Technologie informatyczne w klasycznym i inteligentnym sterowaniu

1. Aplikacje mobilne

Opracował: Maciej Penar

Spis treści

[1. Aplikacje mobilne - stan na dziś 3](#_Toc35268416)

[2. Jak zacząć 4](#_Toc35268417)

[Na urządzeniu 4](#_Toc35268418)

[Przy instalacji Android Studio 4](#_Toc35268419)

[Po instalacji Android Studio – sprawdź ADB 4](#_Toc35268420)

[ADB 5](#_Toc35268421)

[Mój pierwszy projekt 6](#_Toc35268422)

[Odpalamy projekt 7](#_Toc35268423)

[3. Budowa aplikacji Androidowej 9](#_Toc35268424)

[Aplikacja z punktu widzenia architekta – dużo nieprawdy 9](#_Toc35268425)

[! Activity 10](#_Toc35268426)

[Cykle życia 11](#_Toc35268427)

[Aplikacja z punktu widzenia Programisty 11](#_Toc35268428)

[Napsujmy coś 15](#_Toc35268429)

[Zamiana napisu HelloWorld – statyczne podejście 15](#_Toc35268430)

[Zamiana napisu HelloWorld – statycznie, ale z zasobami 15](#_Toc35268431)

[! Zamiana napisu HelloWorld – Dynamicznie w kodzie 16](#_Toc35268432)

[! Otworzenie Nowej Aktywności 16](#_Toc35268433)

[Aplikacja z punktu widzenia architekta – sprostowanie 17](#_Toc35268434)

[4. Kotlin 18](#_Toc35268435)

[5. Android 18](#_Toc35268436)

[6. Zaliczenie 18](#_Toc35268437)

# 1. Aplikacje mobilne - stan na dziś

Na chwilę obecną mamy dwa główne systemy operacyjne na smartphony:

* Android – i wszystkie jego pochodne (MIUI/Lineage’y)
* iOS

W branży można też znaleźć obecność innych systemów, które albo odciskają piętno swoją obecnością (Tizen) albo dopiero zamierzają się ujawnić (Fuchsia). Każdy z systemów wypracowuje swój własny sposób developmentu aplikacji mobilnych. Oczywiście jak to w życiu bywa, nie wszystkim się to podoba i jak grzyby po deszcz pojawiają się różne „alternatywne” sposoby wytwarzania aplikacji mobilnych. Ogólnie są trzy metodyki:

1. **Aplikacje natywne** – pisane z wykorzystaniem dedykowanych SDK
   1. W Androidzie z wykorzystaniem (kiedyś) Javy lub (dziś) Kotlina
   2. W iOS z wykorzystaniem (kiedyś) Objective-C lub (dziś) Swift
2. **Aplikacje hybrydowe/webowe** – pisane jako aplikacja natywna zawierająca jeden ekran z jednym widgetem tzw. WebView – czyli komponentem przeglądarki. Aplikacje te otwierają specjalna stronę webową stanowiącą de-facto aplikację mobilną. Development odbywa się za pomocą javascriptu i css-a i HTML’a. Kiedyś istniał do tego React Native i PhoneGap/Cordova, ale nie wiem czy te frameworki istnieją**,** bo są ogólnie uznawane za kiepskie.
3. **Aplikacje hybrydowe/kompilowane –** pisane w dziwnych językach takich jak Dart/C#. Wymagają dołączenia środowisk wykonawczych do aplikacji mobilnych, przez co apki ważą **bardzo dużo**. Przedstawicielami są Xamarin (teraz mniej popularny) i Flutter (pitchowany przez Google’a jako zbawca mobilek).
4. **Aplikacje webowe** – dostępne z poziomu przeglądarki – dopisujemy CSS-y – to rozwiązanie nigdy nie działało dobrze. W założeniu miało działać dobrze na obu systemach, w praktyce nie działa na żadnym.
5. **PWA –** podejście numer dwa do sprzedawania aplikacji webowych jako mobilnych (powodem tej upartości jest trudny proces umieszczania aplikacji w Apple AppStore)

Na tych zajęciach **jedynym** sposobem developmentu który będzie nas interesował to aplikacje **natywne pisanie w Kotlinie.**

*Wiem co teraz myślicie – nowy język, nowe problemy – ale Kotlin powstał po to żeby uprościć wytwarzanie aplikacji na Androida. I robi to bardzo skutecznie. Tam gdzie w Javie potrzebowalibyście napisać bardzo dużo linijek kodu, Kotlin robi dużo rzeczy ‘pod spodem’.*

Jestem zdania, że jeśli mobilki mają Wam się przydać to apki natywne to jedyna opcja dla Automatyków – być może znajdziecie sposób parowania mikrokontrolerami, czy pojawi się potrzeba pchania danych przez USB z komórki itp. (raz spotkałem się z apkę która wypychała/pobierała dane z urządzenia zewnętrznego). Taką możliwość daje tylko natywny sposób pisania apek.

