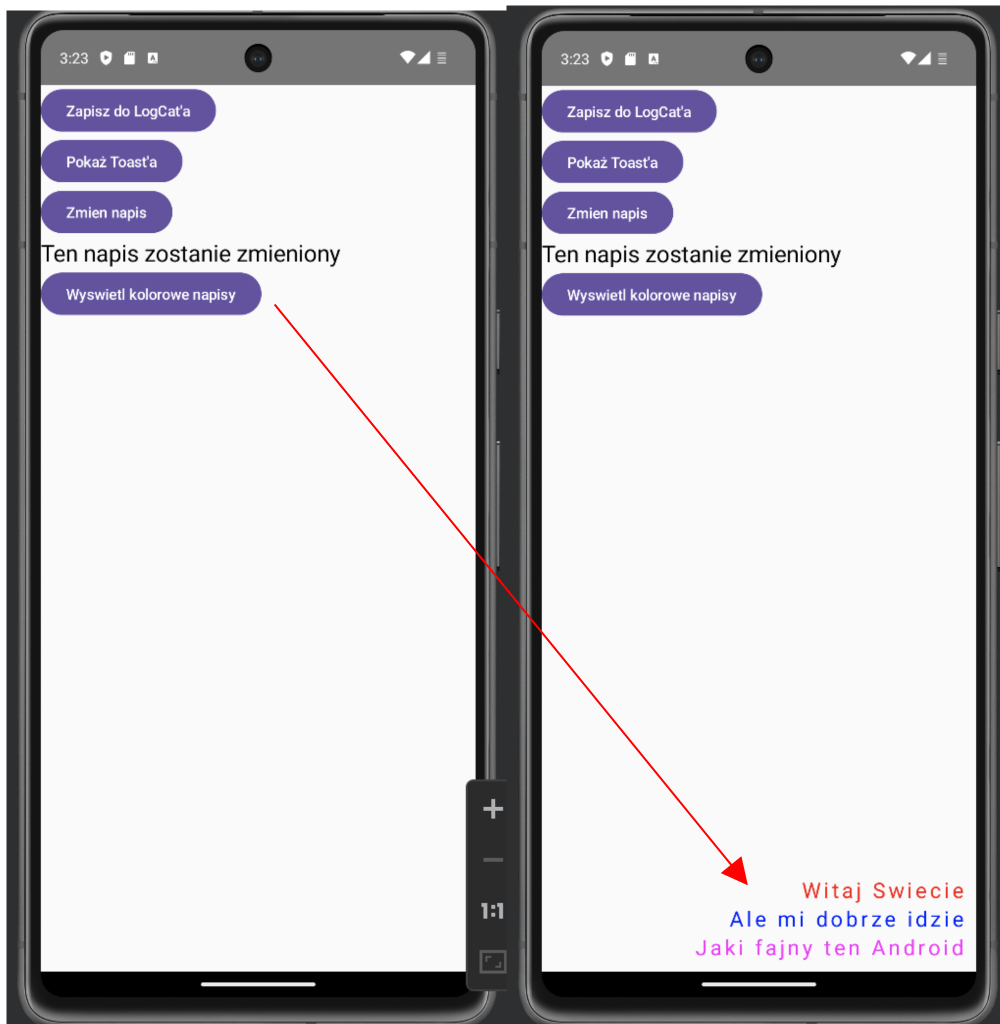
```
if(isButtonClicked){
    MyTexts()
}
```

- I zmieniajmy wartość tej zmiennej na true kiedy klikniemy w nasz przycisk „Pokaz kolorowe napisy"