# 2. Jak zacząć

Żeby zacząć programowanie na Androida musicie posiadać:

* Urządzenie z Androidem (bądź symulator) + kabel USB
* Android Studio [link](https://developer.android.com/studio/install)

## Na urządzeniu

Co by odblokować urządzenie do developmentu należy:

1. Wejść w **Ustawienia**
2. Wejść w **Telefon – Informacje**
3. Wejśćw **Informacje o oprogramowaniu**
4. Kliknąć 5 razy na Numer Wersji – System będzie powiadamiał o liczbie kliknięć
5. Wejść w Ustawienia -> Opcje Programisty, odblokować „Debugowanie USB”

Jak coś się nie zgadza to pooglądać jak to wygląda tu: [link](https://www.wikihow.tech/Unlock-Developer-Options-on-Android)

## Przy instalacji Android Studio

Wybrać Kotlina

## Po instalacji Android Studio – sprawdź ADB

Po instalacji Android Studio upewnijcie się, że macie zainstalowane **adb**. Na Windowsie binaria powinny być zainstalowane w:

$FolderUżytkownika\AppData\Local\Android\Sdk\platform-tools

**adb** służy do komunikacji poprzez usb z urządzeniem, więc żeby sprawdzić czy wszystko jest ok to otwieramy shell:

|  |  |
| --- | --- |
| Wpisujemy: | **adb** |
| Powinno wyświetlić: | Android Debug Bridge version 1.0.40  Version 4986621 |

Jeśli nie wyświetla to musimy wyeksportować w/w ścieżkę do zmiennej %PATH% - liczę, że to powszechna wiedza (na pewno było na Informatyce na pierwszym roku :P). Jeśli wyświetla to podłączcie telefon pod USB (w trybie MTP lub plików) i wpiszcie

|  |  |
| --- | --- |
| Wpisujemy: | **adb devices** |
| Powinno wyświetlić: | List of devices attached:  …… |

Możliwe, że w tym momencie telefon poprosi o autoryzację wg. komputera (może prosić później). Zaakceptować.

## ADB

Tu zostawiam info dla ciekawskich:

Dwie przydatne komendy ADB to:

* **adb start-server** - ensure that there is a server running
* **adb kill-server** - kill the server if it is running

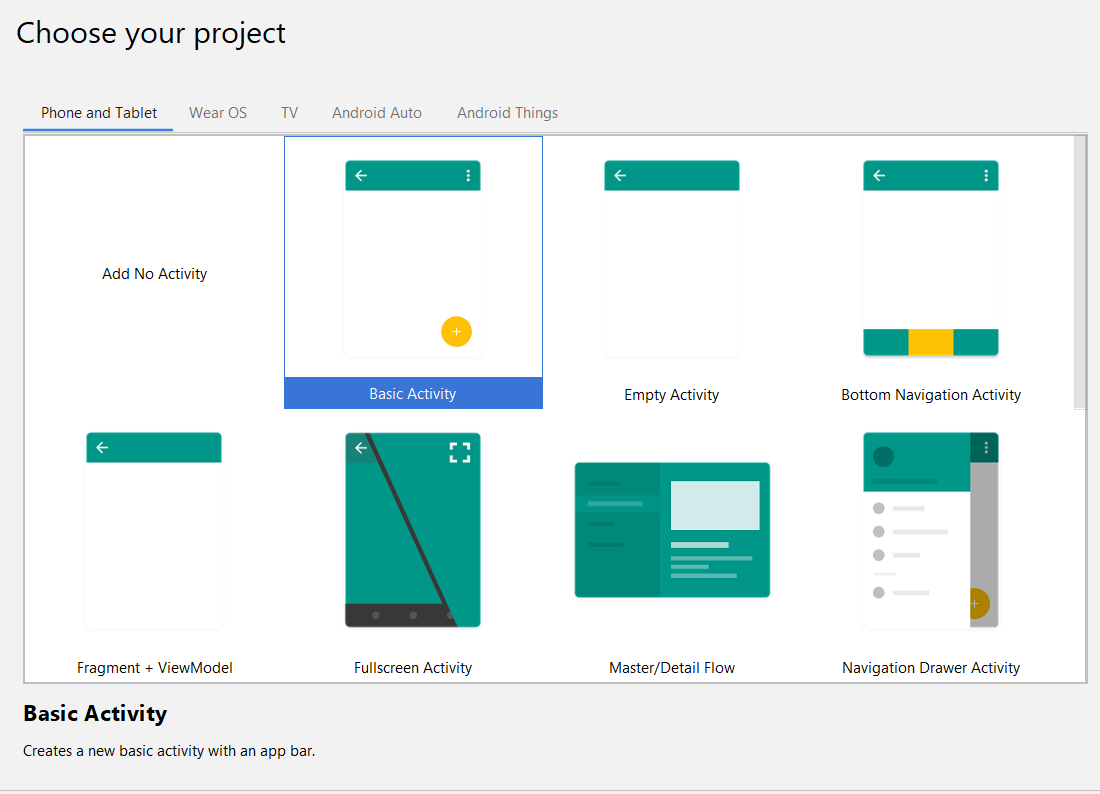
Które zazwyczaj wykorzystywane są gdy coś nakłapiecie i ADB się zatnie. Wtedy kombinacje kill-server + start-server resetuje ADB.

Inne fajne komendy to:

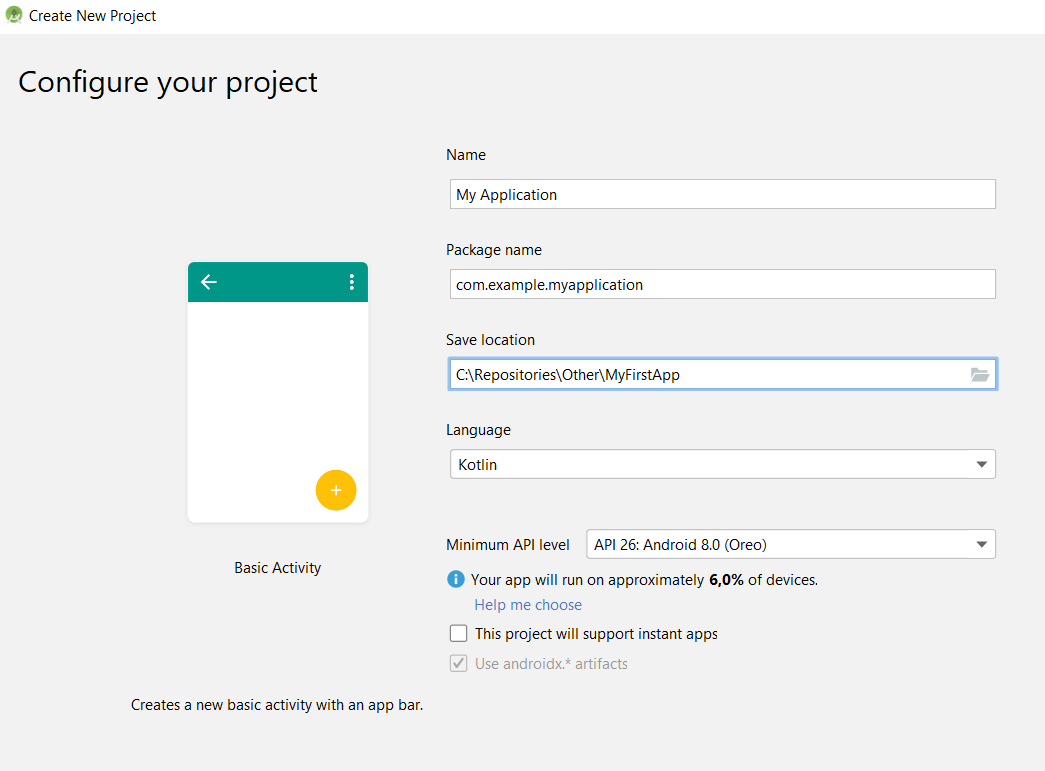
* **adb root** - restart adbd with root permissions
* **adb shell** - run remote shell command (interactive shell if no command given)
* **adb push** - copy local files/directories to device
* **adb pull -** copy files/dirs from device

## Mój pierwszy projekt

1. Otworzyć Android Studio i utworzyć projekt na szablonie **Basic Activity**: File -> New -> New Project



1. Wybrać język Kotlin i odpowiednią nazwę pakietu (Package name) – w przypadku gdybyście wystawiali apke do sklepu to pakiet musi być **globalnie unikatowy.** Wybrać odpowiedni API level – zaraz o tym więcej.



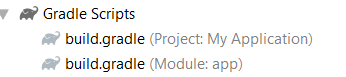
Teraz Android Studio zacznie indeksować pliki i po chwili – jeśli wszystko poszło zgodnie z planem – i Wasza komórka obsługuje wybrany poziom API – to powinniście móc projekt odpalić, jeśli nie możecie to nie stresujcie się, tylko czytajcie dalej.

Istotnymi miejscami w Android Studio są:

1. Opcja File->Sync Project with Gradle Files – służy do synchronizacji środowiska budującego z plikami projektowymi, zazwyczaj ta operacja wykonuje się automatycznie, ale w razie fiaska trzeba uruchamiać ten proces pod podaną ścieżką
2. Odpalenie projektu:  - jeśli sprzęt podłączyliście pod USB i jest widoczne za pomocą **adb devices** – jeśli macie odpalony symulator to w dropdownie też możecie go wybrać
3. Po prawej na górze powinniście widzieć:  . Dwie ostatnie ikony to:
   1. AVD manager – służący do tworzenia symulatorów smartphoneów
   2. SDK manager – służący do instalacji różnych poziomów API.