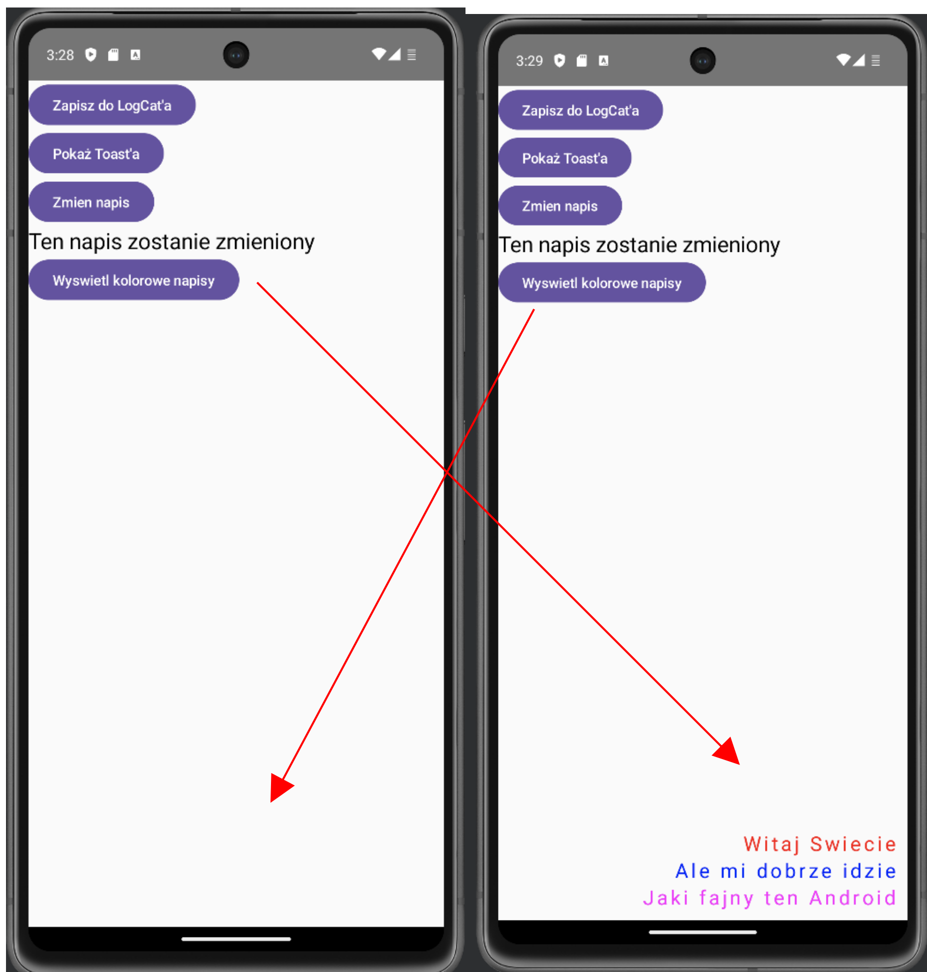
```
Button(onClick = { isButtonClicked = true }) 
    Text(text = "Wyswietl kolorowe napisy")
}

if(isButtonClicked){
    MyTexts()
}
```

- Ale znowu, wszystko wydaje się ok, tylko ze nam to nie zadziała, bo o ile przy kliku modyfikujemy wartość zmiennej isButtonClicked to nic/nikt nie wymusza odswiezenia stanu aplikacji / UI. Ale to już wiemy jak sobie z tym poradzić. Proszę zatem wprowadzić odpowiednie modyfikacjie i zweryfikowac czy/ze po kliknieciu wysietlone zostaną nasze kolorowe napisy. Czyli zachowanie którego oczekujemy jest następujące:

- 

-
- Idąc dalej, zauważmy, że gdyby w obsłudze onClick'a nie tyle ustawiać na sztywno wartość isButtonClicked na true ile przestawiać go z true na false bądź z false na true czyli gdyby zrobić tam przypisanie:
- isButtonClicked.value = !isButtonClicked.value
- To dostaniemy jeszcze fajniejsza funkcjonalność pozwalającą wyświetlac i odpowiednio chować nasze napisy po każdym kolejnym kliknięciu. Czyli dostaniemy:
-



- To może tylko, żeby było już całkiem pięknie, zmieńmy nazwę naszej zmiennej isButtonCliked na coś lepiej oddającego aktualną funkcjonalność, np. shouldTextsBePresented (dla odmiany tym razem proponuje to zrobić korzystając z funkcjonalności refactoringu dostępnej w Studio), no i dodatkowo, powinniśmy zadbać o to aby na naszym przycisku nie wyświetlał się zawsze napis „Wyświetl kolorowe napisy" tylko stosownie do sytuacji albo to co mamy albo cos w stylu „Ukryj kolorowe napisy". Proszę spróbować dokonać odpowiednich modyfikacji.