## Odpalamy projekt

Odpalmy projekt, podpinamy urządzenie pod USB:

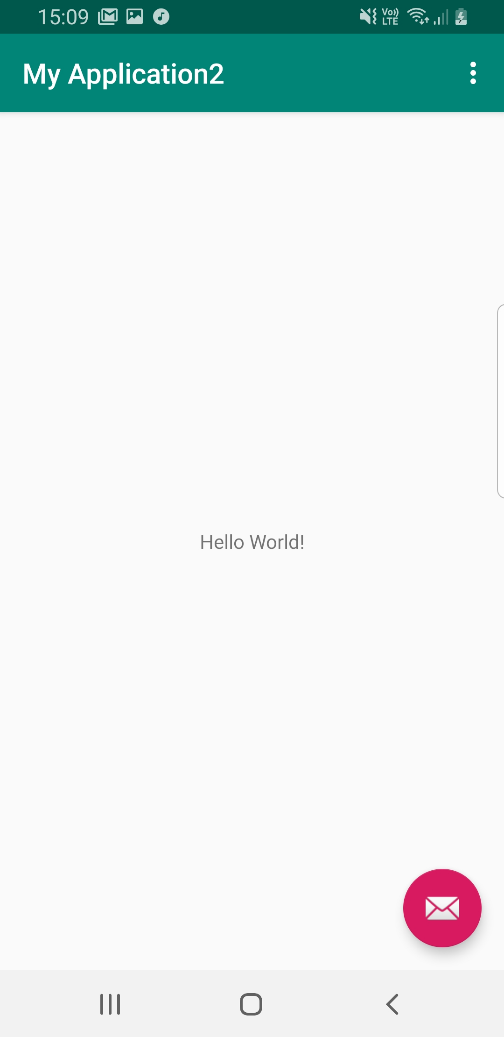
1. Jeśli przycisk wygląda  to odpalamy i cieszymy się apką na telefonie.
2. Jeśli przycisk robi problemy to Google’amy wg. komunikatów:
   1. Jest ryzyko, że nie adb nie wykrywa urządzenia. To możecie zdiagnozować przez **adb devices**
   2. Jest ryzyko, że nie zaakceptowaliście urządzenia jako zaufanego – Android Studio będzie wyświetlać je jako [UNAUTHORIZED]
   3. Jest ryzyko, że nie macie zainstalowanego odpowiedniego SDK, tj. Wasz sprzęt jest stary. W tym celu:
      1. Klikamy  i instalujemy odpowiedni pakiet dla naszego telefonu. Pakiety ważą dużo – nie instalujcie wszystkich jak leci.
      2. W AS znajdujemy pliki o nazwie build.gradle 

Wybieramy plik należący do modułu **app** i modyfikujemy sekcje:

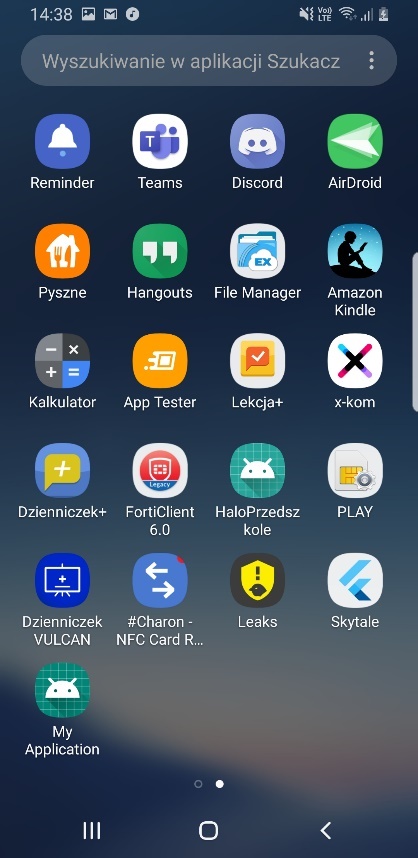
|  |
| --- |
| defaultConfig {  applicationId **"com.example.myapplication"** minSdkVersion 20  targetSdkVersion 29  versionCode 1  versionName **"1.0"** testInstrumentationRunner **"androidx.test.runner.AndroidJUnitRunner"** } |

Tak by wartość minSdkVersion odpowiadała poziomowi który zainstalowaliśmy w zakładce: .

Po odpaleniu projektu powinniście zobaczyć na telefonie:

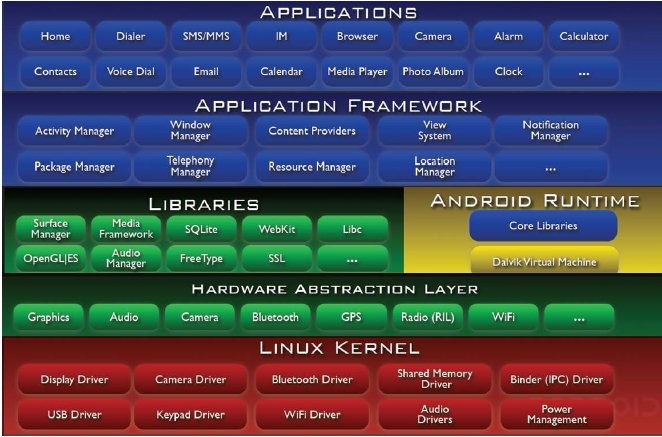


A na pulpicie powinniście widzieć ikone apki.