- I jeszcze jedna wprawka. Mamy na ekranie przycisk po kliknięciu na który zmieniamy tresć napisu „Ten napis zostanie zmieniony" na „Mój nowy napis". Proszę dokonać odpowiednich modyfikacji, tak aby po każdym kliknięciu w ten przycisk następowała zmiana napisu odpowiednio z „Mój stary napis" na „Mój nowy napis" i przy kolejnym kliknieciu ponownie na „Mój stary napis" itd.

- Ok. przećwiczyliśmy wypisywanie/modyfikowanie czegoś na ekranie, no to zobaczmy jak pozwolić użytkownikowi coś wpisywać w aplikacji. Zanim może jednak przejdziemy dalej, proponuje znowu troszkę posprzątać w naszym kodzie. Tradycyjnie po pierwsze poprawmy ewentualne niedoformatowania, a następnie wydzielmy wszystko co aktualnie mamy w setContent do osobnej funkcji. Niech się ona nazywa ButtonExercises. Po wydzieleniu, „zwińmy" sobie jej zawartość, i możemy przystąpić do dalszych prac .

- Elementem który wykorzystamy do wprowadzania danych przez użytkownika będzie TextField. Idziemy zatem do naszego setContent, piszemy TextField, Studio pokaże nam dostępne opcje,

-

```
import ...

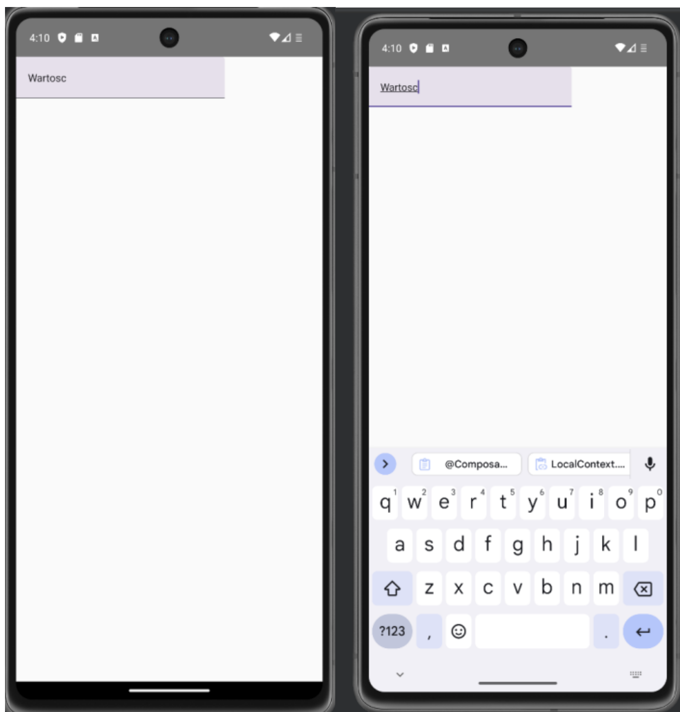
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            TextFiel
        }
    }
}

@Composabl
```

```
TextField(value: String, onValueChan…
TextField(value: TextFieldValue, onV…
TextFieldDefaults (androidx.compose.…
TextFieldColors (androidx.compose.ma…
TextFieldValue (androidx.compose.ui.…
OutlinedTextField(value: String, onV…
OutlinedTextField(value: TextFieldVa…
OutlinedTextField(value: TextFieldVa…
BasicTextField(value: String, onValu…
BasicTextField(value: TextFieldValue…
```

```
@OptIn(markerClass = {androidx.compose.material3.Ex
@Composable
public fun TextField(
    value: String,
    onValueChange: (String) -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    readOnly: Boolean = false,
    textStyle: TextStyle = LocalTextStyle.current,
    label: @Composable() (() -> Unit)? = null,
    placeholder: @Composable() (() -> Unit)? = null
    leadingIcon: @Composable() (() -> Unit)? = null
    trailingIcon: @Composable() (() -> Unit)? = nul
    prefix: @Composable() (() -> Unit)? = null,
    suffix: @Composable() (() -> Unit)? = null,
    supportingText: @Composable() (() -> Unit)? = n
    isError: Boolean = false,
    visualTransformation: VisualTransformation = Vi
    keyboardOptions: KeyboardOptions = KeyboardOpti
```

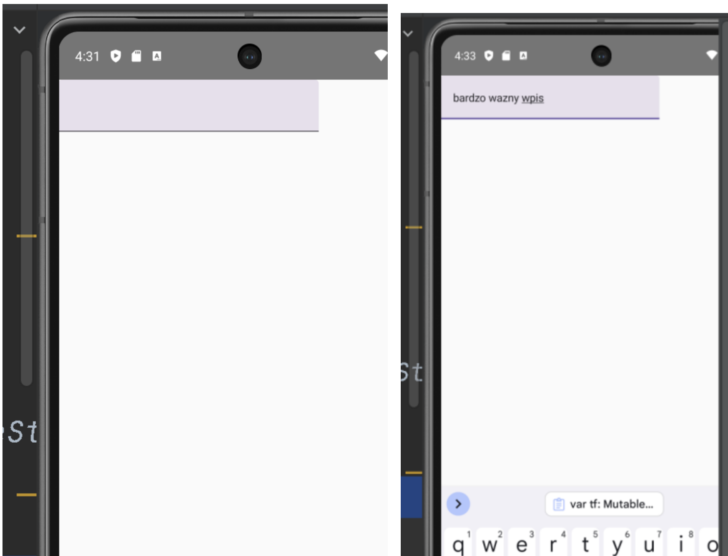
- interesuje nas pierwsza, naciskamy enter, i uzupełnijmy na początek value na „Wartosc" a onValueChange na {} (puste nawiasy klamrowe) i sprawdzmy co mamy

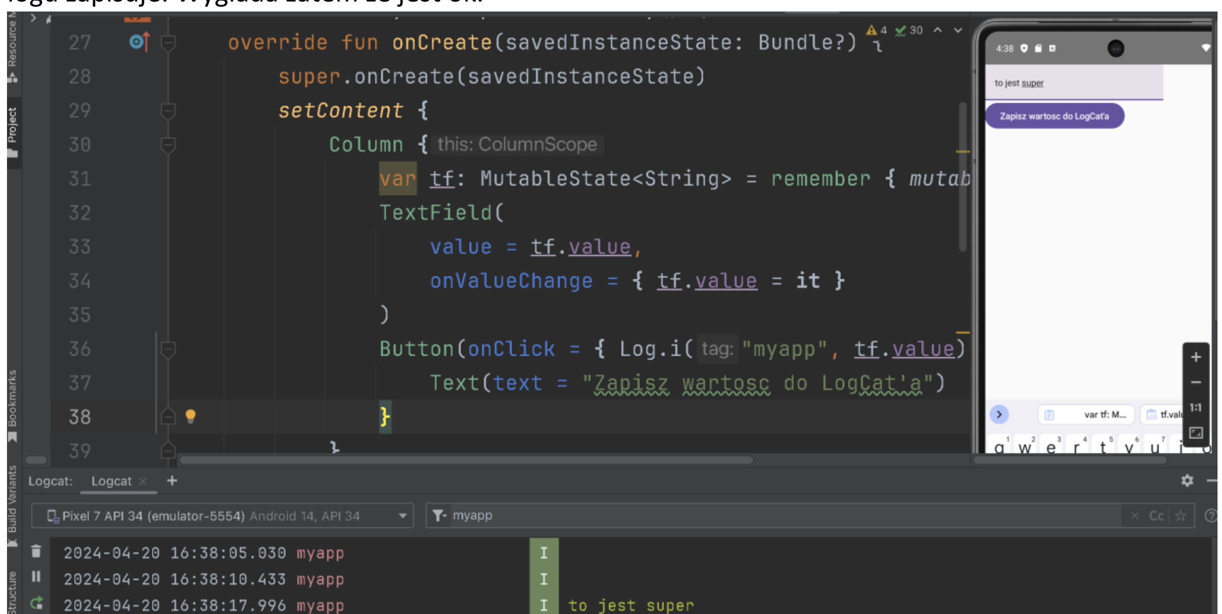
- 

- No więc dostajemy pole tekstowe, w które teoretycznie daje możliwość wpisania czegoś (i nawet pokazuje się klawiatura która ma to umożliwiać) tyle tylko że co byśmy nie wpisali to zawartość tego pola się nie zmienia.