# 3. Budowa aplikacji Androidowej

Zamieszczam słynny rysunek architektury Androida obrazujący jego możliwości:



Na diagramie zaznaczono różne funkcjonalne obszary które Android (i jego SDK) oferują użytkownikom. Dla nas oznacza to tyle, że dowolna funkcjonalność z tego diagramu jest do wykorzystania w przystępny sposób. Na przykład: nie programujemy Maila / wysyłki SMS-ów / przeglądarki internetowej / SQLite’a / GPS-a: istnieją specjalne klasy umożlwiające wykorzystanie tych funkcji. Im wyżej na diagramie, tym „łatwiejsze” do wykorzystania są one.

## Aplikacja z punktu widzenia architekta – dużo nieprawdy

Ikona na pulpicie Androida reprezentuje **aplikację. Aplikacja** możemy podzielić na ekrany zwane **aktywnościami.** Każdy ekran musi być jawnie zarejestrowany w pliku manifestu aplikacji tzw. AndroidManifest.xml. Na aplikację mogą składać się też dwa inne komponenty podlegające obowiązkowi rejestracji: **Broadcast Receivery** oraz **serwisy** – oba komponenty realizują długotrwałe/krótkotrwałe działania w tle.

Z punktu widzenia programisty:

* Aplikacja reprezentowana jest przez klasę Application ([link](https://developer.android.com/reference/android/app/Application?hl=en))
* **Aktywność reprezentowana jest przez klasę Activity (**[**link**](https://developer.android.com/reference/android/app/Activity)**)**
* Serwis reprezentowany jest przez klasę Service ([link](https://developer.android.com/reference/kotlin/android/app/Service?hl=en))
* Broadcast reprezentowany jest przez klasę BroadcastReceiver ([link](https://developer.android.com/reference/android/content/BroadcastReceiver))

Bez klasy Activity ani rusz. Klasy Application, Service oraz Broadcasty można sobie na razie podarować. Aktywności stanowią funkcjonalności uruchamiane za pośrednictwem systemu operacyjnego – **Intencja** opisuje … cóż… intencję uruchomienia aktywności. Ekrany składamy z użyciem widgetów (widoków) (ang. View).

## ! Activity

**TO JEST BARDZO WAŻNA SEKCJA.**

Aktywność, czyli to co jest wyświetlane użytkownikowi, podlega tzw. cyklowi życia. Obrazuje go następujący diagram:



Diagram przedstawia w jakiej kolejności i w jakich warunkach wywoływane są odpowiednie metody klasy Activity. W momencie w którym aplikacja jest uruchomiona, „odpowiednia” aktywność wywołuje metodę onCreate(), potem onStart() – w tym momencie następuje wyświetlenie widoku użytkownikowi – po czym onResume() i użytkownik może cieszyć się aplikacją.

Jeśli użytkownik wejdzie na kolejny ekran – **aktywność nie jest domyślnie usuwana** – dzięki temu przycisku cofnij nie musimy oprogramowywać (mechanizm ten jest obsługiwany przez tzw. BackStack [link](https://developer.android.com/guide/components/activities/tasks-and-back-stack)). Najpierw wywoływana jest metoda onPause(), potem gdy już użytkownik ma zaprezentowany widok nowej aktywności onStop().

Dopiero gdy aplikacje ubijemy ręcznie wywoływane jest onDestroy() (lub w przypadku nieobsłużonego wyjątku).

No dobra, czym jest „odpowiednia” aktywność:

* Jeśli otwieramy aplikację z pulpitu, która nie działała w tle, to otwierana jest aktywność opatrzona w pliku AndroidManifest.xml następującym filtrem:

<**intent-filter**>  
 <**action android:name="android.intent.action.MAIN"** />  
 <**category android:name="android.intent.category.LAUNCHER"** />  
</**intent-filter**>