- No i nie ma się czemu dziwić bo sami wpisaliśmy w kodzie że value dla tego pola to „Wartosc" i nigdy tego nie zmieniamy. Żeby nasze pole zaczęło zachowywać się tak jakbyśmy tego oczekiwali to po pierwsze zdefiniujmy zmienną która będzie przechowywać nam

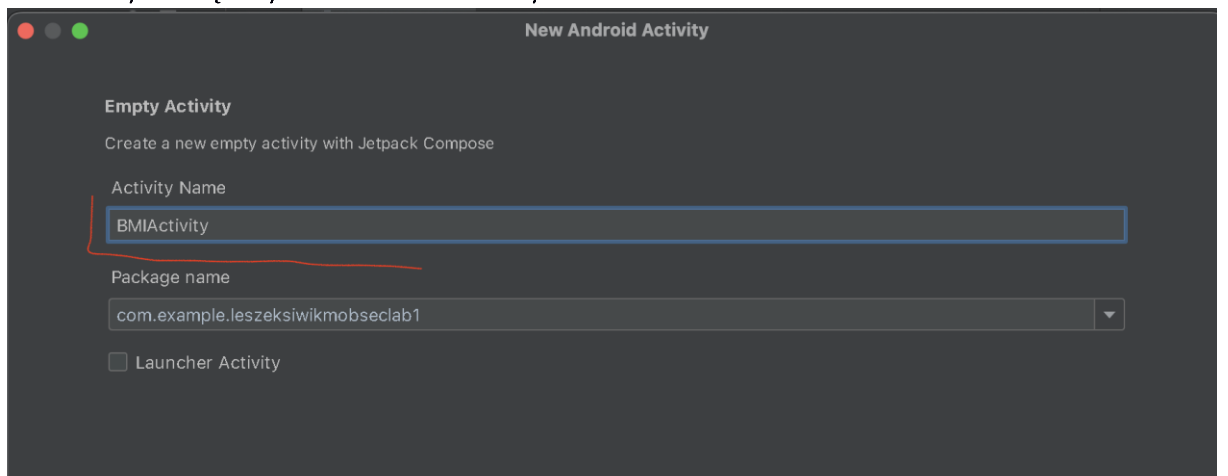
(za)wartość naszego pola tekstowego. Co do ozasady potrzebujemy zmiennej stringowej, ale ponieważ chcemy żeby była ona powiązana ze stanem aplikacji definiujemy ją jako MutableState<String> i ustawmy jej wartość początkową jako pusty string czyli:

- var tf: MutableState<String> = remember { *mutableStateOf*("") }
- I teraz atrybut value naszego pola tekstowego ustawiamy na tf.value, a w onValueChange aktualizujemy tą wartość w następujący spsób:
- onValueChange = { tf.value = it }
- (it reprezentuje w onValueChange bieżącą zawartość naszego pola tekstowego którą zapamiętujemy w naszej zmiennej tf (a w zasadzie w jej właściwości value). Zobaczmy jak nam to aktualnie działa:


- No więc wydaje się że zachowuje się tak jakbyśmy oczekiwali. Aby zweryfikować że wszystko działa ok proszę dodać przycisk z napisem „Zapisz wartość do LogCat'a" w obsłudze którego aktualna zawartość pola tekstowego zostanie wpisana do Logu urządzenia.
- No więc przy pustym polu dostajemy pusty wpis do Loga. Jak cos wpiszemy to tez to się w logu zapisuje. Wyglada zatem że jest ok.


- Kolejną rzeczą jaką chciałbym żebyśmy przećwiczyli to otwieranie nowego ekranu (które w Androidzie nazywamy aktywnościami).

- W pierwszym kroku, (żebyśmy mieli co otwierać dodajmy do projektu nową aktywność, nawijmy ją BMIActivity. Wracamy zatem do Android Studio. Następnie File -> New -> Compose -> Empty Activity
-
- Ustawiamy nazwę aktywności na BMIActivity:

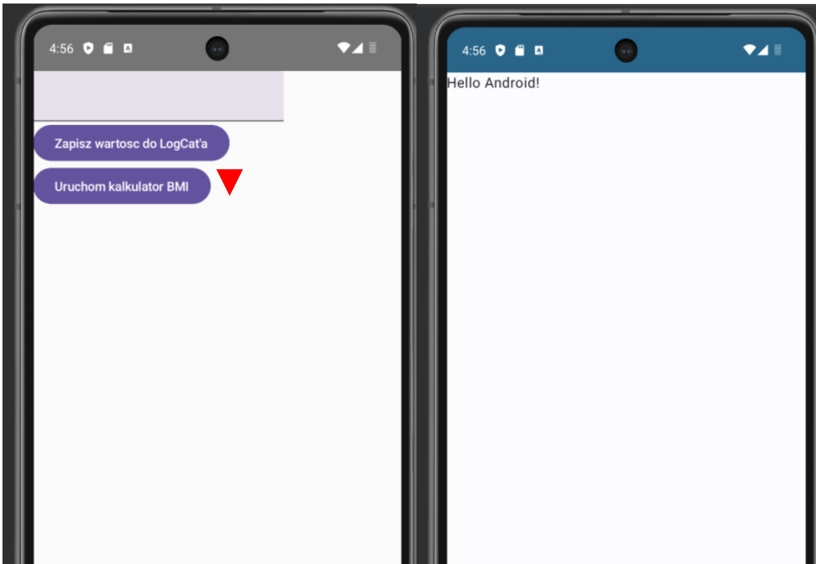


-
- I klikamy Finish.
- Wracamy do naszej funkcji setContent i dodajemy nowy przycisk z napisem „Uruchom kalkulator BMI”.

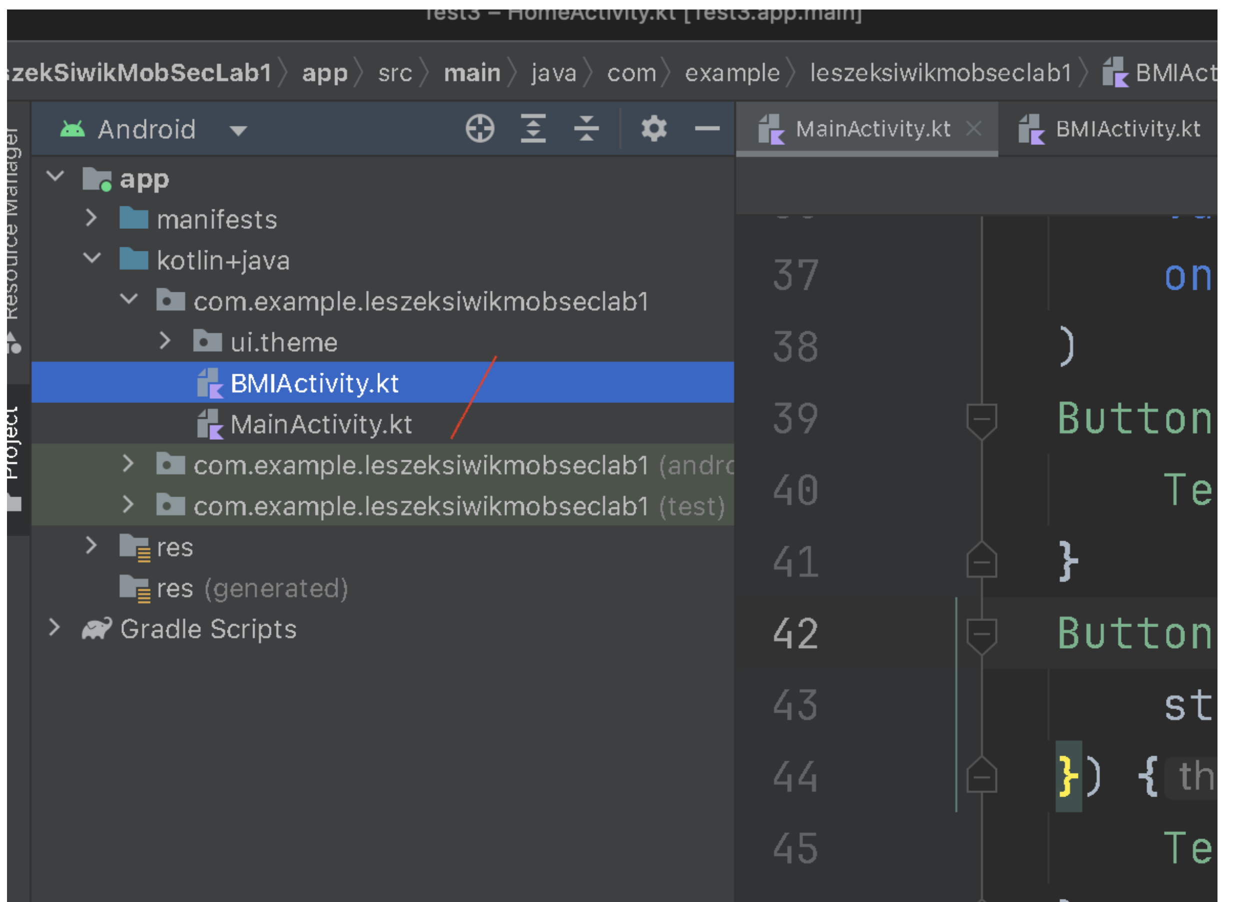
- Żeby uruchomić nową aktywność potrzebujemy kontekst bieżącej, więc gdzieś na początku funkcji setContent dodajemy sobie jego zapamiętanie, tak jak robiliśmy to wcześniej:
- `val ctx = LocalContext.current`
-
- a następnie w obsłudze onClick przycisku który ma uruchomić nową aktywność dodajemy następujący fragment kodu:

```
Button(onClick = {
    startActivity(Intent(ctx, BMIActivity::class.java))
}) { this: RowScope
    Text(text = "Uruchom kalkulator BMI")
}
```