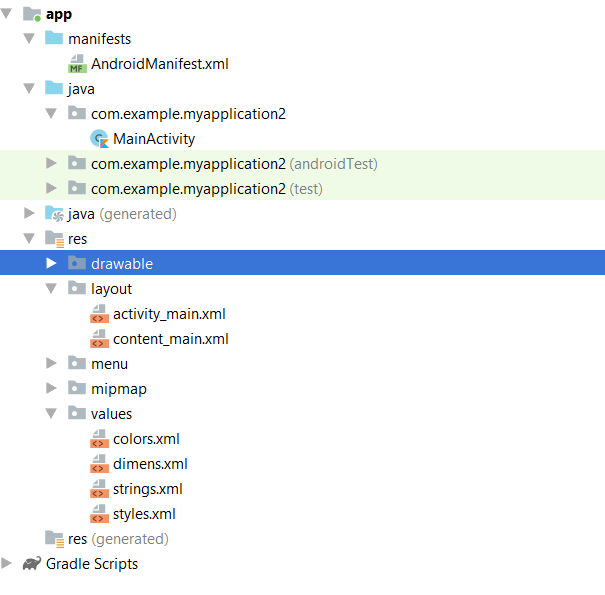
* Jeśli otwieramy aplikację z pulpitu, która działała w tle, to prezentowana jest ostatnia aktywność i cykl przechodzi od onRestart()
* Każdy inny przypadek wywołania aktywności wymaga podania jej nazwy ręcznie

## Cykle życia

Tak jak aktywność podlega cyklom życia, tak inne komponenty też są poddane podobnym cyklom. Nazwy konkretnych stanów mogą się oczywiście różnić – odsyłam do dokumentacji. Najbardziej istotne jest to, że definiowanie własnej klasy **Application** wykonujemy gdy chcemy wpłynąć na działanie aplikacji w którymś z jej cykli. Analogicznie sprawa wygląda z klasą **Fragment –** ale o niej na razie cicho sza.

## Aplikacja z punktu widzenia Programisty

Moje drzewko projektu utworzone z szablonu wygląda w następujący sposób:



Możemy wyróżnić 4 główne obszary na których pracujemy:

* **Plik AndroidManifest.xml** – który opisuje co nasza aplikacja może / powinna zrobić i na otwieranie jakich aktywności zezwalamy jako programiści
* **Folder Java/nazwa pakietu** – tu kodujemy w plikach Kotlinowych – **centralne miejsce pracy**
* **Folder res** – używany do przechowywania różnych stałych np. kolorów, ciągów znaków…. **Ale też do przechowywania widoków.** Odwołania do zasobów odbywają się z kodzie za pomocą klasy statycznej R. Na przykład R.layout.xxxxx albo R.strings.aaaaa.
* **Gradle Scripts** – pliki dla środowiska budującego Gradle. Prosty projekt ma zazwyczaj dwa: jeden konfigurujący projekt (w skład projektu może wchodzić wiele aplikacji mobilnych) oraz plik dla już konkretnej aplikacji mobilnej (Modułowy). Tak naprawdę nie ma potrzeby żebyście się tymi plikami bawili – jeśli ktoś znajdzie jakieś fajne libki, to w modułowym w sekcji dependencies może zamieścić odpowiednie wpisy. Warto tutaj zaznaczyć, że Kotlin kompiluje się do kodu bajtowego Javy i też z Javowych libek można korzystać, więc nic tylko grzebać w <https://mvnrepository.com/>. Np. jeśli chcecie mieć dostęp do lepszej libki obsługującej daty ([link](https://mvnrepository.com/artifact/joda-time/joda-time/2.10.5)) to można wpisać:

compile group: 'joda-time', name: 'joda-time', version: '2.10.5'

Lub po nowemu

implementation 'joda-time:joda-time:2.10.5'

Jak te obszary się spinają u mnie:

Sablon zawiera prostą implementację klasy **aktywności** w folderze Java/com/example/myapplication2 **MainActivity.kt.** Jej definicja wygląda mniej więcej tak:

|  |
| --- |
| **class** MainActivity : AppCompatActivity() {   **override fun** onCreate(savedInstanceState: Bundle?) {  **super**.onCreate(savedInstanceState)  setContentView(R.layout.*activity\_main*)  }  } |

Podczas uruchamiania tej aktywności, onCreate() ustawia widok z **res/layout/activity\_main.xml** (z kody odwołujemy się przez klasę R: R.layout.*activity\_main*) za pomocą **bardzo istotnej metody setContentView().**

Patrząc na zawartość (nieprzyjemnego) pliku XML **activity\_main.xml** widzimy jego definicję:

|  |
| --- |
| <**androidx.coordinatorlayout.widget.****CoordinatorLayout xmlns:android="http://schemas.android.com/apk/res/android"  xmlns:app="http://schemas.android.com/apk/res-auto"  xmlns:tools="http://schemas.android.com/tools"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  tools:context=".MainActivity"**>   <**com.google.android.material.appbar.AppBarLayout  android:layout\_width="match\_parent"  android:layout\_height="wrap\_content"  android:theme="@style/AppTheme.AppBarOverlay"**>   <**androidx.appcompat.widget.Toolbar  android:id="@+id/toolbar"  android:layout\_width="match\_parent"  android:layout\_height="?attr/actionBarSize"  android:background="?attr/colorPrimary"  app:popupTheme="@style/AppTheme.PopupOverlay"** />   </**com.google.android.material.appbar.AppBarLayout**>   <**include layout="@layout/content\_main"** />   <**com.google.android.material.floatingactionbutton.FloatingActionButton  android:id="@+id/fab"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:layout\_gravity="bottom|end"  android:layout\_margin="@dimen/fab\_margin"  app:srcCompat="@android:drawable/ic\_dialog\_email"** />  </**androidx.coordinatorlayout.widget.CoordinatorLayout**> |