-
- I aktualnie po kliknięciu w nasz przycisk „przejścia" powinniśmy zostać przekierowani do nowej aktywności:



-
- No to zacznijmy pracować z naszą nową aktywnością BMIActivity. Generalnie plik z jej definicją znajdziemy w tym samym miejscu gdzie plik z definicją naszej MainActivity:

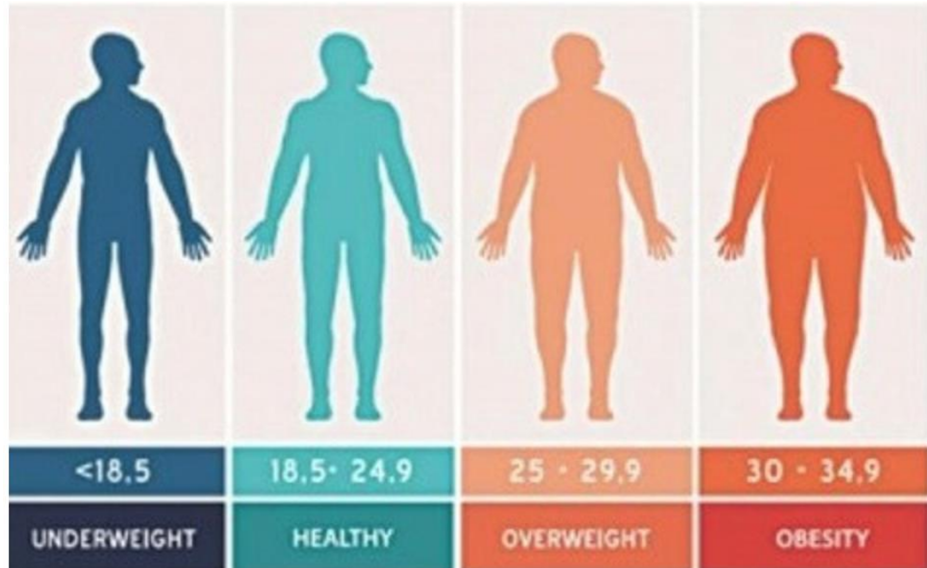
- 

- Otwórzmy sobie ten plik. Usuwamy zawartość setContent oraz funkcje Greeting i GreetingPreview i jesteśmy gotowi do pracy z nową aktywnością, i spróbujemy teraz połączyć elementy których się dziś nauczyliśmy.
- Proszę zatem przygotować następujący ekran:
-

- A następnie proszę zaimplementować funkcjonalność obliczania (i wyświetlania) wartości indexu BMI w zależności od podanej wagi i wzrostu. Wzór na wartość BMI to:

$$BMI = \frac{\text{masa w kg}}{(\text{wzrost w m})^2}$$

- Dodatkowo, należy wyświetlić „diagnozę” (na razie w wersji „tekstowej” z odpowiednim kolorem czcionki), zgodnie z poniższym rysunkiem:



- Aplikacja powinna być (w miarę □ odporna na błędne dane wejściowe, dzielenie przez 0 etc.
- Zadanie z gwiazdką: zamiast diagnozy „tekstowej” proszę wyświetlić diagnozę „obrazkową” korzystajac z porzykladowych infografik zaczerpnietych z internetu