Składa się on z kontenera **CoordinatorLayout** wewnątrz którego znajdują się trzy elementy:

* AppBar (pasek narzędzi): **AppBarLayout**
* Właściwy layout: <**include layout="@layout/content\_main"** />
* Przycisk na dole ekranu tzw. FAB: **FloatingActionButton**

Taki szablon tak naprawdę nigdy nie ulega zmianie – „mięso” widoku zamieszczone zostało w osobnym pliku: **content\_main.xml** wstrzykniętem za pomocą <**include layout="@layout/content\_main"** />.

Gdy popatrzymy w środek tego pliku zobaczymy:

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"***?>* <**androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"  xmlns:app="http://schemas.android.com/apk/res-auto"  xmlns:tools="http://schemas.android.com/tools"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  app:layout\_behavior="@string/appbar\_scrolling\_view\_behavior"  tools:context=".MainActivity"  tools:showIn="@layout/activity\_main"**>   <**TextView  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="Hello World!"  app:layout\_constraintBottom\_toBottomOf="parent"  app:layout\_constraintLeft\_toLeftOf="parent"  app:layout\_constraintRight\_toRightOf="parent"  app:layout\_constraintTop\_toTopOf="parent"** />  </**androidx.constraintlayout.widget.ConstraintLayout**> |

Właściwy layout składa się tutaj z klasy kontenera **ConstraintLayout** oraz z widgeta wyświetlającego tekst: **TextView.**

*Ogólnie rzecz biorąc zrobili w tym szablonie przerost formy nad treścią – nie przejmować się jak te pliki xml-owe wyglądają. One stają się po pewnym czasie bardziej czytelne. Tutaj prezentują w jaki sposób można sobie dzielić po prostu widoki.*

Android Studio renderuje ten widok jak:



Ostatnią rzeczą jaka trzeba obejrzeć jest plik AndroidManifest.xml

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"***?>* <**manifest xmlns:android="http://schemas.android.com/apk/res/android"  package="com.example.myapplication2"**>   <**application  android:allowBackup="true"  android:icon="@mipmap/ic\_launcher"  android:label="@string/app\_name"  android:roundIcon="@mipmap/ic\_launcher\_round"  android:supportsRtl="true"  android:theme="@style/AppTheme"**>  <**activity  android:name=".MainActivity"  android:label="@string/app\_name"  android:theme="@style/AppTheme.NoActionBar"**>  <**intent-filter**>  <**action android:name="android.intent.action.MAIN"** />  <**category android:name="android.intent.category.LAUNCHER"** />  </**intent-filter**>  </**activity**>  </**application**>  </**manifest**> |

W skład którego wchodzi element **application** którego wartość atrybut **label** to nazwa wyświetana na pulpicie, **icon** to ikona na pulpicie. Wewnątrz węzła application znajduje się jedyna nasza aktywność: MainActivity która została oznaczona intencją : **android.intent.category.LAUNCHER** dzięki której po naciśnięciu ikony na pulpicie uruchamia się aktywność MainActivity.kt.

## Napsujmy coś

### Zamiana napisu HelloWorld – statyczne podejście

Jeśli chcielibyśmy zmienić napis w **content\_main.xml** na coś innego niż „HelloWorld” to możemy wpisać docelowy tekst prosto w atrybut **text** widgetu **TextView**.

**android:text="Lubie placki!"**

### Zamiana napisu HelloWorld – statycznie, ale z zasobami

Jeśli chcielibyśmy zmienić napis w **content\_main.xml** na coś innego niż „HelloWorld” to możemy wpisać docelowy tekst w pliku **strings.xml** w zasobach. Tu dodam tekst „FishyFish” identyfikowany kluczem „nothing\_fishy”.

|  |
| --- |
| <**resources**>  <**string name="app\_name"**>My Application2</**string**>  <**string name="action\_settings"**>Settings</**string**>  <**string name="nothing\_fishy"**>FishyFish</**string**> </**resources**> |

Teraz w pliku widoku mogę **content\_main.xml** odwołać się do zasobu – w XML’ach odwołuje się przez notację @[typ zasobu]/[klucz].

**android:text="@string/nothing\_fishy"**

### ! Zamiana napisu HelloWorld – Dynamicznie w kodzie

Każdemu widokowi można przyporządkować identyfikator umożliwiający odwołanie się do elementu w XML, albo w kodzie. Na przykład modyfikując węzeł TextView w **content\_main.xml** w taki sposób (dodano atrybut id):

|  |
| --- |
| <**TextView  android:id="@+id/view\_my\_first\_view"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="@string/nothing\_fishy"  app:layout\_constraintBottom\_toBottomOf="parent"  app:layout\_constraintLeft\_toLeftOf="parent"  app:layout\_constraintRight\_toRightOf="parent"  app:layout\_constraintTop\_toTopOf="parent"** /> |

Mogę odwołać się teraz w kodzie za pomocą nadanej nazwy **view\_my\_first\_view** do elementu klasy TextView.

|  |
| --- |
| **override fun** onCreate(savedInstanceState: Bundle?) {  **super**.onCreate(savedInstanceState)  setContentView(R.layout.*activity\_main*)  view\_my\_first\_view.*text* = Date().toString() } |

Ważne jest to, że do widoków możemy odwołać się tylko wtedy gdy aktywność wywoła **setContentView().** Teraz po wywołaniu onCreate() nastąpi modyfikacja tekstu i zostanie tam wpisana data.

### ! Otworzenie Nowej Aktywności

By otworzyć kolejny ekran naszej aplikacji musimy:

* Utworzyć klasę aktywności
* Utworzyć layout (albo reużyć stary)
* Zarejestrować aktywność w manifeście
* Ustalić w kodzie jak odpalić aktywność

Zacznijmy od utworzenia prostego layoutu:

|  |
| --- |
| res/layout/activity\_secondary.xml |
| *<?***xml version="1.0" encoding="utf-8"***?>* <**FrameLayout  xmlns:android="http://schemas.android.com/apk/res/android"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"**>  <**TextView  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="New Activity Has Launched"**/> </**FrameLayout**> |

Teraz tworzymy klasę która w setContentView() odwoła się do tego layoutu:

|  |
| --- |
| **class** SecondaryActivity : Activity() {   **override fun** onCreate(savedInstanceState: Bundle?) {  **super**.onCreate(savedInstanceState)  setContentView(R.layout.*activity\_secondary*)  }  } |

W AndroidManifest.xml rejestrujemy aktywność:   
<**activity android:name=".SecondaryActivity"**></**activity**>

I na koniec modyfikujemy klasę MainActivity – ma ona już Floating Button o id: fab. Wykorzystajmy to:

|  |
| --- |
| **class** MainActivity : AppCompatActivity() {   **override fun** onCreate(savedInstanceState: Bundle?) {  **super**.onCreate(savedInstanceState)  setContentView(R.layout.*activity\_main*)  fab.setOnClickListener **{** startActivity(Intent(**this**, SecondaryActivity::**class**.*java*))  **}** }  } |

Klasa Button ma metodę setOnClickListener() przyjmującą jako argument funkcję wywoływaną przy aktywacji widoku (przy naciśnięciu). W przypadku gdy jednym argumentem funkcji jest inna funkcja – Kotlin dopuszcza skróconą notację: za pomocą nawiasów klamrowych { } (wyrażenie lambda).

Odpalenie aktywności następuje na wskutek wywołania metody: startActivity() która przyjmuje dwa argumenty: kontekst (który komponent systemu prosi o wywołanie aktywności) oraz docelową klasę aktywności. Kontekstem może być zarówno obiekt Aktywności jak i Aplikacji (czy Fragmentu) – tu jest to aktywność reprezentowana przez wywołującego czyli **this**.

## Aplikacja z punktu widzenia architekta – sprostowanie

Ikona na pulpicie Androida reprezentuje **aktywność** w ramach **aplikacji** oznaczoną **intencją**: **”android.intent.category.LAUNCHER”. Aplikacja** dzielona jest na zwane **aktywnościami**, które podlegają obowiązku rejestracji w pliku AndroidManifest.xml. **Fragmenty** reprezentują części ekranu odpowiadające stałym obszarom funkcjonalnym aplikacji – jedna **aktywność** może wyświetlać wiele **fragmentów**. Na **aplikację** mogą składać się też trzy inne komponenty podlegające obowiązkowi rejestracji: **Broadcast Receivery**, **serwisy** (oba komponenty realizują działania w tle) oraz **Content Resolver** – egzotyczny komponent do wymiany danych pomiędzy aplikacjami systemu.

Z punktu widzenia programisty:

* Aplikacja reprezentowana jest przez klasę Application ([link](https://developer.android.com/reference/android/app/Application?hl=en))
* **Aktywność reprezentowana jest przez klasę Activity (**[**link**](https://developer.android.com/reference/android/app/Activity)**)**
* Serwis reprezentowany jest przez klasę Service ([link](https://developer.android.com/reference/kotlin/android/app/Service?hl=en))
* Broadcast reprezentowany jest przez klasę BroadcastReceiver ([link](https://developer.android.com/reference/android/content/BroadcastReceiver))
* Fragmenty ([link](https://developer.android.com/guide/components/fragments))

# 4. Kotlin

Strona zawierająca referencyjne materiały jest tu: [link](https://kotlinlang.org/docs/reference/)

# 5. Android

# 6. Zaliczenie